MANAGEMENT OF THE OBJECT-ORIENTED DEVELOPMENT PROCESS

Liping Liu and Borislav Roussev

Management of the Object-Oriented Development Process

Liping Liu
The University of Akron, USA

Boris Roussev
University of the Virgin Islands, USA

IDEA GROUP PUBLISHING
Hershey • London • Melbourne • Singapore

Acquisitions Editor:	Renée Davies
Development Editor:	Kristin Roth
Senior Managing Editor:	Amanda Appicello
Managing Editor:	Jennifer Neidig
Copy Editor:	Maria Boyer
Typesetter:	Larissa Zearfoss
Cover Design:	Lisa Tosheff
Printed at:	Integrated Book Technology

Published in the United States of America by
Idea Group Publishing (an imprint of Idea Group Inc.)
701 E. Chocolate Avenue, Suite 200
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@idea-group.com
Web site: http://www.idea-group.com

and in the United Kingdom by
Idea Group Publishing (an imprint of Idea Group Inc.)
3 Henrietta Street
Covent Garden
London WC2E 8LU
Tel: 44 20 7240 0856
Fax: 44 20 7379 3313
Web site: http://www.eurospan.co.uk

Library of Congress Cataloging-in-Publication Data

Management of the object-oriented development process / Liping Liu and Borislav Roussev, editors.
p. cm.
Summary: "This book consists of a series of high-level discussions on technical and managerial issues related to object-oriented development"--Provided by publisher.
Includes bibliographical references and index.
ISBN 1-59140-604-8 (hard cover) -- ISBN 1-59140-605-6 (softcover) -- ISBN 1-59140-606-4 (ebook)
1. Object-oriented programming (Computer science) 2. Computer software--Development. I. Liu, Liping. II. Roussev, Borislav.
QA76.64.M348 2005
005.1'17--dc22
2005004533

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

All work contributed to this book is new, previously-unpublished material. Each chapter is assigned to at least 2-3 expert reviewers and is subject to a blind, peer review by these reviewers. The views expressed in this book are those of the authors, but not necessarily of the publisher.

Management of the Object-Oriented Development Process

Table of Contents

Preface .... **vi**

Chapter I
Object-Oriented Modeling in UML2 .... **1**
Boris Roussev, University of the Virgin Islands, USA

Chapter II
MDA with xUML: Model Construction and Process Management **36**
Boris Roussev, University of the Virgin Islands, USA

Chapter III
Management Planning in a Changing Development Environment ... **61**
Melissa L. Russ, Luminary Software and Space Telescope Science Institute, USA
John D. McGregor, Luminary Software and Clemson University, USA

Chapter IV
MDA Design Space and Project Planning **87**
Boris Roussev, University of the Virgin Islands, USA

Chapter V
Agile Outsourcing to India: Structure and Management **109**
Boris Roussev, University of the Virgin Islands, USA
Ram Akella, University of California, USA

Chapter VI
User Requirements Validation and Architecture Discovery through Use Case Invariants and Model Animation **141**
Boris Roussev, University of the Virgin Islands, USA
Yvonna Rousseva, University of the Virgin Islands, USA

Chapter VII
RUP and eXtreme Programming: Complementing Processes **183**
Gary Pollice, Worcester Institute of Technology, USA

Chapter VIII
Managing Complexity with MDSD **200**
Jorn Bettin, SoftMetaWare, USA

Chapter IX
Agile RUP: Taming the Rational Unified Process™ **231**
Gary K. Evans, Evanetics, Inc., USA

Chapter X
Planning and Managing the Human Factors for the Adoption and Diffusion of Object-Oriented Software Development Processes **247**
Magdy K. Serour, University of Technology, Sydney, Australia

Chapter XI
Web Services in Service-Oriented Architectures **278**
Gerald N. Miller, Microsoft Corporation, USA

Chapter XII
Model-Based Development: Metamodeling, Transformation and Verification **289**
Juan de Lara, Universidad Autónoma de Madrid, Spain
Esther Guerra, Universidad Carlos III, Spain
Hans Vangheluwe, McGill University, Canada

Chapter XIII
Agile Project Controlling and Effort Estimation **313**
Stefan Roock, it-agile GmbH, Germany
Henning Wolf, it-agile GmbH, Germany

Chapter XIV
Improving OO Design Process Using Rules, Patterns and Refactoring 325
Javier Garzás, University Rey Juan Carlos, Spain
Mario Piattini, University of Castilla-La Mancha, Spain

Chapter XV
The BORM Method: A Third Generation Object-Oriented Methodology 337
Roger Knott, Loughborough University, UK
Vojtech Merunka, University of Agriculture in Prague, Czech Republic
Jiri Polak, Deloitte & Touche, Prague, Czech Republic

About the Authors 361

Index 368

Preface

Introduction

Dilemmas involving notation, project planning, project management, and activity workflow pervade the world of software development. Object-orientation provides an elegant language for framing such problems, and powerful tools for resolving them.

In this book, we have brought together a collection of presentations, giving the reader an in-depth look into the technical, business, and social issues in managing object-oriented development processes, as well as presenting new technologies, making software development more effective. The chapters in the book examine many topics in the research frontier of software development, including methods, technologies, strategies, and the human factor. The book also presents the fundamentals of object-oriented project management.

The various backgrounds of the contributing authors—industrial, consulting, research, and teaching—yielded presentations, complementing and enriching each other. As a result, the book paints a holistic picture of the multi-faceted problems in object-oriented software development. It should be of interest to software developers, project managers, system analysts, and graduate and upper-level college students majoring in information systems and computer science who would like to deepen their knowledge in the field of object-oriented project management.

Very briefly, some of the major topics discussed in this book include: software development life cycle; development strategies, for example, open source, outsourcing, and product lines; componentization; the human factor; object-oriented notation and techniques, such as xUML, MDA, and MDSD; requirements engineering; design patterns; project management; and system integration with Web services.

Organization

The book is organized into 15 chapters. Each chapter emphasizes a particular area, identifies important shortcomings, discusses current activities, offers new insights into the problematics, and suggests opportunities for improving the management of object-oriented software development projects.

Motivated by computer simulation, the notions of object, class, and class generalization were formulated by Dahl and Nygaard in 1967. However, it was not until the mid-1990s that the first industrial-strength, object-oriented notations were complemented by sound development methods. Today, the object-oriented world is dominated by UML to the extent that UML and object-orientation have become synonymous. The book naturally begins with an introduction to UML2. The emphasis is on the novel features of UML and the new trends in object-orientation, namely, modeling of large things, a higher level of modeling abstraction, design automation, precision, and freedom from the constraints of the implementation platform.

In Chapter II, the themes from the introductory chapter are re-examined in the framework of xUML (executable UML) and MDA (model-driven architecture). MDA and xUML are among the latest initiatives of OMG. They promise to change the way software is created by combining a modeling language with a model manipulation language, rendering implementation programming obsolete. The chapter presents the two methodologies. It also discusses the MDA activity workflow and presents a development method for projects relying on xUML.

In Chapter III, Russ and McGregor present a model for planning object-oriented projects. The authors structure the software development landscape into a triad of high-level dimensions—technology, method, and organizational strategy—where each dimension is further divided into several sub-dimensions. The model defines project planning as navigating through a multi-dimensional hyperspace.

In Chapter IV, the Russ-McGregor model has been applied to evaluate the strength and weaknesses of xUML and MDA. The analysis sheds light on the economics of model-driven software development, and on the difficulties project managers and developers alike may encounter in adopting the two technologies in an industrial setting.

In Chapter V, Roussev and Akella present a new approach to managing outsourcing projects. Drawing on experience with Indian software firms, the authors closely analyze the problems faced by outsourcing clients and offshore developers. Roussev and Akella show how these problems can be successfully resolved by scaling down a large outsourcing project to meet the Agile "sweet spot," and by carefully managing the communication patterns among all stakeholders.

In Chapter VI, Roussev and Rousseva present a process extension applicable to both lightweight and heavyweight development methods. The extension is based on a business value invariant, and views the iterative and incremental model of software development as a communication model. The proposed techniques link the informal user requirements world to the system model, which makes it possible to derive mechanically the system architecture from the user requirements, and automatically to validate it with respect to the system's use case model through model animation.

It is a well-known fact that many of the agile practices are incompatible with the context of large-sized projects. Gary Pollice and Gary Evans, two nationally recognized methodologists, independently present their approaches to reproducing the conditions for agility in large-sized projects by balancing agility and discipline. Pollice and Evans look out for common grounds between Agile and RUP to get the best of both worlds.

In Chapter IX, Jorn Bettin, director of an international strategic technology management consultancy, addresses the question of how to create durable and scalable software architectures, so that the underlying design intent survives over a period of many years. Bettin goes beyond object-orientation and traditional iterative software development to define a set of guiding principles for component encapsulation and abstraction, and to form the foundation for a model-driven approach to software development.

In Chapter X, Magdy Serour from the Centre for Object Technology Applications and Research (COTAR) at the University of Technology, Sydney, delves into a gray area of object-orientation, namely, the effect of various human factors on the adoption and diffusion of an object-oriented software development process. Serour defines a process to assist organizations in planning and managing their transition to object-oriented development. The author discusses key "soft" factors, such as motivation, leadership, and overcoming the resistance to culture change, which are critical in promoting the process of organizational change.

In Chapter XI, Gerald Miller from Microsoft addresses a very important area of the new technological wave. Integration of systems in a cost-effective way is crucial for most enterprises, as many integration efforts fail to bring about the promised return on investment. Miller's presentation discusses how to resolve the system integration nightmare by building a service-oriented architecture with Web services which integrates disparate systems, both within organizations and across business partners' firewalls.

In Chapter XII, de Lara, Guerra, and Vangheluwe give an overview of model-based software development, and propose ideas concerning meta-modeling and the use of visual languages for the specification of model transformations, model simulation, analysis, and code generation. They also examine the impact of model-based techniques on the development process.

The Agile methods are based on the presumption that a complete and stable requirements specification is generally impossible. This assumption invalidates the very vehicle for computing project velocity, progress, deadline prognosis, and budget allocations, as project managers cannot track the number of closed vs. open requirements. In Chapter XIII, Roock and Wolf demonstrate a practical technique, integrating lightweight mechanisms for project controlling into Agile methods. They propose to combining an (incomplete) hierarchical decomposition of a system with abstract measurements. Their approach addresses pressing management needs without incurring the burden of a waterfall-like exhaustive specification upfront.

Object-oriented knowledge comes in different forms, for example, principles, heuristics, patterns, refactoring, lessons learned, defects, and best practices. In Chapter XIV, Garzás and Piattini define an ontology of object-oriented micro-architectural design knowledge to systematize this knowledge so that it can be easily comprehended by developers and used in practical cases.

In the final chapter, Knott, Merunka, and Polak propose a new object-oriented methodology, which makes extensive use of business process modeling. The authors contrast and compare their approach to similar development approaches, and provide a case study to demonstrate the feasibility of their methodology.

Acknowledgments

We would like to thank Mehdi Khosrow-Pour, Idea Group Inc. Senior Editor, for inviting and accepting our book proposal and for his insistence on timely completion of this project. Thank you also to Jan Travers, Idea Group Inc. Managing Director, for her patience that made it possible to bring this project to fruition. We would also like to thank all referees for their assistance in reviewing and improving on some of the chapter materials.

The second editor would also like to thank Professor Aubrey Washington and Chancellor Jennifer Jackson at the University of the Virgin Islands for their continued support and encouragement. He is indebted to his colleague, Professor John Munro, for his professional advice, valuable suggestions, and feedback. He is grateful to Professor Akella at the University of California, Santa Cruz, for providing me with the opportunity to work and do research in the Silicon Valley.

Above all, he expresses his deep gratitude to Professor Yvonna Rousseva, his spouse, for her insightful comments, incisive critique, and numerous thought-provoking discussions. Without her, this book would have been impossible to finish.

Chapter I

Object-Oriented Modeling in UML2

Boris Roussev
University of the Virgin Islands, USA

Abstract

Object-orientation (OO) is a powerful design methodology, which has firmly moved into the mainstream of software development. In 2002, both the IEEE John von Neumann Medal and the ACM Turing Award (the Nobel Prize for Computing) were awarded to the scholars who started the object-oriented journey back in 1967. Despite this recognition, object-orientation is far from being the dominant technology of choice. Contrary to the common belief, a recent authoritative study reports that only 30% of the software companies rely on OO technologies, and that the waterfall model is still the most popular lifecycle model of software development. In this introductory chapter, we present the fundamental concepts and principles of object-oriented modeling with UML version 2. Born out of the efforts to resolve the software crisis, UML has taken such a hegemonic role that we fear object-orientation may undergo a population "bottleneck." In biology, this is an event that dangerously cuts the amount of genetic diversity in a population. The objectives of this chapter are as follows: 1) to present the influential ideas in the evolution of object-orientation; 2) to identify the lasting trends in object-orientation; 3) to

introduce the core UML modeling languages and some of the techniques closely associated with them; and 4) to discuss some of the challenges with object-oriented modeling. In addition, we present the E-ZPass system, a system of moderate complexity used as a running example in the first five chapters of this book. This presentation is the book's cornerstone. There is not a single chapter in the rest of this volume that does not assume an overdetermined <<UML>> reader.

Introduction

Building large and complex software systems is notoriously difficult. To build such systems, we need methodologies backed by languages that would feature characteristics addressing the following needs:

	Features	Needs
❶	To structure a system into modular components that can be developed, maintained, and reused separately.	Control complexity.
❷	To base the semantic structure and behavior of the solution on the problem being solved.	Reduce the cognitive burden.
❸	To raise the level of abstraction of the artifacts being constructed.	Curb complexity.
❹	To use a common vocabulary with the client. To describe the problem in a notation that is client and designer friendly.	Link business and technology, and improve communication.
❺	To describe a problem precisely and in a way that avoids delving into technical details.	Testability, executability, portability, productivity, and design automation.
❻	To allow for reuse at all levels: requirements, analysis, design, architecture, and domain, and at all levels of interest: structural, behavioral, and communicational.	Improve quality and productivity.
❼	To provide the basis for effective management of the development process.	Develop cost-effectively.
❽	To automate repetitive design and implementation tasks.	Improve quality and productivity.
❾	To facilitate iterative and incremental, architecture-centered, test-driven processes.	Risk mitigating, exploratory processes.
❿	To respond to change quickly and in a cost-effective manner.	Satisfy the ever-evolving user needs.

Object-orientation (OO) is a powerful design methodology based on the principles listed above. The importance of OO has now been firmly recognized by industrial and research communities alike. In 2002, both the IEEE John von Neumann Medal and the ACM Turing Award (the Nobel Prize for Computing) were awarded to Ole-Johan Dahl and Kristen Nygaard for their pioneering work in OO through the design and implementation of SIMULA67. A quote from the ACM address confirms the importance of this seminal work:

> *Their [Dahl and Nygard's] work has led to a fundamental change in how software systems are designed and programmed, resulting in reusable, reliable, scalable applications that have streamlined the process of writing software….*

In modern history of science, it is not very common to have such a low adoption rate as in the case of OO. The work of Dahl and Nygaard was made almost extinct by two programming languages called COBOL and C, which muscled their way through, backed by several major industry players and governmental agencies.

Despite the recognition, OO (with the exception of use cases) is not yet the dominant technology of choice. In a comprehensive study, Laplante and Neill (2004) report that only 30% of the software companies rely on OO technologies and that the waterfall model is still the most popular lifecycle model in use. The authors conclude that OO techniques are not dominant, which is in sharp contrast with the common belief about the popularity of OO methods and technology.

In this introductory chapter, we lay the fundamental concepts of OO using the Unified Modeling Language (UML, 2004). Since it was adopted by the OMG in 1997, UML has been widely accepted throughout the software-modeling world and successfully applied to diverse domains. Therefore, it will not be much of an exaggeration to paraphrase from Wittgenstein[1]: namely, the limits of UML mean the limits of my OO world.

Since OO modeling is dominated by UML in all areas, it seems unlikely for an OO notation or method to be able to survive away from UML. In the foreseeable future, the evolutionary history of OO will be tied in very closely to that of UML. As a result, OO methods may undergo a population "bottleneck." In biology, this is an event that cuts dangerously the amount of genetic diversity in a population. On the bright side, OMG is backed by a large industrial consortium, and important decisions are taken with broad consensus and (in truly OO spirit) after an incremental and iterative process. Despite the dominant role of UML, there are alternative approaches, for example, the OPEN Modeling Language (Firesmith, Henderson-Sellers, & Graham, 1998) and the Business Object Notation (Paige & Ostroff, 1999).

The rest of the chapter is structured as follows. The next section presents a brief history of OO focusing on influential ideas and contribution. Then we introduce the core set of modeling language of UML2. We use as a running example a real-world system of moderate complexity. We next discuss some of the challenges with OO modeling, before concluding the chapter.

Brief History of Object-Orientation

The crisis in software development is not from yesterday. Being relatively young, software development has lacked until recently a proven apparatus for design and evaluation of software products. The historic conference, sponsored by NATO, that diagnosed the crisis of software development and proposed "software engineering" as a remedy, took place in Garmisch, Germany, in the Bavarian Alps, in 1968. After the conference, Dijkstra, one of the participants, wrote: "For me it was the end of the Middle Ages. It was very sunny." However, almost 40 years later, resorting to best practices, documented as development methods, such as RUP and Agile, testifies to the fact that it might have marked not so much the end of the Middle Ages, but rather it might have foreshadowed the harbinger of a Renaissance.

In this section, we attempt to summarize chronologically the OO Renaissance. The review of the early OO analysis and design methods is based on the insightful discussion in Graham and Wills (2001).

The rapidly improving performance/price ratio of hardware entailed wide adoption of client-server multi-tier information systems in every sphere of human activity. In the 1980s, the software development community came under increased pressure to develop usable, dependable, and scalable distributed systems in a cost-effective way.

To increase productivity and quality, the software industry had to adopt methods of building systems out of reusable components, and at the same time to raise the level of abstraction of the designed artifacts. There were two major challenges to materializing these ideas. First, with structured methods, less than half of a system could be built out of reusable components (Biggerstaff & Richter, 1989). And second, raising the level of abstraction leaves developers with less meaningful artifacts—that is, less semantic richness and fewer venues for extensibility.

Structured methods view a system as a hierarchy of functions and rigidly separate process knowledge from data. Self-contained, customizable, and reusable components did not naturally fit in this environment.

The software industry responded to these challenges by shifting its focus to OO methods. Up until the early 1980s, OO was largely associated with the development of graphical user interfaces and AI. In the 1980s, interest extended to OO design, mainly to support the OO programming language Smalltalk and the OO extension of the imperative language C, named C++.

In the late 1980s and early 1990s, interest moved away from OO design to OO analysis. Several influential works helped gradually shape the field of OO analysis. The coinage *object-oriented design* is attributed to Booch, who published a paper in 1988 and a book (Booch, 1991, 1993) with the same name.

The switch from well-established structured analysis techniques to OO analysis methods naturally began with extending the entity-relationship model (Codd et al., 1980) and endowing entities with behavioral aspects, the so-called data-driven approach. Data-driven methodologies (Coad & Yourdon, 1991; Rumbaugh et al., 1991; Shlaer & Mellor, 1988) approached application development as relational database development.

Coad and Yourdon (1991) added operations to entities to define class models as a basis of a simple OO analysis method, which immediately became popular. OMT (Rumbaugh et al, 1991), built on the work of Coad and Yourdon, and Shlaer and Mellor (1988) proposed the use of finite state machines (FSMs) to describe the lifecycles of class instances. This idea came to fruition in the work of Shlaer and Mellor (1992), who also advanced the notion of translational modeling, thus laying the foundation for Executable UML (Mellor & Balcer, 2002) and Model-Driven Architecture (MDA, 2004).

A second group of OO methods emerged in parallel with the data-driven group, called responsibility-driven methods. They shied away from the entity-relationship model and more strictly adhered to OO principles.

Wirfs-Brock et al. (1990) developed a set of responsibility-driven design (RDD) techniques. Among the most important contributions of RDD are the extension of the idea of CRC cards (Beck & Cunningham, 1989) and the introduction of stereotypes. CRC cards are physical cardboard pieces visualizing classes with their responsibility and collaborations (hence the name CRC). Developers are encouraged to take the view of an instance of a class (called anthropomorphism) and act out the object's lifecycle in order to discover new classes in the problem domain. Then, these CRC cards are used as a starting point for design. This technique became one of the underpinnings of the Agile methodologies (Beck, 1999).

Cook and Daniels' (1994) Syntropy added rigor to OO modeling by introducing a pure expression language, later called OCL to express modeling concepts, such as uniqueness constraints, invariants, transition guards, and operation pre- and post-condition, that cannot even be represented on class diagrams and finite state machines.

As OO notations mushroomed in the early 1990s, so did the OO methods that lived on them. MOSES (Henderson-Sellers & Edwards, 1994) was the first OO method to include a full-fledged development process and metrics. SOMA (Graham, 1994), following in the footsteps of MOSES, attempted to fuse the best practices of all methods, and in addition emphasized requirements engineering, process, and rigor.

Arguably the most influential work, which later became the foundation of RUP (Kruchten, 2000), was the OO incarnation of Objectory (Jacobson, 1992). Objectory started out as a proprietary method in Erickson, a Swedish telecommunication company, in the late 1960s. The major contributions of Jacobson's work are the following: use cases (clusters of scenarios of system usage); use case analysis; the use of sequence diagrams for class discovery and distribution of responsibilities among classes; the boundary-controller-entity design pattern, which allowed the containment of "changeability" through "locality of change;" and the definition of software development as model transformation from more abstract to more detailed and precise models.

In 1997, OMG standardized and published UML version 1. UML1 quickly became the norm for OO modeling. UML was influenced by OMT, OCL, the Booch notation, activity diagrams for process modeling (Martin & Odell, 1992), stereotypes (Wirfs-Brock et al., 1990), and multiple interfaces from Microsoft's COM+ technology. UML's belated conception was the major cause for its inconsistencies and deficiencies (Miller, 2002). The industry had already invested in the creation of expensive CASE tools, and CASE tool vendors resisted many innovations and simplifications that would render their tools obsolete.

The two most popular methods used for OO software development with UML are RUP, developed at Rational by Jacobson et al. (Kruchten, 2000), and the lighter-weight Agile methods (Beck, 1999). Both RUP and Agile are based on an iterative and incremental lifecycle, in which the system architecture evolves (hence the term *evolutionary*) incrementally from a prioritized set of user requirements in a series of mini-waterfall cycles of business-modeling/requirements-elicitation/analysis/design/implementation/test. The evolutionary approaches are requirements-driven (each artifact can be traced to a requirement) and architecture-centered (or architecture first). They outperform in terms of customer satisfaction and cost methods based on the waterfall model in fast-paced, dynamic environments, with a great risk exposure due to the multiple feedback opportunities for customers to situate and orient the development effort. The evolutionary approaches are based on a reciprocal relationship between system and user. On the one hand, the user envisions the system. On the other hand, the system changes the user's perception about the system. Therefore, we could expect that a partially completed prototype (increment) will be able to reveal early on the anticipated change in the users, and in this way to reduce the backtracking in the development process.

The inconsistencies in UML1 prompted several minor revisions (UML1.x), followed by a major one, whose result will be UML2. The standardization process is scheduled for completion by the end of 2004. UML2 promises to be a breakthrough in model-driven development by making models executable and by automating the relationship between models and code.

Objects turned out to be disproportionately small compared to the size of the systems being built with them. The second wave of innovations in OO analysis was marked by an increased interest in things bigger than objects, namely things allowing reuse on a larger scale, but foremost, things automating the software design process. The trend, which began in the mid-1990s, is toward more complex structures at a higher level of abstraction, gradating from design patters and components, through aspects, to domains. The second trend informing OO today is design automation.

Introduced in a seminal work (Gamma, Helm, Johnson, & Vlissides, 1995), design patterns have quickly gained popularity among developers. A design pattern is a description of a problem, a best practice for its solution, and when and how to apply the best practice (typically, via instantiation) in new situations. Gamma et al. showed the advantage of reusing aggregations of objects along with the mechanisms for their interactions. The reusability of design knowledge with design patterns, however, comes at a certain price. Garzás and Piattini (2005) list a number of problems with design patterns: difficult application, temptation to recast everything as a pattern, pattern overload, high-dependence on a programming language (except for analysis patterns), complex non-homogeneous catalogs, and a steep learning curve for some counter-intuitive patterns, to name a few. It is important to note that design patterns do not raise the level of abstraction at which design is carried out. They describe solutions at the object level.

OO provided the foundation for component-based development. Components are big things that do raise the level of abstraction in design. A component is a set of closely related objects, encapsulated together to support a set of clearly defined interfaces. With components, a system is described in terms of interactions among component interfaces. Designed with reusability in mind, components lead to increased productivity and quality. To use components, developers ought to be equipped with tools for component customization, initialization, and interface inspection.

To address the issue of crosscutting concerns, Kiczales (1996) introduced aspect-oriented programming (AOP). AOP deals with abstractions on a plane different from object-oriented modeling. AOP advocates specifying separately the various pervasive requirements of a system, for instance, security and transaction management, which do not decompose into behavior centered on a single locus, and then, to weave them together into a coherent system. AOP focuses on the system's infrastructure, something that cannot be done effectively only in terms of components and objects. AOP's major drawback is that aspects are defined at code level.

The largest truly reusable constructs in OO are domains. An executable domain model captures precisely the conceptual entities of a single subject matter (Mellor & Balcer, 2002). Each domain forms a cohesive whole, semantically autonomous from other domains, which makes domains the largest units of reuse. A domain is modeled as a set of communicating objects or more precisely as a set of communicating object state machines. Executable domain models can be simulated and debugged, woven together through bridges, and automatically compiled to code based on the principles of generative programming (Czarnecki & Eisenecker, 2000). Domains bear certain similarity to aspects, but they are, in distinction, defined at a very abstract model level.

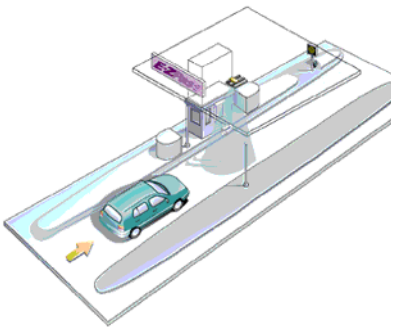
In line with model-driven development and domain modeling, OMG launched the Model-Driven Architecture initiative (MDA, 2004). The essence of the MDA approach is the clear separation between executable platform independent models (PIM) and platform specific models (PSM), and the application of marks and mappings to transform PIMs to PSMs, and to generate code from PSMs. The notions of model executability and model mappings bring in the so-much-sought-after design automation. MDA raises the level of abstraction and avoids technology dependence, thus addressing the complexity and portability problems. Furthermore, MDA applies very successfully the separation-of-concern principle by capturing design knowledge in machine-readable mappings and application knowledge in executable domain models. Model executability affords model testing and debugging, thus realizing the "proving with code" principle, only at a much more abstract level.

The level of reuse culminates in product lines (Withey, 1996; PL, 2004), breaking easily the 90% reuse barrier. The product lines method aims at identifying units of common functionality in a family of products and defining a customized development process, minimizing the effort necessary to duplicate this functionality in the related products. Several methods (Bosch, 2000; Bayer et al., 1999; Atkinson, Bayer, & Muthig, 2000) attempt to combine component-based development, design patterns, and product lines, and to provide developers with the necessary automation tools.

Design automation reaches its highest in Model-Driven Software Development (MDSD) (Bettin, 2004, 2005). MDSD is a synergy of product line, DSLs[2], and MDA. MDSD is characterized by the conscious distinction between building software factories and building software applications (also referred to as domain engineering). MDSD supports domain-specific specialization and mass customization.

With less emphasis on programming, requirements engineering and systems analysis will grow more and more important as time goes by. Research efforts in the requirements engineering area are toward understanding the role of requirements in the wider systems engineering process and toward validating user requirements in a manner similar to validating software—that is, requirements should be automatically translated to code and executed via simulation techniques (Lamsweerde, 2000).

Figure 1. E-ZPass system



Modeling with UML

In this section, we introduce the core set of the UML modeling languages needed to design OO systems. The UML models we discuss are class diagrams modeling the information structure of a system, FSMs representing the behavior of individual objects, use cases expressing user requirements, and interaction diagrams (sequence and communication diagrams) modeling interactions among societies of objects, collaborating to realize user requirements.

Case Study

In this and the next four chapters, we will use the following system as a running example.

In the road traffic E-ZPass system, drivers of authorized vehicles are charged at tollgates automatically. They pass through special lanes called E-Z lanes. To use the system, a driver has to register and install an electronic tag (a gizmo) in his/her vehicle. The vehicle registration includes the owner's personal data, credit card or bank account, and vehicle details. As a registered vehicle passes through the tollgate, an antenna electronically reads account information on the gizmo, and the toll is automatically deducted from the prepaid account. The amount to be debited depends on the kind of the vehicle. When an authorized vehicle passes through an E-Z lane, a green light comes on, and the amount being debited is displayed. If an un-authorized vehicle passes through an E-Z lane, a yellow light comes on and a road camera takes a photo of the plate, used to fine the vehicle's owner (fine processing is outside the system scope). There are E-Z lanes where the same type of vehicles pay a fixed amount, for example at a toll bridge, and there are E-Z lanes where the amount depends on the type of vehicle and the distance traveled, for example on a highway. For the latter, the system stores the entrance tollgate and the exit tollgate.

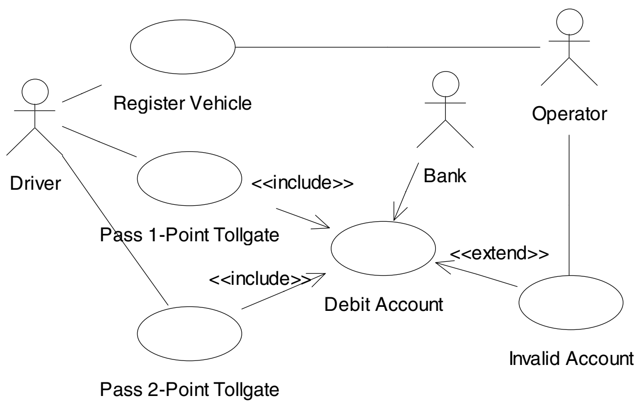
User Requirements Model

A user requirement is a dialogical specification of what the system must do. As user requirements are elicited, they are organized into use cases. Requirements are not yet first-order elements in UML, but they are scheduled to become such in the forthcoming Systems Engineering UML profile. The systems engineering profile defines about 10 different types of requirements, which fall into three broad categories: operational, functional, and quality of service (QoS). Operational and functional requirements are best expressed as usage scenarios modeled with sequence diagrams or communication diagrams (formerly collaboration diagrams). QoS requirements come in several types (e.g., usability, security, and performance, to mention a few) and are modeled as constraints.

Since their introduction in the late 1980s (Jacobson, 1987), use cases have proven to be an immensely popular software development tool. Use cases organize requirements—such as tens of usage scenarios—around a common operational capability. They describe the system as a black box. Use cases present a conceptual model of the intended system behavior by specifying services or capabilities that the system provides to its users.

In use case models, the system's environment, or context, is modeled as actors, which are essentially roles played by end users or external systems (a single user may perform many roles). In Figure 2, the actors for the E-ZPass system are Driver, Bank, and Operator. They are rendered as stick figures. The actors communicate with the use cases rendered as ovals via communication links. A use case contains and organizes multiple scenarios. A scenario is a linear sequence of transactions performed by actors in a dialogue with the system that brings value to the actors.

Figure 2. E-ZPass use case model



A system is conceptually decomposed into subsystems of related use cases linked by <<include>> and <<extend>> dependency relationships or sharing common actors. <<>> indicates a stereotype. Stereotypes are a lightweight UML extension mechanism. A stereotype defines the meaning of a new building block, which has derived from an existing one. The <<include>> relationship is used to extract out a coherent part of a use case, typically for the purpose of reuse. It can also be used to decompose a system-level use case into "part" use cases to be realized by different subsystems. The <<extend>> relationship is used when the behavior of one use case, called base use case, may be extended with the behavior of another use case under certain conditions. Users find the direction of the <<extend>> relationship counter-intuitive (from the extension use case to the base use case). It is best to think of it as the extension use case pushing functionality to the base use case. Figure 2 illustrates the discussed concepts related to use cases.

A use case diagram shows only the use case's organization. The flows of events in every use case are described in text. The following is a fairly standard template for use case descriptions.

Use case name

Description: brief narrative description.

Actors: a set of Actors

Basic flow of events: numbered sequence of events

Exceptional flow of events: numbered sequence of events

Pre-conditions: contract between actors and use case

Post-conditions: result of a use case execution

We show below the structure of PassOnePointTollgate use case.

Use Case: PassOnePointTollgate

Description: This use case describes the system's behavior in response to a vehicle passing through a single tollgate.

Actors: Driver

Basic Flow

1. The use case begins when a vehicle with a gizmo passes through a single tollgate. The tollgate sensor reads the gizmo's ID. The system records the passage, including date, time, location, and rate; displays the amount the driver will be charged; and turns the green light on.

Exceptional Flow of Events:

- The gizmo is invalid or missing. The system turns the yellow light on and a photo of the vehicle is taken.

Pre-Conditions: None

Post-Conditions: The vehicle's account is updated with the passage information and the driver's credit card.

Modeling the System's Information Structure

The concepts used for describing systems' information structures in UML are class, object, relationship, component, subsystem, and OCL constraints.

Objects, Classes, and Relationships

The real world is populated with objects (or instances) of different kinds: books, bank accounts, and tollgates, to mention a few. Each object has a number of characteristic attributes that identify it. For example, a book has a title, author, and publisher. A bank account has an account number, owner, and interest rate. Figure 3 shows the characteristic attributes of several kinds of objects. Such an object description is called *class*, because it describes a class (set) of similar objects.

Although the object's attribute values give us information about the object's identity, this information is somewhat static. The attributes alone fail to represent the dynamic nature of many objects. What we need to know are the operations in which the objects can be involved, or in other words, the behavior the objects can exhibit. For example, a bank account can accumulate interest rate, can be debited or credited.

Graphically, a class is represented as a rectangle with three compartments as shown in Figure 4. The top compartment contains the class name. The middle

Figure 3. Class attributes

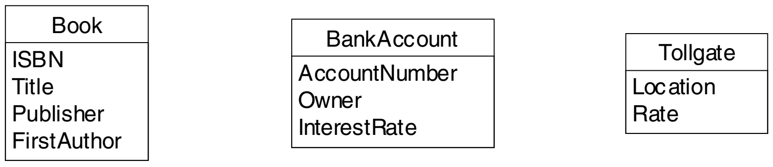
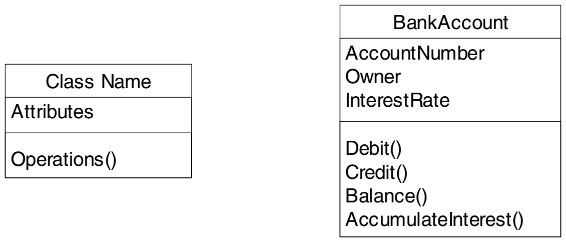


Figure 4. Class structure with characteristic attributes and operations



compartment contains the characteristic features, called attributes. The bottom compartment defines the operations in which the objects of the class can be involved. Depending on the modeler's goals, any comportment, except for the name, can be omitted.

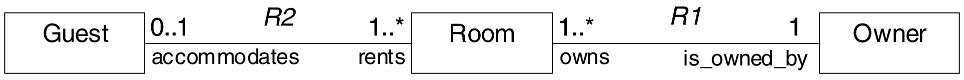
Relationships

The attributes and operations alone are not sufficient to understand the "essence" of an object. An object in isolation does not have a complete "self-identity." The human mind distinguishes between objects by difference. This difference is determined by the relationships into which objects can enter, by the ways the objects can be used. The relationships into which an object can enter are determined by the object's responsibilities to other objects (implemented as publicly accessible operations). Similar to how words mean in a language, we can say that an object acquires "self-identity" through different relationships with other objects. Objects are never strictly defined and identified unless they enter into relationships.

Jacques Derrida, the French linguist-philosopher, has famously argued that everything has an originary lack. By entering into relationships with other objects, the thing compensates for this originary lack. Let us consider a bed and breakfast room. The room alone is a lump of bricks and furniture, inconsequential for understanding its purpose, which can be articulated and manifested only through the relationships into which the room participates. The relationship room-owner and the relationship room-guest are the room's "essential" characteristics. For the owner, the room is a source of income, while for the guest, the room is a place to rest. The former relationship satisfies the owner's lack or desire (for a source of income), while the latter satisfies the guest's lack (a home away from home). We can say that the room's relationships came into being to fill in the owner's and the guest's voids.[3]

In OO, relationships are formalized as associations of various kinds. Associations model the rules of a problem domain. On class diagrams, associations are drawn as lines connecting classes. An association between two classes implies that at runtime the instances of the classes can be linked in some way, enabling them to invoke each other's operations. An arrowhead is indicative of the association's navigability. A line with no arrowheads denotes a bi-directional association—that is, at runtime objects on either side of the association can invoke an operation on the object on the opposite side.

Figure 5. Class relationships



Each association has an identifier, which can be a meaningful name or an automatically generated label (e.g., R2). The meaning of an association is expressed with a pair of association ends and a pair of multiplicity ranges. An association end can be either a role (e.g., "buyer") or a verb phrase (e.g., "rents"), as shown in Figure 5. The association end indicates the meaning of the association from the point of view of the class on the opposite end. Roles help overcome the following deficiency of class models. Once created, an instance belongs to its class forever. To show that this instance may serve different purposes at different times, we use roles: for example, an instance of class Employee can be acting as developer in one relationship and as a supervisor in another one.

A multiplicity range shows how many instances of the class on the near (next to the range) end can participate in a relationship with a single instance from the class on the far end. For instance, a room has exactly one owner (we have consciously excluded partnership), but an owner can own (at least) one or more rooms (see Figure 5). The most commonly used multiplicity ranges are: 1..1 (1, for short), 1..* (one or many), 0..1 (0 or 1), 0..* (0 or many).

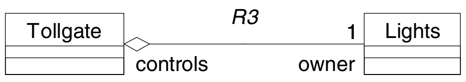
To describe a relationship, modelers take the view of an instance of each participating class, a technique called anthropomorphism. For example, let us consider association R2 in Figure 5. From the room's point of view: "A room is owned by one owner;" and from the owner's point of view: "An owner owns one or more rooms."

Aggregation and Composition

An aggregation is a kind of association denoting a "whole-part" relation between two classes. Since aggregation is a kind of association, all properties applying to associations apply to aggregations too. In class diagrams, the "whole" end is represented by an empty diamond.

Composition is a stronger form of aggregation, where the "whole" is responsible for creating and destroying the "part." The composition end is signified

Figure 6. Aggregation and composition

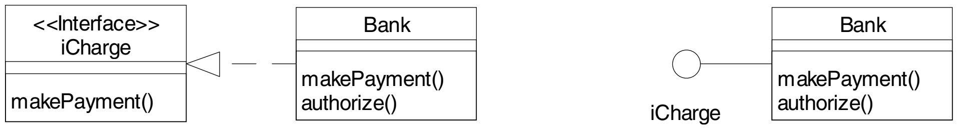


with a filled diamond. The real difference between aggregation and composition stands out only in coding as containment by reference and containment by value. In terms of modeling there are no semantic differences between the two associations.

Interface

An interface is a named collection of operation signatures (name, parameter types, return type), attributes, and a protocol state machine (a subset of FSM) defining a service. An interface specifies a contract for a service to be offered by a server class, without revealing any implementation details. It has no behavior by itself, and it cannot not be instantiated. The operations an interface specifies are implemented in a class (could be a structured class, see below), and at runtime they are provided by an instance of this class. The relationship between an interface and its realizing class is best described as type inheritance, since it signifies that the realizing class conforms to the contract specified by the interface—that is, the realizing class is type conformant to the interface type. In class diagrams, this relation is stereotyped as <<realize>> (see Figure 7). UML also uses the ball-and-socket notation, where the ball represents an interface to be realized by a class (see Figure 7). Interfaces are there to separate the specification of a service from its implementation. We say that a class realizes an interface if it provides operations for all the operations specified in the interface. One class may realize multiple interfaces.

Figure 7. Interface specifying a service and a class implementing the service



Generalization

When two or more classes share a common structure and behavior, the commonalities can be factored out and placed in a more general super class, from which the rest of the classes inherit (or derive). The super class has attributes, associations, or behavior that apply to all derived subclasses. The derived classes are said to specialize the super class by adding or modifying attributes, associations, or behavior.

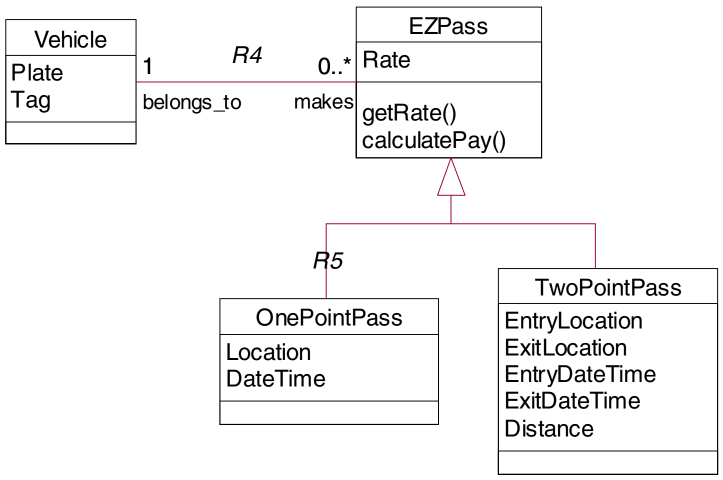
In Figure 8, EZPass is a superclass, and OnePointPass and TwoPointPass are subclasses. The subclasses can either specialize or extend the super class.

Generalization can be described as "is_a" relationship, for example, OnePointClass "is_a" kind of EZPass. The "is_a" relationship imposes interface compliance on the participating classes. This means that any occurrence of the superclass can be substituted with an instance of the subclass without semantically breaking the model.

Specializing, also called polymorphism, means that the same operation has a different implementation in the subclass. In UML, only operations and FSMs can be specialized (overridden or redefined); for example, operation CalculateRate in TwoPointPass is implemented differently from its analog in EZPass, even though it has the same interface. Extending means that the subclass can have new attributes, operations, states, or transitions.

Through generalization, we may avoid class modification by elegantly reusing the superclass functionality for creating specialized subclasses. The specialized

Figure 8. Generalization-specialization hierarchy



classes have some of their inherited responsibilities redefined and possibly new ones added. For example, if a need arises for a super-fast E-Z tollgate, we will not have to change anything in the existing tollgates. All that is required is to extend the generalization hierarchy with a new subclass for the new type of tollgate.

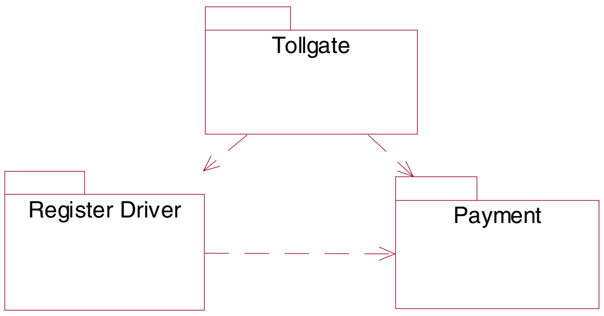
Package, Structured Class, Component, and Subsystem

To effectively develop large systems, we need to model their large-scale parts and how these parts interact with each other. To be scalable, a modeling notation should allow for parts to contain smaller parts, which can have, in turn, even smaller parts, and so on. At the most detailed level, parts are class instances. For modeling things at the large end of the spectrum, UML provides structured classes, components, and subsystems, all based on the notion of class. To develop large systems, it is also important to be able to divide the work into manageable units. For this purpose, UML provides packages.

Package

UML packages are a mechanism for organizing modeling elements, including other packages, into groups. A package defines only a namespace for the elements it contains. Packages are used to divide a system model into independent parts, which can be developed independently. Graphically, a package is rendered as a tabbed folder (see Figure 9).

Figure 9. Packages and <<import>> relation



A dependency is a using relation signifying that a change in the independent thing may affect the dependent thing, but not vice versa. <<import>> is a kind of dependency relation, which grants permission to the elements in one package to access the elements contained in another one. Relation <<import>> is rendered as a dotted arrow. The arrowhead points to the package being imported, as shown in Figure 9. It is important to note that the permission is one-way.

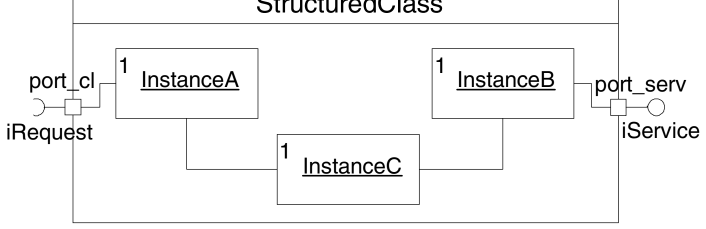
Structured Class

A structured class is a runtime container for instances of classes and other structured classes, collectively called parts or elements. These parts are interconnected with communication links named connectors, as shown in Figure 10. Apart from being in charge of creating and destroying its parts and connectors, a structured class may coordinate its parts' activities, for example, through an FSM.

A structured class contains its part through composition. An instance multiplicity number written in the corner of a part shows the number of instances from that part. The parts and the connectors constitute the part topology of the structured class.

A structured class offers a collection of services to its environment, published via <<provided>> interfaces and accessed at runtime via ports. A provided interface is a contract for service. If, in turn, a structured class uses the services of other instances (including structured classes), it can place demands on these instances by <<required>> interfaces. UML uses the ball-and-socket notation for the provided (ball) and required (socket) interfaces (see Figure 10). Ports can be thought of as being typed by their interfaces. Typed ports serve as

Figure 10. Structured class, ports, provide and required interfaces, and connectors



access points for interactions between the internal parts of the structured class and its environment.

Ports relay the incoming messages to the internal parts and the outgoing messages from the internal parts to the external objects attached to the structured class through connectors without revealing the part's identity.

To summarize, a port publishes the operations implemented by a collaboration of internal parts on the structured class border. The structured class boundary and ports decouple the internal elements from the external environment, thus making a structured class reusable in any environment that conforms to the required interfaces imposed by its ports.

Components

A UML component is a structured class providing a coherent set of services used and replaced together. A component's behavior is specified in terms of provided and required interfaces. We say that a component is typed by its interfaces. A component may be substituted by another one, only if the two are type conformant. There are no differences between structured classes and components, except for the connotation that components can be configured and used like Lego blocks in different contexts to build things.

At present, the most widely used commercial component frameworks are Enterprise Java Beans, .NET, COM+, and CORBA Component Model.

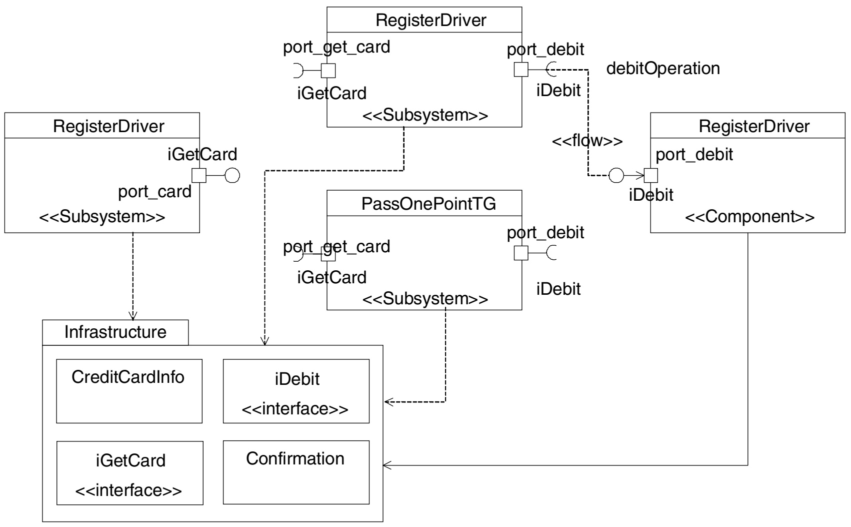
Subsystem

A subsystem is a large-scale component and a unit of hierarchical decomposition. At the highest level, a system is decomposed into several subsystems. In addition to a collection of services, a subsystem defines a namespace for its parts. A subsystem may have separate specification and realization parts stereotyped respectively as <<specification>> and <<realization>>. This makes it possible for one specification to have multiple realizations.

Components can be assembled in an enclosing <<subsystem>> container by wiring together their required and provided interfaces. The components' interfaces are linked either by connectors or by dependency relationships (see Figure 11).

The Infrastructure package is a container for common elements such as types (classes and interfaces) exposed in component interfaces, usage points (services), and extension points (classes to subclass in other packages).

Figure 11. Components, subsystems, and packages



Since structured classes, components, and subsystems are classes (unlike packages), they may participate in associations and generalizations.

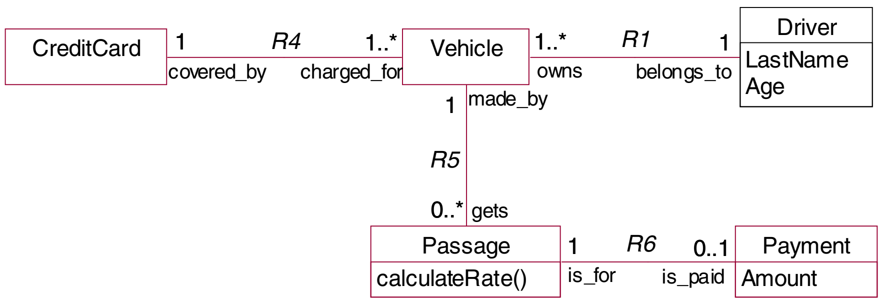
Constraints and Object Constraint Language

Constraints are a fundamental part of a system's structure and semantics. The Object Constraint Language (OCL) (Warmer & Kleppe, 1998) is a formal, pure expression language augmenting graphical UML models to produce unambiguous and precise system descriptions. OCL is an integral part of UML, and it is used to define the semantics of UML. OCL combines first-order predicate logic with a diagram navigation language. It provides operations on sets, bags, and sequences to support the manipulation and queries of collections of model elements.

UML uses OCL to express constraints, navigability, action semantics, and object queries. Even though visual models define some constraints, like association multiplicities, in OCL we can specify richer ones, such as uniqueness constraints, formulae, limits, and business rules. OCL constraints provide precision, which facilitates design by contract and executability (see Chapter II).

Very succinctly, a constraint is a semantic restriction on one or more model elements. Types of constraints include, but are not limited to, constraints on

Figure 12. Class diagram



associations between classes, pre- and post-conditions on class operations, multiplicity of class instances, type constraints on class attribute values, invariants on classes, and guards in state models.

Each OCL expression is written and evaluated in the context of an instance of a model element. Let the context be an instance of class Driver, shown in Figure 12 (the irrelevant elements on the class diagram are suppressed). We can restrict the driver's age to 16 years with the following constraint.

```
context Driver inv:
 self.Age > 16
```

As another example, if we want to make sure that all tags have unique IDs, we can write:

```
context Tag inv:
 Tag.allInstances()->forAll(t1, t2 |
 t1<>t2 implies t1.TagID <> t2.TagID)
```

Pre- and post-conditions specify the effect of a class operation without stating its algorithm or implementation. To indicate the operation for which the conditions must hold, we extend the constraint's context with the operation name. For example,

```
context CreditCard::Charge(Amount: Real): Boolean
 pre: self.IsValid
 post: self.Balance = self.Balance@pre + Amount
```

where Balance@pre refers to the value of the attribute at the start of the operation.

Next, we show how starting from a specific instance we can navigate its class associations to refer to related objects and the values of their attributes. To navigate association R6 in Figure 12, we use the association's label and the role predicate at the far end:

```
context Payment inv:
 self.Amount = self.R6.is_for.CalculateRate()
```

It is important to note that OCL is not a programming language, and OCL expressions cannot, therefore, express assignment statements or flow of control. The evaluation of an OCL expression is instantaneous, meaning that the states of the objects in a model cannot change during expression evaluation.

Often OCL expressions are critiqued for having poor readability and for being inefficient in specifying requirements-level and analysis-level concepts (Ambler, 2002). This critique comes from methodologists favoring code over models and from system analysts using UML diagrams as sketches of OO designs, the so called UMLAsSketch.[4] To improve OCL readability, Mellor (2002, p.128) introduces the notion of constraint idiom as a general pattern for commonly occurring types of constraints and suggests the use of predefined tags for these types. The latter makes models less cluttered and less cryptic (see Chapter 2). Since constraints are a fundamental part of the semantics of a problem domain, there is no simple alternative for building precise models.

With OCL, an object's interface can be defined as a set of protocols to which the object conforms. A protocol consists of the following three invariants defined for each operation: pre-condition, post-condition, and operation signature.

Constraints do not have to be written in OCL. They can be specified in any action language supporting the Precise Action Semantics for UML (UML AS, 2001), for example, OAL (2004) and KC (2004). However, action languages are not specification languages. Their purpose is to define computations in executable models.

Object Behavior

We are interested in how objects act both in isolation and in collaboration with other objects. In UML, the former behavior is modeled with FSM models, while the latter is modeled with communication (formerly collaboration diagrams) and sequence diagrams.

State Dependent Behavior

In order to determine the future actions of a class instance, we need to know its past, a rather Newtonian approach.[5] For example, the answer to the question "Can I withdraw $50 from my account?" depends upon the history of deposit and withdrawal transactions carried out. If the owner has deposited $60 and withdrawn $20, the answer will be "No," given that the account cannot be overdrawn.

We could succinctly represent the history of the bank account by its balance, $40, rather than the whole sequence of past bank transactions. To do so, we can add an attribute, say Balance, to class BankAccount, and base the behavior of operation Debit() on the value of the new attribute.

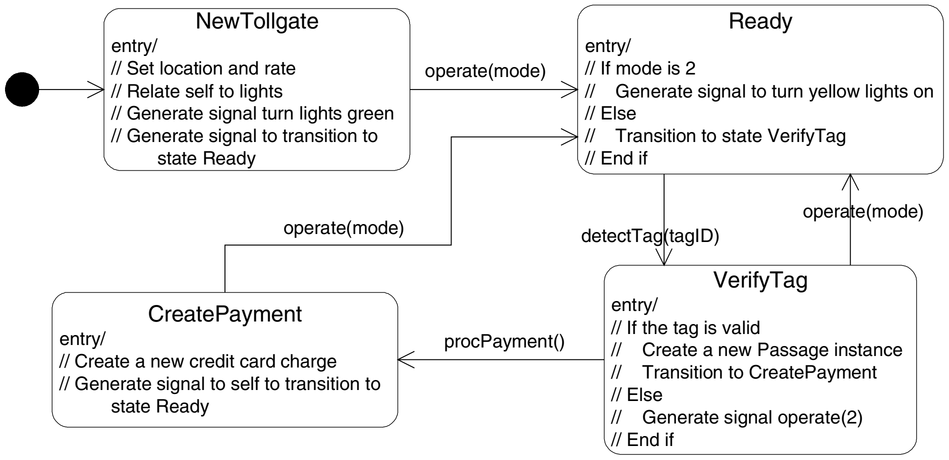
We call the abstract representation of an object's history *state*. The state of an object represents that object's potential, or in other words, the actions in which the object can be engaged.

It takes a lot of sophisticated and error-prone code to manage the concurrent access to the data and operations of objects with a great number of states. For such objects, it is better to model explicitly their states, state evolution, and the relation between states and allowed actions. An object can progress through different states over its lifetime. This progression is called lifecycle. An FSM model defines the lifecycle of a state-dependent class. In UML, statecharts diagrams visualize FSMs. FSM models comprise four types of elements: states, transitions, events, and procedures.

A state represents a stage in the lifecycle of an object. The states are a bounded set of mutually exclusive conditions of existence: at any moment, an object is in exactly one state, referred to as current state. Signals are incidents in the domain modeled as events. They can be internal (generated by the FSM) or external (generated by other FSMs). A transition points to "the next" state, which the object will enter from the current state upon receipt of a particular event. States are drawn as rectangles with rounded corners and transitions as arrows. Each state has a name. A transition is labeled with the event that causes the transition.

In the executable UML profile, the FSM model includes procedures of actions (executable sections), one per state, executed on entry to a state. Figure 13 shows the FSM for class Tollgate. For readability's sake, we have written the state procedures in pseudo code. The initial state for the FSM is NewTollgate. In this state, the tollgate controller sets its own location and rate, and then it relates itself to an instance of class Lights. Once related, the tollgate controller signals the Lights instance to turn its green lights on. Next, the initialized tollgate generates signal operate(mode) to itself to transition to state Ready. This signal is shown as an event attached to the transition from state NewTollgate to state Ready. In state Ready, the tollgate checks the mode value provided by event operate(mode) to see if it has to send a signal to the lights controller to turn its yellow lights on, and then waits passively to receive a "tag detected" signal from its sensor. Upon receipt of detectTag(), the tollgate enters state VerifyTag and executes its state procedure. The procedure verifies the detected tag, creates a new Passage instance, and transitions to state CreatePayment. If the tag is invalid, the tollgate goes back to state Ready. In state CreatePayment, the

Figure 13. FSM model for class Tollgate


NewTollgate
entry/
// Set location and rate
// Relate self to lights
// Generate signal turn lights green
// Generate signal to transition to state Ready
operate(mode)
Ready
entry/
// If mode is 2
// Generate signal to turn yellow lights on
// Else
// Transition to state VerifyTag
// End if
operate(mode)
detectTag(tagID)
operate(mode)
VerifyTag
entry/
// If the tag is valid
// Create a new Passage instance
// Transition to CreatePayment
// Else
// Generate signal operate(2)
// End if
procPayment()
CreatePayment
entry/
// Create a new credit card charge
// Generate signal to self to transition to state Ready

tollgate retrieves the credit card, serving the vehicle related to the detected tag, and creates a Payment instance, taking care of charging the credit card. At this point, the tollgate returns to state Ready to process the next vehicle.

Not all classes exhibit interesting lifecycles. The behavior of a class can be defined as state dependent or simple (state independent). FSM models are not constructed for simple classes.

Collaborative Behavior

In order to work toward achieving a common goal, objects interact by calling synchronously each other's operations, called message passing, or by sending asynchronously signals modeled as events. Interactions can be visualized on interaction diagrams, which come in two main forms: sequence and communication diagrams.

Unlike FSMs, interaction diagrams do not tell the complete story about the involved objects. Instead, they focus on particular sequences of messages or events, called scenarios or traces. Scenarios are used to communicate with stakeholders or to conduct interaction analysis. Interaction analysis aims at understanding and defining the communication patterns within a collaboration of objects realizing one or more user requirements.

If we have to compare interaction diagrams to FSMs, we can say that, apart from their scope—individual versus group—the former provide a black box view of the participating instances, while the latter afford a white box view of the ways in which instances respond to signals and messages. For this reason, interaction diagrams are said to be partially constructive, and as such, they cannot be used to generate code.

Sequence Diagrams

Once the elicited user requirements are modeled with use cases and the classes supporting the use cases are discovered, it is time to verify the decisions made. An important objective in the early stages of a project is to mitigate risks and to gain a high level of confidence in the discovered classes and their responsibilities. Mistakes at early stages are costly since they result in backtracking and extensive rework.

Interaction analysis is a technique employing sequence diagrams to analyze the dynamics of the classes derived from a use case description. Sequence diagrams have the following uses: (1) to analyze the dynamic interactions between class instances and to reduce the risk of incorrect class discovery; (2) to create traceability relationships from use cases to class diagrams;(3) to tackle the problem of allocating responsibilities among classes and to define the class interfaces (public operations); and (4) to plan functional system testing.

Sequence diagrams display the causal dependencies among messages and signals exchanged between lifelines of instances and their relative time ordering. Figure 14 shows how a sequence diagram portrays the time ordering of the messages exchanged among instances during the execution of a scenario. The instances taking part in the scenario are arranged along the X-axis, with the initiating instance (usually an actor) being on the far left side. The lifelines of the instances are represented as vertical bars, with time flowing from top to bottom. Messages and signals are modeled as arrows pointing to the receiver. Signals are modeled with a half arrowhead, while messages are modeled with a full arrowhead. The direction of the arrowhead is indicative of the invocation direction and not of the data flow. The lifeline of an object created or destroyed during an interaction starts or ends, respectively, with the receipt of a message or signal. A large X marks the end of a lifeline. The focus of control is a rectangle superimposed over a lifeline, indicating that the instance is performing an action. The top of the rectangle coincides with the start of the action, while its bottom aligns with the action's completion.

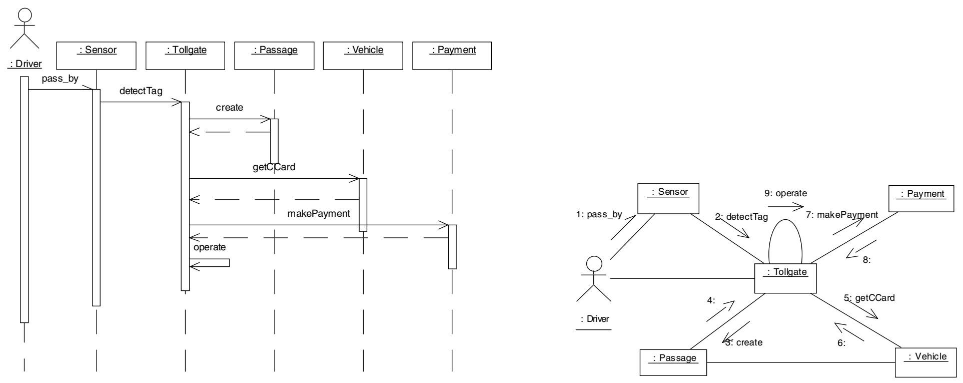
The black box view offered by a sequence diagram can be cracked open if instead of focus of control, the modeler shows the names of the states through which the instance progresses.

Communication Diagrams

A communication diagram is an object diagram showing the messages and their ordering attached to the static links between the objects, as illustrated in Figure 14. A communication diagram emphasizes the organization of the objects.

To sum up, interaction diagrams lack the completeness of FSMs. They are used primarily to explore and verify a particular scenario rather than build an artifact. During exploration, new messages, signals, operations, and even classes can be discovered and added to the model.

Figure 14. Sequence and communication diagrams



Some Merits and Deterrents

Advantages

According to many authors, OO is advantageous because it allows for a seamless transition between the phases of software development by using a uniform language for requirements engineering, analysis, design, and programming.[6] This is a major prerequisite for transformational development, where the system is built in a sequence of transformations or refinements from more abstract to more detailed models. Each transformation or refinement preserves the source model properties in the target model.

OO narrows the semantic gap between entities in the problem and the solution domains, leading to better traceability, and ultimately, to better chances of validating the software artifacts by customers.

Improved interpersonal communications through objects is another success factor. OO encourages a common vocabulary between end users and developers (Booch, 1996).

As a system evolves, the function or the business processes it performs tend to change, while the object abstractions tend to remain unchanged. Thus, removing the emphasis on functions results in more maintainable designs. When changes are necessary, OO contains changeability through locality of change.

Inheritance and polymorphism make OO systems easier to extend.

OO analysis emphasizes the importance of well-defined interfaces between objects, which led to the development of components.

Disadvantages

Traditionally, OO methods have paid little attention to business modeling, a very important area of software development. The enterprise business model represents the business concepts in the problem domain. An explicit description of the system's business processes leads to a better understanding of the role of the system in the organization, and therefore, the resulting system is more likely to be appropriate to the problem being solved.

Though OO modeling is supported by object abstraction, this device alone cannot capture aspects bigger than small-scale entities. There are many pervasive requirements (e.g., security, concurrency, transaction management) that do not decompose into behavior centered on a single locus. In addition, the gains from reusability at the object (i.e., class) level are insignificant. The real benefit from reusability comes at the component level, and even more so at the architecture level and domain level.

OO might exacerbate the "analysis-paralysis" problem. Because of the uniform notation, there is a strong temptation to design rather than perform analysis.

Another downside of OO modeling is that it takes myriads of models to understand and express objects: static (class and deployment diagrams), behavior models (sequence, communication, diagrams, and use cases), and state-change models (statechart and activity diagrams).

UML2 is expected to add new features to the OO modeling repertoire. This raises reasonable concerns and is rightfully perceived by developers as a language bloat. To counter this effect, the creators of UML2 maintain that the modeling language has been compartmentalized in two ways. First, the different sub-languages are meant to be independent of each other, which, if true, threatens to invalidate the multi-view approach to system modeling. Second, they say that learning UML2 is not an "all or nothing" proposition. UML modelers will be able to select only those parts that are useful in solving their problem and safely ignore the rest. In our view, the latter is easier said than done. In order to choose which part of UML to leave out, the developer should be familiar with the capabilities and limitations of all parts of the language, so sufficient breath of knowledge is still required.

Conclusion

It is firmly believed that the solution to the current software crisis depends on raising the level of abstraction of the constructed software artifacts and on increasing the level of automation in the software design process. “Divide and conquer” proved insufficient to contain the software complexity. This point is illustrated by the revised Roman adage, “Divide to conquer, but unite to rule.” Abstraction and automation are the only effective means we can apply to curb complexity that overwhelms our cognitive capacities. “Un-mastered complexity” is the root cause for the software crisis.

The use of models with executable semantics is set to move into the mainstream of OO software development. Automation through executability challenges the tacit assumption that software development will remain the same type of mental activity it has always been—that is, given a problem, a developer should liberally apply creativity to synthesize a solution. The “liberal creativity” approach rules out a quantum leap in the ability to develop software, because the human mind does not evolve in sync with the growing software complexity. As Herbert Robins has stated, “Nobody is going to run 100 meters in five seconds, no matter how much is invested in training and machines. The same can be said about using the brain. The human mind is no different now from what it was five thousand years ago. And when it comes to mathematics [understand software], you must realize that this is the human mind at an extreme limit of its capacity.”

Dijkstra’s reaction to this statement was “So reduce the use of the brain and calculate!” and then, Dijkstra went on to elaborate that “for going from A to B fast, there exist alternatives to running that are orders of magnitude more effective.” Robbins’ argument fell flat on its face.

The advantage of adding automation to object-oriented models is, in Dijkstra’s spirit, increased use of calculation at the expense of “liberal creativity” to master the software complexity and put the software crisis to rest.

With less emphasis on programming, requirements engineering and systems analysis will become more and more important as time goes on. Mellor has aptly said that the model is the code. Since models are derived from user requirements, we can conclude by transitivity that in the OO future, the user will be the code.

References

Ambler, S. (2002). Toward executable UML. *Software Development Journal*, (January).

Atkinson, C., Bayer, J., & Muthig, D. (2000). Component-based product line development. The KobrA approach. *Proceedings of the 1st Software Product Lines Conference* (SPLC1) (pp. 289-309).

Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., & DeBaud, J. (1999). PuLSE: A methodology to develop software product lines. *Proceedings of the Symposium on Software Reusability* (SSR'99).

Beck, K. (1999). *Extreme programming explained: Embrace change*. Reading, MA: Addison-Wesley.

Beck, K., & Cunningham, W. (1989). A laboratory for teaching object-oriented thinking. *Proceedings of OOPSLA '89. ACM SIGPLAN Notices, 24* (pp. 1-6).

Bettin, J. (2004). Model-driven software development: An emerging paradigm for industrialized software asset development. Retrieved from *http://www.softmetaware.com*

Bettin, J. (2005). Managing complexity with MDSD. In B. Roussev (Ed.), *Management of the object-oriented software development process*. Hershey, PA: Idea Group Inc.

Booch, G. (1993). *Object-oriented analysis and design with applications* (2nd ed.). Reading, MA: Addison-Wesley.

Booch G. (1996). *Object solutions*. Reading, MA: Addison-Wesley.

Booch, G. (1996). *Object solutions: Managing the object-oriented project*. Reading, MA: Addison-Wesley.

Bosch, J. (2000). *Design and use of software architectures: Adopting and evolving a product line approach*. Reading, MA: Addison-Wesley.

Brooks, F. (1998). *The mythical man-month: Essay of software engineering* (anniversary ed.). Reading, MA: Addison-Wesley.

Chen, P. (1977). *The entity-relationship approach to logical data base design.* Wellesley, MA: Q.E.D. Information Sciences.

Coad, P., & Yourdon, E. (1991). *Object-oriented design.* Englewood Cliffs, NJ: Prentice-Hall.

Codd, A.F. (1970). A relational model for large shared data banks. *Communication of the ACM*, *13*(6).

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., & Jeremaes, P. (1994). *Object-oriented development: The fusion method*. Englewood Cliffs, NJ: Prentice-Hall.

Cook, S., & Daniels, J. (1994). *Designing object systems: Object-oriented modeling with Syntropy*. Englewood Cliffs, NJ: Prentice-Hall.

Czarnecki, K., & Eisenecker, U. (2000). *Generative programming: Methods, tools, and applications*. Reading, MA: Addison-Wesley.

D'Souza, D., & Wills, A. (1998). *Objects, components and frameworks with UML: The catalysis approach*. Reading, MA: Addison-Wesley.

Firesmith, D., Henderson-Sellers, B., & Graham, I. (1998). *OPEN modeling language reference manual.* New York: Cambridge University Press.

Fowler, M. (2003). *UML distilled: A brief guide to the standard object modeling language* (3rd ed.). Reading, MA: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.

Garzás, J., & Piattini, M. (2005). Improving the OO design process using rules, patterns, and refactoring. In B. Roussev (Ed.), *Management of the object-oriented software development process*. Hershey, PA: Idea Group Inc.

Graham, I. (1994). The SOMA method: Adding rules to classes. In A. Carmichael (Ed.), *Object development methods* (pp. 199-210). New York.

Graham, I., & Wills, A. (2001). UML—A tutorial. Retrieved from *www.trireme.com*

Henderson-Sellers, B., & Edwards, J.M. (1994). *BOOKTWO of object-oriented knowledge: The working object*. Prentice-Hall.

Jacobson, I. (1987). Object-oriented development in an industrial environment. *ACM SIGPLAN Notices*, *22*(12), 183-191.

Jacobson, I. (1992). *Object-oriented software engineering: A use case driven approach*. Reading, MA: Addison-Wesley.

Kiczales, G. (1996). Aspect-oriented programming. *Computing Surveys*, *28*(4), 154.

Kruchten, P. (2000). *The rational unified process®: An introduction.* Reading, MA: Addison-Wesley.

Lacan, J. (1977). *Ecrits: A selection* (A. Sheridan, trans.). London: Tavistock.

Lamsweerde, A. (2000). Requirements engineering in the year 00: A research perspective. *Proceedings of the 2nd International Conference on Software Engineering.* Limerick: ACM Press.

Laplante, P.A., & Neill, C.J. (2004). "The demise of the waterfall model is imminent" and other urban myths. *ACM Queue*, (February), 10-15.

Martin, J., & Odell, J. (1992). *Object-oriented analysis & design*. Englewood Cliffs, NJ: Prentice-Hall.

MDA. (2004). OMG Model-Driven Architecture. Retrieved from *www.omg.org/mda*

Mellor, S.J., & Balcer, M.J. (2002). *Executable UML: A foundation for Model-Driven Architecture.* Reading, MA: Addison-Wesley.

Miller, J. (2002). What UML should be. *Communications of the ACM*, *45*(11).

OAL. (2004). BridgePoint object action language. Retrieved from *www.projtech.com*

Paige, R.F., & Ostroff, J.S. (1999). A comparison of the business object notation and the Unified Modeling Language. In R. France & B. Rumpe (Eds.), *Proceedings of <<UML>> '99—The Unified Modeling Language Beyond the Standard*, Fort Collins, Colorado, USA.

PL. (2004). SEI collection of resources on product lines. Retrieved from *www.sei.cmu.edu*

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Loensen, W. (1991). *Object-oriented modeling and design*. New York: Prentice-Hall.

Shlaer, S. & Mellor, S.J. (1988). *Object-oriented systems analysis: Modeling the world in data*. Englewood Cliffs, NJ: Prentice-Hall.

Shlaer, S., & Mellor, S.J. (1992). *Object lifecycles: Modeling the world in states*. Englewood Cliffs, NJ: Prentice-Hall.

UML. (2004). Unified Modeling Language. Retrieved from *www.uml.org*

UML AS. (2001). UML Action Semantics. Retrieved from *www.omg.org*

Warmer, J., & Kleppe, A. (1998). *The Object Constraint Language: Precise modeling with UML.* Reading, MA: Addison-Wesley.

Wendorff, P. (2001). Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. *Proceedings of the European Conference on Software Maintenance and Reengineering* (pp. 77-84).

Wirfs-Brock, R., Wilkerson, B., & Wiener, L. (1990). *Designing object-oriented software.* Englewood Cliffs, NJ: Prentice-Hall.

Withey, J. (1996). Investment analysis of software assets for product lines. Retrieved from *http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.010.html*

Endnotes

1 "The limits of my language mean the limits of my world" - Ludwig Wittgenstein.

2 DSLs (domain specific languages) are modeling languages targeting a particular technological domain or problem.

3 The linguistic approach to object-orientation is an ongoing interdisciplinary project with Yvonna Rousseva, who is credited with the idea of redefining objects through the premises of the theory of deconstruction.

4 Fowler (2003) divides the use of UML into UMLAsSketch, UMLAs BluePrint, and UMLAsProgrammingLanguage.

5 In Newtonian mechanics, if we know the trajectories of a system of bodies, we can determine the positions of the bodies at any future moment.

6 Other authors see this as a weakness, for example, Knott et al. (2005), included in this book.

Chapter II

MDA with xUML: Model Construction and Process Management

Boris Roussev
University of the Virgin Islands, USA

Abstract

xUML epitomizes the convergence of visual modeling with model manipulation programming. The results of this merger are executable models and model-driven software development. This chapter presents the fundamental notions of constructing executable domain models with xUML, as well as the principles of the MDA approach. In particular, we define the new roles of the developers in development processes based on MDA and the MDA activity workflow. We discuss also the output artifacts from each activity. As new technologies require new software development processes, we present an iterative and incremental model-driven process, combined with techniques for project planning and progress estimation based on BERT and ERNIE. We show how model executability creates congenial conditions for the application of higher-order cognitive skills in the software development process, and for the substitution of liberal creativity with design automation.

Introduction

C.A. Petri was the first to define formally the notion of communicating state machines in his PhD thesis, "Kommunikation mit Automaten," back in 1962. His visual modeling language for modeling concurrency and synchronization, later known as Petri nets, is an extension of the Finite State Machine (FSM) theory. Even though Petri nets is an intuitive and powerful process coordination language, several pieces were crucially missing to bridge the Petri nets paradise to the real world. Petri nets lacked an information structure model such as class diagrams, a development process for effective product development from informal requirements such as Agile (Beck, 1999), and action semantics (other than transition firing rules). Action semantics defines a virtual machine for model interpretation, giving substance to the notion of model "executability."

The Shlaer-Mellor method (Shlaer & Mellor, 1988, 1992), one of the first object-oriented (OO) analysis methods, uses class diagrams to represent the information structure of a system. This information model was influenced by the relational theory of data (Codd, 1970) and database modeling with entity-relationship diagrams (Chen, 1977). Shlaer and Mellor reinvented the idea of communicating FSMs in the context of OO by employing FSMs to abstract lifecycles of objects, whose progression is driven by external or internal asynchronous signals. They describe objects' execution behavior as state procedures consisting of actions. The actions perform tasks on modeling elements; for example, they traverse an association link to retrieve the value of an attribute in a related instance, or generate signals to the FSMs of related objects. Shlaer and Mellor advanced the idea of composing complete systems out of executable models.

Shlaer and Mellor put at the top of the OO agenda the notion of model executability. Their method evolved into a pure object-oriented notation (Mellor & Balcer, 2002; Mellor et al., 2004), which is currently shaping up the future of UML.

Model-Driven Architecture (MDA) (Mellor & Balcer, 2002) is a term defined by OMG. The MDA approach to software development relies on Executable UML (xUML) (MDA, 2004), a UML profile with executable semantics. MDA distinguishes between Platform Independent Models (PIM) and Platform Specific Models (PSM).

In MDA, software does not need to be programmed at all, or at least not by humans. This can be achieved through the integration of a programming language at a high level of language abstraction into a visual modeling language. The result is a modeling language with executable semantics. The models constructed with such languages can be simulated and debugged—that is, validated at model level and then compiled to source code for any target platform. This eliminates the verification gap between visual software models and end users. As a result, a system can be delivered in small increments of executable models produced in fast cycles.

MDA spearheads the new wave of OO technologies aiming at design automation and at IT artifacts construction at a higher level of abstraction. MDA is kith and kin with technologies such as generative programming (Czarnecki & Eisenecker, 2000) (transformational mappings), aspect-oriented programming (Kiczales, 1996) (merging mappings), design patterns (Gamma, Helm, Johnson, & Vlissides, 1995) (modeling societies of objects along with their interactions), product lines (PL, 2004) (clear separation between application logic and software architecture), MDSD (Bettin, 2004) (design automation), Microsoft's software factories, domain engineering (PL, 2004) (software assets), and domain-specific languages (Czarnecki & Eisenecker, 2000) (metamodeling and domains).

This chapter has several objectives: 1) to examine the relationship between cognitive skills and model-driven software design; 2) to justify the adoption of the MDA approach; 3) to present in sufficient detail the xUML technology; 4) to define the developer roles in the MDA process; 5) to present the operational structure of an MDA process; and 6) discuss issues in planning xUML projects.

The examples in this chapter are from the E-ZPass system described in Chapter 1. The rest of the chapter is organized as follows. First, we discuss how merging mappings, called bridges, resolve the interface problem in component-based software design and justify the adoption of MDA. Next, we show how to construct executable models and present an iterative process for model-driven development with xUML. In the final section, we offer our conclusions.

Abstract Executable Assets

In this section, we discuss how model executability, model mapping, and software design at a higher level of abstraction blend together to produce software assets.

Constructing Assets with Late Component Externalization

Objects have not lived up to all expectations. They have turned out to be disproportionately small compared to the size of the systems being built with them. Gamma et al. (1995) showed the advantages of reusing collections of related objects. In the E-ZPass system, for example, a vehicle inevitably belongs to a driver, and it only makes sense to reuse these two objects together. Societies of closely related objects provide the foundation for components. A component is an encapsulation of objects exposing a set of interfaces. With components, a system is described in terms of interactions among component interfaces and is built by wiring up these interfaces. There are two major challenges to component-based software development. The first is active dependency management—how to minimize the dependencies among component interfaces. A component is said to have a fully externalized interface if all types exposed in its interfaces are defined in a package residing outside the component (Bettin, 2005). The fully externalized interface defines the usage environment of the component, and decouples the component from its peers. However, even with active dependency management, a small change in one of the interfaces entails locating every place where the interface is used, changing it to use the new interface, unit test the new code, reintegrate, and then integration test the whole system (Mellor et al., 2004). The second challenge to component-based development is how to wire up the component interfaces. Wiring is done with glue code, which prevents the reuse of the individual components in a new environment since the components become dependent on the glue code. The remedy to this problem is to apply late (or dynamic) externalization, meaning to externalize the interfaces and apply the glue code just before the system is deployed. Late externalization makes each component a reusable asset, and not a liability (Bettin, 2005).

MDA achieves late externalization through bridges. Bridges are formal mappings between modeling elements of two executable models. Ironically, interfaces decouple components, and in turn, bridges decouple (or localize) interfaces.

Do Not Work Harder, Work Smarter and Get Help

It is well known that software creators labor away at their products under the intense pressure of the time-to-market force and the ever-increasing demand for product complexity. Given a problem, there are three ways to devise a solution faster: work harder, work smarter, or get help. In this chapter, we focus on "work smarter" and "get help."

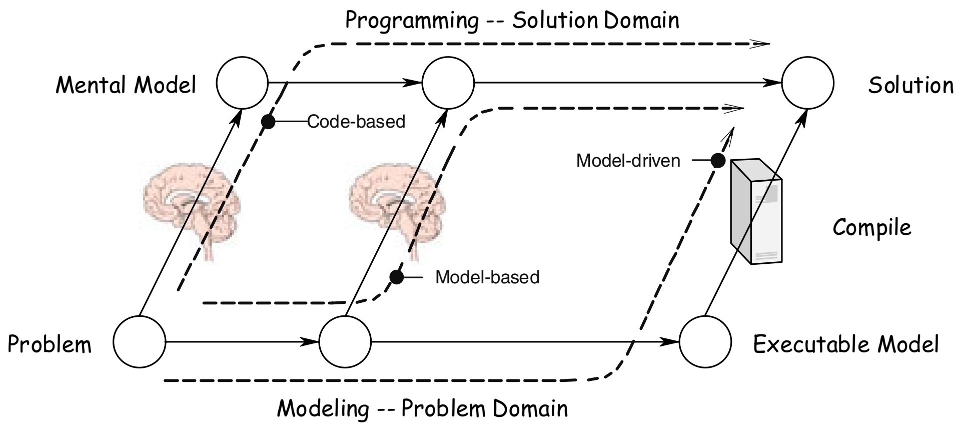
Devising a solution is an iterative process of four steps: comprehension, analysis, synthesis, and evaluation (Bloom, 1956). In comprehension, the problem solver acquires some (not necessarily deep) understanding of the problem domain. The ability to identify and interpret conceptual entities is essential. Comprehension is followed by analysis, where the problem is divided into sub-problems, after which partial solutions are designed. The next higher level is synthesis. Here, the problem solver needs to fit the partial solutions together and design the overall solution. The ability to predict the behavior of a system resulting from composing components or from applying a transformation to a component is central. Finally, evaluation is the most advanced level. It requires judging a solution by different criteria and finding logical fallacies. The ability to judge implies the abilities to analyze and synthesize in order to consider alternative solutions. This makes evaluation the most difficult sub-process.

In OO software development, UML graphical models are used to create visio-spatial metaphors for software systems. In many realms, pictorial code for organizing and presenting information is superior to verbal code, for example, maps and civil engineering blueprints. By analogy, the assumption is that visual models in software development facilitate comprehension, communication, and inference better than text (source code) models.

In conventional software development, synthesis equates to programming, while evaluation amounts to testing the executable code. Unfortunately, UML models cannot be of much help in the higher-level cognitive skills, namely, synthesis and evaluation. This is used as a cogent argument against modeling, or at best, against the large-scale use of modeling. In Agile, for example, modeling is relegated to drawings for the purpose of communication because modeling fails the "proving with code" principle, the Agile term for evaluation. Agile modelers suggest that the model be extracted from the code each time a program is changed, which is right under the assumption that the long-lasting asset of the development effort is the system code.

The model of a language is called *metamodel*. We would like to distinguish between the language describing a problem and the language describing its solution. The problem is described in the language of the customer. The solution, on the other hand, is expressed in an implementation language, at a much lower level of language abstraction. The semantic gap between the

Figure 1. Software production approaches



problem and the solution is the conceptual distance between the metamodels of the problem and solution languages.

The commutative diagram in Figure 1 shows three possible ways of producing software.

With the code-driven approach, the leftmost dotted line, the developer comprehends the problem in terms of some programming language, or figuratively, climbs up the "hard" way the semantic slope from the problem to the solution domain. Analysis, synthesis, and evaluation are carried out in the solution domain. Since the customer is the ultimate authority accepting the software solution, there is a need to switch constantly between the problem and solution domains.

With the model-based approach, on the other hand (the middle dotted line in Figure 1), comprehension and analysis are done in the problem domain, that is, in the language of the customer, or at least, in a language whose metamodel is very close to that of the customer's language. Then, the developer transforms manually[1] the analysis artifacts to a mental image expressed in some programming language. Synthesis and evaluation, as with the code-driven approach, are done in the solution domain. In the model-based approach, models are a little more than documentation for the code. Models help developers comprehend the problem domain, conceive the software architecture, and partially shape the design. Since models are visual and abstract, they are easier to understand, communicate, and synthesize than code. In theory, model-based development should improve productivity and quality dramatically, but in practice there is little evidence to that effect.

With model-driven development, the rightmost dotted line in Figure 1, comprehension, analysis, synthesis, and evaluation are done in the language of the customer because models are endowed with executable semantics, and as such, they do not fall short of the "proving with code" principle. This approach is "smarter" because no mental effort on the part of the developer is required to switch between the problem and solution domains. Since a model compiler automatically generates (understand design) the implementation from the conceptual solution, we "get help" from the computer. The model compiler trades "creativity" for calculation in much the same way as spreadsheets reduce repetitive "creative" addition of numbers to a computation defined by a formula.

Both in code-driven and model-based development, program code is the sole business asset to be produced. Artifacts other than code are considered aiding communication at best, and red tape at worst. OO or not, code is amorphous and cannot be easily reused. In model-driven development, however, executable models become the major business assets to be built and to last.

Building Executable Models

A model is expressed in a modeling language with abstract syntax and semantics. The abstract syntax defines the modeling primitives that can be composed to build a model. The language semantics defines the meaning of these primitives. A model that can be converted mechanically to executable code is executable. xUML is a modeling language based on the Precise Actions Semantics for UML (UML AS, 2001). The action semantics is not an action language. It defines only the semantics for creating objects, sending signals to objects, and accessing objects' data, as well as the semantics for functional computations. A particular action language must be used to express these computations.

A distinguishing feature of all executable models is their precision and consistency. Without being formal and accurate, executable models cannot be simulated and later translated to code. To build precise executable models, we need to construct at least three different but entirely consistent types of models. Figure 2 gives a high-level view of model-driven development with xUML.

Class diagrams represent the information structure of the system, FSMs represent the objects' lifecycles and execution behavior, and sequence diagrams represent objects' communication patterns over a relative time axis. FSMs and communication diagrams jointly define the behavioral aspects and constraints of the modeled objects. Each of these separate views can be thought of as a projection of the underlying system model on a different plane.

An important distinction of xUML models is that the code is not considered a first class (text) view of the system. In xUML, the relation model-code is not commutative. The code is the result of an irreversible transformation of the executable model. There is no round-trip engineering since code is not allowed to deviate from the model.

The first step in MDA is requirements gathering and domain identification. When developing complex software systems, the problem being addressed often includes a number of different and unrelated subject matters. In xUML, each subject matter is abstracted as a separate *domain*. Domains are analyzed independently. Different domains may interact with each other using the client-server model: for example, one domain acts as a service provider, while another takes the role of a client. To partition the application into domains, developers must first gain good understanding of the user requirements through use case modeling. The result of requirements gathering and domain identification is the informal system model shown in Figure 2, where user requirements are

Figure 2. Views of executable models in xUML

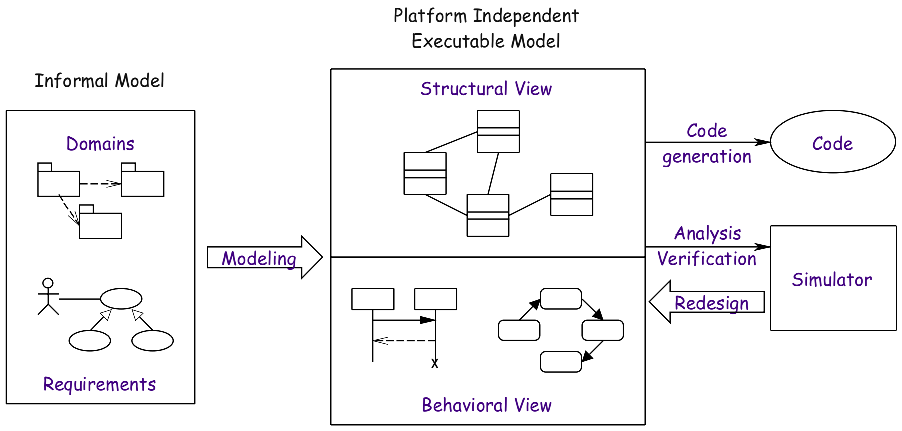
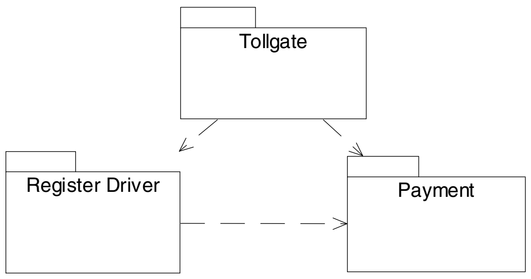


Figure 3. Domain chart



modeled as use cases and subject matters as domains. To show the client-server relationships among domains, we depict them on a domain chart (see Figure 3). Each domain is represented with a package symbol (tabbed folder), and each client-server dependency with a dotted line from the client domain to the server domain (similar to package source code dependency relationship). Domain dependencies are called bridges. Packages are used to model domains because they are a mechanism for organizing modeling elements into groups.

The platform-independent executable model is developed from the informal requirements model. It has two complementary views, structural and behavioral. A class diagram describes the conceptual entities and their relationships in a single domain. The lifecycle of a class is specified by an FSM model, while object interactions are portrayed on sequence diagrams. Each state in an FSM has a state procedure. A state procedure consists of a sequence of actions, and it is executed upon state entry. The actions access instance data and perform functional computations and synchronization.

As domain modeling progresses, the partially completed executable model can be simulated to test its behavior against scenarios derived from the use case model. Testing may expose both modeling and requirements errors. The executable model (and if necessary, the requirements model) is redesigned to fix the errors and is tested again. Upon completion, the executable domain models of the different domains are linked together through the bridges to form the complete system, and then are compiled to source code for the target software platform.

Next, we discuss in greater detail the construction of executable domain models.

Domains and Use Cases

Requirements analysis starts with use cases. As usual, use cases model the actors communicating with the system, the goals of the actors, the value those actors expect to get from the system, and the way actors interact with the system.

Domains differ from subsystems in a subtle way. Domains are logically consistent because they are populated with the conceptual entities of a single subject matter. These conceptual entities belong together and cannot be reused separately. In contrast, the boundaries of a system package are somewhat arbitrary. They are defined mainly to slice a big project into manageable subprojects.

A domain may group several use cases, but it is also possible for a use case to cut through several domains. If domains are too large for a single team, they may be decomposed into subsystems, but not into sub-domains, because the results of any domain decomposition will violate the logical consistency requirement. Since domains are independent subject matters, they can be built iteratively and incrementally.

Classes, States, Events, and Actions

xUML is a platform-independent language, defined at a language level of abstraction high enough so that it can be translated into any software platform. For example, a class in xUML may be mapped to a Java class, running on an enterprise server, or to a C structure, running on an embedded device.

xUML uses standard class diagrams brimming over with constraints to model the conceptual entities and their relationships living in the domain of interest. Most concepts fall into one of the following five categories: tangible things, roles (e.g., people), incidents (e.g., events), interactions (e.g., transactions), and specifications (e.g., type of service or type of product) (Mellor & Balcer, 2002).

Constraints are a fundamental part of a domain's semantics. They are rules that restrict the values of attributes and/or associations in a class model. For example, objects may have one or more attributes that are required to be unique. In xUML, analysts capture such domain knowledge through identifiers. An identifier is a set of one or more attributes that uniquely identify each object of a class (e.g., CardNumber in class CreditCard). We use the tag {I} to designate identifying attributes. This tag is a constraint idiom for the more cryptic OCL expression. Identifiers are also used to formalize associations by the automatic creation of referential attributes in associated classes.

xUML relies on associations, association classes, and generalization to abstract the relationships among the conceptual entities in a domain. Every association has a unique association number (an automatically generated label, e.g., R2). The association has a pair of roles (one at each end) and a pair of multiplicity ranges (one at each end). The roles indicate the meaning of the association from the point of view of each class. A multiplicity range indicates how many objects of one class are related to a single object of the other class.

Now we can define precisely what referential attributes are. An attribute referring to an instance of an associated class is called referential. For example, attribute CardNumber in class Payment is a referential attribute since it refers to attribute CardNumber in CreditCard. Referential attributes are tagged with {R*n*}, where *n* is the association number of the association being navigated. The value of a referential attribute of an instance is the value of the identifying attribute in the related instance of the associated class. Analysts are not required to use the same name for the identifying attribute and referential attribute, but it certainly helps. The tag {R*n*} is another example of a constraint idiom. Figure 4 illustrates the concepts we discussed.

Figure 4. Domain information structure

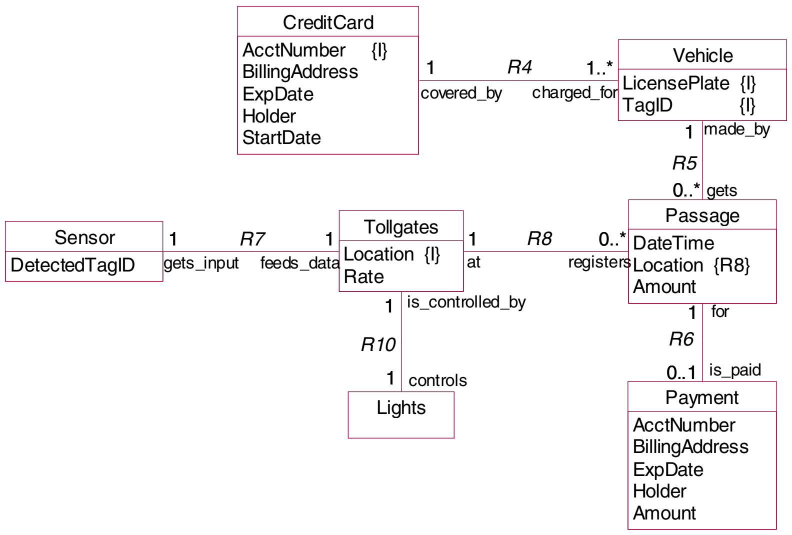
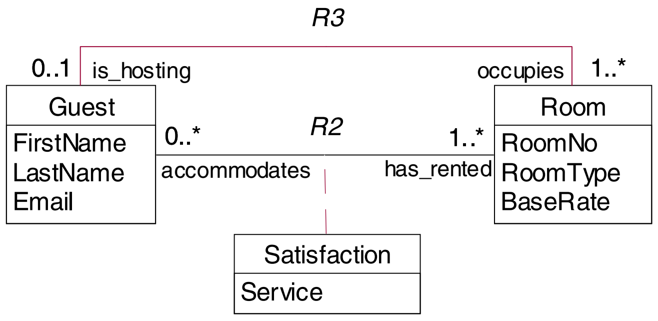


Figure 5. Association class Satisfaction capturing data structure of association R2



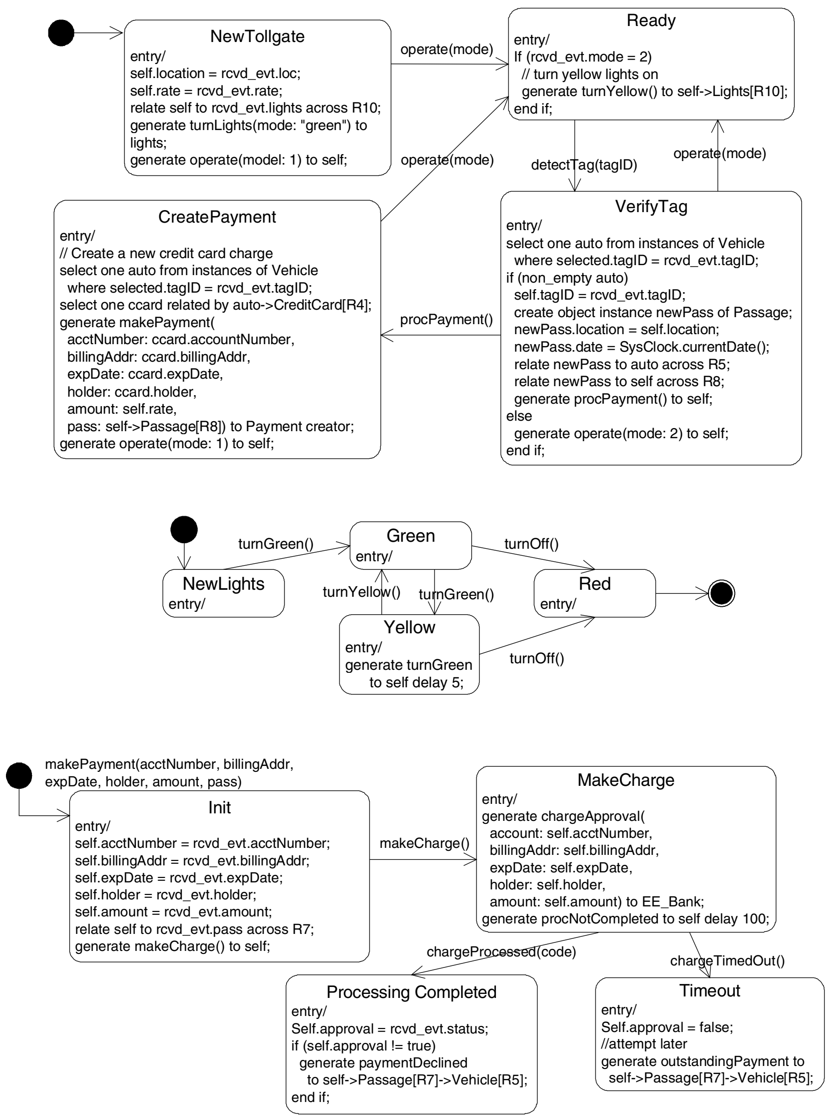
To summarize, in xUML, attributes fall into one of the following three categories: descriptive attributes are pertinent characteristic features of a class, identifying attributes are class identifiers, and referential attributes describe class associations.

Some associations require additional classes to capture data that do not properly belong to any of the participating classes (see Figure 5). An instance of an association class comes into existence when an association link between the two principal instances is created. For example, to assess the guests' satisfaction, we need an association class attached to R2. Class Satisfaction qualifies the relationship Guest–Room.

With each object, we associate an FSM model to specify its state-dependent behavior. In Figure 6, we show the FSMs for classes Tollgate, Lights, and Payment. The execution of xUML models differs fundamentally from sequential program execution. An executable model is a set of interacting FSMs. All object FSMs evolve concurrently, and so are the sequences of actions of their state procedures, unless constrained by synchronization. It is up to the developer to sequence the actions of the different FSMs and ensure object data consistency.

An FSM synchronizes its behavior with another one by sending a signal that is interpreted by the receiver as an event. The order of the exchanged signals between sender and receiver FSM pairs is preserved. Upon receipt of an event, a state machine fires a transition, arrives at the next state, and then executes the procedure associated with the new state. The state procedure is a set of statements in some action language that must run to completion before the next

Figure 6. Communicating object FSMs



```
NewTollgate
entry/
self.location = rcvd_evt.loc;
self.rate = rcvd_evt.rate;
relate self to rcvd_evt.lights across R10;
generate turnLights(mode: "green") to
lights;
generate operate(model: 1) to self;
```

```
Ready
entry/
If (rcvd_evt.mode = 2)
  // turn yellow lights on
  generate turnYellow() to self->Lights[R10];
end if;
```

```
CreatePayment
entry/
// Create a new credit card charge
select one auto from instances of Vehicle
  where selected.tagID = rcvd_evt.tagID;
select one ccard related by auto->CreditCard[R4];
generate makePayment(
  acctNumber: ccard.accountNumber,
  billingAddr: ccard.billingAddr,
  expDate: ccard.expDate,
  holder: ccard.holder,
  amount: self.rate,
  pass: self->Passage[R8]) to Payment creator;
generate operate(mode: 1) to self;
```

```
VerifyTag
entry/
select one auto from instances of Vehicle
  where selected.tagID = rcvd_evt.tagID;
if (non_empty auto)
  self.tagID = rcvd_evt.tagID;
  create object instance newPass of Passage;
  newPass.location = self.location;
  newPass.date = SysClock.currentDate();
  relate newPass to auto across R5;
  relate newPass to self across R8;
  generate procPayment() to self;
else
  generate operate(mode: 2) to self;
end if;
```

```
Green
entry/
```

```
NewLights
entry/
```

```
Red
entry/
```

```
Yellow
entry/
generate turnGreen
    to self delay 5;
```

```
Init
entry/
self.acctNumber = rcvd_evt.acctNumber;
self.billingAddr = rcvd_evt.billingAddr;
self.expDate = rcvd_evt.expDate;
self.holder = rcvd_evt.holder;
self.amount = rcvd_evt.amount;
relate self to rcvd_evt.pass across R7;
generate makeCharge() to self;
```

```
MakeCharge
entry/
generate chargeApproval(
  account: self.acctNumber,
  billingAddr: self.billingAddr,
  expDate: self.expDate,
  holder: self.holder,
  amount: self.amount) to EE_Bank;
generate procNotCompleted to self delay 100;
```

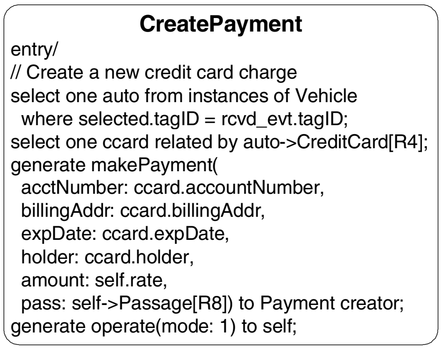
```
Processing Completed
entry/
Self.approval = rcvd_evt.status;
if (self.approval != true)
  generate paymentDeclined
    to self->Passage[R7]->Vehicle[R5];
end if;
```

```
Timeout
entry/
Self.approval = false;
//attempt later
generate outstandingPayment to
  self->Passage[R7]->Vehicle[R5];
```

event is processed. The statements are executed sequentially, as dictated by any control logic constructs. If the received event is not attached to one of the outgoing transitions of the current state, the event is simply ignored (events are not buffered).

An action language provides four types of actions: data access, event generation, test, and transformation. It supports these through control logic, access to the data in the class diagram, access to the data supplied by events

Figure 7. A State procedure is a sequence of actions



```
CreatePayment
entry/
// Create a new credit card charge
select one auto from instances of Vehicle
  where selected.tagID = rcvd_evt.tagID;
select one ccard related by auto->CreditCard[R4];
generate makePayment(
  acctNumber: ccard.accountNumber,
  billingAddr: ccard.billingAddr,
  expDate: ccard.expDate,
  holder: ccard.holder,
  amount: self.rate,
  pass: self->Passage[R8]) to Payment creator;
generate operate(mode: 1) to self;
```

triggering transitions, and access to timers. Even though action languages have assignments, conditionals, and loops, they should not be viewed as programming languages, but rather as model manipulation languages. The purpose of the actions is to perform tasks on modeling elements appearing on the diagrams of an executable model. Examples of model manipulation actions include navigating class associations; setting the values of attributes; generating signals to FSMs (including self); and creating, deleting and relating objects. In Figure 7, we present the state procedure associated with state CreatePayment in the FSM of class Tollgate. First, the state procedure identifies the vehicle related to the tag detected by the tollgate's sensor.

```
select one auto from instances of Vehicle
 where selected tagID == rcvd_evt.tadID;
```

Then, it retrieves the credit card charged for the vehicle's bill.

```
select one ccard related by auto->CreditCard[R4];
```

Next, the state procedure generates a signal (with all the necessary information in the signal's parameters) to the Payment class constructor to create a new Payment instance. And finally, the procedure generates a signal to its own FSM to fire its transition to state Ready.

```
generate operate(mode:1) to self;
```

From a given state, an FSM determines which transition to fire by consulting that state's transition table. The transition table is a list of events, and "the next" states resulting from these events.

Though the actions are executable, they are free of platform-dependent detail, such as data structures, thus providing support for model simulation and, later on, for code generation.

Apart from state procedures, sequences of actions can be used to define constraints (though OCL is better suited for this purpose) and domain bridges.

Sequence diagrams visualize the domain dynamics by portraying the execution traces, resulting from the signals exchanged among the objects—that is, by the objects' FSMs. In xUML, sequence diagrams emphasize state change over a relative time axis. Since the different domain views must be consistent, each signal that appears on a sequence diagram must be defined as an event attached to at least one transition in the FSM of the object whose lifeline is targeted by the signal.

Testing

In MDA, due to the formal nature of executable models, software testing comes closer to hardware testing, and it is possible to use automatic test generators. A developer can ascertain that an executable model satisfies the use case specification by simulating the model through a discrete event simulation. The bad news is that concurrency and asynchronous communications yield non-reproducible timing errors, which can take forever to isolate with testing.

A model verifier tool simulates (discrete event simulation) an executable model by interpreting its actions. The model execution is driven by an event queue. Events generated externally or internally during model execution that have yet to be processed are placed on this queue. As model execution proceeds, events are de-queued and processed in order. The model continues to process events from the simulation event queue until the queue becomes empty or a deadlock occurs. The model execution results are recorded in a simulation trace, which can be examined by testers to evaluate the tests.

Translation and Code Generation

The executable model is run through a series of transformations preserving its semantic content. The output model (or code) resulting from the application of a mapping preserves the causal order of the actions in the input model(s). A transformation may only reduce the degree of concurrency.

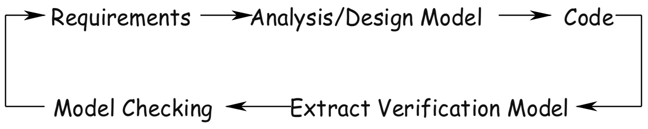
The last transformation step is code generation. The validated models are translated to source code for the target platform by a model compiler. The ability to translate an application model without modification to code for a variety of platforms is the essence of the MDA approach. At compile time, the model compiler adds the needed design detail, such as code implementing persistence or transaction management and allocation of computations to processors and threads.

Model Checking

Model checking verifies the properties of a system under all possible evolutions, and not just under scenarios defined by analysts. Early attempts to use aggressively formal methods in industrial software design have collapsed for economic reasons. There was no return on investment in formal verification. We bring up the issue of model checking because executable models are natural representations for model-based verification, which means higher yield on investment in formal methods.

As illustrated in Figure 8, the conventional approach to software verification is to extract a verification model from the program code and to perform model checking on this verification model. There are two serious problems with such techniques. First, the state space of a fully functional program is unbounded and far too complex for formal verification. Formal verification is feasible only when applied to an abstract model. Second, the business properties are the interest-

Figure 8. Conventional approach to software verification



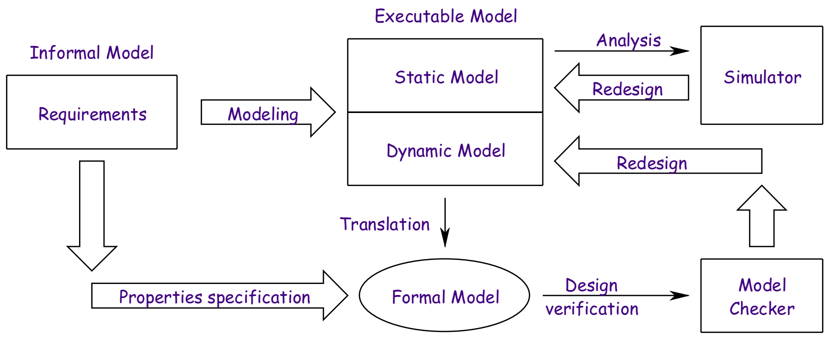
ing properties to check. However, the logic implementing the business properties is so deeply buried and scattered in the amorphous implementation code that it is practically lost.

In MDA, model checking can be performed on a system representation, which is automatically derived from the executable model. In addition, model checking can be conducted in terms of domain problem properties (see Figure 9). Sharygina et al. (2002) propose to conduct model checking by translating the validated xUML model to COSPAN (Hardin, Har'El, & Kurshan, 1996). COSPAN is an FSM-based model checking system.

In the methodology presented in Figure 9, modeling provides the construction of validated executable specifications, while model checking assures that the executable specifications meet the desirable requirements. Model checking checks that a given system satisfies the desired behavioral properties through exhaustive enumeration of all reachable states. When the design fails to satisfy a desired property, a counter-example is generated, which is used to identify the cause for the error.

The abstract executable models resolve the state space explosion problem typical of software systems. Strong name space containment, resulting from domain partitioning, is equivalent to hardware physical decomposition. For the first time, it is possible to apply successfully model checking on commercial scale software (Syriagina et al., 2002). If model checking becomes mainstream, it would have a tremendous effect on quality assurance and control.

Figure 9. Model checking in MDA



MDA Process

Model-driven development can be done with waterfall and iterative processes alike. Since executable models are operationally the same as executable code, Agile (Beck, 1999) principles and practices can be readily applied to the construction of executable models. Agile MDA (Mellor, 2004) is suitable for projects with unstable requirements, where it is more important for the process to be adaptive (agile) than to be predictive (planned).

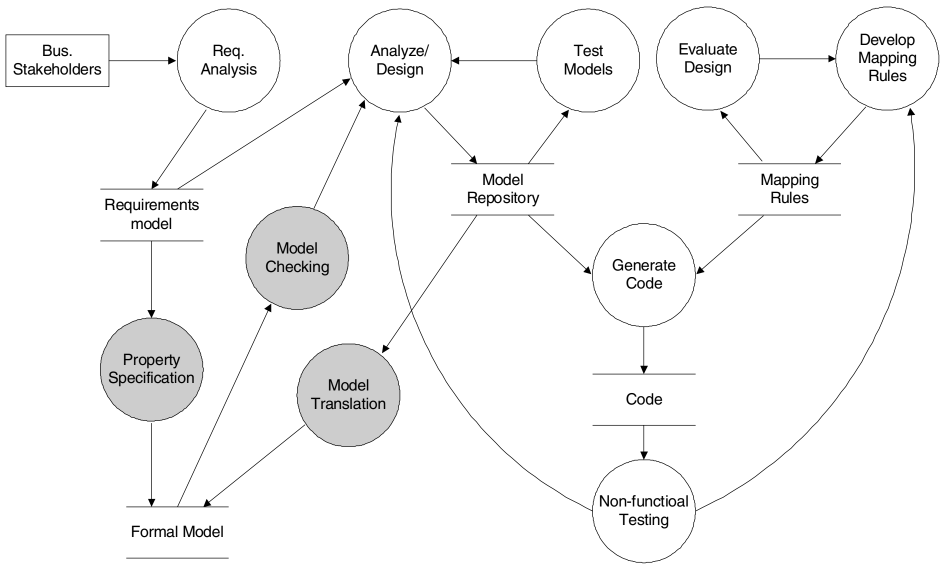
Operational Structure of the Process

To show the operational structure of an MDA process, we use a data flow diagram (DFD) model. In DFDs, bubbles represent activities, boxes represent external information sources/sinks, arrows represent data flows, and open-ended rectangles represent data stores.

The Requirements Analysis activity receives inputs from the business stakeholders and generates as output a requirements model. The requirements model, developed by the requirements analyst, consists of a use case model and a domain chart. From the use case model, the analysts develop the PIMs for each domain on the domain chart and store them in a model repository. Each domain is executed to test its behavior with test cases derived from the use case model. Testing the executable model may expose errors in the user requirements. The cycle Analyze/Design–Test continues until all requirements are realized and all detected errors are fixed. Concurrently with modeling, designers can develop, acquire, or update the mapping rules, and evaluate their correctness and performance. Once the executable models and the mapping rules are complete, the domains are woven together, after which the system code is generated.

In activity Non-Functional Testing, the tester needs to assess the following aspects of the implementation: timing behavior, performance, and interactions with external entities (what has been modeled as actors). Reliability and functionality have already been tested in the analysis activity Test Models. If model checking is conducted (the gray bubbles in Figure 9), the executable model is translated automatically to a form required by the selecting model checking package. The model translation activity gets its input from the model repository. The requirements analyst specifies the properties to be tested from

Figure 10. Data flow diagram for MDA workflow



the requirements model. The Model Checking activity verifies the behavior of the derived formal model against the specified properties under all possible evolutions of the model. The results are used by the analysts in the next Analyze/ Design–Test loop. The mappings knowledgebase can be implemented concurrently with the executable models. The cycle Develop Mapping Rules–Evaluate Design continues until all the mappings are defined and tested. During the Non-Functional Testing activity, the tester may expose errors in the executable models and the mappings. After the errors are fixed, the system code is generated and tested again.

Planning xUML Projects

Past statistical data shows that only one-third of all projects are considered completely successful in terms of meeting planned goals (e.g., see the Standish Group reports). This fact alone testifies to the difficulty of project planning. The major reason for poor scheduling is the innate inability to plan with accuracy under changing and evolving requirements, although other factors have a negative impact as well (e.g., planning for motivation).

The two main objectives of planning are to allocate tasks to developers and to provide a benchmark against which the progress of the project is measured. Planning is an ongoing activity throughout the project lifecycle. It is based on estimation, and unfortunately, estimation is hardly ever accurate. Collected historical data are used to inform planning. Based on this data, the project manager constantly adjusts the plan to reflect the new knowledge being acquired in the development process.

We present a useful technique for constructing estimates, called BERT, and another technique for improving the ability to estimate, called ERNIE (Douglas, 2004). With BERT, the work to be done is divided into atomic tasks of no more than 80 hours in duration, called estimable work units (EWUs). The manager constructs three estimates: 1) the mean (50%) estimate; 2) the optimistic (20%) estimate; and 3) the pessimistic (80%) estimate.

The mean estimate is the one that developers will beat half of the time. According to the central limit theorem, if all mean estimates are true 50% estimates, then the project will come in on time. The mean estimate does not take into account the risk associated with project activities and with their estimates. Many decisions in planning are driven by the desire to reduce risks. Risk can be modeled as a statistical variable with a distribution whose values represent the chances of EWU failure. In iterative approaches, the risk mitigation strategy boils down to immediate proactive steps to reduce risk probability or risk impact.

BERT uses the following risk management adjustment. The optimistic and pessimistic estimates are the times the developer will beat 20% and 80% of the time, respectively. The difference between the two estimates is the confidence the manager has in the mean estimate. The following formula is used to calculate the estimate for the task at hand,

$$(\mathit{Pessimistic} + 4 \times \mathit{Mean} + \mathit{Optimistic}) \;/\; 6 * E_c,$$

where E_c is the estimate confidence coefficient. The confidence coefficient is based on the manager's personal track record. The ideal value for this coefficient is 1.0, but its typical range is from 1.5 to 5.0. The manager aims at improving his/her ability to estimate over time—that is, to get his/her coefficient as close as possible to 1.0.

Managers use the ERNIE technique to calculate the current value for E_c from their past performance,

$$E_c^{n+1} = \Sigma(\text{deviations using } E_c^{\,n}) / (\#\text{ of estimates}) + 1.00$$

In an iterative process, project managers develop two kinds of plans: a coarse-grained phase plan and a fine-grained iteration plan. The phase plan is very concise and is produced very early. It defines the phases of the lifecycle, the phases' milestones, and the project's staffing profile. The iteration plan is produced using traditional planning techniques, like Ghant charts and PERT charts, based on the estimates and risks (the risk is inverse proportional to the confidence estimate) constructed with BERT and ERNIE. The iteration plan should take into account the expertise and availability of human resources. Kruchten (2000) visualizes the iteration plan as a sliding window through the phase plan. Agile methods, true to their lightweight nature, by and large ignore the phase plan or the milestones of the phase plan.

Roles in the MDA Process

A development method defines who is doing what, how, and when (i.e., a method defines roles, activities, artifacts, and partial order on activities).

Think of a role as an actor. One and the same developer, just like an actor, could play different roles at different times of the project lifecycle. A role is defined by a set of activities and responsibilities related to the artifacts being created or controlled (Kruchten, 2000). The roles in the MDA process are requirements analyst, analyst, architect, designer (programmer), tester, and maintainer (Mellor & Watson, 2004).

Requirements Analysts

In the MDA process, the purpose of requirements analysis is to elicit, model, and manage the user requirements. The requirements analyst may employ use case models, activity diagrams, sequence diagrams, or other models to accomplish the job. This phase remains by and large people driven. Because the artifacts produced in the subsequent analysis-design phase are executable (i.e., testable), customers have ample opportunities to provide feedback.

Analyst

The analyst transforms the informal system requirements to executable models. The analyst renders the application business logic into class diagrams, FSMs, and actions grouped into procedures. These three primary models can be complemented with sequence diagrams (also called execution traces) and communication diagrams.

Architect

The architect is in charge of the application's architecture. The architect coordinates the models and the mappings being developed or reused and makes sure they are compatible. Mellor envisions that as the market for model compilers matures, the primary activities in the architect's portfolio would be model acquisition, model integration, compiler selection with the concomitant performance tuning, and managing the application structure.

Designer

The designer is in charge of implementing or adapting the model compiler's archetypes. Archetypes define the rules for translating the application into a particular implementation. The programming language is determined by the QVT (Query, Views, Transformations), the knowledgebase for mapping rules. Even when writing the mappings, the designer is not involved in particular implementation details. His/her work is to create generic templates (e.g., a FIFO queue for objects), which at design time are instantiated to generate a particular implementation (e.g., a FIFO queue of driver's payments).

The relationship architect–designer is that of policy–mechanism. The architect decides on the policy for model transformations, while the designer implements the mechanism by programming the archetypes. The work of the designer is to encapsulate knowledge in micro-level abstractions and using his/her implementation platform expertise to fine-tune the application's performance.

A similar policy–mechanism relationship exists between architect and analyst.

Tester

The purpose of testing is to assess the quality of the executable models. Quality has two aspects, which the tester has to address. The aspects are absence of defects and fitness for a purpose. The tester performs two types of testing, corresponding to unit testing and integration testing in code-driven design. First, executable models are tested with a simulator to verify that they exhibit the required functionality, and then sets of domain models are tested together to verify the bridge mappings. Testing is of course use case driven. Test-generation tools are available in most development environments. Such a tool can quickly produce a large set of tests from a model, and then execute them against this model.

Maintainers

The maintenance phase of a product begins upon completing the product acceptance test and continues until the product is phased out. With conventional software products, maintenance only delays the inevitable obsolescence (in 3-5 years). Over time, the product's complexity grows until the maintenance cost surpasses the production cost, at which point further maintenance is blocked by overwhelming complexity (Bettin, 2004).

In MDA, the role of the maintainer is quite different. The maintainer incrementally builds software assets, which can be reused across a large number of enterprise applications over a long period of time. When a request for change is approved, either a model (functional change) or a mapping (platform change) is modified, but never the final software product. The code for the modified system is generated after the model or mapping update. The job of the maintainer bears similarities to those of the analyst and the designer.

The higher level of abstraction of the maintenance activity in MDA, and the fact that executable models are aligned with the enterprise business process and strategy, turns software products based on executable models from liabilities, depreciating over time, to software assets incrementally acquiring ever greater value (Bettin, 2004).

Conclusion

The adoption of UML as an OMG standard in 1997 changed the way software is produced. Model transformation added the much sought-after rigor to OO software development, and culminated in MDA and xUML. In this chapter, we introduced the principles of the model-driven approach to software design, and we showed how to construct systems out of executable models, using as a running example a system of moderate complexity. We defined the new roles and responsibilities of developers, and presented the activity workflow with the MDA approach. We showed the operational structure of the MDA process, and discussed how to plan and estimate the progress of Agile MDA projects.

In Chapter IV, we assess the opportunities and challenges MDA presents to software firms. We analyze how this new technology changes the software design space and project planning. We also give guidelines aiding developers and software product managers in the transition to model-driven development with xUML.

References

Beck, K. (1999). *Extreme programming explained: Embrace change*. Reading, MA: Addison-Wesley.

Bettin, J. (2004). Model-driven software development: An emerging paradigm for industrialized software asset development. Retrieved from *http://www.softmetaware.com*

Bettin, J. (2005). Managing complexity with MDSD. In B. Roussev (Ed.), *Management of the object-oriented software development process.* Hershey, PA: Idea Group Inc.

Chen, P. (1977). *The entity-relationship approach to logical data base design.* Wellesley, MA: Q.E.D. Information Sciences.

Codd, A.F. (1970). A relational model for large shared data banks. *Communications of the ACM*, *13*(6).

Czarnecki, K., & Eisenecker, U. (2000). *Generative programming: Methods, tools, and applications*. Reading, MA: Addison-Wesley.

Douglas, B. (2004). *Real time UML* (3rd ed.). Boston: Addison-Wesley.

Hardin R., Har'El, Z., & Kurshan, R. (1996). COSPAN. *Proceedings of CAV'96* (LNCS 1102, pp. 423-427).

Henderson-Sellers, B. (1996). *Object-oriented metrics, measures of complexity.* Englewood Cliffs, NJ: Prentice-Hall.

Kiczales, G. (1996). Aspect-oriented programming, *Computing Surveys*, *28*(4), 154.

Kruchten, P. (2000). *The rational unified process®: An introduction*. Reading, MA: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.

MDA. (2004). OMG Model-Driven Architecture. Retrieved from *www.omg.org/mda*

Mellor, S.J. (2004). Agile MDA. Retrieved from *http://www.omg.org/agile*

Mellor, S.J., Scott, K., Uhl, A., & Weise, D. (2004). *MDA distilled*. Boston: Addison-Wesley.

Mellor, S.J., & Balcer, M.J. (2002). *Executable UML: A foundation for Model-Driven Architecture.* Boston: Addison-Wesley Professional.

Mellor, S.J., & Watson, A. (2004). Roles in the MDA process. Retrieved from *www.omg.org/mda*

PL. (2004). SEI collection of resources on product lines. Retrieved from *www.sei.cmu.edu*

Shlaer, S., & Mellor, S.J. (1988). *Object-oriented systems analysis: Modeling the world in data*. Englewood Cliffs, NJ: Prentice-Hall.

Shlaer, S., & Mellor, S.J. (1992). *Object lifecycles: Modeling the world in states*. Englewood Cliffs, NJ: Prentice-Hall.

UML AS. (2001). UML actions semantics. Retrieved from *www.omg.org*

Endnote

1 Little help might be available, e.g., a modeling tool automatically generating class templates or supporting some kind of round-trips.

Chapter III

Management Planning in a Changing Development Environment

Melissa L. Russ
Luminary Software and Space Telescope Science Institute, USA

John D. McGregor
Luminary Software and Clemson University, USA

Abstract

Technologies such as aspect-oriented and generative programming bring new capabilities to object-oriented software development. They do not simply replace existing techniques; they change the shape of the development environment. Development is a multi-dimensional landscape of organizational patterns such as software product lines, of meta-information such as build scripts and templates, and of technologies such as model-driven development. Traditional patterns of development management are not sufficient to effectively manage development in this emerging context. In this chapter we provide an overview of some existing

and emerging elements, describe how they affect development, and present a management framework for planning software development.

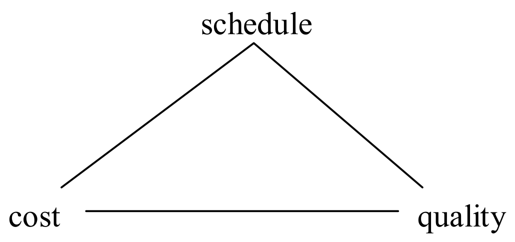
Introduction

Every manager knows the fundamental relationship shown in Figure 1. To increase quality, you must increase schedule or cost or both. Exactly how much you must increase the cost or the schedule depends upon a number of factors such as the technologies being used to construct the software and the scheduling techniques being used to manage it. For example, if the schedule is optimized, it can no longer be traded off for improved quality. It also depends upon the process that defines how the software will be constructed.

An effective process helps humans carry out an activity correctly and efficiently, producing quality outputs. It is particularly useful for those inexperienced in that particular activity. The process describes a proven, repeatable approach. It describes what to do, what to use to do it, and when to do it. As new corporate goals are set or new tools and techniques are developed, processes are modified to remain useful and realistic.

A number of software development processes have been defined by individual projects and consulting companies. The Software Engineering Institute (SEI) even has a framework for measuring how well an organization defines its processes (Paulk, Curtis, Chrissis, & Weber, 1993). Often the process definitions are easy to read and understand, but less than effective when put into practice. Many definitions make so many simplifying assumptions that they do

Figure 1. Project management trade-offs



not provide useful guidance in actual situations. We will discuss techniques for planning that will make your current processes more effective.

The software development landscape becomes more expansive daily as new technologies, new methods, and new strategies emerge, while existing technologies, methods, and strategies remain in use. Organizational and technical managers have an increasing number of options from which to choose. There is no clear mapping between these new elements and existing processes that shows how a process must change when new elements are incorporated into an organization's portfolio. The purpose of this chapter is to help managers define that mapping.

Software is a strategically significant factor in an increasing number of products and hence of increasing strategic significance to the organization. We will focus on those development organizations that construct strategically significant, software-intensive products. The achievement of an organization's strategic objectives is dependent upon the reliability of the software portion of the organization's supply chain. Many events can disrupt the supply chain such as the software being delivered with poor quality or being delivered in an untimely manner. While exact figures differ, reports consistently indicate that less than half of the software developed is completed on time and is of acceptable quality. The approach described in this chapter can, we believe, increase this percentage.

Our thesis is: the success of a software development effort depends upon the combination of development strategies, models and methods, and technologies that an organization selects. When software becomes an integral part of corporate strategy, and technologies are selected to achieve specific goals, a standardized, one-size-fits-all approach to development is not adequate to faithfully represent actual development practice. Development models of a few years ago are not effective at capturing the dynamism and concurrency of emerging development technologies. The approach provided in this chapter is intended to be modular and configurable. This allows the construction of effective plans and processes that reflect the realities of the development environment for a particular organization.

The remainder of this chapter is structured as follows. First we survey the software development landscape and consider some trends and new ideas. Then we show how to use that information to define effective plans and processes. Finally we present a case study that illustrates a number of the points.

Landscape

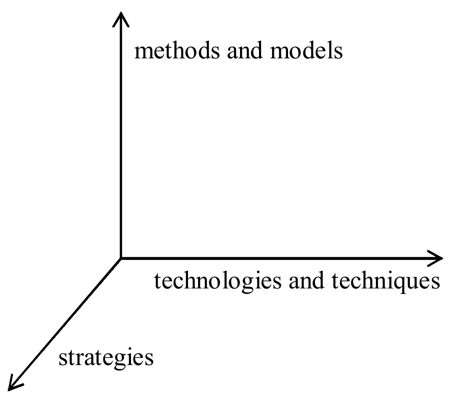
The landscape we are addressing is vast and not easily categorized. To allow for flexible classifications, we will use the concept of "concerns," as used by Tarr, Ossher, Harrison, and Sutton (1999). Concerns are not orthogonal nor are they mutually exclusive. They are a way of grouping ideas that address a common issue or "concern." We will separate concerns so that we can more easily discuss the variations between types of techniques and within a set of related techniques.

We divide the software development landscape into three concerns: technologies and techniques, models and methods, and strategies, as shown in Figure 2. A software development method is a specific ordering of tasks that employ technologies in order to implement strategies. We consider each separately, including characteristics of each that influence decisions about the other.

Technologies and Techniques

Technologies are the fundamental elements in a development effort and the most visible selection that a project manager makes. The Java language is a technology. Technologies are usually accessed through tools. The Java compiler, the Java debugger, and the Java class documentation tool are some of the tools that make the Java technology accessible to users.

Figure 2. Three dimensions of plan definitions



Most software-intensive products are built with a variety of technologies. For example, the executable code of an Eclipse plug-in may be written in Java, but its accompanying configuration file is written in XML (Eclipse, 2004). That same plug-in may generate Java Server Page (JSP) templates that are instantiated in an HTML browser.

Each technology is chosen for a specific job based on its characteristics (or its publicity). In some cases a technology is chosen after an engineering study shows it to be the best fit to the needs of the organization or project. All too often however, a technology is chosen because it is popular at the moment or because it is what was used in the last project. These decisions are sometimes made without sufficient consideration of the impact on cost, schedule, and quality.

Technologies bring with them techniques for carrying out development activities. Changing programming languages, for instance to Java from C, change certain patterns of programming. Changing from one technology to another incurs a loss in productivity as personnel adjust to the new patterns. The cost of adopting a new technology should be balanced by benefits within the scope of the project or in the context of some development strategy.

Technology Concerns

In this section we decompose the technologies and techniques concern into several concerns, including structures, notations, levels of abstraction, and transformations.

Fundamental Structures

Behavior and data are grouped together to facilitate the design and construction of software products. There are several techniques for accomplishing this, including:

- Functions and independent data structures
- Objects
- Components
- Aspects

The Java language uses objects as its fundamental structuring element. AspectJ, an extension to Java, uses an aspect as the fundamental structure augmented by the Java objects. The implications of choosing a particular structuring approach can be significant. For example, when many companies began using objects as the structuring element, it soon became apparent that this led to a design technique that focused first on concepts in the domain and then implementation-specific concepts. It also led to a more abstract approach, which worked best with up-front design modeling. The change to objects resulted in changes to the existing development processes and to the skills needed by developers. Early adopters of new approaches, such as aspects, often must discover the new process patterns to accommodate the new structures. For example, the crosscutting nature of aspects is giving more attention to the infrastructure areas of software systems such as security and persistence. The trend is toward more complex structures that require a longer learning curve initially.

Notation

Programming languages and modeling languages are needed in every development effort. In some development efforts, information is captured in several different notations at different times in the process. The manager needs to achieve a balance between using one very general notation, which can be used across the entire process, but may not adequately represent the development information and using several different notations, each of which expresses the information of a particular development phase precisely, but results in translation errors as one type of information is transformed into another. The trend is toward more comprehensive, general-purpose modeling notations accompanied by specialized extensions for specific domains (e.g. the Unified Modeling Language, or UML). The formal semantics of UML should reduce translation errors between stages in the development process.

Levels of Abstraction

A technique operates on information that is to some degree abstract. Source code is compiled into object code. Templates are instantiated into source code, which is then compiled into object code. Model types are often defined by metamodels. The trend in software development currently is toward the developer working with higher levels of abstraction and using tools to produce concrete products from those abstractions. This, in turn, requires the developer be able to use more powerful notations and possess higher-order analytic skills.

Transformations

Transformation of one type of information into another type or from one syntax into another is a common occurrence. Source code is transformed into object code. The Report Program Generator (RPG) transformed report specifications into standard reports. CASE tools routinely transform UML diagrams into the user's choice of source-level programming languages. The release of UML2.0 is supporting more complete transformations and new development strategies such as model-driven development (Soley, 2000). This allows the development organization to concentrate more on their core business and spend less time on implementation details. The trend is toward safer, but more generic transformation technologies.

Methods and Models

A software development method defines the set of activities that an organization uses to create a piece of software. For example, an agile software development method defines steps for developing test cases, writing code, and executing test cases, in that order (Beck, 2002). Different methods define different sets of activities and different orderings of those activities. Traditional development methods define the same activities as the agile method, but define the ordering as: writing code, writing test cases, and executing test cases.

Because a method defines activities, it is often referred to as a process. In this chapter we want to make a distinction between a method and a process. We classify several processes, the Rational Unified Process® for one, as methods because they go beyond the normal bounds of a process definition to specify the notations and models that should be used. Process, which we treat later, is limited to describing a flow of activities and information.

In addition to defining activities, a development method defines how to model the software that is to be built in order to reduce the complexity of managing the problem. This includes notations and types of diagrams that provide a comprehensive means of representing the intended software. The more mathematically rigorous the model, the more useful the diagrams to the development team, although the team will have taken longer to prepare the diagrams.

Methods

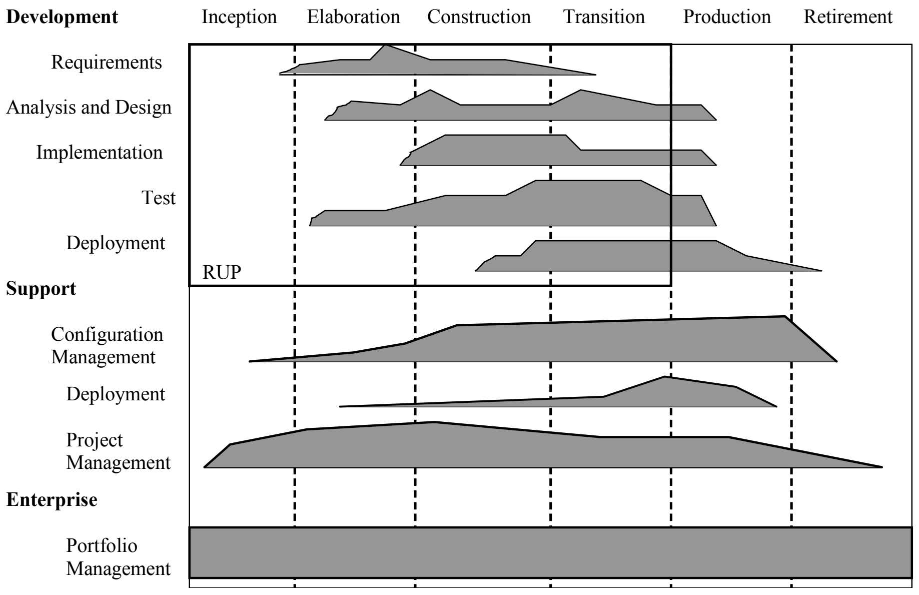
Here we give an overview of a few of the more popular methods.

Rational's Unified Process® (RUP) (Kruchten, 2000) became one of the most widely used methods for object-oriented software development. It is based on a hybrid iterative, incremental process model. The process covers the basic software development process, shown in the shaded area of Figure 3. The representation of the process is a matrix in which the basic steps of the software development process form the vertical axis. The horizontal axis covers four broad stages, shown as the leftmost four items on the horizontal axis in Figure 3.

Enterprise Unified Process is an extension of RUP developed by Ambler and Nalbone (2004). The extension is shown in Figure 3. They added two additional stages: Production and Retirement. They also extended beyond basic development activities to include the supporting activities and the broader enterprise activities.

The shaded shapes in Figure 3 represent the amount of effort being exerted on the particular activity at any point in the process.

Figure 3. Enterprise Unified Process



Agile is the most recent addition and is the most flexible of all the methods discussed here. As the name implies, the intention is to be able to respond quickly to changes in knowledge or customer requirements. Agile methods, such as Extreme Programming, use an incremental style of development (Beck, 1999) to direct the work of very small teams, often only two people. Products are designed by collective discussion among the teams. Each team is then assigned responsibility for implementing very small pieces of product functionality. The agile approach contradicts the trend mentioned earlier of more formality. There is more emphasis on human-to-human communication than on a formal, comprehensive design prior to the start of coding.

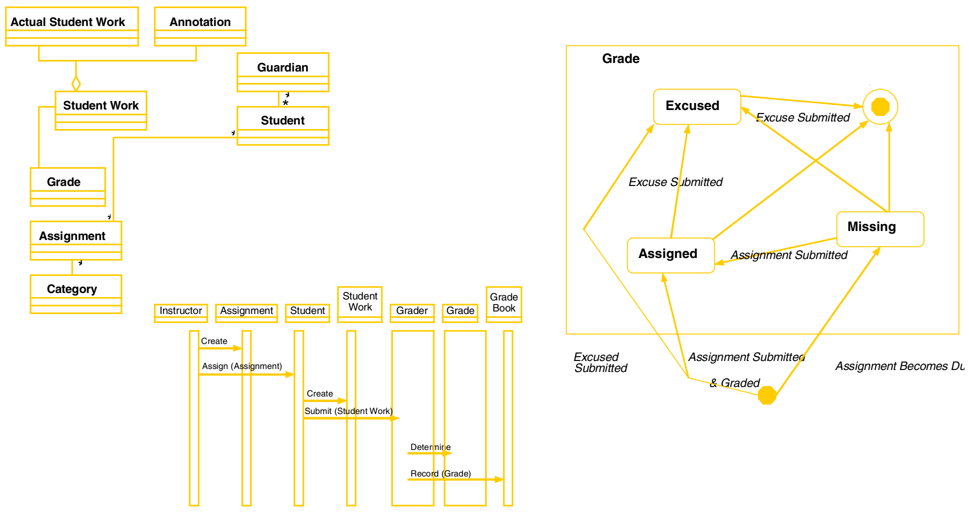
Rapid Application Development (RAD) was used for many years, particularly in the 1980s and early1990s, as a method for developing "cookie cutter" applications. That is, applications that followed very closely a particular design pattern could be produced rapidly because of specialized tools. The primary example is the development of client-server applications where a simple client accessed a database to manipulate or add data to the database. Many call centers and customer contact systems were setup using this basic approach. The developer uses a tool that walks him or her through a sequence of actions that result in a completed application. There are similar tools now for applications that use the J2EE architecture. Some of the tools start with a wizard that sequences the developer through specifying key elements of the architecture pattern.

Team Software Process (TSP) and Personal Software Process (PSP) are methods limited in scope to a development team and an individual developer respectively (Humphrey, 2002). These processes take a very different view of development from the previous processes we have discussed. These processes provide only a high-level view of what development actions the developers take. They provide a very detailed view of how the developers track their work. They stress the collection of productivity and accuracy data, and the use of that data to improve performance.

Models

The development method defines a specific modeling notation(s) to be used and the models that are built using that notation. While several notations exist, the Unified Modeling Language is widely used for representing development information. UML is used to represent analysis information in the form of use

Figure 4. UML diagrams



cases and design information in terms of a number of diagrams, some of which are shown in Figure 4. UML even defines an extension specifically for testing.

Analysis and design models provide visibility into the structures and definitions used to create the software, but additional models are needed for additional facets of development. Graphical views of version control hierarchies and the roadmap of product releases would allow developers to more quickly and accurately identify where they need to work or where inconsistencies may exist.

Models and Methods Concerns

In this section we separate the models and methods concern into three more specific concerns: organization, granularity, and scope.

- **Organization:** Every method, and the processes we have identified as methods, defines relationships among development activities. These relationships sequence the activities. It may be a partial ordering allowing concurrency of activities. The trend is toward more concurrency.
- **Granularity:** The number of fundamental activities encapsulated within a step in the process varies from one process to another. "Design" may be

a step in a process, while another process has three steps: "architecture design," "class design," and "application design." The broader the definition of a step, the less guidance the process gives the developer. The trend is to view a method definition in a hierarchical manner. Different viewers see different granularities of activity definitions. A manager may only see three activities in the method definition, while a developer might see those three broken into nine or ten more specific definitions.

- **Scope:** The traditional "development" method usually includes only the technical steps that directly affect the creation of source code. More recent methods encompass the broader issues of product planning or even the planning of multiple products. The trend is to be more inclusive, and more realistic, in identifying the enterprise activities that are necessary for development methods to be successful.

Strategies

A product development strategy is a set of guiding principles and high-level objectives that are defined to achieve specific goals. The software product line development strategy (Clements & Northrop, 2002) achieves large gains in productivity and reduces the time to get a product to market. Similarly, outsourcing is a strategy that achieves reduced production costs, but sometimes with increased time to get a product to market. Having a documented, formal product development strategy (Butler, 2003) facilitates aligning development projects with organizational goals. A development strategy defines broad directions for certain activities, such as division of responsibilities and determining the scope of the analysis and design activities. A strategy typically does not mandate a specific method or technologies, but there are definite interactions. For example, outsourcing requires the use of a more formal design notation than development involving co-located teams.

Below, we very briefly describe several recent strategies.

- **Software Product Line:** This strategy (Clements & Northrop, 2002) increases the productivity of personnel and reduces the time required to bring a product to market. A software product line is a group of related products that are planned and managed as a single unit. The common functionality among the products is identified and leveraged to achieve strategically significant levels of reuse of software. A product line may achieve 85-95% reuse with careful planning. The strategy encompasses both management and technical aspects of software product development.

- **Outsourcing:** This strategy involves using resources outside the organization to accomplish tasks (Clarke, 2004). The current popular trend is to outsource tasks to organizations in countries where costs are drastically lower than where the outsourcing organization is located. This is not a new idea. The United States government has long split tasks where government employees defined requirements for products and outside contractors develop the products. This has led to a community referred to as the "acquisition" community.
- **Open Source:** This often-misunderstood strategy provides public access to the source code an organization produces. What is misunderstood is that this does not mean that the source code is free for any use. This strategy rests on the belief that others will contribute new ideas and perhaps some code so that the contributing organizations will receive benefit from participating in the development of a common tool or standard technique. The Eclipse project, sponsored primarily by IBM, is a current example (Eclipse, 2004).
- **Enterprise Architecture:** Enterprise architecture is a strategy for aligning the information technology actions of various groups spread across the entire enterprise. Such an architecture defines the structure for all of the components of an organization's computation resources. It makes choices for the entire organization about certain key technologies and techniques. The goal of an enterprise architecture is a coherent set of technologies and techniques for the enterprise, not just a single business unit. Organizations that utilize this strategy impose an additional layer of constraint on development and acquisition projects. The project team is not allowed to make decisions or choices that violate this high-level structure.

Strategy Concerns

We separate the Strategy concern into three more specific concerns: goals, scope, and abstraction.

- **Goals:** A strategy has a set of goals that it is designed to achieve. There should be a clear mapping between the goals of the strategy and the actions that implement the strategy. The trend is toward supporting a more diverse set of goals including hard-to-quantify goals such as "improve customer satisfaction."

- **Scope:** A strategy has a "sphere of influence," those activities and decisions for which it provides direction and constraint. The software product line strategy has a wide scope of influence. It influences both business and technical decisions concerning software-intensive product design. For example, the team assigned to produce one of the products in a product line cannot make design decisions that contradict the overall product line architecture. The trend is to be more encompassing as executive and technical managers recognize the need for coordination of corporate goals and development goals.
- **Abstraction:** A strategy is a high-level philosophy that does not constrain specifics, but does impose broad directions. A strategy may be implemented in many ways. Some ways will be more effective than others. The trend is toward strategies that abstract development issues and concerns to a level where they can be reasoned about in terms of the business goals.

Plans and Processes

Managers define project plans and processes for each development effort. Our thesis stated that the effectiveness of these plans and processes is related to choices that are made. In the previous section we illustrated that the development landscape is multi-dimensional. There are relationships and constraints among those dimensions. A project plan, which does not properly resolve these constraints, cannot lead to a successful result. In this section we identify the characteristics of plans and processes that will be affected by the technologies, methods, and strategies of the development effort.

Process Models

A process model gives an overall "shape" to the specific process that will be used for a development effort. It takes a few essential assumptions and guides the development of specific development processes. First we briefly define a few process models, and then we discuss how these are used in defining a process.

The fundamental shape of the process derives from its philosophical basis. In this section we discuss four fundamental process models.

- **Waterfall:** The waterfall model makes the basic assumption that each major activity in the process uses information from the previous activity as input, but there is no interaction between the activities. The phrase often used is "throwing it over the wall" to the next activity. Mistakes made in an activity are fixed by the recipients of the output rather than the persons who made the original mistake. This assumption is realistic in certain situations, such as a government software development where a government analyst develops a set of requirements and hands those off to a contractor to actually create the software. Some large corporations have structured their departments to have business analysts in one structure and software developers in another structure. The waterfall model was even codified in a U.S. military standard, MIL STD2167A (DoD, 1988).
- **Spiral:** The spiral approach defined by Boehm (1988) did two things differently from the waterfall model. First, it indicated, by the spiral shape, that activities that were performed early in the process would be revisited later. Second, it expanded the scope of the activities defined as part of the software development process. The revisiting of an activity provides experts in an area such as requirements or architecture with the opportunity to fix mistakes they made, but the model limits when this can happen.

  The spiral model was adopted by many organizations because of the more realistic view of handling mistakes. However, the apparent limitation of when errors could be corrected made the process less realistic than the iterative model discussed next. The spiral model also implied a different organization than did the waterfall, with more interaction among the various management and technical groups.
- **Iterative:** The iterative model provides more flexibility than the spiral model, but often with a smaller scope. An iteration is a pass through a set of development activities. One iteration might only pass through analysis and initial design before ending, while the next might go as far as architectural design. Iterations may be planned to end at a certain activity, or they may be ended by the discovery of a major problem with information developed earlier in the iteration.

  The iterative model provides a means of representing exploratory studies early in a project when the team may develop a set of requirements and explore the initial design of a solution to support those requirements. The iterations are more flexible in terms of content and duration than a pass around the spiral.

- **Incremental:** The incremental model is most often used in conjunction with the iterative model but can be used with any other model. This is an alternative means of slicing the project for work packets. An increment is some subset of the total product functionality. An increment might be a "horizontal" slice involving the development of the user interface, or it might be a "vertical" slice running from a subset of the interface through the business logic to a subset of the database.

  An increment provides a "divide and conquer" strategy for addressing complex products. When combined with the iterative model, an iterative, incremental process can be formed that encourages concurrent exploration of high-risk functionality.

Project Plans

A project plan is an instantiation of a development process. It defines who will operate the development process, when the phases in the process will be executed, and the risks associated with each of the activities. The plan provides justification for the development effort, the strategy to be followed, and estimates of the resources needed for the development of products. These functions may be accomplished in separate documents, but we will think of them as a single plan.

The technologies, methods, and strategies selected for a project heavily influence the plan. The software product line strategy invokes an investment approach that requires careful justification based on estimates of costs and benefits. One approach to this is described in Boeckle, Clements, McGregor, Muthig, and Schmid (2004). Outsourcing strategies also have to be justified, usually by discussions about liability and ownership issues. The project plan describes how the strategy will be implemented in the context of achieving the goals of the enterprise.

The development plan must justify each of the technologies chosen for producing products. The programming language may be selected based on the level of developer skill, the level of automated type checking desired, the presence or absence of a virtual machine, or the range of platforms on which the product must run. The justification is usually based on the attributes of the technology such as binding time or rigor of checking during compilation.

The development plan explains and details the development method chosen for producing products. The method may be an enterprise-mandated method, or it may have been selected for specific reasons such as a need for discovery of requirements.

In each of these sections of the plan, the goals of the development effort should play a prominent role. Project plans are often neglected or ignored. This is usually because they do not perform an essential function. Making the plan goal oriented can make it a must-read by newly assigned personnel, development partners, and upper-level management.

Relating Technologies, Methods, and Strategies to Plans and Processes

The job of managers who have some level of responsibility for software development is to define a set of plans and processes that will result in effective, efficient production of products. When preparing for a new project, the manager either adopts the process definitions used in a previous project or decides to define new ones. All too often the decision is to continue to use existing processes. This may work if the existing processes were effective, the goals of the enterprise have not changed, and if no changes are made in the development environment for the new project. In this section we use the landscape survey to talk about emerging trends and how to adapt to them. The number of variables is large, and we do not pretend to be able to cover all of them. We will present a case study that illustrates the use of the tables.

Any new process definition is constrained by the previous investment and the culture and tradition of the organization. We are not free to change everything about a process at the same time. By focusing on the changes in technologies, methods, and strategies that are anticipated, the changes to processes can be confined to those that can be directly justified. These changes can be analyzed in terms of their impact on the activities and artifacts of the process and the impact mitigated with appropriate support.

We are now concerned with how our original three concerns—technologies and techniques, models and methods, and strategies—interact as a manager makes decisions about the development environment. Due to space limitations

Table 1. Technology concerns vs. model concerns

	Scope	Granularity	Organization
Structure	Does the chosen model notation completely represent the chosen structure?	Does the chosen model notation at least reach the level of granularity of the chosen structure?	Does the chosen model notation organize information about the chosen structure in an intuitive manner?
Notation	Is the chosen model notation compatible with the other notations within the scope of the project?	Is the granularity of the model compatible with the notation of the rest of the project?	Can the notation accompanying the technology express the organization of the model?
Transformations	Does the chosen model notation support transformations that map every model element to targets?	Is the granularity of the model compatible with the level of transformation available in the technology?	Do the transformations of the technology operate with input/output pairs that reflect the organization of the model?
Levels of Abstraction	Is the scope of interest sufficiently broad to encompass the levels of abstraction?	Does the granularity of the models and methods match the levels of abstraction of the technology?	Is the organization of the models compatible with the levels of abstraction in the technology?

Table 2. Model concerns vs. strategy concerns

	Goals	Scope	Abstractions
Scope	Does the model scope cover all concerns of the strategy?	Does the scope of the models match the scope of the strategy?	Does the model scope encompass the abstractions of importance to the strategy?
Granularity	Are the goals of the strategy stated in terms that the granularity of the model can handle?	Is the scope of the strategy covered by the model?	Do some of the abstractions in the strategy match the fundamental units of the model?
Organization	Does the organization of the model facilitate the goals of the strategy?	Can the model be organized to cover the scope of the strategy?	Does the organization of the model make sense with the set of abstractions in the strategy?

we cannot discuss every interaction in detail. In Table 1, Table 2, and Table 3, we provide a series of questions intended to illustrate the relationships among the various concerns. The manager should use these questions to stimulate analysis before a decision is made.

Table 3. Strategy concerns vs. technology concerns

	Structure	Notation	Transformation	Levels of Abstraction
Goals	Does one of the chosen technologies provide structures that represent the goals of the strategy?	Does one of the chosen technologies provide notation capable of expressing the concepts in the strategy's goals?	Does one of the chosen technologies provide transformations that result in goal-level artifacts?	Do the levels of abstraction in the technology contribute to achieving the goals of the strategy?
Scope	Do the structures in the chosen technology cover the scope of the strategy?	Are the notations in the technologies capable of expressing the complete scope of the strategy?	Are the results of a transformation still within the scope of the strategy?	Do the levels of abstraction in the technology fit the scope of the strategy?
Abstractions	Are the abstractions defined in the strategy compatible with the structures defined in the technologies?	Can the notations in the technologies express the abstractions in the strategies?	Can the transformations in the technologies operate on the abstractions in the strategies?	Can the levels of abstraction in the technology represent the abstractions in the strategy?

Case Study

We use as our case study the Arcade Game Maker (AGM) software product line (McGregor, 2004). This is an online artifact used for illustrating software product line concepts. It provides a realistic environment in which to study the product line strategy. In this section we describe the software development process for this product line. This example illustrates the general approach, but does not cover all technologies, methods, and strategies.

We chose this example in part because the SEI has developed a framework that lists many of the techniques, methods, and strategies that are used in software product line practice (SEI, 2004). The framework makes explicit a number of relationships. Users of this strategy can easily understand the implications of making specific choices.

Overview

AGM intends to build nine products, three different variations of three different games. The product line manager decided these products should be built using a product line strategy. The strategy encompasses both the technical and business portions of software-intensive product development. The scope of the software product line strategy is more comprehensive from the previous development strategy, so some number of processes need to be defined and coordinated.

A software product line realizes increases in productivity and decreased time to market by achieving strategically significant levels of reuse, often 85-95%. The product line organization defines two main development roles: core asset builders and product builders. The core asset builders create all the pieces needed to build the common pieces of the products. The product builders use the core assets to produce products. They also build those portions of the product that are unique. Previously, analysis of the requirements was to be handled by one group and then a second group handled all of the implementation. Teams had to be reassigned and trained in these new roles.

Before adopting the software product line approach, AGM used a traditional approach to software development:

- textual requirements,
- flowcharts of individual procedures, and
- implementation in the C language.

The process was largely waterfall. The process maturity of the organization was low, with each new project being conducted in isolation from other projects that had been attempted previously. In keeping with national averages, about half of the projects at AGM were canceled with no delivery.

For the products in the product line, AGM made certain strategic decisions and then focused on process definition. In addition to adopting the software product line strategy, the decision was made to outsource the actual implementation. An initial technology decision was also made. The product line managers decided the Arcade Game Maker product line would be implemented using the Microsoft Visual Studio .Net and the C# language, which uses component-based software engineering technology. The outsourcing strategy required a strong separation between the design and implementation. C# provides an interface structure that provides the separation, which C lacks. At the same time, C# was more compatible with the development culture of AGM than Java would have been.

Process Definition

We will address a number of issues about the changes in existing processes, but we will not attempt to provide a complete set of processes.

It is the goal of process definition to support the two chosen strategies by making compatible tactical decisions. The waterfall process model was not sufficiently flexible for the exploration that has to be accommodated when planning nine products simultaneously. An iterative process is flexible, but by itself is incompatible with the outsourcing strategy. The final basic process model for the AGM product line is two consecutive iterative processes in which the first process produces design outputs used as inputs to the second process where the design is implemented.

AGM had to coordinate requirements and design information for all nine products over the life of the product line. The new development process had to contain an advanced configuration management process to coordinate the many pieces for all of the products. The discipline was an important cultural feature.

Method Selection

To support the product line approach and the use of outsourcing, AGM needed improved communication of requirements and design information. Flowcharts only cover a very small portion of the design space and text descriptions of requirements and other design information are too ambiguous to ensure clear communication with an implementation partner. They adopted the Unified Modeling Language as the language to be used prior to source code. The Together modeling environment was selected. This became a requirement for the outsourcing contractor as well. It meant that the artifacts that serve as input and output between processes and phases within the process would all be in a single language.

Table 4. Technologies vs. models for the case study

	Scope	Granularity	Organization
Structure	Does the chosen model notation completely represent the chosen structure? *UML is a complete object representation notation.*	Does the chosen model notation at least reach the level of granularity of the chosen structure? *UML supports objects, the primary structure of C#.*	Does the chosen model notation organize information about the chosen structure in an intuitive manner? *UML has a number of diagrams that present the different facets of a system design.*
Notation	Is the chosen model notation compatible with the other notations used in the project? *UML and C# are compatible.*	Is the granularity of the model compatible with the notation of the rest of the project? *C# uses objects as the main structure, as does UML.*	Can the notation accompanying the technology express the organization of the model? *C# can express the relationships available in UML. It also has separate constructs for the static and dynamic aspects of the UML model.*
Transformations	Does the chosen model notation support transformations that map every model element to targets? *UML still uses text notes to explain some intent. These cannot be automatically transformed.*	Is the granularity of the model compatible with the level of transformation available in the technology? *A number of transforms work at the object level so they are compatible.*	Do the transformations of the technology operate with input/output pairs that reflect the organization of the model? *Analysis models are transformed into design models. The design models are transformed into code.*
Levels of Abstraction	Is the scope of interest sufficiently broad to encompass the levels of abstraction? *The definition of UML includes a metamodel that can correspond to the reflective language level.*	Does the granularity of the models and methods match the levels of abstraction of the technology? *Yes, as already shown.*	Is the organization of the models compatible with the levels of abstraction in the technology? *The UML models are seamless. Different levels of abstraction are handled in a compatible manner.*

Production Planning

The software product line strategy requires a production planning process. The core assets are designed to make the development of individual products an efficient process. In building the production plan, the AGM core asset developers make choices of technologies and tools that will be used. The

Table 5. Models vs. strategies for the case study

	Goals	Scope	Abstractions
Scope	Does the model scope cover all concerns of the strategy? *There are not sufficient models to cover all the goals of product lines. UML does not provide means of indicating measures of productivity, for example.*	Does the scope of the models match the scope of the strategy? *There are not sufficient models to cover the managerial aspects of product lines.*	Does the model scope encompass the abstractions of importance to the strategy? *The model can represent the programmatic abstractions, but no managerial aspects.*
Granularity	Are the goals of the strategy stated in terms that the granularity of the model can handle? *No, UML deals with objects while a product line deals with products.*	Is the scope of the strategy covered by the model? *No.*	Do some of the abstractions in the strategy match the fundamental units of the model? *Yes, the strategy produces reusable assets. Objects and components can be reusable.*
Organization	Does the organization of the model facilitate the goals of the strategy? *Yes, it contributes.*	Can the model be organized to cover the scope of the strategy? *No.*	Does the organization of the model make sense with the set of abstractions in the strategy? *The current models are a subset of the strategy.*

decision to use Microsoft Visual Studio.Net was made because the component structure supports the outsourcing strategy. Components are assembled based on interfaces. The implementations behind the interfaces can be done by different people with a minimum of communication. The design process outputs a set of interface definitions that can then be used as a basis for communication with the remote implementation team.

The AGM case study illustrates connections between a number of decisions. Table 4, Table 5, and Table 6 give connections between several decisions. The strategic decisions made before the product line organization was initiated led to specific selections of process model and development technologies. The result was a consistent, coherent development environment in which it was possible to meet corporate goals.

By answering the questions in the tables, we have identified a problem in the development environment, as shown in the shaded cells in the tables. The software product line strategy encompasses both technical and managerial aspects of product production. The manager has not yet identified a modeling

Table 6. Strategy vs. technology for case study

	Structure	Notation	Transformation	Levels of Abstraction
Goals	Does one of the chosen technologies provide structures that represent the goals of the strategy? *No.*	Does one of the chosen technologies provide notation capable of expressing the concepts in the strategy's goals? *No.*	Does one of the chosen technologies provide transformations that result in goal-level artifacts? *Yes, C# produces products.*	Do the levels of abstraction in the technology contribute to achieving the goals of the strategy? *Yes, the goals of reusability rely on the abstractions in UML and C#.*
Scope	Do the structures in the chosen technology cover the scope of the strategy? *Yes, the program-level structure in C# covers the scope of the strategy.*	Are the notations in the technologies capable of expressing the complete scope of the strategy? *No, it does not adequately cover non-functional issues.*	Are the results of a transformation still within the scope of the strategy? *Yes, the transformations create objects and programs.*	Do the levels of abstraction in the technology fit the scope of the strategy? *Yes.*
Abstractions	Are the abstractions defined in the strategy compatible with the structures defined in the technologies? *Yes, although the modeling technology cannot represent the full range.*	Can the notations in the technologies express the abstractions in the strategies? *Yes.*	Can the transformations in the technologies operate on the abstractions in the strategies? *Partially, there are abstractions in the strategy that are beyond the "program" scope and therefore beyond the technology.*	Can the levels of abstraction in the technology represent the abstractions in the strategy? *Yes.*

notation that addresses the managerial aspects. Boeckle et al. (2004) provides a modeling kit for addressing these abstractions. The manager will have to carefully monitor text documents until such time as this notation is in use.

Future Trends

New business models and strategies continue to emerge at an increasing rate. The software content of products is also increasing. We believe that our framework will remain in tact, and the landscape described in Figure 2 will simply grow larger to accommodate the choices facing managers and developers.

The increasing formality of modeling notations, advances in method engineering, and other process-related work will provide improved techniques for identifying dependencies and constraints between strategies and the technologies and models that support them.

Conclusion

We began with a view of the current state of software-intensive products:

- Complexity is increasing.
- Development times are getting shorter.
- Demands for flexibility are increasing.

In attempts to meet these challenges, software development organizations are exploring the development landscape shown in Figure 2. In order to be successful in adapting the development process to the new technologies, methods, and strategies:

- The manager must make systematic and controlled changes in the development environment.
- The manager must understand the relationships among the technologies, methods, and strategies, and how these relationships constrain the changes that can be made.

The development process, which blends these elements, is instantiated as a project plan for a specific product development effort. The project plan determines the trade-offs among the three factors shown in Figure 1.

We have illustrated the validity of our thesis by first defining the landscape of elements and then showing how the selection of certain elements causes changes to the process. Through the case study we have illustrated how the concerns are visible in a project and how they can be handled. We have outlined a number of tasks for the development manager and the development team. Our experience in a variety of environments and situations, and the experience of those we have referenced, shows that the return on investment for the time spent on these tasks will be high.

References

Ambler, S., & Nalbone, J. (2004). *Enterprise unified process.* Ronin International.

Beck, K. (1999) *Extreme programming explained: Embrace change*. Reading, MA: Addison-Wesley.

Beck, K. (2002). *Test driven development*. Reading, MA: Addison-Wesley.

Boeckle, G., McGregor, J.D., Clements, P., Muthig, D., & Schmid, K. (2004). Calculating return on investment for software product lines. *IEEE Software,* (May/June).

Boehm, B. (1988). A spiral model of software development and enhancement. *IEEE Computer, 21*(5), 61-72.

Butler Group. (2003, November). Application development strategies.

Clarke, D. (2004). The dangers of outsourcing (and what to do about them). *CIO,* (February).

Clements, P., & Northrop, L. (2002). *Software product lines*. Reading, MA: Addison-Wesley.

DoD (Department of Defense). (1988). DoD-STD-2167A: Defense System Software Development.

Eclipse. (2004). Retrieved from *www.eclipse.org*

Humphrey, W. (2002). Three process perspectives: Organization, teams, and people. *Annals of Software Engineering, 14,* 39-72.

Kruchten, P. (2000). *The Rational Unified Process®: An introduction.* Reading, MA: Addison-Wesley.

Major, M.L., & McGregor, J.D. (1999). A software development process for small projects. *IEEE Software*.

McGregor, J.D. (2004). Retrieved from *www.cs.clemson.edu/~johnmc/productLines/example/*

Paulk, M.C., Curtis, B., Chrissis, M.B., & Weber, C.V. (1993). Capability maturity model, version 1.1. *IEEE Software, 10*(4), 18-27.

Software Engineering Institute. (2004). Retrieved from *www.sei.cmu.edu/plp/*

Soley, R. (2000, November). *Model-Driven Architecture.* Object Management Group.

Tarr, P., Ossher, H., Harrison, W., & Sutton, S.M. Jr. (1999, May). N degrees of separation: Multi-dimensional separation of concerns, *Proceedings of the 21st International Conference on Software Engineering*.

Chapter IV

MDA Design Space and Project Planning

Boris Roussev
University of the Virgin Islands, USA

Abstract

The change to Model-Driven Architecture (MDA) with Executable UML (xUML) results in changes to the existing object-oriented development practices, techniques, and skills. To use a transformational approach to create objects with Finite State Machines (FSMs), communicating by exchanging signals, adopters of MDA and xUML have to acquire expertise in new areas such as domain modeling, concurrency, non-determinism, precise modeling with FSM, programming in model-manipulation action languages, writing OCL constraints, and clear separation of application from architecture. The much more complex xUML object model presents system analysts with a longer and steeper learning curve. In this chapter, we critically evaluate the opportunities, capabilities, limitations, and challenges of MDA based on xUML. The purpose of our analysis is to aid organizations, software developers, and software product managers in their transition to this new development paradigm, and to assist them in understanding how MDA and xUML change the software design space and project planning.

Introduction

With the conventional way of developing software, analysts create high-level system models independent of implementation detail. Then, designers adapt these models to the constraints imposed by the implementation platform and the non-functional system requirements. Finally, programmers write the system code from the design models. The software design activity is the act of elaborating informal analysis models by adding increasing amounts of detail (Kruchten, 2000). Analysis-level models are informal because they do not have executable semantics, and therefore cannot be formally checked.

With MDA (Mellor & Balcer, 2002), analysts create executable models in xUML, test these models, translate them to design models, and finally compile (translate) the executable models into source code for a particular platform. The MDA design activity is the act of buying, building, or adapting mapping functions and model compilers, and targeting a particular software platform (e.g., J2EE or .NET). We say that MDA is a translational approach to systems development because executable analysis-level models are translated to system code.

Standardized approaches to software development, such as Agile (Beck, 1999) and RUP (Kruchten, 2000), are weathered best practices applicable to wide ranges of software projects. These methods are not sufficient in constructing effective project plans and processes, reflecting the unique realities in software firms and markets. As useful and distilled as they are, best practices have two major shortfalls: 1) lack of predictive relationship between adopted process and product quality; and 2) inability to adapt to changing environments. The following example illustrates our point.

Consider the statement, "XP (an Agile methodology) without pair programming is not XP." This is a dogmatic, but not theoretically sound statement. There is no sufficient data in support of the proposition that the immediate feedback cycle of pair programming is better or worse than, say, solo programming with team software inspections, with respect to quality or productivity. There is evidence that in many contexts, software inspections produce products of superior quality and/or increase productivity (ePanel, 2003).

To overcome the deficiencies of the standardized approaches to software development, Russ and McGregor (2005), see the previous chapter, propose a model which structures the software development landscape into three cornerstone concerns: technology, process, and strategy. Each concern is further decomposed into several sub-concerns. A *concern* is a group of concepts, activities, roles, and models addressing a common issue, or in other words, a concern encapsulates micro-level knowledge about some aspect of project management. Concerns are neither orthogonal nor independent. To show briefly the relations between the three concerns, one can say that a process implements a strategy by making use of technologies.

In this context, Russ and McGregor define project planning as navigation in a multi-dimensional development space. A distinguishing feature of Russ and McGregor's model is its account for the economic context through the strategy concern. Putting strategy on an equal footing with technology and process accounts for the complex sources and processes of value creation. The result is a holistic model that treats risk and competition in a sophisticated manner by linking technical and economic dimensions.

The objectives of this chapter are to assess how MDA based on xUML enriches and at the same time limits the software development landscape, and to assist adopters of MDA in the transition to model-driven object-oriented (OO) software development.

The remainder of the chapter is structured as follows. First we offer a high-level overview of model-driven development. Then we assess MDA and xUML in the framework of Russ-McGregor's model of the software design space. That is followed by a discussion of the challenges adopters of MDA are likely to experience in the transition to model-driven development with xUML. The final section outlines a promising area of model-driven development and concludes.

MDA with xUML

MDA is one of the latest OMG initiatives. It is based on the ideas that code and executable models are operationally the same, that executable models can be simulated and tested upon construction, and that the verified model can be compiled to system code. One mature technology for designing executable models is xUML. xUML is a UML profile that defines executable semantics (UML AS, 2001) for a subset of the UML modeling languages (UML, 2004). xUML is an executable and translatable specification language. The constructive core of xUML includes object (class) models, object (class) FSMs, and communication diagrams.

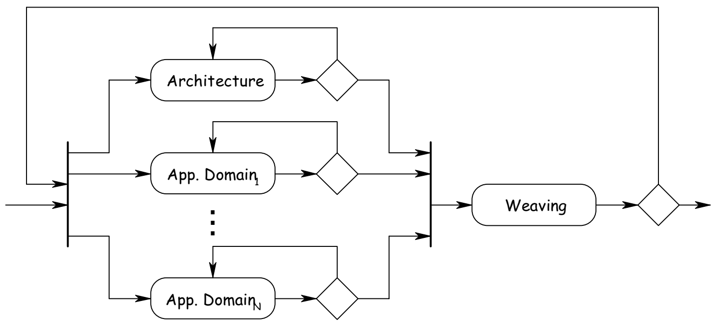
MDA separates systems into subject matters called domains, and specifies each domain with an executable platform-independent model (PIM) expressed in xUML. The executable PIMs are translated to platform-specific models (PSM), which, in turn, can be combined and translated mechanically into source code. A model compiler, called also architecture, encapsulates design and decision knowledge in the form of machine-readable mappings. It translates a system specified in xUML into a target programming language. Architectures can be bought or developed in-house. Executable models can be simulated with model simulators and debuggers before translation to code begins. Since the architecture is also a subject matter, the model's translation is the act of weaving together subject matters. Marks are modeling extensions used when there is ambiguity as to which mapping rule to apply to a modeling element, or when there is a need for additional information in order to apply a mapping. For example, a mark can designate a class attribute as persistent or a class as remote. Marks are defined by a marking model, which describes their structure and semantics.

To sum up, MDA defines the process of producing system code from a problem specification in xUML as a chain of mappings applied to marked models.

Separating the application model from the architecture model ensures that the architecture is applied uniformly and systematically to all application domains. The separation also simplifies and speeds up the development process by providing parallel tracks for the two activities. Figure 1 shows a workflow diagram for MDA development. The architecture modeling and the application domains modeling are activities carried out in parallel swim lanes. In addition, each application domain can be modeled independently of (concurrently with) the rest of the application domains. Another benefit from separating application and architecture is the possibility to enhance system performance without affecting system behavior, and vice versa, to change or extend the application functionality without modifying the underlying architecture.

The architecture is a set of rules, most commonly archetypes (similar to C++ templates), mapping model elements to blocks of code or to another model's elements. In MDA, the implementation phase is completely automated, while human expertise in the design phase is restricted to at most two junctions. One is the decision on what model compiler to use, based upon the targeted software platform. And second, if an in-house model compiler is developed or updated, a designer has to program the compiler's archetypes.

Figure 1. Concurrent activities in MDA development with xUML

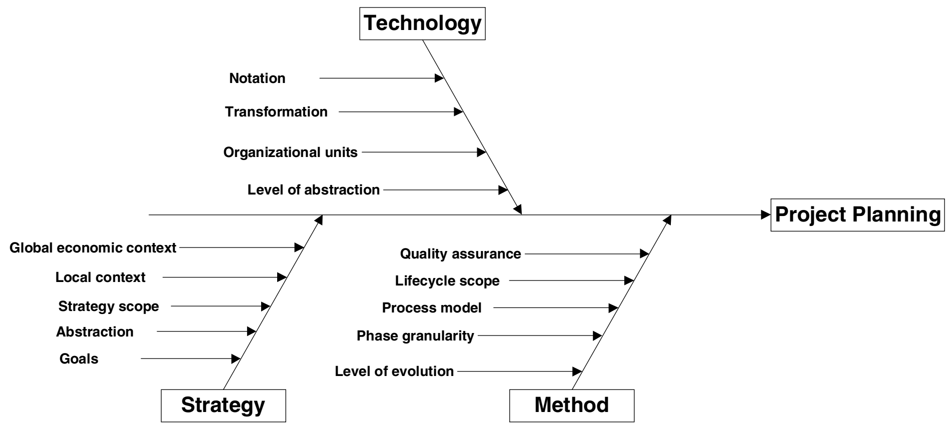


Analysis of Model-Driven Development with xUML

The change to model-driven development with xUML results in changes to the existing OO development practices, techniques, skills, and strategies. In this section, we analyze how MDA affects the OO software design space and project planning. Our assessment is done in the context of Russ-McGregor's model.

The cause-and-effect diagram in Figure 2 summarizes the Russ-McGregor model. We propose the following changes to the original model. The strategy concern is extended with two dimensions: internal (microeconomic) context and external (macroeconomic) context. The microeconomic discourse is defined by the organization itself. The macroeconomic discourse is the economic environment in which the organization operates. Two new dimensions are added to the process concern: quality assurance and level of evolution. The level of evolution captures the stability of the produced artifacts. Waterfall-shaped processes have a lower level of evolution. Iterative processes exhibit a higher and very dynamic level of evolution. Agile methods, for instance, are characterized by relentless refactoring and rework, and therefore exhibit an extremely high level of evolution. The level of evolution is an economic issue. If the risk exposure for a module is high, evolutionary approaches tend to perform better.

Figure 2. Russ-McGregor model



Next, we consider the impact of xUML on the different aspects of OO software development.

Technology Concerns

Software technologies are important enablers, as the demand for software and its complexity continue to grow. MDA with xUML is a tool-centric technology. It is unthinkable to use xUML without a proper tool support. xUML technology is accessible to developers by the following minimal set of tools: xUML development environment, xUML simulator and debugger, and xUML compiler.

At present, the time-to-market pressure mandates that software be produced to a great extent by adapting and integrating existing software. The trend in software technology is toward tools, models, and languages addressing the integration challenge. Integration with xUML is easier than with programming languages because of the clear and formal interfaces between the organizational structures (i.e., domain models) and the higher-level of abstraction of executable models.

A related trend is the shift to technologies expressing software without fixing the design to an implementation context, thus making software adaptable to unanticipated contexts. The xUML PIMs are not contaminated with implementation details, and with an appropriate architecture, the PIMs can be translated to any target platform.

The fact that xUML responds well to these two trends goes a long way toward building morphogenic software, software able to adapt to new contexts (Ossher & Tarr, 2001).

The technology concern is further decomposed into organizational units, notations, levels of abstraction, and transformations.

Organizational Structures

Russ and McGregor identify the following four ways of grouping together behavior and data: functions, objects, components, and aspects (Kiczales, 1996). xUML adds domains to the above list. In addition, xUML extends the object concept with the notion of lifecycle, defined by an FSM model, where the states of the object's FSM act as pre-conditions or guards for the invocation of the object's operations. Unification of data and behavior in objects with lifecycles is far more coherent and consistent than it is in conventional objects with operations, where the state-dependent behavior is represented implicitly.

Notation

True to its UML heritage, information in xUML is captured in several different notations at different times in the process. In distinction with UML, the different views are compatible, consistent, and completely interdependent. xUML models are formal models and not just descriptive drawings or charts. In addition to objects with FSMs, xUML relies heavily on capturing domain semantics with constraints written in OCL (Object Constraint Language). OCL is a formal, pure-expression language combining first-order predicate logic with a diagram navigation language (Warmer & Kleppe, 1998).

Level of Abstraction

To be executable, models must be precise and have actions associated with them. Actions are expressed in a language with action semantics. The action language used to manage objects, access attributes, traverse links, make choices, and iterate, inevitably brings down the level of abstraction of xUML models, and consequently makes analysis less abstract. But overall, the development process is at a higher level of abstraction than processes relying on UML, since the action language is simpler than typical implementation languages (e.g., C/C++, C#, or Java), has precise semantics, and is integrated with the visual models. The action language increases the expressivity of UML models.

With UML, system analysts are neither required to program nor to be clinically precise. xUML changes the responsibilities of the system analyst. Analysts must use FSMs, an action language, and OCL extensively. Even though OCL is a pure expression language, its constructs are somewhat cryptic; further, knowledge of logic, set theory, and multisets are prerequisites for writing successful OCL constraints (pre- and post-conditions, invariant, and guards). FSMs, action languages, and OCL increase the precision in the analysis phase at the expense of lowering its level of abstraction.

Transformations

The success of OO software processes is predicated on the fact that software development is defined as model transformation from more abstract to more detailed and precise models (Jacobson, 1992). Model transformations allow for user requirements to be traced to the modules of the implemented system, and ensure that user needs are met. It also plays an important role in change management. There is a broad understanding that in order to capture best the different system facets, modeling must be done from different perspectives, and that the different system views must be advanced simultaneously (Kruchten, 1995). The latter also relies on model transformation to ensure consistency among the models of the different views.

With respect to model transformation, xUML is the undisputed champion. The xUML technology is based on safe model transformations and mappings. Since MDA is a transformational approach to software development, a development team practicing MDA can concentrate on modeling the core business of the problem domain under consideration, and to spend less (or no) time on design and implementation detail.

In xUML, class diagrams, FSMs, communication diagrams, and state procedures are all different representations of the same underlying domain model. As a result, the freedom to advance simultaneously the different system views at intra-domain level is restricted. This might negatively affect the process of domain conceptualization.

Tools

Software development in the large requires appropriate tools and technologies. The already partly in place technical infrastructure needed to support MDA includes: model builders, verifiers, compilers, debuggers, analyzers, and testers. To make the most of these tools, they should be able to interoperate and interchange models seamlessly. The major impediment toward attaining seamless interchange is the specification of UML (Mellor, 2002). At present, UML specifies the abstract syntax of each modeling language separately. Mellor argues that in order for UML to be truly translatable and executable, UML should have a small executable kernel for defining the meaning of a problem solution and a separate presentation layer for representing elements or aspects in different forms (diagrams). This will not only resolve the consistency problems among different diagrams, but will also enable the free interchange of models among tools coming from different vendors, thus giving rise to a powerful tool chain.

There are several mature visual design tools on the market supporting MDA with xUML, for example, Kennedy Carter (KC, 2004) and BridgePoint (BP, 2004). These tools allow developers to construct visual models of the designed system, and to verify the models' behavior and properties through model-level simulation. The tools also provide model compilers for a number of software platforms.

Also, it should be noted that commercial MDA tools are expensive, support only small subsets of UML (might limit creativity or expression), and rely on proprietary languages to specify transformations between PIM, PSM, and code. The element of vendor dependence conflicts with the increasing demand for open standards and software integration (Bettin, 2004). The promise of MDA is that users will not be locked into a single vendor. This is difficult to imagine, given that there are no standards for an action and transformation languages. The xUML standard defines only the semantics of executable models, and not the syntax of the action language.

As a tool-centric technology, xUML forces organizations to collect data, which can help an organization move up the CMM maturity ladder (Paulk et al., 1993) faster.

Method

A software method describes a repeatable approach guiding the assignment and management of tasks in a development team or organization.

The rigor of executable models and their more-complex object model make executable models more difficult to design, but at the same time more useful. Since there is no implementation phase in the product development lifecycle, the overall time to complete a MDA project is shorter.

Organization

Every method allows a certain degree of concurrency among activities. The trend is toward more concurrency. The precise nature of xUML restricts concurrency at intra-domain level. The clean interfaces among problem domains, however, increase the degree of concurrency at inter-domain level, as shown in Figure 1. With MDA, concurrent modeling of domains is possible and almost a must. An activity, which can proceed in parallel with analysis, is building the mappings knowledge base (the architecture). The inter-domain concurrency, combined with concurrent architecture design, outweighs the limited concurrency at intra-domain level. xUML proponents maintain that the MDA development lifecycle is shorter than that of any other OO method (and it is only logical); however, to the best of our knowledge, formal studies have not been conducted.

Scope

The use of xUML spans the entire product lifecycle. In this respect, MDA is second to none. All other OO methods have to deal with the code view of the system separately from the system model. The MDA approach also outperforms other OO methods when it comes to aligning the development process to an organization's business goals.

Quality Assurance

Precise domain modeling results in system functionality meeting better client needs. Model executability enables a developer to ascertain that the high-level analysis model satisfies the use case specification by simulating the models through a discrete event simulation.

Another technique for quality control that can be considered is model checking. Model checking attempts to verify the properties of a system under all possible evolutions, not just under scenarios defined by analysts or tests generated by testing tools. We bring up the issue of model checking because executable models are a natural representation for model-based verification (Syriagina et al., 2002). This fact means higher yield on investment in formal methods. The conventional approach to software verification is to extract a verification model from the program code and to perform model checking on this verification model. However, the state space of a fully functional program is unbounded and far too complex for application of formal verification, except for toy examples or selected features of carefully selected blocks of code. Verification is only feasible when executed on an abstract model. Since xUML models are abstract and their complexity level is much less than the programs they are translated to, it is feasible to apply model checking to tested and validated executable models. Another advantage with executable models is that model checking can be performed directly on the PIM in terms of the domain properties. In conclusion, xUML has the potential to change the role of formal verification, and model checking can become mainstream. This, in turn, will have enormous impact on quality assurance.

Clients can expect improved quality with MDA, since design decisions are not intertwined with application logic. Implementation automation is another driver for superior quality. In xUML, implementation is replaced with mechanic construction—that is, applications are no longer re-implemented over and over again, but instead, existing architectures and design knowledge are reused. The complexity of reuse at design level is not something to be underestimated. This complexity is multiplicative, not additive (Mellor, Scott, Uhl, & Weise, 2004). For example, if there are three possible database servers, three different implementations of J2EE app server, and three operating systems, there are 27 possible implementations, even though there are only 10 components (the above nine, plus one for the application). Once the code of the application is mixed with that of the other parts, none of them can be reused separately because they all rely heavily on glue code that holds the system components together. This is not just an interface problem, even though it often manifests itself as such. The different system components may not match architecturally, a problem dubbed architectural mismatch (Garlan, Allen, & Ockerbloom, 1994), which complicates matters even more.

Translating executable models to application code minimizes coding and code inspection efforts, coding errors, and component integration problems. According to Project Technology experts (PT, 2004), organizations that have adopted xUML have experienced up to a ten-fold reduction in defect rates. What is particularly interesting is that the benefit of reduced defect rate is more significant for large-sized, complex projects—MDA scales up very well.

It would be interesting to see if a quality control technique such as pair modeling, the analog of pair programming, will find its way in MDA. Beck (1999) states, "If code reviews are good, we'll review code all the time (pair programming)." The rationale for pair programming is the error-prone nature of coding. If initial results are anything to go by, pair modeling should not be needed. Model inspections can be used in the way they have been used in programming (Gilb & Graham, 1993) for years.

Process Model

Model-driven development with xUML can be used with a waterfall process or an evolutionary process. A waterfall lifecycle works well for projects with stable requirements, known architecture, and a standard work breakdown structure (e.g., many real-time systems). In less deterministic and fast-paced environments, development methods are based on iterative and incremental lifecycle models.

Mellor (2004) proposes Agile MDA, a process based on many of the principles and practices of the Agile Alliance (2001). The basic premise of Agile MDA is that since executable models eliminate the verification gap between visual software models and end users, a running system can be delivered in small increments produced in short iterations, of one to two weeks.

As we already mentioned, MDA does not have a separate implementation phase, and its design phase differs substantially from conventional design activities in other OO approaches. There are new realities in the analysis phase as well. With MDA, analysis is more rigorous than ever before. The latter brings up two issues: 1) Are system analysts adequately prepared to handle mathematical rigor and programming in an action language (conversely, are designers, with experience in programming, ready to do analysis)? and 2) Are there situations where too much formalism too early would be a stumbling block? MDA defines the Computation-Independent Model (CIM), which, in our view, deals with these two issues. So, we end up with two different models, informal charts and formal executable models, the very idea against which MDA revolted.

Strategies

A development strategy outlines general directions for project activities to achieve certain strategic goals. From an engineering point of view, the goal of a software process is to produce high-quality software in a cost-effective way. This definition does not take into account the economic context in which software is produced. In engineering, cost is commonly equated with value added. From an economics point of view, software development is an investment activity, and as such, the ultimate goal of a software process is to create maximum value for any given investment (Boehm & Sullivan, 2000).

The difference between cost and value is best pronounced in a strategy targeting a dynamic marketplace, though there are more dimensions to it. Most software cost estimation models are calibrated to a minimal cost strategy, which does not include the time-to-market force. A software product produced at a minimal cost and brought to a competitive market three months after the competitor's analog has no value added, but possibly value destroyed. In other words, minimal cost is not equivalent to maximal value, because minimal cost captures only the direct cost of resources (e.g., time and money) spent on a project, without considering the opportunity cost of delay.

Abstraction

Russ and McGregor (2005) observe the trend toward strategies that make it possible to reason about development issues in terms of business goals. In xUML, developers model the semantics of a subject matter without any concern about the implementation details of the targeted platforms. The higher level of abstraction of executable domain models free of platform-specific detail brings closer software process, end product, and business goals. As a consequence, development concerns can be abstracted and mapped to business goals.

Macroeconomic Context

Both legacy and new software systems are rarely based on maintainable and scalable architectures. Design for change is a value-maximizing strategy only when changes can be anticipated correctly or development is not time-constrained. The question is, should one design for change if doing so would delay bringing a product to a competitive market or make a product expensive? In other words, a non-maintainable or non-scalable architecture might be the result of a value-optimal strategy compatible with the market force pushing for lower software development costs and shorter development lifecycles.

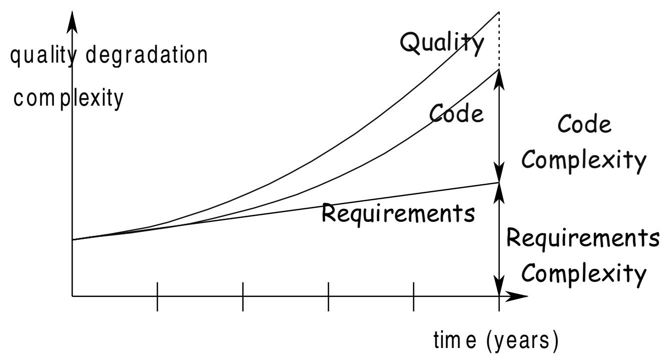
Figure 3 shows how code complexity grows over time (Bettin, 2004). From a certain point on, the aging takes its toll and a small requirements change entails disproportionate code complexity. From that threshold on, it is economically justified to scrap the system and to build or acquire a new one.

With MDA, the problem of change management in development and maintenance is alleviated because of the formally defined "trace" relationships among modeling and software artifacts, coming in the form of formally defined mapping functions. MDA pushes the threshold point forward in the future for two reasons: 1) technological changes do not affect the core domain models; and 2) requirements changes are handled at a higher level of abstraction, and due to the formal <<trace>> relationship, their impact is easier to assess.

Microeconomic Context

Henderson-Sellers (1996) and Bettin (2004) observe that software quality degrades faster when software is treated as capital cost—that is, maintenance only delays the inevitable obsolescence (see the Quality trend line in Figure 3). This view is incompatible with incrementally building software assets reused across a large number of enterprise applications, software assets whose value appreciates rather than depreciates (Bettin, 2004). In general, building software assets is a long-term investment strategy and must be carefully planned to ensure return on investment. Since xUML leverages domain-specific knowledge captured in executable domain models, strategic software assets are built at no extra cost. More importantly, the executable models acquire lasting value and become business assets because they are not byproducts of writing code, but they are the code. Furthermore, executable domain models do not

Figure 3. Rate of code complexity over time



degenerate into liabilities over time because they are aligned with the business processes of the enterprise (Bettin, 2004).

Strategies

We consider the impact of xUML on two widely practiced strategies for software development.

Outsourcing Strategy

Outsourcing is a product development strategy aiming at reduced production cost. With this strategy, product design is delegated to a team in a region where costs are favorable for the outsourcing client. Apart from the issue of intellectual property, the outsourcing strategy hangs on a balance between the cost of in-house training, expertise, and availability of human resources, and the inability of outsourcing suppliers to interact with clients. Depending on the conditions, outsourcing may increase or decrease the time required to bring a product to market. The isolation of the implementation partners and the concomitant poor communication may preclude the use of evolutionary process models in the entire lifecycle and impose a more rigid, waterfall-shaped process model. Russ and McGregor (2005) report a case study, where a two-phase waterfall model is combined with mini-iterations at phase level. The lack of clear communication calls for the use of a formal and rigorous notation to bring up the information richness of the dialogue between the outsourcing clients and outsourcing suppliers.

The analysis above suggests that an outsourcing strategy would benefit from the formal and precise nature of xUML. A viable option is to carry out domain partitioning and domain interface specification at the outsourcing client (onsite), and then to communicate these models to the implementation partners to develop the final product offsite.

Product Line Strategy

The product line strategy achieves strategically significant levels of reuse by identifying common functionality among multiple products. The product line strategy reduces the average time required to bring a set of products to market, at the expense of the time required to develop the first products in the set.

xUML fits well within the thrust of the product line strategy. On the one hand, the elevated level of abstraction makes it easier to modify the functionality of the core executable domain models to manufacture customized products. This ease of extensibility is combined with design (compiler) reuse. A single software architecture can be applied to all products targeting a common platform. On the other hand, an application can be translated to different platforms without any changes to the application model.

Challenges

As with any approach, there are pros and cons to adopting MDA. Separating application logic from software architecture facilitates portability and reuse of domain, application, and design intellectual property embodied in executable models. Though unlikely, these may come at the price of lower performance and even at the price of increased complexity.

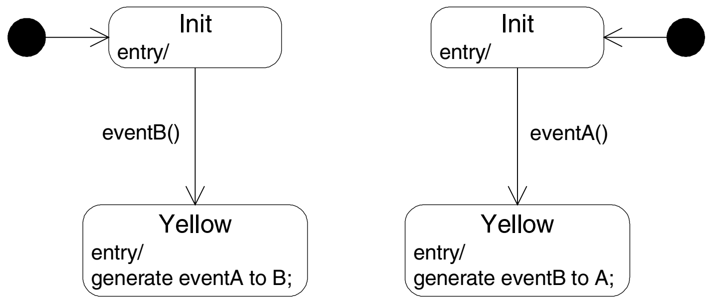
Concurrent Composition vs. Sequential Composition

In executable models, each object has its own FSM that defines its behavior. The FSMs evolve concurrently, and so the actions of the procedures are associated with the states of the different FSMs (unless, of course, constrained by synchronization). The onus is on the developer to sequence the actions of the different FSMs and to ensure object data consistency. The developer has to prove that the system is deadlock free (bad things will never happen) and that it is live (good things will eventually happen). The latter are the subject matter of the theory of concurrency. We have not seen a single presentation on xUML or MDA touching upon this issue.

Let us take as an example the composition of the innocuous state machines A and B shown in Figure 4. The behavior of each FSM taken in isolation is easily predictable. Composing the two, however, leads to a deadlock. A will wait forever for B to generate event EventB(), and B will wait forever for event EventA(). As a result of the composition, the modeler has to deal with the active states of the two state machines—the distributed system state.

In programming, for example, concurrency is restricted to a small collection of classes, instantiated usually from a catalog of concurrent design patterns (e.g., Lea, 1999). Alternatively, concurrency can be implemented as an aspect and weaved with the application logic as a service, for example, concurrency control in J2EE servers or in CORBA implementations. If we again compare xUML to programming, we will see that in programming, the default composition of the behaviors of a pair of objects is sequential, whereas in xUML models it is concurrent. Even though a concurrent composition has multiple advantages, such as effective mapping to different hardware architectures, better performance, and faithful representation of concurrent business pro-

Figure 4. Deadlock



cesses, verifying concurrent systems through simulation and testing alone is far less effective than doing so with sequential systems.

MDA Modeling Skills

The success of a software project depends on a synergy of soft and hard design techniques. System analysts working with customers and end users acquire expertise in the "soft" side of systems development. These skills are important because software systems are social systems, built for humans by humans. System analysts, in general, lack training in formal methods, and in particular, in the theory of concurrency. Formal methods, the "hard" part of system design, are typically mastered by computer engineers, who, on the other hand, are not versed in soft methods.

The origins of xUML, and the background of its creators, for that matter, can be traced to the realms of real-time systems, computer engineering, and formal models. This is indicative of the skills, rigor, and discipline required from xUML modelers. MDA demands from developers both "soft" and "hard" thinking, as this approach blurs the boundary between analysis and design. In MDA, it is more appropriate to speak of analysis/design models rather than just analysis models.

Not surprisingly, activity models are not among those commonly employed in MDA. Activity diagrams lack rigor, and they do not relate seamlessly to the rest of the UML modeling languages. This contradicts the idea of transformational development. However, activity diagrams are beneficial in business modeling. They capture in an intuitive way the business processes in an enterprise. Is there another rift here between "soft" and "hard" approaches?

According to Booch (2004), it takes as little as two months of on-the-job training and formal guidance to adopt the MDA approach. This time is certainly not more than what is required with any breakthrough technology. Booch also maintains that the learning curve for MDA is not steeper than that of UML. We would have agreed if there were no issue with concurrency and mathematical rigor.

Model Executability and Testing

The value of model simulators is in proving logical correctness. Since simulators do not test the real system, testing has to be done twice, once at model level and once at code level. Further, since simulation is not carried out in the target environment, its results have limited merit. On the upside, once a model compiler is acquired, the system code can be generated instantaneously, and tested and debugged in the real execution environment.

Conclusion

In this chapter, we analyzed how MDA and xUML change the software design space and project planning. We also discussed the difficulties that MDA adopters are likely to encounter in the transition to this new software development methodology.

MDA differentiates clearly between building abstract domain models and designing software architectures. The model-driven development makes possible the large-scale reuse of application models and software architectures, and allows for inter-domain concurrency and compression in the software development lifecycle. In addition, MDA automates all repetitive tasks in software development.

The cost of developing and maintaining systems with xUML is significantly lower. Once the models are constructed, they have greater longevity than code because they evolve independently of other models, in synch with the evolution of the enterprise business process.

The MDA approach is a nice fit for designing systems that are expected to have a long lifetime—that is, for lasting software assets. xUML is instrumental in preventing fast architectural degradation in large-sized systems and in building product lines producing families of sustainable software assets using highly automated processes.

By raising the level of abstraction of the software design process, MDA improves communication among stakeholders. The MDA approach aligns well with the business strategy and the micro- and macro-economic contexts, in which software products are being developed.

However, precise modeling, concurrency issues (synchronization, deadlock, liveness, boundedness, livelock), complex object model, and model-manipulation programming impose demanding requirements on developers. To meet these requirements, modelers need to acquire both higher-order analytical skills, technical skills, and skills in formal methods. This is not something to be underestimated. We spotlight system analysts' lack of expertise and experience in formal design as a major impediment toward the fast adoption of xUML. We see Domain-Specific Languages (DSLs) as a technology holding out a great promise to automate software development and overcome UML deficiencies in the areas of requirements engineering and initial systems analysis. DSLs are modeling languages targeting a particular technological domain or problem: for example, accountants use spreadsheets (not OO modeling) to express their accounting artifacts. DSLs bridge the gap between business and technology. In this area, the challenge ahead is the development of DSL models that naturally fit within their respective domains. The MDA framework can be used to define DSLs and to serve as a backbone holding together DSLs, product lines, and evolutionary agile processes. In this way, DSLs can become a viable alternative to UML modeling, addressing the needs of specific technological domains.

References

Agile Alliance. (2001). Agile Alliance manifesto. Retrieved from *www.aanpo.org*

Beck, K. (1999). *Extreme programming explained: Embrace change*. Reading, MA: Addison-Wesley.

Bettin, J. (2004). Model-driven software development: An emerging paradigm for industrialized software asset development. Retrieved from *http://www.softmetaware.com*

Boehm, B., & Sullivan, K. (2000). Software economics: A roadmap. In Finkelstein (Ed.), *The Future of Software Engineering, International Conference on Software Engineering,* Limerick, Ireland.

Booch, G. (2004). MDA: A motivated manifesto? *Software Development.* Retrieved from *www.sdmagazine.com*

Douglas, B. (2004). *Real-time UML* (3rd ed.). Boston: Addison-Wesley.

ePanel. (2003). Panel on software inspections and pair programming. Retrieved from *www.cebase.org*

Garlan, D., Allen, R., & Ockerbloom, J. (1994). Architectural mismatch: Why reuse is so hard. *IEEE Software, 12*(6), 17-26.

Gilb, T., & Graham, G. (1993). *Software inspection*. Reading, MA: Addison-Wesley.

Hardin R., Har'El, Z., & Kurshan, R. (1996). COSPAN. *Proceedings of CAV'96* (LNCS 1102, pp. 423-427).

Henderson-Sellers, B. (1996). *Object-oriented metrics, measures of complexity.* Englewood Cliffs, NJ: Prentice-Hall.

Jacobson, I. (1992). *Object-oriented software engineering: A use case driven approach*. Reading, MA: Addison-Wesley.

KC. (2004). iUML. Retrieved from *www.kc.com/MDA/xuml.html*

Kiczales, G. (1996). Aspect-oriented programming. *Computing Surveys, 28*(4), 154.

Kruchten, P. (1995). The 4+1 view model of software architecture. *IEEE Software*, *12*(6), 42-50.

Kruchten, P. (2000). *The Rational Unified Process®: An introduction*. Reading, MA: Addison-Wesley.

Mellor, S. (2002). Make models be assets. *Communications of the ACM*, *45*(11), 76.

Mellor, S.J. (2004). Agile MDA. Retrieved from *http://www.omg.org/agile*

Mellor, S.J., Scott, K., Uhl, A., & Weise, D. (2004). *MDA distilled*. Boston: Addison-Wesley.

Mellor, S.J., & Balcer, M.J. (2002). *Executable UML: A foundation for model-driven architecture.* Boston: Addison-Wesley Professional.

Ossher, O., & Tarr, P. (2001). Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, *44*(10), 43-50.

Paulk, M., Curtis, B., Chrissis, M., & Weber, C. (1993). Capability maturity model, version 1.1. *IEEE Software*, *10*(4), 18-27. Retrieved in 2004 from *www.sei.cmu.edu*

PT. (2004). Project technology. Retrieved from *www.projtech.com*

Russ, M., & McGregor, J.D. (2005). Management planning in a changing development environment. In B. Roussev (Ed.), *Management of the object-oriented software development process.* Hershey, PA: Idea Group Inc.

UML. (2004). Unified Modeling Language. Retrieved from *www.uml.org*

UML AS. (2001). UML actions semantics. Retrieved from *www.omg.org*

Warmer, J., & Kleppe, A. (1998). *The Object Constraint Language: Precise modeling with UML*. Reading, MA: Addison-Wesley.

Chapter V

Agile Outsourcing to India: Structure and Management

Boris Roussev
University of the Virgin Islands, USA

Ram Akella
University of California, USA

Abstract

The combination of low labor costs, technological sophistication, project management skills, and successful software establishment makes India a particularly attractive location for software production outsourcing. Furthermore, in most situations, information and communication technologies render virtual presence practically equivalent to physical presence, thus enabling effective communication and cooperation in a distributed mode. This chapter introduces a project structure creating agile conditions for large outsourcing software projects. The agility advantage is achieved by scaling down a large project into a number of small-sized projects working in agile settings. We divide the work into R&D activities, located onsite, and production activities, located offsite.

The proposed approach makes Agile applicable to the stressed condition of outsourcing environments without compromising the quality and the pace of the software development effort. Creating a context congenial to agile methods hinges on maintaining a good balance between the functions and sizes of onsite and offsite teams, on redefining the developers' roles, and on reorganizing the information flow between the different development activities to compensate for the lack of customer onsite, team co-location, and tacit project knowledge.

Introduction

We live in a digital world, where any activity not requiring a "physical presence" can be outsourced to any place that is connected. Even the term "physical presence" comes with a qualification. Information and communication technologies (ICTs) enable cooperation in a distributed mode. Technologies, such as groupware and video-conferencing, are increasingly becoming feasible for organizations to use in international projects. In addition, these technologies are changing the way we perceive presence and absence. The largely digital nature of software development allows changing its geography of provision. Advances in ICT have been essential for loosening the spatial constraints on software development.

The combination of low labor costs, technological sophistication, project management skills, and successful software establishment makes India a particularly attractive location for software production outsourcing. From 1996-1997 to 2003-2004, the export software sales of the Indian IT industry had an average annual growth rate of 38.7% to reach a total of US$12.2 billion in 2003-2004 (Nasscom, 2004).

Even though India has had a qualified labor pool and the enabling technologies, along with the great pressure exerted on firms in developed nations to lower costs, the Indian software industry still operates in the low value-added segments, typically in applications development, testing, and maintenance, while the high-end work, such as developing the IT strategy, building the software architecture, designing the system, integrating the project with enterprise packages, and designing custom components are all discharged by firms in developed countries (Dossani & Kenney, 2003).

Agile methods (Beck, 1999) are popular software development processes designed to be used on small- to mid-sized software projects. The Agile movement started in the mid-1990s. The first major project to apply an agile method was the Chrysler Comprehensive Compensation system, a payroll system developed in 1996. This method, called extreme programming, was described in Beck (1999) and became the foundation for the Agile Alliance Manifesto (Agile Alliance, 2001).

Agile methods are based on the notion that object-oriented software development is not a rigidly defined process, but an empirical one that may or may not be repeated with the same success under changed circumstances.

Agile methods are based in four critical values—simplicity, communication, feedback, and courage—informing a set of key practices (Pollice, 2004), which will be considered later on. Boehm and Turner (2004) define agile methods as "very light-weight processes that employ short iterative cycles; actively involve users to establish, prioritize, and verify requirements; and rely on tacit knowledge within a team as opposed to documentation."

Many of the agile practices are incompatible with the context of outsourcing, for example, customer onsite, team co-location, short lifecycle, and embracing change. But above all, agile methods can be applied only to small-sized projects.

In the past, there have been several attempts to reproduce the conditions for agility in large-sized projects. To the best of our knowledge, no such attempt has been made for outsourcing projects. Pollice (2001), Evans (2004), and Boehm and Turner (2004) propose to scale up agile methods by balancing agility and discipline. Pollice and Evans, for instance, look out for common grounds between Agile and RUP (Jacobson, Booch, & Rumbaugh, 1999), while Boehm and Turner try to get the best of both agile and plan-driven (waterfall) worlds. In contrast, Kruchten (2004) proposes to scale down large projects to meet the Agile "sweet spot," based on experience reported in Brownsword and Clements (1996) and Toth, Kruchten, and Paine (1994).

In this chapter we show how to reengineer large-sized (and small-sized) outsourcing projects to benefit from the "sweet spot" of Agile, while avoiding its "bitter spot." The proposed approach makes Agile applicable to the stressed condition of outsourcing environments, without compromising the quality of the software development effort. Creating a context congenial to agile methods hinges on maintaining a good balance between the functions and sizes of onsite and offsite teams, on redefining the developers' roles, and on reorganizing the information flow between the different development activities.

The rest of the chapter is structured as follows. First, we examine the state of the Indian software industry and show the problems experienced by Indian software suppliers. Next, we present the structure of the Agile outsourcing project, customized to the needs of the Indian outsourcing context. We then elaborate the inception and architectural activities, giving enough detail of how large-sized projects can be decomposed to a number of relatively independent agile projects and showing how the resulting builds are integrated. A discussion of many of the issues likely to be faced by Indian suppliers when applying the proposed approach follows, and the final section concludes and outlines plans for future work.

Outsourcing to India

Brief History

The genesis of outsourcing to Indian software firms can be traced back to the 1970s. By the late 1990s, India had become a leading supplier of contract software programming due to its combination of skilled, low-cost labor and project management skills (D'Costa, 2003). Indian software firms such as HCL, Infosys, and Wipro have become globally competitive. Further, their interaction with the global economy has contributed to the development of executive and managerial talent capable of securing overseas contracts, managing the interface with foreign customers, and migrating software development activities across national and firm boundaries.

Multinationals in developed countries and domestic firms quickly understood that there was a significant opportunity to undertake labor-cost arbitrage offshoring to India and moved, beginning in the 1990s, to establish Indian operations. Because the economics were so compelling, Indians living in the U.S. and the UK soon followed suit by establishing outsourcing firms in the U.S. and the UK, respectively, which discharged the work to India.

Drivers

The foremost reason for outsourcing is the potential cost savings, with quality commensurate or even better than that currently provided by the developed nation software developers. India-based outsourcers estimate that, in general, savings on a given activity would have to be at least 40% to make the relocation worthwhile (Dossani & Kenney, 2003). We group the rest of the drivers for outsourcing into technological and economic.

During the last decade, the cost of data transmission has dropped by more than 50%. The lowered telecom costs make it feasible to undertake even communication-intensive tasks outside of the U.S.

Another technological enabler is the ongoing revolution in the world of documents. Today's industrial-strength scanners digitize documents at the rate of 400 pages per minute. The digitized documents can be viewed anywhere in the world on a computer with a high-speed connection.

The last technological development we would like to consider is the widespread use of standard software platforms in corporate information systems, for example, database systems and ERP systems. Standard software platforms facilitate outsourcing since domain knowledge becomes uniform and transferable across industries, or branches thereof. This means that employees ought to acquire only standard, portable skills, thus lessening risks for both suppliers and clients.

Technology is necessary, but not sufficient to convince companies that they should outsource software development to India. Clients need assurance that the process would not be to their detriment and would bring in good return on investment. India has a very successful software establishment, with a proven track record in satisfying international customers in the area of application programming and maintenance. This creates the necessary comfort levels in the clients. The comfort-building process is assisted by the visible presence of large multinationals like IBM, Dell, Microsoft, General Electric, and British Airways, who have established Indian operations over the past years.

Another very significant factor driving software development outsourcing is the profitability crisis in the developed countries. With revenues largely stagnant for the last four years, firms are under intense pressure to cut costs while retaining quality.

Analysis of the Indian IT Sector

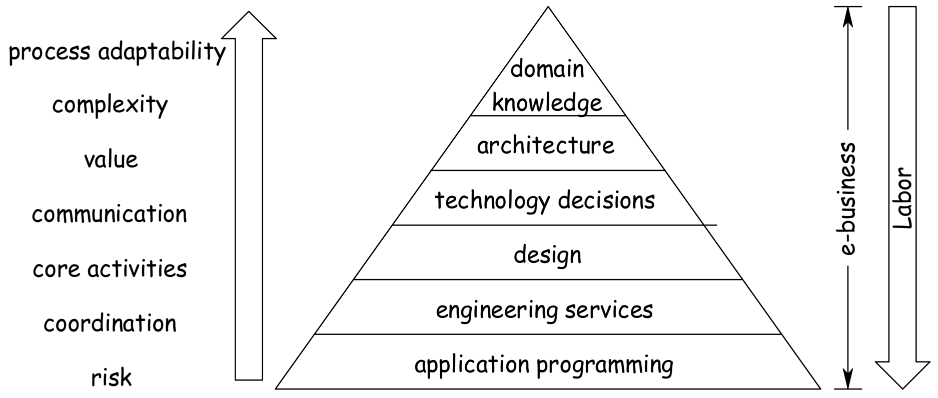
In their analysis, Akella and Dossani (2004) divide the Indian software export into two functional areas: services and products. The authors conclude that, in countries like India, the ability to develop global (but not local) products is more difficult relative to producing services. They observe that, in the Indian software export, services dominate products, and also that services in particular are supplied mostly to non-IT firms in the core, non-critical components of their activities.

A second division is between serving clients from the IT sector and clients outside the IT sector. The ratio between the two sectors offers an insight into the maturity of a software industry: the greater the numbers and the greater the relative weight for the non-IT sector, the more mature the software industry is.

Akella and Dossani classify the Indian software industry output into the following nearly exhaustive list of categories: 1) Web-based support; 2) data processing and management; 3) embedded software; 4) maintenance and legacy systems; and 5) application software development and maintenance. The complexities of the activities involved in all five categories fare low in the complexity pyramid of labor, as shown in Figure 1. Moreover, all five categories predominantly serve clients from the IT sector.

Both IT and non-IT client activities can be divided into core and non-core on the one hand, and critical and non-critical on the other hand. Core activities are manufacturing and engineering services that differentiate a firm from its competitors, and they are key to its continued growth. Non-core activities are the firm's non-competitive services, such as payroll and human resources. The boundary between core and non-core is not clearly cut. Within both core and non-core activities, there are activities crucial to their functioning, termed critical activities.

Figure 1. Activities and pyramid of labor



From Indian suppliers' point of view, non-core and non-critical activities are the easiest to obtain because they tend to be the most standardized across industries, and clients are most willing to outsource them because they do not involve knowledge transfer. Indian suppliers commonly work on applications at the next more difficult level—core, non-critical activities, and rarely on core, critical activities (Akella & Dosssani, 2004).

Strengths of Indian Software Firms

The substantial share of services provided to non-IT clients suggests that Indian firms have expanded their project management capabilities and started to acquire domain knowledge. This could explain the trend of moving away from consultants to direct supplying of end users.

The resource of skilled and motivated people is the most important component of any project. The level of experience and educational qualification of staff at Indian software firms is commensurate to that of most developed countries. The major Indian corporations, for example, are overwhelmingly qualified for ISO 9001, a European standard on quality assurance, and most of them have SEI/CMM certification with a high average score of 4.2 (out of 5) (Nasscom, 2004). SEI/CMM is the Carnegie Mellon University-based Software Engineering Institute's Capability Maturity Model (Paulk, Curtis, Chrissis, & Weber, 1993).

Availability of skilled workers is complemented by a rising supply of CS/EE graduates, which was expected to reach about 99,000 new graduates in 2004-2005 (Nasscom, 2004). Furthermore, Akella and Dossani's study implies that Indian suppliers have the skills to manage projects remotely.

It has been observed that the ratio of onsite/offsite workers increases with firm size. This trend reflects the growing capabilities of larger Indian firms to provide more sophisticated consulting services. The approach proposed in this work relies on an increased onsite presence.

Outsourcing Software Development Activities

What Can Be Outsourced?

When the software development process is considered in its totality, it appears to resist relocation because software production requires face-to-face interactivity with clients (and among co-developers), for example, user requirements elicitation and testing. The development workflow has to be parsed into activities requiring different levels of interactivity, and the less client-communication-intensive activities can be potentially outsourced.

The software complexity chain is frequently described as a series of processes, each of which is less complex than the earlier one. The initial processes are less labor intensive than the later ones. The complexity of a process is roughly inverse-proportional to its labor intensity. The pyramid model in Figure 1 gradates the processes in terms of labor, complexity, risk, and communication intensity.

Theoretically, outsourcing is possible for all the levels of the complexity pyramid, but there are important differences of how outsourcing for each level is likely to be implemented. The processes at or closer to the pyramid's apex, such as domain knowledge acquisition, architecture design, and technology determination, are objectively more difficult to outsource.

Moving up the complexity pyramid entails establishing an intimate client-supplier rapport and acquiring knowledge about the client's core and critical activities. The higher the pyramid level is, the more communication-intensive the activities become, and the bigger the demand grows for domain, architectural, and design knowledge. Adopting agile methods can help Indian suppliers move up the "value chain" because agile methods address the issues enumerated above.

Developing domain expertise, however, is difficult and risky because the firm becomes dependent on a single industry sector or process. And yet, specialization offers the potential of finding a niche outside the ferocious competition in highly commoditized sectors.

Smaller software firms are at a disadvantage because they lack domain skills and acquiring domain knowledge may be risky. The step transforming the business proposition from simple labor cost arbitrage to significant value addition involves acquiring deep-enough domain expertise.

Interactivity

Interactivity always comes across as a stumbling block for outsourcing. Interactivity has two dimensions—interaction among co-developers and interaction with clients. Requirements elicitation and acceptance testing are the activities requiring the most active involvement on the part of the client, which makes them impossible to offshore.

The greater the need of co-developers to interact across a set of different activities of the software process, the higher the risk threshold of outsourcing a subset of the activities is. Outsourcing the entire set of activities might be considered as a way of retaining interactivity at the new location. But, if some activities cannot be outsourced because that would disrupt the interaction with the client, then outsourcing the others might need rethinking.

Savings from Concentrating Activities in One Location

Often, a number of teams distributed across multiple time zones and reporting to different companies work on a large common project. There might be different reasons for having multiple teams with an inefficient spatial posture cooperating on a project. Most commonly, as companies expand, they outgrow the local labor pools and must establish teams in other regions. It might be too expensive to consolidate these teams in a single location in a developed country, especially for multinationals (e.g., a company with teams in France, Switzerland, Canada, and the Silicon Valley).

The advantages of concentration by relocating to India stem from the larger team size, which relates to economies of scale, such as retaining interactivity at the new location, standardized management practices, and tacit knowledge dissemination. Furthermore, a larger team is likely to have experts with precious domain knowledge in different industry sectors or technologies. In addition, an outsourcing supplier could pool many clients' businesses. A big firm in India can offer guarantees of quality that smaller domestic software firms could not match, owing to the larger labor pool and the division of labor.

For foreign equity participation companies (FEPs)—Indian firms whose foreign equity funding is 100%—the software development process can be reengineered during the act of outsourcing, and inefficient and ineffective practices can be easily abandoned. Very often such practices are legacies of earlier methodologies, such as waterfall lifecycle, iteration planning, and project controlling, that were not eliminated as the development process evolved. As Arthur (1994) has observed, all too often processes evolve in a path-dependent manner, which may not be the most efficient configuration. These inefficiencies can be addressed during the transfer without disrupting work patterns, since the developers at the new location would not have the outsourcing company culture.

Rethinking of Earlier Cost-Benefit Decisions

The lower cost of more highly skilled personnel calls for rethinking of established cost-benefit decisions. The much lower cost of software engineers in India compared to the U.S. makes it possible to increase the density of software inspections or to dispel managers' doubts about the feasibility of pair programming. Other activities that are candidates for reconsideration are regression and integration testing, iteration planning, and metrics for tracking project progress. In all of the above-mentioned practices, lower labor costs can change the break-even point, and thus create new sources of revenue.

There may also be diseconomies of scale. There are quite naturally risks associated with centralizing the development process at one location. The most significant of these is the danger of losing touch with the national environment(s). In this respect, the effectiveness of the communication process between supplier and outsourcer is critical.

Problems Experienced by Indian Suppliers

The following issues are likely to have an impact on outsourcing. The major challenge faced by Indian software firms is shortage of domain expertise in some function areas. According to the Outsourcing Institute (2004), domain knowledge and expertise are needed in subject areas that are new to India, such as finance, insurance, healthcare, and logistics. However, Indian software firms have acquired sufficient expertise in accounting and banking, which have already been successfully outsourced to India. Maintaining a seamless relationship between the outsourcing supplier and the outsourcing organization back in the developed country is always a challenge.

The problems experienced by Indian suppliers can be "objective" or "subjective." For instance, it is objectively difficult to communicate effectively with a remote client. The lack of expertise in project scheduling is, on the other hand, subjective, particular because it is not universally true. More often than not, a lot more could be done to alleviate the subjective problems rather than the objective ones.

Outsourcing problems fall into three categories: 1) communication—relationship with remote clients; 2) technical know-how—domain expertise, experience in building software architectures, design experience, and inadequate quality thereof; and 3) management—project coordination and project scheduling.

In order to compete successfully on the global market for software products and long-lasting assets (reusable domain solutions), Indian software firms need to resolve the communication, technical know-how, and management issues listed above.

The Agile Outsourcing Project

In this section, we present an agile method of software development geared toward the context of outsourcing.

Core Agile Practices

Agile methods are iterative and incremental. In an iterative process, the activities that were performed earlier in the process are revisited later. Revisiting activities provides developers in areas such as requirements engineering and software architecture with the opportunity to fix mistakes they have made. The iterative model implies more interactions among the various management and technical groups. It is a means of carrying out exploratory studies early on in a project when the team develops the requirements and discovers a suitable architecture.

Some important agile practices are the following (Beck, 1999):

- **Embracing Change:** Respond to change quickly to create more value for the customer.

- **Fast Cycle:** Deliver small releases frequently, implement highest priority functions first, speed up requirements elicitation, and integrate daily.
- **Simple Design:** Strive for a lean design, restructure (refactor) the design to improve communication and flexibility or to remove duplication, while at the same time preserve the design's behavior.
- **Pair Programming:** To quote from Beck, "If code reviews are good, we'll review code all the time." With pair programming, two programmers collaborate side by side at one machine. This quality control practice also helps disseminate tacit knowledge among the team members.
- **Test-Driven Development:** Quality control technique, where a developer first writes a test, and then writes the code to pass that test.
- **Tacit Knowledge:** Preference for project knowledge in team members' heads rather than in documents.
- **Customer Onsite:** Continuous access to a customer to resolve ambiguities, set priorities, establish scope and boundary conditions, and provide test scenarios.
- **Co-Location:** Developers and onsite customer work together in a common room to enhance tacit project knowledge and deepen members' expertise.
- **Retrospection:** A post-iteration review of the work done and work planned. This reflective activity facilitates learning and helps improve the estimates for the rest of the project.

The enumerated practices (or disciplines) vary with the method. The disciplines can be divided into three broad categories: 1) communication, such as pair-programming and tacit knowledge; 2) management, such as planning game, scrums (short daily planning sessions, in which the whole team takes part), short cycle, and frequent delivery; and 3) technical, such as test-driven design, simple design, and refactoring. These categories correspond to the three groups of problems experienced by Indian suppliers, as previously discussed.

The fast cycle of agile methods gives business stakeholders multiple and timely feedback opportunities, which makes agile methods explorative (aid in architecture discovery and requirements verification) and adaptive to change. There is a broad consensus that agile methods outperform other software development methods for small projects in dynamic environments, such as in e-commerce.

Figure 2. Impact of agile practices on Indian suppliers' concerns

Practices\Concerns	Quality	Coordination	Communication	Scheduling	Design	Architecture	Domain expertise
Customer onsite			✓				✓
Fast cycle	✓		✓		✓	✓	
Embracing change	✓			✓		✓	
Test-driven development	✓					✓	
Refactoring				✓	✓		
Simple design	✓				✓		
Retrospection	✓	✓	✓			✓	
Risk mitigation				✓			
Tacit knowledge		✓	✓		✓		
Co-location		✓	✓		✓		
Planning game		✓		✓			
Scrums		✓		✓			
Pair-programming	✓				✓		
Immediate customer feedback	✓		✓			✓	✓

What we see as a problem with all agile methods (e.g., extreme programming) is that they do not provide guidelines for building high-quality software, which is a natural consequence of their informal nature.

The feasibility-impact grid in Figure 2 shows how the core agile practices can alleviate the Indian suppliers' concerns.

Structuring the Agile Outsourcing Project

The main question we address below is how to reproduce the conditions ideal for agility in an outsourcing project.

Even a quick look at the core agile practices above would reveal that some of them are incompatible, while others are not entirely consistent with the context of outsourcing, for example, customer onsite, co-location, short lifecycle, and embracing change. Above all, agile methods can be applied only to small projects.

Kruchten (2004) proposes to scale down large projects to meet the Agile "sweet spot." The author describes the organization of a large project as an evolving structure, starting out as one, co-located team, which over time is transformed to a team of teams (with a tree-like structure).

Kruchten suggests organizing the iterative process into four typical RUP phases, as shown in Figure 3. Each phase shifts the focus of the development effort onto a different activity. A phase consists of one or more iterations, where iterations can be thought of as mini waterfalls.

The structure of the agile outsourcing project is based in Kruchten's approach. However, in an agile outsourcing project, the primary goal is not to slice a big project into multiple agile subprojects (which might be required anyway), but to outsource the project to one or more agile teams, which are co-located at a remote site and share common culture and educational background.

The structure of the development process is illustrated in Figure 4. The phases of the lifecycle are separated between "research and development" (R&D) activities and "production" activities, as suggested in Royce (2002). R&D is carried out onsite—close to the client—while production takes place offsite (the supplier's site). Elaboration is split between R&D and production. The two new phases are called architectural elaboration and production elaboration.

Figure 3. Typical RUP phases of the iterative process

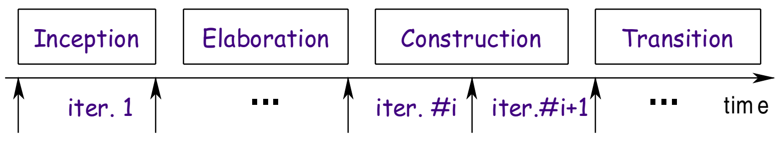
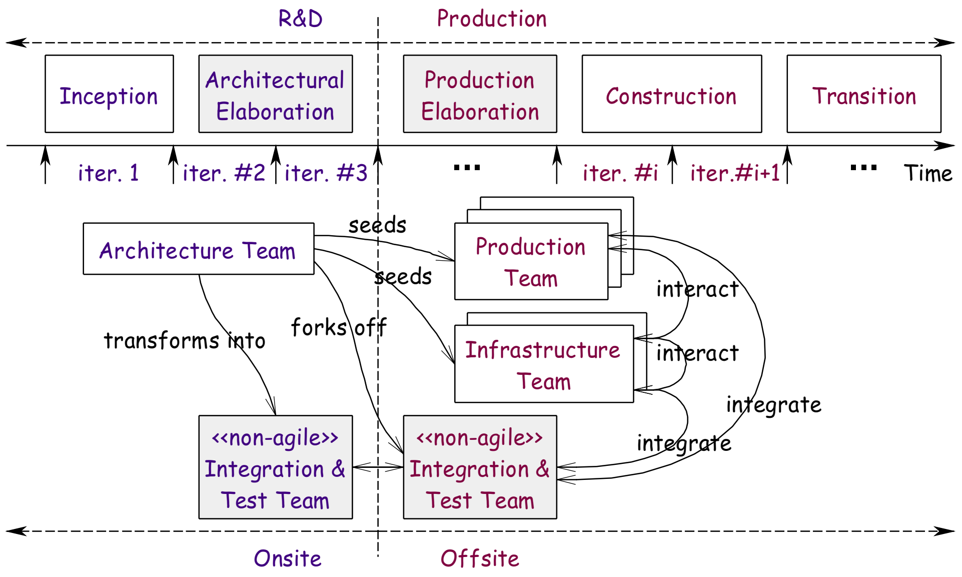


Figure 4. Team structure of agile outsourcing



A team comprising requirements engineers and architects starts the development process. Initially, this team focuses on the business case, vision, and requirements. The team works closely with the client in a typical agile setting. The team's objective, however, is not to deliver executable code (a must in agile methods), but to set up an architectural prototype, including prioritized system-level use cases.

When the team has got enough clarity on key architectural choices, it can set about building and testing an architectural prototype.

Towards the end of the architectural elaboration phase, when the architecture stabilizes, additional teams are created offsite. Each new team is seeded preferably with two members of the architecture team. For large projects, the early architectural prototype is used to slice the project into smaller, considerably independent agile projects. Each agile project "owns" part of the system architecture.

The "seed developers" transfer domain expertise from the client to the supplier's site. The "seed developers" take on several roles. They lead the teams, serve as local architects, act as customer surrogates, communicate with the initial architecture team, and if necessary, communicate directly with the client. This organization creates near-perfect conditions for agility in each offsite team.

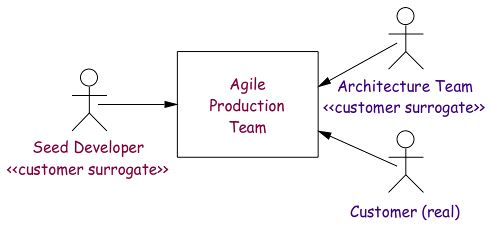
Each agile offsite team works on a subsystem and develops its detailed subsystem use case model—hence the second elaboration phase, named "production elaboration" in Figure 4.

For small projects, one offsite production team will suffice to develop the complete code. For large projects, however, several production teams might be necessary to do the job. In addition, one or more infrastructure teams must be set up to develop common elements such as middleware, services, and any reusable assets. The customers for the infrastructure teams are all other teams. Thus, no matter how many teams are spawned offsite, they all work in agile conditions, even though predominantly with customer surrogates.

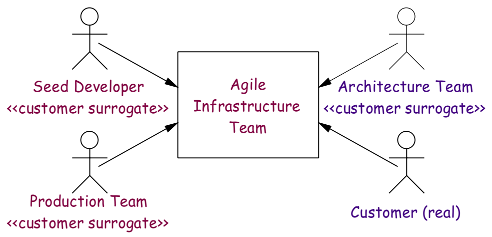
It is important to use a common software tool to reduce the risk of miscommunication, to track and communicate project requirements, to identify replicated modules, to map test cases to requirements, and to successfully integrate the outputs from the agile teams' builds.

The context diagrams in Figure 5 model the environments in which the different offsite teams operate. Note the dual nature of "seed developers." On the one

Figure 5. Context diagrams for offsite teams


Agile Production Team
Seed Developer <<customer surrogate>>
Architecture Team <<customer surrogate>>
Customer (real)

(a) Production team


Agile Infrastructure Team
Seed Developer <<customer surrogate>>
Production Team <<customer surrogate>>
Architecture Team <<customer surrogate>>
Customer (real)

(b) Infrastructure team

hand, a "seed developer" impersonates a customer, and on the other hand, he/she is part of the team.

The interactions of offsite teams with the real customer are supposed to be infrequent, and to be mediated by the "seed developers" and the architecture team. For large projects, somebody needs to put together the builds delivered by the production and infrastructure teams, and to test the assembled system. This job is assigned to the Integration & Test Team, or integration team for short. The testing engages the client so that the client's feedback situates the project and steers the future effort.

The problem with the integration team is that it gets input from the offsite teams, but receives feedback from the client. Normally, the input and the feedback are coming from the same place—the customer. To account for this anomaly, we split the integration team into two teams, one located onsite and the other offsite (see Figure 4). Both integration teams derive directly from the initial architecture team. This guarantees that the developers in charge of integration and testing thoroughly understand the client's needs.

A sore issue stemming from the proposed organization is that of fault propagation. For example, a defect committed by one of the production or infrastructure teams can propagate through the integration process before the client detects it. Owing to the late stage of its detection, isolating and removing such a defect is expensive.

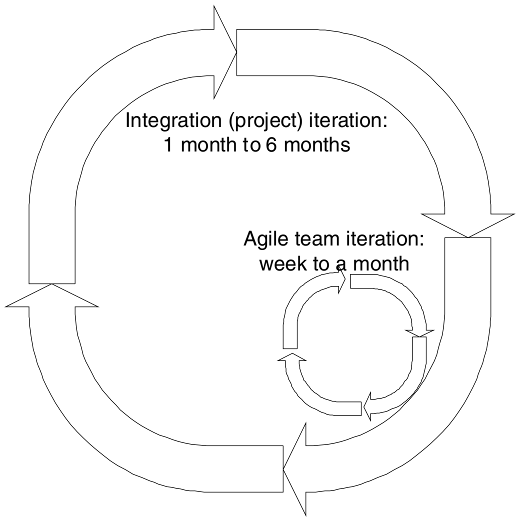
We have provisioned for two floodgates preventing fault propagation: 1) the "seed developers" in each offsite team; and 2) the test engineers in the integration team. Since they all come from the primary architecture team, it is very likely that they would be able to detect many of defects, which are normally revealed with help from customers.

The only two teams operating in non-agile, but still iterative and incremental mode, are the onsite and offsite integration teams.

The iterations (heart beats) of both the agile and integration teams can be best illustrated with the dual beat structure shown in Figure 6. Striking a balance between the lengths of the agile and integration iterations is the underlying objective of the management team.

The management team, composed of all local team managers, is led by the managers of the integration teams. We do not show the management team as a separate box in Figure 4 because management is thinly distributed across all teams.

Figure 6. Nested iterations



Since all offsite teams reside in the same country, and most probably in the same city, say Bangalore, there are no cultural differences to overcome, and communications among production, infrastructure, and integration teams are not as ineffective as they are with geographically distributed teams. The management team is in a favorable position because good communication is a prerequisite for effective coordination.

Activities and Workflow

In this section, we give more detail about the activities in an agile outsourcing project, as well as their workflow and information artifacts. We focus on inception, architectural elaboration, and production elaboration because they are critical to creating the agile contexts for the offsite teams.

Inception Phase

During inception, the emphasis is on the user requirements. A user requirement is a specification of what the system must do. As user requirements are elicited, they are organized into use cases. Very briefly, use cases are dialogical descriptions of system usage. By coalescing related transactions and scenarios into identifiable chunks, use case models focus on the dynamic behavior of a system at a macro level.

Use case descriptions can smoothly scale to large and small systems alike. They promote refinement and traceability from system-level usage goals down to low-level (subsystem, component, and instance) usage goals. Use cases are sufficiently flexible to be used in highly iterative and incremental development environments, as the one proposed in this work.

If a system consists of several subsystems, use case analysis can be applied recursively to the subsystems as well. This defines clearly the requirements and responsibilities of each subsystem. Subsystem-level use cases are derived from the system-level use cases and the system architecture (the architectural decomposition of the system into subsystems).

A system is decomposed into subsystems of related use cases, for example, use cases linked by <<include>> and <<extend>> dependency relationships or use cases sharing common actors. The <<include>> relationship is used to extract out a coherent part of a use case, typically for the purpose of reuse. It can also be used to decompose a system-level use case into part use cases to be realized by different subsystems.

The following technique is helpful in mapping a system-level use case to two or more subsystem use cases. The high-level use case is decomposed into a partially ordered set of activities on an activity diagram, where the different subsystems are modeled as swim lanes. Concurrent activities can be mapped to different swim lanes. Each such activity forms a subsystem use case. The actors of the subsystem use cases may be the original actors or other subsystems.

Architectural Elaboration

The onsite team carries out the architectural elaboration. The goals of the architecture team are to partition the system into multiple semantic domains centered around different subject matters, to define the architectural decomposition of the system into subsystems, and to map system-level use cases to subsystem use cases.

The architecture team defines the model organization of the system being developed, and thus determines the allocation of jobs to teams. Appropriate model organization would allow teams to work effectively on a common project. The two main issues in model organization are how to allow different teams to reuse parts of the projects they do not "own" and how to efficiently implement a build process.

A UML package is a container for modeling elements, and as an organizational unit, it is a natural choice for a configuration item (CI) (a piece of ownership) in a configuration management (CM) tool. A package defines a namespace for the modeling elements it contains. Since UML offers no guidelines as to what constitutes a package, the question is what modeling elements should go into one package versus another.

Domain Modeling

A domain is a subject area with a shared vocabulary (Mellor & Balcer, 2002), for example, user interface (UI) or payment transaction management. Each domain contains many classes organized around a single subject matter. Most domains require specialized expertise, such as experience and knowledge in UI design or in payment transaction management. It makes sense to allocate the modeling of a domain to a developer with domain knowledge in that particular subject area.

Since a domain model captures precisely the conceptual entities of a single subject matter, it can be said that domain models are logical models. Logical models are in sharp contrast to subsystem models, which are pieces of the physical system. Typically, a physical subsystem is constructed from instances of several logical models. For example, a collaboration realizing a system-level use case would involve instances from a UI domain, a business logic domain, a transaction management domain, a persistent storage domain, and a security domain.

At first, it might seem that domains add yet another concern to a software development initiative, but it is all about economic numbers. Constructing domain models reduces production cost by simplifying the development process and by improving product quality.

The underlying philosophy is that a domain-specific model captures precious domain knowledge (e.g., persistence). Domain modeling leverages scarce domain knowledge normally limited to very few team members and shields the rest of the team from the domain implementation detail. For example, to make an object persistent, a developer needs only to mark its class or one of its attributes as persistent, and a persistent software entity is automatically generated at compile time. The result is a simplified development process, where only a few developers need detailed knowledge about domain technicalities.

Henderson-Sellers (1996) and Bettin (2004) observe that software quality degrades faster when software is treated as capital cost (i.e., maintenance only delays the inevitable obsolescence). Treating software as capital cost is incompatible with incrementally building software assets[1] and strategic software[2] assets reused across a large number of enterprise applications, software assets whose value appreciates rather than depreciates. Since domain modeling leverages domain-specific knowledge captured in the form of domain models, strategic software assets are built at no extra cost. Domain models do not degenerate into liabilities[3] over time because they are aligned with the business process architecture of the enterprise and they have the potential to achieve mass customization[4] (Bettin, 2004).

Domains, unlike objects, are not elemental, but just like objects they are cohesive. The classes and components in a domain are tightly coupled and interdependent, and yet the domain is autonomous—its classes and components are decoupled from entities lying outside the domain boundary. Once constructed, domain models have greater longevity than an application because they evolve independently of other domain models out of which the application is built; that is, they become corporate assets and the biggest units of reuse.

Companies want to be able to adapt their software systems to the constantly changing business environment with minimum effort. It is easy to cope with the intra-domain effect of a change because the domain model naturally fits to the domain's subject matter, and the effect of the change is expressed in the language of the domain (i.e., in familiar domain concepts). The latter taken to the extreme is called end user programming. This scenario works only if no changes are made to the modeling notation. Changing the modeling notation leads to domain engineering and software product line engineering (SEI, 2004).

Domain modeling also alleviates the problem with the inter-domain effect of a change. The semantic autonomy of each domain implies that a domain can be replaced by another one using different conceptual entities or different implementation without affecting the application.

Domain modeling is a pragmatic approach that avoids heavy up-front investment and long-term design degradation (Bettin, 2004). It provides an incremental path of building sustainable domain-specific assets. Software products built out of domain models are not liabilities designed as one-off systems.[5]

Partitioning the project into domains promotes the development of domain models, with externalized interface definitions serving the goal of active dependency management. This is especially true for a domain model that has been developed by domain experts over several years. The market for such software assets has not matured yet, but it is projected to grow up (Booch, 2004; Mellor et al., 2004).

Structured Classes, Subsystems, and Components

In UML2.0, developers represent the containment hierarchy of a system through structured classes. A structured class may contain internal structured classes and instances, each of which may be further decomposed into structured classes and instances, ad infinitum.

The concept of a structured class is based on decomposition and encapsulation. The parts contained in the structured class define the decomposition aspect. Of course, the structured class is more than a simple container for the parts. A structured class has parts connected with connectors, publishes services via provided interfaces, provides runtime connections via ports, and imposes demands on other structured classes (servers) via required interfaces.

Components are structured classes constituting the primary replaceable units of software. In addition, components have an <<artifact>> section used mainly to specify a unit of deployment, such as a .dll file (dynamic library).

In UML 2.0, a subsystem is defined as a subordinate system within a larger system. The subsystems define the large-scale physical architecture of the system. Subsystems are specialized components (i.e., structured classes) used to decompose a system into parts. A subsystem has one compartment for its specification and one compartment for its realization. Either or both of these compartments could be empty or omitted. Developers can use ports and interfaces to show how subsystems relate to other subsystems. The components and objects within a subsystem cooperate to realize a common set of use cases. Ports and interfaces aid in encapsulating the subsystem's internal structure.

Model Organization

Use Case-Based Model Organization for Small-Sized Systems

For small-sized systems, the package structure can be organized by use cases. The system model is divided into packages of related use cases. This model organization is straightforward and allows tracing requirements easily to model elements. The downside of the use-case-based model organization is that it does not scale up well and encumbers reuse. Developers are forced to reinvent similar classes in different use case collaborations.

A quick remedy is to add a framework package for common elements such as usage points (services) and extension points (classes to subclass in use case packages). Regardless of this remedy, the use-case-based model organization still works well only for small-sized systems. Identifying commonalities in large systems leads to an explosive context and inter-team communication growth.

Domain-Based Model Organization—Large-Sized Systems

We propose to derive the package structure and contents for large-sized systems from the requirements model, the architecture model, and the domain model. The top-level packages of the domain-based system model are:

- System use cases and actors package
- Domains package
- Infrastructure package
- Subsystems package
- Builds package

The system use cases package contains system-level use cases and their actors. The domain package has one sub-package for each domain.

The infrastructure domain is a special type of domain. Infrastructure domains contain services and extension points for system communication and infrastructure, and they are dependent on the selected implementation technology (e.g., RMI/J2EE or CORBA). The infrastructure package contains one sub-package for each infrastructure domain. An infrastructure domain evolves into a self-contained subsystem package, which is assigned to an infrastructure team.

Several production teams may use the classes and components, realizing the services of the infrastructure package.

Each subsystem package contains the classes and components of a subsystem, the largest-scale pieces of system functionality. If any of the subsystem packages is large enough, it can be recursively divided into sub-packages until the size of the leaf packages becomes manageable. A leaf package is a unit of ownership, such as a CI in a CM tool. A subsystem package normally refers to classes of several domain packages.

The builds package is divided into sub-packages, one per prototype to allow for easy management of incremental builds. This package also includes the system test model, such as test cases, test procedures, and test scripts, along with the drivers, stubs, and collaborations realizing the test model.

The domain-based model organization scales up well to large-sized projects for three main reasons. First, subsystem package decomposition can be applied recursively, resulting in subsystems realizing subsystem use cases. Second, classes from different domains may be reused in different deployment settings of the designed system, and third, domain models are assets that can be reused across projects.

The following are the major problems teams may experience with the domain-based model organization. Developers tend to blur the distinction between domain models and subsystem models. There is an overhead associated with maintaining two separate types of models: domain models (logical) and subsystem models (physical).

If reuse is not a primary objective or if the system being designed is a small-sized one, a use-case-based model organization might work better because of the lower added overhead associated with model organization.

Production Elaboration

All strategic decisions for the overall organization structure of the system have been made in the architectural elaboration phase. Production elaboration drills down inside the subsystems to specify the semantic objects whose collaborations deliver the system's structure and behavior.

The demarcation line between architectural and production elaborations is the divide between logical and physical models, and between system-level use cases and subsystem use cases.

In production elaboration, the subsystem use cases are detailed and mapped to collaborations of components and objects. The following technique can assist in elaborating a system-level scenario. The developer starts out with the high-level sequence diagram for the scenario, and then adds lifelines for the instances of the identified classes. In this way, the developer can trace the elaborated scenario back to the original one and verify the design. Communication diagrams are extensively used to represent the stabilized semantic object structure. The measure of goodness at this level is benchmarked by whether the architectural design can realize the usage scenarios defined at the system level.

The described approach scales up well to many levels of subsystem decomposition. As mentioned earlier, the seed developers and production teams serve as customer surrogates.

Next we show how packages are assigned to teams, for example, the architecture team is in charge of developing the system use cases, and therefore "owns" the system use cases package.

Packages	Team
System use cases package	Architecture team
Domains package	Reused if already developed or assigned to the architecture team
Infrastructure domain packages	Infrastructure teams
Subsystem packages	Production teams
Builds package	Integration and test teams

Discussion

Challenges to Agile Outsourcing Projects

Managers of agile outsourcing projects ought to be aware of the following challenges looming ahead.

The architecture team is the key to project success. It is of paramount importance that this team be a good mix of requirements analysts and architects. According to Akella and Dossani (2004), the total percent of technical staff in Indian software firms is about 85% and of MBAs about 4.9%. In contrast, in Silicon Valley, 28% of the Indian IT engineers have MBA degrees, which is a significant comparative advantage. According to the same study, Indian suppliers are mostly "pure-IT" firms, that is, they have limited domain knowledge outside the IT sector, and the average percentage share of architecture/ technology consulting amounts to about 2.9%.

In this context, we see the formation of a balanced architecture team as a major challenge to agile outsourcing projects, for members of this team elicit requirements, build the primary system architecture, and subsequently take the roles of surrogate customers, local managers, integration and test engineers, and communicate with the customer throughout the entire lifecycle.

Inter-team dependencies reduce teams' abilities (especially affected is the infrastructure team) to test code and deliver builds. Production teams should provide early on the infrastructure team(s) with stubs, so that the work of the infrastructure team(s) proceeds smoothly.

In large projects, it is difficult to balance teams' workloads. Adaptive scheduling algorithms in grid computing have proven superior to static scheduling algorithms (Blumofe & Leiserson, 1999; Roussev & Wu, 2001). We propose employee stealing as an adaptive technique for load balancing. An overloaded team (thief) picks up a victim team at random and tries to steal a developer from the victim. If the victim is under-loaded, the stealing succeeds. Otherwise, a new attempt to steal a developer is made.

The random choice of a victim team and a developer makes the technique dynamic and adaptive. The geographical proximity of the offsite teams and the agile method applied are essential prerequisites for employee stealing because they facilitate fast learning.

Employee stealing is a knowledge dissemination technique, which can be viewed as a generalization of pair programming and developer rotation in extreme programming. Employee stealing may be applied as a routine practice, even with balanced projects, in order to improve inter-team communication and coordination, to arbitrate in problems straddling two teams, and to preclude loss of common goal.

With multiple teams, there is always the danger of replicating functionality across teams. The proximity and common culture of the offsite teams work against the chances of duplicating functionality. The participation of the members of the architecture and later integration teams in design reviews and software inspections helps detect replication early. Employee stealing is another effective mechanism to curb duplication.

Why Agility is the Right Path for Indian Suppliers

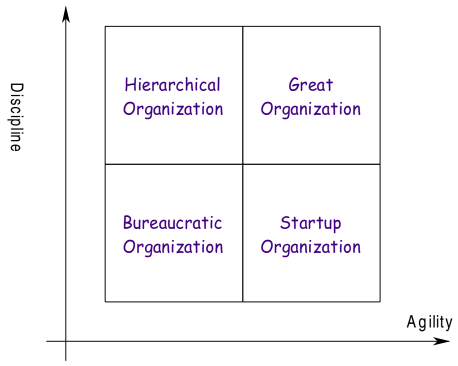
In his book *Good to Great*, Collins (2001) characterizes how self-discipline and entrepreneurial agility contribute to greatness at the corporate level (see Figure 7). Boehm and Turner (2004, p. 156) observe similarity between the patterns of creativity at corporate management and in software development, and apply Collins' method for performing self-assessment to software organizations.

Small Indian firms are mostly startup organizations, and agility improves their chances of survival. Both foreign equity participation companies and major Indian companies aim at the North-East quadrant in Figure 7, and therefore agility will be instrumental in attaining their goals.

The rapid growth of Indian outsourcing suppliers has created a dynamic labor market with a large demand for lead developers for new projects. The average turnover level exceeds 15%. Suffice it to mention that in some cases contracts have placed a liability on the supplier for certain levels of turnover in an attempt to protect the interests of the client. Agile outsourcing could break the impact of high turnover. Several core agile practices (e.g., preference for tacit knowledge, pair programming, and retrospection), along with employee stealing, allow new team members to be brought quickly up to speed.

Agile outsourcing could aid the transfer of domain knowledge to Indian suppliers. With more domain knowledge and the pressing need to communicate directly with customers, agile outsourcing would help organizations move farther away from consultants to directly supply end users.

Figure 7. Good-to-great matrix of creative discipline



The increased revenue per employee, arising from moving up the "value chain," could be partly offset by the increase in the onsite/offsite developers ratio. The small agile team size would allow small and medium enterprises to catch up with foreign equity participation companies and major Indian companies. As a result, small and medium companies could directly participate in complete service and product development.

Executable Domain Models

Below we discuss briefly executable domain models, which are believed to become mainstream in the near future (Booch, 2004; Mellor et al., 2004). Executable models are at the heart of the latest OMG initiative—the Model-Driven Architecture (OMG, 2004).

When working on a model or part of it, a developer must be able to answer the question, "Is this model correct?" Agile methods, through the "prove with code principle" and fast cycle, answer this question almost instantaneously, hence the shorter development times and the better customer satisfaction.

In our view, a problem with agile methods is the level at which the software artifacts are tested. When a system is designed in UML, it has to be tested in UML and not in the underlying source language. While, at times, it is necessary to drill down to the lowest level of detail (e.g., to stress-test the build), the majority of testing ought to be at model level, where the precious business logic and knowledge are captured.

In order for a model to be testable, it has to be executable, which means that the modeling language must have executable semantics. Executable UML (xUML) is a UML profile with precise action semantics (AS, 2001) that allows system developers to build executable system models, and then to map these models using code generators to source code for a target platform (Mellor & Balcer, 2002).

Executable domain models are expected to play an increasing role in software production. Executable domain models fit naturally with the domain-based model organization of agile outsourcing projects.

Conclusion

In the late 1990s and early 2000s, agile methods took the imagination of software developers by storm. This fact clearly indicates that heavyweight methods have not been embraced wholeheartedly by developers and managers alike, and are found either impractical or costly (or both) in many environments. Driven by low labor costs for commensurate quality in India, and also by stagnant revenues in developed countries, firms have been increasingly outsourcing software production to India. In this chapter, we introduced a novel project structure creating agile conditions for large outsourcing software projects. The proposed structure is tailored to the needs of the Indian outsourcing suppliers. Several assumptions make agile outsourcing Indian-unique. First, we assume that there are large pools of qualified developers residing in one geographical area and even in one city. Second, the outsourcing country has grown a sustainable software industry that is supported by advanced information and telecommunication technologies, and led by talented managers, allowing it to secure large software projects.

We showed how to slice a large software project into multiple agile projects. We proposed to separate the development activities to R&D activities, carried out onsite, close to the client, and production activities, carried out offsite in India. The onsite, architecture team starts work on the project. In cooperation with the client, the architecture team develops the high-level use case model of the system and completes an architectural prototype with the strategic decisions for the overall system structure. The agile offsite teams are seeded with members of the architecture team and start functioning toward the end of the architectural elaboration. The "seed developers" transfer domain expertise to the supplier's site and act as customer surrogates to help reproduce agile conditions offsite. Outsourcing the entire set of production activities retains interactivity at the offsite location.

The domain-based package structure of the system model is organized by domains, subsystem use cases, and builds. This model organization scales up well to large-sized projects, because the subsystem packages can be recursively divided into smaller ones. The domain packages facilitate reuse of components and classes from different subject matters. The domain models could become software assets reused across projects.

To balance the teams' workloads and avoid replicating functionality by different teams, we proposed a new adaptive technique called employee stealing. Employee stealing is instrumental in disseminating tacit knowledge about the project and in lifting up the developers' expertise. It also improves the inter-team communication and coordination, and precludes the loss of common goal.

We plan on combining aspects of domain engineering and executable domain models in xUML with agile outsourcing to raise the level of abstraction and reuse, and to automate the repetitive tasks in the software process. We are especially interested in employing techniques preventing architectural degradation in large systems and in building "software factories" that produce families of sustainable software assets using highly automated processes. We would like to support the emerging "supply chains" for software development, which enables mass customization. To fully support domain-driven design, we ought to differentiate clearly between building a product platform and designing an application. Gearing executable domain models to the proposed agile outsourcing methodology will result in making agile outsourcing even more agile.

References

Agile Alliance. (2001). Agile Alliance manifesto. Retrieved from *www.aanpo.org*

Akella, R., & Dossani, R. (2004). Private communication.

Arthur, B.W. (1994). *Increasing returns and path dependence in the economy*. Ann Arbor: University of Michigan Press.

AS. (2001). UML actions semantics. Retrieved from *www.omg.org*

Beck, K. (1999). *Extreme programming explained: Embrace change*. Boston: Addison-Wesley.

Bettin, J. (2004). Model-driven software development: An emerging paradigm for industrialized software asset development. Retrieved from *http://www.softmetaware.com*

Blumofe, R.D., & Leiserson, C.E. (1999). Scheduling multithreaded computations by work stealing. *Journal of ACM*, *46*(5), 720-748.

Boehm, B., & Turner, T. (2004). *Balancing agility with discipline*. Boston: Addison-Wesley.

Booch, G. (2004). MDA: A motivated manifesto? *Software Development*, (August). Retrieved from *http://www.sdmagazine.com*

Brownsword, L., & Clements, P. (1996). *A case study in successful product line development.* Technical Report CMU/SEI-96-TR-035, Software Engineering Institute.

Collins, J. (2001). *Good to great*. New York: HarperCollins.

D'Costa, A.P. (2003). Uneven and combined development: Understanding India's software exports. *World Development*, *31*(1), 211-226.

Dossani, R., & Kenney, M. (2003). Went for cost, stayed for quality?: Moving the back office to India. Asia-Pacific Research Center, Stanford University. Retrieved from *http://APARC.stanford.edu*

Evans, G. (2004). Agile RUP: Taming the Rationa® Unified Process®. In B. Roussev (Ed.), *Management of object-oriented software development*. Hershey, PA: Idea Group Inc.

Henderson-Sellers, B. (1996). *Object-oriented metrics, measures of complexity.* Englewood Cliffs, NJ: Prentice-Hall.

Jacobson, I. (1987). Object-oriented development in an industrial environment. *ACM SIGPLAN Notices*, *22*(12), 183-191.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified software development process.* Boston: Addison-Wesley.

Kruchten, P. (2004). Scaling down large projects to meet the Agile "Sweet Spot." *The Rational Edge*, (August).

Mellor, S.J., Kendall, S., Uhl, A., & Weise, D. (2004). *MDA distilled.* Boston: Addison-Wesley.

Mellor, S.J., & Balcer, M.J. (2002). *Executable UML: A foundation for Model-Driven Architecture*. Boston: Addison-Wesley.

Nasscom. (2004). Indian software and services exports. Retrieved from *www.nasscom.org*

OMG. (2004). OMG Model-Driven Architecture. Retrieved from *http://www.omg.org/mda/*

Outsourcing Institute. (2004). Retrieved from *http://www.outsourcing.com*

Paulk, M.C., Curtis, B., Chrissis, M.B., & Weber, C.V. (1993). Capability maturity model, version 1.1. *IEEE Software*, *10*(4), 18-27. Software Engineering Institute. (2004). Retrieved from *www.sei.cmu.edu*

Pollice, G. (2001). RUP and XP, part I: Finding common ground, and part II: Valuing differences. *The Rational Edge*.

Pollice, G. (2004). RUP and eXtreme Programming: Complementing processes. In B. Roussev (Ed.), *Management of object-oriented software development*. Hershey, PA: Idea Group Inc.

Roussev, B., & Wu, J. (2001). Task scheduling on NOWs using lottery-based workstealing. *Annual Review of Scalable Computing*, *3*, World Scientific, Singapore University Press.

Royce, W. (2002). The case for results-based software management. *The Rational Edge*, (June).

SEI. (2004). Carnegie Mellon Software Engineering Institute, Product Line Practice. Retrieved from *www.sei.cmu.edu/productlines/*

Toth, K., Kruchten, P., & Paine, T. (1993). Modernizing air traffic control through modern software methods. *Proceedings of the 38th Annual Air Traffic Control Association Conference*, Nashville, TN.

Endnotes

1 In domain modeling, a software asset is anything from models, components, frameworks, generators, to languages and techniques.

2 Strategic assets are the software assets at the heart of a business—assets that grow into an active human- and machine-usable knowledge base about a business and its process.

3 A software liability is software that is cost burden—that is., software that costs more than it is delivering to the business (Bettin, 2004).

4 Mass customization meets the requirements of heterogeneous markets by producing goods and services to match individual customer's needs with near mass production efficiency.

5 One-off systems have a very short lifespan and are too expensive for most practical purposes.

Chapter VI

User Requirements Validation and Architecture Discovery through Use Case Invariants and Model Animation

Boris Roussev
University of the Virgin Islands, USA

Yvonna Rousseva
University of the Virgin Islands, USA

Abstract

This work proposes a technique for requirements validation and logical structure discovery, compatible with evolutionary process models. The technique is based on a conservation law, called business value invariant, which quantifies the exchange of business objects between a system and its environment. With the invariant, the logical class structure of the designed system is algorithmically derived from its use case model. To *validate that the modeled requirements and derived structure faithfully reflect the user requirements, the behavior of the constructed prototype is projected on the business objects exchanged on the system's boundary, and the projected behavior is animated with a labeled transition system analyzer. The model animation approach explicitly describes the interface between the system and its environment, and through OCL pre- and post-conditions, it distinguishes between system and environment responsibilities. The animated prototype links the outwardly visible "interobject" behavior to the information structures and the behaviors of the comprising parts, or "intraobject" behavior. Unlike formal notations based on logic, the proposed approach does not preclude the owners of the problem from taking part in the problem-solving process, that is, the knowledge locked in the prototype can be validated. The proposed prototyping technique acts as a discursive communication instrument, bringing the worlds of clients and developers a step closer.*

Introduction

Software development is a problem-solving activity where a problem has been identified and a software system is commissioned to address this problem. Brooks (1995) points out that "the hardest single part of building a software system is deciding precisely what to build"; in other words, the principal challenge faced by a development team is the elicitation of precise user requirements.

Requirements engineering (RE) is a branch of systems engineering concerned with the elicitation, modeling, validation, and management of evolving user requirements. The activities in requirements engineering both situate and orient the software development effort to a real-world problem and give it a narrow compass toward satisfying the goals of the various system stakeholders.

The results from a recent, and so far the only, significant field survey (Neill & Laplante, 2003) on RE practices indicate that 33% of companies do not employ any methodology for requirements modeling and analysis. Out of the remaining 67%, only 7% use formal techniques, compared to 51% using informal natural language representations. In an earlier study, Nikula, Sajaniemi, and Kälviäinen (2000) report that none of the 15 participants they surveyed had a requirements management tool in use. Against this backdrop, the conclusions of the CHAOS report that most software project failures are caused by requirements-related shortfalls such as lack of user input, incomplete requirements, changing requirements, unrealistic expectations, and unclear objectives should not come as a surprise (Standish Group, 2003).

Adopting a systems view in RE is of utmost importance, because software cannot function in isolation from the environment in which it is embedded. RE is a multi-disciplinary, human-centered process. In this chapter, we draw upon a variety of disciplines including systems engineering, object models, finite state machines (FSMs), logic, linguistics, communication theory, semiotics, and cognitive science, to validate user requirements.

The most popular RE approach to modeling user requirements is the use case model (Jacobson, 1987). Over 50% of the software firms use scenarios and use cases in the requirements phase (Neill & Laplante, 2003). Use cases are popular even among teams that do not use object-oriented (OO) technologies.

Requirements validation is the act of showing that the elicited requirements provide an accurate account of stakeholder goals. Prototyping (Davis, 1992) is a class of requirements validation and elicitation techniques used in environments marred with uncertainty. Even within the waterfall model, about 60% of the teams perform some sort of prototyping, with half of them applying evolutionary prototyping and only one-fourth applying throwaway prototyping. Requirements validation (with formal methods or with prototyping) is difficult for two reasons: one philosophical and one social.

To be recognized as an engineering discipline, requirements engineering had to adopt a logical positivist approach, meaning it had to work under the assumption that there is an objective external reality to which the designed software must relate. This, however, poses the question of what is true in this objective world. On the one hand, we have Karl Popper's thesis about the limits of empirical observation (essentially, a theory cannot be proven correct via observation, but only refuted). Derrida (1967) and Lacan (1977) approached the question of what is true through language and questioned the idea of logical positivism. They showed that truth is language-driven and language-biased. For RE, this translates into the problem being dependent on the notation used to describe it. Furthermore, since validating requirements is a dialogic process between business stakeholders and analysts, an analyst could ascribe new meanings to the business stakeholders' requirements, a process called counter-transference or projective introspection (project one's own desires and agendas onto others). The bottom line is that we cannot draw a sharp demarcation line between, on the one hand, validating requirements and, on the other hand, convincing business stakeholders about what their requirements are.

The social problem in requirements validation is related to negotiating an agreement satisfying the competing and often conflicting needs of the various stakeholders (Briggs & Grünbacher, 2002).

The constructive core of UML (2004), the language we use to create our truths, includes class diagrams and FSMs. Since their introduction in the late 1980s, use cases have proven to be an immensely popular software development tool. This clearly indicates a shift in emphasis, one departing from modeling information flow and system state towards modeling stakeholders' goals (Dardenne, Lamsweerde, & Fickas, 1993) and scenarios that illustrate how these goals can be attained. Use cases organize requirements (e.g., tens of usage scenarios) around a common goal. The use cases describe dialogically the outwardly visible system capabilities, and in addition, they are used to decompose the system into conceptual subsystems.

A scenario, often represented on a sequence diagram, describes a collaboration of relevant conceptual entities and one or more actors working together to achieve a certain goal. This is called *interobject* behavior. In contrast, an FSM describes the behavior of a single object, called *intraobject* behavior. Harel, the inventor of statecharts, refers to the stark difference between interobject and intraobject behaviors as the grand duality of system behavior, and states that we are far from having a good algorithmic understanding of this duality (Douglas, 2004). Not only do we not have the means available for deriving one view from the other, but we also do not exactly know how to test efficiently if two such descriptions are mutually consistent.

In situations with a great deal of uncertainty, prototyping (Davis, 1992) can be used to elicit and validate requirements. Developing a prototype is an aggressive risk management strategy. Ideally, the upfront investment in designing the prototype will be offset by reducing the probability of risk exposure later in the development lifecycle. An early prototype provides feedback opportunities for business stakeholders and can be further used to provoke discussions or think-aloud protocols (Shaw & Gaines, 1996), thus acting as a driver for information gathering. A prototype facilitates the requirements analysis and validation, and may give an insight into the logical system architecture. Depending on the notation used, a prototype can make possible the application of model checking or other V&V techniques.

The objective of this chapter is twofold. First, it has the purpose of offering a precise, user- and developer-friendly technique for validating user requirements through animating a system prototype, and second, it aims at providing a lightweight process extension addressing the grand duality problem: to derive algorithmically the system logical class model system (intraobject state and behavior) from interobject behavioral descriptions.

The rest of the chapter is structured as follows. First, we take the view of analysis as learning aided by interactive animation. Next, we present a discursive communication model of the process of eliciting user requirements. Then, we present the business value invariant (BVI) of a use case, and a process extension for requirements and architecture validation. After further discussion, we outline plans for future work and conclude.

Analysis Viewed as Learning Aided by Animation

During and after the construction of a model, the modeler wants to evaluate the model's quality. There are two types of model review pertinent to RE: model animation (Gravel & Henderson, 1996) and model verification (Easterbrook et al., 1998).

Model animation is a discrete, even simulation, and as such, it is a form of partial validation. Model animation is a quick and intuitive way of checking the model that allows for business stakeholders to take part in the validation activity. Modelers and end users can examine example behaviors, communications, and interactions.

The benefits of animating the use case realization can be understood better if use case analysis is viewed as a cognitive process. The use case model specifies what needs to be accomplished. The job of the system analyst is to elicit and communicate this knowledge to all stakeholders by using a modeling notation and technique. From this point of view, RE is a process of accumulating valid information about the problem domain and of communicating it clearly to developers and clients alike. This makes the process of requirements development analogous to the process of learning (Gemino & Wand, 2003). From this perspective, the usefulness of the modeling technique should be evaluated based upon its ability to represent, communicate, and develop understanding of the application domain.

Cognitive sciences have produced a variety of models providing insight into the mechanics of learning. Although theories differ in knowledge representation, they all agree on two elements of learning, namely the use of analogical reasoning and the effect of prior knowledge. There has been a general understanding that the notion of metaphor can be brought to bear on the questions of how prior knowledge organizes new learning and of how analogical reasoning operates. Carroll and Mach (1985) define metaphor as a "kernel comparison statement whose primary function in learning is to stimulate active learner-initiated thought process." Gentner (1998) defines metaphor as "a kind of similarity in which the same system of relations holds across different objects that supports further inferences." These two definitions shed light on the function and mechanics of metaphor. In active learning, the metaphor provides an orienting framework guiding learners to hypothesize and verify new knowledge. The metaphor "seeds" the constructive process of knowledge acquisition.

For centuries, graphic models have been used to present spatiovisual entities and concepts, like maps and building blueprints. In software development, graphic models have been used to represent things that are metaphorically spatiovisual, like class diagrams and state machine models. The underlying assumption is that a graphic model enhances the developers' ability to think and to apply the power of spatial inference to reasoning about software artifacts, hoping to achieve an increase in productivity and software quality. Graphic models are also used to record information (blueprints), in other words, to externalize internal knowledge (i.e., memory and processing off-loading).

In this work, we propose to validate the system requirements by animating the dialogue between the end users modeled as actors and the designed system modeled as use cases. Model animation can facilitate requirements validation because, according to the congruence principle for effective graphics, the content and format of the graphics should correspond to the content and format of the concepts being conveyed. This principle suggests that animated graphic models should be more effective than static graphic charts in presenting change over time. However, research results show that this is not always the case (Tversky, Morrison, & Betrancourt, 2002). Animated graphics have a proven success of outperforming static ones only when the animated graphics convey more information, or involve interactivity or prediction.

One possible explanation for the ineffectiveness of animation is the violation of the second principle of effective graphics, called apprehension principle. According to this principle, graphics should be accurately perceived and appropriately comprehended. If the animation is too fast, the receiver (learner) cannot accurately perceive the conveyed information. Overwhelmed by the continuous pictorial information flow, the receiver has no time to process and comprehend the transmitted information.

Only judicious use of interactivity is known to cancel out these negative effects by allowing the receiver to control the speed of the information flow and the evolution of the system under study. Pausing, starting, and replaying an animation all allow for re-inspection and focusing on specific sequences of actions. Overcoming the negative effects of inaccurate perception and inappropriate comprehension results in bringing out several key advantages of animation.

Animations are a richer source of information than static graphic models. Static graphics portray a relatively smaller number of coarse segments, whereas animations portray both the coarse and the fine segments. Thompson and Riding (1990) demonstrate that animation facilitates learning when it presents the fine-grained actions that static graphics do not present.

Prediction is another factor known to facilitate self-initiated learning (Hegarty, Quilici, Narayanan, Holmquist, & Moreno, 1999). In prediction, participants anticipate outcomes, and then view animated graphics to check if their predictions were fulfilled. With model animation, analysts and clients learn actively about the problem domain by building hypotheses (predictions) and by testing the validity of these hypotheses.

Prototyping as Discursive Communication Model

In this section, we consider the mechanisms through which prototyping elicits knowledge about the organizational context in which the system is embedded.

Client needs elicitation and system design are both communication-intensive and communication-driven processes. The expression of clients' needs is mediated through language; therefore, developers need to examine the clients' language, and be attentive to its slippery, unstable, and indeterminate nature. As meanings are not immutably fixed, getting acquainted with the clients' context should bring about a clearer understanding of clients' needs. A change in context duly calls for a change of meaning. The use case model does not explicitly account for this context-sensitive meaning of user requirements.

Very abstractly, a client communicates a requirement (message) to an analyst in natural language in the dialect of some problem domain. Correctly understanding the intention of a message not only requires prompt feedback from the receiver, but it also demands an ongoing mutual communication between sender and receiver. Given the dynamics of this client-analyst dialogue, the receiver can, and actually does, affect the sender's context, thus altering the meaning of the sender's messages. The constantly evolving meaning necessitates considering the production of meaning (i.e., making meaning in discourse).

The main factor for the success of OO analysis is the improved communication between the interlocutors, or stakeholders in our case in point. Object-orientation narrows the semantic gap between the problem and the solution domains by encouraging a shared vocabulary between end users and developers. The shared vocabulary allows for information-rich communication between interlocutors. This results in improved traceability, and ultimately allows clients to validate the software artifacts being produced.

Even assuming that it may be possible for the client to present the system analyst with un-contradictory and complete user requirements, the analyst would still be unable to fully comprehend the user requirements, because doing so requires that the analyst consider the requirements in context, and the context in systems development is constituted as domain knowledge.

The human mind cannot interpret things in the world as unrelated bits and pieces because individual entities and concepts look different in different contexts—that is, the human mind needs to situate things in the context of their production. For example, we could recall the famous duck-rabbit puzzle, shown in Figure 1, where the interpreter cannot tell whether the picture shows a duck or a rabbit without taking into account the picture's context.

Figure 2 shows a model of linguistic communication, devised by Roman Jacobson. The client uses a code (common or shared vocabulary) familiar to both analyst and client to express the message. The message has a context or referent and is transmitted through a contact, a medium such as live speech.

In order for the message (the requirements) to be interpreted correctly by the analyst, the analyst must be familiar with the client's context. By context we mean the environment in which the client operates. The context is constituted of tangible and conceptual entities and their relationships. Conceptual entities include, but are not limited to, the client's beliefs about other entities and the client's beliefs about the beliefs of the other clients. A message can have more than one context. This view of the meaning of a message (or more generally a

Figure 1. Duck-rabbit puzzle

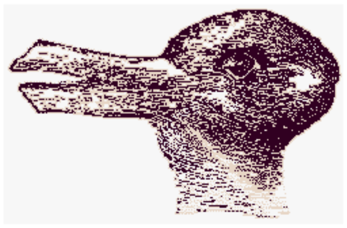
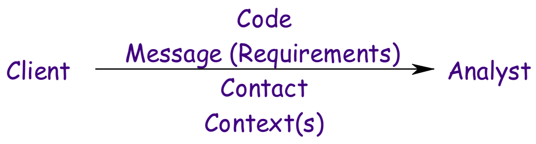


Figure 2. Roman Jacobson's model of communication



text) is maintained by the theory of hermeneutics. The hermeneutic approach emphasizes the central role of the message. The context is there only to inform about the meaning of the message. The context is independent of the text and is considered to be static and flat (no multiple layers). The context can be viewed as constituting the rules, norms, and assumptions governing the environment of the message.

One last note worth highlighting is that the message is a consistent and indispensable part of the context. This key role makes message interpretation spiral. The meaning of preceding message constituents might need revision in light of subsequent message constituents; in other words, antecedents explain successors, which in turn shed light on the antecedents, or the so-called hermeneutic loop.

Even though Jacobson's model brings up the legitimate issue of the all-important context of a message, this model is still found wanting in resolving the interpretation ambiguity arising from the recursive nature of the problem. The client still needs to send the context of the context of the message, and so on.

The main disadvantage of the hermeneutics theory is the inability to express the dynamic nature of the message. A message is not simply part of one or more contexts, but rather part and parcel of an interactive deictic process, a discussion, in which many agents participate. In discourse analysis, to understand the meaning of a message, one should know the goals and responsibilities of the message sender. In discourse analysis, to understand the meaning of a message, which is never immutable, one should know the goals and responsibilities of the message sender. Therefore, to understand and correctly interpret a message, which is always caught up in the process of becoming itself, the receiver must inquire about the development that has led to the message creation. Because the receiver must become an active agent in the discourse, the receiver, in turn, influences the sender in this ongoing, dynamic course of discussion.

According to Jacques Lacan's formula of communication, "Human language constitutes a communication in which the emitter receives from the receiver his own message in an inverted form" (Lacan, 1977, p. 329). The way we read Lacan, "inverted form" of a message should be understood as the posing of questions.

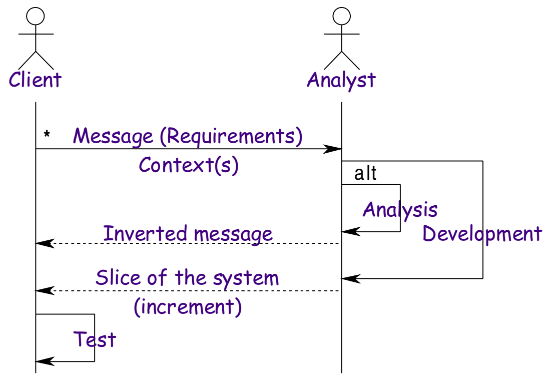
The impact of the system analyst on the client is on two planes. First, the system analyst helps clients articulate their needs, and, second, by building the information system, the analyst changes the client's context, the nature of work, and the organization.

Helping clients articulate their needs is not trivial. The client's needs are often displaced by the strong desire for immediate satisfaction or consummation, adding executable semantics to the ordinary need. Kruchten (2000) gives the following example. Instead of describing his/her need to communicate better, a client states, "I want automatic e-mail notification." The real need (better communication) is displaced by the strong desire for immediate satisfaction.

Schleirmacher, credited with founding hermeneutics as a discipline in its own right, claimed that a successful interpreter could understand the author of the message as well as, or even better than, the author understood himself, because the interpretation highlights hidden motives and strategies (agendas), also referred to as projective introspection or empathy. This, however, could be egregiously counterproductive. Psychoanalysis would call this counter-transference—the interpreter ascribes to the author his or her own repressions and desires, thus falling in the "intentional fallacy" trap. The analyst has to resist this as much as possible. The danger of counter-transference increases with developers who lack domain knowledge, for example, computer engineers without experience in the particular problem or functional domain.

The second major role of the analyst, and of the whole development team for that matter, is to become part of the client's discourse. The dialogue is usually

Figure 3. Discourse involving clients and analysts



initiated by the client. The analyst, or the development team, respectively, responds with an increment, a piece of executable functionality, produced in one iteration. Then the client tests the increment, and the communication cycle repeats. In the testing subprocess, or the ensuing discourse between analyst and client, the client may become aware of a compelling necessity to redefine his/her needs.

The sequence diagram in Figure 3 shows the discourse between client and analyst in progress (i.e., the spiral of message exchange).

The asterisk denotes iteration. The alt operator expresses branching. The feedback loop in the communication model, along with the subsequent testing of the analyst's understanding by the client, narrows the gap between the floating signifier (articulated needs) and signified (ever evolving real needs).

The prototype should be seen as the analyst's materialized and animated understanding of the stakeholders' needs. In Figure 3, the building of the prototype is modeled with messages "analysis" and "development."

The process described by the discursive communication model is an interminable one. It continues after the software system has been delivered and accepted by the client. This part of the process is referred to as system maintenance. From a certain point onward, the aging starts taking its toll, and a small user requirements change entails disproportionate implementation complexity. From that threshold onward, it is economically justifiable to have a complete system rewrite, to start all over again.

The communication model in Figure 3 can be thought of as generalization of the widely cited "hump" diagram (Kruchten, 2000) and spiral model diagram (Boehm, 1988).

The Business Value Invariant of a Use Case

We will use the E-ZPass system described in Chapter 1 as a running example.

User Requirements and Model Transformation

User Requirements Model

A user requirements model should possess the following characteristics. First, it must explicitly describe the interface between the system and its environment, and then distinguish between system and environment responsibilities. Since a system is evaluated with respect to its usage in its environment, the specification should provide a basis for validating the system's role in that environment. Second, the user requirements must be justified. A justification can be achieved if each constraint placed on the system is aligned with and traceable to the goals of the various stakeholders. Third, the system specification should be given in a form that is accessible to both developers and stakeholders, and should provide all the information needed to design the system and no more, hence no under- or over-specification (Parnas, 1972; McDermid, 1994). An over-specification implies premature design decisions, whereas an under-specification means that the designers do not have sufficient information, thus degrading the specification's usefulness as a contract.

The second characteristic, namely the justified user requirements, suggests that a design contract cannot be constructed without considering the goals of the stakeholders for the system. However, if the requirements model explicitly describes the stakeholders' goals, it may include contextual information. The contextual information describes situations that are outside the system's scope, causing confusion as to whether the system should or should not provide a means to deal with the situation. Therefore, explicitly describing the stakeholders' goals would conflict, by and large, with the objective of attaining a "designer-friendly" specification. This reveals two disparate requirements regarding the requirements model: on the one hand, to provide a valid description of the stakeholders' goals, and on the other hand, to serve as a precise description of the system to be designed. To appease these conflicting requirements, the system's functional description must be sufficiently user- and developer-friendly. In addition, it must facilitate effective communication among all stakeholders. The requirements model will be user-friendly, if the modeling language employed is expressive, precise, and understandable to all stakeholders. In this respect, it is of utmost importance to express requirements at interface level using declarative specifications, such as Catalysis' OCL pre- and post-conditions (D'Souza & Wills, 1998) and invariants (Roussev, 2003).

We assume that the system functional specification is represented as a set of use cases and that an evolutionary development process has been adopted.

The proposed process extension builds an executable model of a use case, which can be animated jointly by all stakeholders to validate the requirements and the initial logical decomposition (analysis level architecture). The executable model is generated using the BVI extension to use cases (Roussev, 2003). The BVI facilitates the algorithmic transformation of a use case to a set of FSMs and class models. The precision of the generated models allows for model animation. Model animation projects (or reverse-engineers) the analysis level architecture to communication events occurring on the system-environment boundary. This makes it possible for end users and developers to validate the user requirements and the class model generating the animated requirements early in the development process, and to avoid costly backtracking.

Model Mapping

Shannon and Weaver (1949) define information as a reduction in uncertainty. For a system to be of value to its users, it has to reduce the user's (we equate receiver with user) level of uncertainty. To ensure that the software solution is adequate to the problem at hand, modern software processes, such as RUP (Kruchten, 2000) and Agile (Beck, 1999), are requirements-driven and actively engage business stakeholders in the development process.

Furthermore, the development team has to demonstrate that the solution satisfies the user requirements, by proving the system (code view) and the requirements models isomorphic. A proven technique for bridging the gap between two models is to use a series of transformations and/or refinements preserving the properties of interest, such as observational equivalence (Milner, 1989) and Model-Driven Architecture (MDA) with Executable UML (xUML) (Mellor, Scott, Uhl, & Weise, 2004). For instance, the success of RUP is predicated on the fact that it is defined as model refinement from more abstract to more detailed models, whereas that of MDA is attributed to model transformation. Refinement and transformation allow for user requirements to be traced to the implementation modules, and therefore ensure that user needs be met, thus making effective change management possible.

The Business Value Invariant

OO methods are weak in guiding the transition from informal requirements towards formal model (Wieringa, 1998). The discontinuity in model transformation is due to the lack of mature user-friendly formal models for requirements modeling. As a result, round-trips between RE and analysis are encumbered, which in turn makes the validation of software architectures against systems requirements hard to accomplish.

The BVI is an extension to the use case model and the underpinning of a development process deriving the logical class model (analysis architecture) of a system through a clearly defined sequence of model transformations. The invariant is a design contract defined over business domain objects exchanged between actors and system in the course of a use case performance. The behavior of each actor and counter-actor—the agent within the system responding to actor-initiated transactions—is bound by the design contract and is described by an FSM (non-hierarchical statechart). Each state in an FSM is characterized by gaining or giving away a business object. The logical class model is obtained as the composition of the actors' and counter-actors' FSMs. The BVI approach fills the gap between the outwardly visible system behavior as offered by use cases and the "first cut" at system architecture, the analysis class model. The BVI abstracts the detail of the system architecture concerned with data representation and resource allocation. It focuses on the interaction between the environment (actors) and the system (use cases).

Actors, Counter-Actors, and Design Contract

From the perspective of an actor, a use case performs something that adds value to the actor, such as calculating a result, generating a new object or changing the state of another object. Isaac Newton observed that contact forces are always exerted by one body (initiator) on another body (respondent). If we imagine that the actor initiating the use case instance is the body exerting the contact force, then there must be at least one respondent on the receiving end.

Definition 1. The respondent trading values with the actor initiating the use case instance is called *counter-actor* class. Actors and counter-actors are collectively called *agents*.

For each use case, there exists a conservation law (i.e., an invariant).

Axiom 1. The joint distributed count of business values exchanged between the actors and the counter-actors in the course of a use case execution is constant. This property is called the *business value invariant* of a use case.

The invariant is instrumental in identifying classes and OCL constraints. Jacobson (1992) gives two criteria helpful in determining the scope of a use case, the key phrases being "resulting value" and "participating actor." He went a step further by saying that "in some cases, actors are willing to pay for the value returned," but he stopped short of generalizing and defining the exchange of values. Below, we show how the exchange of business objects in the course of the execution of a use case instance is defined by a business value invariant and how this invariant is used to discover the system structure.

The use case diagram for the E-ZPass system is shown in Figure 4. Below we show the description for use case Register Vehicle (the reset is given in the Appendix).

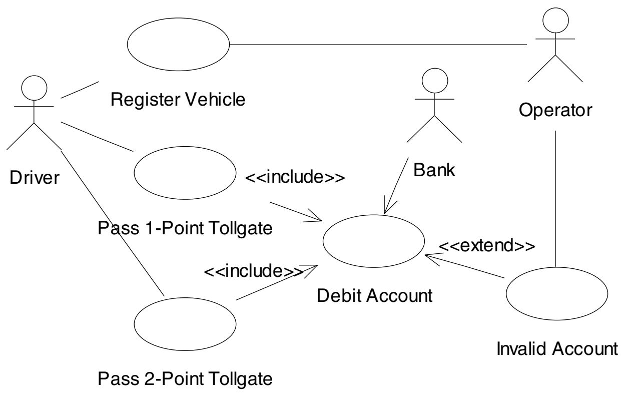
Use Case: Register Vehicle

Description: This use case describes vehicle registration.

Actors: Driver, Operator, and Bank

Basic Flow

Figure 4. E-ZPass use case diagram



1. A driver provides an operator with contact information, including name and mailing address, the vehicle type, and the vehicle's registration. In response, the operator stores the information and prompts the driver to provide a valid credit card, which will be debited automatically at every E-Z lane.
2. The driver provides a credit card. The system verifies the card with the bank.
3. The system generates a unique ID, provides the driver with a gizmo with the generated ID, and stores the credit card information and the gizmo ID associated with the registered car.

Exceptional Flow of Events:

- The driver can cancel the registration at any point. The system rolls back to its state prior to the registration.
- In step (2), if the verification fails, the registration is cancelled.

Pre-Conditions: The driver has a valid vehicle registration and a valid credit card.

Post-Conditions: The driver receives a gizmo with a unique ID. The driver's personal information, vehicle, and gizmo are entered into the system.

Driver is the actor initiating the execution of the use case. The value gained by Driver is a gizmo with a unique ID. The value given away is the contact, vehicle, and credit card information. The respondent verifying the credit card is Bank. Another actor, named Operator, is the respondent in charge of verifying the vehicle registration and the driver's license. It is not clear from the use case model where the Tag ID gained by Driver is coming from. Who is giving away this business object? To balance the invariant, we introduce a counter-actor, called TagGenerator, which is giving away this tag ID. The invariant for the use case is shown in Figure 5. Note that TagGenerator is a counter-actor. It was discovered using the invariant. Prior to introducing it, the invariant for the use case was not satisfied, that is, the count of business objects gained or given away by actor Driver did not balance with the business objects gained or given away by actors Operator and Bank.

At the highest level of abstraction, the evolution of an agent is defined as a contract. A contract is an exact specification of the interface of an object. The supplier (server) knows the conditions under which its services will be provided. If the client does not live up to the expectations, the supplier is not held responsible. The client makes sure the conditions are met. The rights of each party can be mapped directly to the obligations of the other one. The business values given away (obtained) by an agent are described in the notes attached to the agent on the left (right) side (see Figure 5). These values specify the state of an agent before and after the execution of a use case, or in other words the agent's pre- and post-conditions. The invariant for Register Driver expresses the fact that the joint distributed count of business objects before the execution of the use case (described in the notes attached to the left) is the same as the joint distributed count of business objects after the execution of the use case (the business objects described in the notes attached to the right).

The invariants for use cases PassSingleTollGate and PassTwoPointTollgate are given in the Appendix.

Figure 5. BVI for use case Register Driver

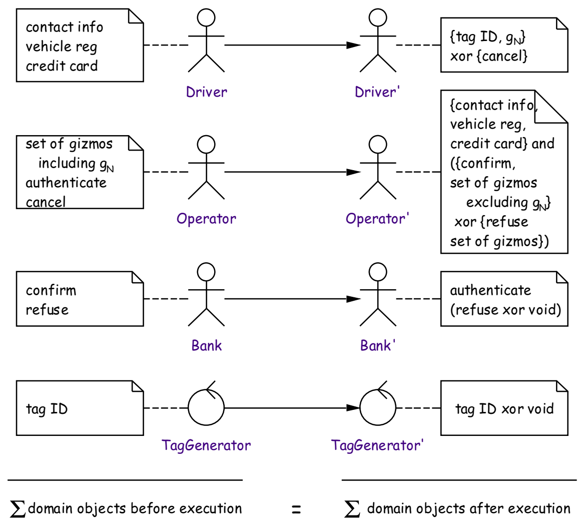
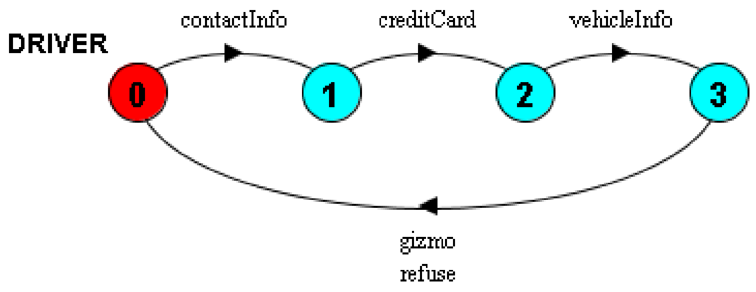

contact info
vehicle reg
credit card
Driver
Driver'
{tag ID, g_N} xor {cancel}
set of gizmos including g_N
authenticate
cancel
Operator
Operator'
{contact info, vehicle reg, credit card} and ({confirm, set of gizmos excluding g_N} xor {refuse set of gizmos})
confirm
refuse
Bank
Bank'
authenticate (refuse xor void)
tag ID
TagGenerator
TagGenerator'
tag ID xor void
∑domain objects before execution
=
∑ domain objects after execution

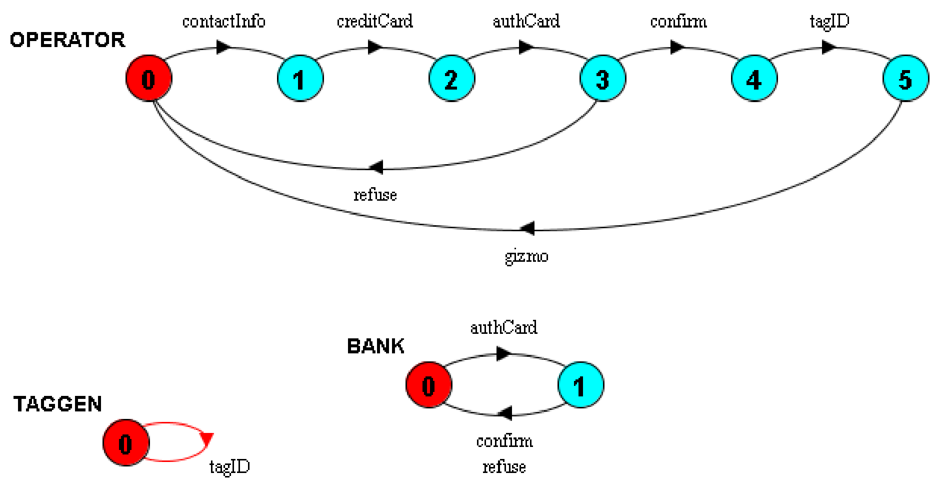
Figure 6. FSM model for actor Driver



Deriving Class Diagram and OCL Specification for a Use Case

Having identified all agents of a use case, we describe the evolution each of them undergoes by an FSM. An FSM model can represent a system at an arbitrary level of detail. In analysis, it is mandatory that we stay out of design. This requirement is met by associating each state with a business object that is either obtained or given away as the event associated with the transition leading

Figure 7. FSM models for Operator, Bank, and TagGenerator



to the state. Figure 6 shows the FSM for actor Driver derived from the main flow of the use case.

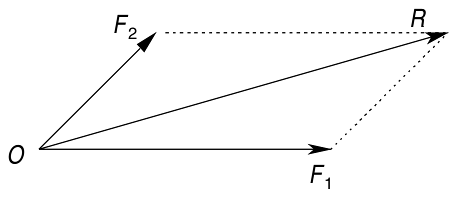
To each agent gaining a value corresponds an agent giving away that value and vice versa. For convenience's sake, we have numbered the states in each model, so that they can be referred to and unambiguously distinguished from other models. The FSMs for Operator, TagGenerator, and Bank are shown in Figure 7.

Reduction of a Set of FSMs to a Class Diagram

The parallelogram law for the composition of forces was stated by Newton (see Figure 8). F_1 and F_2 are called concurrent forces. R is called the resultant, and it is equivalent to the combined action of the two forces. Milner (1989) was among the first to apply the parallelogram of forces in computer science. Unlike Milner's, the composition we propose derives the static structure of the system from the concurrent FSM descriptions.

Maciaszek (2001) observed that there is an interesting dichotomy with regard to actors. Actors can be both external and internal to a system. They are external because they interact with the system from the outside. They are

Figure 8. Parallelogram of forces



internal because the system may maintain information about actors, so that it can knowingly interact with the external actors. Hence, the specification needs to hold two models related to an actor—a model of the actor and a model of what the system records about the actor. It is important to note that Jacobson (1992) did define actors as stereotyped classes. However, to the best of our knowledge, the definition is for the purposes of generalization and instantiation, as well as for uniform notation; for example, actor Manager who inherits from actor Clerk can initiate instances of all use cases that a Clerk can and, in addition, some others. Like external entities in context diagrams, actors are meant to model interactions and communications between the system and its environment.

We use the actor's dichotomy as a heuristic for discovering entity and controller classes. The system must create an object of class Driver (if the object already exists, it is linked) for each vehicle registration. The agent FSMs are considered one at a time. The order of their processing is not significant. We select an FSM not processed yet, and then we create a class for the agent and stereotype it

Figure 9. Class diagram generated from the FSM model for Driver

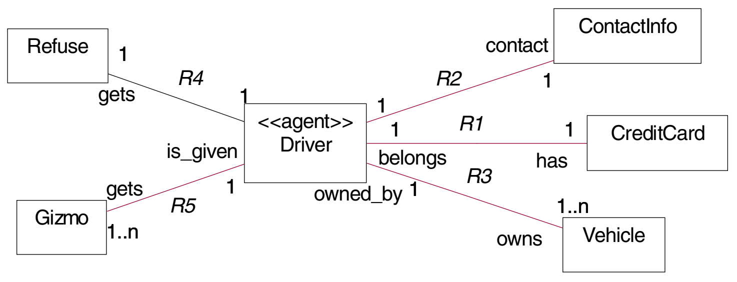
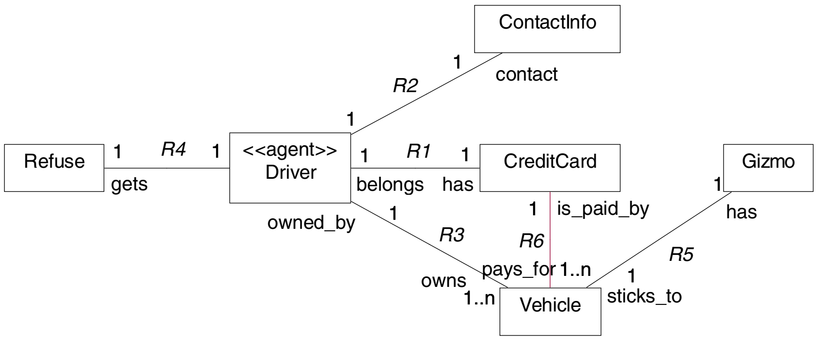


Figure 10. Refactored class model



accordingly (see Figure 9). Next, we create one class for each value gained or given away by the agent and link the class to the agent class. Finally, we determine and add the associations creating the collaboration paths between classes necessary for the use case execution (e.g., association R6 in Figure 10). The class model can be refactored at any time, so long as the new model does not invalidate the BVI for the use case (see Figure 10).

Figure 11 shows how as a result of analyzing the FSM for Operator, we extend the initial class model with a new actor class and several association links. The

Figure 11. Class model extended with knowledge from the FSM for Operator

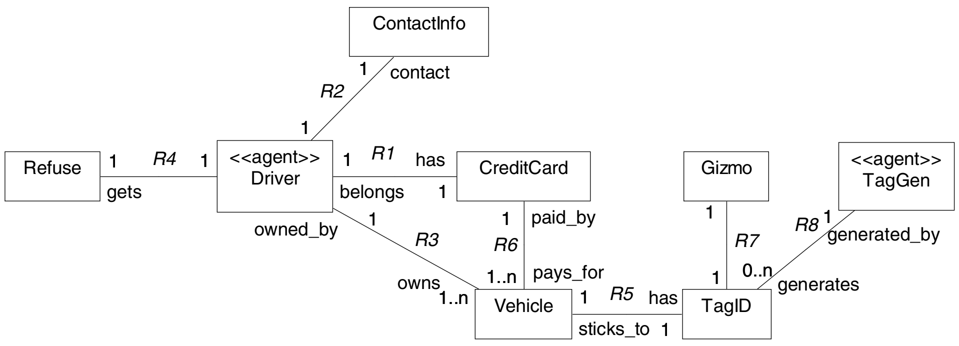
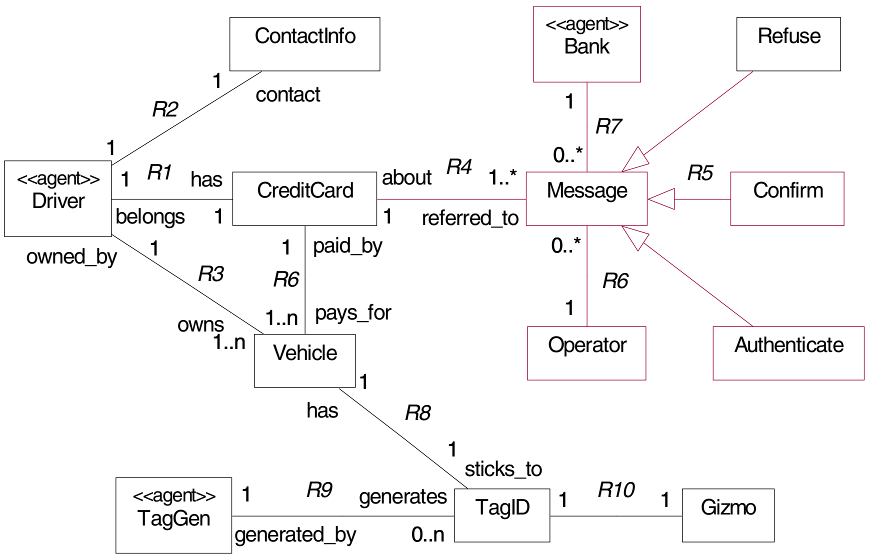


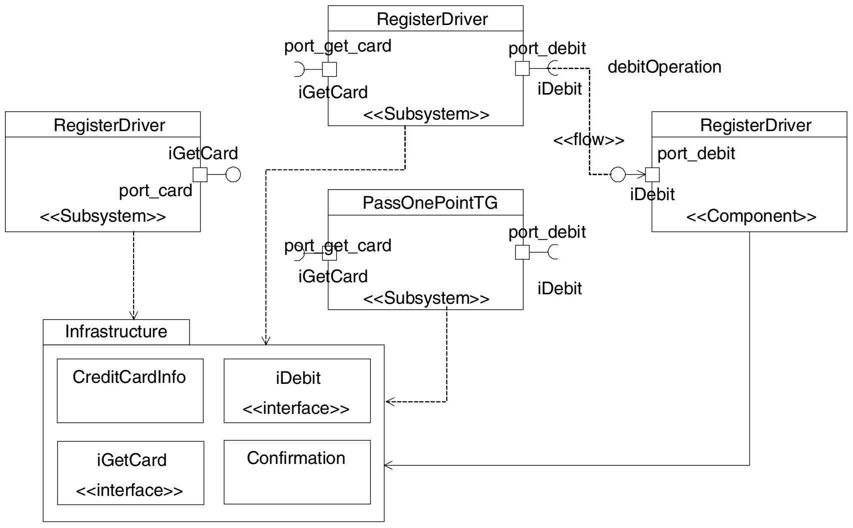
Figure 12. Compete class model for use case Register Driver



complete class model after processing agents Bank and TagGenerator is presented in Figure 12. Agent classes are stereotyped as entity classes to record the long-lived changes in the redistribution of business values in the system. Since each agent interacts with other agents, we duplicate the agent classes, and the clones form a pool of initial candidates for controller classes in the Boundary-Controller-Entity design pattern (Jacobson, 1992). This is justified by the behavioral knowledge in the FSMs and the existence of collaboration paths between agents and relevant business objects.

Figure 13 shows the physical decomposition of E-ZPass into subsystems. The package structure is organized by use cases. Each subsystem publishes a set of required and provided interfaces, derived from the interactions, formalized by the invariants. The runtime connections between the subsystems are presented with ports. It is important to note that all subsystems make use of the same well-defined set of types (interfaces and classes) in their public interfaces, hence the <<import>> dependencies to the infrastructure package. The infrastructure package defines and exposes the "common currency" for communication between the subsystems. The reasons for externalizing the sub-

Figure 13. Subsystems of E-ZPass



system interfaces are two: to limit the dependencies among the subsystems and to define their usage environment—that is, "what" the subsystem needs to provide to its environment (Bettin, 2004).

Converting the FSMs to OCL Specifications

The Object Constraint Language (OCL) (Warmer & Kleppe, 1998) is a formal, pure expression language augmenting graphical UML models to produce unambiguous and precise system descriptions. OCL is an integral part of UML, and it is also used to define the semantics of UML.

OCL combines first-order predicate logic with a diagram navigation language. It provides operations on sets, bags, and sequences to support the manipulation and queries of collections of model elements.

UML uses OCL to express constraints, navigability, action semantics, and object queries. Even though visual models define some constraints, like association multiplicity, in OCL we can specify richer ones, such as uniqueness constraints, formulae, limits, and business rules. OCL constraints provide precision, which facilitates design by contract.

A constraint is a semantic restriction on one or more model elements. Types of constraints include, but are not limited to, constraints on associations between classes, pre- and post-conditions on class operations, multiplicity of class instances, type constraints on class attribute values, invariants on classes, and guards in state models.

Each OCL expression is written and evaluated in the context of the instances of some model element, for example:

```
context Driver inv:
 self.Age > 16
```

As another example, to make sure that all tags have unique IDs, we write:

```
context Tag inv:
 Tag.allInstances()->forAll(t1, t2 |
 t1<>t2 implies t1.TagID <> t2.TagID)
```

Pre- and post-conditions specify the effect of a class operation without stating an algorithm or implementation. To indicate the operation for which the conditions must hold, we extend the constraint's context with the name of the operation.

```
context CreditCard::Charge(Amount: Real): Boolean
 pre: self.IsValid
 post: self.Balance = self.Balance@pre + Amount
```

where Balance@pre refers to the value of the attribute at the start of the operation.

OCL is not a programming language, and therefore OCL expressions cannot express programming logic or flow of control. OCL expressions have been criticized for having poor readability and for being inefficient in specifying requirements-level and analysis-level concepts (Ambler, 2002). The critique comes from methodologists favoring code over models and from system analysts using UML diagrams as sketches of OO designs, the so-called UMLAsSketch (Fowler, 2003). Mellor (2002) introduced the notion of constraint idiom as a general pattern for commonly occurring types of constraints and suggested the use of predefined tags for these types. The latter makes models less cluttered and less cryptic. Since constraints are a fundamental part of the semantics of a domain, there is no simple alternative for building precise domain models.

Constraints can also be written in any action language supporting the Precise Action Semantics for UML (AS, 2001). However, action languages are not specification languages, and their purpose is to define computations in executable models.

Declarative Specification

The evolution of each agent, for example Driver, is described by two consecutive frames (snapshots). The service that a use case provides must be a complete sequence of actions after the performance of which the system will be in a state allowing the execution of the sequence all over again.

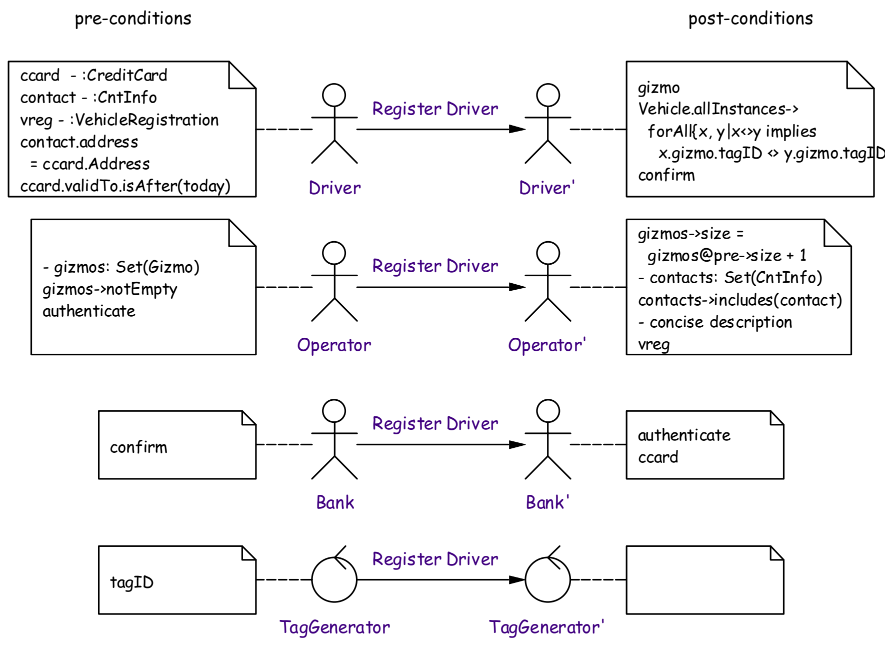
Let us consider again the FSM for Driver. We define declaratively the behavior of use case from the driver's point of view as a contract in terms of OCL expressions. The contract is an exact specification of the service provided by the use case w.r.t. Driver. The service is described by two sets of constraints whose context is an instance of Driver:

- *Pre-Conditions:* The conditions under which the service will be provided.
- *Post-Conditions:* A specification of the result, given that the preconditions are fulfilled.

The service pre- and post-conditions are described in the notes attached to the agents on the left- and right-hand sides of Driver and Driver', respectively, as shown in Figure 14. We split complicated constraints into several separate constraints to improve their readability and writeability (and diagnostic power, in case runtime checking is employed).

The precondition for a Driver to register a vehicle is to have a valid credit card and a mail address that coincides with the card's billing address. This is an example of a constraint that cannot be expressed in a class model alone. The Drier's post-condition includes a gizmo with a unique ID.

Figure 14. OCL contract for use case Register Driver



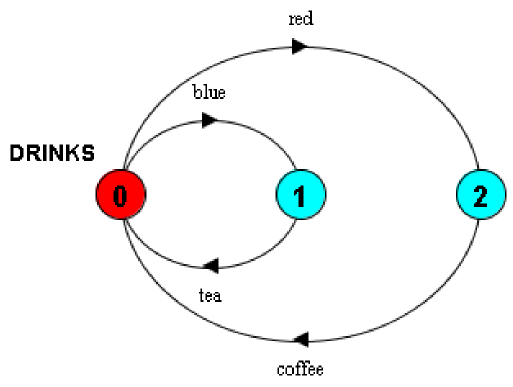
In addition, we define the following two constraints. gizmos->notEmpty is a precondition indicating that an instance of Gizmo is available at the outset. The constraint uses the predefined property notEmpty of the OCL type Set. The second constraint is a post-condition using the predefined feature of all OCL types allInstances. It is instrumental not only in specifying a uniqueness constraint, but also (implicitly through the logic-and of all postconditions) in expressing a semantic relationship between the value of vehicle.gizmo and the association Vehicle–Gizmo. This expression accesses the meta-level UML model to express the fact that a gizmo is assigned to exactly one vehicle.

User Requirements Validation

Model Animation

Labeled Transition System Analyzer (LTSA) (Magee & Kramer, 2003) is a tool for analyzing concurrent systems. A system in LTSA is represented as a set of

Figure 15. FSM model of a drinks dispensing machine



interacting FSMs. LTSA supports animation to facilitate interactive exploration of system behavior. The analyzer can also perform progress and safety checks. Though useful, the latter are outside the scope of our interest. Progress and safety checks require training in formal methods. Our goal is to enrich, at no extra cost, system analysts' toolsets, as well as to offer a lightweight process extension for requirements and system architecture validation.

LTSA takes as input the text descriptions of the FSMs and generates their composition. Graphical FSM models, though expressive, become unwieldy for a large number of states and transitions. An FSM text description is formed by the events of its transitions using action prefix, ->, and choice, |, operators. By action we mean the sending or receiving of an event.

Figure 15 displays the model of a drinks dispensing machine. It dispenses hot coffee if the red button is pressed, or iced tea if the blue button is pressed. The text description for the drinks dispensing machine is as follows:

```
DRINKS = (red->coffee->DRINKS | blue->tea->DRINKS).
```

The initial state is always numbered 0, and transitions are drawn in a clockwise direction. The text descriptions for the FSMs of use case Register Driver are shown in Figure 16.

To define FSM parallel composition, we have to introduce the notion of trace. An FSM describes all possible evolutions (runs) of a system. A particular run of a system is captured by a trace. A trace is a path in the FSM graph starting

Figure 16. Text descriptions of FSMs for use case Register Driver

```
DRIVER = (contactInfo->creditCard->vehicleInfo->(gizmo->DRIVER | refuse->DRIVER)).

OPERATOR = (contactInfo->creditCard->authCard->

                     (confirm->tagID->gizmo->OPERATOR | refuse->OPERATOR)).

BANK = (authCard->(confirm->BANK | refuse->BANK)).

TAGGEN = (tagID->TAGGEN).
```

from the initial state. It is formed by the events on the path using the action prefix operator. The following is a trace of the drinks dispensing machine:

red->coffee->blue->tea.

The parallel composition of two FSMs S and T, denoted by S || T, yields an FSM generating all possible interleavings of the traces of S and T (Magee & Kramer, 2003). Figure 17 shows the parallel composition of FSMs SEA and YAGHT, written as ||VOYAGE = (SEA || YAGHT).

Events common to two or more FSMs are called shared. Shared events model FSM interactions. A shared event must be executed at the same time by all participating FSMs. Figure 18 shows the composition of two FSMs, which have a common event called buffer. The composition is written as ||WORKSHOP = (PRODUCER || CONSUMER).

Figure 17. Parallel composition

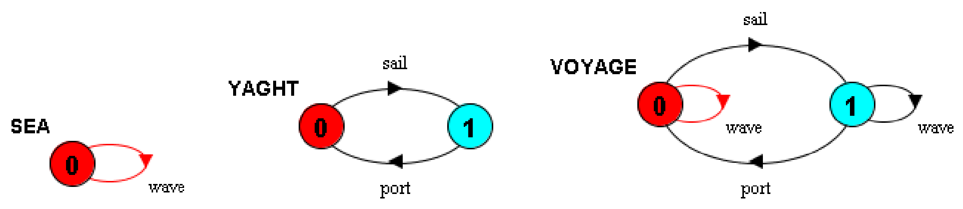
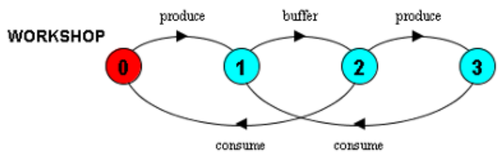


Figure 18. Parallel composition with shared events



```
PRODUCER = (produce->buffer->PRODUCER).
CONSUMER = (buffer->consume->CONSUMER).
||WORKSHOP = (PRODUCER || CONSUMER).
```

Figure 19. Composite FSM for use case Register Driver

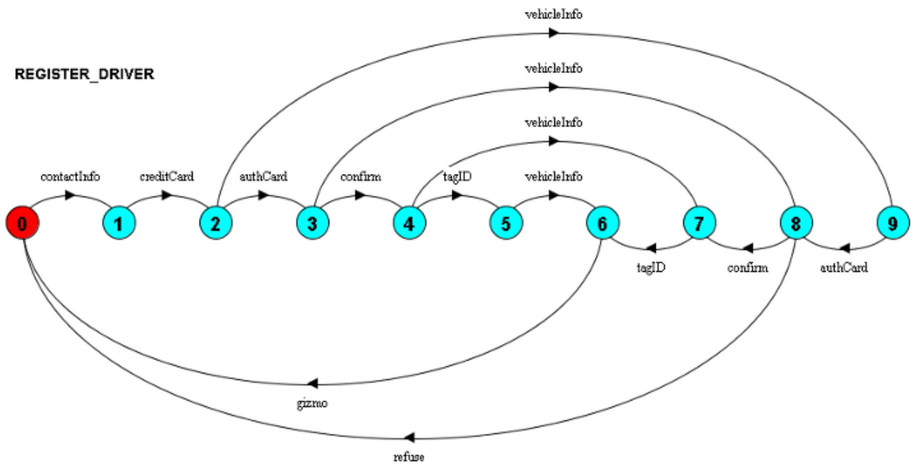
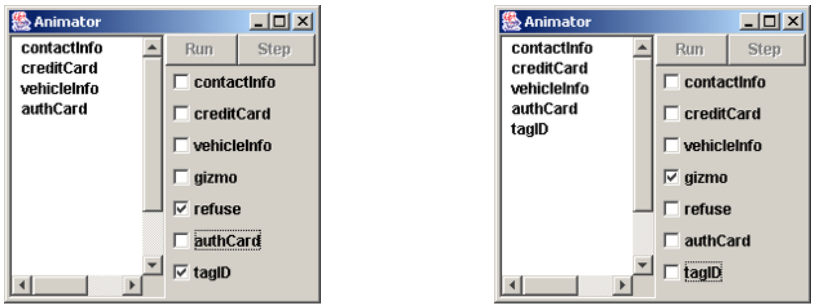


Figure 19 depicts the composite FSM for use case Register Driver. The component FSMs are DRIVER, OPERATOR, BANK, and TAGGEN. The composition operation is written as: ||REGISTER_DRIVER = (DRIVER || OPERATOR || BANK || TAGGEN).

LTSA parses and compiles every agent's text description to an FSM, and then generates a single FMS by composing the agent FSMs. The resulting FSM may be viewed graphically or textually and analyzed.

Figure 20. Two consecutive screenshots of the LTSA Animator window



contactInfo
creditCard
vehicleInfo
authCard

☐ contactInfo
☐ creditCard
☐ vehicleInfo
☐ gizmo
☑ refuse
☐ authCard
☑ tagID

contactInfo
creditCard
vehicleInfo
authCard
tagID

☐ contactInfo
☐ creditCard
☐ vehicleInfo
☑ gizmo
☐ refuse
☐ authCard
☐ tagID

FSM Animation

LTSA performs a user-controlled animation of the composite FSM by allowing explicit control of all events in the model. Figure 20 shows two screenshots of the LTSA Animator window. The analyzer lets the user control the events offered by the FSM model to its environment. The set of all events is shown in the right pane. The events, which can be chosen for execution at a particular step, are ticked. The event executed on the previous step is outlined. The sequence of events, shown in the left pane, is the trace that has brought the FSM to its current state. In the current state, the user can choose between events refuse (refuse authentication) and tagID (tag ID will be generated after a successful authentication). Let us assume that the user selects refuse for execution. In response, LTSA transitions to the next state and offers a new set of events for execution, the screenshot on the right. The animation game continues until the system reaches a final state, deadlock, or the user quits analysis.

Requirements and Architecture Validation

The essence of any evolutionary process boils down to specifying the system capabilities as use cases, and then to prioritize the use cases according to their risk magnitude. The use case with the highest priority is analyzed first. Analysis is followed by design, implementation, and testing of a collaboration of objects realizing the use case under consideration. The testing is done with test cases derived from the use case description. The resulting increment is integrated into a new build, after which integration and regression tests are run. Then, the next highest priority use case is selected, and the described process is repeated all over again.

In analysis, a use case is realized by a collaboration of logical classes. To validate that the functionality of the analysis-level architecture satisfies the user requirements, we suggest the following extension applicable to any evolutionary process.

For every use case:

1. Identify the counter-actors for the use case under consideration.
2. Identify the business objects exchanged between actors and counter-actors, needed to satisfy the goal of the initiating actor.
3. Define and balance the BVI for the use case. This activity may require revisiting steps (1) and (2), and revising some of the decisions already made, such as adding business objects and/or counter-actors.
4. Describe the evolution of each agent with an FSM. Associate every transition with an event modeling the acquiring of or giving away a business object.
5. Derive the FSMs' text descriptions and enter them into the LTSA tool. Then, generate the composite FSM for the use case.
6. In collaboration with representatives of the various stakeholder groups, animate the composite and component FSMs to validate that the analysis-level architecture supports the desired use case behavior. If there is no consensus that the animated model behavior is a faithful account for the stakeholder requirements, repeat the process.

Our experience shows that the described process converges very quickly. We attribute this to the fact that the prototype's animation involves interactivity and prediction. We view the BVI prototype of a use case as a discursive instrument of communication, enabling a meaningful and information-rich dialogue among stakeholders.

Discussion

The proposed process extension is advantageous for several reasons.

Formal guidelines for transforming a use case model to class models are practically missing in the literature. The presented BVI technique is an attempt to fill this void.

FSM modeling and FSM animation are intuitive and comprehensible to all stakeholders. Our experience confirms the findings of Davis (1988) that introducing FSM modeling to non-technical people takes on average an hour. Since the BVI process extension is defined over domain objects, all stakeholders can take part in the requirements validation, and also it guarantees that analysts stay out of design.

With the proposed approach, there is no up-front data modeling. The business object model is built incrementally in the process of designing the prototype.

To reflect the reality of a changing world, the requirements model should facilitate evolution over time. Requirements models play an important role in software development change management because the requests for change inevitably originate from business stakeholders. Requirements traceability deals with the impact of change, and helps to scope its possible effect and manage requirements' evolution over time. When a requirement is changed or added, the developer ought to be able to determine which system views will be affected and to change them accordingly. With the proposed technique, a change in the required or provided interface of an agent will invalidate the use case invariant. The BVI allows for a change to be traced forward to the agent FSMs and to the use case logical class model, and vice versa (traced backward). The analyst copes in a controlled way with the effect of a change by balancing the violated invariant.

The BVI approach differs from requirements prototyping and evolutionary prototyping in some fundamental ways. Requirements prototyping (Hickey, Davis, & Kaiser, 2003) is used as a requirements determination method that can be employed in requirements elicitation. Essentially, it is an "add-on" technique for improving user-developer communication and not a systems development methodology because it does not include any design activities. In contrast, with evolutionary prototyping, the developed artifacts are not discarded. Evolutionary prototyping constructs a preliminary system release prototype, which grows into a production system. The proposed process extension has all the benefits of requirements prototyping, without incurring the extra cost of a “throw-away” prototype. It is not only meant to elicit initial user reactions, but also to guide the discovery of the system’s logical structure. Unlike evolutionary prototyping, we do not advocate the construction of a preliminary system with all the negative consequences imposed on developers by an inadequate start. Instead, we propose a process extension, producing a prototype that can be animated interactively and formally analyzed, a prototype compliant with any evolutionary process model.

Davis’ (1988) is one of the first noteworthy, comprehensive attempts, still tenable today, that points out the desirable traits of a behavioral specification technique. It: 1) is understandable to all stakeholders including non-computer-oriented users; 2) serves as a design contract for developers; 3) should provide automated checks for ambiguity, incompleteness, and inconsistency; 4) encourages the requirements engineer to think and write in terms of external system behavior, not internal components; 5) should organize information in the requirements model; 6) should provide basis for automated prototype generation; 7) should provide basis for automated test generation; and 8) should be suitable to the particular application.

The use case model extended with invariants and model animation satisfies every single criterion Davis has defined, and in addition, links the outwardly visible, high-level system behavior to the behaviors and the information structures of the individual comprising parts. The FSMs’ traces, for example, constitute readily available functional tests. The formal nature of the model allows for automatic test generation, and given a suitable fault model, the traces can be used to assess quantitatively the tests’ fault coverage.

With the BVI approach, the cost of writing an invariant for a use case is offset by the ease and precision of classes, associations, and state behavior discovery. This cost is further reduced by the multiple feedback opportunities for end users to get the “yes, buts” out as early as requirements engineering, and thus make the processes of requirements elicitation and logical architecture converge faster. In a typical evolutionary process, developers cannot get feedback from end users before an iteration ends, which depending on the process can be anywhere between two to three weeks with Agile, and up to three months with RUP. With the proposed approach, the prototype for a large system, like the one used as a running example in this chapter, can be built within several working hours.

Precision is the basis for requirements analysis and for validation that the modeled requirements are indeed what business stakeholders want. Since requirements engineering straddles the informal world of stakeholders and the formal world of software developers, the question is not whether to use formal methods, but when to formalize. For example, Jacobson (2002) shares the view that formalism should creep into the system model. In contrast, Catalysis and BVI rely on rigorous OCL specifications early in the process. In formal methods, logic provides the rigor for performing analysis and reasoning. In recent years, many formal techniques have been proposed by the research community, for example, temporal logic for timing constraints, deontic logic for contracts, linear logic for resources, and Z and VDM for specifying operations with pre- and post-condition (Easterbrook et al., 1998; Jones, 1986; Spivey, 1989), but have not been widely adopted in practice (Neill & Laplante, 2003). Though very desirable, formal methods are expensive to introduce in a software process. This cost-related impediment owes to the high requirements on the developer for mathematical maturity. It is also due to the high cost of keeping a specification up-to-date in dynamic project settings, and the amount of effort that needs to go into constructing and analyzing a formal specification. A notable exception is Alloy (Jackson, 2002), which is proposed as a lightweight formal specification language. However, Alloy does not address modeling of interactions and is not tightly coupled to UML.

It is important to point out how our approach differs from formal methods. Unlike formal notations based on logic, the BVI approach does not preclude the owners of the problem from taking part in the problem-solving process; that is, the knowledge locked in the prototype can be validated. The capability to perform analysis and simulation on the model is the most noteworthy feature of the proposed technique, which is not currently addressed by OCL. It is still unknown to what extent OCL would be analyzed, though research toward this task has been done (e.g., Richters & Gogolla, 2002). Jackson (2002) discusses the difficulty of putting a language, like OCL, in an analyzable form.

On the down side, the BVI prototype cannot represent non-functional requirements, such as security and performance requirements.

Future Work

To scale up better to large-sized projects, we are working on a decomposition technique, decomposing a BVI to sub-BVIs, and later composing the sub-BVIs' prototypes to yield the use case prototype.

What we see as a promising venue is to link (or even map) the BVI prototype to executable UML models. This will enable the modeling of non-functional requirements and will further automate the development process.

We intend to extend the modeling technique and apply it to building domain models (Jackson & Zave, 1993; Mellor et al., 2004). A domain model is a complete but abstract description of the existing system. OMG's MDA uses the term PIM (platform independent model) to describe a similar, though less inclusive notion. A domain model is a software asset, leveraging domain knowledge and requirements reuse. The proposed technique makes it easy to reuse requirements across families of software products, so it can be used successfully in building software product lines.

Finally, we are interested in integrating the BVI prototyping technique with techniques for conflict resolution, requirements prioritization, and group consensus, such as EasyWinWin, JAD (August, 1991), User-Centered Design (Carrol, 1996), AND Interviews and Focus Groups (Macaulay, 1996).

Conclusion

In this work, we presented an algorithmic method of deriving the logical class model for a use case and for projecting the behavior of the use case realization over a set of observable business objects, exchanged between the system and its environment. The set of business objects is discovered using an algebraic invariant, called business value invariant. The invariant steers the process of building the information structure supporting the use case, as well as the formal description of a use case as a set of communicating state machines. To validate the modeled requirements, the derived system behavior is animated with a labeled transition system analyzer. Animating the system prototype allows for powerful visualization, study, and validation procedures such as interactivity and prediction. The proposed approach strengthens the traceability between use cases and use case realizations. It allows the owners of a problem to participate actively in the specification of the problem and in the validation of the logical structure of the solution. The proposed approach can be used in conjunction with any evolutionary process. Since it affords roundtrips between requirements modeling and analysis, it shortens these phases of the software lifecycle. To conclude, the proposed prototyping technique acts as a discursive communication instrument, bringing the worlds of clients and developers a step closer.

References

Ambler, S. (2002). Toward Executable UML. *Software Development,* (January).

AS. (2001). UML Actions Semantics. Retrieved from *www.omg.org*

August, J. (1991). *Joint application design: The group session approach to system design.* Yourdon Press.

Beck, K. (1999). Embracing change with extreme programming. *IEEE Computer, 32*(10), 70-77.

Bettin, J. (2004). Model-driven software development: An emerging paradigm for industrialized software asset development. Retrieved from *www.softmeta.com*

Boehm, B. (1988). A spiral model of software development and enhancement. *IEEE Computer*, *21*(5), 61-72.

Briggs, B., & Grünbacher, P. (2002). EasyWinWin: Managing complexity in requirements negotiation with GSS. *Proceedings of the Hawaii International Conference on System Sciences.*

Brooks, F. (1995). *The mythical man-month* (anniversary edition). Reading, MA: Addison-Wesley.

Carroll, J. (1996). Encountering others: Reciprocal openings in participatory design and user-centered design. *Journal of Human-Computer Interaction*, *11*(3), 285-290.

Carroll, J., & Mack, R. (1985). Metaphors, computing systems, and active learning. *International Journal of Human-Computer Studies*, *22*, 39-57.

D'Souza, D., & Wills, A. (1998). *Objects, components and frameworks with UML: The catalysis approach*. Reading, MA: Addison-Wesley.

Davis, A. (1988). A comparison of techniques for the specification of external system behavior. *Communications of the ACM*, *31*(9), 1098-1115.

Davis, A. (1992). Operational prototyping: A new development approach. *Software*, *9*(5), 70-78.

Derrida, J. (1967). *L'Ecriturie et la différence.* Paris.

Douglas, B.P. (2004). *Real Time UML* (3rd ed.). Boston: Addison-Wesley.

Easterbrook, E., Lutz, R., Covington, R., Kelly, J., Ampo, Y., & Hamilton, D. (1998). Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, *24*(1), 4-14.

Dardenne, A., Lamsweerde, A., & Fickas, S. (1993). Goal-directed requirements acquisition. *Science of Computer Programming, 20,* 3-50.

Davis, A. (1992). Operational prototyping: A new development approach. *Software*, *9*(5), 70-78.

Fowler, M. (2003). *UML distilled: A brief guide to the standard object modeling language* (3rd ed.). Reading, MA: Addison-Wesley.

Gemino, A., & Wand, Y. (2003). Evaluating modeling techniques based on models of learning. *Communications of the ACM*, *46*(10), 79-84.

Gentner, D. (1998). Analogy. In W. Bechtel & G. Graham (Eds.), *A companion to cognitive science* (pp. 107-113). Oxford: Blackwell.

Gravell, A., & Henderson, P. (1996). Executing formal specifications need not be harmful. *IEEE Software Engineering Journal*, *11*(2), 104-110.

Hegarty, M., Quilici, J., Narayanan, N., Holmquist, S., & Moreno, R. (1999). Designing multimedia manuals that explain how machines work: Lessons from evaluation of a theory-based design. *Journal of Educational Multimedia and Hypermedia*, *8,* 119-150.

Hickey, A., Davis, A., & Kaiser, D. (2003). Requirements elicitation techniques: Analyzing the gap between technology availability and technology use. *Comparative Technology Transfer and Society Journal*, *1*(3), 279-302.

Jackson, M., & Zave, P. (1993). Domain descriptions. *Proceedings of the 1st International Symposium on Requirements Engineering* (pp. 56-64), San Diego.

Jackson, D. (2002). Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering, 11*(2).

Jacobson, I. (1987). Object-oriented development in an industrial environment. *ACM SIGPLAN Notices*, *22*(12), 183-191.

Jacobson, I. (1992). *Object-oriented software engineering: A use case driven approach*. Reading, MA: Addison-Wesley.

Jacobson, I. (2002). Private communication.

Jones, C. (1986). *Systematic software development using VDM*. Englewood Cliffs, NJ: Prentice-Hall.

Kruchten, P. (2000). *The Rational Unified Process®: An introduction*. Reading, MA: Addison-Wesley.

Lacan, J. (1977). *Ecrits: A selection* (A. Sheridan, trans.). London: Tavistock.

Macaulay, L. (1996). *Requirements engineering.* Berlin: Springer.

Maciaszek, L. (2001). *Requirements analysis and system design*. Reading, MA: Addison-Wesley.

Magee, J., & Kramer, J (1999). *Concurrency: State models and Java programs*. New York: John Wiley & Sons.

McDermid, J. (1994). Requirements analysis: Orthodoxy, fundamentalism and heresy. In Jirotka & Goguen (Eds.), *Requirements engineering: Social and technical issues* (pp. 17-40). Academic Press.

Mellor, S.J., Scott, K., Uhl, A., & Weise, D. (2004). *MDA distilled*. Boston: Addison-Wesley.

Mellor, S.J., & Balcer, M.J. (2002). *Executable UML: A foundation for Model-Driven Architecture.* Boston: Addison-Wesley Professional.

Milner, R. (1989). *Communication and concurrency*. International Series in Computer Science. Englewood Cliffs, NJ: Prentice-Hall.

Neill, C., & Laplante, P. (2003). Requirements engineering: The state of the practice. *IEEE Software*, 40-45.

Nikula U., Sajaniemi, J., & Kälviäinen, H. (2000). *A state-of-the-practice survey on requirements engineering in small- and medium-sized enterprises*, Telecom Business Research Center Lappeenranta, Lappeenranta University of Technology, Finland. Retrieved from *www.lut.fi/TBRC/*

Parnas, D. (1972). A technique for software module specification with examples. *Communications of the ACM*, *15*(5), 330-336.

Richters, M., & Gogolla, M. (2002). OCL: Syntax, semantics, and tools. In Clark & Warmer (Eds.), *Object modeling with the OCL, the rationale* *behind the Object Constraint Language.* Berlin: Springer (LNCS 2263).

Roussev, B. (2003), Generating OCL specifications and class diagrams from use cases: A Newtonian approach. *Proceedings of the 36th Hawaii International Conference on System and Sciences* (HICSS-36), Hawaii.

Shannon, C.E., & Weaver, W. (1949). *The mathematical theory of communications*. Urbana, IL: University of Illinois Press.

Shaw, M., & Gaines, B. (1996). Requirements acquisition. *Software Engineering Journal*, *11*(3), 149-165.

Spivey, J.M. (1989). *The Z notation: A reference manual*. Englewood Cliffs, NJ: Prentice-Hall.

Standish Group. (2003). *CHAOS research report.* Retrieved from *www.standishgroup.com*

Thompson, S., & Riding, R. (1990). The effect of animated diagrams on the understanding of a mathematical demonstration in 11- to 14-year-old pupils. *British Journal of Educational Psychology*, *60*, 93-98.

Tversky, B., Morrison, B., & Betrancourt, M. (2002). Animation: Can it facilitate? *International Journal of Human Computer Interaction*, *57*, 247-262.

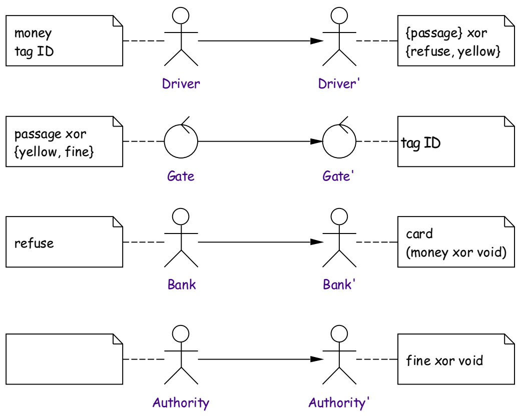
UML. (2004). UML Specification. Retrieved from *www.omg.org*

Warmer, J., & Kleppe, A. (1998). *The Object Constraint Language: Precise modeling with UML.* Reading, MA: Addison-Wesley.

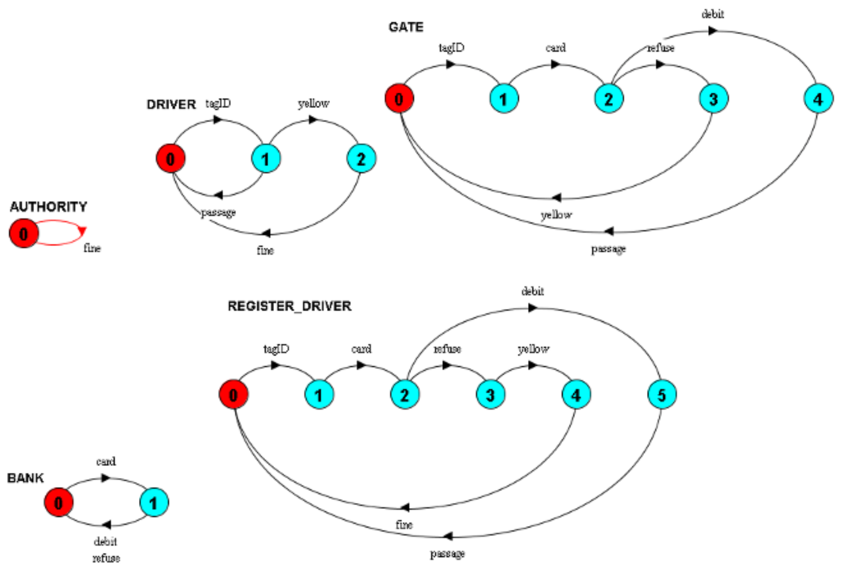
Wieringa, R. (1998). A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, *30*(4).

Appendix

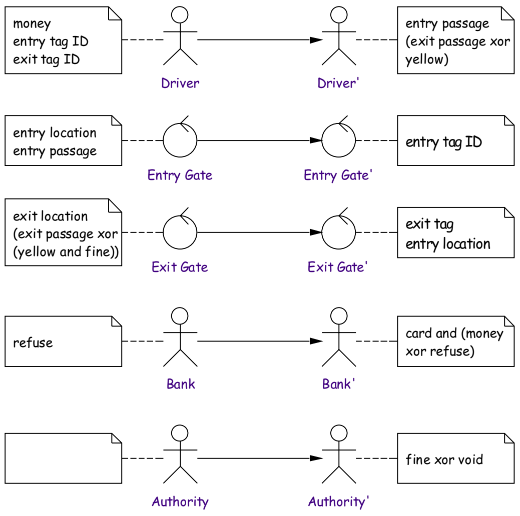
PassOnePointTollGate



```
AUTHORITY = (fine->AUTHORITY).
BANK = (card->(debit->BANK | refuse->BANK)).
DRIVER = (tagID->(passage->DRIVER | yellow->fine->DRIVER)).
GATE = (tagID->card->(debit->passage->GATE | refuse->yellow->GATE)).
||REGISTER = (DRIVER || GATE || BANK || AUTHORITY).
```



PassTwoPointTollgate



DRIVER = (entryTagID->entryPassage->exitTagID->(yellow->DRIVER | exitPassage->DRIVER)).

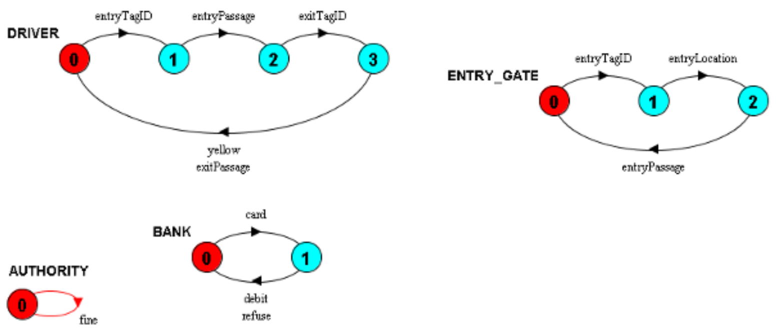
ENTRY_GATE = (entryTagID->entryLocation->entryPassage->ENTRY_GATE).

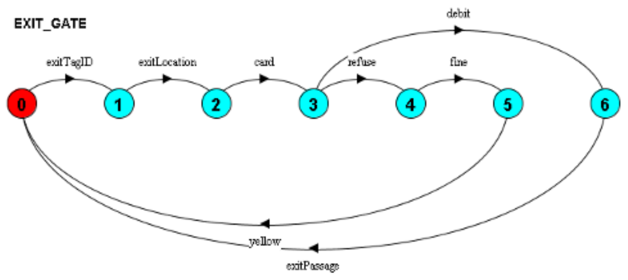
EXIT_GATE =(exitTagID->exitLocation->card->
    (debit->exitPassage->EXIT_GATE | refuse->fine->yellow->EXIT_GATE)).

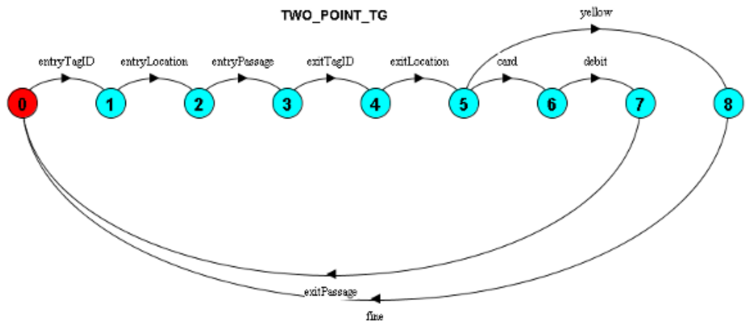
BANK = (card->(debit->BANK | refuse->BANK)).

AUTHORITY = (fine->AUTHORITY).

||TWO_POINT_TG = (DRIVER || ENTRY_GATE || EXIT_GATE || BANK || AUTHORITY).




EXIT_GATE
exitTagID
exitLocation
card
refuse
fine
debit
yellow
exitPassage
0
1
2
3
4
5
6


TWO_POINT_TG
entryTagID
entryLocation
entryPassage
exitTagID
exitLocation
card
debit
yellow
exitPassage
fine
0
1
2
3
4
5
6
7
8

Chapter VII

RUP and eXtreme Programming: Complementing Processes

Gary Pollice
Worcester Institute of Technology, USA

Abstract

The Rational Unified Process®, or RUP®, and eXtreme Programming (XP) are two popular software development methodologies or processes. Most people tend to think of them as opposing methods, where a project may adopt one or the other, but certainly not both. This essay describes how the two can be applied together to improve the software development practices of an organization.

Apples and Oranges

Trying to compare RUP to XP, or vice-versa, is truly like comparing apples to oranges. They were designed for completely different purposes and apply to different software development contexts and different levels of process. This does not mean that you have to choose one or the other. XP and RUP are based upon similar principles and contain valuable techniques to guide you through software development projects. They can be applied in a complementary manner with a little bit of forethought and planning. The following sections are designed to help you understand both of these practices and how to apply them to your organization and project. These sections are:

- Introduction to RUP and XP
- Core Values and Principles
- Similarities and Differences
- Applying the Practices

Introduction to RUP and XP

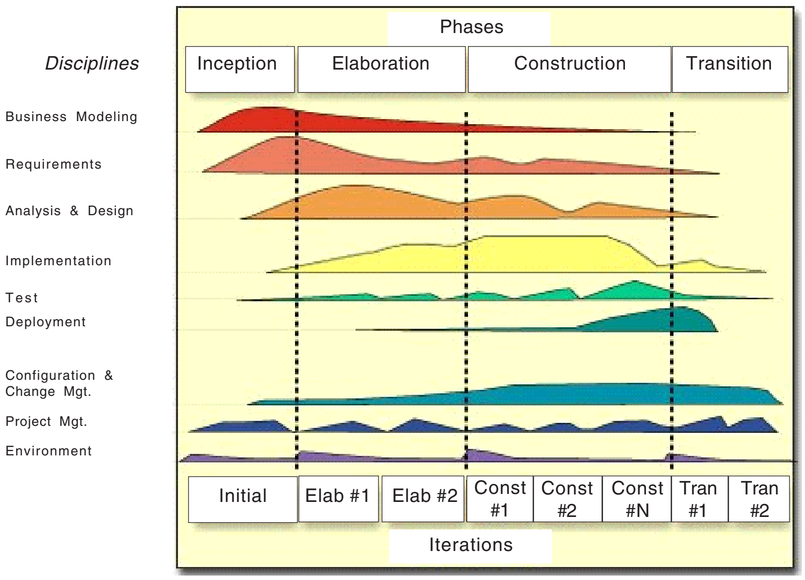
RUP

RUP is not a single process. It is a process framework that describes, in detail, a set of activities, artifacts, and responsibilities that may apply to a software development organization. The framework is designed as a consistent description of an incremental, iterative software development lifecycle. RUP describes process along two dimensions, time and discipline, as shown in Figure 1, which is referred to as the RUP "hump" chart. The disciplines are applied at different levels, depending upon where the project is according to the time dimension.

The time dimension describes a software project as it moves through four phases: Inception, Elaboration, Construction, and Transition. At the end of each phase, there is a milestone that determines if the phase is complete, and whether it is appropriate and advantageous to proceed to the next phase. Each milestone provides for a "go or no-go" decision by the stakeholders of the project. The milestones allow timely cancellation of projects that are unfeasible or too costly, usually due to changing requirements and business climate.

The Inception phase is the first of the RUP phases. During this phase you work to understand all of the stakeholders in the project, their needs and expectations, and describe the system's requirements at a high level. You build a *shared vision* of the project's results. At the end of the phase, you are in a

Figure 1. The RUP "hump" chart



position to state what you plan on building, who wants it, and whether the results deliver sufficient value to the stakeholders to warrant continued work. This is called the Lifecycle Objectives (LCO) milestone. If the stakeholders agree that the project is worth continuing, you proceed to the Elaboration phase.

During the Elaboration phase you attack technical risks. The core architecture of the project is developed. In a project that extends a previous software product, you enhance the existing architecture to accommodate new features. While the developers, usually architects, develop the technical framework, the stakeholders work with analysts and other team members to refine the requirements that were defined at a high level during Inception. At the end of Elaboration, the team is able to exhibit an *executable architecture,* and the Lifecycle Architecture (LCA) milestone is assessed. An executable architecture is running software that embodies just enough functionality to give evidence that the project is technically feasible and can be done within acceptable cost and time constraints. At this point it is common to add team members for the Construction phase.

The bulk of development work is performed during the Construction phase. Remaining features are added to the executable architecture, and earlier features are refined to make them ready for deployment. Requirements continue to be analyzed and managed, and possible changes are factored into each iteration's plans. At the end of Construction, the software is functionally complete and ready to be deployed to an end user community. While it is rare, the project may still be cancelled. The stakeholders review the work and the ongoing costs during the Initial Operation Capability (IOC) milestone review at the end of the Construction phase. If the business environment has changed and/or the cost to complete the deployment of the software is not acceptable, the project may be cancelled; otherwise, the project enters the final phase.

The purpose of the Transition phase is to ensure that the complete product is customer ready. Typically the Transition phase encompasses the beta test period. This allows the documentation, packaging, and complete release cycle to be tested. Any problems or last-minute defects or problems are worked out during Transition. At the end of Transition, the final milestone, Product Release (PRM), is evaluated and the product released. Usually, requirements for the next release and defect have been deferred to begin the cycle over again. Subsequent releases are handled as separate RUP projects.[1]

While the phases are sequential, the RUP process framework is not. It follows the spiral model described by Barry Boehm in 1988. During each iteration, all or most of the RUP disciplines are employed to produce working software. The disciplines are simply a way of grouping related activities and responsibilities. The disciplines in the current RUP product are:

- Business Modeling
- Requirements
- Analysis and Design
- Implementation
- Test
- Deployment
- Configuration and Change Management
- Project Management
- Environment

Different activities from the disciplines are emphasized depending upon whether the iteration is early in the project (Inception) or later in the project (toward the end of Construction or in Transition). This produces the "humps" in Figure 1, but the exact amount of effort shown is an estimate based upon experience over many different types of projects.

eXtreme Programming

eXtreme Programming is a mostly developer-centered set of tightly collaborative practices that are designed to be used together on small to medium-sized software projects. XP was originally designed by Kent Beck and Ward Cunningham in the 1990s. The practice was described by Beck (1999) in his book, *Extreme Programming Explained*. The first major project to apply the practice was the Chrysler Comprehensive Compensation system. This is a payroll system developed for Chrysler Corporation around 1996.

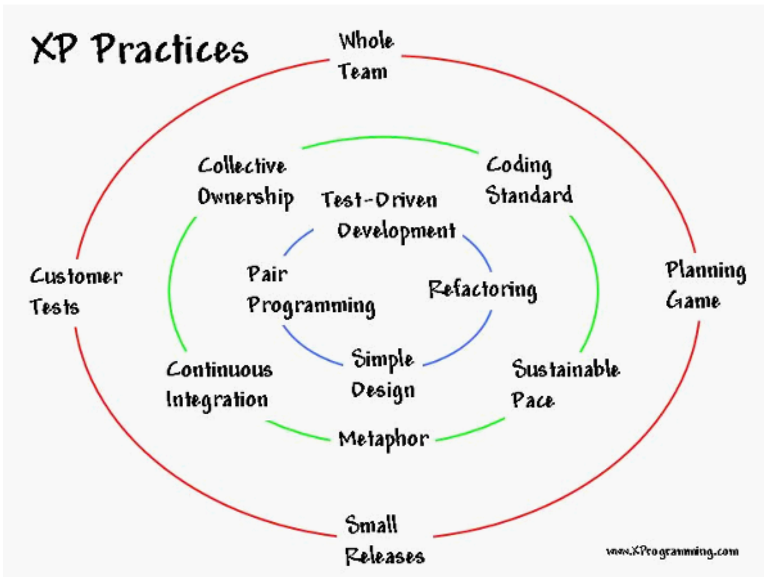
The original set of practices has been modified slightly and extended, but not too much. XP has captured the imagination of many programmers and software professionals. There are many articles, conference presentations, and conferences today that focus on XP and its application and evolution.

XP is based upon four values: simplicity, communication, feedback, and courage. The key practices are designed to support these values when they are applied consistently throughout the project. There are 12 practices in the original description of XP. At the current time, there are 13 that seem to be widely accepted.[2] The 13 practices and brief descriptions of each follow:

- **Whole team:** The complete team, developers, business people, and any others work together daily in the same location. Usually there is a large common room where the work occurs.
- **Planning game:** This practice involves the team planning process. Business people prioritize and select the work to be done, and technical people estimate the difficulty and amount they are able to deliver during an iteration.
- **Small releases:** The team releases working software to the customer (not necessarily to end users) for every iteration.
- **Customer tests:** Customers are responsible for writing and running the acceptance tests for the system. They may have technical people help them implement the tests using automated testing tools, but they are the ones responsible for determining that acceptance criteria are met.

- **Simple design:** The team implements the simplest possible solution for the current requirements and does not concern itself with things that might occur in the future. They strive to have the simplest, leanest design. The actual practice is one of continuous design where the design is continually undergoing refinement and evolves as necessary.
- **Pair programming:** Two programmers work together, actively, on all code that is put into production. They sit at one computer and work on the same piece of code. One programmer is the driver, but both are contributing to the intellectual effort that goes into producing correct software.
- **Test-driven development:** Originally this was called programmer testing and Test-First Programming (TFP). It has evolved into test-driven development. The practice requires programmers to write a test that fails before writing code that implements needed functionality in the final system. The claim is that the practice leads naturally into a clean design.
- **Refactoring:** In order to keep the design simple as the code evolves, the team continually takes time to refactor and keep the code as simple as possible, understandable, and clean.[3]
- **Continuous integration:** The team keeps the complete system integrated at all times. Often tools are employed that automatically start integration builds whenever there is new code checked into the version control system. This ensures that any problems with integration are caught and fixed as early as possible.
- **Collective code ownership:** On an XP project any programmer has the right and the responsibility to make changes to any code when necessary. Used in conjunction with simple design and refactoring, it ensures the code base is kept as clean as possible.
- **Coding standard:** The team follows a coding standard in order to have the code look as if it were written by a single person. The standard identifies not only the placement of syntactic items, but naming and documentation conventions.
- **Metaphor:** Metaphor is a practice that has been hard to nail down. An XP team develops a metaphor to communicate the vision of the product and how it works. Often more than one metaphor is necessary.
- **Sustainable pace:** This was originally called the *40 hour week* practice, but it soon became evident that different teams had ideas about what is

Figure 2. XP circle of life



reasonable in terms of weekly effort. The general rule is that a team member will not put in two weeks in a row of overtime.

The practices are shown graphically in Figure 2, which is referred to as the "circle of life." The outer circle represents those practices that apply to the whole team throughout each iteration. The inner circle are the practices that the developers practice daily, several times during the day. The middle circle are practices that support the other two.

Core Values and Principles

The core XP values were described in the previous section. The core values of RUP are not as apparent and are not explicitly stated in RUP. But after working with RUP for a while, you begin to understand that the values are quite similar to the agile values, but allow for more variability in how the values are emphasized.

Along with the agile values, there are several supporting principles of agile software development. These are the principles behind the Agile Manifesto that was drafted to launch the agile software development movement. The principles are presented here, somewhat paraphrased:[4]

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress. Agile processes promote sustainable development.
- The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Again, RUP had not specifically elaborated the principles behind successful use. In December 2001, Per Kroll, the director of the RUP team, wrote an article titled "The Spirit of the RUP." In the article he identified eight specific principles that embody the spirit behind RUP. These are:

1. Attack major risks early and continuously…or they will attack you.
2. Ensure that you deliver value to your customer.
3. Stay focused on executable software.
4. Accommodate change early in the project.
5. Baseline an executable architecture early on.
6. Build your system with components.
7. Work together as one team.
8. Make quality a way of life, not an afterthought.

The RUP principles were further expanded in Kroll and Kruchten's (2003) book, *The Rational Unified Process® Made Easy*.

Similarities and Differences

There are clearly many similarities between RUP and XP that can be inferred from looking at the principles upon which they are based. There are other similarities that can be seen upon closer examination. There are also significant differences that make one approach a better choice than the other in the right situation. In many cases it is quite appropriate to combine RUP and XP practices on a project to optimize the project's chance for success.

Similarities

The first and possibly the major similarity between RUP and XP is that they are both iterative and incremental in nature. Although RUP has the sequential phases, the real work is done iteratively. Except for early Inception iterations, each iteration results in working software that delivers potential value to the customer. In XP the iterations are made as short as possible, typically a week. On a RUP project, the length of iteration is not specified, but the spirit of RUP is that the iterations should not be too long. Iterations of two weeks to a month are common.

The incremental nature of both methods is apparent when you consider that each iteration builds upon the work done in previous iterations. XP tells you to refactor the code, but you do not throw away the work you have completed. You add onto it. In RUP, you plan iterations by implementing use cases, or scenarios of particular use cases, so that you deliver value that is usable, and then you add onto that by including more use cases or scenarios in future iterations. In both approaches work done in early iterations forms the basis for later work.

Team reflection is a key to successful adoption of both XP and RUP. RUP explicitly describes a point at the end of each iteration where the team reviews the iteration and decides what changes are needed. Following the last principle of the Agile Manifesto, XP teams are continually looking for ways to improve their process.

Until "The Spirit of the RUP" was published, many people had the mistaken impression that RUP was mostly concerned with the *form* of the project, rather than the ultimate deliverable, which is working software. Successful RUP practitioners knew this to be false, but there were cases where people tried to do everything RUP said, exactly as shown in RUP, and they failed. This led to the view that RUP completely defined exactly what each team member should do from the beginning of a project to the conclusion.

There is a very important, yet unstated, similarity between RUP and XP—if you are trying it for the first time, you probably need a coach. Whether you call the person a coach, mentor, companion, or some other appropriate name, the role is important to maximize your team's potential for success. The coach's responsibilities may be different for XP than for RUP, but it is just as important.

The first time a team attempts to apply XP, they will find that it is highly disciplined and often difficult to consistently apply the practices at all times.[5] Any change is hard and more difficult the farther it is from your normal operational mode. Many of the XP practices, such as pair programming and test-driven development, are very different than what developers might be used to. The role of the coach on an XP project is to help the team adapt to the changes and understand how to work together effectively.

The RUP coach's main responsibility is to help the team determine which set of practices, the level of formality, and the activities that best suit the project at hand. Like the XP coach, the RUP coach also has to educate the team on new practices. Bergström and Råberg (2004) describe many of the characteristics and responsibilities of the RUP coach in their book, *Adopting the Rational Unified Process®*.

Change is the one constant factor that both XP and RUP agree on. While XP follows the agile principle of allowing change at any time, RUP takes a slightly different viewpoint. Change can happen at any time, but it is managed change. In truth, both are saying the same thing. If there is a very late change in an XP project, the customer will have to decide what gets done and what gets deferred in the next iteration in order to accommodate the change. And, this has to be done within the sustainable pace practice. In RUP, the change will go through a prioritization process where the benefits and risks are analyzed and the hard decisions of whether to include the change, remove something, or change the end date are made.

Dealing with changing requirements relates to another common feature between RUP and XP—managing risk. Both approaches to software development are based upon minimizing the risk of failure. RUP has explicit artifacts to help manage risk. One of these is the Risk List that is created early in the project, and continually updated and reviewed to help teams deal with risk. During each iteration the team works on activities that mitigate the top risks.

The Elaboration phase in a RUP project is designed to address and mitigate technical risks. This means that the architects evaluate the requirements and design the architecture to ensure that the software can be built, and provide enough code to prove the solution. In XP, when a technical risk occurs, the team will often work on a *spike solution*, where developers take a short period to investigate and try different approaches to the problem. This is similar to the work done during Elaboration, but on a smaller piece of the project.

Testing and quality are another area of agreement between RUP and XP. The TDD practice in XP places a high value on correct code that matches the requirements as described in the test. Customer tests ensure the software meets the customer's needs. Pride in the quality of their code has become a very noticeable trait of XP developers.

One only need look at the Test discipline in the RUP hump chart to see that testing and quality are an integral part of a RUP project from the beginning. RUP identifies several activities and practices that are applied throughout the project to help develop a quality product and validate the product against the requirements.

The way requirements are gathered and expressed is both a similarity and difference between the two methods. RUP does not require, but advocates use cases as the primary way of describing functional requirements. A use case is a complete sequence of actions that are initiated by an actor[6] that delivers visible results. XP recommends user stories as the preferred method for capturing requirements. User stories are short descriptions of functionality that the system must deliver. They are written by the customer on index cards. Allistair Cockburn has said that user stories are promises for conversations between the customer and developer.

Usually when you plan an iteration, the customer decides which user stories will be implemented during the iteration. The user stories are rated as far as the effort that will be required, and the customer has the ability to select user stories that do not require more effort than the team has budgeted for the iteration. Clearly, there are some connections between the stories, and some must precede others. Often a team will package user stories that must be implemented in some sequence together. This is very similar to a use case. The user stories are like steps of a use case and can be very effective tools for planning.

Differences

While there are many similarities between XP and RUP, there are significant differences. Understanding both the similarities and differences is necessary if you want to adopt an approach to developing software that effectively utilizes practices from both. The main differences fall into four categories:

1. The nature of the process
2. Scope
3. Project size
4. Information content

Each of them will be explained in the following paragraphs.

A popular misconception about RUP is that it is a *process*. In fact, RUP is a process framework. It is based upon the Software Process Engineering Metamodel (SPEM). This is a metamodel for how to describe processes. As such, RUP contains a lot of information that *must* be tailored for each instance—the context in which the software development project lives. A sure way to fail with RUP is to try and apply all of the guidance to any single project. This is the reason that a main responsibility of a RUP coach is to select the activities and features of RUP that are appropriate for the team and project at hand.

XP on the other hand is a specific instance of practices that are interdependent. There is an ongoing debate as to what it means to "do XP." If you consider XP to be the application of all of the practices continually and consistently, then it is easy to decide whether a team is doing XP. That is not necessarily a realistic viewpoint, but one we can use to illustrate the difference between the nature of XP and RUP.

The scope of the process refers to how much of the complete software lifecycle it addresses. XP is mainly code and developer-centric. It does address very well the interaction of the business stakeholders (Customer) with the developers, but does not really address the business context, deployment, and other issues that are included in a full software project lifecycle. In terms of the RUP phases, XP focuses on the Elaboration and Construction phases.

The scope of RUP is broader than XP. It explicitly takes into account many of the environmental aspects, business context, delivery, and support issues that are not covered or implicit in XP. RUP does not cover all of a software product's lifecycle however. There is no explicit description of the support, maintenance, and retirement of the software. Several extensions have been made to RUP by third parties to cover these and other areas.

RUP is usually applied across an organization. Each project or department will customize it as necessary, but the general approach is the same. XP is typically applied at the project level, although some large departments in organizations like Sabre Airline Solutions have adopted XP as the primary development process throughout.

Project size is another differentiator. When Kent Beck described XP, he described it as a method that was quite appropriate for teams of approximately 10 developers. It is easy to see that when the project team gets too large, the ability to co-locate the whole team could become a problem. There are other impediments to the rapid feedback and communication that XP promotes when applied to a large team. There are several case studies of how larger teams have tried to apply some of the XP practices with varying success.[7]

RUP was designed to apply to many different-sized project teams. It was widely thought to be applicable only to large teams, but this is not the case. Grady Booch made the statement that most software development is done, when you get to the actual working units, in teams of about seven developers. Philippe Kruchten describes how one might use RUP on a single person project, and Pollice et al. (2004) describe how they applied RUP to a project with four people. But RUP has also been used successfully in large, multinational organizations like Volvo IT. A description of that project can be found in the appendix of Bergström and Råberg (2004).

The final major difference between RUP and XP is the information content. XP can be described quite succinctly. There have been numerous books written about it, but the key practices and the values are quite easy to describe. XP does not describe explicit artifacts, has very few roles, and provides little description of the sequence of activities, prerequisites, timing, and other characteristics of the process that are described in RUP.

RUP is more like a software engineering encyclopedia or textbook. It contains not only a high-level description of the different process elements; it contains a lot of detailed information about how to perform activities, what templates might be appropriate for certain artifacts, what tools are useful, how to use them, and so on. If RUP were printed, it would be over 3,000 pages of text—clearly too much to absorb for any single project.

Applying the Practices

The worth of any process is not in having a process, but in applying it successfully to help you deliver value to your customers. Questions like "Are we doing RUP?" or "Are we doing XP?" are not that important. Better questions to ask are "Is our process helping us provide value to our customer?" and "What can we do even better?"

Knowing the similarities and differences between XP and RUP, as well as other methodologies, can help you select the right practices, at the right level, for your organization and project. The most important characteristic you should possess is common sense. Do not be wedded to a specific approach or practice until you know what your ultimate goal is. Then look for the best match. There are, however, some general guidelines for selecting the appropriate approach that are described in the following paragraphs.

- **Use only what you need:** Often, people rely on the process too much. They think that as long as there is a process in place and it covers everything, they are guaranteed to have a successful project. Because they are afraid of missing something, they try to include everything. This leads to a bloated process that either no one follows, or that people follow and produce a lot of needless artifacts, forgetting the primary goal of delivering valuable software. The best advice is to include only those practices, artifacts, activities, and so on that you *are sure* you need, and leave out the rest. If you omit something that you need, you will find out soon enough and have time to add it to your process. If you include something that you do not need, you may never know.

There is one caution about applying this suggestion. Do not omit something just for the sake of keeping your process small or lean. Be objective in your assessment of your needs. For example, if you have a distributed development team or there are other projects that depend upon yours, then using user stories on index cards and getting the real details through face-to-face conversations with the customers is probably not enough. You may have to write out some use cases or Software Requirement Specification (SRS) documents.

- **Don't be afraid to change things:** Just as requirements change for software projects, the needs of the team may change or the initial process may be inadequate. Both RUP and XP encourage reflection and change. Self-examinations and evaluations are needed for a healthy team and organization.
- **Apply process at appropriate levels:** It would probably be foolish to try to get all enterprise-wide software development using only XP practices in a very large organization. At the highest levels of the organization, there might be teams responsible for strategic systems planning, corporate infrastructure and architecture, and so on. Much of the work these teams do is not code-centric. The XP practices are not appropriate. Activities like business modeling and architectural analysis described in RUP are better suited to these groups.

At some point, software projects are started in organizations like the one just described that get down to building and delivering applications. The teams responsible for this work can certainly apply many of the practices from processes like XP. Test-driven development can be used by almost any software developer who is writing code, as can pair programming. In general you don't want the developers to be spending the bulk of their time developing

UML diagrams or writing use cases. You want them creating a solution that transforms the requirements to software.

- **If you have multiple levels of process, make sure they are coordinated:** Even if you have applied different processes in your organization as suggested above, things may not go smoothly. A good bit of coordination is required. At the very minimum, there will be requirements and schedules that cross individual projects. Someone must ensure that they are scheduled and managed properly. In a large organization, there is a need for project and program managers who are responsible for just this type of coordination.

Summary

RUP and XP are both effective approaches to developing high-quality, valuable software. They have many similarities and some significant differences. While they may be used alone, one approach to improving the effectiveness of your organization and project team might be applying practices from both at the right level. By using common sense and following a few guidelines, you can identify and apply the right set of practices to help your organization succeed.

References

Beck, K. (1999). *Extreme programming explained: Embrace change.* Reading, MA: Addison-Wesley.

Bergström, S., & Råberg, L. (2004). *Adopting the Rational Unified Process®: Success with the RUP.* Reading, MA: Addison-Wesley.

Kroll, P., & Kruchten, P. (2003). *The Rational Unified Process made easy: A practitioner's guide to Rational Unified Process.* Reading, MA: Addison-Wesley.

Pollice, G. et al. (2004). *Software development for small teams: A RUP-centric approach.* Reading, MA: Addison-Wesley.

Endnotes

1 Some organizations will enter a cycle of continuous enhancement to a product by performing very limited Inception and Elaboration phases, and concentrating on Construction and Transition for each release. Some people extend RUP to add a Maintenance phase that embodies this type of cycle.

2 This list is taken from Ron Jeffries Web site: *http://www.xprogramming.com/xpmag/whatisxp.htm*.

3 XP teams talk about refactoring as a way to remove "code smells." They say that like dead fish, code will smell the longer it is allowed to decay.

4 Taken from *http://agilemanifesto.org/principles.html*.

5 One of the reasons that XP is *extreme* is that you are extreme in the application of the practices by doing them all of the time. Kent Beck described this as "turning the knobs all the way up." This is a metaphor to thinking of a machine with dials for each practice that indicate how much you apply them on your project.

6 An actor is an entity, not necessarily a human, outside of the system being built.

7 Papers have been presented on this subject at many of the XP and Agile Software Development conferences. Look there for specific case studies.

Chapter VIII

Managing Complexity with MDSD

Jorn Bettin
SoftMetaWare, USA

Abstract

This chapter addresses the question of how to successfully create durable and scalable software architectures that enable the underlying design intent of a system to survive over a period of many years, such that no accidental dependencies are introduced as part of further software development and maintenance. The answer involves looking beyond object-orientation and traditional iterative software development. In order to prevent long-term design degradation, and in order to efficiently execute software development in the large, the introduction of dependencies between components needs to be actively managed, relying on a set of guiding principles for component encapsulation and abstraction. The guiding principles required turn out to be a natural extension to the principles of design by contract, they have a direct impact on the modular structure of software source code, and they form a foundation for model-driven approaches to software development.

Introduction

Object-orientation was invented for the purpose of modeling physical systems (Dahl & Nygaard, n.d.), and the concept of models lies at the root of object-orientated approaches. The prediction that objects would lead to substantial software reuse and productivity gains in the software industry has turned out to be wrong. Objects are useful, but industrial use of object-oriented languages has shown that objects are too small-scale for meaningful reuse. The lack of constructs for modularization and encapsulation at a level of abstraction above classes has led to the concept of components where specifications are cleanly separated from implementation and where components can be nested. The concept of nested components is critical for achieving reuse. Nesting enables a group of collaborating components to be packaged as a reusable piece of functionality, and modern object-oriented languages such as Java and C# directly support the component concept in the form of interfaces, Java packages, and C# namespaces.

Unfortunately, components still depend strongly on specific implementation technologies, which makes it difficult to build complex solutions by assembling components from arbitrary sources. Furthermore, practitioners realized that component interfaces expressed as a set of operations do not provide sufficiently precise specifications. This has led to design by contract (Meyer, 1997), and to the distinction between provided and required interfaces that we now find in the UML (*www.omg.org/uml/*).

Motivation to go Beyond Objects and Components

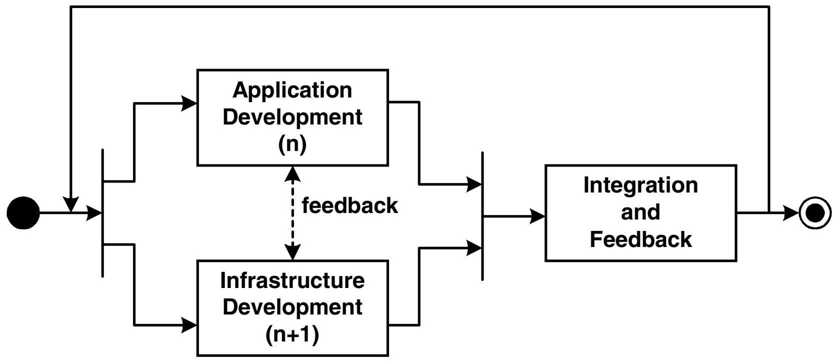
Objects and components together provide one coherent modeling paradigm. A number of scientific and engineering disciplines have developed their own modeling paradigms and notations, which are geared to the specific requirements of the problem domain. Mathematicians use a mature notation that is extremely useful and unlikely to change, accountants have been using spreadsheets for many years and are unlikely to switch to object-oriented programming, electrical engineers are using a standardized notation for designing electronic circuits, and the list goes on. This leads to some important lessons for software development. Perhaps it is not such a good idea to assume that all software development should be ruled by one modeling paradigm. Models expressed in the notations listed above provide examples of non-object-oriented models that are translated mechanically into working code by applications such as symbolic mathematics packages, spreadsheet packages, and electronic circuit production systems.

There are many more examples of highly specialized domains, where there is no industry standard modeling notation, but where it still makes sense to develop a domain-specific modeling notation instead of specifying software using the "one-fits-all" object/component paradigm. The advantage of domain-specific modeling notations lies in their natural fit to the problem domain. This is especially true for those notations that have been developed over many decades by domain experts. Companies want to be able to adapt their software systems to a constantly changing business environment with minimum effort. Ideally this is realized through model-driven applications, where domain experts are provided with modeling tools that allow them to express relevant changes in a notation that is built on familiar domain concepts. This is also known as end user programming. Of course, this only works to a point; it does not extend to changes to the modeling notation—which leads into the topic of domain engineering[1] and software product lines (*www.sei.cmu.edu/plp/*)—that is, sets of software-intensive systems sharing a common, managed set of features that satisfy the needs of a particular market segment or mission and that are developed from a common set of core software assets in a prescribed way.

The Model-Driven Software Development (MDSD) paradigm (Bettin, 2004a) is a pragmatic approach to software product line engineering that avoids the heavy up-front investment into an application platform. It provides an incremental path for the development of suitable domain-specific notations. The underlying philosophy of software product line engineering and Model-Driven Software Development is that every software development initiative beyond a certain magnitude validates the development of appropriate domain-specific languages to express software specifications. The MDSD best practices (VB, 2004) are intended to bring down the costs of building a domain-specific software production facility to the level where domain-specific languages become attractive for many middle-of-the-road software development projects.

A key element in MDSD is iterative dual-track development—that is, developing an application, and in parallel, at the same time, developing the infrastructure for a highly automated application production facility. Both parts are developed incrementally and iteratively as show in Figure 1. In any particular

Figure 1. Iterative dual-track development



iteration, infrastructure development is one step ahead. New releases of infrastructure are only introduced at the start of application development iterations. In practice it is a good idea to use a fixed duration for all iterations, which minimizes the disruption caused by upgrading to new iterations of the production facility.

It is important to realize that the term "domain" need not always relate to a specific industry or a product family; the term domain can just as well relate to a sub-domain of the domain of software engineering. Modern component-based software systems make extensive use of frameworks that implement various technical concerns (i.e., "domains") such as transaction handling, logging, persistence, and so forth. The nature of object-oriented frameworks means that framework usage code is highly repetitive and pattern based. The MDSD approach advocates the development of an appropriate modeling notation that reflects the core framework concepts, which can be used to automatically generate the framework completion source code. Hence, object-oriented frameworks can be considered as the first step towards developing a domain-specific language.

The use of technical object-oriented frameworks is common practice, and the use of MDSD is likely to become common practice in this context within the next few years. Without support by MDSD tools, object-oriented frameworks are notoriously difficult to understand and use, which is why successful frameworks tend to be small and focused. MDSD enables framework developers to do a proper job and shield framework users from framework implementation details. The result is a simplified application development process, where only a few team members need detailed knowledge about the technical object-oriented frameworks used.

Software Development Processes

MDSD is designed for scaling up agile software development to large, multi-team environments involving more than 20 people. The approach is also suitable for smaller teams if they are building a series of related applications or a product that needs to be maintained and supported over a period of three years or more.

There are a number of factors that affect software development in the large, that is, factors that are relevant to MDSD:

- The software development skills vary across the organization. Hence, the notion of a globally applicable software development process that prescribes activities at the micro scale is impractical, as it would unnecessarily constrain the productivity of the most experienced team members (Cockburn, 2001).
- The depth of domain expertise varies across the organization, and typically, the number of true domain experts is limited.
- The work is distributed across several teams that need to be carefully coordinated and managed. Often teams are separated geographically and by time zone differences, and sometimes also by culture and language barriers.
- The number of stakeholders is large. Especially in the case of product development, a formal yet unbureaucratic process is required to manage the diverging interests and priorities of various customers.
- Business milestones are imposed externally and are non-negotiable. Late delivery is unacceptable.
- The software development budget is fixed and is not expected to be exceeded.

Table 1. Comparison of approaches for iterative software development

Provides techniques for	**MDSD**	**SPLE**	**RUP**	**XP**
Dealing with a range of experience levels	+++	+++	+	++
Dealing with a range of depth of domain expertise	+++	+++	+	++
Scaling to a distributed organization	+++	+++	++	+
Balancing the interests of a range of stakeholders	+++	+	+	++
Meeting fixed business milestones	+++	+	++	+++
Maximizing the return from a fixed budget	+++	+	++	+++
Separation of concerns	+++	++	+	
Iterative adjustment of the balance between infrastructure development and application development	+++	++	+	+
Achieving agility	++	+	+	+++
Minimizing the up-front investment required to implement and roll-out the approach	++	+	+++	+++

Given that these are the typical constraints and expectations, the statistics about software project failures should come as no surprise. Table 1 provides an overview of how different software development approaches deal with these context parameters.

The ratings in Table 1 are certainly subjective, and they are only intended to indicate relative strengths and weaknesses, but they do provide insight into the motivating factors that have contributed to the creation of the MDSD paradigm. It should also be noted that MDSD should be seen as a paradigm that is compatible in principle with any agile method, and that is also compatible with RUP. After all, RUP is a process framework, and according to its definition, any process that is use case driven, architecture centric, and iterative and incremental can be interpreted as an instance of RUP. Thus as long as initial functional software requirements are captured in the form of use cases; an MDSD approach is within the scope of the RUP framework. Using a similar line of reasoning, the proponents of RUP have argued that XP can be seen as a minimalist implementation of RUP called dX (Booch, Martin, & Newkirk, 1998). Instead of another instance of a “method war,” it is much more productive to concentrate on techniques from different methods, which can be strung together into a working approach that is suitable for a specific context. The following paragraphs provide a brief perspective on each of the approaches listed in Table 1, highlighting areas where MDSD can be used to compensate specific weaknesses.

Extreme Programming

Extreme Programming (Beck 2000), also known as XP, is a highly disciplined approach that relies heavily on verbal communication, and that assumes above-average skill levels. Hence, XP does not provide concrete guidance in terms of software design patterns and focuses on collaboration, on aspects of project management, and on requirements management. The assumption about above-average skill levels does not hold in a sufficiently large environment and imposes limitations on the scalability of XP. Some XP techniques such as pair programming cannot be applied to distributed teams. The larger the environment, the more verbal communication needs to be supplemented with externalized knowledge and documentation. However, XP encourages timeboxed iterations, and provides concrete techniques for management of scope and priorities that are compatible with the MDSD recommendations for iterative development.

The Rational Unified Process®

The sheer size of the Rational Unified Process® (Jacobson, Booch, & Rumbaugh, 1999) framework means that it can be more expensive to instantiate RUP than to develop a tailored and streamlined software development process from the ground up. Unless an experienced process engineer configures RUP, there is also a real danger that the number of work products to be produced is not trimmed down sufficiently for fear of missing something, resulting in a bloated high-ceremony process. The documentation of RUP contains descriptions of activities at an unrealistic level of detail. Despite this level of detail, RUP provides no concrete guidance in terms of software design patterns, and no support for active dependency management—that is, RUP does not leverage deep domain knowledge (Bettin 2004a). Although RUP is clearly iterative, it does not mandate timeboxed[2] iterations, and it provides very little guidance for managing scope from iteration to iteration. The core value of RUP lies not in the details, but in its core concepts: use case driven, architecture centric, and iterative and incremental.

Traditional Product Line Approaches

Product line approaches are ideal to address the varying skill levels and varying levels of domain expertise in large teams. Product line approaches have been used successfully with distributed teams (Bosch, 2000). A key advantage is that the application development process is not derived from a fixed template such as in RUP, but that instead the application development process is molded around the concrete needs and requirements of the specific context, and is entirely defined by the team that develops the software production facility. The main disadvantage of traditional product line approaches is the lack of agility, in particular the up-front investment into an application platform[3] during the initial creation of the product line. This drastically reduces the practical applicability of traditional software product line approaches.

Non-Software Product Line Approaches

Non-software product line approaches in general, including RUP and XP, do not explicitly distinguish between building a software production facility and building software products or applications. In practice, however, the skills requirements for these two activity streams are different. If the two streams are not separated, it becomes very hard, in a large team, to establish priorities that are fully aligned with the objectives of stakeholders. When using an approach such as RUP, this issue can be addressed by introducing an appropriate team structure that mirrors the setup suggested by software product line engineering approaches.

Model-Driven Software Development

Model-Driven Software Development provides explicit techniques to apply the principles for agile development to software product line engineering. This means the development of a software production facility occurs incrementally, and in parallel with the development of a reference implementation that defines the software architecture for a family of applications or products. One practical lesson from software development in the large is that consistency of work products is only achievable to the degree that consistency can be enforced by the use of tools and automation. Therefore, MDSD has a very strong emphasis on automation and the construction of specialized tools that leverage scarce domain knowledge that traditionally tends to remain locked within the brains of a few team members.

This remainder of this chapter concentrates on techniques for dependency and complexity management that become practical with the help of MDSD tools. The objective of these techniques is the creation of durable and scalable software architectures that enable the underlying design intent of the system to survive over a period of many years, such that no accidental dependencies are introduced as part of further software development and maintenance.

Is this goal achievable? In real-world software projects, the concern about software architecture usually increases with team size, but in the end, when it comes to building a system, software is often cobbled together without effective control over the dependencies that are created between various pieces. In order to prevent long-term design degradation, and in order to efficiently execute software development in the large, the introduction of dependencies between components needs to be *actively managed*, relying on tools for the mechanical enforcement of standards and rules.

Managing Complexity

Model-driven generation can be used to enforce specific patterns and to prevent arbitrary dependencies. That still leaves the work of those who produce prototypes and reference implementations from which generator configurations and templates are derived. If a single person produces the complete reference implementation, and if that person consciously and carefully thinks about every dependency in the design, then all is fine. In reality, consciously managing the impact of every dependency is only possible by adhering to a set of guiding principles for encapsulation and abstraction, even in a single-developer project. The larger the software development effort, the greater the need for conscious dependency management.

The principles required for effective dependency management turn out to be a natural addition to the six principles for design by contract.

- Whereas the *design by contract principles* focus on precision in the specification of component behavior,
- the *principles for fully externalized interface definitions* focus on precision in the delimitation of types that can be used in the communication between components.

The latter principles, which are defined in detail later on, have a direct impact on the package/module/component structure of software code, and therefore their correct application can easily be verified mechanically in the code base. If correct usage is confirmed, then a potential component user can understand what each component is doing by looking at the component specification, without having to look inside. The concept of fully externalized interfaces adds value to all non-trivial software development projects, and is particularly well suited for use in a model-driven software development process.

A major influence on MDSD is the concept of *uniformity* that underpins the KobrA software product line approach (Atkinson et al., 2002), in which every behavior-rich software entity is treated uniformly, regardless of its granularity or location in the "containment tree" that describes a software system. In other words, software entities—which in KobrA are called "Komponents"—have the property of a *fractal*.

Taken together, the two sets of principles mentioned above form the foundation for *Industrialized Software Asset Development*, serving the following goals:

- Promotion of reuse and pluggability of software assets.[4]
- Provision of patterns for the design of Open Source software assets that help maximize the (re)use potential of Open Source software assets.
- Provision of support for the mass-customization[5] of software products (i.e., the management of variability in a software product family).
- Provision of a set of simple rules that limit the creation of dependencies between software assets, so that Model-Driven Software Development tools can be used to enforce software architecture specified in the form of models.

In describing standards for software component development, we work from the inside out: starting with a single component and its specification, then looking at its use, at customization/extension, and lastly taking one step back, looking at the containing component, raising the level of abstraction by one level.

The Problematic Side of Design Patterns

Design patterns—and patterns in general it seems—tend to encourage describing the solution in more precise terms than the problem; that is, by leading from context to the solution, they focus inwards, on the "how" rather than the "what." There are exceptions, but software pattern authors generally tend to prefer precise descriptions of a solution over precise descriptions of the context to which the solution applies. In a model and domain-driven approach however, the need for precision starts at the level of abstraction of the problem space.

If we want to build durable software, we need to use component specification techniques that force us to focus on understanding the "what" first. What services do clients want from my component? And precisely what is the "language" that my clients understand or want to talk in?

A classical design pattern to control and limit dependencies between components/subsystems is the façade pattern (Gamma, Helm, Johnson, & Vlissides, 1994), and that is about as far as most projects worry about dependencies. In fact, the façade pattern is a good example of a pattern that is obviously very useful, but is not specific enough to achieve active dependency management without further qualification. The usage example in the book by Gamma et al. (1994) completely ignores the dependencies created by the types used in the signatures of the operations exposed in the façade. The particular types used in the operations of the example seem to reside inside the subsystem that is supposed to be encapsulated by the façade. The façade pattern says nothing about whether these types must be defined within the subsystem, or whether these types could also be defined externally.

No subsystem lives in a vacuum. If a second subsystem communicates with the first via the façade, it depends not only on the façade, but also on the types exposed via the façade. If the second subsystem makes use of one of these types in its own façade, all of its clients end up depending on the first subsystem. The more subsystems end up depending on the types exposed in the façade of the first subsystem in the way just illustrated (i.e., not as direct dependents, but rather as "second cousins"), the less justification there is for keeping the type definitions in the first subsystem. However, none of the other subsystems would provide a more logical home either. This should be enough to set the scene for the concept of fully externalized interfaces.

What is a Component?

Although the concept of a component has been around for a number of years, the concept is still (2004) not used consistently across the software industry. The problem is mainly educational, exacerbated by the slightly different spin that vendors tend to put on new concepts to differentiate themselves. Just as with object orientation, a lot of damage has been done by hype and setting wrong expectations. At least with the UML standardization effort, vendors now have come to a common ground that is reflected in the UML 2 (*www.omg.org/uml/*) definition of "component":

> *A **component** is a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics).*

Even if this definition is not necessarily a literary gem, it provides a common denominator on which we can build—further relevant OMG specifications of component semantics are part of the UML metamodel and MOF. For practical purposes, and untainted by vendor-specific interpretations, Brown (2000, pp. 62-63) provides an excellent description of the essence of components. We quote the most important points that complement the UML 2 definition:

> *Component approaches concentrate design efforts on defining interfaces to pieces of a system, and describing an application as the collaborations that occur among those interfaces. The interface is the focal point for all analysis and design activities. Implementers*

of a component can design and build the component in any appropriate technology as long as it supports the operations of the interface and is compatible with the execution environment. Similarly, users of another component can define their use of that component by reference only to its interfaces. This provides a measure of independence between component developers, and improves the flexibility of the system as the components are upgraded and replaced… Solutions are designed as collections of collaborating components. However, there are different abstract levels on which this component design may be considered.

The first paragraph of this description simply spells out the basic implications of the UML definition of a component. The second paragraph stresses the importance of the usage environment of components, and the relevance of different levels of abstraction—which is taken much further by KobrA (Atkinson et al., 2002).

Beyond the basic definition of a component, the UML 2 standard provides several improvements for the specification of software components, notably the distinction between provided and required interfaces, and the concept of ports which provides a natural way to precisely model nested components.

Precise Component Specifications

Design by contract (Meyer 1997), which defines a component interface as a collection of operations with signatures, invariants, and pre- and post-conditions, is a good starting point, and can be applied to provided and required interfaces. The design by contract principles are best expressed using the following terminology from Mitchell and McKim (2002):

*A **command** is a stereotype of operation that can change the state of an object but does not return a result.*

*A **query** is a stereotype of operation that returns a result but that does not change the state (visible properties) of an object.*

Armed with these definitions, the design by contract principles are the following (again taken from Mitchell & McKim, 2002):

The Principles of Design By Contract

1. **Separate queries from commands.**
2. **Separate basic queries from derived queries.** Derived queries can be specified in terms of basic queries. In other words, reuse simple queries to build more complex queries, and avoid duplication in query specifications.
3. **For each derived query, write a post-condition that specifies what result will be returned in terms of one or more basic queries.** Then, if we know the values of the basic queries, we also know the values of derived queries.
4. **For each command, write a post-condition that specifies the value of every basic query.** Now we know the total visible effect of each command.
5. **For every query and command, decide on a suitable pre-condition.** Pre-conditions constrain when clients may call the queries and commands.
6. **Write invariants to define unchanging properties of objects.** Concentrate on properties that help the reader build an appropriate conceptual model of the abstraction that the class embodies.

Design by contract is a useful technique as part of a test-driven approach, and just as with any test-driven approach, it is necessary for a project to agree up to what level the technique is applied. In a system of nested subsystems, design by contract adds the most value when applied to the interfaces of the highest-level subsystems.

In fact, writing automated tests can be a very convenient and natural way to provide "sufficiently complete and precise interface specifications" for components. Allowable interaction patterns between a component and its clients are an important part of a component's interface specification. These interaction patterns are ideally captured in the form of tests, which then provide a component user with valuable knowledge on how a component is intended to be used. So writing automated tests adds significant value beyond "testing." It can enable component users to make use of a component without having to dig deep into the implementation to understand it.

As required, the principles of design by contract and test-driven development can be applied at lower levels—for example by pragmatically creating automated unit tests, and writing appropriate constraints to catch defects that have been uncovered in manual function testing.

In order to be able to use a component properly, we need a specification of the intended or valid usage protocol, such as a precise specification of allowable interaction patterns between a client and the component. Above we have already indicated how test-driven development can be used to that end. On the topic of visual, non-code-based specifications of interaction patterns, we refer to the interesting work on CoCoNuts (Reussner, n.d.).

Cleaveland (2001) introduces the useful concept of a dependent-type-set for a component c, which is defined recursively as follows:

1. The type of c is a member of the dependent-type-set.
2. If t is in the dependent-type-set, then so are all superclasses of t.
3. If t is in the dependent-type-set, then so are all types referenced in the source code of t, including the types of parameter values, instance variables, local variables, and each expression.

For many objects in software systems encountered in practice, the dependent-type-set consists of most or even all the types of the system. In other words: these systems are not component-based, as they do not allow reuse of specific objects in the form of self-contained components.

Component-based software development focuses on minimizing the dependent-type-sets of components, so that individual components or small groups of collaborating components can be deployed as a unit without having further external dependencies. When building software using an object-oriented language, there will always be some parts of the software that depend on intrinsic types and a set of base classes provided by the language. It may be appropriate to allow the limited set of intrinsic types and a well-defined set of base classes to be used throughout a system. The use of all other types needs to be carefully managed, and this includes types defined in libraries and frameworks that are part of the wider language environment. Typically, these non-trivial types only need to be used in one or a few specific components, and their use should not be scattered arbitrarily throughout a system.

Hardly any software development project starts with a clean slate. Often a given system exists and needs to be extended, or a new system needs to fit into an existing set of systems and subsystems. This means it makes sense to examine how loose coupling of components can incrementally be achieved in such an environment.

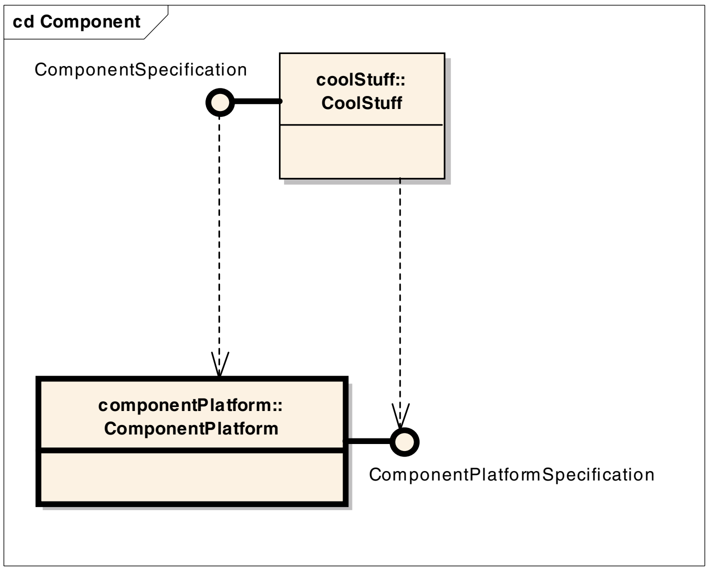
The first step to active dependency management within software is to introduce component interfaces, and then to limit the use of types in component interfaces to well-defined sets. We distinguish two stereotypes of components: <<*sub-systems*>> and <<*platforms*>>, with the former containing the component implementation, and the latter serving as the dependent-type-set for the former. In the remainder of this chapter, we simply talk about *Components* and *Component Platforms* to refer to <<subsystem>> components and <<platform>> components respectively.

Initially the implementation of some Component interfaces may need to be wired up to non-component-based existing code, and we concentrate on defining sensible boundaries and corresponding interfaces for our Components based on the *principles of Fully Externalized Component Interface Definitions*:

The Principles of Fully Externalized Component Interface Definitions

1. The types exposed in a Component interface are all defined in a Component Platform that resides outside the Component. Thus a Component Platform exposes the types contained within it, and is therefore fundamentally different from a Component.
2. Components may either contain realizations of the types used in their interface, or they may obtain access to instances of the types used in their interface via an interface of the corresponding Component Platform.
3. The interfaces of several Components within an architectural layer may share a common Component Platform.
4. For convenience purposes, a Component Platform may be defined as a union of other Component Platforms.
5. The level of abstraction can be raised by constructing a Component Platform from a group of collaborating Components and their common Component Platform, and using a façade and any required helper

Figure 2. Fully externalized interface definitions[6]



classes to lift the API. The implementation—the "contained" Component Platform and the collaborating Components depending on it—is physically contained in a separate implementation package, so that the *macro-level* Component Platform only exposes the types that are used by *macro-level* Components.

Principles 1 and 2 are illustrated in Figure 2.
We see that:

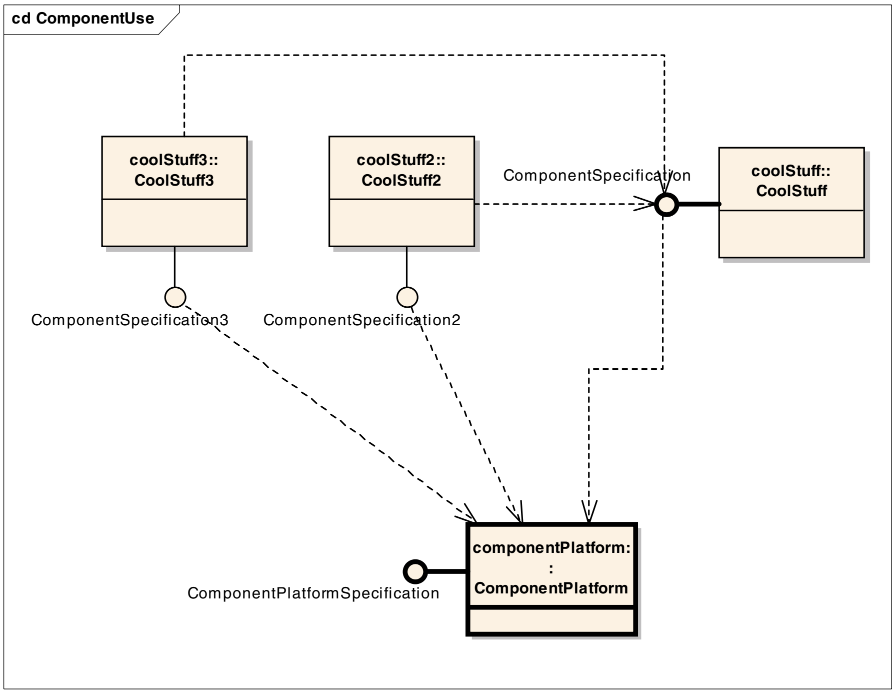
- The ComponentSpecification interface declaration sits outside the component that realizes the interface.
- The types that are allowed to be used in ComponentSpecification (interface) are defined in a separate ComponentPlatform package. These types may be realized in the component (if the ComponentPlatform is a simple collection of type definitions), or they may be obtained via the ComponentPlatformSpecification interface (if the ComponentPlatform is a framework). Note that the ComponentPlatformSpecification interface sits outside the ComponentPlatform package. Hence the "cool stuff" compo-

nent needs to import the ComponentPlatformSpecification interface, so that it can instantiate types defined in the ComponentPlatform package via operations in the ComponentPlatformSpecification interface.

- If our component contains "cool stuff" that we want to promote for widespread use and reuse, then it is worthwhile to consider making certain pieces available in Open Source format (with an appropriate Open Source license, for which the requirements vary depending on the intended usage environment of the component).

To limit dependencies between a set of collaborating components, it is advisable to define a "common currency" for communication between components. This is achieved by mandating that all collaborating components make use of the same well-defined set of types in their public interfaces, which leads us to principle 3.

Figure 3. Component usage environment



The reasoning for introducing the concept of a *Component Platform Specification* separate from *Component Specifications* becomes clear when we look at Figure 3: externalized component interface definitions lead us to think about the usage environment of our component, about "what" our component needs to provide. In Figure 3 there are two components that use the "cool stuff" component, which is indicated through corresponding dependency arrows in the diagram. To keep the diagram simple, we have not shown any potential dependencies on ComponentSpecification2 and ComponentSpecification3. The interesting point to note is that any further dependencies between the three components do not lead to any new dependencies. All types used in the component interfaces are defined in the ComponentPlatform.

The Fully Externalized Component Interface Definitions principles are applicable at all levels of granularity, and they provide a practically usable interpretation of *the relativity of platforms* in MDA (*www.omg.org/mda/*), as we shall see later. Figure 3 also hints at the reasoning behind the specific pieces that are potential candidates for an Open Source implementation.

Using Open Source as a Quality/Standardization Driver

Making a Component Specification and the corresponding Component Platform, including its specification Open Source, promotes competition for cost-efficient implementations. The first step might be a competitor who comes up with a better implementation of our "cool stuff"—it may perform better, require less memory, but adheres to the same specification. In particular if our "cool stuff" has been successful (others have built on it), there will be a strong incentive for the competitor to support the interface of our original "cool stuff" component.

Why would a commercial software vendor be interested in opening up commercially successful software in that way? Think about the following:

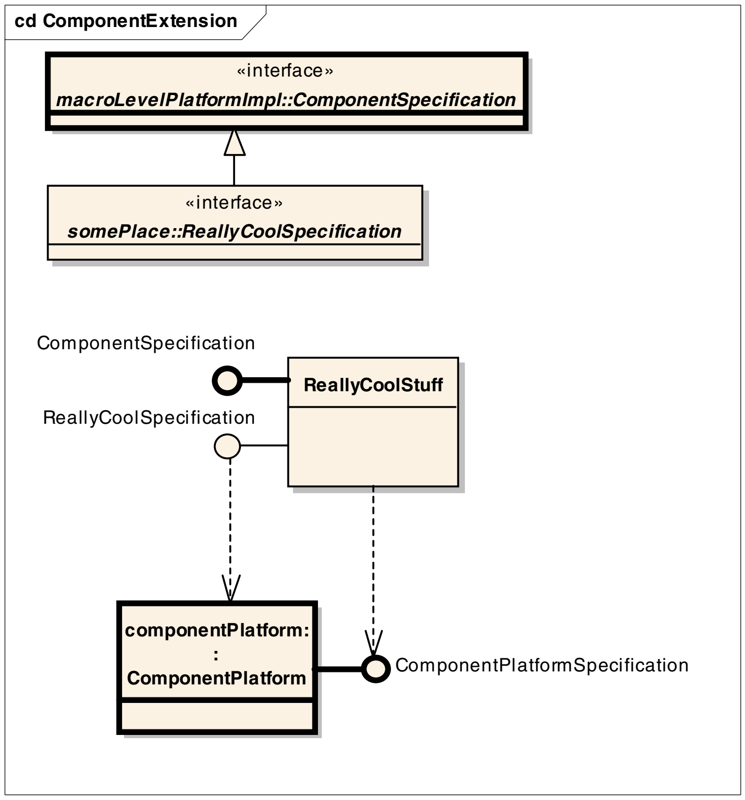
- Making the component specifications available in Open Source form sends a strong message to the market: we have this "cool stuff," and we want you to use it without needing to worry about vendor lock-in.
- Price is not everything. If the potential market is large, a quick expansion and development of the market may lead to more revenue than a high price and a completely closed architecture.

- The timing in making the relevant source code Open Source can be used as a tool to manage revenue. You want to have a fairly well-featured component/product before you Open Source the specification. The competition will take a while to produce a cheaper clone, which provides a window of time for recouping the investment into the first release. In case your implementation is of high quality, the competition might never catch up. Otherwise, as we shall see later, you can use a competitor's improvement as a stepping stone forward.

Extension and Evolution

Usage of a software component in a number of systems and contexts leads to new insights and new requirements, and possibly to a demand for variations of the component. By focusing on the "what," externalized

Figure 4. Component extension

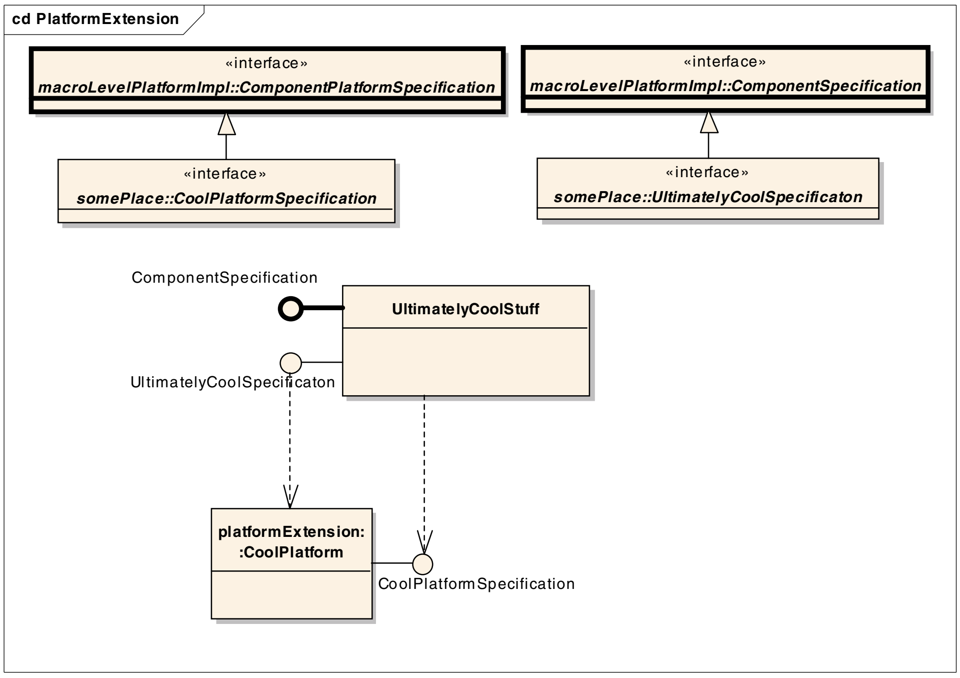

cd ComponentExtension
«interface»
macroLevelPlatformImpl::ComponentSpecification
«interface»
somePlace::ReallyCoolSpecification
ComponentSpecification
ReallyCoolStuff
ReallyCoolSpecification
componentPlatform: : ComponentPlatform
ComponentPlatformSpecification

component interface definitions lead to the expression of variabilities in the language of the problem domain rather than in the language of the solution domain. The simplest form of evolving a component is by extending the interface as shown in Figure 4 to provide "really cool stuff."

In this scenario, the Component Platform stays the same. The new ReallyCoolSpecification may or may not need to be Open Source, depending on whether the ComponentSpecification is Open Source, and depending on the flavor of the Open Source license used. For backwards-compatibility, and to enable incremental migration of clients, the "really cool stuff" component not only implements the new ReallyCoolSpecification interface, but also implements the ComponentSpecification interface.

Open Source licenses that require extensions/modifications to be bound to the same license are beneficial to the entire software community if some organization develops a useful extension. This may mean that a competitor overtakes the original developer of CoolStuff. However, this model also enables the original developer to leap onto any improvements and take them to the next

Figure 5. Component Platform extension


cd PlatformExtension
«interface»
macroLevelPlatformImpl::ComponentPlatformSpecification
«interface»
macroLevelPlatformImpl::ComponentSpecification
«interface»
somePlace::CoolPlatformSpecification
«interface»
somePlace::UltimatelyCoolSpecificaton
ComponentSpecification
UltimatelyCoolStuff
UltimatelyCoolSpecificaton
platformExtension: :CoolPlatform
CoolPlatformSpecification

level. Hence the resulting competition is constructive, not destructive; it enables the software community to build on the achievements of others; and market economics automatically lead to the selection of the most maintainable implementations.

The next possible kind of component extension involves an extension of the Component Platform to be able to deliver "ultimately cool stuff."

In this scenario, the comments and incentives regarding Open Source are the same as in the previous scenario: depending on the licensing model, the CoolPlatform may or may not be Open Source.

Stacks of Component Platforms

The concept of fully externalized interfaces can be applied at all levels of granularity, and we start again with the picture of collaborating components to illustrate this point.

To reduce the level of coupling for a client component that needs access to functionality of all three components in Figure 6, it may be advisable to wrap the three components into one "Macro-Level" Component Platform, which leads to Figure 7. Basically, this is a form of API lifting.

Figure 6. Macro-level platform

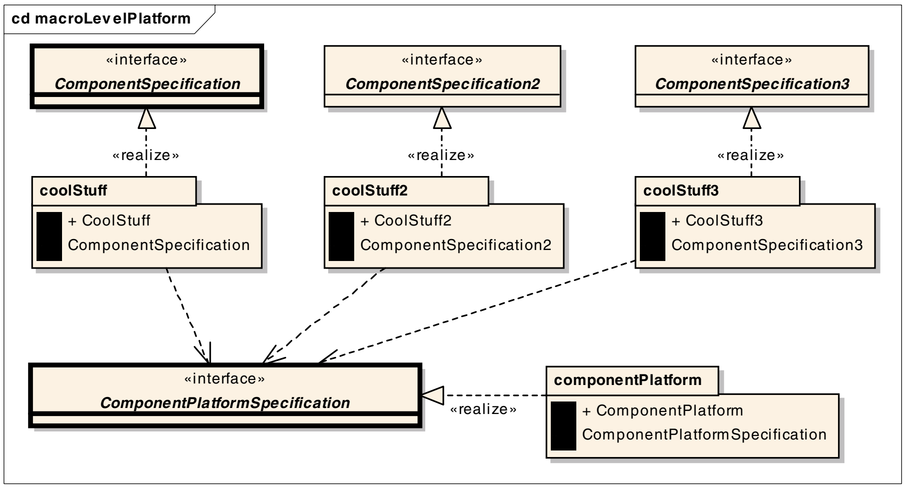


Figure 7. Nested platforms

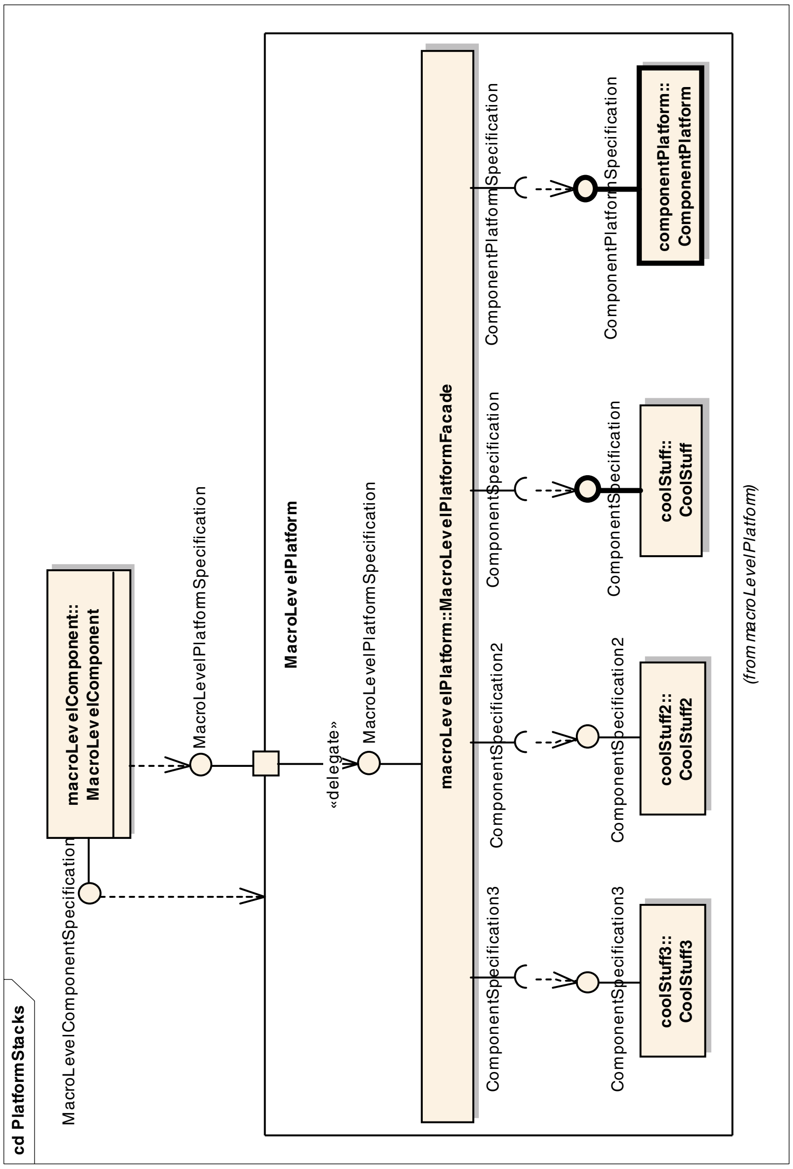
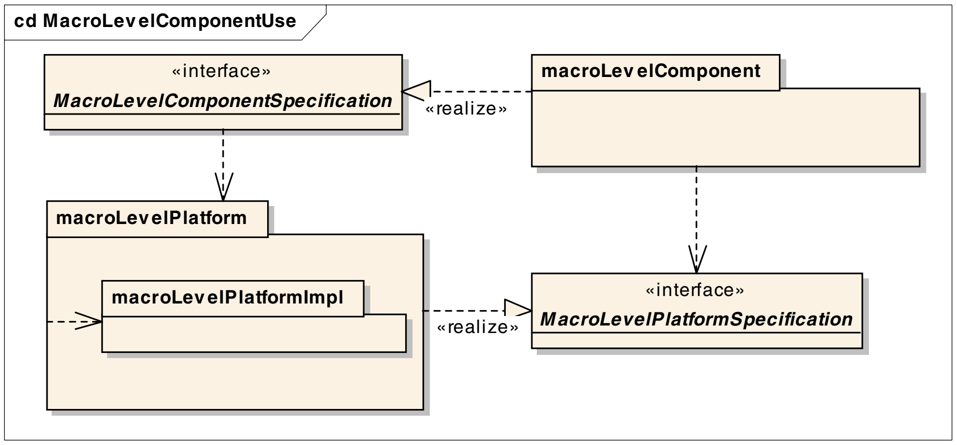


Figure 8. Macro-level component use



The façade depicted in Figure 7 does not have to be a single class in all implementation scenarios. Also the diagram does not show details such as all the other types beyond the façade that are defined in the macroLevelPlatform, which are used by macroLevelComponent. The Macro Level Platform in Figure 7 is implemented with the help of a macroLevelPlatformImpl package (Figure 8), which is the container of the content in Figure 6.

We now have a scalable and well-defined technique for managing dependencies. The interfaces of all our components have well-defined and limited dependent-type-sets, and the implementation of our components can be exchanged without affecting clients of our components.

Full componentization is achieved in a second step, by also channeling all dependencies of a component's implementation through a component platform, which then defines the dependent-type-set of the entire component, and starts to grow from a simple collection of types into a framework. In summary this means:

1. Start componentization by defining appropriate component boundaries based on available domain knowledge. Usually some domain analysis is required to uncover domain knowledge and to represent it in formal models. A combined top-down and bottom-up approach assists in the identification of appropriate components for various architectural layers.[7]

Incrementally put in place appropriate component interfaces using the principles of fully externalized interfaces, and route communication between components through these interfaces to limit the dependent-type-sets of the component interfaces.

2. Incrementally refactor the component implementations, to limit the dependent-type-sets of each component by removing external dependencies and evolving the relevant component platforms. Start with those components that cause the most pain (maintenance costs), and continue until the dependent-type-sets of all strategic[8] components are restricted to well-defined component platforms.

Modern object-oriented languages that directly support the basic component concept via packages and interfaces allow a straightforward implementation of component platforms and stacks of component platforms.

The difficulty in practice is to enforce the principles of externalized interface definitions in large team environments where resources of various skill levels are working with a code base. This is where MDSD (Bettin, 2004b) is extremely useful, as it can be used to introduce component specification languages that represent the various stereotypes of components that occur in the software architecture of a product or product family as first-class citizens—the application modeling language used by application developers then typically does not allow creation of "components," but only allows the creation of "Enterprise Components," "Business Components," and so forth by selecting from a list of component stereotypes that is predetermined by the software architect (Bettin, 2003). The connections between components can be wired-up by defining and using a module interconnecting language as illustrated in Cleaveland (2001).

Practical Considerations

Components Supporting Multiple Interfaces

The techniques described in the previous sections are applicable to both peer-to-peer communication between components and to communication between neighboring architectural layers. This may mean that a given component may need to expose two or more types of interfaces: one for peer-to-peer communication, and one for each architectural layer that touches the component.

Each type of interface may require a quite different Component Platform. This is especially obvious when constructing a domain layer, where functional components have peer-to-peer interfaces—which should be as coarse grained as possible, and where additionally, application and presentation layers sit on top of the domain layer and require fine-grained access to domain objects so these can be represented/manipulated on the user interface.

There is nothing that prevents the externalization of interfaces for each interface of a component, and hence a component may be associated with multiple component platforms.

A component platform may be a simple collection of types, or it may be a full-fledged application framework. The evolution of a component platform is determined by the characteristics of the components that are built on top of it. If the components are sufficiently different, the component platform may always remain a simple collection of types. If however there are significant commonalties in the components using the component platform, then an evolution towards an application framework is a natural consequence. Insisting on externalized interface definitions not only serves the goal of active dependency management, but also reduces the necessary amount of refactoring required to distill an application platform from a set of related components. It is worthwhile noting that the distillation of an application framework not only reduces the size of the code base, but also may eliminate the need for some peer-to-peer component interfaces.

Mass Customization

The techniques described for component extension and evolution can obviously be used to define variants as required. A component specification may however need to accommodate several independent variabilities, resulting in a huge number of possible variations. For example, a component specification may allow, for example:

- a choice of locale to determine the language in user interfaces and documentation,

- a choice of target technology environment—Microsoft DotNET or J2EE, or
- a choice of various persistence mechanisms.

In this type of situation, which is the typical mass customization scenario, it is impractical to rely on reuse of pre-fabricated components. Instead, the implementation of decisions that resolve the variabilities in component specifications are more practically addressed just before compile-time (at *component configuration-time*), or in some cases even at run-time.

A model-driven variant configurator that implements a decision model for a software product family, in conjunction with a model-driven generator, is a powerful tool to assist with automated component configuration and assembly. If a code generator is used to configure a component based on the values in a decision model, then component-configuration-time is equivalent with *generation-time*. The generator reads the decision model, and is used to generate necessary configuration information and any required code to glue the generated components to the relevant component platforms. Additionally, for a product instance, the generator may produce at least the skeletons of any necessary code artifacts that are not covered by domain-specific frameworks contained in component platforms.

Componentization in Non-Software Product Line Settings

Not many software organizations develop large software product families and make use of software mass customization. What value does componentization add in other settings? The answer depends on the scale of the software that needs to be supported, maintained, and developed. The scale is determined not only by the physical size of the code base in terms of lines of code or a similar measure, but it is also determined by the expected useful life of the software. If a software system is built as a one-off system that is only expected to be used for, say, two years, then componentization may not add much value. However, only very few systems fall into this category. It is usually much too expensive to develop software that has such a short life span.

Most software systems are expected to last at least three years, and in many cases software systems remain in use over 10 years or longer. The reason why parts of software systems are sometimes redeveloped from scratch is because software maintenance and enhancements could not keep up with the speed of change of requirements, or with the speed of change in the prevalent technological environment in which the software needs to execute.

Non-componentized software is a major reason for inability to economically implement software changes, because spurious complexity is introduced through unlimited dependent-type-sets. This means that changes are accompanied by unpredictable side effects, and it means a highly time-consuming trial-and-error coding cycle for the software development team. In the absence of well-defined component boundaries that provide known limits to the impact of a change, it is a sheer impossibility for software developers to fully understand the implications of a code change in a large base of non-componentized software.

Thus any software that has a life expectancy of three years or longer, and that consists of several 100,000 lines of code, is worthwhile to componentize.

The low life expectancy that some organizations attach to their software is the result of experience with non-componentized software, which for reasons explained above has a natural tendency to become less and less maintainable, and therefore typically leads to a complete re-write at some point. However, componentization is no silver bullet:

- If reasonably componentized software is neglected, it can easily degrade into non-componentized software.
- There is no hard boundary between "non-componentized" and "componentized," there are various grades from poorly componentized to fully componentized, and different people may have different opinions of what represents a sufficient degree of componentization.

Component boundaries need to be drawn to maximize separation of concerns, and need to take into consideration the commonalities and variabilities between components, both at a technological as well as on a functional level. Arbitrarily componentized software will be nearly as hard to maintain as non-componentized software. In the end, hard economic numbers count. If maintenance costs are not sustainable, the reasons can be any combination of the following:

Problem	Solution
1. The software is insufficiently or inappropriately componentized.	Identify and replace software liabilities9 through incremental componentization. Introduce economically driven build/buy/Open Source decisions into the software development process.
2. There is a lack of quality assurance measures, and low quality code leaks into production software, making defects very expensive to fix.	Institute a rigorous and effective QA regime. *Note:* This problem often goes hand in hand with problem 1.
3. The software development process is overly bureaucratic.	Adopt an agile approach; eliminate all "write-only" work products that no one reads.
4. The software is feature-saturated, and the value of nice-to-have features to the business is less than the cost of developing new features.	Stop developing new functionality. *Note:* This is a rare problem, and it can be the perception of an organization where the real problem is 1.

References

Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wuest, J., & Zettel, J. (2002). *Component-based product line engineering with UML*. Reading, MA: Addison-Wesley.

Beck, K. (2000). *Extreme Programming explained: Embrace change*. Reading, MA: Addison-Wesley.

Bettin, J. (2004a). Model-Driven Software Development: An emerging paradigm for industrialized software asset development. Retrieved from *www.softmetaware.com/mdsd-and-isad.pdf*

Bettin, J. (2004b). Model-Driven Software Development teams, building a software supply chain for distributed global teams. Retrieved from *www.softmetaware.com/distributed-software-product-development.pdf*

Bettin, J. (2003). Best practices for component-based development and Model-Driven Architecture. Retrieved from *http://www.softmetaware.com/best-practices-for-cbd-and-mda.pdf*

Booch, G., Martin, R.C., & Newkirk, J. (1998). The process. Retrieved from *www.objectmentor.com/resources/articles/RUPvsXP.pdf*

Bosch, J. (2000). *Design & use of software architectures, adopting and evolving a product-line approach*. Reading, MA: Addison-Wesley.

Brown, A.W. (2000). *Large-scale component-based development*. Englewood Cliffs, NJ: Prentice-Hall.

Cleaveland, C. (2001). *Program generators with XML and Java*. Englewood Cliffs, NJ: Prentice-Hall.

Dahl, O.-J., & Nygaard, K. (n.d.). How object-oriented programming started. Retrieved from *heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html*

Cockburn, A. (2001). *Agile software development*. Reading, MA: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development Process*. Reading, MA: Addison-Wesley.

Meyer, B. (1997). *Object-oriented software construction*. Englewood Cliffs, NJ: Prentice-Hall.

Mitchell, R., & McKim, J. (2002). *Design by contract by example*. Reading, MA: Addison-Wesley.

Reussner, R. (n.d.). CoCoNuts. Retrieved from *www.dstc.monash.edu.au/staff/ralf-reussner/coconuts/coconuts.html*

Endnotes

1 In Software Product Line Engineering terminology, Domain Engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of a system in a particular domain in the form of

reusable assets (i.e., reusable work products), as well as providing adequate means for reusing these assets (i.e., retrieval, qualification, dissemination, adaptation, assembly, and so on) when building new systems.

[2] Timeboxing can be used to achieve a regular cycle of software delivery and to establish a familiar rhythm in an iterative process. Over the course of a few iterations, teams gain speed by following a well-engrained routine, and the easiest way to coordinate multiple teams is through synchronized timeboxes.

[3] An application platform is the set of core assets in a product line that is designed for reuse across members of the product line.

[4] Model-Driven Software Development defines a *software asset* as "anything from models, components, frameworks, generators, to languages and techniques."

[5] *Mass customization* meets the requirements of increasingly heterogeneous markets by "producing goods and services to match [an] individual customer's needs with near mass production efficiency."

[6] In the UML diagrams in this section and in all the following sections, dependency arrows indicate <<import>> dependencies at the source code level, and code that may benefit from publication using an Open Source license is indicated using bold lines.

[7] In a *layered architecture*, source code is organized in a hierarchy of layers, with code in each layer only dependent on code in the layers below it, ideally only on the layer immediately below it.

[8] Model-Driven Software Development defines strategic assets as the software assets at the heart of your business—assets that grow into an active human- and machine-usable knowledge base about your business and processes.

[9] A software liability is software that is a cost burden, that is, software that costs more than it is delivering in value to the business.

Chapter IX

Agile RUP: Taming the Rational Unified Process®

Gary K. Evans
Evanetics, USA

Abstract

The Rational Unified Process® (RUP) is the de facto iterative software development process in use today. But it is huge (over 3,200 files), prescriptive, and generic rather than concise, agile, and specific. Organizations moving to RUP are often confused about how to apply it to their culture, unsure how much of it to adopt, and wary of how they can adapt it to their specific software projects. This chapter starts with a brief summary of the traditional waterfall development process, then offers an overview of RUP, its philosophy and distinctive features. Then the general philosophy of agile development is discussed. The body of the chapter defines a small set of activities that have been successfully applied in commercial "Agile" RUP projects by the author. The chapter then discusses how some of the major stakeholder groups on a software project are affected by moving to an agile process.

Introduction

The Rational Unified Process® (RUP) and the Unified Modeling Language (UML) are present in some way on many object-oriented projects today. UML is a visual notation for concisely expressing object-oriented concepts. RUP is a process description that uses UML to express the content of its own artifacts and tasks. RUP is not so much a specific process, as it is a description of a *process framework*. It is large and complex (over 3,500 files in more than 200 folders) because it is a generic framework. But despite its girth, RUP incorporates just a few very basic principles and advocates the best practices of [RUP]:

- Develop Iteratively
- Manage Requirements
- Use Component Architectures
- Model Visually
- Continuously Verify Quality
- Manage Change

These practices are predicated on the principles that an effective process should be:

- **Iterative:** Do the same activities in small pieces, over and over.
- **Incremental:** Gain a bit more understanding of the problem, and add a bit more solution, at each iteration, building on what was done previously.
- **Risk-Focused:** Address risk early and often, focusing on the most architecturally significant properties of the system, and the highest risk areas before developing the easy, "low hanging fruit" of the system.
- **Controlled:** Control the process to meet your needs, do not allow the process to blindly control you.
- **Use Case (i.e., requirements) Driven:** The goal is established by the total requirements, and the operational requirements are captured in a form known as use cases.
- **Architecture-Centric:** Architectural integrity and stability are emphasized over ad hoc software design details.

One of RUP's appealing characteristics is that it emphasizes activities over documentation. This observation always surprises those learning RUP for the first time because of the many templates, diagrams, and other artifacts that RUP employs. But it is the activities, and the roles that perform those activities, that produce the many documents defined by RUP. And it is a very legitimate question whether a group using RUP should do all the defined activities, define all the RUP roles, and produce all the RUP artifacts.

The answer is "No," and RUP clearly dictates that its contents must be tailored to specific organizations and projects. Companies using RUP should define a RUP *development case* that specifies those parts of RUP that the company or project will use, those parts that will not be used, and non-RUP artifacts and activities that will be added to the project.

So the very definition of RUP accommodates flexibility: it is designed to meet your organization's needs, not designed to force your organization to RUP's very comprehensive, yet generic definition. This flexibility is critical to the future success of RUP. I approach RUP as a catalog of best practices, received wisdom, and some interesting new insights into the process of software development. But while RUP is a flexible framework, it is still a *prescriptive* definition of software development. By prescriptive, I mean it pre-scribes (literally, *writes before*) with some rigor what should be done, who should be doing it, and when it should be done. Not that a prescription is inherently bad: for those companies or projects in the seat-of-the-pants development mode, having a checklist and tour guide such as RUP can dramatically improve their chances for success in transitioning to an actual, defined process. But for projects that have a need for a looser, less prescriptive, and lower overhead personality, out-of-box RUP must be either abandoned, or defined in a lighter, more facile way. The remainder of this chapter will address the latter option: making RUP *agile*.

Agile RUP

Agile software development is both a process issue and a mindset issue. In mindset it appropriates the best of the "let's try something different" attitude, and abandons the worst of the "you have to do it this way" attitude. Philosophically, the agile approach is very humble:

- There is no single best way to build software.
- You cannot really plan the execution of the project until you are doing the project.
- If you need to do something to produce great software, do it; otherwise, do not do it.

But these simple statements can seem wildly radical for organizations that accept the received wisdom of the last 30+ years that says software is manufactured, or engineered, much as automobiles and airplanes are engineered. But those of us who do software for a living have long recognized that how we really develop software does not fit neatly into this manufacturing box. For many of us it is self-evident that software is organically composed, like music or a short story, rather than linearly manufactured, step by step, like bridges and automobiles.

How do I determine if I need to do X, or Y, or Z? There is an inevitable tension in answering this question, because in the discourse of software process improvement, agility is a disruptive element. The agile philosophy promises that "doing less will get you more." It sounds a little too good to be true. Yet, the principles and practices of agility assault our thinking and our comfort zones infinitely more than they affect our code. Agility does not just change how we write code—it changes everything about how we develop software, and I will discuss later how several of the stakeholder groups in software development are affected by agile practices.

But the agile approaches are relatively new and popular mostly among early adopters. Most commercial software today is being developed with either a seat-of-the-pants approach (that is, no defined process) or with the venerable *waterfall* process. As anecdotal evidence I submit the following experience.

In April 2002 I participated in a panel on software development methods at the Software Development Conference in San Jose, California. Before the panel discussion started, the moderator asked for a show of hands from the attendees about the development process they were using in their work. About six hands were raised to signify eXtreme Programming (XP) or a variant; about a dozen or so hands were raised for agile development processes of any kind—including RUP. But most of the 800 people attending raised their hand for waterfall or variants of waterfall. My consulting experience, however, suggests that many of those acknowledging waterfall or some variant thereof were really uncomfortably close to the seat-of-the-pants approach and thought it was properly called "waterfall."

The traditional waterfall approach has been validation "in the large." This is carried out by the practices of "Big Requirements Up Front" and "Big Design Up Front." These are justified by the twin assumptions that: a) we must understand everything before we start building the system; and b) if we write it down and "freeze" the content, it must, therefore, be correct. The waterfall approach implicitly assumes that software is manufactured, that it is actually possible to understand everything up front, and that it is possible to do all the design before coding, and all the coding before system test, and so on.

Agile approaches perform validation "in the small." They advocate appropriating a "good enough" understanding of the problem to be solved and building our software in small steps. The agile mindset is a humble view: it simply says that we cannot understand everything about our system until we actually build the system. And to build the correct system, we must build it in small, verifiable pieces that we continuously validate against the customer's expectations.

Simply put, in the waterfall approach we try to understand so we can then go forward to build the system. In the agile approach we begin building so we can understand. In the traditional approach we do not expect, nor schedule time to go backward and make changes to what we have already done (because we should have figured it all out before we started). But in the agile approach, we go forward so we can justify going backward to fill-in what we inevitably missed in a previous step. This backtracking is planned. As the eXtreme Programming community says, agility embraces change.

This is the essential difference between planned iteration and the unplanned rework so common in waterfall-based projects. In any prescriptive approach, adherence to the prescribed software process is considered the major determinant of success. In the agile approach, adaptation toward achieving the end-goal—working software—is the major factor in success.

Table 1 summarizes some contrasts between these remarkably different approaches.

Table 1. Contrasts between the prescriptive approach and the agile approach

	Waterfall	**Agile**
Guiding Metaphor	Manufacturing/engineering	Organic/emergent
Focus	Documentation, schedule	People, Working code
Dynamic Structure	Cause & effect, Preventive approach	Chaordic (Ordered chaos), Adaptive approach

Making RUP Agile

My goal in agile RUP has always been to keep the best of RUP, but reduce, or eliminate, its prescribed tasks in the most agile manner possible. An instructive example of this is the issue of software modeling. In the process description below, I clearly advocate the practice of modeling software. While the eXtreme Programming community diminishes or eliminates the practice of software modeling, my experience clearly demonstrates that modeling is an essential practice for typical development organizations populated with mere mortals rather than software gods. But I do not advocate modeling for the sole purpose of constructing models. Rather, the models should be produced only if they actually help you to better understand both your problem and solution, so they can directly help you produce the right code. Similarly, I do not advocate using highly structured, large-overhead document templates, or capturing every project artifact in a CASE tool. This kind of over-control can guarantee the death of a project before it even gets started. Until a model is stable, continually updating its representation in a CASE tool can quickly drain a project's energy from its real goals of understanding what is to be built, and actually building it. Agile RUP says yes, capture meaningful models in a CASE tool, but only after the models have stabilized. It is not at all uncommon for a domain class diagram to go through dozens of versions before it starts to stabilize. Until it does, just maintain the model on a large white board and keep revising the white board. Think agile, and do agile. For an excellent discussion of the practices of agile modeling, see AMBLER.

The following section offers a brief outline of an agile RUP process that I follow in my software development projects with clients in diverse industries. This description assumes you are familiar with core UML elements such as sequence diagrams and use cases.

And I make an immediate disclaimer here: I am focusing on the actual tasks of software development, not on the RUP's supporting disciplines of configuration management, project management, environment, and so forth. These disciplines certainly are important in an agile RUP process, but the scope of this chapter is to describe the minimum tasks needed to build executable software within an iteration.

Getting Prepared

Figure 1 lists activities you do at the beginning of the project, to understand enough of the goals and "big picture" so you can confidently establish an *initial* plan for iteratively building the system. We are not doing any code development yet. We are exploring the breadth and depth of the system's requirements and goals. This initial plan will change. In fact, if your initial plan does not change through the life of your project, you are most certainly still adhering to a waterfall mindset. In an agile project the initial planning is done relatively quickly, with a minimal investment of time and resources, because of the reality that the plan will change. My rule of thumb from my projects is to spend three to five weeks on the initial planning for a team of five to six people, with an estimated system size of up to 250,000 lines of code.

For Each Iteration

As you complete the last task above, you will have an initial project plan enumerating the goals of the project, your overall staffing needs, the current known risks, and planned deliverables and delivery schedule. This schedule will be supplemented with an iteration schedule listing each iteration, its staffing and duration, and the services that will be written in that iteration.

This initial plan is our best estimate of the breadth of the project, but it is necessarily very shallow. Now we are in a position to provide some depth to the description of our iterations. The steps in Figure 2 are performed as a group in each iteration, so you will perform all nine steps 15 times if you have 15 iterations. As executable code is delivered from one iteration, the next iteration builds on that one, until all functionality is delivered in the last iteration.

Figure 1.

Task/Activity	**Description**
a) Identify the architecturally significant functional requirements for the system, and the major non-functional requirements for the system.	The non-functional requirements (e.g., scalability, reliability, performance, security, etc.) determine the architectural choices you will make. The functional requirements describe the value the system will provide to its users. The *architecturally significant* functional requirements are those that provide the most important or most frequently requested services to the user, and will determine the chosen architecture from the choices available. Services visible to end users are captured as *use cases*. Do not try to embrace every use case or every detail in the beginning. Be selective and focus on those 20% of use cases which give you 80% coverage of your major system services. During the iterations you can add in the less significant use cases.
b) Identify the major business abstractions that are part of the domain for which your system is being developed.	These business abstractions are the "things" in your system and will become the *classes* in your system. An insurance domain includes policies, owners, coverages, claims, and so forth.
c) Define the responsibilities and relationships for each class in the domain.	Classes have three basic properties: *structure* (the data the class owns), *behavior* (the services the class provides through its methods), and *responsibilities* (the justification for the class's existence). The responsibilities of the class determine the other two properties. The class's relationships are derived from its responsibilities, and usually represented through the class's data (e.g., as references to other classes/objects).
d) Construct the initial domain class diagram for your system.	The domain class diagram is a UML analysis artifact, not a design or technology artifact. It captures the business abstractions in your domain, and the relationships among these abstractions. Each class must have a responsibility specification, but very little if any internal data or functions are defined yet. The value of this diagram cannot be underestimated: it is your source of agreement with your business people on the "things" in your system, and provides a common metaphor for the entire development team.
e) Identify the major risk factors on your project, and use this risk profile to prioritize the most architecturally significant use cases.	Now you know enough about the major goals and services of your system to determine which use cases contain architecturally significant functional requirements. You should also have an initial understanding of the major business and high-level technical risks on the project. Start a written Risk List, and write the content of the architecturally significant use cases.

Figure 1. (continued)

Task / Activity	Description
f) Partition the major use cases, and their contained scenarios, across a timeline of planned iterations.	Based on estimated or known team size, allocate the architecturally significant use cases (or scenarios, which are specific paths through a use case) into time-based "buckets". These buckets are *iterations*, within which your team will analyze, design and implement a small subset of the requirements of the system. Each iteration is a short-duration project in itself. Determining how many iterations you will need is totally subjective at this point. You will change the number and duration of these planned iterations as you actually measure the development team's ability to deliver functionality (i.e., their *velocity*). The iteration plan at this point is analogous to a Table of Contents, rather than a deeply researched article. For each iteration you have identified, describe the goals of the iteration, the staffing, the schedule, the risks, inputs and deliverables. Keep the iterations focused and limited (I prefer three to four weeks per iteration). Your goal is to make each iteration a "mini-waterfall" project so you can "eat the elephant one bite at a time!" Each iteration description should cover all of the software activities in the process: requirements, analysis, design, implementation and test. Each iteration will also involve QA, Development, Product Management, etc. and each iteration will produce an executable. Using iterations is a "divide and conquer" strategy, and allows you to know within days or weeks if you are getting off schedule.

What All This Means for the Project Team

Now that we have a better appreciation for how to apply an agile personality to RUP, let us look at how the agile, iterative approach affects various stakeholder groups in the software development process.

Development Team

Too often, programmers, designers, and architects work in isolation. Many times, corporate policies actually encourage the "lone hero" persona. Performance appraisal practices are invariably focused on individual merit and on an

Figure 2.

Task/Activity	**Description**
1. For the use cases in this current iteration, construct analysis-level interaction diagrams (i.e., UML sequence diagrams or collaboration diagrams) to merge the functional flow in the use cases and scenarios with the classes and relationships in the domain class diagram.	Use cases capture services visible to an end user. These services must be carried out by the classes in your system. UML interaction diagrams illustrate how your classes will carry out the work of your use cases. Developing these diagrams will allow you to discover operations, which meet the business requirements in your requirements artifacts.
2. Test and challenge the analysis-level interaction diagrams.	Your analysis classes, their responsibilities, and their interactions must all work together to meet the business goals of your system. Review the use cases and the interaction diagrams with your business analysts, and customers, to validate that your understanding is congruent with theirs.
3. Develop analysis-level statechart diagrams for each class with “significant” state.	Not all classes have state. But for those that do have state-driven behavior, developing a UML statechart diagram is very useful. Statecharts provide information on new operations for the class being modeled, especially private operations.
4. Develop design-level interaction diagrams and statechart diagrams with technical content.	When your customer and business analysts agree your models capture the business problem to be solved, it is time to take technology issues into consideration. Analysis models describe “what” the problem is. Design-level models describe “how” the solution will be defined. In this step you will introduce platform, language, and architectural artifacts, for example, collection classes to manage one-to-many and many-to-many relationships, or Enterprise Java Beans and Application Servers for those platform component models.
5. Challenge the design-level interaction diagrams and statechart diagrams for the iteration’s use cases, discovering additional operations and data assigned to your classes; produce a design-level class diagram with these operations and data.	This update cycle is the heart of iterative design: look at your system using interaction diagrams, introduce new design classes as necessary, assign responsibilities and operations as needed, then add these discoveries to the (now) design-level class diagram.

employee’s ability to “stand out” from the rest of the team. Developers usually are not chastised for working alone, or for working too many hours. We love heroes and heroic effort. We are suspicious of someone who cannot “do it alone.” And worst of all, in the non-iterative world where project status is equated with signed-off documentation rather than delivered code, developers can hide behind documentation reviews and long, vague schedules.

Figure 2. (continued)

6. Develop the code for the use cases in the current iteration from the current design-level diagrams.	You have done enough at this point with the use cases and classes for this iteration. Now, go and write the code that implements the design decisions you have captured in your agile models. Do not try to make the models perfect, or even complete. The models' purpose in life is to help you understand what is needed so you can write actual, working code.
7. Test the code in the current iteration.	You have to test the code you have written. In the agile world, however, we often write our tests before we write our system code. This is an approach called *Test-Driven Design*, and two excellent introductions to this development practice are ASTELS and HUNT. By testing against the known requirements, you can validate whether you have produced the features you were supposed to in this iteration.
8. Conduct an iteration review	This is a short review. I ask only five questions of the development team: • Did we achieve the iteration goal? • What went right (and why)? • What went wrong (and why)? • What do we want to change for the next iteration? • Do we need to update the project plan? The last question sends tremors through management, but it is the most important question. Better to honestly adjust the plan based on our learning from the current iteration than to stick our head in the sand and ignore the forces beyond our immediate control.
9. Conduct the next iteration (i.e., go to Step 1) adding in the next set of use cases or scenarios until the system is completely built.	The iteration plan for the next iteration is actually constructed in this current iteration. Only in this current iteration can you really understand what needs to be done next. The plan may or may not be affected by the results of the review of the current iteration. This loop is the heart of an iterative, incremental development process: do the same things over and over, building in the current iteration on your results from previous iterations.

In the agile, iterative world, developers are challenged to let go of:

- seeking details too soon,
- designing before the problem is defined,
- isolation and working alone,

- writing code, and then
- trying to devise tests.

In agile projects, developers experience total exposure: if you do not deliver code every few weeks that fulfill the requirements of the iteration…you have failed. There is no middle ground, no gray area to "spin" the results. As Martha and the Vandellas recorded in Motown, there is "no where to run, no where to hide…."

In an iterative project, real, working software is delivered at every iteration. This software may only contain a few new features. It may be just "plumbing" with little business logic. But real, working software is delivered at every iteration. First, the interfaces are proven. Then, integration of the components is proven. Then, complex algorithmic processing is added. Software is available for review at every iteration. Project status is not determined by signed-off documents promising code. Project status is demonstrated with working code. For the development team this means "victory" is achieved again and again at each iteration. Developers love real results, and on well-run iterative projects, morale is usually very high because real code is delivered early and often.

Business Analysts

Traditional, waterfall-process projects invariably over-invest in what is called Big Requirements Up Front (BRUF). Large requirements specifications are produced at the beginning of the project before any code is allowed to be written. These specifications are usually produced by many analysts and subject matter experts.

In an agile, iterative project, Business Analysts (BAs) and Project Managers do not have to have all of the requirements up front. Developers do not need all of the requirements up front. They just need the major requirements up front. They will solicit requirements as needed from the BAs. BAs just have to be able to find, supply, or obtain the requirements when they are needed. The philosophy here is that software development is not manufacturing, it is an organic, emergent process of discovery.

This statement that all project requirements are not needed up-front is, in my experience, the single most contentious area for groups moving to agile process. BRUF is a habit (or crutch) that most organizations find very difficult to give up. I worked on an ambitious, $5 million project for a U.S. company. They brought me in as a consultant and mentor on their first iterative project using RUP. During our initial interview, they shared with me that they had spent six months of effort with nine business analysts and subject matter experts gathering the project's requirements. I delicately inquired about their commitment to the iterative approach, and their straight-faced response was, "Oh, we are committed to iterating on this project, just as soon as we get all the requirements!" My involvement with that project was very short-lived.

But it is legitimate to challenge my view. You might ask, "How can you say you don't need all the requirements? You need to know what to build, don't you?" And my answer is: yes, of course, you need to know what to build. You just do not need all the requirements before you start. My justification is very simple. To build the right system, you need to implement the features the customer really wants, and these features have to live and execute in the right architectural geography that supports those features. Can you agree with this? OK, now consider any project you have managed or developed. What percentage of the total requirements determined the design and architecture of the system you developed? Typically, 5% to 10% of the total requirements determined the design. The other 90% to 95% of requirements merely followed whatever design or architecture you chose. What we need at the beginning of a project is a good enough understanding of that 5% to 10% of major and architecturally significant requirements that determine the design. The remainder are given, discovered, explored, and implemented as the project continues through its iterations. It is the role of the BA on an iterative project to be a conduit for these emergent requirements throughout the project lifecycle.

At this point it is not unusual to hear an objection: "But what if you miss a major requirement in the beginning, and you realize in iteration 6 that your design must be changed?" OK. We change it. This is what you would do in a BRUF project. No matter how much time you spend doing BRUF, you will miss something, probably something that will affect the chosen design, architecture, or deployment of the system. The real issue here is that in an agile project, we embrace these discoveries and make the changes at planned points in the process (i.e., at the iteration boundaries). In a BRUF and waterfall project, there are no defined points to accommodate these changes in a controlled manner. Change will happen, and when it does you can be agile or fragile. The choice is not difficult.

Project Management

The most pervasive myth in project management today is this: if we put process controls in place, the project will succeed. The extrapolation of this delusion is obvious: the more controls in place, the more success you will have. So pile controls on top of controls.... Obviously, no one would really recommend this. But if our rejection is based only on the non-scalability of this notion, we are perhaps not seeing that this notion of the value of control is wrong in itself.

Certainly, we need to have some level of control. The issue is the *kind* of control, not *how much*. The kind of control we need is where we control the process, not let the process control us. A valuable process is controlled, not controlling. In the waterfall world, a common *gestalt* is: if we build a plan, the plan is true; if we stick to the plan, we will achieve our goal. This only works when nothing changes, and when nothing goes wrong.

In our world of internet time and software death marches, our biggest need today is not *prescription*, it is *adaptation*. How many PMI-certified project managers believe they are controlling their project with huge, plotter-sized Microsoft™Project wallcharts—wallcharts that detail four hours of work for Fred Developer nine months and three days from now? We cannot be sure what Fred will actually be doing tomorrow! Iterative project management requires...humility.

Someone once said, "The value is not in the plan, but in the planning." Agile processes are agile because they enable adaptation. Agile project management has a different dynamic. Here are just a few examples:

- **Attack risk:** In my experience, a major cause of project failures is in this statement: "They spent all their time doing the easy stuff." Attack risk early or it will attack you later. The project manager must work with the development team to identify the major business and technical risks. The iteration plans must address these risks head-on, with the courage to state early in a project, "We're having some challenges that may affect the entire approach we have chosen." Not pleasant, but much better than investing 80% of your project funds and discovering in the 11th hour that you have to go back into major re-design activity.
- **Expect rework (slips) early in the project rather than later:** Because an agile RUP team will address risk early in the project, obstacles will arise

early. But when these obstacles are overcome, there should be no slips attributed to technical risk near the end of the project.

- **Monitor iterations to prevent gold-plating by the developers:** Gold-plating is the adding of "cool" or enhanced features not in the requirements. The Project Manager must continually monitor the iteration plans and the iteration deliverables for "unnecessary creativity" from the development team.
- **You do not have to have all the requirements in order to start the project:** Get the right requirements at the right time. The Project Manager is like a sports coach and has to anticipate when to get the information that will be needed by the team as they progress in their understanding of the system being developed.
- **Manage "in your headlights:"** Do not add detail beyond what you actually know. The detail of your plan must match the proximity of the execution: iterations close in time will have greater detail in tasks, resources, and features than iterations further away in time.
- **Put the software first:** If a task does not directly contribute to delivering proper, working software, *do not do it.* This requires the project manager to map the value stream on the project, and identify and eliminate waste, where waste is any task that does not contribute to delivering the software. This is a sensitive political area, but it is an ethical issue as well. Waste is not good, and sometimes it is our own corporate policies that are causing waste.
- **Do not believe your own press releases:** If you become convinced you will succeed, do not start piling new "essential" features onto the project. Some call this "scope creep." The project manager must keep the project focused. If the customer truly wants new features added, agile RUP does not call this scope creep, it says: "Renegotiate the project." If your time, budget, and effort are sized to a given set of features, when this set changes, it is now a different project and everything is up for redefinition.

Conclusion

As we have seen in this chapter, agile RUP affects virtually every part of software development. But the investment is worthwhile. RUP provides a process framework with pre-defined value for conducting object-oriented projects, and even classic procedural projects in COBOL or C. But it is generic and prescriptive, and no organization can afford to do every task or produce every artifact defined in off-the-shelf RUP. Making RUP agile is a simple matter of vision and humility: keep your focus on delivering software that meets a business need, and acknowledge that today's software is just too complex to completely characterize in a lump-sum effort up-front.

Acknowledgment

Portions of this chapter come from articles previously published by the author in IBM/Rational Software Corporation's online magazine *The Rational Edge,* available online at *www.therationaledge.com.*

References

Ambler, S. (2002). *Agile modeling*. New York: John Wiley & Sons.

Astels, D. (2003). *Test-driven development: A practical guide*. Englewood Cliffs, NJ: Prentice-Hall.

Hunt, A., & Thomas, D. (2003). *Pragmatic unit testing*. The Pragmatic Bookshelf.

RUP. Rational Unified Process®, version 2003.06.00.

Chapter X

Planning and Managing the Human Factors for the Adoption and Diffusion of Object-Oriented Software Development Processes

Magdy K. Serour
University of Technology, Sydney, Australia

Abstract

Although there are a large number of contemporary software development processes/methodologies available to assist and guide software professionals in developing software systems, there is no specific process that can assist organizations in planning and managing their transition to this new work environment. As a result, there are still a large number of information technology (IT) organizations that have not yet implemented any object-oriented (OO) process. For them, the transition to a new work *environment and the adoption and utilization of a software process implies a number of problems, commonly including necessary human and organizational resistance to the ensuing cultural change. This chapter provides IT organizations and professionals with insights into the most important key success factors that may promote the entire process of organizational change. We investigate the effect of various human factors on the adoption and diffusion of an object-oriented software development process. Some of the human factors include motivation, leadership, resistance to culture change, and willingness and readiness to change. In addition, this chapter explores the significant role of these factors in controlling the entire process of implementing an OO process in practice, emphasizing the significance of planning and managing these "soft" factors to achieve clear advantages and gain enviable results.*

Introduction

This chapter investigates and examines the effect of various human behavioral patterns during the organizational transition to an object-oriented (OO) work environment, and the adoption and diffusion of an OO software development process. Technology is only a tool; what makes the difference is the individual who makes use of the technology, and the culture that motivates people to realize and understand the advantages of adopting such technology (Zakaria & Yusof, 2001).

During any paradigm shift, human tendencies play a critical role that may invariably result in either success or failure. Examples of such human aspects may include cultural change coupled with people's resistance, motivation, education and training, communications, and leadership. Collectively, these factors can form either opposing or supporting forces that may influence and impact on the entire transition process. Therefore, human aspects must be seriously considered, well addressed, planned, and managed for a rewarding result.

Past studies (e.g., Gibson, 1999; Ioannidis & Gopalakrishnan, 1999; Nambisan & Wang, 1999; Auer & Dobler, 2000; Jurison, 2000; Burshy, 2001) of the process of organizational transition have related the transition process to how organizations adopted innovation, ideas, new technologies (e.g., Web services and e-business), or new "ways of doing things" (e.g., the adoption and deployment of an OO process).

What these processes missed in the past was the first (and the most critical) step towards the adoption of a new technology. They all missed the study of moving organizations from their current state or environment to their desired one where they can feel comfortable, familiar, and confident to adopt and diffuse an innovation or new technologies such as OO processes. Getting organizations ready to adopt and diffuse a new technology involves a number of serious managerial decisions that must be made to provide full management support, dedication, and commitment. Organizations must feel comfortable and familiar with the new way of "doing things" before any attempt is made to implement these new ways in practice to avoid or lessen people's natural resistance to change, and also increase their acceptance and readiness.

Hence, the main objective of investigating the impact of human issues is to gain a full understanding of individual behavior during the transition and also to examine different human factors that influence the response of individuals within organizations toward the adoption of an OO software development process.

Organizational Change and Human Factors

"The greatest difficulty in the world is not for people to accept new ideas, but to make them forget about old ideas." John Maynard Keynes

People Behavior During Organizational Change

During an organizational transition, different people play different roles, such as motivators, adopters, resistors, opposers, and neutral or observers (Bridges, 1995). How they respond to change during transition can, and in most cases does, dominate and determine the success or failure of the entire project. The inextricable reality is that people are different, and so act and react to changes differently; from time-to-time even the same person can behave in a different manner.

Bridges (1995) claims that changes are always accompanied by natural resistance, as changes often drive people out of their comfort zone. Consequently, people can develop a resistance to change and become the main obstacle to the whole organizational change.

Once an organization comes to realize what it needs to achieve and decides how it will accomplish its goals, the main challenge becomes the issue of effective and efficient management of human factors. It is quite surprising to know that 80% of project failures are traced back to mismanagement of human factors (Jacobson, Ericsson, & Jacobson, 1995).

Unfortunately, there are many organizations still struggling to deal with difficulties related to the effective management of human or sociological factors during technology adoption and diffusion. This type of problem is usually caused by management's lack of commitment to the human factors of IT. Szewczak and Khosrow-Pour (1996) relate this problem of mismanagement of human aspects to the fact that, in general, organizations traditionally invest a significant proportion of their resources to obtain the necessary hardware and software technologies, but with insignificant investment in the human aspect of technology. An experienced software development team that has been around for more than a decade, for example, is likely to have superior expertise in (and consequently be comfortable with) traditional software modeling techniques such as Flow Charts, Data Flow, and Entity Relationship Diagrams. Professionals of this type would not be openly receptive to changing their existing work culture and switching to modern OO techniques such as Object Model, Use Case, and interactions Diagrams. This kind of human culture change can form a major challenge during the transition that may increase people's resistance to change.

The Challenges of Human Factors

In general, human aspects are the most difficult challenge to be addressed during any organizational change (Zakaria & Yusof, 2001). The transition process to OO and the adoption of an OO process usually involves a large number of technical as well as non-technical issues. Various authors have referred to these non-technical issues as *soft factors* (Constantine, 1995) and *sociological factors* (DeMarco & Lister, 1987). We call these 'human' Key Success Factors in this discussion, since they deal with the 'human aspect' of change. These human factors necessarily require complete understanding and managing alongside the technological factors. Since OO contains elements relevant to all stages of the development lifecycle (not only coding), and includes models for requirements engineering, project management, team building, and so on, adopting OO process requires a combination of learning about technical issues as well as larger scale sociological issues.

People, procedures, and tools are the three critical aspects that must be well planned and managed during any organizational change (Serour, Henderson-Sellers, Hughes, Winder, & Chow, 2002). Certainly out of these three aspects, people are the most demanding aspect to be changed and managed. Eason (1983) argues that the human factors of technology are far more important than technical factors. The most challenging aspects of the transitioning to object-orientation remain in establishing a new software development environment and in introducing a new work culture to managers, software developers, and customers (Ushakov, 2000).

As a result, it is imprudent for management to expect every team member to agree with all the proposed changes. It may even be unwise to anticipate full commitment and belief in the new organization mission, especially at the early stage of transitioning. Organizations must face reality by realizing that change is often difficult in the best of circumstances and seldom goes exactly to plan. Management must be mindful that only a minority of individuals will wholeheartedly welcome any proposed changes to the way they have done things for so long.

The challenge to software professionals is to change and adapt to a new environment. A possible scenario may ensue where developers need to adopt new ways of 'thinking' about software, followed by OO modeling/designing, developing, quality assuring, and testing the software. Adopting a new approach of developing software may include technical factors (e.g., CASE tools, Programming languages, and databases) that require a reasonable degree of human culture change in order to utilize them to their maximum potential. Individuals, especially those working on software projects, work under the influence of various personal, motivational, and social factors. These factors, more often than not, remain in the background. Due to their elusive and intangible nature, these human factors are often difficult to discern, analyze, and improve on when change is about to be implemented.

As a result, moving working professionals to new ways of perceiving and undertaking various tasks is very difficult (Fingar, 1996). However, many studies such as Szewczak and Khosrow-Pour (1996) suggest many organizations are still struggling to deal with problems related to the human aspects of technology. Furthermore, they correlated this problem to management's lack of commitment to the human side of IT.

It is vital, then, that organizations pay careful attention in addressing all human or soft factors, and be open and ready to confront and solve any conflicts that may influence the entire transition. Organizations need to appreciate individual behavior when examining the transition and the adoption of new OO process; they need to investigate and explore the different factors that influence the response of individuals within the organization towards the new technology. To accomplish a successful transition, an organization needs to advance people's motivation and enthusiasm, maintain management commitment, and provide efficient and persuasive mentors whose leadership may be seen as the source of expert guidance (Jacobson et al., 1995).

The art of managing cultural interfaces has become an everyday business challenge at every organizational level (O'Hara-Devereaux & Johansen, 1994). The main question here, and management's major challenge, is how an organization makes the transition a driving force for all people involved in being adopters and supporters of the change, instead of opposers or neutral players. Constantine (1996), during the OOPSLA'96 panel discussion pertaining to human factors, contended that it is quite easy to communicate with machines and solve problems, but with people it is difficult. The main reason is because people are very difficult to "generalize."

Human Factors and Risk Issues

Organizations must be aware of the consequences and the possible risks involved in mismanaging the human factors during the introduction of a new work culture such as OO process. The improper planning and managing of human factors can easily lead to undesirable consequences, such as building resistance to change, and adding more confusion, uncertainty, and fear that can considerably diminish the chance of success. Schein (1999) interestingly states that there was no such failure for technology adoption; instead it was a failure to understand the organizational and the individuals' culture.

As a result, organizations that are aiming to adopt an OO process need to address not only the technological factors of the adoption, but also the human factors. As an example, organizations must provide adequate education and training to their people in order to understand and grasp the fundamentals and underpinning concepts of object-orientation and development processes, as it is a critical factor to avoid those risky consequences. Bridges (1995) argues that the most prevailing cause for the unsuccessful implantation of organizational changes can be attributed to a lack of planning in managing the impact of change on individuals. He also emphasizes that many organizations that are not properly geared to handle this facet of change can run the major risk of jeopardizing their own existence.

Therefore, management may be very well advised to create a plan to manage and mitigate these potential risk factors by answering the following questions:

- How are people likely to respond to the change?
- What is required to convince people that the change is worth the effort?
- What actions are necessary to earn people's support and commitment to the change?

In addition, management needs to identify the ultimate and most effective means in achieving the following objectives:

- Selling the change to all people involved, including customers.
- Motivating people to make the transition.
- Reducing people's resistance to change.
- Eliminating people's fear and uncertainty.
- Minimizing the change's disruption to people.
- Mitigating the increasing tensions between people during the change.
- Encouraging people to be enthusiastic for the change, as opposed to being apathetic and an obstacle.

By answering the above questions and achieving the related objectives, organizations will be able to use persuasive approaches to human change, ensuring that everyone is comfortable and willing to accept, and make use of, the new OO process with all its associated changes.

The Human Factors

Human Culture and Necessary Culture Change

"In a time of rapid change, standing still is the most dangerous course of action." Brian Tracy

The Oxford English Dictionary (9th edition) broadly defines human culture as the arts and other manifestations of human intellectual achievement regarded collectively as the improvement by mental or physical training. In particular, Palvia, Palvia, and Roche (1996) define the culture of IT professionals as the set of values and practices shared by these members of an organization involved in information technology activities, including managers, developers, and customers/end users.

Personal culture is usually characterized and distinguished by the individual's values, such as behavior, attitude, experience, and beliefs. There are people who work well under pressure, whereas others work well only when properly supervised and directed, and then there are the 'cowboys' (Constantine, 1993) who prefer to work on their own. Cooper (1994) asserts that an IT person's culture may resist the introduction of a new process which realigns status, power, and working habits, especially when they violate some of the group's shared values. Current personal culture can be incompatible with certain new processes to the degree that risky consequences may be incurred, including resistance to change, a negative attitude and behavior, implementation failure, or the achievement of totally unsatisfactory results. Human culture change—that is, the physiological change that people undergo to alter the way they carry out their work on a daily basis—is one of the hardest and most longstanding parts of the adoption of an OO software development process (Fayad & Laitinen, 1998).

Natural Resistance to Change

Coupled with the introduction of a new work culture, people tend to naturally build resistance to any challenge of changing their culture and/or learning new things. The unfamiliarity with the new changes can lead to a discomfort that naturally increases people's resistance. In general, change is often seen as a personal threat by those involved in transition (Huse, 1975).

Resistance to change can also come from management, project leaders, and customers/end users for similar reasons, including fear of change and uncertainty of their capability of carrying out these changes (Fayad & Laitinen, 1998). Furthermore, adopting an OO process also requires people to advance their skills and knowledge, and/or gain new ones, as well as learning new tools and techniques. All these changes can lead to a threat that, if people are not capable of changing their culture, they will be out of the workforce, and be replaced by others who possess the required OO knowledge and skills and are capable of utilizing the new process. Resistance may also happen during the course of adoption when people are faced with serious impediments. For example, people may reach a stage when they feel that they cannot use their old ways (ad hoc) and at the same time they are not comfortable with the new ways (OO process). They then try to escape or oppose the changes. This leads to an increased level of resistance.

During organizational change, managing people's resistance becomes a critical issue that must be seriously considered so as to accomplish satisfactory results. For that reason, organizations must be able to effectively manage people's resistance to leading and directing the change process. To do so, management must first understand what resistance really means. What do people really resist? Do they resist the new environment, new technology, or the changes they have to undertake? And finally, why do people really resist? Do they resist for psychological reasons, technological issues, personal concerns, or a combination of all?

What Resistance Really Means?

Naturally, people want to improve and find better ways of doing things. A part of making improvements is causing changes, and changes are always faced with different types of resistance. People's resistance can be a result of different human reactions that sometimes form obstacles, impediments, hindrances, and difficulties to change. One of the risky issues regarding people's resistance is that, sometimes, managers see resistance as a sign of laziness, stupidity, or just unwillingness and opposition to change (Fayad & Laitinen, 1998).

In actual fact, resistance can be a sign of people's disinterest or they could be busy with more pressing issues. Resistance could also be a signal of conflict of interest or contradictory point of views. Moreover, resistance could be a silent request for assistance, more information, or an assertion of different priorities. Resistance can be viewed as an opportunity to gather information and learn better about current and desired state (Bamberger, 2002).

What People Really Resist?

Naturally, people do not like to lose. They do not like their own things to be taken away from them, and that is exactly what people resist. People associate change with the loss of their existing comforts. People do not resist the change itself, so much as they resist the uncertainties, fear, and discomforts associated with it. People, especially those who are confident and comfortable with their existing culture, resist the idea of changing their ways of doing things, and facing the risk of becoming unfamiliar and uncomfortable with the new ways. In addition, every change involves a degree of risk, and people are naturally reluctant to take risks and face the unknown.

Why People Really Resist?

To manage the resistance to change, it is important first to understand the various reasons behind it. Resistance could happen at an early stage of the introduction of the new change and/or at a later stage, during the change process. People resist change when they are unaware of the need to change and, accordingly, are uncertain of the final result. On the other hand, Bamberger (2002) notes that change often meets with resistance because it is seen as a personal threat that leads to fear of failure and rejection. Fayad and Laitinen (1998) relate resistance to the lack of a clear view of the current state and objective goals. They further note that resistance to change often exists as a result of structural conflicts within the organization. Lack of management's commitment and inconsistent actions with the new change can also elicit resistance.

Bridges (1995) declares that when changes take place, people get angry, sad, frightened, depressed, and confused. These emotional states can be mistaken for bad morale, but they rarely are. Rather they are more likely to be a sign of grieving, the natural sequence of emotions people go through when they lose something that matters to them. People resist the loss of competence that they once had and which was associated with their old familiar tasks. Transition is tiring; during the change, people build resistance when they feel unfamiliar and uncomfortable with the new ways. Resistance happens when people feel that they cannot use their existing old ways, and at the same time they are not comfortable and familiar with the new ways. From a different perspective, people build resistance when they feel that the new ways they have to follow can negatively affect their productivity. For example, an inappropriate new process or technique can discourage people to change, as it can lead to a drop in people's productivity.

Accordingly, with the introduction of a new OO process, people's resistance to transition and culture change could be a result of one or more of the following human behavior factors:

- They see the transition with its necessary changes as a threat to their jobs.
- They do not have inadequate knowledge and experience related to the new OO process.
- They are afraid of learning new ways of carrying out their jobs.
- They doubt the benefits of adopting a new OO process.
- They are afraid of failure or that they will not be able to understand the new process.
- It is exhausting to learn new things, and some think they are too old to learn a new process.
- Some prefer to do what they know best, even when they acknowledge there may possibly be a better way *(The devil you know!)*.
- They are overloaded with their current projects with no time to learn new things.
- They are not in favor of the new process with its associated modeling language and/or CASE tools.

Even during the transition process, and when changes take place, people can develop more resistance and be less motivated for different reasons including:

- Anxiety rises and motivation falls.
- They are afraid of failing.
- They are unsure of the new way.

- They are afraid that they may be blamed if something goes wrong.
- They try to avoid learning new things.
- They become self-protective.
- They respond slowly and want to go back to the "old way."
- They doubt the benefits of the new way.
- They feel panic and confused.

Unsurprisingly, even customers may develop some resistance and be reluctant to accept the changes as a result of the risk factors involved in introducing a new process that would affect their products. Therefore, customers must be involved in the change process owing to their effective role in supporting the organization's adoption of a new OO process. The customer role needs to be changed from being just a customer requesting and running a software application to that of being an effective and supportive partner in the whole process of producing their products.

Defeating People's Resistance to Change

As discussed above, many managers see resistance as a sign of laziness, stupidity, or just plain perversity on the part of employees (Fayad & Laitinen, 1998). Lawrence (1969) suggests that resistance to change should not be treated as a problem, but rather as an expected symptom and an opportunity, or a request, for learning and better knowing the unknown. Also, Senge (1990) stated that resistance to change generally has a real basis that must be understood in order for it to be dealt with. In order to manage people's resistance to change, organizations need to understand better the very human reactions they face when they are asked to do something differently or change their work habits. Management must plan and practice some strategies to manage and deal with people's resistance.

To do so, management must begin to understand what "resistance" really is? What are the major reasons for people to build resistance to change? Then they must establish a suitable work environment for people to undergo the required changes. Management cannot change people, they have to change themselves and they only change when they have the appropriate and supportive environment (Boyett & Boyett, 2000). Bridges (1995) reported that most managers and leaders put only 10% of their energy into selling the problem, but 90% into selling the solution to the problem. Management must put more energy into "selling" the problem that is the reason for the change, because people are not interested in looking for a solution to a problem they do not know they have. Management must realize that culture change is very tiring. People will feel tired, overwhelmed, down, and depressed. Management must look at those symptoms as a natural human reaction, and work hard to rebuild its people self-confidence and give them the feeling of competence in mastering the new way. Management must realize their people's capabilities and give them what they can do best without any fear of failure. Anxiety is natural, and the best way to defeat it is to educate and train people on their new environment. Adequate education and training regarding the newly adopted process can easily eliminate the fear of the unknown, the uncertainties about the final result, and thus lead to elimination of people's resistance.

Defeating Resistance with Participation

One of the most effective ingredients to defeat people's resistance to change is by encouraging them at an early stage to participate in planning for the change; Huse (1975) confirms this fact by assuring that people—during transition—see change as a threat unless they have participated in its planning. Lawrence (1969) has demonstrated through one of his case studies, where an identical change was introduced to several factory groups, that the first group, who was not offered any explanation, resisted all management's efforts, whereas the second group, who was involved in the change planning, carried out its transition with minimal resistance, and an initial small productivity drop was rapidly recovered.

Involvement of people in planning their change allows them to understand why they need to go through it and what they should expect. Humphrey (1995) affirms that people's resistance can be gradually changed to acceptance, once people are convinced of the necessity of the changes, and also they can see the value of undergoing the change. So, management must show and convince people that the change they need to go through is not a threat, but rather an opportunity for improvement.

Defeating Resistance with Small Wins

Humphrey (1995) asserts that people's resistance to change is proportional to its magnitude. Therefore, resistance can be reduced by planning a number of small changes instead of a hefty change. Transitioning an IT organization to OO environment and adopting an OO process involves outsized changes. Psychological, organizational, and technological changes that can be planned in an incremental manner are "*small wins.*" By introducing the new changes in small increments, each increment will be easy to sell and implement. People will feel confident and positive every time they successfully achieve one increment, and become even more enthusiastic and motivated to implement the next increment. This technique can lead to a smooth transition by enhancing people's willingness to participate and reducing their total resistance. For example, the process of adopting a new OO method could well be started by addressing the Requirements Engineering (RE) activity as a major focus to engender everyone's involvement. An initial RE approach, using a well-defined technique such as the use case technique, can be introduced and utilized without the burden of all other activities and techniques. Once people feel confident in carrying out the RE activity, another major activity such as user interface design can be introduced in the same manner and so on for all other activities.

Education and Training (Knowledge and Skills)

Younessi and Marut (2002) define education as an opportunity to learn and ideally master a number of theories—principles that enable the recipient to assess, analyze, and act appropriately to a broad range of relevant situations. The impact of education is usually more abstract and wide in scope. They also defined training as the provision of an opportunity to learn and ideally practice some skills in a controlled environment to carry out a particular task(s). The impact of training is usually more focused, direct, and narrow in scope. In other words, education provides people with the answer to "*know what,*" whereas training provides them with the answer to "*know how.*" People can gain knowledge and skills either through education and training courses provided by their organization and/or through real-life work experience.

In general, people need education and training in their discipline to enable and empower them to assess and perform their duties in a professional and satisfactory manner. In the context of the adoption of a new OO process, technical people need to learn about the new process and associated technology to feel comfortable when using it (Zakaria & Yusof, 2001). The individual's knowledge and experience related to the new process play an effective role in making the decision for the adoption. People with adequate and appropriate knowledge and experience of their new OO process can be more self-motivated and enthusiastic to make a transition than others. On the other hand, during an organizational change, lack of knowledge and/or experience could elicit people's resistance to change and increase their feeling of incompetency and frustration.

Necessity of Education and Training for Transition

Younessi and Marut (2002) have emphasized the vital role of education and training during the adoption of a new OO process by suggesting that education and training is an obligatory component for the successful introduction of software processes into an organization. Furthermore, they considered education and training as a Critical Success Factor for adopting these processes. Perkins and Rao (1990) stated that the more knowledge and experience people have, the more they are able to contribute to decisions regarding the adoption of OO processes. They also emphasized the impact of training related to the new technology that increases their ability to adopt, diffuse, and master processes, techniques, and tools. Conner (1992) demonstrates the imperative role of education and training in adoption by saying that people change only when they have the capacity to do so.

People's experience towards OO processes should have a positive impact on the adoption (process) of such technology. In the context of adopting a new OO process, Sultan and Chan (2000) stated that the greater the knowledge and skills of individuals, the more likely they are to adopt it.

Management must recognize the critical nature of proper education and training, since experience has shown that between 25 and 40% of the total cost of an extensive project will be spent on education and training (Mize, 1987). More experience enables people to contribute more towards the transition and the adoption of new OO processes. People's appropriate knowledge, experience, and education may have a positive impact on their transition. The greater the work experience of individuals with the organization, the more likely they are to transition and adopt the new process. Highly skilled staff can manage themselves more easily, particularly during a paradigm shift. Therefore, an individual's satisfactory level of knowledge and education towards the new technology and processes forms another imperative management challenge.

Motivation

"Motivation is the release of power within a person to accomplish some desired results. It is a major key for achievement and success." Dr. Len Restall

Motivation has a remarkable power and is a major influence on people's behavior towards any achievement (Humphrey, 1997). In order to investigate the importance of motivation during the organizational adoption of OO processes, we need to understand what motivation is, its role during the adoption, and what motivates people.

What is Motivation?

Bolton (2002) defines motivation as a sociological concept used to describe individual factors that produce and maintain certain sorts of human behavior towards a goal. Hence, motivation, as a goal-directed behavior, is a driving force that stimulates people to achieve a set of planned goals.

Motivation is often driven from a desire or an inspiration to accomplish a defined objective, combined with the ability to work towards that objective. In other words, Motivation is the ability of taking good ideas or serious changes, and coupling them with appropriate knowledge and skills to achieve desired objectives.

Consequently, people who are aiming to achieve a specific goal must be both motivated and have the power and ability to work towards that goal. People who are motivated towards an accomplishment must be also capable and empowered to do so (carry out the work).

In the context of this chapter, motivating people during the adoption of a new OO software process could mean those factors which cause individuals within the organization to do more than they otherwise would. For example, to transit a development team to a totally new OO environment and/or adopting a new process, people need to work more than usual to change their existing work culture and adopt a new way. Then, motivation becomes a measure of an individual's level of readiness and willingness to participate effectively towards a successful transition.

Role of Motivation During Adoption

Pfeffer (1982) states that when people understand and accept a goal, and believe they can meet it, they will generally work very hard to do so. Additionally and from a different perspective, Maslow (1970), when he described his "*Hierarchy of Needs,*" stated that people are capable of achieving their goals if they believe that they can achieve them.

Motivation provides people with the understanding of the reasons for the change that increases their acceptability, which in turn improves their ability to accomplish a successful mission. Moreover, motivated people more often than not are capable of defining and achieving their own goals.

During an organizational change, people need to have compelling and persuasive reason(s) why they have to go through changes. Once they are convinced and believe in their mission, they will feel more competent to carry out their necessary changes successfully. This becomes a positive motivation that makes people desire to accomplish their goals that result in their very best performance (Humphrey, 1997).

On the other hand, lack of motivation during adoption can lead to negative consequences that may contribute to undesirable results. Those consequences may include fear, confusion, frustration, and uncertainty. Humphrey (1997) confirms the vital role of motivation by saying, "Without motivated and capable employees, no technical organization can prosper."

As a result, management must improve and maintain people's motivation to help them to be more effective and efficient in moving to their desired work environment with the adoption and utilization of a formal OO process. Motivation addresses the degree to which people want to, and are willing to, complete the work necessary to change their existing work culture. Furthermore, during transitioning to a new work environment, management must maintain people's motivation if they try to give up easily, or too soon, so as to encourage them to keep trying as long as they believe in their goals.

What Motivates People?

Bolton (2002) suggests that a good first step towards understanding what motivates people is to know what people want from their jobs. The answer could be gaining financial advantages, acquiring more skills and knowledge, or working with the latest technologies. In reality, it is very difficult to predict and judge on people's desire because it depends on the individual's values and beliefs.

What motivates people to change their work culture and adopt a new process differs, all depending on their needs and perception of the new process. Hence, not all people can be motivated by the same things and to the same degree. Therefore, management—especially during transition—needs to understand why their people do their work, and consequently elicit what motivates them and how to maintain that motivation through the entire process. Understanding people's aspiration can help management not only motivate people, but also support their motivation by maintaining and increasing the reasons for motivation.

Motivation is often seen as the driving force that moves people to perform some actions. Then, in the context of the adoption and diffusion of an OO process, people need a comfortable, familiar, valuable, and convincing driving force or "*motivation factor*" to move them to perform more effectively than they usually do.

People feel comfortable when they confidently know how to do their jobs. Enhancing people's skills and knowledge to the required level for transition makes them feel comfortable with the new process. Formal and professional education and training pertaining to OO processes are considered to be effective and efficient techniques to achieve people's self-confidence. In order to motivate people, training must include clarification of language and jargon or commands used to avoid further frustration and anxiety, as new technologies usually have a mystifying and alienating potential. The more comfortable people feel with the new technology (here OO process), the more willing they are to experiment with it (Zakaria & Yusof, 2001).

Motivation factors must also add extra values for people such as financial reward, or learning a new language, process, or tool. For example, someone likely to retire in a year is unlikely to learn a new way of doing things. On the other hand, an enthusiastic newcomer to the project is likely to put in the extra effort needed to pick up new methods of thinking and modeling.

People who are in the move (e.g., changing profession, changing company, or reaching retirement) will not be interested in changing their culture, and they become difficult to motivate to change their current culture because they will not gain any benefits. To motivate people to change their culture, they need assurance of benefiting from the change.

The driving force must also be convincing, it must be relevant to what people usually do, and it must be in a way that people can believe and make use of. Most people must be convinced of the need to change before they will willingly comply (Humphrey, 1995).

Leadership

"Leadership is the ability to provide those functions required for successful group action." Weldon Moffitt

Leadership is well thought-out by many researchers and authors as a fundamental activity of project management, and they simply describe it as motivation plus organization (Stogdill, 1974; Thite, 2001; Phillips, 2002; Castle, Luong, & Harris, 2002).

In general, leadership plays a significant role in how people effectively perform their jobs. Rogers (1995) defines leadership as the degree to which an individual is able to guide, direct, and influence other individuals' attitudes informally in a desired way. Stogdill (1974) describes leadership as a process of influencing group activities toward goal setting and goal achievement.

Leadership is a practice that needs special skills and talent to motivate people to get things done in a favorable and constructive way. Bridges (1995) affirms that leading a team of professionals efficiently is an invaluable resource to any leader who is eager and willing to understand how to inspire team members. Also, it is a vital ability to augment teams' cooperation and individuals' commitment and productive participation. Above all, an effective leadership is a Critical Success Factor on how team members become adaptable to changing circumstances (Thite, 2000).

Leadership is basically situational. Fiedler (1967) suggests that there is no one style of leadership that suits every project situation. Rather, it should be contingent upon the nature of the organization, the team, and the project situation. Leadership style should be flexible and agile enough to be adapted to suit the project at hand in the most appropriate manner. Phillips (2002) claims that the "appropriate" leadership style is the style with the highest probability of success. Additionally, a good leader should effectively mix and cooperate with team members.

Leadership Style for Adoption

"A leader is best when people barely know that he/she exists."
Whitter Bynner

During an organizational change situation, leadership style must be adapted to promote and support the organizational change. Social factors such as the style of leadership can influence the individual's ability to transit to a new work climate. Thite (2000) emphasizes the importance of the appropriate leadership style during a change situation by considering it as a Critical Success Factor.

Changing people's work culture and introducing them to a new way of doing their job is usually accompanied by increasing anxiety, ambiguity, and insecurity. During such time, a supportive leadership style can effectively contend with and overcome these problems.

Due to the fact that an effective leadership style should depend on the follower and the project situation, leaders must use the most appropriate style of leadership that best suits the project situation that yields the best chance of success (Phillips, 2002). Furthermore, Phillips states that managers must be able not only to determine the most appropriate leadership style, but also to apply that style correctly.

Stodgill (1974) supports that argument by saying, "The most effective leaders exhibit a degree of versatility and flexibility that enables them to adapt their behavior to the changing and contradictory demands made on them." Moreover, Hersey, Blanchard, and Johnson (1996) proclaim that successful and effective leaders are able to adapt their leadership style to best fit the requirements of the situation. Therefore, leaders within an organization about to introduce a major change to their work environment—such as the adoption of a new OO process—must have adequate resources (mainly time), a strong interest in the mission, and a greater exposure to the new process they are about to adopt. Sultan and Chan (2000) firmly assert that the greater the opinion leadership among group members, the greater the chance of a successful transition and adoption.

Leadership always means responsibility. Consequently, an effective leadership style for a transitioning organization must be driven from top management with full support and commitment, and it must be also coupled with a rational degree of authority. Co, Patuwo, and Hu (1998) support that by saying: "Ideally, the team leadership should come from top management: one who is a 'doer', and who commands respect throughout the entire company."

During the organizational transition to a new work environment, leadership should foster strong alignment with the organization's vision. Team members that depend primarily on strong alignment with their organization's vision are at their best with well-understood practices applied to well-understood problems. Furthermore, successful leaders must be able to create a new work environment in which team members feel comfortable and familiar, and also enjoy what they are doing.

Lockwood (1991) and Constantine (1994) affirm that the working culture shapes the style in which software development is carried out. Different styles of organization during different situations require somewhat different forms of leadership style and management, and each will tend to have somewhat different software practices.

Leaders who are leading their teams through a serious change—such as adopting a new OO process—must be able to:

- Define a clear vision and mission.
- Provide convincing and compelling reasons for the change.
- Focus on and be concerned with the *why to* aspect of the transition, rather than the *how to.*
- Manage and motivate team members.
- Maintain an authority level to make appropriate decisions in the face of uncertainty.
- Provide full support and commitment to their followers.
- Provide adequate and appropriate resources including time, education, and training.
- Establish clear communication channels with team members.
- Monitor and review tasks allocated to team members.
- Recognize and appreciate achievements, and reward people for doing good work.

Perception of OO Processes

While IT organizations perceive new OO processes as an essential means to advance their existing software development culture and effectively compete in the marketplace, individuals could have a different perception. Since people are different, their needs and expectations are also different. The more positively people perceive their new process with regard to its characteristics—advantages, observability, compatibility, complexity, and trialability—the more likely it is that the process will be adopted and utilized.

Management of Expectations

From past projects, it has been proven that people with life experience are able to deal with conflicts of interest that may arise due to their ability to make any decision regarding the transition (Hill, Smith, & Mann, 1987). To emphasize the vital role of education on conflict resolution, Barclay (1991) stated that any organization's members who have adequate education and significant real-life experience are well prepared to deal with interdepartmental conflict. It has to be made very clear from the beginning, to both managers and developers, that following a new OO process is not a magic wand or a silver bullet to make the entire organization's dreams come true, but rather is simply today's best approach option for software development. Nonetheless, it is also important that managers appreciate not only the benefits of adopting a new process, but also become aware of all the pitfalls and consequences of changing people's existing work culture.

Different stakeholders such as managers, developers, and customers may look at, assess, and evaluate OO processes, as a new technology to be adopted and utilized, from a different perspective. Senior managers are always looking for the dollar value return for their spending and investments. They evaluate the technology based on how much it will reduce the cost of software production with improved quality. Project managers and team leaders may assess the new OO processes from a different perspective. They emphasize project management aspects, gaining more control and meeting customer's expectations with on-time delivery based on building the software system from the pre-tested and proven software components. Software developers may value the proposed process for adoption as the new fad and trend to developing software. They view the transition process as a good opportunity to acquire new skills, and learn new tools and techniques. Unquestionably, this will insert desirable additions in their resume, add value to their worth in the market, and make them in more demand.

From the above discussion, unrealistic expectation from adopting a new OO process and also any conflict of people's interest can have a negative impact on the entire transition process. People will be reluctant to work effectively together as a team if they do not share the same interest and/or have different expectations.

In order to overcome the barrier of the conflict of interest, management and champion teams must include a specific plan to resolve that conflict between people, and reach mutual perceptions, consensus, and understanding.

Proper education, mentoring, and open discussions are examples of good techniques that can be used to achieve good understanding and realistic expectations of the new process. Stakeholders must come to some sort of consensus as to what they can expect from adopting a new process and also should be fully aware of the new technology pitfalls as much as its benefits. Stakeholders' consensus and shared understanding, as well as realistic expectations of the new process, can lead to a good working environment that can positively impact on the process of transition and thus the quality of the software produced.

Sharing the Vision with the Organization

People perform better if they feel that they share the value of the change with their firm. Managers need to provide the right organizational climate to ensure that their employees can see that, by working towards the organizational goals, they are also achieving some of their own goals. These goals could be financial rewards, or personal rewards such as the respect of their colleagues, or job satisfaction, or a combination of any number of things that the employee considers to be important.

During organizational change, such as the adoption and diffusion of an OO process, to gain people's acceptance, management should get people involved during the initial stage, listen to what they have to say, respect their view, and involve them in discussions and debates regarding the change. Follow up with them on any further progress. Give them the feeling that they are the owners and supporters of the change. Give them the impression that we are only interested in the change if it will help them and make their job more efficient and more enjoyable. Never blame the people for any faults or mistakes, but rather blame the product, the techniques, and/or the tools being used. Organizations need to try everything they can during the adoption of a new process to gain their people's willingness and support such as:

- Choose a flexible software development process/methodology to be tailored to best suit the organization's environment.
- Obtain senior and middle management support and commitments.
- Set an appropriate change management plan.
- Establish a transition support team to sell the changes to everyone and to deal with all transition issues.
- Plan for adequate resources.
- Plan for rewarding people behind the transition.
- Carry out the changes gradually and avoid overloading people.
- Plan for the appropriate type of education and training to enhance people's skills that minimize their fear and confusion.
- Introduce new ways in formal and informal ways as required.
- Never blame people for any failure; instead blame the process (*no* criticism).
- Listen to people and encourage their individual contribution.
- Reinforce the new way of following the new adopted process.
- Start with small jobs with high chance of success so they to gain self-confidence.
- Celebrate success with people.

Organizations should set a well-defined set of guiding beliefs that the mission of the transition and adopting a new OO process is worthy and must be shared with people. Individuals must be in harmony and agreement with an organization's value and goals. When individuals share the same values and beliefs as the organization, they form a psychological union as a group. They become more congruent to the organization. Thus, the efforts of adopting a new process are determined and rigorous. The sharing of ideas and commitments through cooperation and harmony within an organization often leads to an effective change and positive participation. Needless to say, whenever individual goals are in alignment, or close to alignment, with the organizational goals, then any initiative undertaken by the organization to adopt and diffuse a new technology—in our case, OO process—will be easily accepted and utilized by individuals. When people share the vision and beliefs with their organization, they feel that they share the ownerships of their new process and the accountability of the whole transition procedure. This feeling can motivate the individuals and assist management to defeat resistance to change should it arise. A strong organizational culture is usually measured by the way in which the key values are intensely held and widely shared. It has a greater influence upon employees than weak culture.

Communication Channels

When Zakaria and Yusof (2001) analyzed the implementation of technological change from a user-centered approach, they asserted that communication is a critical element in ensuring the success of a new technological change such as the adoption of OO processes.

Communications are defined as a process by which individuals exchange information through a common system of behavior. The process of communication between individuals has an influence on organizational transition, because it has a strong impact in conjunction with the leadership. A strong and effective communication network has a strong positive impact on adoption of a new technology. Proper and open communication channels can, from one side, help management to express their commitment and support to their people. From the other side, these channels can give people opportunities to discuss and exchange their concerns regarding their transition in order to adopt the new process. This can strongly increase people's cooperation, which enhances their ability to carry out the required changes effectively. From a project management point of view, open communications keep management in continuous contact with their people, and they can closely observe the transition progress to quickly resolve any problems that may occur. Communication can take different forms such as meetings. During an organizational transition to a new work environment and adopting a new process, meetings—driven by management—are seen as ways of undoubting management's commitment and support to change.

Conclusion

It can be clearly noticed that by considering and managing the human factors in great detail, the chances of successful adoption and diffusion of a software development process are enhanced significantly. Based on our research projects and several empirical studies, we conclude the following:

- Human factors play a vital role during the organizational transition to adopt and diffuse an OO process, as they can form either a promoting or resisting force.
- Understanding of the new desired OO process work environment can assist management in putting together a plan to manage the necessary cultural change.
- During an organizational change, managing people's resistance becomes a critical issue that must be seriously considered so as to accomplish satisfactory results.
- Organizations must be able to effectively manage people's resistance to leading and directing the change process.
- Involving individuals in planning and making decisions can positively eliminate their resistance to change and enhance their ability to carry out a successful transition.
- The individual's knowledge and experience related to the new process play an effective role in making the decision for the transition.
- More knowledge and experience enable people to contribute more towards the transition and the adoption of new technologies, including OO processes.
- The greater the work experience of individuals with the organization, the more likely they are to transition and adopt the new process.
- Gaining adequate knowledge and proper training on OO processes and associated tools can significantly contribute to enhancing people's ability to positively involve themselves in the transition process.
- The more knowledge and experience the organization's individuals have regarding OO processes, the more likely they are to change and adopt the new work culture and thus enhance the chances of a successful transition to a totally new OO software development environment.

- When people understand and accept a goal and believe they can meet it, they will generally work very hard to do so.
- Motivation provides people with the understanding of the reasons for the change that increases their acceptability, which in turn improves their ability to accomplish a successful transition.
- Maintaining and enhancing people's motivation during transition can be a driving force for people to wholeheartedly and positively participate.
- Successful and effective leaders are able to adapt their leadership style to best fit the requirements of the situation.
- The greater the opinion leadership among group members, the greater the chance of a successful transition and adoption.
- The more positively people perceive the new process with regard to its characteristics—advantages, observability, compatibility, complexity, and trialability—the more likely that the process will be adopted and effectively used.
- Management of expectations of the new process and conflict resolution can lead to a good working environment that can positively impact the change process.
- When individuals share the same values and beliefs with the organization, they are more likely to effectively participate towards the change.
- When people share the vision and beliefs with their organization, they feel that they share the ownerships and the accountability of the whole change practice and thus their new OO process.
- The sharing of ideas and commitments through cooperation and harmony within an organization often leads to an effective change and positive participation.
- The more individuals find values in the organizational change, the more likely they are to contribute to the transition success.
- The more congruent and rewarding the individuals perceive the organization's values, the more positive is the influence on the transition and the adoption.

References

Auer, D. & Dobler, H. (2000). A model for migration to object-oriented software development with special emphasis on improvement of acceptance. *Proceedings of TOOLS Sydney 2000 Conference* (pp. 132-143). Los Alamitos, CA: IEEE Computer Society Press.

Bamberger, J. (2002). Managing resistance—techniques for managing change and improvement. *Asia Pacific Software Engineering Process Group (SEPG) Conference Handbook and CD-ROM* (p. 30), Hong Kong.

Barclay, D.W. (1991). Interdepartmental conflict in organizational buying: The impact of the organizational context. *Journal of Marketing Research, 18*(28), 145-159.

Bolton, L. (2002). Information technology and management. Retrieved December 15, 2002, from *http://opax.swin.edu.au/~388226/howto/it2/manage1.htm*

Boyett, J.H., & Boyett, T. (2000). The skills of excellence: The new knowledge requirements for the twenty-first century workplace. Retrieved June 3, 2002, from *http://www.jboyett.com/skillsof.htm*

Bridges, W. (1995). *Managing transitions, making the most of change.* Nicholas Brealey.

Burshy, D. (2001). Technology adoption—many roadblocks slow it down. *Electronic Design, 49*(9), 20-21.

Castle, R.D., Luong, H.S., & Harris, H. (2002). A holistic approach to organisational learning for leadership development. *Proceedings of the IEEE International Engineering Management Conference*, Cambridge, UK.

Co, H.C., Patuwo, B.E., & Hu, M.Y. (1998). The human factor in advanced manufacturing technology adoption: An empirical analysis. *International Journal of Operations & Production Management, 18*(1), 87-106.

Conner, D.R. (1992). *Managing at the speed of change.* New York: Villard Books.

Constantine, L.L. (1993). Coding cowboys and software sages. *American Programmer, 6*(7), 11-17.

Constantine, L.L. (1994). Leading your team wherever they go. *Constantine: Team Leadership, Software Development, 2*(12), 1-6.

Constantine, L.L. (1995). *Constantine on Peopleware.* Englewood Cliffs, NJ: Prentice Hall.

Constantine, L.L. (1996, October). Panel on soft issues and other hard problems in software development (Ward Cunningham, Luke Hohmann, Norman Kerth). *Proceedings of OOPSLA '96* (pp. 6-10), San Jose, CA.

Cooper, R.B. (1994). The internal impact of culture on IT implementation. *Information and Management, 27*(1), 17-31.

DeMarco, T., & Lister, T. (1987). *Peopleware: Productive projects and teams.* Dorset House.

Eason, K.D. (1983). *The process of introducing information technology: behavior and information technology* (pp. 197-213). New York: Prentice-Hall.

Fayad, M.E., & Laitinen, M. (1998). *Transition to object-oriented software development.* New York: John Wiley & Sons.

Fiedler, F.E. (1967). *A theory of leadership effectiveness.* New York: McGraw-Hill.

Fingar, P. (1996). *The blueprint for business objects.* New York: SIGS Books.

Gibson, S. (1999). Videoconferencing still shy of adoption. *PC Week, 16*(13), 130-131.

Hersey, K.H., Blanchard, & Johnson, D.E. (1996). Management of organizational behavior. *Utilizing human resources* (8th ed.). Upper Saddle River, NJ: Prentice-Hall.

Hill, T., Smith, N.D., & Mann, M.F. (1987). Role of efficacy expectations in predicting the decision to use advanced technologies: The case of computers. *Journal of Applied Psychology, 72*(2), 307-313.

Humphrey, W.S. (1995). *A discipline for software engineering.* Reading, MA: Addison-Wesley.

Humphrey, W.S. (1997). *Managing technical people-innovation, teamwork, and the software process* (6th ed.). Reading, MA: Addison-Wesley-Longman.

Huse, E.F. (1975). *Organization development and change.* St. Paul, MN: West.

Ioannidis, A., & Gopalakrishnan, S. (1999). Determinants of global information management: An extension of existing models to firm in a developing country. *Journal of Global Information Management, 7*(3), 30-49.

Jacobson, I., Ericsson, M., & Jacobson, A. (1995). *The object advantage, business process reengineering with object technology.* Wokingham, UK: ACM Press.

Jurison, J. (2000). Perceived value and technology adoption across four end user groups. *Journal of End User Computing, 12*(4), 21-33.

Lawrence, P.R. (1969). How to deal with resistance to change. *Harvard Business Review,* 4-6.

Lockwood, L.A.D. (1991). Strategies for managing application development. *Fox Software Developers Conference Proceedings.* Toledo, OH: Fox Software.

Maslow, A. (1970). *Motivation and personality* (2nd ed.). Harper & Row.

Mize, J.H. (1987). *Success factors for advanced manufacturing systems.* Dearborn, MI: Society of Manufacturing Engineering.

Nambisan, S., & Wang, Y. (1999). Roadblocks to Web technology adoption? *Communications of the ACM, 42*(1), 98-101.

O'Hara-Devereaux, M., & Johansen, R. (1994). *GlobalWork* (p. 35).

Palvia, P.C., Palvia, S.C., & Roche, E.M. (1996). *Global information technology and systems management: Key issues and trends.* Nashua, NH: Ivy league Publishing.

Perkins, W.S., & Rao, R.C. (1990). The role of experience in information use and decision-making by marketing managers. *Journal of Marketing Research, 18*(27), 1-10.

Pfeffer, J. (1982). *Organizations and organization theory.* Marshfield, MA: Pitman.

Phillips, D.A. (2002). How effective is your leadership style? *IEEE Antenna's and Propagation Magazine, 44*(2), 124-125.

Schein, E.H. (1999). *Process consultation revisited: Building the helping relationship.* Reading, MA: Addison-Wesley.

Senge, P.M. (1990). *The fifth discipline: The art & practice of the learning organization.* New York: Doubleday/Currency.

Serour, M.K., Henderson-Sellers, B., Hughes, J., Winder, D., & Chow, L. (2002). Organizational transition to object technology: Theory and practice. *Proceedings of Object-Oriented Information Systems: 8th International Conference* (pp. 229-241), Montpellier, France. Berlin, Heidelberg: Springer-Verlag.

Stogdill, R.M. (1974). Historical friends in leadership theory and research. *Journal of Contemporary Business,* pp.7-7

Szewczak, E., & Khosrow-Pour, M. (1996). *The human side of information technology management* (p. 1). Hershey, PA: Idea Group Publishing.

Thite, M.R. (2000). Leadership styles in information technology projects. *The International Journal of Project Management, 18*(4), 235-241.

Thite, M.R. (2001). Help us but help yourself: The paradox of contemporary career management. *Career Development International, 6*(6), 312-317.

Ushakov, I.B. (2000). Introducing an OO technology in non-OO standard environment. *Proceedings of the 4th IEEE International Symposium and Forum on Software Engineering Standards* (pp. 1-5), Curitiba, Brazil.

Younessi, H., & Marut, W. (2002). *The impact of training and education in the success of introducing object-oriented into an organization.* Unpublished Paper.

Zakaria, N., & Yusof, S. (2001). The role of human and organizational culture in the context of technological change. *IEEE Software,* 83-87.

Chapter XI

Web Services in Service-Oriented Architectures

Gerald N. Miller
Microsoft Corporation, USA

Abstract

There is little debate in either corporate or academic circles that Web services comprise a large part of the next technological wave. Clearly, Web services will be instrumental in building service-oriented architectures that integrate disparate systems, both within organizations and across business partners' firewalls. The question is not if, or even when, to implement Web services—it is how.

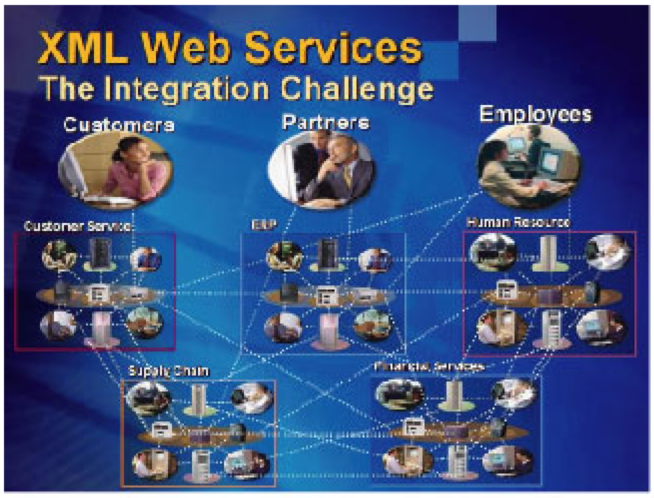
Introduction

According to nearly every industry pundit, including those from ardent competitors such as Sun Microsystems, IBM, and Microsoft Corporation, integration of systems is critically important for most enterprises. The ability to quickly assimilate and aggregate large amounts of information from disparate systems can mean the difference between life and death for an organization. Ease of access by customers, seamless supply chain management with business partners—these are quickly becoming the only distinguishing factors in an increasingly commoditized marketplace.

One of the problems with integrating computer systems is the incredible complexity and associated cost of doing so. Many systems are old and scantily documented; still others are proprietary with no natural hooks into their data. And these are just problems that exist within a company's firewall—imagine how the complexity increases as an enterprise begins integrating its systems with those of its business partners and customers, with the added security ramifications brought on by Internet communications!

The sheer number of interconnections is another problem. Companies have many systems—the alphabet soup of ERP, HR, CRM, SCM—each with many constituents. The geometric complexity of all these interconnections begins to stagger the imagination. It is no wonder that so many integration efforts fail to bring about promised savings or other business benefits.

Figure 1. The complexity of integrating systems



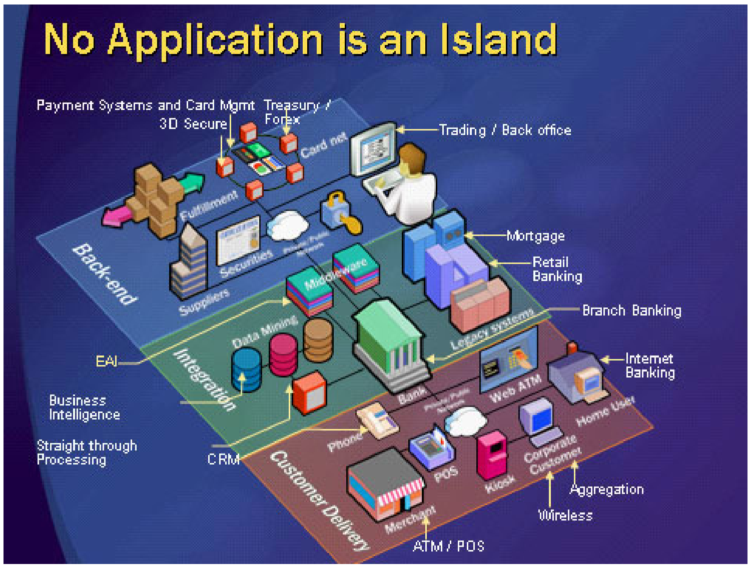
Web services offer a tidy solution to this integration mess. Rather than having to understand each system's deep underlying data structures, or having to write point-to-point application integration code, companies can simply add a Web services layer around individual systems that exposes necessary information and functionality in a standard way. Integration then becomes an effort to orchestrate business process by making Web services calls, rather than a massive retooling effort.

Service-Oriented Architecture

In a global business environment characterized by increasing competition, the demands on IT organizations continue to press for more and more agility. IT is being asked to do more with less, to provide quicker and higher returns on investments while faced with shrinking budgets. At the same time, these organizations are looking to provide their constituents with systems that "just work" reliably and securely.

A service-oriented architecture can play a key role in helping IT organizations be successful in these endeavors. Simply put, a service-oriented architecture,

Figure 2. Systems need to talk to each other

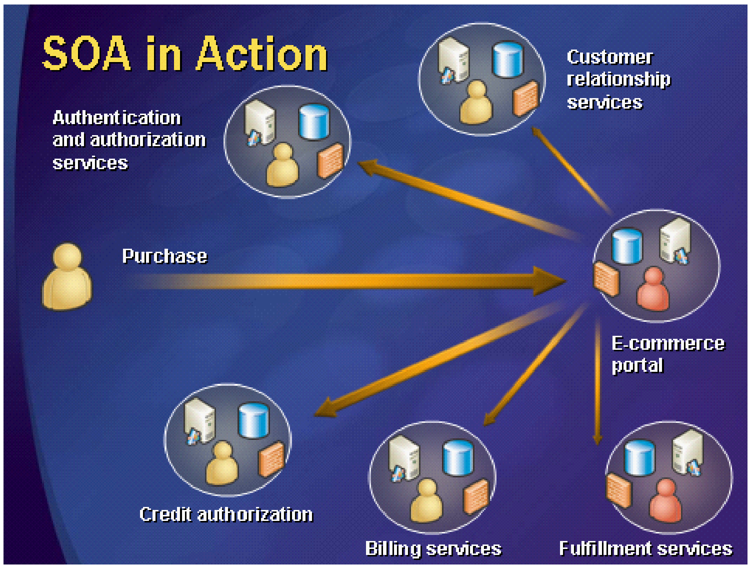


or SOA, is an approach to organizing IT systems such that data and logic are accessed by routing messages between network interfaces. In doing so, the specific implementation of each service becomes irrelevant so long as its interface remains the same. In other words, SOA allows consistent, stable interfaces to diverse and volatile implementations.

The services invoked in a service-oriented architecture represent unique business capabilities or operations, and as such align closely with the semantics of the business. For example, services for most organizations can be broken into categories such as human resources, corporate procurement, and manufacturing. The set of human resource services might include functions such as hire and fire, while procurement might includes services to retrieve catalogs of items, or to place orders.

In a service-oriented architecture, organizations explicitly delineate service boundaries and make sure that each service is entirely autonomous. In order to communicate with each other, services share their schema—the definition of how they expect to receive information, and how they plan to return information. They do not share implementation details. This type of intercommunication is described as policy based—services declare the policy necessary to

Figure 3. Describing business functions as services



communicate with them, never the details of how they plan to perform their job.

While object-oriented programming techniques are often used to develop individual services, services are definitely not objects. Objects are assumed to be nearby; services are assumed to be far away. Objects are assumed to have limited latency; services are assumed to have variable latency. Objects are described in terms of methods and properties; services are described in terms of messages and routing. Where objects are generally accessed synchronously, services are most often accessed asynchronously.

Because service orientation describes how the business actually works while defining a stable set of interfaces to reach these capabilities, it significantly reduces the complexity of connecting the various systems that exist inside organizations. An SOA makes a company more agile by facilitating controlled change for continuous improvement. In short, a service-oriented architecture is an approach to manage the complexity of enterprise application portfolios and an architecture for cross-organizational process integration.

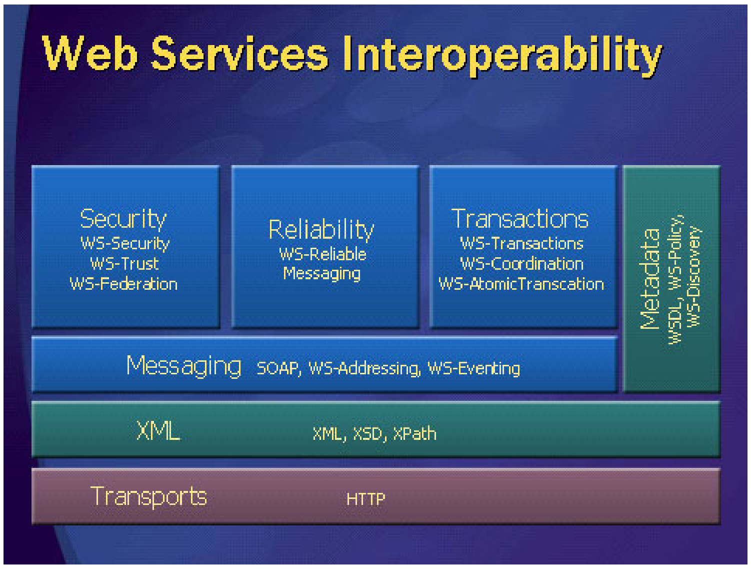
Why Web Services?

In order to be useful in the real world, a service-oriented architecture requires a way to actually connect the network of services. This means that SOA needs a new distributed component model that provides remote access to logic in a standard way. This is where Web services come to the rescue.

At its very root, a Web service is nothing other than a server that listens for and replies with SOAP, generally via HTTP. In practice, a Web service will support WSDL to describe its interfaces, and should also be listed in a UDDI registry. Of course, at this point Web services are deceptively simple—but the industry is coalescing around Web services standards for security, transactions, state management, and workflow. Every member of the technology community has a vested interest in developing these standards.

One interesting group of standards is referred to as the set of WS-* standards. These standards add functionality to the base Web services specification to provide for business-critical attributes such as end-to-end security, reliable delivery, message routing, and transactions. As these standards become ingrained in servers, applications, and even hardware—it is not too hard to

Figure 4. The WS- standards*



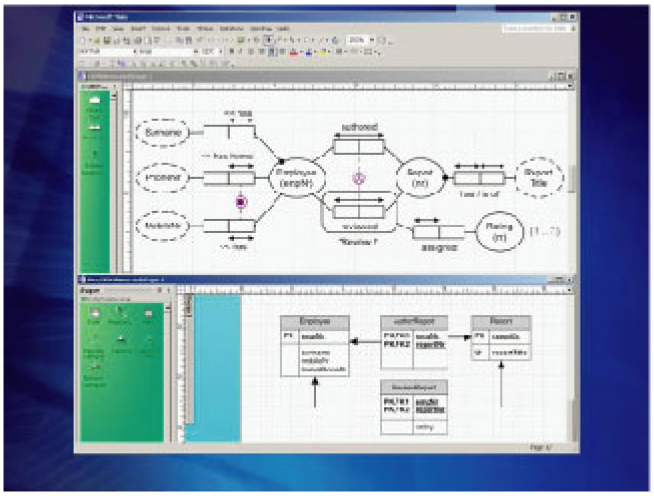
imagine XML routers—more and more service-oriented architectures supported by Web services will emerge.

Which Platform?

Most Web services development is being accomplished today using either Microsoft .NET or Sun Microsystems' J2EE specification. Both are able platforms that have a distinct place in corporate development. Because Microsoft .NET is a newer specification, the rest of this chapter will describe its advantages over J2EE. Still, bear in mind that J2EE is a highly capable platform with years of successful implementations behind it. Also, as Microsoft and the J2EE community come closer together, these platform differences will become less and less important as the platforms themselves provide more and more interoperability between themselves.

The business world is very large, and there is certainly room for several Web services development tools. In some cases it will be very clear whether Microsoft .NET or J2EE (or some other technology) is appropriate—for example, when a company has existing investments in a particular technology,

Figure 5. Microsoft Visual Studio .NET design tools for architects



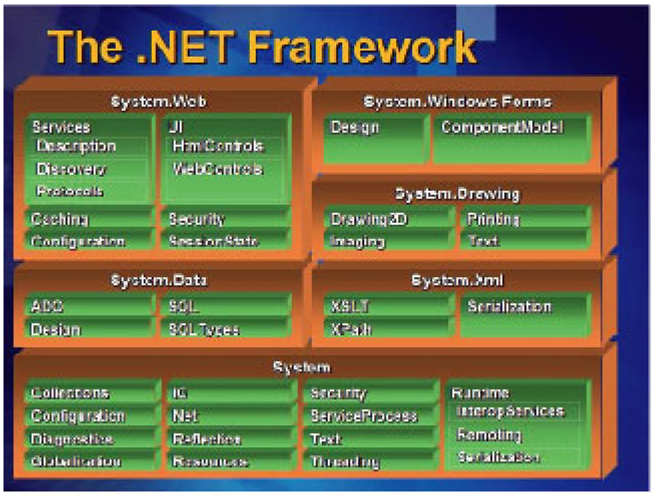
is experiencing resource constraints, or has limitations based on software they are already using. In most cases, however, Microsoft .NET will currently offer a considerable advantage based on time-to-market, performance, and overall cost of solution.

In Business, the Faster the Better

One of the main goals of application development is to get the best solution possible as quickly as you can. To do this, the development tools must address the needs of enterprise architects, system designers, application developers, and quality assurance testers.

Microsoft Visual Studio, the flagship suite of tools for Microsoft .NET development, is almost unanimously hailed as the best development suite on the market. The product allows enterprises to define templates for consistency across all development projects; it allows architects to use graphical design tools to generate program documentation; it provides developers with the widest choice of languages and features anywhere; and it provides testers a rich debugging environment to monitor end-to-end program flow.

Figure 6. A sample of the .NET framework



As compelling as Visual Studio is, Microsoft .NET programs do not have to be built with that tool; other vendors such as Borland offer great development suites, and there are even shared source C# compilers available.

Microsoft .NET programs run inside the Common Language Runtime (CLR), just as J2EE programs run inside the Java Virtual Machine (JVM). The .NET Framework adds a rich library of functionality to the CLR, considerably stronger than the additional capabilities J2EE brings to the JVM. In J2EE, even simple tasks become difficult—for example, EJB performance problems cause most programmers to use bi-modal data access, which requires them to write twice as much code. Because the .NET Framework is so rich, programmers will typically have to write significantly fewer lines of code than they will with J2EE—for example, one of J2EE's most ardent supporters recently concluded that the most optimized Java Pet Store possible requires 14,000 lines of code under J2EE, but only 4,079 using the .NET Framework.

So, what does a 71% reduction in coding mean? It means the application is finished that much quicker. It also means a shorter QA cycle and a more stable and secure product, since bug counts grow with line counts.

Simply put, .NET applications will see the light of day long before their J2EE brethren.

Performance

Much has been made about .NET vs. J2EE performance, so I will briefly mention how much faster .NET solutions will tend to run. Still, overall performance is rarely a critical factor in enterprise systems. More important are figures such as how many users the system can support per server, because this translates directly into deployment and maintenance cost.

The Middleware Company's recent Java Pet Shop study is highly relevant to this discussion. The Middleware Company is a significant player in the J2EE world, deriving nearly all its revenue from Java and J2EE publishing and consulting. The company also runs TheServerSide.com, one of the most popular J2EE Web sites.

The Java Pet Shop study concluded that Microsoft .NET requires one-fifth the code as J2EE, supports 50-600% more users (depending on the application server), and offers nearly twice the transactions/second on a significantly less expensive platform—all with no errors, while both J2EE application servers studied threw exceptions, and one could not even complete the benchmark. This from a J2EE-biased company that even Sun regards as a definitive source of information!

Even Sun's own data from the JavaOne 2002 conference shows that 86% of J2EE users surveyed had performance concerns. It is telling, too, that there are no J2EE-based applications in the TPC-C benchmarks. For example, IBM's TPC-C submissions use either Microsoft COM+ or IBM's older, non-Java/non-J2EE transaction processing monitor formerly known as Encina.

The J2EE community often counters the performance argument by claiming that J2EE shops trade speed for portability. This is a red herring, as there is no real portability between J2EE application servers—even Oracle's Web site admits, "Though in theory, any J2EE application can be deployed on any J2EE-compliant application server, in practice, this is not strictly true." IBM's Web site contains a white paper over 200 pages in length explaining how to port a J2EE application from BEA WebLogic to WebSphere. The portability argument is truly nothing but FUD and misdirection.

Simply put, .NET applications are faster and support more users than comparable J2EE solutions.

Cost of Solution

The total cost of a solution consists of how much money an enterprise spends to build and then deploy the application. In both cases, Microsoft .NET offers a significant advantage over J2EE.

When building a J2EE solution, programmers have a severely limited range of language choice—they can only use Java. On the other hand, the .NET Framework supports almost 30 languages. In fact, Visual Studio supports four languages out of the box (J#, a Java-syntax language, C#, C++, and Visual Basic .NET), giving each language identical support. While C# is a great language for .NET development, the .NET Framework is in no way optimized for C# only. In addition, other vendors offer a wide array of additional languages for .NET development, including JScript, Perl, COBOL, Fortran, Smalltalk, even Eiffel and Mondrian! Clearly, companies will generally not build solutions using, say, 10 languages at once. However, the *choice* to use whichever language makes sense for a particular project can generate significant savings, because companies can use existing programmers without retraining them for Java. In addition, Visual Basic programmers tend to be more plentiful and less expensive than Java programmers. And, we previously established that once developers begin writing code, they will have to write a lot less code, translating again into significant savings.

Once the application is ready, it must be deployed. In the .NET world, this means running it on an Intel-based server with Windows 2000 Server or the upcoming Windows Server 2003. According to The Middleware Company, with software costs running approximately $5,990, a fully configured server for the .NET solution will cost around $36,990. Contrast this with the same server running a commercial J2EE application server, which adds between $40,000 and $48,000 per server! Even if an enterprise chooses Linux to eliminate the $5,990 Windows license, they still have to add the application server. Even today's 'new math' can't justify eliminating less than $6,000 for an additional $48,000 cost.

Finally, because performance data indicate that J2EE solutions support fewer users than .NET solutions on comparable hardware, the J2EE solutions will generally require more of these more expensive servers.

.NET solutions are simply less expensive to build, less expensive to deploy, and less expensive to maintain.

Conclusion

Web services are clearly critical for the next wave of enterprise computing—integrating disparate business systems for more effective use of information. Companies like Microsoft and Sun should be commended for their clear commitment to work together toward a common industry standard for all our customers' benefit.

There should always be more than one choice of development environment for customers, because there will never be one solution that is always appropriate for everyone in every situation. The world is big enough for both Microsoft .NET and J2EE. Still, for the reasons outlined herein, Microsoft .NET will generally be a better choice for most companies in most situations.

Acknowledgment

The opinions expressed herein are those of the author and may not reflect those of his employer. This chapter is an expansion of an article that originally appeared in *Communications of the ACM.*

Chapter XII

Model-Based Development: Metamodeling, Transformation and Verification

Juan de Lara
Universidad Autónoma de Madrid, Spain

Esther Guerra
Universidad Carlos III, Spain

Hans Vangheluwe
McGill University, Canada

Abstract

Since the beginning of computer science more than 50 years ago, software engineers have sought techniques resulting in higher levels of quality and productivity. Some of these efforts have concentrated in increasing the level of abstraction in programming languages (from assembler to structured languages to object-oriented languages). In the last few years, we have witnessed an increasing focus on development based on high-level, graphical models. They are used not only as a means to document *the analysis and design activities, but also as the actual "implementation" of the application, as well as for automatic analysis, code, and test case generation. The notations used to describe the models can be standard and general purpose (for example, UML) or tightly customized for the application domain. Code generation for the full application is only accomplished for specific, well-understood application domains. A key initiative in this direction is OMG's Model-Driven Architecture (MDA), where models are progressively transformed until executable code is obtained. In this chapter, we give an overview of these technologies and propose ideas following this line (concerning metamodeling and the use of visual languages for the specification of model transformation, model simulation, analysis and code generation), and examine the impact of model-based techniques in the development process.*

Introduction

Stakeholders in the development process have different interests. Managers want the product on time and within cost, users want more functionality and low prices, and developers want to reduce the effort in building the application. One of the means of reducing this effort is by increasing the level of abstraction of programming languages (that is, conceptually using a higher-level virtual machine). Using higher abstraction level notations, programs become more compact and easier to understand, write, and maintain. In this way, developers deal with less (accidental) details about the system they are building and concentrate on describing its essential properties (Brooks, 1995). Usually, powerful abstract constructs are only available in well-understood application domains, such as editors for visual languages (de Lara & Vangheluwe, 2002a). Ideally in these domains, from (possibly graphical) high-level descriptions of the application to be built, the program code is automatically generated. Other times, these models of the application are used for analysis of the program properties (such as efficiency, scalability, or design correctness). This is possible if the model semantics are formally defined (and have adequate analysis techniques) or if the model is translated into a formalism for which verification techniques are available (Guerra & de Lara, 2003; Heckel, Küster, & Taentzer, 2002).

For specific domains, the application to be generated can be described through metamodels. These are models specified using a high-level notation (such as UML class diagrams) and describe the set of all possible instances the user can build. Note how metamodels are static descriptions of the application to be generated. To describe its behavior, several approaches can be used. For specific domains, it is possible to offer a number of predefined functionalities in the generated tool. For example, in the case of generation of customized visual editors, one could have predefined functionality in the generated editor, for example to create, edit, or connect modeling primitives. Another possibility is to specify the functionality using the programming language of the generated code. Finally, this functionality can be specified using visual notations at the meta-level. The latter approach is the one we follow in this chapter, although usually, one finds a mixing of the three approaches.

A further step in model-based development is the proposed Model-Driven Architecture (MDA) initiative by OMG (MDA, 2004; Raistrick, Francis, Wright, Carter, & Wilkie, 2004). In this envisioned paradigm, software engineers create a Platform-Independent Model (PIM) of the design, which is automatically transformed (refined) into a Platform-Specific Model (PSM) from which executable code is generated. Model transformation is thus a key issue in this technology. Code generation can be seen as a special case of the latter.

In this chapter, we present our vision of metamodeling, and show how its combination with graph transformation is a powerful approach for domain-specific application generation and for model transformation with the aim of verification. These concepts are illustrated with some examples using the AToM^3 tool (de Lara & Vangheluwe, 2002), built by the authors, and the implications of these technologies in the development process are discussed.

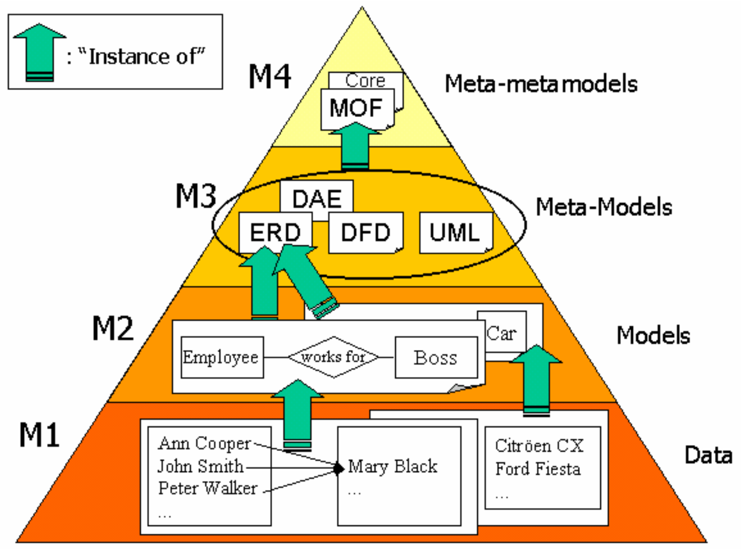
Metamodeling

Metamodeling allows describing visual languages (formalisms) using a (possibly graphical) high-level notation that we call a "meta-formalism." These are formalisms expressive enough to describe other formalisms' syntax. A formalism is defined by a metamodel, while a meta-formalism is described by a meta-metamodel.

Most modeling environments (e.g., UML 1.5 Specification, 2003) are organized in four metamodeling layers as Figure 1 shows. The highest layer (M4) contains the description of different meta-formalisms used for specifying formalisms at the M3 layer. At the M4 level we must have at least a means to specify entities (*MetaClasses*), data (*MetaAttributes*), operations (*MetaOperations*), and relationships (*MetaRelationships*). These concepts are organized in the description of the different meta-formalisms. For example, in the UML meta-architecture, these elements are organized as the MOF (Meta Object Facility—UML 1.5 Specification, 2004) meta-metamodel, which is very similar to a core subset of UML class diagrams. This meta-metamodel is used to describe the UML syntax (the UML metamodel) at the M3 level.

In the M3 layer we describe the syntax of the different formalisms we are interested in, using the meta-formalisms in the M4 layer. For example, in this layer we could have descriptions of State Automata, the different UML diagrams, Differential Algebraic Equations, and so forth. The different models that can be described using one of the formalisms in layer M3 belonging to the M2 layer. Finally, the M1 layer contains data resulting from the execution of the models in the M2 layer. All meta-levels contain models, which should be

Figure 1. Metamodeling levels

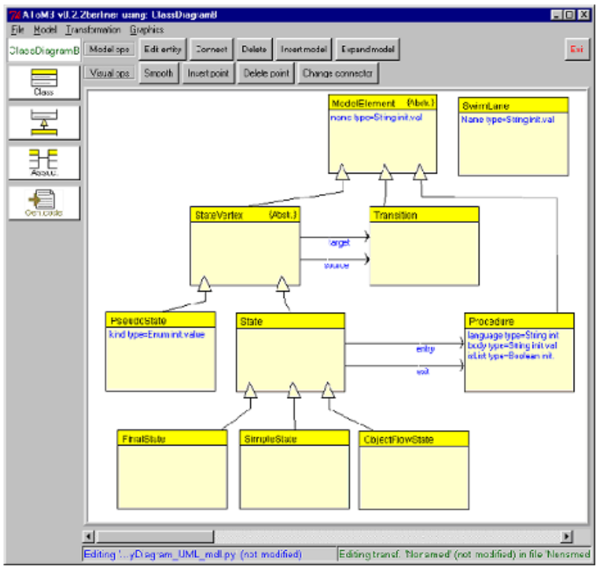


consistent with the description of the model at the upper layer and are instances of these. Some metamodeling approaches are strict (Atkinson & Kühne, 2002), in the sense that every element of the *n* meta-layer is an instance of some other element at the *n+1* meta-layer. The exception is the highest layer, in which each meta-formalism definition must be consistent with the meta-formalism in which it was defined (possibly itself).

A metamodeling environment is able to generate a modeling tool for a formalism, given the description of its metamodel. For the generation of such a tool, not only must one include in the metamodel information about the entities and their relationships (abstract syntax), but also about the visualization of the different elements (concrete syntax). It may be possible to have arbitrary mappings from abstract to concrete syntax.

This metamodeling approach is immediately applicable to the generation of visual modeling environments (de Lara & Vangheluwe, 2002; Lédczi el al., 2001), where the modeling language is described by a metamodel. Figure 2 shows a metamodel for UML Activity Diagrams (built with the AToM^3 tool) which closely follows the UML 1.5 specification. Note how we have included a visualization element (the SwimLane), which does not belong to the abstract

Figure 2. A metamodel for Activity Diagrams

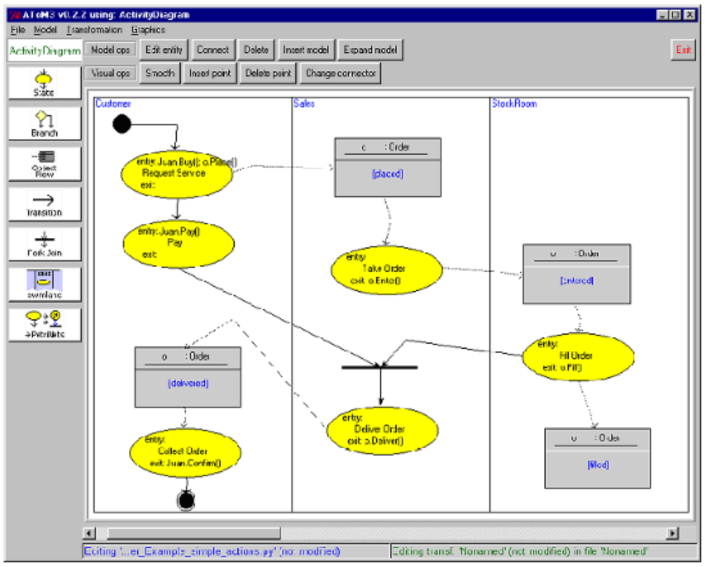

AToM3 v0.2.2berlinе: using: ClassDiagramB
File Model Transformation Graphics
ClassDiagramB
Model ops
Edit entity
Connect
Delete
Insert model
Expand model
Exit
Visual ops
Smooth
Insert point
Delete point
Change connector
Class
Assoc.
Gen.code
ModelElement (Abst.)
name type=String init.val
SwimLane
Name type=String init.val
StateVertex (Abst.)
Transition
target
source
PseudoState
kind type=Enum init.value
State
Procedure
language type=String init
body type=String init.val
isList type=Boolean init.
entry
exit
FinalState
SimpleState
ObjectFlowState
Editing '...yDiagram_UML_mdl.py (not modified)
Editing transf. 'Nonamed' (not modified) in file 'Nonamed

syntax, but to the concrete one. Swim lanes are a means to visually arrange the activity states. In addition, we have to define the concrete visualization for each element in the metamodel. In AToM^3 this is done with a visual editor, where the user can draw icon-like graphics (for classes) or arrow-like graphics (for associations).

Figure 3 shows the generated tool from the metamodel in Figure 2, where a model of a sales process has been drawn. Note that the user interface of the tool has changed with respect to Figure 2, in particular the row of buttons to the left. These buttons are used to create the modeling primitives defined in the metamodel, as well as to access further user-defined functionality.

In general, one has three options regarding the means to specify the functionality of the generated environment (although they can be combined). The generated environment may be made of some predefined code (that we call the kernel), which is completed by some other code generated from the metamodel. The predefined code may implement some core functionality, common to all the generated applications. This is a good approach if the application domain is well understood, but sometimes different, specific functionality is needed for each generated application. This is the approach in AToM^3, which is made of a

Figure 3. An Activity Diagram model



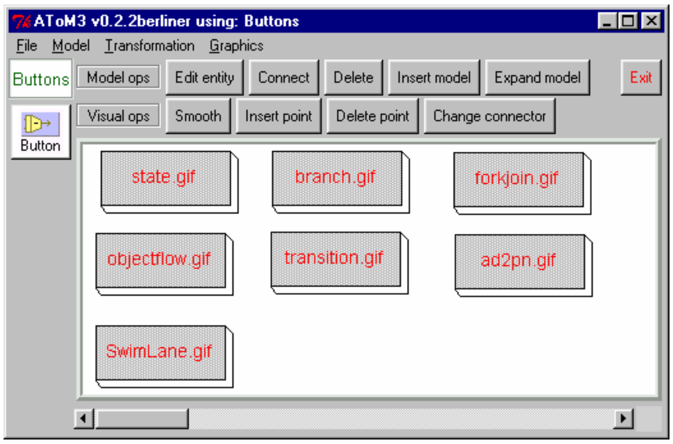
kernel, with the functionality to load and save models, generate code from them, and create the entities of the loaded metamodels.

Extra functionality can be included in the generated application by coding (by hand) in the implementation language. This has the disadvantage of being low-level (in comparison with the approach used to describe the metamodel) and requires knowledge about the kernel API in order to implement the new functionality. This can also be done in AToM^3 by adding code in Python.

Finally, one can include information about the functionality of the generated application at the meta-level, using high-level notations. In AToM^3 one can configure a part of the user interface by modifying a model that is automatically generated from the metamodel. In this model the user creates the buttons that will appear in the user interface when the formalism that he has designed is loaded (left row of buttons in Figures 2 and 3). Figure 4 shows the model of the user interface for the Activity Diagrams tool shown in Figure 3.

Some of the buttons in the model in Figure 4 were automatically generated (the ones to create the primitives defined in the metamodel), but the user can add other buttons. Each button is provided with functionality specified either in Python code or as graph transformation rules (Rozenberg, 1997). The functionality usually consists on some computation performed using the current instance of the described metamodel. Models, metamodels, and meta-metamodels can be described as attributed, typed graphs. In this way,

Figure 4. Model of the user interface for the Activity Diagrams modeling environment



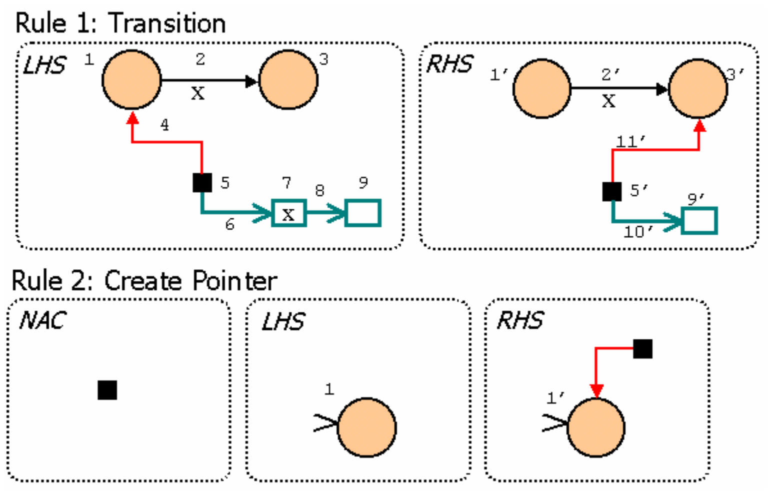
computations on models can be naturally, graphically, and formally described as graph transformation rules. This is a concept that will be explained in the next section.

Model Transformation with Graph Grammars

Graph grammars (Rozenberg, 1997) are similar to Chomsky grammars (Aho et al., 1986), which are applied on strings, but rules have graphs in left- and right-hand sides (LHS and RHS). Graph grammars are useful to generate sets of valid graphs or to specify operations on them. In order to apply a rule to a model, a graph rewriting processor looks for graph matchings between the LHS of a rule and a zone of an input graph (called host graph). When this happens, the matching subgraph in the host graph is replaced by the RHS. Rules may have conditions that must be met in order for the rule to be applied and actions that are performed once the rule is applied. Some graph rewriting processors (such as AToM^3) iteratively apply a list of rules (ordered by priority) to the host graph until none of them is applicable. When a rule can be applied, the processor again starts trying the rule at the beginning of the list. Other processors have a (possibly graphical) control language to select the rule to be considered next.

In our approach, we use graph grammars to specify operations on models (typically model execution, optimization, and transformation) at any meta-level, as these can be expressed as attributed, typed graphs. In this case, the attributes of the nodes in the LHS must be provided with the matching conditions. In AToM^3, we can specify that either a specific value or any value will make a match. Nodes in both LHS and RHS are also provided with labels to specify the mapping between LHS and RHS. If a node label appears in the LHS of a rule, but not in the RHS, then the node is deleted when the rule is applied. Conversely, if a node label appears in the RHS but not in the LHS, then the node is created when the rule is applied. Finally, if a node label appears both in the LHS and in the RHS of a rule, the node is not deleted. If a node is created or maintained by a rule, we must specify in the RHS the attributes' values after the rule application. In AToM^3 there are several possibilities. If the node label is already present in the LHS, the attribute value can be copied. We also have

Figure 5. Two rules for the specification of a simulator for automata



the option of giving it a specific value or assigning it a program to calculate the value, possibly using the value of other attributes.

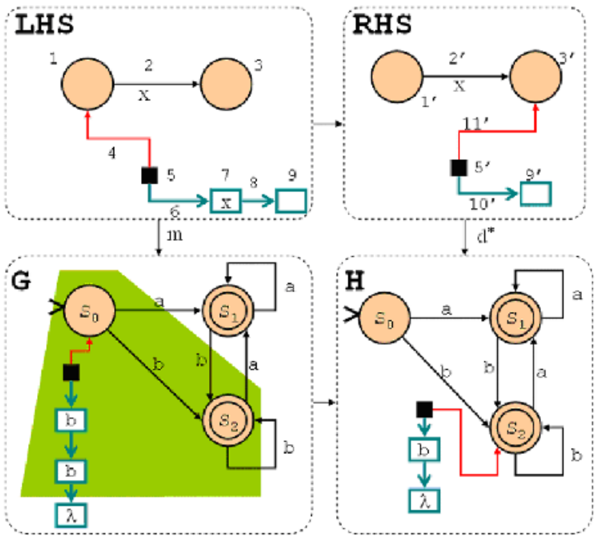
Figure 5 shows an example, where two rules for the definition of a simulator for finite automata are shown. The arrow labeled as "4" in the LHS of Rule 1 is the pointer to the current state. The black rectangle labeled as "5" points to a list that contains the input stream. The first element is labeled as "7" and its value is the same that the one in transition 2. If the rule is applied, then the first element in the input stream is consumed and the current pointer is moved through transition 2 to the next state. The second rule has a Negative Application Condition (NAC). NACs specify conditions that should not be present in the host graph in order for the rule to be applied. The rule creates a pointer to the automaton initial state in case there is not one already.

Figure 6 shows the application of Rule 1 to a model G, resulting in a model H. The occurrence of the LHS in G has been shaded. The marked elements have been substituted by the corresponding elements in the rule RHS, resulting in model H. Note how the successive application of Rule 1 results in the simulation of the model. Morphisms between LHS and the host graph can be non-injective; in this way, nodes 1 and 3 in Rule 1 can be identified into a single node in the model (that is, we do not need an extra rule for the case of self-loop transitions), so we can again apply Rule 1 to Model H in Figure 6.

Using a model of the computation in the form of a graph grammar has several advantages over embedding the computation in a lower-level, textual language. Graph grammars are a natural, graphical, formal, and high-level formalism. Its theoretical background can help in demonstrating the termination and correctness of the computation model. Nonetheless, its use is constrained by efficiency as in the most general case, subgraph isomorphism testing is NP-complete. However, the use of small subgraphs on the LHS of graph grammar rules, as well as using node and edge types and attributes, can greatly reduce the search space in the matching process.

The example transformation showed in Figures 5 and 6 "animate" a model, by defining the formalism operational semantics. In the context of model-driven development, there are other useful transformations, for example those translating a model instance of a source metamodel into a model instance of a target metamodel. There are several reasons to do this. On one hand the formalism represented by the target metamodel may have appropriate methods for analysis. In this case, the property under investigation in the source model has to be preserved during the transformation. If the transformation is expressed as a graph grammar, one can use its theoretical results to show certain properties of the transformation itself (de Lara & Taentzer, 2004), among them semantic

Figure 6. Application of a rule to a model

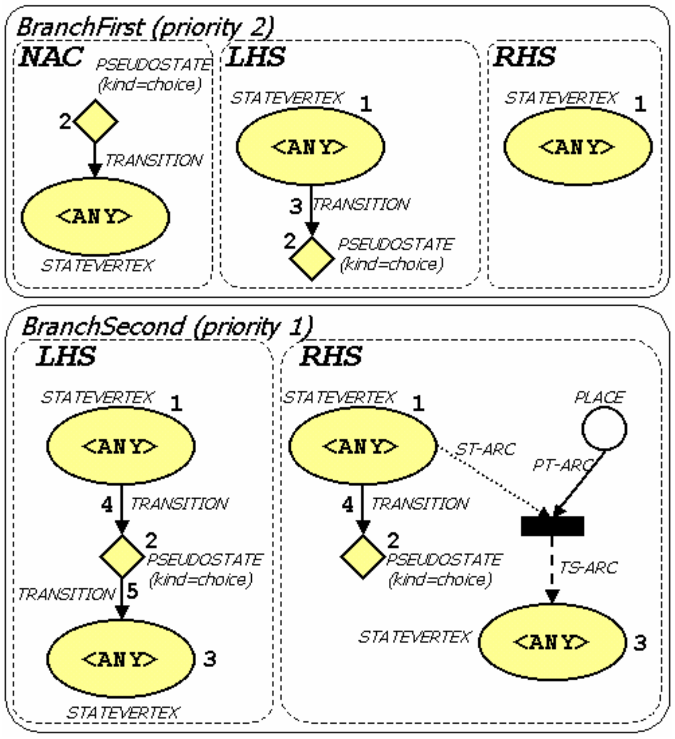


consistency (preservation of semantic properties—such as behavior—of source models).

On the other hand, the target metamodel may be a more concrete (lower abstraction level) representation of the system. For example, a design model can be refined into a model of the code. Note that in this case, a metamodel for the coding language is needed, and the design is transformed into a model similar to the abstract syntax graph (Aho et al., 1986) that a compiler obtains when parsing a textual program. These transformations for refinement are the ones that can be found in the context of the MDA. In this case, one is also interested in showing the preservation of certain model properties.

Figure 7 shows two of the rules for transforming Activity Diagrams into Petri nets (Murata, 1989). Petri nets offer analysis methods that allow us to verify properties that cannot be analyzed in UML, as its semantics are not formally defined. Both rules deal with the translation of choice pseudostates (with

Figure 7. Some rules for transforming Activity Diagrams into Petri nets



diamond shape). The second rule adds a Petri net place for each condition, and the first rule eliminates the choice once no more output transitions remain. After the repeated application of both rules, the input and output states to the choice pseudostate are connected to a number of Petri net transitions. That is, during the transformation, models are a mixing of the source metamodel elements and target metamodel elements. At the end of the transformation, the model only has elements of the target metamodel. In order to define the possible relationships of source and target elements during the transformation, we create a metamodel to describe the kind of models that can appear during the translation. For example, the RHS of rule "BranchSecond" has a link of type "ST-ARC" that relates states (from Activity Diagrams) to transitions (from Petri nets). A similar situation occurs with the link of type "TS-ARC."

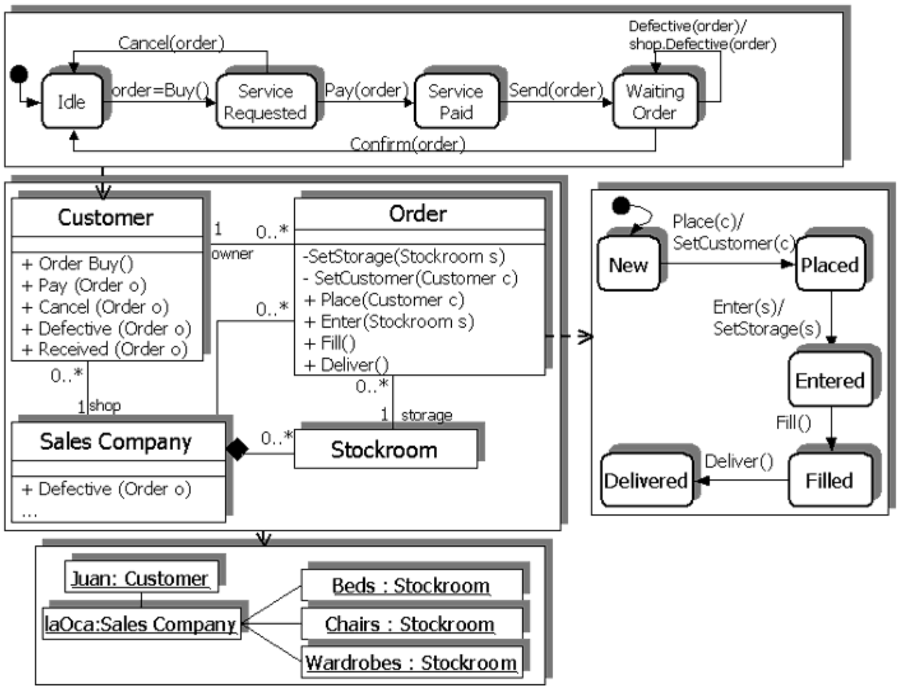
The next section shows an example, where we use transformations into Petri nets for the analysis of a UML model composed of Statecharts, Class, Object, and Activity Diagrams.

An Example

Figure 8 shows a UML Class Diagram with the classes involved in the design: a Sales Company is composed of a number of Stockrooms and has Customers that place Orders. We have defined a Statechart for Order objects (the diagram in the right-hand side of Figure 8). An Order object can be in one of the following states: New, Placed, Entered, Filled, or Delivered. The Statechart changes the state in reaction to method invocations (Place, Enter, Fill, and Deliver). Additionally, on receiving method invocation "Enter" with the stock-room where the "Order" is to be processed, private method "SetStorage" is invoked in order to store the "Stockroom" object (in "storage").

The Customer class has an associated Statechart, which models the fact that customers can buy items (calling the Buy method). Once the order has been issued, customers can either cancel or pay the order. Once the order is sent to the customer (method Send), he can either confirm that the content is correct (method Confirm), or that there are some defects (method Defective). Figure 8 also shows an object diagram that depicts the initial system configuration for the analysis (one Customer object, one Sales Company Object, and three Stockroom objects). The Activity Diagram in Figure 3 is also part of the

Figure 8. The Class, Object, and Statechart Diagrams for the example



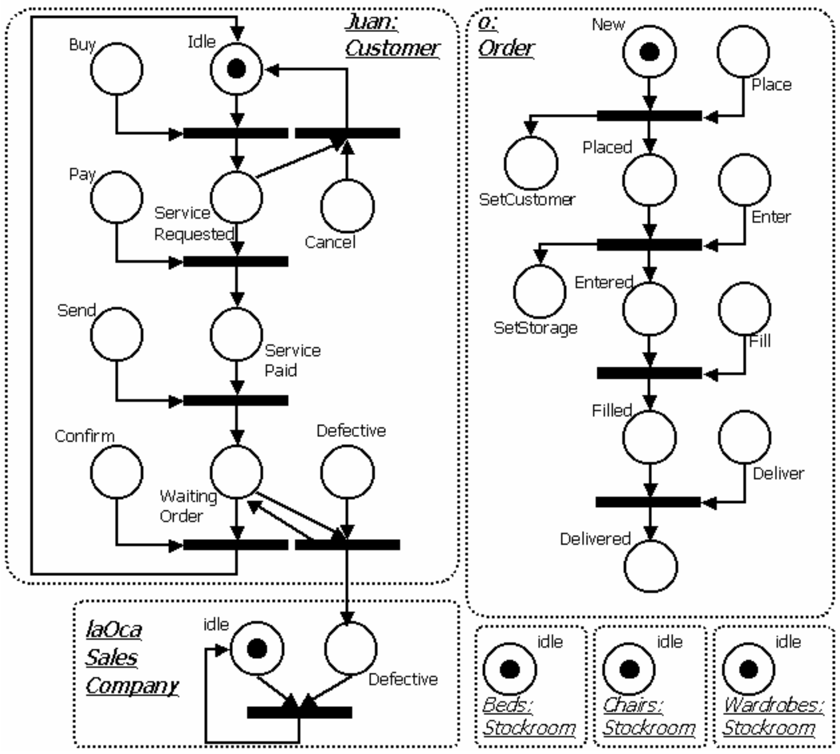
example and shows a typical sales process. This diagram could have been created in order to specify a use case, like "shell goods," for example.

The example we present here shows a typical situation in the development process, because there is incomplete information, as some classes (Sales Company and Stockroom) do not have an associated Statechart. We assume a general Statechart for all of them, composed of a single state, from which all the available class methods can be invoked.

In order to analyze the models correctness, we translate the Activity Diagram and the Statecharts into a single Petri-Net model to analyze if the models are consistent and to check properties of the resulting net (deadlock, reachable states, etc.) using Petri net analysis methods. Further properties can be investigated by generating (implicitly or explicitly) the state space of the net and performing model checking (Clarke, Grumberg, & Peled, 1999). This technique allows checking if the model verifies a certain property, which is specified using some kind of temporal logic (Computational Tree Logic in the example).

Our approach for the translation of Statecharts into Petri nets was shown in de Lara and Vangheulwe (2002b). The main idea is to translate each Statechart state into a place and represent methods as special places that we call "*interface places.*" These places form the interface of the Statechart, and when an object wants to make a method invocation, it puts a token in the corresponding place. The result of transforming the Statecharts in Figure 8 is shown in Figure 9. We have translated the Statechart of each object in the Object diagram (shown inside the rounded rectangles in Figure 9, that we call Petri net modules). Note how some of these Petri net modules are coupled because of the method invocations in the Statechart transitions. For example, in the Statechart for the customer, there is a method invocation to the Defective method of class Sales Company (in one of the transitions departing from the "waiting order" state). This is translated as an arc that puts a token in the place corresponding to the Defective method of the Petri net module for class Sales Company.

Figure 9. Transformation of the Statecharts models into a single Petri net model

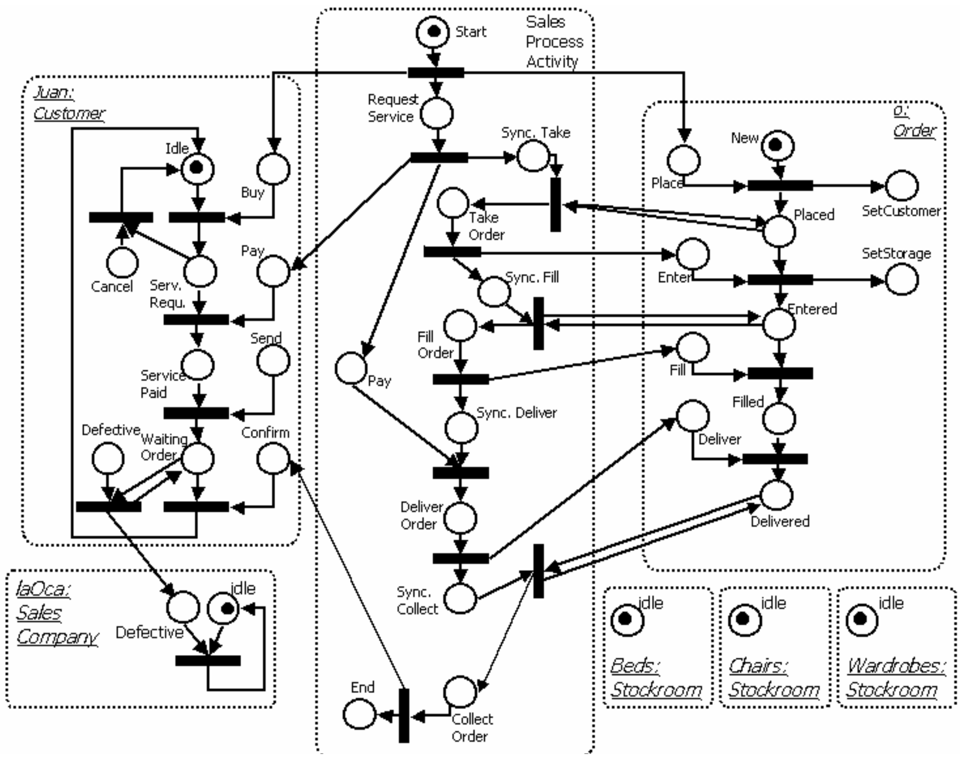


Once we have translated the Statecharts, the Activity Diagram is also translated. This diagram couples the already translated modules, as Activity Diagram states contain entry and exit actions with method invocations. The result of this transformation is shown in Figure 10.

Once all the diagrams have been transformed, we can use Petri net analysis methods (Murata, 1989)—for example, the ones based on the reachability graph, matrix of equations, simplification, and structural techniques. In general, the properties that one is interested in analyzing using these methods are:

- **Reachability of a certain state:** We may be interested in knowing whether one object or a combination of objects in our system can reach a certain state. This includes also error or inconsistent states, which can indicate a flaw in our models. As we model method invocations as places, we can also use this property to check if certain methods will be called.

Figure 10. Transformation of all the models

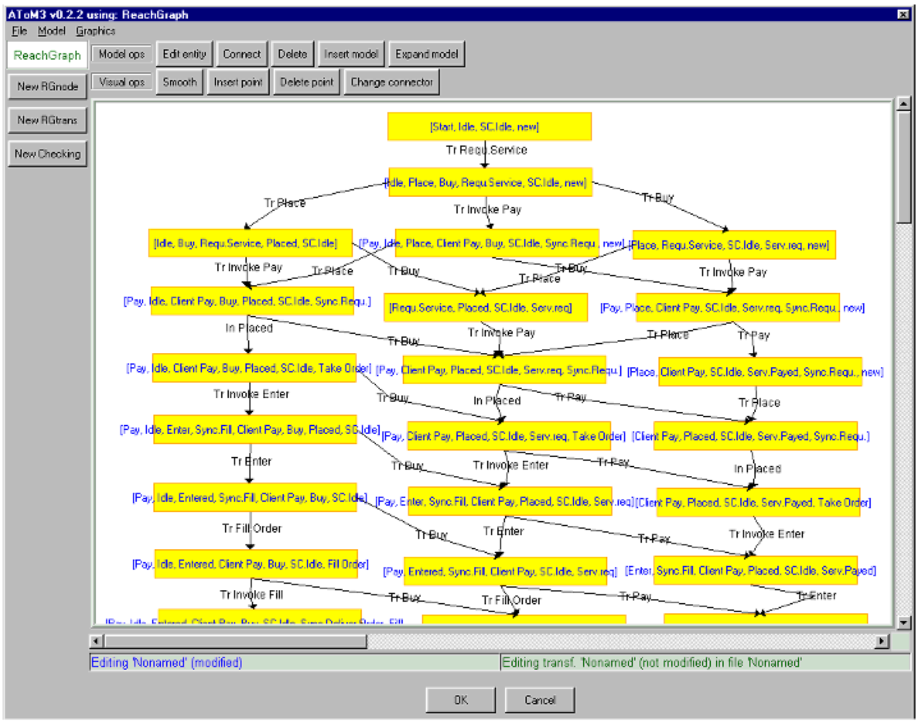


- **Liveness, which shows how often transitions can fire:** We can use this property to find deadlocks in our system. Deadlocks are situations when the execution of our system cannot progress and we typically want to avoid in our designs. Liveness can also tell us about how often a certain method can be called, or how often a certain state can be reached.
- **Boundedness and safeness, which investigate if the number of tokens in the net is bounded:** This property of the Petri net is granted by the translation procedure.
- **Persistence, which investigates if enabled transitions remain enabled after the firing of other transitions:** In our context this is related to the interruption of some process when another process is executed.
- **Place invariants (also related to conservation) show the relationships among the number of tokens in places that hold during all the possible executions:** In our context, we can use them to show if a certain relationship between the actual states of the object Statecharts hold in all executions.
- **Transition invariants show that a firing sequence leaves the net in the same state:** This is related to *reversibility*, which investigates if after an execution, we can again reach the initial state. We can use transition invariants to show that after several method invocations, the state of the system remains unchanged.

In the example, we use techniques based on the reachability graph, which represents the space state of the net. This graph is an approximation if the net is not bounded, and then is called coverability graph. Figure 11 shows a part of the reachability graph (in this case it is an exact representation of the state space) produced by the net in our example; the full reachability graph has 50 states. Note how there are many "interleavings" due to the possible parallel execution of method invocations and state changes.

Once we have the state space of the model (which represents all the possible executions of our system), we can use model checking (Clarke et al., 1999) to verify if certain properties hold. Properties can be specified using temporal logic, in our case "Computational Tree Logic." This logic allows us to express properties in computation paths using path quantifiers "A" (for all computation paths) and "E" (in all computation paths). For example, one can check if the system reaches deadlock (always or sometimes); if a certain object, or a

Figure 11. Reachability graph of the example


AToM3 v0.2.2 using: ReachGraph
File Model Graphics
ReachGraph
Model ops
Edit entity
Connect
Delete
Insert model
Expand model
New RGnode
Visual ops
Smooth
Insert point
Delete point
Change connector
New RGtrans
New Checking
[Start, Idle, SC.Idle, new]
Tr Requ.Service
[Idle, Place, Buy, Requ.Service, SC.Idle, new]
Tr Place
Tr Invoke Pay
Tr Buy
[Idle, Buy, Requ.Service, Placed, SC.Idle]
[Pay, Idle, Place, Client Pay, Buy, SC.Idle, Sync.Requ., new]
[Place, Requ.Service, SC.Idle, Serv.req, new]
[Pay, Idle, Client Pay, Buy, Placed, SC.Idle, Sync.Requ.]
[Requ.Service, Placed, SC.Idle, Serv.req]
[Pay, Place, Client Pay, SC.Idle, Serv.req, Sync.Requ., new]
In Placed
Tr Pay
[Pay, Idle, Client Pay, Buy, Placed, SC.Idle, Take Order]
[Pay, Client Pay, Placed, SC.Idle, Serv.req, Sync.Requ.]
[Place, Client Pay, SC.Idle, Serv.Payed, Sync.Requ., new]
Tr Invoke Enter
[Pay, Idle, Enter, Sync.Fill, Client Pay, Buy, Placed, SC.Idle]
[Pay, Client Pay, Placed, SC.Idle, Serv.req, Take Order]
[Client Pay, Placed, SC.Idle, Serv.Payed, Sync.Requ.]
Tr Enter
[Pay, Idle, Entered, Sync.Fill, Client Pay, Buy, SC.Idle]
[Pay, Enter, Sync.Fill, Client Pay, Placed, SC.Idle, Serv.req]
[Client Pay, Placed, SC.Idle, Serv.Payed, Take Order]
Tr Fill Order
[Pay, Idle, Entered, Client Pay, Buy, SC.Idle, Fill Order]
[Pay, Entered, Sync.Fill, Client Pay, SC.Idle, Serv.req]
[Enter, Sync.Fill, Client Pay, Placed, SC.Idle, Serv.Payed]
Tr Invoke Fill
OK
Cancel

Editing 'Nonamed' (modified)

Editing transf. 'Nonamed' (not modified) in file 'Nonamed'

combination of them, reaches (or leaves) a certain state (always or sometimes); if a certain method (or a chain of methods) is invoked always (or sometimes); or if certain property holds until some other property becomes true.

In our example, we can verify using model checking that the system enters in deadlock (in all possible computation paths), obtain the state in which this is produced and the chain of transitions leading to it. It is then easy to check that the Customer object is in state "Service Paid" when deadlock is present. Inspecting the Statechart, one finds that in order to leave that state, method "Send" has to be invoked, and this invocation has not been specified in any of the Statecharts nor in the Activity Diagram. That is, the order is in state "Delivered," but it has not been sent, so the customer cannot confirm its arrival. We can correct the design error by adding an extra exit action (Juan.send()) in the Activity Diagram state "Deliver Order."

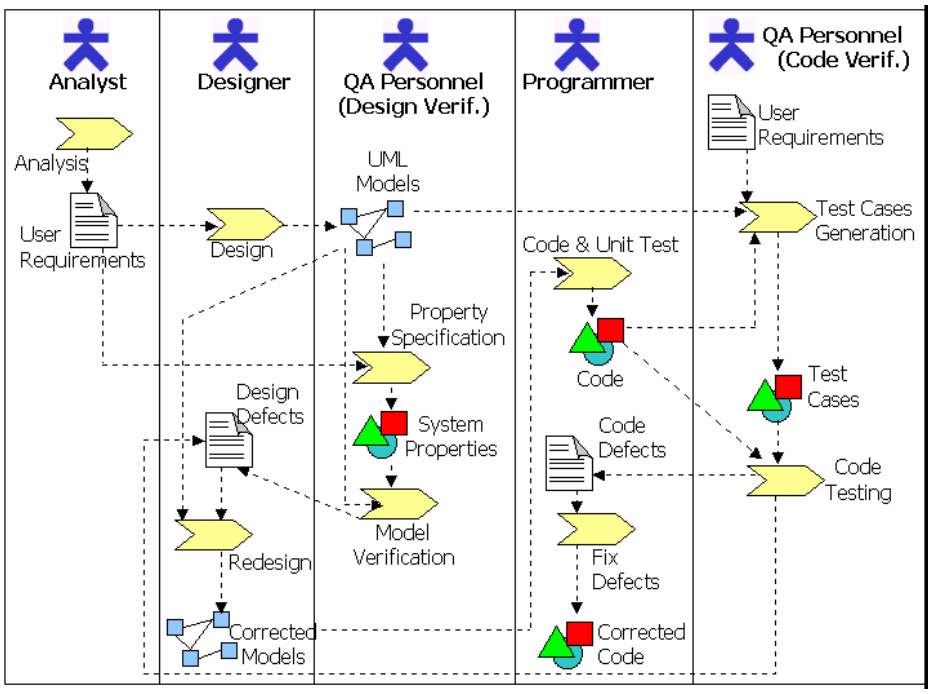
Implications for the Development Process

In a typical development process, the code is the main product, and models are just a documentation of this. Most of the times, these models are inexistent, incomplete, or are not up to date (as maintenance is frequently done only at the level of code). Also frequently, code is the only subject for testing. In this way, sometimes testing occurs too late in the development process. In contrast, the presented approach makes stress in the modeling phase. Models are the main products to be generated during development. These are graphical, higher level, and usually easier to understand than code, therefore the development based on models promises higher levels of productivity and quality. When provided with adequate semantics, models can be verified in order to find defects earlier in the development phase. This contrasts with the usual approach based on static inspection of documents and (late) testing of code. In addition, models are easier to maintain than code, and if code is automatically generated from models, then both can be easily kept up to date.

Thus, our proposal based on model verification applies to the analysis and design phases, and in principle could be used with any development process. It must be remembered that the cost to correct an error exponentially increases as the project development advances. In this way, our approach allows an early identification of analysis and design errors.

Figure 12 shows a very simplified model of the development process and how our approach integrates in it. We have used an Activity Diagram from the OMG standard Software Process Engineering Metamodel (SPEM 1.0 Specification, 2002). For simplicity, we have omitted some activities (like reviews and integration testing), and we have not explicitly included the relationships between activities, but only the products they consume and produce. Note however that iterations are possible and common between some of these activities. In the process model, from the user requirements and the initial design, some properties or conditions that the system should verify are specified (activity "*Property Specification*"). These can be, for example, safety conditions in critical systems or consistency conditions in database systems. These conditions should be translated into some notation (Computational Tree Logic in our example) for their verification. A key issue here is the possibility of automatic or assisted translations of these conditions. Once the conditions are translated, they can be verified on some of the defined models

Figure 12. Integrating our approach in the development process



("*Model Verification*" activity). For this purpose models have to be translated, but this can be transparent to the developer. Note how there is an analogy here between verification of models and verification of code ("*Code Testing*" activity). But while we verify (by proving the property on all computations) properties on models, code should be verified by deriving test cases. These select the best computation paths to be tested. In model verification, however, verifying all paths of computation can be done due to the higher level of abstraction of models.

The work we have presented is an example of the use of formal methods. These are attempts to use mathematics in the development of a software application, in order to reduce the number of errors of the resulting system (Berry, 2002). Although they offer significant benefits in terms of improved quality, their use in industry nowadays is usually reduced to some safety-critical systems. They are not broadly used due to several reasons, among them their high cost (which only pays if the cost of delivered errors is prohibitive) and the need of expert personnel in a certain formal method. This expert knowledge is seldom found among the average software engineers.

As we have seen, we propose hiding the verification process by letting the developers specify the system in a well-known modeling language (UML) and then automatically translating these models into a semantic domain for further analysis. In this way, we free developers with the need to learn new notations or methodologies in order to use a formal method. The use of common modeling notation, together with appropriate tool support for the verification, can sensibly lower the cost of introducing formal methods, as well as reduce the learning curve for software developers. It is our belief that the development of adequate theoretical results, combined with tool support, can make the use of formal techniques very attractive, even for regular software projects.

Our approach also makes stress in code generation from high-level models. In general, we can distinguish three different scenarios for code generation. In the first one, code generation is used for rapid prototyping, as a means to work with users during the analysis phase. Once (part of) the application requirements are established, the generated prototypes are discarded and the application is developed probably using more efficient languages.

In the second scenario, a generative approach can be used in the context of product lines (Pohjonen & Tolvanen, 2002). The idea here is to customize by means of high-level models the application to be generated. The generated code is then combined with a common, kernel code, which results in the final application. Note how this approach is possible for well-understood, domain-specific applications, where domain abstractions and their mapping into code can be adequately identified. An example of this is the approach of AToM^3, where visual languages are defined by means of metamodels and the generated code is combined with the AToM^3 kernel.

In the third scenario (the most powerful approach), the full application code is generated from the models and is not restricted to domain-specific applications. It is promoted by the MDA and benefits from executable models (Raistrick et al., 2004). With the xUML approach one can execute a reduced set of UML models by means of the action semantics. Note how the three scenarios are benefited from the possibility of performing verifications at the level of models before code generation or implementation. In particular, the approach presented in this chapter is complementary to the xUML approach. While in xUML it is possible to test models by simulation (that is, some paths in the execution are tried), in our approach we can formally verify certain properties in all computation paths.

Related Work

The presented technologies have a precedent in the already classical notion of Fourth Generation Languages (4GL) (Martin, 1985), which were also known as "report generation languages." These languages were designed for specific application domains (typically related to database queries) and had a syntax close to natural language. A typical example of this kind of languages was SQL.

Another related approach is Generative Programming (GP), which allows one to automatically generate software from a generative domain model (Czarnecki & Eisenecker, 2000). A generative domain model describes a system family by means of a problem space, a solution space, and configuration knowledge. The problem space contains domain-specific concepts and features; the solution space defines the target model elements that can be generated and all possible variations. The configuration knowledge specifies illegal feature combinations, default settings and dependencies, construction rules, and optimization rules. GP introduces generators as the mechanisms for producing the target. In our approach we model these generators as graph transformation rules.

There are other examples of verification of UML models based on transformation into semantic domains. For example, in López-Grao, Merseguer, and Campos (2004), Stochastic Petri nets are used to evaluate system performance, specified with the UML profile for schedulability. In the approach of Heckel et al. (2002), CSP was chosen as the semantic domain for verification. Nonetheless, to our knowledge, our approach is unique in the sense that it combines metamodeling (for the definition of the visual languages), graph transformation (for specification of any model manipulation), and model checking for the analysis of combinations of several UML models.

Conclusion

In this chapter, we have presented an approach for model-based development based on metamodels and graph transformation. Models are the main assets during development. These can be transformed for verification, optimization, or code generation. The approach is in the line of the MDA sponsored by the OMG, and has the potential to result in higher levels of quality (properties are verified in the models) and productivity (code is automatically generated).

Transferring these concepts for their wide use in industry is only possible with adequate automation. In this direction, we are working in the transparent verification of the models. In this way, models are transformed to the most adequate target formalism (one with appropriate analysis methods) and results are translated back and given to the software engineer in the context of the source language. In this way, the engineer does not need to be familiar with the target notation in order to perform the verification.

Acknowledgments

Juan de Lara's work has been partially sponsored by the Spanish Ministry of Science and Technology (TIC2002-01948). Hans Vangheluwe gratefully acknowledges partial support by a National Sciences and Engineering Research Council of Canada (NSERC) Individual Research Grant.

References

Aho, A.V., Sethi, R., & Ullman, J.D. (1986). *Compilers, principles, techniques and tools*. Reading, MA: Addison-Wesley.

Atkinson, C., & Kühne, T. (2002). Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation, 12*(4), 290-321.

Berry, D.M. (2002). Formal methods: The very idea. Some thoughts about why they work when they work. *Science of Computer Programming, 42,* 11-27.

Brooks, F.P. (1995). *The mythical man month*. Reading, MA: Addison-Wesley.

Clarke, E.M., Grumberg, O., & Peled, D.A. (1999). *Model checking*. Boston: MIT Press.

Czarnecki, K., & Eisenecker, U. (2000). *Generative programming: Methods, tools, and applications*. Reading, MA: Addison-Wesley.

de Lara, J., & Vangheluwe, H. (2002). AToM^3: A tool for multi-formalism modeling and metamodeling. *Proceedings of ETAPS/FASE'02* (pp.

174-188). Berlin: Springer-Verlag (LNCS 2306). See also the AToM^3 homepage at *http://atom3.cs.mcgill.ca*

de Lara, J., & Taentzer, G. (2004). Automated model transformation and its validation with AToM^3 and AGG. *Proceedings of Diagrams 2004* (pp. 182-198). Berlin: Springer-Verlag (LNAI 2980).

de Lara, J., & Vangheluwe, H. (2002). Computer-aided multi-paradigm modeling to process Petri nets and Statecharts. *Proceedings of ICGT'2002* (pp. 239-253). Berlin: Springer-Verlag (LNCS 2505).

Guerra, E., & de Lara, J. (2003). A framework for the verification of UML models. Examples using Petri nets. *Proceedings of Jornadas de Ingeniería del Software y Bases de Datos* (JISBD'03) (pp. 325-334). Alicante.

Heckel, R., Küster, J., & Taentzer, G. (2002). Towards the automatic translation of UML models into semantic domains. *Proceedings of AGT'02/ETAPS'02* (pp. 12-22).

Lédczi, A., Bakay, A., Maróі, M., Vögyesi, P., Nordstrom, G., Sprinkle, J., & Karsai, G. (2001). Composing domain-specific design environments. *IEEE Computer,* (November), 44-51. See also the GME homepage at *http://www.isis.vanderbilt.edu/Projects/gme/default.html*

López-Grao, J.P., Merseguer, J., & Campos, J. (2004). From UML Activity Diagrams to Stochastic Petri nets: Application to software performance engineering. *Proceedings of the 4th ACM International Workshop on Software and Performance* (pp. 25-36).

Martin, J. (1985). *Fourth-generation languages. Volume I: Principles*. Upper Saddle River, NJ: Prentice-Hall.

MDA. Homepage at *http://www.omg.org/mda/*

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE, 77*(4), 541-579.

Pohjonen, R., & Tolvanen, J.-P. (2002). Automated production of family members: Lessons learned. *Proceedings of the 2nd International Workshop on Product Line Engineering—The Early Steps: Planning, Modeling, and Managing* (PLEES'02) (pp. 49-57). See also the MetaEdit+ homepage at *http://www.metacase.com*

Raistrick, C., Francis, P., Wright, J., Carter, C., & Wilkie, I. (2004). *Model-Driven Architecture with Executable UML*. Cambridge, UK: Cambridge University Press.

Rozenberg, G. (Ed.). (1997). *Handbook of graph grammars and computing by graph transformation* (vol. 1). World Scientific.

SPEM (Software Process Engineering Metamodel) 1.0 Specification. (2002, November). Retrieved from *http://www.omg.org/UML*

Unified Modeling Language (UML) 1.5 Specification. (2003, March). Retrieved from *http://www.omg.org/UML*

Chapter XIII

Agile Project Controlling and Effort Estimation

Stefan Roock
it-agile GmbH, Germany

Henning Wolf
it-agile GmbH, Germany

Abstract

Project controlling was not in the focus of agile methods like eXtreme Programming (XP, cf. Beck, 1999) for a long time. Since agile methods are grass rooted, they derive from practitioners and focus on their needs. This chapter shows how to integrate lightweight mechanisms for project controlling into agile methods. The main idea is to combine (incomplete) hierarchical decomposition of systems with abstract measurements. The presented techniques address management needs without building barriers for developers.

Introduction

The main question of project controlling is: "Are we at functionality in time on budget?" The first step is to measure the current state of the project. From this base it is possible to create a prognosis of the project's progress in the future and to answer the question above.

Many approaches to project controlling (all of the non-agile) assume that a complete and stable requirements specification is available up front. Every time a requirement is implemented, the project leader can close it. Tracking closed vs. open requirements allows:

- computing the velocity of the project,
- measuring the progress of implementation, and
- creating a prognosis for the deadline.

Agile methods (for an overview, see Fowler, 2003) are in contrast to conventional project controlling. They claim that in most cases, a complete requirements specification is an illusion. Instead, the requirements specification is inconsistent, incomplete, and continuously changing. Therefore, the requirement specification is perceived as a moving target. This perspective has led to the perception that agile projects are uncontrollable. Since project controlling needs stable data, a *classic* requirements specification is indeed not suitable for controlling agile projects. This chapter proposes hierarchical decomposition of requirements and hierarchical tracking for controlling large agile projects.

Our Background

The authors have worked with agile methods since 1999, with a focus on eXtreme Programming. They work as project managers and consultants for agile projects. Their experience covers projects from *6 person months* (3 developers) to *30 person years* (20 developers). Recently the authors integrated elements from SCRUM, Crystal, and other agile methods into their project management toolbox.

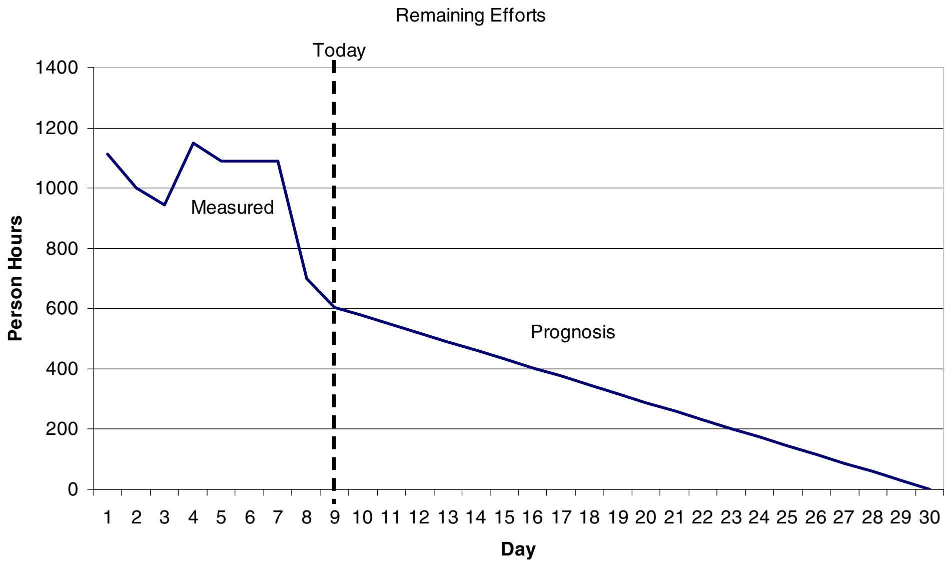
The Agile Way of Tracking

In the agile methodology, development is scheduled for so-called *iterations*. The suggested iteration length depends on the actual methodology used. XP has a iteration duration of one to three weeks, SCRUM iterations last 30 days, and Crystal iterations last three months. Programming tasks (*stories* in XP speak) are assigned to iterations at the iteration planning. Stories assigned to an iteration must not change during the iteration.

On the stable base of the assigned stories, a simple tracking can be done for each single iteration using *story burndown charts* (as shown in Figure 1 for a SCRUM project; cf. Schwaber & Beedle, 2001). The measured project velocity provides the base for the future prognosis.

Burndown charts are a powerful instrument for controlling the development during an iteration. If the prognosis shows that the team is not able to implement all stories, stories are removed from the iteration. If the team works faster than expected, additional stories can be assigned to the iteration.

Figure 1. Example burndown chart



This kind of tracking is effective for the short timeframes of iterations. It does not scale up to large projects since normally it is impossible to know all stories of the whole project up front: the tracking base becomes instable again.

Hierarchical Tracking

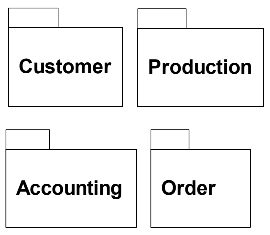
While it is impossible to have a complete set of all stories for medium- to large-size projects, it is always possible to have the *big picture*. Depending on the project size, the big picture varies:

- for small projects: the set of all stories;
- for medium-size projects: the set of all features (a feature is a system functionality that provides business value for the customer; it is a set of stories);
- for large projects: the set of all subsystems (a subsystem has a set of features).

The elements of the big picture are suitable for project tracking as long as they are stable and small in number (should be less than 100 elements).

Figure 2 shows a large project divided into possible subsystems. The subsystems are not detailed up front, but on demand when the developers start to work on a subsystem. Then the subsystem is detailed into features and the features are detailed into stories.

Figure 2. Subsystems of a large project



The subsystems can be linked via the features to the stories, and therefore to the programming tasks of the developers. Every time a story is completed, it can be computed how many stories of the current feature are completed. Knowing the progress of the features allows the project manager to compute the state of the current subsystem, which leads to the progress of the whole project.

Estimation with Effort Points

The efforts needed to implement two stories may vary a lot; the same is true for features and subsystems. Therefore, simply counting stories, features, and subsystems is too rough for project controlling. We use *effort points (EP)* to weight the complexity of items. Stories are weighted with *story effort points (step)*, features with *feature effort points (feep),* and subsystems with *system effort points (syep).*[1]

The estimation by effort points defines the complexity of elements of the same level (subsystem, feature, story) relative to each other. A subsystem with 3 *syep* roughly needs three times the effort of a 1 *syep* subsystem. Our experience shows that it is quite easy to estimate this relative complexity. After estimating the relative complexity of the subsystems of a project, we go on estimating the features of a few subsystems and then the stories of a few features. With simple mathematics we can transform *syep*, *feep,* and *step* into each other. Thus we get an estimation of the whole project with just some of the features and stories.

Figure 3. Estimation with effort points

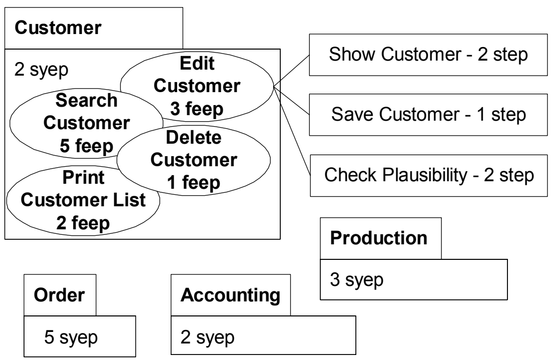


Figure 3 shows an example for an effort estimation with the subsystems *Customer*, *Order*, *Accounting,* and *Production*.

The figure shows that the *3 feep* of the feature *Edit Customer* relate to *5 step* (for the three stories). Therefore *1 feep* computes to *5/3=1.67 step*. In the subsystem Customer *2 syep* relate to *11 feep*, which computes *1 syep* to *11/2=5.5 feep*. In summary *1 syep* computes to *5.5*1.67=9.19 step*.

When we know how much effort is required to implement a *step,* we can compute the efforts for features, subsystems, and the whole project. Assuming *15 person hours (peh)* for a *step* in the example above, we compute that we need *15 peh * 1.67 = 25 peh* per *feep* and *25 peh * 9.19 = 230 peh* per *syep*. Since the whole project has *12 syep* in total, the complete project effort is *230 peh * 12 = 2,760 peh*.

The effort needed to implement a *step* may vary from developer to developer. The basis for the effort calculation in the given example are 3 stories with 5 *step* for just one feature of one subsystem. Assume that we have just finished *75 peh* of *2,760 peh* for the whole project, that represents just 2% of the system effort! If we assume *20* instead of *15 peh* for a *step*, the total effort would be 33% higher (*3,671* instead of *2,760 peh*). Evidently, a higher percentage of already implemented system functionality increases confidence in our estimation. It is useful to implement stories of different features and even different subsystems in the beginning. Knowledge, conditions, requirements, and used technology may vary a lot between features and subsystems. This has an influence on the effort estimation; for example, some feature requiring some unknown technology leads to a higher estimation for this feature. Always our first estimations are fuzzy, but they give an idea about the dimension of the project. The better our data basis, the sharper our estimation becomes.

Putting It All Together

New XP/agile projects start with an initial *exploration phase* in which the ground for the project is built. The team warms up, technology is explored, and the efforts are estimated. For the effort estimation, the big picture of the system functionality is required. In larger projects the big picture will contain a list of the subsystems. We then identify a few subsystems which we investigate further to sketch their features. Few of the features are then broken down into estimateable stories.

During the exploration phase we implement some of the stories to compute the initial velocity of the team (*peh* per *step*). Based on the computation scheme described above, we can roughly estimate the effort for the whole project.

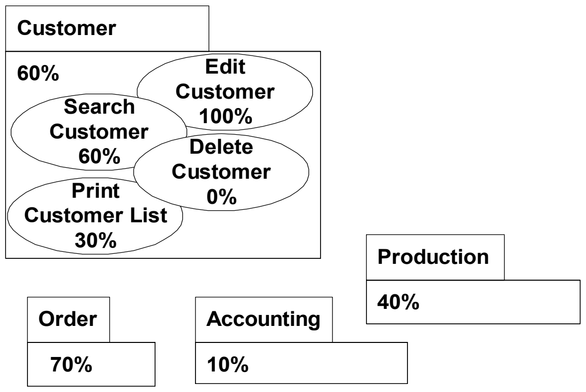
The selection of subsystems, features, and stories that are analyzed in detail reflects their importance for generating business value. That leads to more accurate effort estimations for the important system functionality and reduced accuracy for the "nice to have" features.

In parallel we choose at least a small, a mid-size, and a large subsystem (measured in *syep*) to define all features for them. With the resulting features we also select small, mid-size, and large features (measured in *feep*) to define their stories. Then we implement small, mid-size, and large stories (measured in *step*).

Implementing subsystems and features sequentially would conflict with the idea of short releases. For a usable release one would need features of several subsystems. This is similar for the features—in the beginning the users only need a part of each feature. Therefore we count fully *completed* stories and compute the completion degree (percentage) of features and subsystems (see Figure 4).

SCRUM-like diagrams show us the progress of the project on the different levels (subsystems, features, stories). *Iteration completion diagrams* simply count open *step* every day and show progress for the current iteration (consisting of a number of stories). *Release completion diagrams* show the progress for the current release. Depending on the length of the release, we

Figure 4. Project completeness in agile projects



count open *step* or compute open *feep*. We assume that the stories or features for a release are known. Release completion diagrams are usually updated every week. *Project completion diagrams* show the progress on the level of subsystems and are computed every week or every few weeks.

In most cases all project participants, including the customer and all developers, have access to the diagrams. This is especially necessary in situations where the tracking data suggest adapting the plan while skipping some of the stories or features. But in some cases when our customers insist on making fixed-price projects, we only use the information internally.

The approach presented in this chapter has its limitations. The following prerequisites are essential:

- A rather complete and stable list of subsystems for the project is needed.
- The productivity of the team must be measured or estimated. If a team cannot measure its productivity, the project is still in the exploration phase.

Experiences

The described concepts must be adapted to the project situation at hand. We accumulated experience with the presented techniques:

- The initial estimation is rough. Estimations get better the more we know about the project and the more stories and features we implement.
- Most business sectors have well-established concepts of organizing large systems into subsystems—often oriented to departments. While the relationships between legacy systems often are tangled, the division into subsystems proved to be stable. Therefore it is often possible to gain a complete list of all subsystems of the project in a short period of time.
- Business analysts and experienced developers have a good understanding of the complexity of the subsystems. Therefore estimating the subsystems with system effort points becomes quite easy.
- The levels used for hierarchical decomposition must be adapted to the project situation. For a lot of projects, stories and features are sufficient and the subsystem concept is not necessary.

- Some project trackers prefer to track remaining efforts in person days directly instead of effort points. This is the original SCRUM way of tracking, but it leads to instability when the productivity of the team varies. Estimations based on effort points are not influenced by varying productivity.

Experiences from Project Beaker

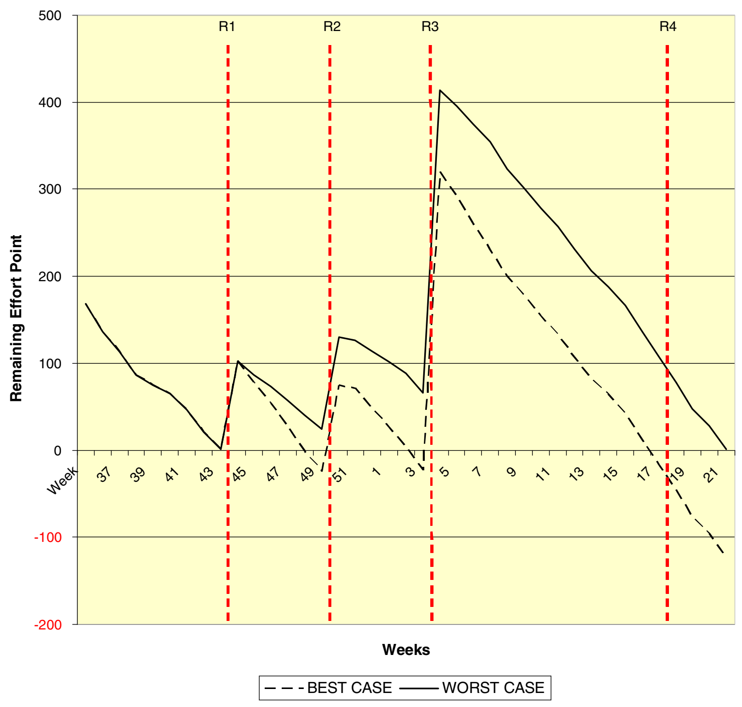
We joined Project Beaker when it had been running several years. The main problem with the project was its unknown state. Once again an important deadline was exceeded.

The project controlling instruments presented in this chapter were useful, even though the project was not agile at all, but had a waterfall-like style. We discovered three types of requirements: change requests, use cases, and subsystems. We interpreted change requests as stories and use cases as features. Then we estimated the efforts with effort points for the three levels and calculated the remaining effort of the project in effort points.

The previous productivity could be analyzed because the project was active for quite a long time. We computed the needed hours per effort point by estimating a representative set of completed change requests and use cases (result: 11 person hours per effort point). We created SCRUM-like completion diagrams based on this analysis of the previous productivity. These diagrams enabled effective communication while programmers and management had severe communication problems. The management saw a visualization of the project state for the first time during the project. This made possible discussions between management and programmers about the future directions of the project.

Several actions were proposed to get the project back on the timeline based on the discussions. Introducing new programmers into the team was controversial: the project team suspected that it would take a very long time for new programmers to become acquainted with the system, while management assumed a linear growth in productivity. In this situation the completion diagrams proved useful. We used a worst- and a best-case scenario to assess the effects of introducing new programmers to the team. We created new completion diagrams based on these worst- and best-case effects of introducing new programmers. The new completion diagram is shown in Figure 5 with releases R1 to R4. The remaining efforts increase after every release since each release has its set of stories.

Figure 5. Completion diagram for Project Beaker



The gap between best and worst case increases with time. It became clear that there was a *chance* to meet all deadlines. It also became clear that the risk was high of failing to meet some or all of the deadlines. It was possible to check the assumptions: We included the measured productivity into the diagram and compared it with the best- and worst-case productivity. (Our assumptions were right: The real productivity was in between best and worse case, and it was closer to the best case.)

Project Bear Experience

Currently we stand at the beginning of Phase 2 of two phases and have just fixed the contract. Phase 1 produced a lot of prototypes and was a long exploration phase. Phase 2 is a variable-scope, fixed-price project. We made requirements that are a “feature size” part of the contract. The customer may change requirements during development as long as the sum of feature points is not increased.

Our estimation basis was quite stable due to the long exploration phase. The project produces four large systems for four different organizational units. The effort to finish a feature is different for each organizational unit, which had a strong impact on our project estimation.

We started out with an initial rough estimation based on the idea of hierarchical effort points described above. To end up with a dimension of 4,000 person days for the whole project, it only took a few days of discussions with customers. Unspecified circumstances led to an effort reduction to 2,800 person days. In discussions between software developers and the customer, the feature lists were reworked to give full particulars and better estimations.

To track the project progress, the hierarchical tracking is now to be installed. To complete the iteration and release planning, hierarchical structures were also necessary so that reasonable releases would be delivered to the customer.

We are not really certain if our estimation of extra effort for producing productive software instead of prototypes is appropriate. But we gained some experiences with two smaller systems we delivered in Phase 1 which led to noticeable additions to the effort estimation.

Conclusion

Our approach combines well-known practices for effort estimation and project controlling (tracking). Abstract measurement (with effort points) is used on different levels: subsystems, features, and stories. That provides a lightweight method for estimating even larger systems.

Future research could focus on empirical data of the accuracy of the estimations, as well as heuristics for the impact of typical context factors (e.g., team size, reachability of customers, unknown technologies).

References

Albrecht, A.J. (1979). Measuring application development productivity. *GUIDE/SHARE: Proceedings of the IBM Applications Development Symposium.*

Beck, K. (1999). *Extreme programming explained.*

Fowler, M. (2003). The new methodology. Retrieved August 11, 2004, from *http://www.martinfowler.com/articles/newMethodology.html*

Schwaber, K., & Beedle, M. (2001). *Agile software development with SCRUM.*

Endnote

[1] Using abstract measures is well known from Function Point Analysis (cf. Albrecht, 1979). Instead of counting fields and dialogues, we estimate effort on different levels relative to each other.

Chapter XIV

Improving OO Design Process Using Rules, Patterns and Refactoring

Javier Garzás
mCentric, University Rey Juan Carlos, Spain

Mario Piattini
University of Castilla-La Mancha, Spain

Abstract

In recent years different areas of knowledge related to the construction of object-oriented (OO) designs such as principles, heuristics, patterns, and refactoring techniques have been consolidated, but there is a lot of work still to be done in order to systematize and offer this knowledge to OO designers in such a way that it can be easily used in practical cases. In order to clarify this, we have defined an ontology of OO Micro Architectural Design Knowledge and the foundations of an OO design method based in the knowledge.

OO Micro Architectural Design Knowledge

Many authors (Shaw, 1990; McConnell, 2003) have commented on the need for defined chunks of knowledge in the software engineering field. In this regard, the software engineering community has advanced greatly in recent years, and we currently have much accumulated knowledge: standards, methodologies, methods, metrics, techniques, languages, patterns, processes, concepts, and so forth. Nevertheless, the field of software engineering is still beset by a lack of structured and classified chunks of knowledge, and not all knowledge is transmitted, accessible, or studied in the same way.

One example of this lack of structured and classified knowledge is the Object-Oriented (OO) Micro Architectural Design. Object-oriented knowledge is popularized in different forms—principles, heuristics, patterns, refactoring, lessons learned, defects, best practices, and so forth—but the difference between these concepts is generally unclear, and moreover, not all of them have received the same amount of attention or have reached the same degree of maturity.

In this sense, we find the OO design principles; in this field, there are several important contributions, such as Meyer (1988), Helm, Johnson, and Vlissides (1995), or Martin (1996) (Table 2 shows examples).Regarding OO design heuristics, the main works to which we can refer are Riel (1996) and Booch (1996) (Table 3 shows examples). On the other hand, bad smells and refactorings are rapidly gaining acceptance, thanks to Fowler (2000) and Beck and Fowler's (2000) work. Finally, patterns are the elements that have undergone the greatest evolution; proof of this is the numerous publications on

Table 1. Examples of OO principles

Dependency Inversion Principle (DIP): Depend upon abstractions. Do not depend upon specifications.	**Interface Segregation Principle (ISP):** Many client-specific interfaces are better than one general-purpose interface.
Do not Concrete Super class Principle (DCSP): Avoid maintaining concrete super classes.	**Interface Design Principle (IDP):** "Program" an interface, not an implementation.

Table 2. Examples of OO heuristics

All data should be hidden within its class.	Do not change the state of an object without going through its public interface.
Minimize the number of messages in the protocol of a class.	Eliminate irrelevant classes from your design.
Eliminate classes that are outside the system.	A class must know what it contains, but it should never know who contains it.

this topic (Coad, 1992; Gamma et al., 1995; Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996; Rising 1998); moreover, a strong community-centered environment to the patterns has consolidated with the years (examples of this are PLOP conferences), discussing and formalizing patterns.

Nevertheless, the problem confronting the designer is how to articulate all this knowledge and to apply it in the Object-Oriented Design (OOD). In fact, even advanced chunks of knowledge like patterns have this problem, and this situation could give rise to incorrect applications of the patterns. At the present time, using exclusive patterns is not sufficient to guide a design in a formal way; again, necessary is the designer's experience to avoid the overload, the non-application, or the wrong use of patterns. In fact, according to our own observation in developers Wendorff (2001) and Schmidt (1995), when patterns are used, several types of problems occur: difficult application, difficult learning, temptation to recast everything as a pattern, pattern overload, and so forth.

Moreover, a solid and structured knowledge does not exist on OO design principles, heuristics, or bad smells, and these are still immature chunks being used in an isolated way or even being ignored. The work of discovering, in order to formalize and to classify these, is still not finished, and it is an important investigation line. On the other hand, a well-defined, classified, and homogeneous catalog does not exist of principles, heuristics, and so forth, where their applicability is detailed.

The problem is how to articulate and apply all this knowledge. The large number of patterns, principles, heuristics, and so forth that have been discovered so far need to be organized. Many of them are competitors; we need to experiment and find which are best to use. According to our own observation, several types of deficiencies happen in catalogs: Search and Complex Application, High Dependence of the Programming Language, and Comparatives. Analyzing existing knowledge, making tools that use this, or determining the effectiveness of this could all be good research topics.

A large part of the previous problems comes from a faulty classification, cataloguing, and formalization of the Object-Oriented Design Knowledge (OODK), almost only centered in the pattern concept. More knowledge—in fact, still knowing the patterns and applying them in a correct way—these allow many details to pass to achieve a good design. Other solutions exist for certain problems that have already repeated for years and that are not patterns; these other solutions (principles, heuristics, etc.) are known by the most experienced designers and are based on experience.

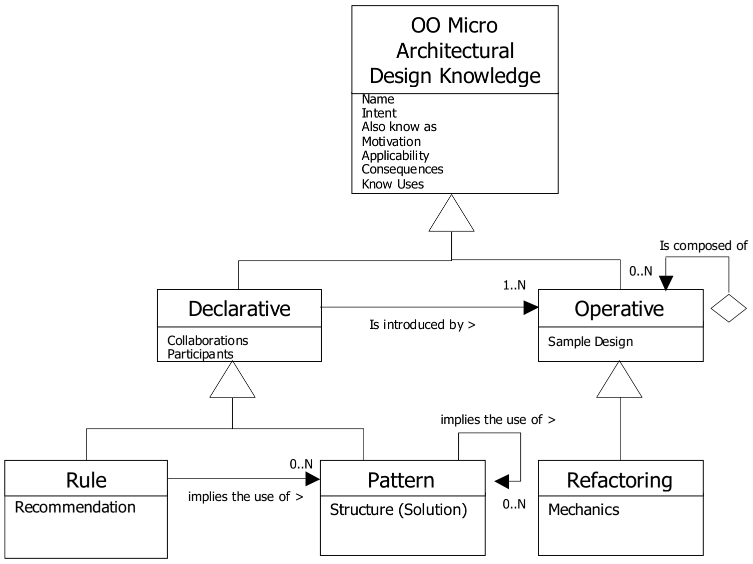
Ontology of OO Design Knowledge

An ontology describes a domain in a generic way and provides an agreed understanding of it. It has a number of advantages, such as structuring and unifying accumulated essential knowledge, benefits for communication, teaching concepts, sharing, and resolving terminological incompatibilities. It would therefore be beneficial to define an ontology for the structuring and unifying of OO Micro Architectural Design Knowledge.

We find many chunks related to Object-Oriented Micro Architectural Design Knowledge; in order to clarify this, we have defined an ontology of OO Micro Architectural Design Knowledge (see Figure 1, where a UML class diagram is used to express the ontology). In this ontology we have grouped the chunks of knowledge into two groups:

- **Declarative Knowledge:** Concepts describing what to do with a problem: Heuristics, Patterns, Bad Smells, Best Practices, and so forth.
- **Operative Knowledge:** Concepts describing operations or processes for carrying out changes in software including concepts such as design refactorings.

Figure 1. OO Micro-Architectural Design Knowledge ontology



We have observed that principles, heuristics, bad smells, and so forth have the same common structure, as they all have the structure and form of a Rule—they posit a condition and offer a recommendation. It should be stressed that the "recommendation" is not a solution like that of the pattern. Patterns are more formalized than rules: their descriptions are broader, they propose solutions to problems, while rules are recommendations to fulfill. Unlike patterns, rules are greatly based on using natural language, which can be more ambiguous (Pescio, 1997). With all, we can distinguish the following entities of declarative knowledge (see Figure 1): rules and patterns. Regarding attribute entities, we have based these on the attributes used by Gamma et al. (1995) to describe a design pattern. Many of these attributes are common to all knowledge elements, and these common attributes are located in the top entity (see Figure 1). However, other attributes are specific. The Structure attribute is a synonym for a solution in a pattern, while we have created the Recommendation attribute for rules that would be close to the solution of the pattern (Pescio, 1997), and the Mechanics attribute for refactorings, our choice of name being taken from Fowler's (2000) refactoring catalog. The attributes Participants (the classes and/or objects participating in the design pattern and their responsibilities) and Collaborations (how the participants carry out their responsibilities together) concern declarative knowledge. The Sample Design attribute concerns operative knowledge.

The Implementation attribute is substituted for Mechanics (this is in Refactoring, as we are dealing with design refactoring, not code refactoring). The Related Patterns attribute has been generalized and appears in each of the relationships between entities.

At this point, we shall now concentrate on the relationships between entities (see Figure 1):

- "To apply a Pattern implies the use of another Pattern." This relationship is obvious, since it has been featured in pattern catalogs for some time. The cardinality is from 0 to n (examples of this we can see in Gamma et al., 1995).
- "To apply a Rule implies the use of a Pattern." Often, when we introduce a rule, we obtain a new design, which needs a pattern. One example of this situation is the application of the "Dependency Inversion" (Martin, 1996), which introduces an abstract class or an interface, which in turn necessitates a creational pattern (Gamma et al., 1995) to create instances and associate objects in the new situation. This does not always happen (cardinality 0 to n), not all the rules imply the introduction of a pattern (for example the "Long Method" rule by Fowler, 2000).
- "The Declarative knowledge is introduced by Operative knowledge." All declarative knowledge (rules and patterns) is introduced in the design by an element of Operative knowledge (a refactoring), cardinality from 1 to n. This is quite obvious since it does not make sense for an element of Declarative knowledge to exist if it cannot be introduced:
  - The relationship between patterns and refactorings can be observed reading some of the refactoring catalogs that concentrate on the design level (see Fowler, 2000). Gamma et al. (1995) state that "design patterns provide the refactorings with objectives," that there is a natural relationship between patterns and refactorings, where the patterns can be the objective and the refactorings the way of achieving them; in fact, as Fowler (2000) says, there should be catalogs of refactorings which contemplate all design patterns. In this way, refactorings, such as "Replace Type Code with State/Strategy," concentrate on introducing patterns within a system.
  - The relationship between rules and refactorings has not been studied as much as that between patterns and refactorings. Generally, we observe

that rules are introduced in the design, just like patterns, by means of the refactorings. And in the light of what has been said previously, it becomes clearer how refactorings store knowledge about how to introduce elements in designs in a controlled way. Continuing with the example of the "Dependency Inversion" rule, we see that in order to resolve the violation of this rule, we insert an abstract entity into the design, which would be carried out with the refactorings.

- "An element of Operative Knowledge is composed of others." Refactoring catalogs such as Fowler's (2000) shows several examples, where, for example, the Refactoring "Extract Method" is not composed, but is used by others (cardinality 0 to n).

Foundations of an OOD Method Based in the Knowledge

"Using design artifacts (i.e., patterns) increments design quality" is a popular sentence; but what does "design quality" mean? Using design patterns increments design quality, we put together a common terminology, we have proven solutions, and so forth. In this sense, there are many works about metrics and design quality (e.g., Genero, Piattini, & Calero, 2000; Brito e Abreu & Carapuça, 1994; Briand, Morasca, & Basili, 1999; Henderson-Sellers, 1996). Since design quality can be measured by quality metrics, the use of design patterns should lead to better measurements. However, many common OOD metrics indicate lower quality if design patterns are used. In this sense, Reibing (2001) comments that if we have two similar designs A and B for the same problem, B using design patterns and A not using design patterns, B should have a higher quality than A. However, if we apply "classic" object-oriented design metrics to both designs, the metrics tell us that design A is better—mostly because it has less classes, operations, inheritance, associations, and so forth. Who is wrong? Metrics or the pattern community? Do we have the wrong quality metrics for object-oriented design? Or does using patterns in fact make a design worse, not better? So what is the cause of the contradiction between the supposed quality improvement by design patterns and the measured quality deterioration (Reibing, 2001)?

A lot of times, when we hear "patterns increments design quality," it is the same as "patterns increments the maintenance of design" or "patterns increments the maintenance quality." But again, what does "maintenance quality" mean? According to the ISO/IEC 9126–2001 "Software Product Evaluation—Quality Characteristics and Guidelines for Their Use," standard maintainability is subdivided into Analyzability, Changeability, Stability, Testability, and Compliance. In our case, if we obtain a correct OO design, we will obtain better maintenance. Considering the ISO 9126 standard, three important parameters for quality maintenance of OO Micro Architectural Design exist:

- **Analyzability in OO Micro Architectural Design:** Analyzability allows us to understand the design. This is an essential requisite in order to be able to modify the design in a realistic period of time.
- **Changeability in OO Micro Architectural Design:** Changeability allows a design to be able to change easily, an important requirement at the time of extended functionality into an existing code. In our case, the element that provides changeability is what it is called indirection. Nordberg (2001) comments, "At the heart of many design patterns is an indirection between service provider and service consumer. With objects the indirection is generally via an abstract interface."
- **Stability in OO Micro Architectural Design:** Stability allows us to reduce risk of the unexpected effect of modifications. Many design classes have this objective. For example, many abstract classes avoid code duplication; others increase the cohesion.

Therefore, when a design artifact is introduced, it affects the maintainability quality in a different way:

- Every time a pattern is introduced, at least one indirection appears in the design.
- Every time an indirection is added, it increases the design changeability.
- Every time a designs class that is not an indirection is added, it increases the design stability.
- Every time we add an indirection class or a stability class, the software moves further away from the analysis. Design classes—indirections or

> others (such as notifications, observer classes, etc.)—are not of business logic; upon adding indirections or design classes, the design becomes less semantic, less comprehensible, or less analyzable. Each design class moves the software further from the "real world" or analysis-level view of the problem and deeper into relatively artificial mechanism classes that add overhead to design comprehension.

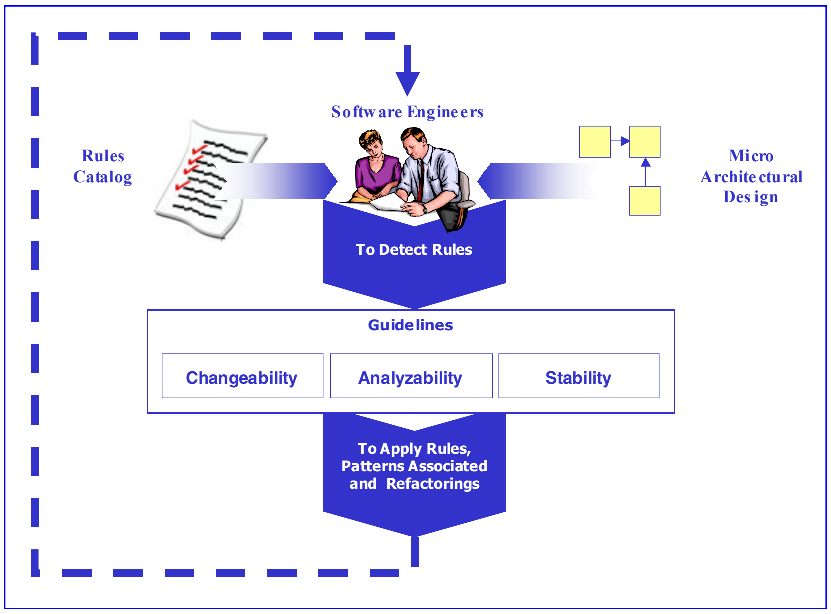
To be specific, at the time of applying patterns to a software design, opposite forces appear; these forces are directly related to maintainability. On the one hand, we have the inconvenience that the solution, once obtained, can be very complex, and this means that the design is less comprehensible, and modifying the design is more difficult (Prechelt, Unger, Tichy, & Bossler, 2000). Thus, to continue with the previous concepts, a curious relation between Changeability, Stability, and Analyzability appears; if we increase the design's Changeability or Stability, then we will decrease the design's Analyzability:

- If a design has many design patterns, this design will have a great amount of Changeability and Stability. This increments the maintenance quality.
- If a design has many design patterns, this design will not have a great amount of Analyzability. This decrements the maintenance quality.

We can see that questions such as "Patterns increments design quality?" or "Patterns increments maintenance quality?" do not have a clear answer. Note that these are important guidelines rather than hard rules or strict criterions, and they are designed to help you improve your design.

With these guidelines and with the Ontology, we can outline a method for improving the quality of Object-Oriented Micro Architectural Design; Figure 2 shows this. This is an iterative method, and each iteration consists of application of rules and patterns, along with associated refactorings. Guidelines about changeability, analyzability, and stability help to maintain and improve the application of design artifacts.

Figure 2. Foundations of a method based in knowledge



Conclusions and Future Projects

The experts have always used proven ideas, although over recent years different areas of knowledge related to the construction of OO designs such as principles, heuristics, patterns, and refactoring techniques have been consolidated. We believe that there is a lot of work still to be done in order to systematize and offer this knowledge to OO designers in such a way that it can be easily used in practical cases. In fact, up until now the different studies that have been published present these elements in a disconnected way that at times makes their application more difficult. This problem occurs when choosing a pattern and incorporating it into an existing model.

Our experience in consulting has demonstrated that there is still a lot of work to be done in order to systematize and offer design knowledge to designers in such a way that it can be easily used in practical cases. In order to facilitate this task, we have developed this ontology, these guidelines, and a basic method.

Acknowledgment

This research is part of the MAS project supported by CICYT (TIC 2003-02737-C02-02).

References

Beck, K., & Fowler M. (2000). Bad smells in code. In M. Fowler (Ed.), *Refactoring improving the design of existing code.* Reading, MA: Addison-Wesley.

Booch, G. (1996). *Managing the object-oriented project*. Reading, MA: Addison-Wesley.

Briand, L., Morasca, S., & Basili, V. (1999). Defining and validating measures for object-based high-level design. *IEEE Transactions on Software Engineering, 25*(5), 722-743.

Brito e Abreu, F., & Carapuça, R. (1994). Object-oriented software engineering: Measuring and controlling the development process. *Proceedings of the 4th International Conference on Software Quality.*

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *A system of patterns: Pattern-oriented software architecture.* Reading, MA: Addison-Wesley.

Coad, P. (1992). Object-oriented patterns. *Communications of the ACM, 35*(9), 152-159.

Fowler, M. (2000). *Refactoring improving the design of existing code.* Reading, MA: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software.* Reading, MA: Addison-Wesley.

Genero, M., Piattini, M., & Calero, C. (2000). Early measures for UML class diagrams. *L´Objet, 6*(4), 489-515.

Henderson-Sellers, B. (1996). *Object-oriented metrics—measures of complexity*. Englewood Cliffs, NJ: Prentice-Hall.

Martin, R.C. (1996, August-December). Engineering notebook. *C++ Report,* (August-December, published in four parts).

McConnell, S. (2003). *Professional software development*. Reading, MA: Addison-Wesley.

Meyer, B. (1988). *Object-oriented software construction*. Englewood Cliffs, NJ: Prentice-Hall.

Nordberg, M.E. (2001). Aspect-oriented indirection—beyond OO design patterns. *Proceedings of OOPSLA 2001, Workshop Beyond Design: Patterns (Mis)Used.* Tampa Bay, FL: EEUU.

Pescio, C. (1997). Principles versus patterns. *IEEE Computer, 30*(9), 130-131.

Prechelt, L., Unger, B., Tichy, W., & Bossler, P. (2000). A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*.

Reibing, R. (2001). The impact of pattern use on design quality. *Proceedings of OOPSLA 2001, Workshop Beyond Design: Patterns (Mis)Used.* Tampa Bay, FL: EEUU.

Riel, A.J. (1996). *Object-oriented design heuristics*. Reading, MA: Addison-Wesley.

Rising, L. (1998). *The patterns handbook: Techniques, strategies, and applications*. Cambridge University Press.

Schmidt, D.C. (1995). Experience using design patterns to develop reusable object-oriented communication software. *Communications of the ACM, 38*(10), 65-74.

Shaw, M. (1990). Prospects for an engineering discipline of software. *IEEE Software, 7*(6), 15-24.

Wendorff, P. (2001). Assessment of design patterns during software reengineering: lessons learned from a large commercial project. *Proceedings of CSMR 2001—European Conference on Software Maintenance and Reengineering* (pp. 77-84).

Chapter XV

The BORM Method: A Third Generation Object-Oriented Methodology

Roger Knott
Loughborough University, UK

Vojtech Merunka
University of Agriculture in Prague, Czech Republic

Jiri Polak
Deloitte & Touche, Prague, Czech Republic

Abstract

BORM (Business Object Relationship Modeling) is an object-oriented system development methodology, which has proven to be very effective in the development of business systems. The effectiveness gained is largely due to a unified and simple method for presenting all aspects of the relevant model. The BORM methodology makes extensive use of business process modeling. This chapter outlines BORM, its tools and methods, and discusses differences from other similar development methodologies.

Introduction

Business Object Relation Modeling (BORM) (Knott, Merunka, & Polak, 2003a, 2003b, 2000; Polak, Merunka, & Carda, 2003) has been in continuous development since 1993 when it was intended as a vehicle to provide seamless support for building object-oriented software systems based on pure object-oriented languages such as Smalltalk and object databases. It has now evolved into a robust system development methodology that has been used successfully to develop a wide range of systems of diverse sizes—in particular:

- to identify business processes in Prague city hospitals as a prerequisite for further cost analysis;
- to model necessary properties of the general agricultural commodities wholesale sector in the Czech Republic;
- for business process reengineering in the electricity supply industry;
- for telecommunication network management in the Czech Republic.

Such systems range through all sizes of software development as can be seen in Table 1.

BORM has proven to be effective and beneficial in the process of describing and subsequently understanding how real business systems evolve. Such knowledge is the key for the success of any business and is especially crucial for those employees who are responsible for business development.

Do We Need Another Object-Oriented Design Methodology?

The first and we think the major problem with existing object-oriented methodologies arises in the initial stages of the system development cycle (Bahrami, 1999; Eriksson & Penker, 2000; Goldberg & Rubin, 1995; Cotterrell & Hughes, 1995; Cantor, 1998; Royce, 1998; Rumbaugh, Blaha, Premerlani, Eddy, & Lorensen, 1991). The initial stage of any object-oriented design methodologies should be concerned with two tasks. The first is the specification

Table 1.

Project	**Number of system functions**	**Number of scenarios**	**Number of process diagrams**	**Number of objects (participants)**	**Average number of states per object**	**Average number of activities per object**
National agrarian chamber (analysis and design of software for fruit market public information system)	4	7	7	6	4	4
Hospital complex (BPR of organization structure)	6	12	12	8	10	12
TV and radio broadcasting company (BPR and company transformation for open market)	4	9	9	14	8	8
Regional electricity distribution company (customer information system analysis)	12	19	19	23	12	12
Regional electricity distribution company (failure handling information system analysis and prototype implementation)	19	31	34	27	13	14
Regional gas distribution company (BPR of all companies)	28	81	97	210	11	12
Regional gas distribution company (BPR of all companies)	23	60	63	120	12	12

of the requirements for the system. The second is the construction of an initial object model, often called an *essential object* or *conceptual model* built out of a set of domain-specific objects known as *essential objects*. Both these tasks should be carried out with the active participation of the stakeholders, in order to ensure that the correct system is being developed. Consequently, any tools or diagrams used at these early stages should be meaningful to the stakeholders, many of who are not 'computer system literate'.

The most common technique for requirements specification in current object-oriented methodologies is use case modeling (Jacobson, 1992). Use case modeling is concerned with the identification of actors, which are external entities interacting with the system. This means that in order to employ use case modeling, it is necessary for developers to already know the system boundary and distinguish between entities, which are internal and external to that boundary. It is our experience that the correct identification of the system boundary is a 'non-trivial' task, which often requires significant understanding of the proposed system, and consequently can only successfully take place at the end of the requirements specification stage.

Use case models are essentially text based; any diagrams employed do not contain any significant information, but only identify the actors involved in each use case. Neither is it an object-oriented process, as the use cases determined could be subsequently developed in any programming paradigm. Moreover, use case modeling is often insufficient by itself to fully support the depths required for initial system specification. Fowler (1999) highlights some deficiencies in the use case approach, suggesting that use case diagrams, if they are to convey all the necessary information, need supplementation by Sequence and Activity diagrams as suggested by Jacobson (1992). These modifications to use case Analysis would result in a number of different diagrams that are used initially to define the interaction between any proposed system and its users. There are many other views on the effectiveness of use cases as a first stage in System Design. Simons and Graham (1999) for example describe a situation where use case modeling obscures the true business logic of a system.

The approach adopted in BORM is based on the fundamental concept of process modeling. This follows from the belief that it is essential, for the deployment of a new system not to view that system in isolation, but to view it in the context of the company's total organizational environment. A new system, when introduced into an organization, will normally totally change the way that the organization operates. In addition, a BORM process model is object-oriented from its conception and is expressed in easy-to-understand graphical notation. From the process model, scenarios are developed. Scenarios were originally developed in Object Behavior Analysis (OBA) (Rubin & Goldberg, 1992) to capture similar information to that presented in use cases. A Scenario however is an *instance* of a User Interaction with the system, whereas a use case is more like a procedural description of a *type* of user interaction. Our experiences on the projects listed above suggest that stakeholders tend to express themselves naturally in terms of scenarios and that the process way of thinking is more natural to business employees. Consequently, stakeholders in the proposed system can more easily understand BORM models and consequently make a greater contribution to the correctness of the system design.

In BORM, initial diagrams are expressed only in problem-domain-specific concepts; any software-orientated concepts are left until later in the modeling process. In Addition, in the early stages BORM uses a single diagram that embodies the same information as the numerous diagrams used by other methodologies. This is an attempt to make it easier for the user to form a complete understanding of the interaction of the various system components.

In BORM concepts and their notation change as the development process proceeds. This is in sharp contrast with UML, which claims to be a universal system, in that the same notation is used for analysis, design, and documenting the implementation. Our reasons for changing notation are based on the observation that this universality of the UML's notation hinders the design process. In this we are in broad agreement with the criticism of this aspect of UML expressed by Simons and Graham (1999).

The second problem that we find with most development methodologies is that, during subsequent stages, they require a number of different diagrams to fully describe the system. Each diagram is used to model an independent aspect of the system. Thus, we have one diagram for the object's static structure and a second for the state changes of particular objects; one diagram showing the message passing between objects and a further diagram to model the activities the system must perform.

The fundamental principle of object-oriented systems is one of encapsulation. This means that all an object's data values are stored in the same place as the functions (methods) that manipulate them. The synergy created by this unification of data and functionality leads to greater understanding of the situation being modeled and to a correctly designed solution being developed.

A diagram is a visual representation of an abstract model, which exists in the brain of the analyst. Developers use diagrams to communicate with customers and other designers. If this model is object-oriented in nature, its representation should reflect it and must not require the viewer to deduce this fact from a number of different diagrams, each of which reveals one particular aspect of the object nature of the model.

Finally, the modeling concepts used in most development methodologies are used throughout the system development cycle. Moreover these notations tend to be specifically designed to represent concepts from object-oriented programming languages at an abstract level.

For example, the models used in OMT (Rumbaugh et al., 1991) or UML (Booch et al., 1998; Rumbaugh, Jacobson, & Booch, 1998) can use quantifiers, link classes, external relations, aggregations, and so forth. While many of these concepts are necessary for software implementation in hybrid object-oriented programming languages such as Java, C++, or C#, they are too 'computer-oriented' to be useful in capturing primary system information. Such diagrams are often 'conceptually rich' and difficult for the customer and other 'non computer' people to fully understand. There is of course no compulsion to use these features, but in our experience, software designers will often use all the facilities provided without any consideration as to their appropriateness. The use of complex concepts too early in the design process often compromises the requirements determined for the system, since users find themselves constricted by the programming nature of the models and consequently are unable to fully articulate their needs. Simons and Graham(1999), speaking of UML, state: "Developers often take the most concrete example of notational element in use and retrofit these interpretations higher up in the analysis process."

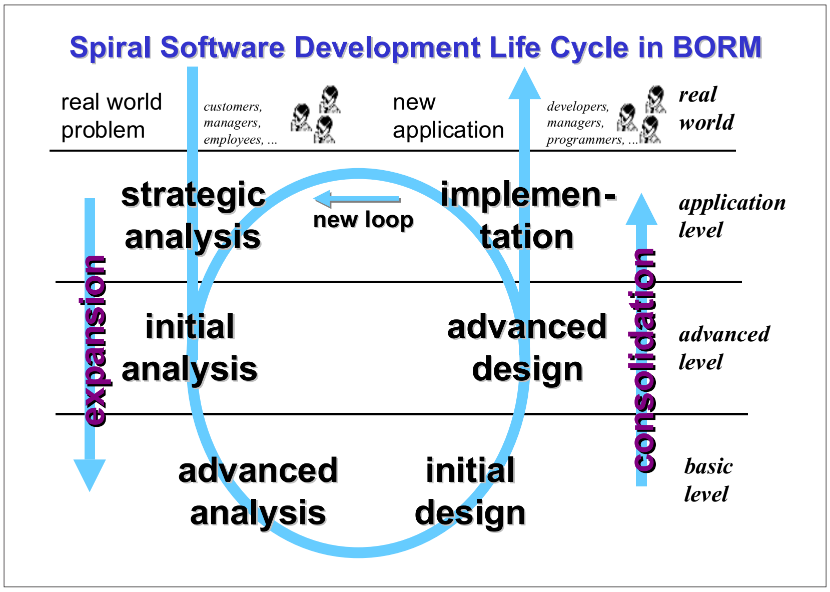
If we compare standard Entity-Relation Diagram (ERD) (Carteret & Vidgen, 1995; Date, 1995) with 'object-class diagram' used in OMT or UML, we find that ERD only uses three basic, but powerful concepts. Object-class diagram, on the other hand, generally uses about 20 different concepts, spread over a number of different levels of abstraction.

In the analysis phase, we need to acquire the best possible knowledge of the problem formulation, and there the implementation details may cause trouble. On the other hand, in the design phase we need to focus on implementing the outputs from the analysis, but we do not need to know some aspects of the reality modeled.

BORM Basics

BORM, like other OOA&D methodologies, is based on the spiral model for the development life cycle (Yourdon, 1995). One loop of the object-oriented spiral model contains stages of *strategic analysis, initial analysis, advanced analysis, initial design, advanced design, implementation,* and *testing* (see Figure 1).

Figure 1. BORM stages



The first three stages are collectively refereed to as the *expansion stages*. Expansion ends with the finalizing of the detailed analysis conceptual model, which fully describes the solution to the problem from the requirements point of view.

The remaining stages are called the *consolidation stages*. They are concerned with the process of developing from 'expanded ideas' to a working application. During these stages the previously completed conceptual model is transformed step-by-step, refined, and finalized into a software design.

In BORM the object-oriented design stage is fluently connected to implementation without any sharp discontinuity, similar to the smooth transition between object-oriented analysis and object-oriented design. As a consequence, we can consider the program coding as the last and the most detailed phase of the design process.

During the progress around the loop, the developer may undertake small excursions (little spirals out from the main spiral) into the implementation of smaller partial applications, used to develop, test, and tune the program incrementally by using previously completed modules.

The behavior of any *prototype* (in BORM we prefer the more accurate name *'deliverable'*) is also interesting. Every finalized application is also a deliverable and may be used as a basis for generating further requirements, which in turn lead to a new development loop.

BORM supports development in pure object-oriented programming environments like Smalltalk, where it is possible to create software applications, which can be changed or updated at runtime.

The main concepts used in initial modeling are

- *objects*, and their *behavior*;
- *association* (data links) between objects; and
- *communication* (reports) between object behaviors.

Every object is viewed as a machine with states and transitions dependent on the behavior of other objects. Each state is defined by its semantic rule over object data associations, and each transition is defined by its behavior, necessary to transform the object from its initial to its terminal state. Consequently BORM objects have the characteristics of Mealy-type automaton (Shlaer & Mellor, 1992).

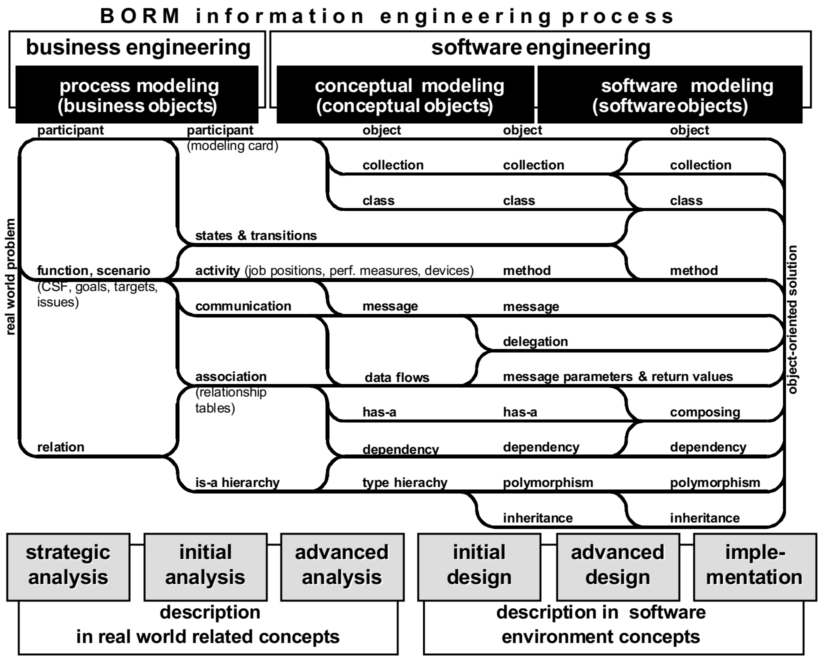
A business object diagram accents the *mutual relationships* (communications and associations) of states and transitions of objects in the modeled system.

In BORM, it is possible for each concept to have some of the following:

1. *A Set of predecessor concepts* from which it could be derived by an appropriate technique, and *a Set of successor concepts*, which could be derived from it by an appropriate technique. For example a conceptual object composition from a business object association.
2. *A validity Range*—The phases (of the development process) where it is appropriate. State–Transition diagrams for example are used extensively in business conceptual modeling poorly supported by current programming language.
3. *A Set of techniques and rules* which guide the step-by-step transformation and the concept revisions between the system development phases. These are the following:
4. Object Behavior Analysis, which is a technique for transforming the initial informal problem description into the first object-oriented representation.

5. Behavioral Constraints, which are a set of determining rules that describe the set of possible transformations of the initial model into more detailed forms, with precisely specified object hierarchies like inheritance, dependency, aggregation, and so forth.
6. Pattern Application, which helps to synthesize the analysis object model by the inclusion of object patterns.
7. *A Set of Structural Transformations* (Class Refactoring, Hierarchies Conversions and Substitutions, solving Legacy problems solving programming environment constraints) which are aimed at the final transformation of the detailed design model into a form acceptable in the implementation environment (programming language, database, user interface, operating system, etc.).

Figure 2. BORM evolution of concepts



An Example: A Car Fleet

In Figure 3, the required system functions are shown as rectangles (equivalent to use cases), while ovals are used to show scenarios. Black, light arrows are used to link scenarios to system functions.

System Scenarios

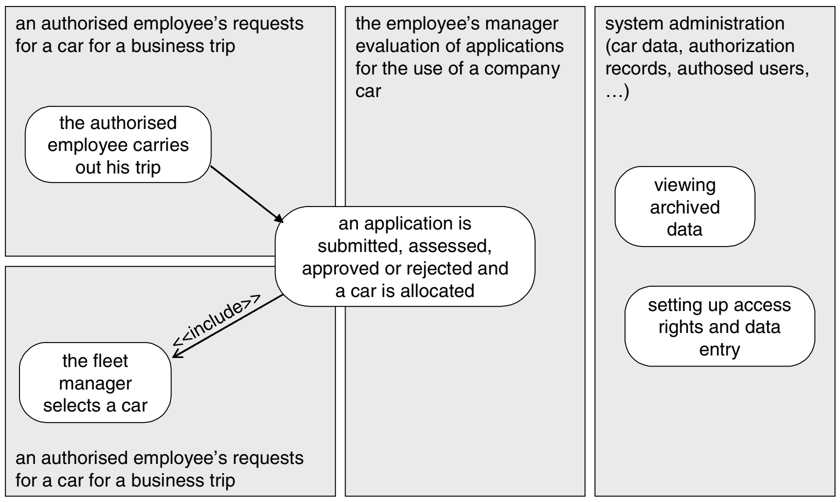
We can describe each scenario by a short textual description, which includes details of its association to functions and participants.

Scenario No. 1—associated with function(s): 1, 3, 2		Continues in scenario No. 2 Uses scenario No. 3	
Initiation	*Action*	*Participants*	*Result*
Employee needs a car for a business trip	**Application submission, application assessment/approval, and car assignment**	Car, Authorized Employee Manager, Fleet Manager	The employee is either assigned a car or must cancel the trip.

Scenario No. 2—associated with function(s): 1, 3		Follows scenario No. 1	
Initiation	*Action*	*Participants*	*Result*
An Authorized employee has a car assigned.	**An Authorized employee makes a business trip.**	Car, Authorized Employee, Fleet Manager	After the completion of the business trip, the car is returned to Fleet Operations. Alternatively, the car was not used and is returned unused.

Scenarios play a similar role for BORM to that played by use cases in those development methodologies supported by UML. However, our diagrams are different to the analogous use case diagrams, as the system functions and scenarios are shown on the same diagram as their associations. In such a diagram, light bold arrows are used to show transition between scenarios.

Figure 3. System functionality


an authorised employee's requests for a car for a business trip
the employee's manager evaluation of applications for the use of a company car
system administration (car data, authorization records, authosed users, …)
the authorised employee carries out his trip
an application is submitted, assessed, approved or rejected and a car is allocated
viewing archived data
setting up access rights and data entry
<<include>>
the fleet manager selects a car
an authorised employee's requests for a car for a business trip

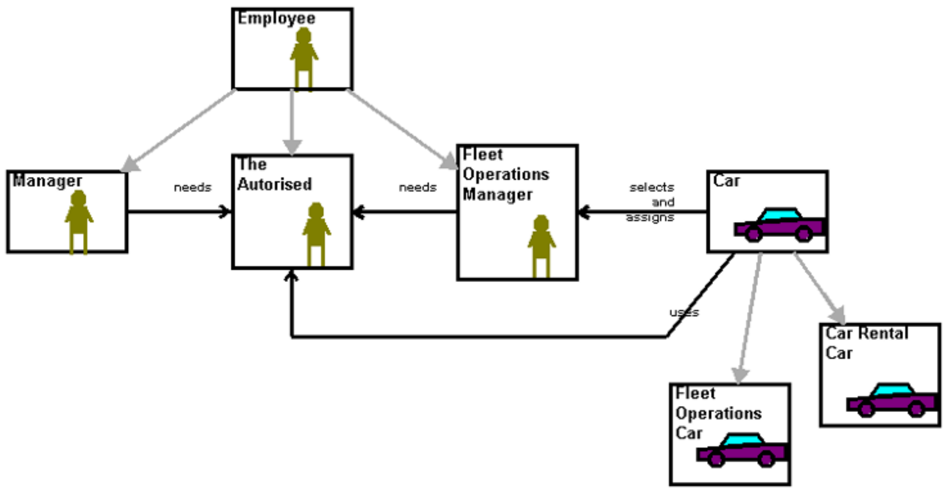
Finally, bold black arrows between scenarios show the existence of a 'includes' relationship. (Our definition of this type of relationship is identical to the <<includes>> relationship in UML 1.3 (Fowler, 1999)).

The construction of the system function diagram takes place in parallel with the development of the business object model. In this latter type of diagram, we do not distinguish whether a participant represents a class, instance, or collection of objects. An example of such a diagram that identifies all processes carried out by the participants is provided in Figure 4. In such a diagram, we consider only the following two kinds of relationships:

- *associations*, which are represented by black arrows; and
- *is-a* hierarchy, represented by gray arrows.

It is important to note that in this phase of development, an is-a hierarchy is not the same as software object inheritance based on a conceptual object type hierarchy. The conditions for two business objects to be in an is-a hierarchy are based on their membership of domains and sub-domains, where a domain is a set of real-world objects.

Figure 4. The Business Object Model—the initial BORM object relationship model



We next develop the process diagram in Figure 5. This diagram expresses all the possible process interactions between participants in the system.

This diagram shows the same associations between the business objects as the previous one, but adds time and behavioral dimensions. Thus we show the states, activities transitions, and operations for the business objects. This is a very powerful diagram. It conveys information that in the UML would require at least two diagrams (State and sequence diagrams). Yet despite conveying large amounts of information, the BORM group has found that it is clearly understood by stakeholders in the system development.

The next diagram marks the transition to the advanced analysis phase of development. Here we develop the conceptual object relation diagram. We are concerned here with conceptual objects and the various associations that exist between them. Gray arrows are used to show type hierarchies, where polymorphism is the determining criteria. Note that this is different than hierarchies in the previous diagrams. In this diagram we identify object classes, which are denoted by rectangles, and collection of classes that are denoted by double-border hexagons. Both these are derived from entities in the previous diagram.

Note the conceptual class 'Car' which is associated with two different multi-objects—, company cars and rental cars. For this association we do not need

Figure 5. Process model: Details of object behavior

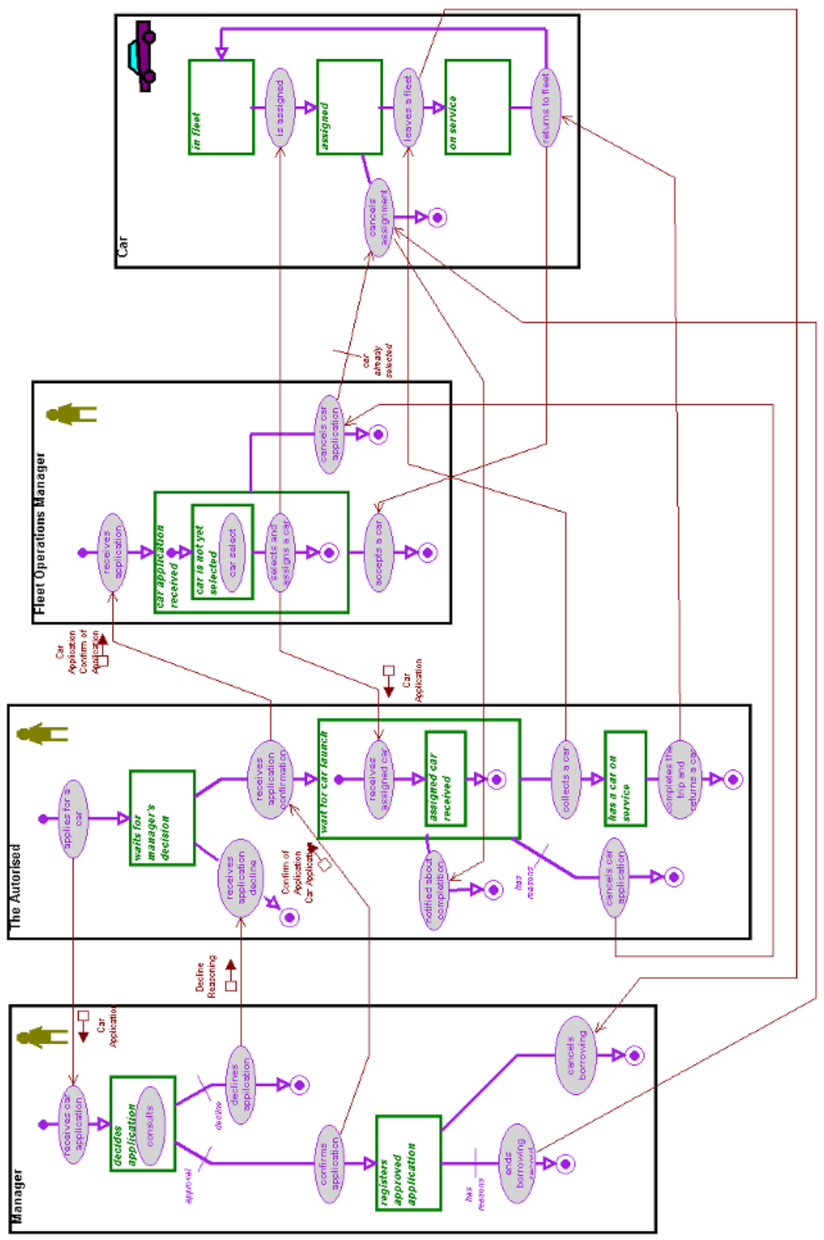
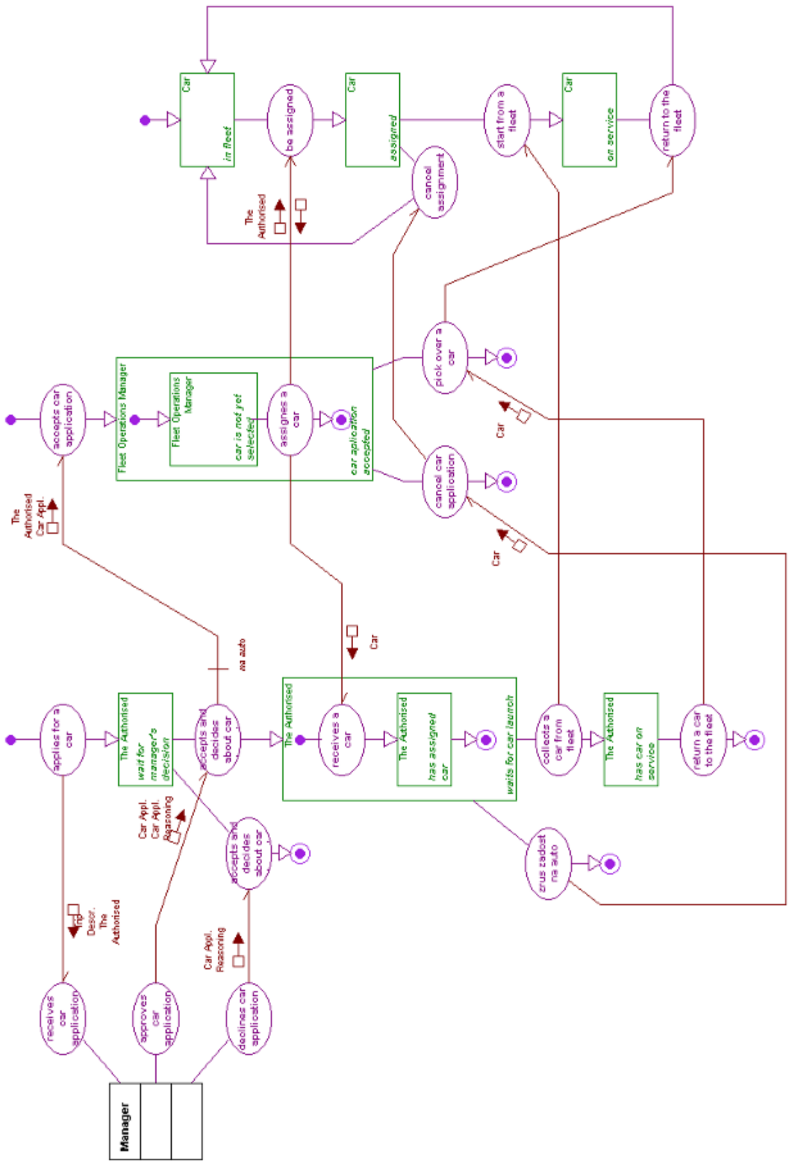


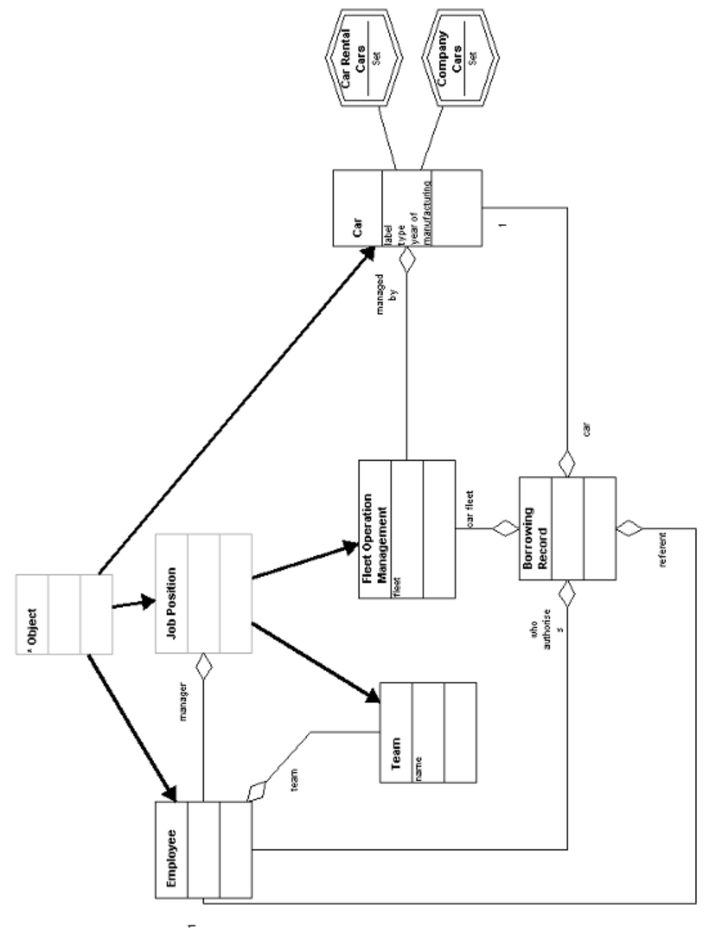
Figure 6. The conceptual diagram, created from Figure 4



to have class hierarchy structure, even though we will have two different collections of 'Car' objects. The criterion we use is to create a new class only if its behavior is different from its subclass.

The ovals are methods (Operations in UML) that the classes provide. The arrows between them show message passing, where the messages could be

Figure 7. Client-side software diagram

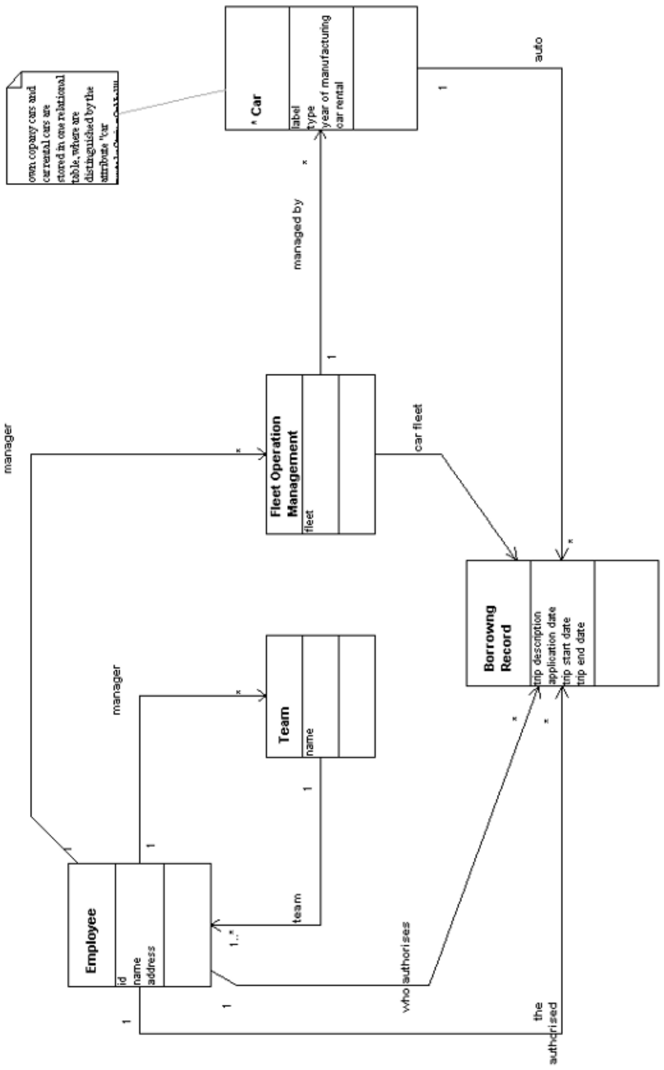


either synchronous (normal arrowhead) or asynchronous (half arrowhead). This diagram shows both object relations and behavior.

Figure 7 shows the client-side software objects. This model is obtained by a simple transformation from the type hierarchy into an inheritance hierarchy. In general, such a transformation is based on the need to fit conceptual object structure into concrete software environment structures (Blaha & Premerlani, 1998), which are limited by:

a. target programming language syntax and paradigm, and
b. reuse of existing objects from legacy components.

Figure 8. Server-side software diagram



In the example, the implementation of the type hierarchy has to take into account the restriction that employee information is held in an existing legacy database. This has necessitated the construction of additional types in the database for job-position, team, and manager. In such a diagram we are working with software objects, which have software relationships with other objects that need to reflect the concrete implementation environment. Black arrows are used to show inheritance. Note, this is classical object-oriented inheritance and hence is not the same as our previous type hierarchy. This is because inheritance is not the only way to implement new object types; we may implement new object types by composition. In our example, three conceptual types—manager, authorized employee, and fleet manager—will be implemented as instances of the legacy class employee composed with the new class job-position.

In this phase, it is also very important to know what classes are reused and what concepts are created as new ones or as reused artifacts. We denote the former by putting an asterisk before the name.

At this stage we also create a server side software object diagram.

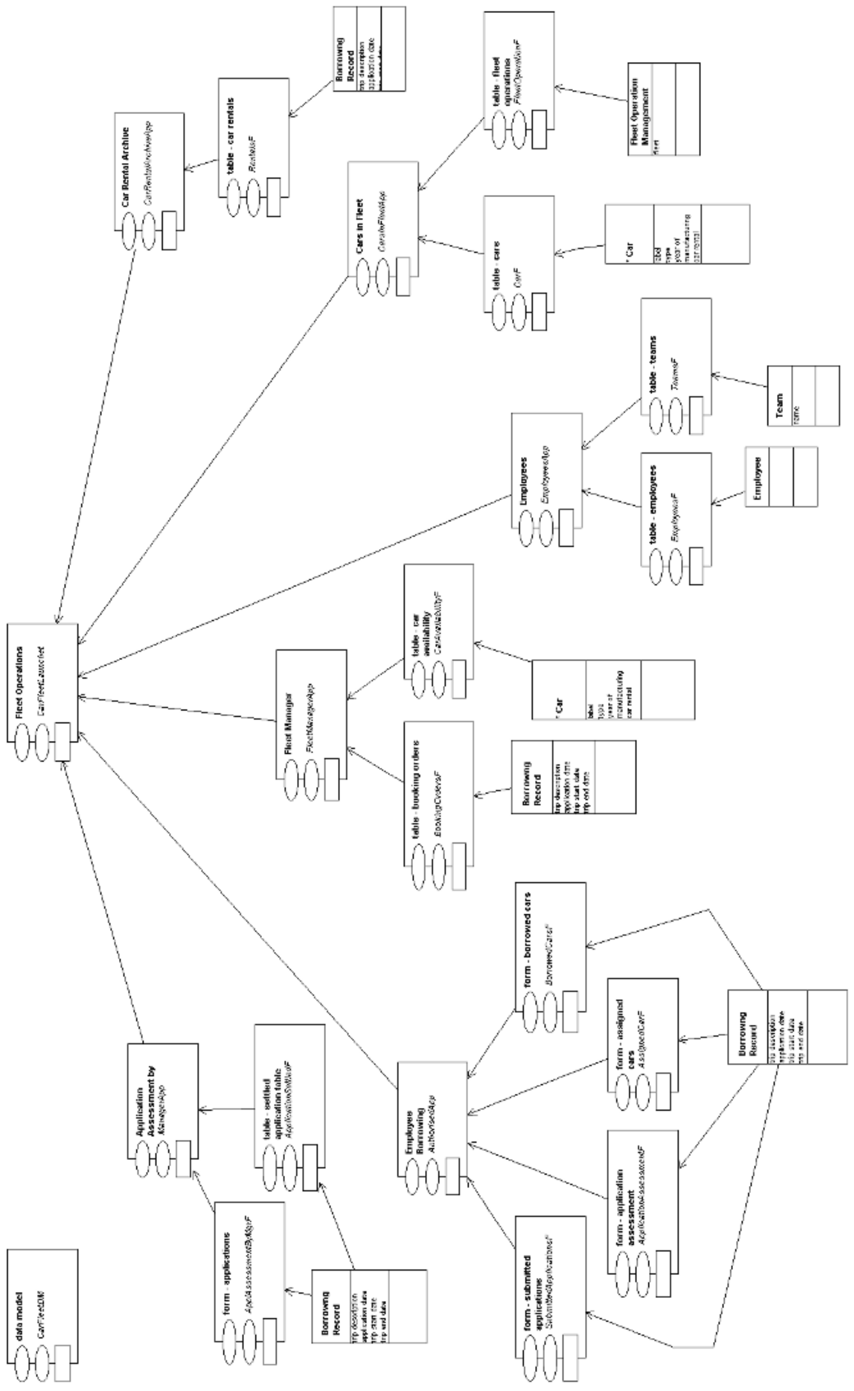
The reason we need to differentiate between client- and server-side software models is because the system is often implemented in two different programming environments: pure object-oriented VisualWorks/Smalltalk for the client and relational ODBC/Oracle DBMS for the server.

Our final diagram is a hierarchy of Software components. Each component has its main (or interface) class and may be constructed out of other components or may serve as the link to some database table on the server. For example, in VisualWork's (Hopkins & Horan, 1995) ObjectLens, special classes are created for implementing database access. If we work with instances of this class, the system internally performs data manipulation operations on the server. Consequently, each instance of this special class is linked with one row in the relation table). The previous hierarchy diagram was directly used to implement the system using this tool.

BORM and XP

XP (Beck, 2000) is a relatively new way to develop software. It is based on a few basic premises, which are also common to other 'lightweight' technologies often described as 'Agile techniques'(Ambler, 2002). However, there are problems using the standard XP approach in the requirements analysis phase. We have used the BORM business process modeling approach to help analysts during this initial phase. In real-world projects with analysts, we found that the

Figure 9. Components



use of BORM functions and scenarios and participant modeling cards provided an ideal tool in which to model user stories. In addition, we found that the BORM process diagram provided an essential tool in those areas when re-engineering of current business systems was a prerequisite for subsequent software development.

One of the factors of XP is that the code should provide the documentation. This approach is fine if the only people who need to read the documents are other programmers who can read the code. However, we have found on a number of occasions a need to provide documentation of the developed system for the end users who did not have this programming expertise. In these cases we found that the BORM process diagrams provided a useful source of information that was comprehensible to such non-technical end users. The diagrams were also used to provide data for the generation of user tests and user manuals. These results were unforeseen outcomes of using BORM to supplement the XP approach.

BORM and RUP (Unified Process)

RUP is one of the most widely used methodologies for software development. Much of RUP is compatible with BORM; both are concerned with the system lifecycle. We think that the major differences between RUP and BORM are the following:

1. BORM provides a gradual transformation of models based on precisely defined constraint rules (for example, by applying patterns and refactoring).
2. BORM provides greater support for pure object-oriented concepts like refactoring which are an integral part of the BORM development process.
3. BORM provides support for the maintenance phase of the system lifecycle. The outcome of this phase is frequently the source for the start of the next development cycle. In this aspect, BORM follows the ideas of Ambler (1998).

In BORM, software development is regarded as an essential sequel to business process analysis. Without the business context being considered first, any software analysis is deficient.

The Sources of UML and of BORM

UML arose from a number of different sources. Whatever their differences, however, most of the initial sources were from methodologies that were developing systems using hybrid languages like C++ and relational database systems. C++ has for many applications now been succeeded by Java or occasionally by C#, but the differences between these languages are not very significant during the design process. These origins are reflected in the nature and structure of UML.

BORM arose from developing systems using Smalltalk and object-oriented databases. Smalltalk is still the most successful current implementation of the pure object-oriented paradigm. BORM was designed as a methodology and related notation with the intent of using pure object-oriented concepts. In was a belief of the BORM developers that the pure object-oriented paradigm was the best way of modeling real-world problems and that this approach should be maintained for as long as possible during the development process. Indeed, for most applications, it is only in the final design stages that the developers have to start thinking about how the model is to be implemented. It is the BORM belief that the development of the initial system model using real-world analysis, rather than software system concepts, leads to a better and more robust design.

If a system is described using UML and RUP, the designers from the initial stages have to constrict their thinking to the restrictions imposed by hybrid languages and relational databases. If the same developers were to use BORM, they would be restricted only in the implementation phase by the limitations imposed by the inability of current programming languages to fully support the object-oriented paradigm.

In BORM we do not separate the static and dynamic aspects of the problem modeled; we put them both in the same model. The BORM approach of combining dynamic and static aspects can be achieved in a UML class diagram, using standard UML elements, but only by the extensive use of a stereotype mechanism and elements normally used in other diagrams. The consequence is that such diagrams are very complex and cluttered, and often unacceptable to both designers and customers. In BORM we have begun to use a slightly modified form of the UML diagrams to obtain a model more easily understood by those from a UML background. The basic modeling elements used in UML diagrams have been extended by the addition of new symbols to denote any pure object-oriented concepts required. This approach is coherent with the other works for extending UML; for example the xUML project with communicating state machines and the UML business extension project based on the former Jacobson's method known as Objectory. The structure of UML allows the notation to be extended and for UML concepts to be combined. There are many academic projects looking at valid ways in which UML can be extended, but it is still true that the significant majority of analysts using UML use only basic UML diagrams. This approach, based on a minimum set of UML diagrams, is the only approach supported by current CASE tools.

BORM is supported by two special CASE tools: Craft.CASE (*www.craftcase.com)* and Meta.

The Advantages of BORM

1. BORM is based on the premise that business process modeling *must* be carried out before any software system development. Consequently, BORM provides a consistent approach and notation to the analysis and design of business environment, the derivation of the software requirement specification, and finally the analysis and design of the software.
2. BORM follows the process-oriented approach (Darnton & Darnton, 1997), combined with the pure object-oriented paradigm, which has proven to be beneficial in software development Generally, we believe the process-oriented approach led to a faster and more comprehensive analysis of the problem being solved.
3. In our experience stakeholders from the problem domain are able to understand the BORM approach very quickly—normally a one-hour introduction at the start of analysis is enough.
4. In Deloitte & Touche's Prague office, a business consulting team has worked for the past four to five years using the BORM system, as well as ARIS and Rational's Objectory/Unified method. They have found BORM to be on average three to four times faster in carrying out the analysis phase compared to other methods.
5. The methodology is easily acceptable to domain experts, analysis consultants, and developers. Because BORM is based on a step-by-step transformation of the model, in each phase only a limited and consistent subset of BORM concepts are used.

6. BORM has been used enthusiastically by Smalltalk and Java programmers and by non-relational object database programmers. One feature of BORM they find attractive is the way it exploits collection concepts, not just classes and the way that these collection classes are seamlessly integrated into the development environment.
7. BORM, as it originated from a pure object-oriented concept, has more object hierarchies (polymorphism, is-a, dependency) than other software development methods, which derived mainly from C++ programming; it further only provides concepts supported by programming languages. Usually only object inheritance is supported. In BORM, we also work with object relationships, which do not have direct implementation in current programming languages, but have proven to be useful in the conceptual modeling of problem domain.
8. These last two features provide a much richer language with which to express modeling ideas.

Conclusion

Today, when improved visual programming tools—combined with the support of rapid application development environments—are available, it would appear that the whole software development process is becoming easier. This statement is true, however, only for those cases where the complexity of the solution and of users' requirements is relatively simple. But business systems developed for real companies often have a very high level of complexity and uncertainty, especially in the initial development phases, which make development much more difficult. Consequently, it is essential (*from the software developer's viewpoint*) to improve the initial phases of software development.

Until recently, it was correctly assumed that conceptual modeling tools and techniques were used through all stages of project development, from the initial phase to the eventual implementation. However, the position of conceptual modeling is currently being used solely in the implementation phase, as a result of the evolution of software development tools. The analysis is now being performed using newly developed techniques and "business" objects modeling tools.

The authors would like to acknowledge the support of the research project MSM6046070904.

References

Ambler, S.W. (1998). *Process patterns.* Cambridge: Cambridge University Press.

Ambler, S.W. (2002). *Agile modeling.* Indianapolis*:* John Wiley & Sons.

Bahrami, A. (1999). *Object-oriented system development.* New York: McGraw-Hill.

Beck, K. (2000). *Extreme Programming explained.* Reading, MA: Addison-Wesley.

Blaha, M., & Premerlani, W. (1998). *Object-oriented modeling and design for database applications*. Upper Saddle River, NJ: Prentice-Hall.

Booch, G., Rumbaugh, J., & Jacobson, I. (1998). *The Unified Modeling Language user guide.* Reading, MA: Addison-Wesley.

Cantor, M. (1998). *Object-oriented project management with UML.* Indianapolis: John Wiley & Sons.

Carteret, C., & Vidgen, R. (1995). *Data modeling for information systems.* London: Pitman Publishing.

Coterrell, M., & Hughes, B. (1995). *Software project management.* London: Thomson Computer Press.

Darnton, G., & Darnton, M. (1997). *Business process analysis.* London: International Thomson Publishing.

Date, C.J. (1995). *An introduction to database systems* (6th ed.). Reading, MA: Addison-Wesley.

Eriksson, H.-E., & Penker, M. (2000). *Business modeling with UML.* Indianapolis: John Wiley & Sons.

Fowler, M., & Scott, K. (1999). *UML distilled* (2nd ed.). Reading, MA: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R ., & Vlissides, J. (1994). *Design patterns.* Reading, MA: Addison-Wesley.

Goldberg, A., & Rubin, K.S. (1995). *Succeeding with objects—decision frameworks for project management.* Reading, MA: Addison-Wesley.

Hopkins, T., & Horan, B. (1995). *Smalltalk—an introduction to application development using VisualWorks.* Upper Saddle River, NJ: Prentice-Hall.

Jacobson, I. (1992). *Object-oriented software engineering—a use-case-driven approach*. Reading, MA: Addison-Wesley.

Knott, R.P., Merunka, V., & Polak, J. (2000). Process modeling for object-oriented analysis using BORM object behavioral analysis. *Proceedings of the 4th International Conference on Requirements Engineering* (ICRE 2000) (pp. 7-16). Chicago: IEEE Computer Society Press.

Knott, R.P., Merunka, V., & Polak, J. (2003a). The BORM methodology: A third-generation fully object-oriented methodology. *Knowledge-Based Systems*, *16*, 77-89.

Knott, R.P., Merunka, V., & Polak, J. (2003b). The role of object-oriented process modeling in requirements engineering phase of information systems development. In Z. Harnos, M. Herdon, & T. Wiwczaroski (Eds.), *Proceedings of the 4th EFITA Conference* (pp. 300-307). Debrecen, Hungary: University of Debrecen.

Polak, J., Merunka, V., & Carda, A. (2003). *Umení systémového návrhu.* Prague: Grada.

Royce, W. (1998). *Software project management: A unified framework***.** Reading, MA: Addison-Wesley.

Rubin, K.S., & Goldberg, A. (1992). Object behavior analysis. *Communications of the ACM, 35.*

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1991). *Object-oriented modeling and design*. Upper Saddle River, NJ: Prentice-Hall.

Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The Unified Modeling Language reference manual.* Reading, MA: Addison-Wesley.

Shlaer, S., & Mellor, S.J. (1992). *Object-oriented systems analysis: Modeling the world in states.* Upper Saddle River, NJ: Prentice-Hall.

Simone, A.J.H., & Graham, I. (1999). 30 things that go wrong in object modeling with UML 1.3. In H. Kilov, B. Rumpe, & I. Simmonds (Eds.), *Behavioral specifications of businesses and systems.* Dordrecht: Kluwer Academic Publishers.

Yourdon, E. (1995). *Mainstream objects—an analysis and design approach for business.* Upper Saddle River, NJ: Prentice-Hall.

About the Authors

Liping Liu is an associate professor of management and information systems at The University of Akron (USA). He earned his PhD in business from the University of Kansas (1995). His research interests are in the areas of uncertainty reasoning and decision making in artificial intelligence, electronic business, systems analysis and design, technology adoption, and data quality. His articles have appeared in *Decision Support Systems*, *European Journal of Operational Research*, *IEEE Transactions*, *Information and Management*, *Journal of Risk and Uncertainty*, and others. He is known for his theories of course utilities and linear belief functions, which have been taught in the nation's top PhD programs in computer science, economics, accounting, management, and psychology. He currently serves as a co-editor for *Classic Works on Dempster-Shafer Theory of Belief Functions*, and on editorial boards and committees of many international journals and conferences. He has strong practical and teaching interests in e-business systems design, development, and integration, and has won several teaching awards. His recent consulting experience includes designing and developing a patient record management system, a payroll system, a course management system, and an e-travel agent.

Boris Roussev is an associate professor of CIS at the University of the Virgin Islands (USA). His diverse background includes teaching and research experience in Europe, South Africa, and the U.S. Dr. Roussev's interests are in the areas of object-oriented and economic-driven software development, requirements engineering, and project management. He conducts research on causes of project failures, working from the presumptions that value-neutral principles and practices in software development are unable to deal with the sources of project failures, and that in order to manage effectively the computer technology, one has to consider the social context in which it is deployed and produced. In addition, Dr. Roussev has experience in software risk management and software process quality factor analysis. All of the above is combined with industry experience in software methods such as domain engineering, product lines, MDA with xUML, generative programming, and aspect-oriented programming. Dr. Roussev's most recent research initiative is an interdisciplinary project on object-oriented linguistics and semiotics.

* * *

Ram Akella is professor and director of information systems and technology management at the University of California at Silicon Valley Center/Santa Cruz (USA). At Stanford, Berkeley, and Carnegie Mellon, as a faculty member and a director, Dr. Akella has led major multi-million-dollar interdisciplinary team efforts in high tech and semiconductors. He earned his BS degree from IIT Madras and his PhD from IISc Bangalore. Dr. Akella completed postdoctoral work at Harvard University and worked at MIT. His research and teaching interests include IT, enterprise software, knowledge management, product lifecycle management, supply chain management, financial engineering and investment, business process optimization, and e-business. Faculty awards include those from IBM, AMD, and KLA, and Dr. Akella has been cited in Marquis' *Who's Who*. He has interacted extensively with industries, including many of the U.S., European, Asian, Japanese, and Indian software and hardware companies. Dr. Akella has served as an associate editor for *Operations Research* and *IEEE*.

Jorn Bettin is a software consultant with a special interest in techniques to optimize the productivity of software development teams and in designing large-scale component systems. He is managing director of SoftMetaWare, a consultancy that provides strategic technology management advice, with resources based in the U.S., New Zealand/Australia, and Europe. Prior to co-founding SoftMetaWare in 2002, he worked for over 13 years as a consultant and mentor in the IT industry in Germany, New Zealand, and Australia. He has implemented automated, model-driven development in several software organizations and has worked in methodology leadership roles in an IBM product development lab. Mr. Bettin and two co-authors have published a book about model-driven software development in German entitled, *Modellgetriebene Softwareentwicklung*, and expect to publish an English edition in 2005.

Gary K. Evans is a nationally known agile process evangelist, object technology mentor, instructor, and course developer for Evanetics (*www.evanetics.com*) based in Columbia, South Carolina (USA). His focus is on reducing project risk and eliminating uncertainty on large, distributed software development projects. He provides object technology mentoring, consulting, and training for Java, C++, VB, and C# development. He speaks on diverse object technology topics at national technical conferences, including Software Development Best Practices, the Rational User Conference, and UML World. He is a recognized expert in use case development, object modeling, the IBM Rational Unified Process™, agile software processes, and in applying object-oriented practices to non-object-oriented languages such as COBOL. He is actively engaged in the agile software development community, and he has published more than a dozen articles for various technical magazines, including *The Rational Edge*. He is a contributing editor for *Software Development* magazine, for which he specializes in object-oriented CASE tools, methods, and agile software process. He holds a BS in computer science (with High Honors) from the School of Engineering and Applied Science, University of Virginia, and he is a member of the Tau Beta Pi National Engineering Honor Society. He also holds a BA in philosophy from Lehigh University, USA.

Javier Garzás (javier.garzas@m-centric.com or jgarzas@gmail.com) is a project manager at mCentric in Madrid, Spain, where he drives the software process improvement and software quality assurance program. He is also a lecturer at the Rey Juan Carlos University of Madrid. His current responsibilities include leading mCentric to CMM Level 3. Due to his experience at several important companies, his research and software engineering skills cover areas such as OO design, CMM, software process, and project management. He obtained his MSc and PhD degrees in computer science at the University of Castilla - La Mancha. He holds a master's degree in enterprise application integration as well.

Esther Guerra works at GFI Madrid as project manager. She is also working toward her PhD at the Universidad Autonóma de Madrid in the area of formal methods for software development. Her main research interests include meta-modeling, verification of UML, and hypermedia systems.

Roger Knott is a lecturer in the Department of Computer Science at Loughborough University in the heart of the British Midlands. He began programming in the late 1960s using machine-code, as no high-level languages were available. Since discovering Smalltalk 80, he has been a keen advocate of pure object-oriented methods.

Juan de Lara is an associate professor at the Universidad Autonóma (UAM) de Madrid in Spain, where he teaches software engineering, automata theory, as well as modeling and simulation. His research interests include Web-based simulation, meta-modeling, graph transformation, distance learning, and social agents. He received his PhD in Computer Science in June 2000 at UAM. During 2001, as a post-doctoral researcher at McGill University, he created the AToM^3 prototype. Later, he also spent time at TU Berlin working on graph transformation.

John D. McGregor is a partner at Luminary Software and an associate professor of computer science at Clemson University (USA). He conducts research, teaches graduate software engineering courses, and serves as a consultant to companies in several domains. Dr. McGregor has conducted research for organizations such as the Software Engineering Institute, National Science Foundation, DARPA, IBM and AT&T. He has applied those research results on projects in telecommunications, insurance, and financial institutions. He is co-author of *A Practical Guide to Testing Object-Oriented Software* (Addison-Wesley). Dr. McGregor's current research interests include software product lines, design quality, testing, and measurement.

Vojtech Merunka is a lecturer in the Faculty of Economics at the University of Agriculture, Prague, Czech Republic. He is a leading light in the computer science community and well known for his advocacy of object-oriented system developments. Many of his former students have set up some of the leading IT companies currently operating the Czech Republic. Professor Merunka also works as a consultant and has been responsible for many software projects, several of which have been developed using BORM.

Gerald N. Miller is the chief technology officer for Microsoft's U.S. Central Region.

Mario Piattini earned MS and PhD degrees in computer science from the Polytechnic University of Madrid, and an MS in psychology from the UNED. He became a certified information system auditor and certified information security manager by ISACA (Information System Audit and Control Association). He is a full professor at the Department of Computer Science at the University of Castilla - La Mancha, in Ciudad Real, Spain, and author of several books and papers on databases, software engineering, and information systems. He leads the ALARCOS research group specializing in information system quality. His research interests are software quality, advanced database design, metrics, software maintenance, information system audit, and security.

Gary Pollice is professor of practice at Worcester Polytechnic Institute, Worcester, Massachusetts (USA). He teaches software engineering, design, testing, and other computer science courses, and also directs student projects. Before entering the academic world, he spent more than 35 years developing various kinds of software, from business applications to compilers and tools. His last industry job was with IBM Rational Software, where he was known as 'the RUP Curmudgeon'; he was also a member of the original Rational Suite team. Professor Pollice is the primary author of *Software Development for Small Teams: A RUP-Centric Approach* (2004, Addison-Wesley). He holds a BA in mathematics and an MS in computer science.

Jiri Polak is a senior partner in Deloitte & Touche's Eastern European division and is based in Prague, Czech Republic. Prior to working for Deloitte, he was a lecturer at the Czech Technical University in Prague. At Deloitte he has been responsible for a number of significant software projects, many of which have been developed using BORM. This has enabled the methodology to be honed in the real-world environment.

Stefan Roock works as a consultant for agile methods, object-oriented architectures, and technologies. He has many years of industrial experience helping software development teams to go agile.

Yvonna Rousseva teaches English at the University of the Virgin Islands (USA). She holds a master's degree in English philology, and continues to be fascinated by the nature of the linguistic sign and the inexhaustible possibilities of language. Among her versatile interests are language theory, object-oriented linguistics, discourse and communication theory, hermeneutics, active learning, modeling, and metaphors. Professor Rousseva is also a poet whose creative journey is inspired by the search for the similar in the dissimilar, and by the notion that there is no disparity between poetry and abstract thought.

Melissa L. Russ is a partner in Luminary Software and a project engineer at the Space Telescope Science Institute in Baltimore (USA). In her current position she directs a team of system engineers in defining the system requirements for future space telescopes. She has many years of experience in developing, implementing, and improving software development processes. As a consultant she has also mentored others in these same tasks. She is co-author of several technical articles and research papers in journals such as *IEEE Software* in the areas of object-oriented software development and requirements, and process definition.

Magdy K. Serour is a research fellow at the Centre for Object Technology Applications and Research (COTAR) at the University of Technology, Sydney (UTS). He is the co-founder of SB, the Software Group Pty Ltd (1982), and has 28 years of experience in Information Technology, being significantly involved in object technology adoption, requirement engineering, modeling, implementation, and IT consulting. Dr. Serour's current interests are in organizational transition, e-transformation, agile methodologies, software process reengineering and software process improvement, and capability determination for object-oriented/component-based software development (OOSPICE). He holds a BS in accounting (Cairo University), a GDipl in computing (ICL, London), a Dipl in computing (Control Data, Sydney), an MS in computing (UWS, Sydney), and a PhD (UTS, Sydney) in the area of migration to object-oriented technology.

Hans Vangheluwe is an assistant professor in the School of Computer Science at McGill University (Canada), where he teaches modeling and simulation, as well as software design. He also heads the Modeling, Simulation, and Design Lab (*http://msdl.cs.mcgill.ca*). Some of his model compiler work has led to the WEST++ tool, which was commercialized for use in the design and optimization of waste water treatment plants. He was the co-founder and coordinator of the European Union's ESPRIT Basic Research Working Group 8467 "*Simulation in Europe,*" and a founding member of the Modelica Design Team.

Henning Wolf works as a project manager in agile software development projects. One of his tasks is the effort estimation of projects.

Index

A

abstraction 93, 99
action language 48
activity diagrams 299
adaptation 244
advanced analysis 342
advanced design 342
agents 155
agile 53, 67, 88, 98, 111, 123, 313
agile alliance 98
agile alliance manifesto 111
agile MDA 53
agile methodology 88
agile methods 111, 313
agile outsourcing 123, 126
agile practices 111
agile software development 67
analysis level architecture 153
analyst 57, 242
analyzability 332
architect 57
architectural design 326
AspectJ 66
aspects 65
association class 47
associations 347
attack risk 244

B

BERT 55
Big Requirements Up Front (BRUF) 242
bridges 39
BRUF (Big Requirements Up Front) 242
business analysts 242
business components 224
business object relationship modeling 337
business value invariant 155

C

capability maturity model 115
CASE 67, 236, 251
changeability 6, 332
class diagrams 10
client activities 114
coding standard 188
collective code ownership 188
common language runtime 285

communication 149
comparison statement 146
complex application 328
complexity pyramid 116
component 65, 211
component platforms 215
component specifications 218
component specification 216
computation-independent model 98
computer-oriented 342
conceptual model 339
concern 64, 89
configuration item 128
configuration management 128
consolidation stages 343
context 148
continuous integration 188
CORBA component model 21
core values 184
cost 62
counter-actor 155
critical success factor 261, 266
Crystal 315

D

data flow 53, 250
deadline 314
declarative knowledge 328
dependency inversion 330
design artifacts 331
design patterns 210
design quality 331
designer 57
development case 233
diagrams 10
differential algebraic equations 292
discourse analysis 150
domain 43, 203
domain-specific languages 106

E

E-Z lanes 10
effort estimation 318
effort points 317
enterprise architecture 72
entity relationship diagrams 250, 342
ERNIE 55
ERP systems 113
essential object 339
estimation 317
executable model 47
executable models 53
executable UML 5, 37, 87, 154
expansion stages 343
exploration phase 318
externalization 39
eXtreme Programming 183, 206, 234, 313

F

feature effort points 317
finite state machine (FSM) 5, 43, 159
flow charts 250
foreign equity participation 117
formalisms 291
fourth generation languages 309
fractal 209
FSM (finite state machine) 5, 43, 159

G

generative programming 309
gestalt 244
graph grammars 296

H

hermeneutics 149
hump chart 184

I

IBM 278
identifier 45
implementation 342
Indian IT industry 110
industrialized software asset development 209
initial analysis 342
initial design 342
initial operation capability 186
interactivity 117
interface places 302
ISO 9001 115

J

J2EE 283
Jacobson, R. 149
Java virtual machine 285

K

kernel comparison statement 146
knowledge 328

L

labeled transition system analyzer 166
language 48, 285, 309
late externalization 39
leadership 265
level of abstraction 93
lifecycle objectives 185

M

macro-level 216
macroeconomic context 100
maintainers 58
mappings 38
maturity model 115
MDA 9, 37, 54, 89
MDA workflow 54
meta-formalism 291
metaAttributes 292
metaClasses 292
metamodel 40, 291
metaOperations 292
metaphor 188
metaRelationships 292
method 63, 96
method war 206
micro architectural design 326
microeconomic context 100
Microsoft .NET 283
Microsoft Corporation 278
Microsoft Visual Studio 284
model of communication 149
model-based verification 51
model-driven architecture
    5, 37, 87, 154, 290
model-driven software development
    9, 202, 207
model-driven software development 36
modeling paradigm 201
modeling/designing 251
models 63
motivation factor 264
mutual relationships 344

N

non-functional testing 53
non-hierarchical 154
non-IT client activities 114
notation 93

O

object behavior analysis 340, 344
object constraint language 22
object model 250
object-orientated analysis 148
object-orientation 1, 201
object-oriented 1, 248, 325
object-oriented design 5, 327
object-oriented programming 282
OnePointPass 18
onsite/offsite developers 136
open source 72, 209
open source software 209
operative knowledge 328
Oracle DBMS 353
organization 96
organizational structures 93
outsourcing 72, 101, 113
outsourcing strategy 101
over-specification 152
Oxford English Dictionary 254

P

pair programming 188
PassOnePointTollgate 13
pattern 330
personal software process 69
Petri nets 37, 299
PIM (platform independent model) 9,
    37, 291

platform independent model (PIM) 9, 37, 291
platform specific model (PSM) 9, 37, 291
platforms 215
precise action semantics 42, 165
prescriptive 233
process 76, 98, 232, 337
process framework 232
process model 98, 337
product development 79
product line strategy 102
programming 188
project controlling 112, 313
project plan 73
prototyping 144
PSM (platform specific models) 9, 37, 291

Q

quality of service 11

R

rational unified process (RUP) 68, 183, 205, 232
rational's objectory 357
refactoring 188
referential attributes 46
requirements analysts 56
requirements engineering 142, 260
requirements validation 143
resistance 258
roles 56
RUP (rational unified process) 68, 183, 205, 232

S

scenario 144
schedule 62
scope 96
SCRUM 315, 321
search application 328
seed developers 123
seeds 146
SEI/CMM 115
sequence diagrams 27
service-oriented architecture 280
Shlaer-Mellor method 37
simple design 188
small wins 260
smalltalk 353
sociological factors 250
software asset development 209
software complexity chain 116
software development 1, 9, 63, 110, 202, 214
Software Engineering Institute 62
software process 69, 117, 174, 306
software process engineering metamodel 306
software-intensive products 63
Spirit of the RUP, The 192
stability 332
state machines 5
statecharts 300
states 344
stories 315
story burndown charts 315
story effort points 317
strategic analysis 342
strategic objectives 63
strategies 63
structural transformations 345
subsystems 215
Sun Microsystems 278
sustainable pace 188

T

Tag ID 157
TagGenerator 157
team software process 69
technologies 63
test-driven 188
test-first 188
tester 58
testing 53
tools 95
transformational mappings 38
transitions 344
TwoPointPass 18

U

UDDI 282
UML (unified modeling language) 3, 40, 66, 232, 293
UML activity diagrams 293
UML graphical models 40
unified method 357
unified modeling language (UML) 3, 40, 66, 232, 293
uniformity 209
use case 232, 250
user requirements model 152

V

velocity 314
virtual machine 285
visual studio 285
VisualWorks 353

W

waterfall 234
WSDL 282

X

X-axis 28
XML routers 283
XP 313
xUML 45, 89