Chapter Five: Object oriented System Design

5.1. Introduction

System design is the transformation of an analysis model into a system design model. During system design, developers define the design goals of the project and decompose the system into smaller subsystems that can be realized by individual teams. Developers also select strategies for building the system, such as the hardware/software strategy, the persistent data management strategy, the global control flow, the access control policy, and the handling of boundary conditions. The result of system design is a model that includes subsystem decomposition and a clear description of each of these strategies.

System design is not algorithmic. Developers have to make trade-offs among many design goals that often conflict with each other. They also cannot anticipate all design issues that they will face because they do not yet have a clear picture of the solution domain. System design is decomposed into several activities, each addressing part of the overall problem of decomposing the system:

- ***Identify design goals*.** Developers identify and prioritize the qualities of the system that they should optimize.
- ***Design the initial subsystem decomposition*.** Developers decompose the system into smaller parts based on the use case and analysis models. Developers use standard architectural styles as a starting point during this activity.
- ***Refine the subsystem decomposition to address the design goals*.** The initial decomposition usually does not satisfy all design goals. Developers refine it until all goals are satisfied.

5.2. An overview of system design.

Analysis results in the requirements model described by the following products:

- a set of *nonfunctional requirements* and *constraints,* such as maximum response time, minimum throughput, reliability, operating system platform, and so on
- a *use case model,* describing the system functionality from the actors' point of view
- an *object model,* describing the entities manipulated by the system
- a *sequence diagram* for each use case, showing the sequence of interactions among objects participating in the use case.

The design goals are derived from the nonfunctional requirements. Design goals guide the decisions to be made by the developers when trade-offs are needed. The subsystem decomposition constitutes the bulk of system design. Developers divide the system into manageable pieces to deal with complexity: each subsystem is assigned to a team and realized independently. For this to be possible, developers need to address system-wide issues when decomposing the system. In this chapter, we describe the concept of subsystem decomposition and discuss examples of generic system decompositions called "architectural styles." In the next chapter, we describe how the system decomposition is refined to meet specific design goals. Figure 5-1 depicts the relationship of system design with other software engineering activities.

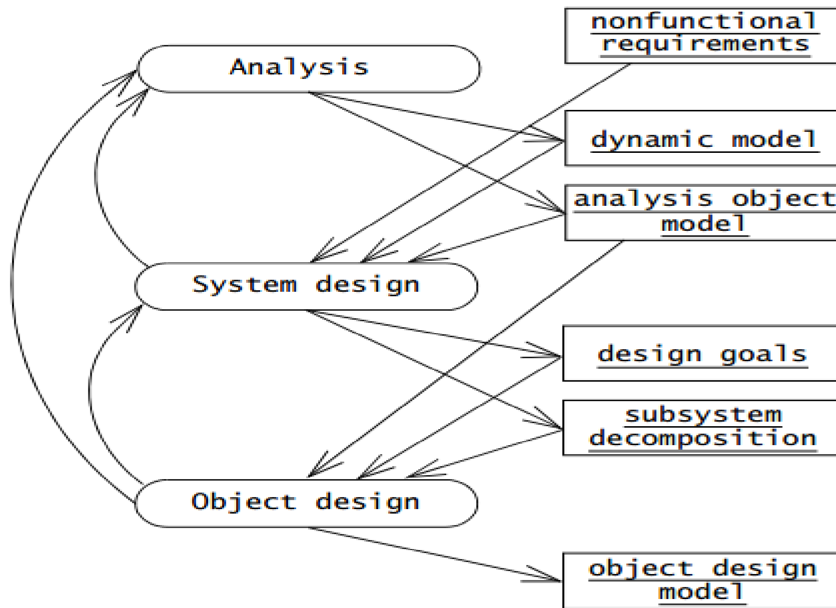

Figure 5-1 the activities of system design (UML activity diagram).

5.3 System design concepts.

5.3.1 Subsystems and Classes

In order to reduce the complexity of the application domain, we identified smaller parts called "classes" and organized them into packages. Similarly, to reduce the complexity of the solution domain, we decompose a system into simpler parts, called "subsystems," which are made of a number of solution domain classes. A **subsystem** is a replaceable part of the system with well-defined interfaces that encapsulates the state and behavior of its contained classes. A subsystem typically corresponds to the amount of work that a single developer or a single development team can tackle. By decomposing the system into relatively independent subsystems, concurrent teams can work on individual subsystems with minimal communication overhead. In the case of complex subsystems, we recursively apply this principle and decompose a subsystem into simpler subsystems (see Figure 5-2)

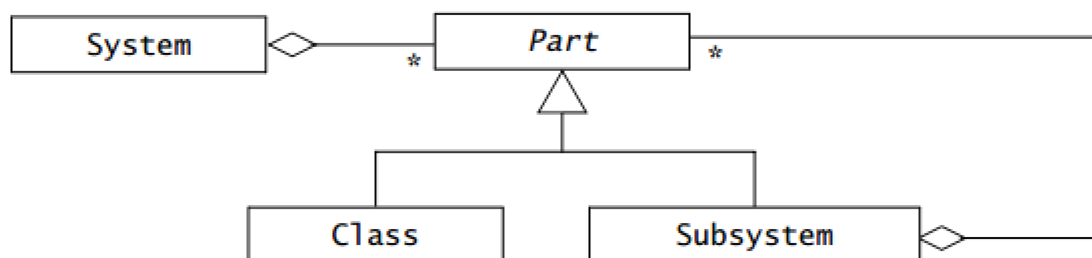

Figure 5-2 Subsystem decomposition (UML class diagram)

5.3.2 Services and Subsystem Interfaces

A subsystem is characterized by the services it provides to other subsystems. A **service** is a set of related operations that share a common purpose. A subsystem providing a notification service, for example, defines operations to send notices, look up notification channels, and subscribe and unsubscribe to a channel. The set of operations of a subsystem that are available

to other subsystems form the **subsystem interface**. The subsystem interface includes the name of the operations, their parameters, their types, and their return values. System design focuses on defining the services provided by each subsystem that is, enumerating the operations, their parameters, and their high-level behavior. Object design will focus on the **application programmer interface** (API), which refines and extends the subsystem interfaces. The API also includes the type of the parameters and the return value of each operation.

5.3.3 Coupling and Cohesion

Coupling is the number of dependencies between two subsystems. If two subsystems are loosely coupled, they are relatively independent, so modifications to one of the subsystems will have little impact on the other. If two subsystems are strongly coupled, a modification to one subsystem is likely to have impact on the other. A desirable property of subsystem decomposition is that subsystems are as loosely coupled as reasonable. This minimizes the impact that errors or future changes in one subsystem have on other subsystems.

Cohesion is the number of dependencies within a subsystem. If a subsystem contains many objects that are related to each other and perform similar tasks, its cohesion is high. If a subsystem contains a number of unrelated objects, its cohesion is low. A desirable property of subsystem decomposition is that it leads to subsystems with high cohesion.

5.4 System design activities: From objects to subsystems

System design consists of transforming the analysis model into the design model that takes into account the nonfunctional requirements described in the requirements analysis document. This material illustrates these activities with an example, MyTrip, a route planning system for car drivers. We start with the analysis model from MyTrip; then we describe the identification of design goals (Section 5.4.1) and the design of initial system decomposition (Section 5.4.2).

6.4.1 Identifying Design Goals

The definition of design goals is the first step of system design. It identifies the qualities that our system should focus on. Many design goals can be inferred from the nonfunctional requirements or from the application domain. Others will have to be elicited from the client. It is, however, necessary to state them explicitly such that every important design decision can be made consistently following the same set of criteria.

Design goals for MyTrip

- **Reliability:** MyTrip should be reliable [generalization of nonfunctional requirement 2].
- **Fault Tolerance:** MyTrip should be fault tolerant to loss of connectivity with the routing service [rephrased nonfunctional requirement 2].
- **Security:** MyTrip should be secure, i.e., not allow other drivers or nonauthorized users to access a driver’s trips [deduced from application domain].
- **Modifiability:** MyTrip should be modifiable to use different routing service [anticipation of change by developers].

6.4.2 Identifying Subsystems

The initial subsystem decomposition should be derived from the functional requirements. For example, in the MyTrip system, we identify two major groups of objects: those that are involved during the PlanTrip use case and those that are involved during the ExecuteTrip use case. The Trip, Direction, Crossing, Segment, and Destination classes are shared between both use cases. This set of classes is tightly coupled as it is used as a whole to represent a Trip. We decide to assign them with PlanningService to the PlanningSubsystem, and the remainder f the classes are

assigned to the RoutingSubsystem (Figure 6-29). This leads to only one association crossing subsystem boundaries. Note that this subsystem decomposition is a repository in which the PlanningSubsystem is responsible for the central data structure.

Another heuristic for subsystem identification is to keep functionally related objects together. A starting point is to assign the participating objects that have been identified in each use case to the subsystems. Some group of objects, as the Trip group in MyTrip, are shared and used for communicating information from one subsystem to another. We can either create a new subsystem to accommodate them or assign them to the subsystem that creates these objects.

Heuristics for grouping objects into subsystems

- Assign objects identified in one use case into the same subsystem.
- Create a dedicated subsystem for objects used for moving data among subsystems.
- Minimize the number of associations crossing subsystem boundaries.
- All objects in the same subsystem should be functionally related.

5.5 Documenting system design

System design is documented in the System Design Document (SDD). It describes design goals set by the project, subsystem decomposition (with UML class diagrams), hardware/software mapping (with UML deployment diagrams), data management, access control, control flow mechanisms, and boundary conditions. The SDD is used to define interfaces between teams of developers and serve as a reference when architecture-level decisions need to be revisited. The audience for the SDD includes the project management, the system architects (i.e., the developers who participate in the system design), and the developers who design and implement each subsystem. Figure 5-3 is an example template for a SDD.

The first section of the SDD is an *Introduction*. Its purpose is to provide a brief overview of the software architecture and the design goals. It also provides references to other documents and traceability information (e.g., related requirements analysis document, references to existing systems, constraints impacting the software architecture).

The second section, *Current software architecture*, describes the architecture of the system being replaced. If there is no previous system, this section can be replaced by a survey of current architectures for similar systems. The purpose of this section is to make explicit the background information that system architects used, their assumptions, and common issues the new system will address.

The third section, *Proposed system architecture*, documents the system design model of the new system. It is divided into seven subsections:

- *Overview* presents a bird's-eye view of the software architecture and briefly describes the assignment of functionality to each subsystem.
- *Subsystem decomposition* describes the decomposition into subsystems and the responsibilities of each. This is the main product of system design.
- *Hardware/software mapping* describes how subsystems are assigned to hardware and off-the-shelf components. It also lists the issues introduced by multiple nodes and software reuse.
- *Persistent data management* describes the persistent data stored by the system and the data management infrastructure required for it. This section typically includes the description of data schemes, the selection of a database, and the description of the encapsulation of the database.

- *Access control and security* describes the user model of the system in terms of an access matrix. This section also describes security issues, such as the selection of an authentication mechanism, the use of encryption, and the management of keys.
- *Global software control* describes how the global software control is implemented. In particular, this section should describe how requests are initiated and how subsystems synchronize. This section should list and address synchronization and concurrency issues.
- *Boundary condition describes* the start-up, shutdown, and error behavior of the system. (If new use cases are discovered for system administration, these should be included in the requirements analysis document, not in this section.)

The fourth section, *Subsystem services,* describes the services provided by each subsystem. Although this section is usually empty or incomplete in the first versions of the SDD, this section serves as a reference for teams for the boundaries between their subsystems. The interface of each subsystem is derived from this section and detailed in the Object Design Document.

The SDD is written after the initial system decomposition is done; that is, system architects should not wait until all system design decisions are made before publishing the document. The SDD, moreover, is updated throughout the process when design decisions are made or problems are discovered. The SDD, once published, is baselined and put under configuration management. The revision history section of the SDD provides a history of changes as a list of changes, including author responsible for the change, date of change, and brief description of the change.

System Design Document

1. Introduction
   1.1 Purpose of the system
   1.2 Design goals
   1.3 Definitions, acronyms, and abbreviations
   1.4 References
   1.5 Overview
2. Current software architecture
3. Proposed software architecture
   3.1 Overview
   3.2 Subsystem decomposition
   3.3 Hardware/software mapping
   3.4 Persistent data management
   3.5 Access control and security
   3.6 Global software control
   3.7 Boundary conditions
4. Subsystem services
   Glossary

Figure 5-3 Example outline for the System Design Document (SDD).

5.6 An overview of object design

Conceptually, software system development fills the gap between a given problem and an existing machine. The activities of system development incrementally close this gap by identifying and defining objects that realize part of the system.

Analysis reduces the gap between the problem and the machine by identifying objects representing problem-specific concepts. During analysis the system is described in terms of external behavior such as its functionality (use case model), the application domain concepts it manipulates (object model), its behavior in terms of interactions (dynamic model), and its nonfunctional requirements.
System design reduces the gap between the problem and the machine in two ways. First, system design results in a virtual machine that provides a higher level of abstraction than the machine. This is done by selecting off-the-shelf components for standard services such as middleware, user interface toolkits, application frameworks, and class libraries. Second, system design identifies off-the-shelf components for application domain objects such as reusable class libraries of banking objects.
After several iterations of analysis and system design, the developers are usually left with a puzzle that has a few pieces missing. These pieces are found during object design. This includes identifying new solution objects, adjusting off-the-shelf components, and precisely specifying each subsystem interface and class. The object design model can then be partitioned into sets of classes that can be implemented by individual developers.

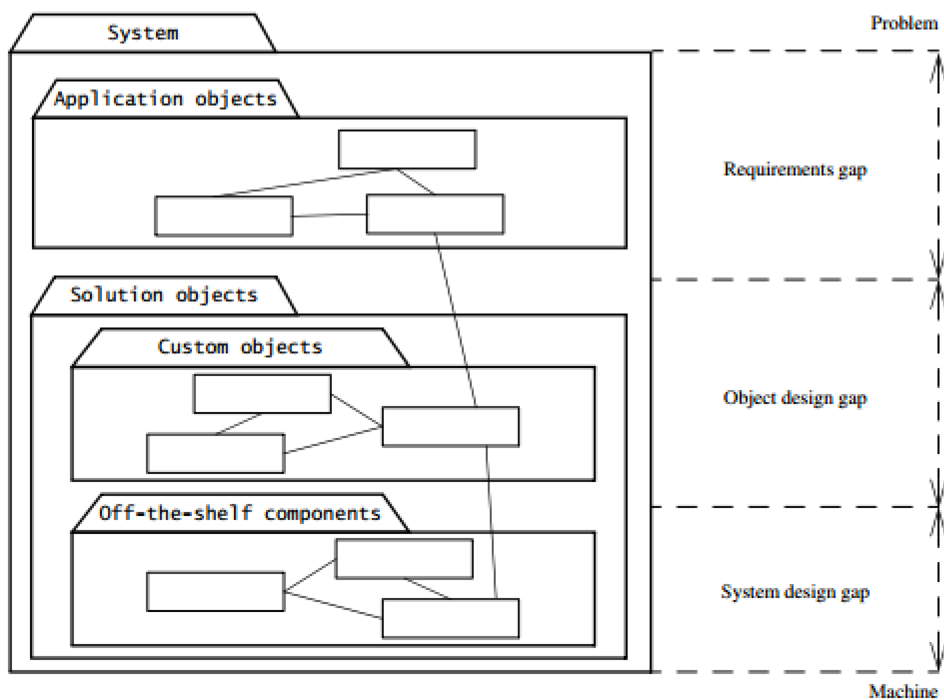

Figure 5-4 Object design closes the gap between application objects identified during requirements and off-the-shelf components selected during system design (stylized UML class diagram)

Object design includes four groups of activities

- *Reuse*. Off-the-shelf components identified during system design are used to help in the realization of each subsystem. Class libraries and additional components are selected for basic data structures and services. Design patterns are selected for solving common problems and for protecting specific classes from future change. Often, components and design patterns need to be adapted before they can be used. This is done by

wrapping custom objects around them or by refining them using inheritance. During all these activities, the developers are faced with the same buy-versus-build trade-offs they encountered during system design

- *Interface specification*. During this activity, the subsystem services identified during system design are specified in terms of class interfaces, including operations, arguments, type signatures, and exceptions. Additional operations and objects needed to transfer data among subsystems are also identified. The result of service specification is a complete interface specification for each subsystem. The subsystem service specification is often called subsystem **API** (Application Programmer Interface)
- *Restructuring*. Restructuring activities manipulate the system model to increase code reuse or meet other design goals. Each restructuring activity can be seen as a graph transformation on subsets of a particular model. Typical activities include transforming N-ary associations into binary associations, implementing binary associations as references, merging two similar classes from two different subsystems into a single class, collapsing classes with no significant behavior into attributes, splitting complex classes into simpler ones, and/or rearranging classes and operations to increase the inheritance and packaging. During restructuring, we address design goals such as maintainability, readability, and understandability of the system model.
- *Optimization*. Optimization activities address performance requirements of the system model. This includes changing algorithms to respond to speed or memory requirements, reducing multiplicities in associations to speed up queries, adding redundant associations for efficiency, rearranging execution orders, adding derived attributes to improve the access time to objects, and opening up the architecture, that is, adding access to lower layers because of performance requirements.

Object design is not sequential. Although each group of activities described above addresses a specific object design issue, they usually occur concurrently. A specific off-the-shelf component may constrain the number of types of exceptions mentioned in the specification of an operation and thus may impact the subsystem interface. The selection of a component may reduce the implementation work while introducing new “glue” objects, which also need to be specified. Finally, restructuring and optimizing may reduce the number of components to be implemented by increasing the amount of reuse in the system.

Usually, interface specification and reuse activities occur first, yielding an object design model that is then checked against the use cases that exercise the specific subsystem. Restructuring and optimization activities occur next, once the object design model for the subsystem is relatively stable. Focusing on interfaces, components, and design patterns results in an object design model that is much easier to modify. Focusing on optimizations first tends to produce object design models that are rigid and difficult to modify.

5.7. Object design concepts

5.7.1. Reuse Concepts: Solution Objects, Inheritance, and Design Patterns

As we saw in Chapter 2, *Modeling with UML*, class diagrams can be used to model both the application domain and the solution domain. **Application objects**, also called “domain objects,” represent concepts of the domain that are relevant to the system. **Solution objects** represent components that do not have a counterpart in the application domain, such as persistent data stores, user interface objects, or middleware. During analysis, we identify entity objects and their relationships, attributes, and operations. Most entity objects are application

objects that are independent of any specific system. During analysis, we also identify solution objects that are visible to the user, such as boundary and control objects representing forms and transactions defined by the system. During system design, we identify more solution objects in terms of software and hardware platforms. During object design, we refine and detail both application and solution objects and identifies additional solution objects needed to bridge the object design gap.

5.7.2. Specification Inheritance and Implementation Inheritance

During analysis, we use inheritance to classify objects into taxonomies. This allows us to differentiate the common behavior of the general case, that is, the **superclass** (also called the "**base class**"), from the behavior that is specific to specialized objects, that is, the **subclasses** (also called the "**derived classes**"). The focus of generalization (i.e., identifying a common superclass from a number of existing classes) and specialization (i.e., identifying new subclasses given an existing superclass) is to organize analysis objects into an understandable hierarchy. Readers of the analysis model can start from the abstract concepts, grasp the core functionality of the system, and make their way down to concrete concepts and review specialized behavior. For example, when examining the analysis model for the FRIEND emergency response system described in Chapter 3, *Requirements Elicitation*, we first focus on understanding how the system deals with Incidents in general, and then move to the differences in handling Traffic Accidents or Fires.

The focus of inheritance during object design is to reduce redundancy and enhance extensibility. By factoring all redundant behavior into a single superclass, we reduce the risk of introducing inconsistencies during changes (e.g., when repairing a defect) since we have to make changes only once for all subclasses. By providing abstract classes and interfaces that are used by the application, we can write new specialized behavior by writing new subclasses that comply with the abstract interfaces. For example, we can write an application manipulating images in terms of an abstract Image class, which defines all the operations that all Images should support, and a series of specialized classes for each image format supported by the application (e.g., GIFImage, JPEGImage). When we need to extend the application to a new format, we only need to add a new specialized class.

5.8. Object design activities

System design and object design introduce a strange paradox in the development process. On the one hand, during system design, we construct solid walls between subsystems to manage complexity by breaking the system into smaller pieces and to prevent changes in one subsystem from affecting other subsystems. On the other hand, during object design, we want the software to be modifiable and extensible to minimize the cost of future changes. These are conflicting goals: we want to define a stable architecture to deal with complexity, but we also want to allow flexibility to deal with change later in the development process. This conflict can be solved by anticipating change and designing for it, as sources of later changes tend to be the same for many systems:

- *New vendor or new technology*. Commercial components used to build the system are often replaced by equivalent ones from a different vendor. This change is common and generally difficult to cope with. The software marketplace is dynamic, and vendors might go out of business before your project is completed.
- *New implementation*. When subsystems are integrated and tested together, the overall system response time is, more often than not, above performance requirements. System

performance is difficult to predict and should not be optimized before integration. Developers should focus on the subsystem services first. This triggers the need for more efficient data structures and algorithms—often under time constraints.

- *New views*. Testing the software with real users uncovers many usability problems. These often translate into the need to create additional views on the same data.
- *New complexity of the application domain*. The deployment of a system triggers ideas of new generalizations: a bank information system for one branch may lead to the idea of a multi-branch information system. The application domain itself might also increase in complexity: previously, flight numbers were associated with one plane, and one plane only, but with air carrier alliances, one plane can now have a different flight number from each carrier.
- *Errors*. Many requirements errors are discovered when real users start using the system

5.8.1. **Encapsulating Data Stores with the Bridge Pattern**

Consider the problem of incrementally developing, testing, and integrating subsystems realized by different developers. Subsystems may be completed at different times, delaying the integration of all subsystems until the last one is completed. To avoid this delay, projects often use a stub implementation in place of a specific subsystem so that the integration tests can start even before the subsystems are completed. In other situations, several implementations of the same subsystem are realized, such as a reference implementation that realizes the specified functionality with the most basic algorithms, or an optimized implementation that delivers better performance at the cost of additional complexity. In short, a solution is needed for dynamically substituting multiple realizations of the same interface for different uses.

5.8.2 Encapsulating Legacy Components with the Adapter Pattern

As the complexity of systems increases and the time to market shortens, the cost of software development significantly exceeds the cost of hardware. Hence, developers have a strong incentive to reuse code from previous projects or to use off-the-shelf components. Interactive systems, for example, are now rarely built from scratch; they are developed with user interface toolkits that provide a wide range of dialogs, windows, buttons, or other standard interface objects. Interface engineering projects focus on re-implementing only part of an existing system. For example, corporate information systems, costly to design and build, must be updated to new client hardware. Often, only the client side of the system is upgraded with new technology; the back end of the system left untouched. Whether dealing with off-the-shelf component or legacy code, developers have to deal with code they cannot modify and which usually was not designed for their system.

5.9. Managing object design

This section discusses management issues related to object design. There are two primary management challenges during object design:

- *Increased communication complexity*. The number of participants involved during this phase of development increases dramatically. The object design models and code are the result of the collaboration of many people. Management needs to ensure that decisions among these developers are made consistently with project goals.
- *Consistency with prior decisions and documents*. Developers often do not appreciate completely the consequences of analysis and system design decisions before object design. When detailing and refining the object design model, developers may question some of these decisions and reevaluate them. The management challenge is to maintain

a record of these revised decisions and to make sure all documents reflect the current state of development.

5.9. Documenting object design.

Object design is documented in the **Object Design Document (ODD)**. It describes object design trade-offs made by developers, guidelines they followed for subsystem interfaces, the decomposition of subsystems into packages and classes, and the class interfaces. The ODD is used to exchange interface information among teams and as a reference during testing. The audience for the ODD includes system architects (i.e., the developers who participate in the system design), developers who implement each subsystem, and testers.

There are three main approaches to documenting object design:

- *Self-contained ODD generated from model*. The first approach is to document the object design model the same way we documented the analysis model or the system design model: we write and maintain a UML model and generate the document automatically. This document would duplicate any application objects identified during analysis. The disadvantages of this solution include redundancy with the Requirements Analysis Document (RAD) and a high level of effort for maintaining consistency with the RAD. Moreover, the ODD duplicates information in the source code and requires a high level of effort whenever the code changes. This often leads to an RAD and an ODD that are inaccurate or out of date.
- *ODD as extension of the RAD*. The second approach is to treat the object design model as an extension of the analysis model. In other terms, the object design is considered as the set of application objects augmented with solution objects. The advantage of this solution is that maintaining consistency between the RAD and the ODD becomes much easier as a result of the reduction in redundancy. The disadvantages of this solution include polluting the RAD with information that is irrelevant to the client and the user. Moreover, object design is rarely as simple as identifying additional solution objects. Often, application objects are changed or transformed to accommodate design goals or efficiency concerns.
- *ODD embedded into source code*. The third approach is to embed the ODD into the source code. Once the ODD becomes stable, we use the modeling tool to generate class stubs. We describe each class interface using tagged comments that distinguish source code comments from object design descriptions. We can then generate the ODD using a tool that parses the source code and extracts the relevant information. Once the object design model is documented in the code, we abandon the initial object design model. The advantage of this approach is that the consistency between the object design model and the source code is much easier to maintain: when changes are made to the source code, the tagged comments are updated and the ODD regenerated. In this section, we focus only on this approach.

The fundamental issue is one of maintaining consistency among two models and the source code. Ideally, we want to maintain the analysis model, the object design model, and the source code using a single tool. Objects would then be described once, and consistency among documentation, stubs, and code would be maintained automatically. Presently, however, UML modeling tools provide facilities for generating a document from a model or class stubs from a model. For example, the glossary of the RAD can be generated from the analysis model by collating the description fields attached to each class. The class

stub generation facility, called **forward engineering**, can be used in the self-contained ODD approach to generate the class interfaces and stubs for each method.

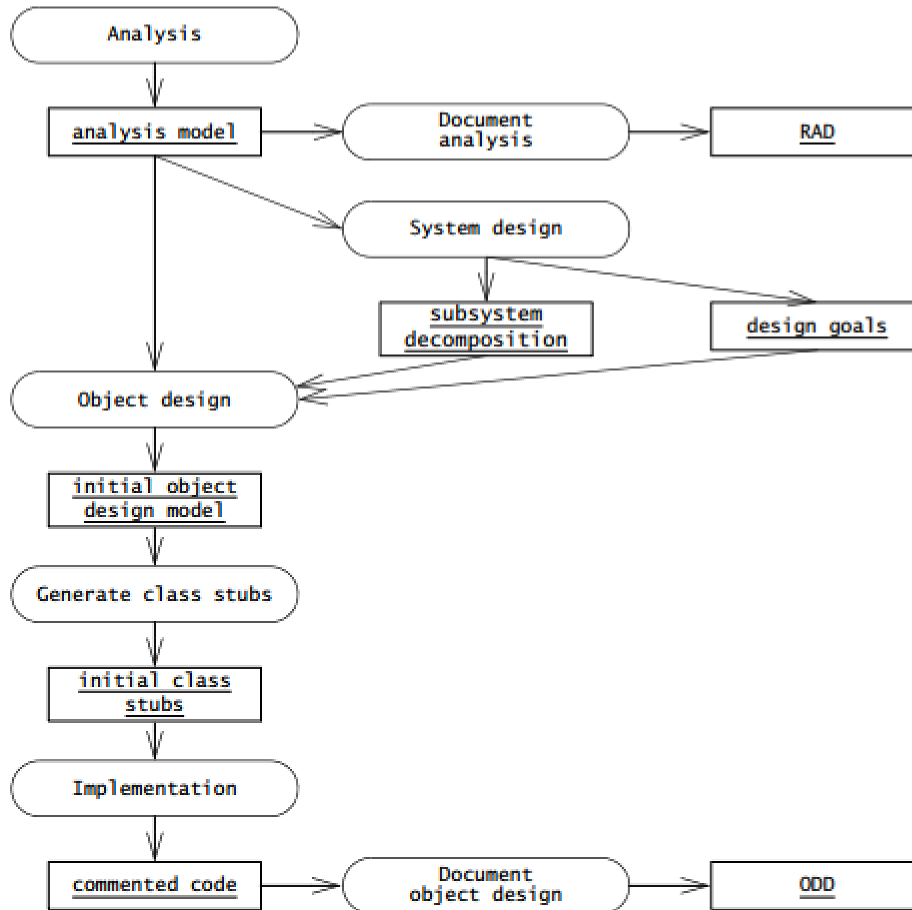

Figure 5-5 Embedded ODD approach. Class stubs are generated from the object design model. The object design model is then documented as tagged comments in the source code. The initial object design model is abandoned and the ODD is generated from the source code instead using a tool such as Javadoc (UML activity diagram).