8. System Design

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

– C.A.R. Hoare

System design is the transformation of the requirements analysis model into a system design model. The system design model views a system as a collection of subsystems, their dependencies, and their interfaces.The system design model includes all decisions impacting the overall system structure, also called the software architecture. System design is not algorithmic. Professionals and academics have, however, developed pattern solutions to common problems and defined notations for representing software architectures. In this chapter, we first present these building blocks that are available to you during system design and then discuss the design activities that have impact on these buildings blocks. In particular, system design includes:

- the definition of design goals,
- the decomposition of the system into subsystems,
- the selection off-the-shelf and legacy components,
- the mapping of subsystem to hardware,
- the selection of a persistent data management infrastructure,
- the selection of an access control policy,
- the selection of a global control flow mechanism, and
- the handling of boundary conditions.

We conclude this chapter by describing management issues related to system design.

8.1. Introduction: a floorplan example

System design, object design, and implementation constitute the construction of the system. During these three activities, developers fill the gap between the system specification, produced during requirements analysis, and reality, that is, the system that is delivered to the users. System design is the first step in this process and focuses on decomposing the system into manageable parts. During requirements analysis, we concentrated on the purpose and the functionality of the system. During system design, we focus on the processes, datastructures, software and hardware components necessary to implement it. The challenge of system design is that many conflicting criteria and constraints need to be met when decomposing the system.

Consider, for example, the task of designing a residential house. After agreeing with the client on the number of rooms and floors, the size of the living area, and the location of the house, the architect must design the floorplan, that is, where the walls, doors, and windows should be located. He must do so according to a number of functional requirements: the kitchen should be close to the dining room and the garage, the bathroom should be close to the bedrooms, and so on. The architect can also rely on a number of standards when establishing the dimensions of each room and the location of the door: kitchen cabinets come in fixed increments and beds come in standard sizes (e.g., in the U.S., twin size, queen size, and king size). Note, however, that the architect does not need to know the exact content of each room and the layout of the furniture, on the contrary, these decisions should be delayed and left to the client.

Figure 106 shows three successive revisions to a floor plan for a residential house. We set out to satisfy the following constraints:

1. This house should have two bedrooms, a study, a kitchen, and a living room area.
2. The overall distance the occupants walk every day should be minimized.
3. The use of daylight should be maximized.

To satisfy the above constraints, we assume that most of the walking will be done between the entrance door and the kitchen, when groceries are unloaded from the car, and between the kitchen and the living / dining area, when dishes are carried before and after the meals. The next walking path to minimize is the path from the bedrooms to the bathrooms. We assume that the occupants of the house will spend most of their time in the dining area and in the master bedroom.

In the first version of our floorplan (at the top of Figure 106), we find that the dining room is too far from the kitchen. To address this problem, we exchange it with bedroom 2 (see gray arrows in Figure 106). This has also the advantage of moving the living room to the south wall of the house. In the second revision, we find that the kitchen and the stairs are too far

from the entrance door. To address this problem, we move the entrance door to the north wall. This allows us to rearrange bedroom 2 and move the bathroom closer to both bedrooms. The living area increased, and we satisfied all constraints we originally set out.

At this point, we can position the doors and the windows of each room to meet localized requirements. Once this is done, we have completed the design of the floor, without detailed knowledge of the layout of each individual room. Plans for plumbing, electrical lines, and heating ducts can proceed.

The design of a floorplan in architecture is similar to system design in software engineering. The whole is divided into simpler components (i.e., rooms, subsystems) and interfaces (i.e., doors, services) while taking into account nonfunctional (i.e., living area, response time) and functional (i.e., number of bedrooms, use cases) requirements. System design impacts implementation activities (i.e., the kitchen layout, the coding complexity of individual subsystems) and results in costly rework if changed later (i.e., moving walls, changing subsystem interfaces). The design of individual components is delayed until later.

In Section 8.2, we provide you with a bird view of system design and its relationship to requirements analysis. In Section 8.3, we describe the concept of subsystems and subsystem decomposition. In Section 8.4, we describe system design activities and illustrate how these building blocks can be used together using an example. In Section 8.5, we describe management issues related to system design.

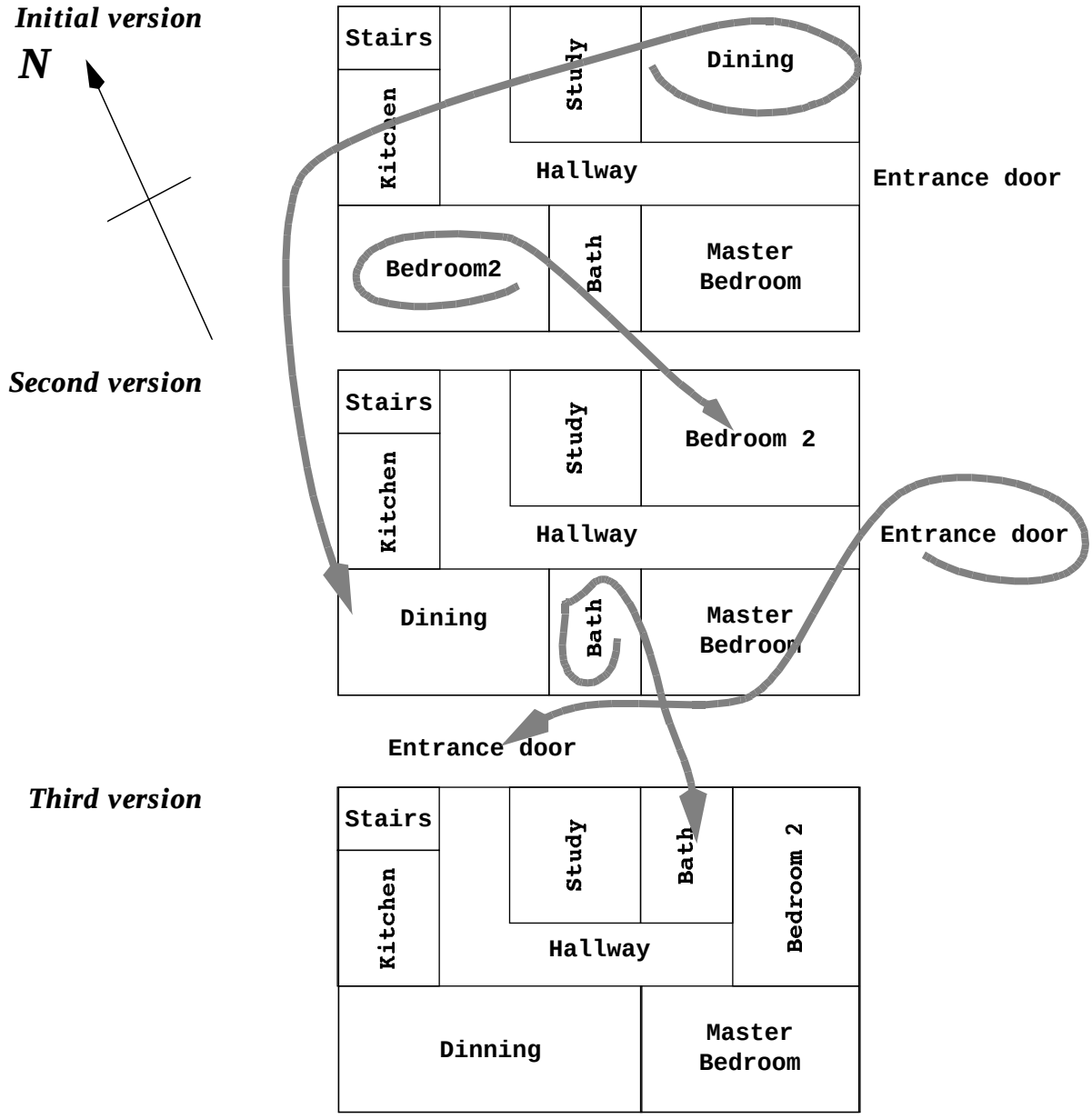


FIGURE 106. Example of floorplan design. Three successive versions show how we minimize walking distance and take advantage of sunlight.

8.2. An overview of system design

Requirements 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 functionality of the system 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 requirements analysis model describes the system completely from the actors' point of view and serves as the basis of communication between the client and the developers. The requirements analysis model, however, does not contain information about the internal structure of the system, its hardware configuration, or, more generally, how the system should be realized. System design is the first step in this direction. System design results in the following products:

- A list of *design goals,* describing the qualities of the system that developers should optimize.
- A *software architecture,* describing the subsystem decomposition in terms of their responsibilities, their dependencies, their mapping to hardware, and major policy decisions such as control flow, access control, data storage.

The design goals are derived from the nonfunctional requirements. They guide the decisions to be made by the developers especially 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. In order for this to be possible, though, developers need to address system wide issues when decomposing the system. In particular, they need to address the following issues:

- *Hardware/software mapping:* What is the hardware configuration of the system? Which node is responsible for which functionality? How is communication between nodes realized? Which services are realized using existing software components? How are these components encapsulated? Addressing hardware/software mapping issues often leads to the definition of *additional subsystems* dedicated to moving data from one node to another, dealing with concurrency, and reliability issues. Off-the-shelf components enable developers to realize complex services more economically. User interface packages and database management systems are prime examples of off-the-shelf components. Components, however, should be encapsulated to minimize

dependencies on a particular component: a competing vendor may come with a better component.

- *Data management:* Which data need to be persistent? Where should persistent data be stored? How is it accessed? Persistent data represents a bottleneck in the system on many different fronts: most functionality in system is concerned with creating or manipulating persistent data. For this reason, access to the data should be fast and reliable. If retrieving data is slow, the whole system will be slow. If data corruption is likely, complete system failure is likely. These issues need to be addressed consistently at the system level. Often, this leads to the selection of a database management system and of an *additional subsystem* dedicated to the management of persistent data.
- *Access control:* Who can access which data? Once an actor has access to data, can it modify it? Can access control change dynamically? How is access control specified and realized? Access control and security are system wide issues. The access control must be consistent across the system, in other words, the policy used to specify who can and cannot access certain data should be the same *across all subsystems.*
- *Control flow:* How does the system sequence operations? Is the system event driven? Can it handle more than one user interaction at a time? The choice of control flow has an impact on the interfaces of subsystems. If an event-driven control flow is selected, subsystems will provide event handlers. If threads are selected, subsystems need to guarantee mutual exclusion in critical sections.
- *Boundary conditions:* How is the system initialized? How is it shutdown? How are exceptional cases detected and handled? System initialization and shutdown often represent the larger part of the complexity of a system, especially in a distributed environment. Initialization, shutdown, and exception handling have an impact on the interface of *all subsystem*s.

Figure 107 depicts the activities of system design. Each activity addresses one of the issues we described above. Addressing any one of these issues can lead to changes in the subsystem decomposition and to raising new issues. As you will see when we describe each of these activities, system design is a highly iterative activity, constantly leading to the identification of new subsystems, the modification of existing subsystems, and system wide revisions that impact all subsystems. But first, let us describe in more detail subsystems.

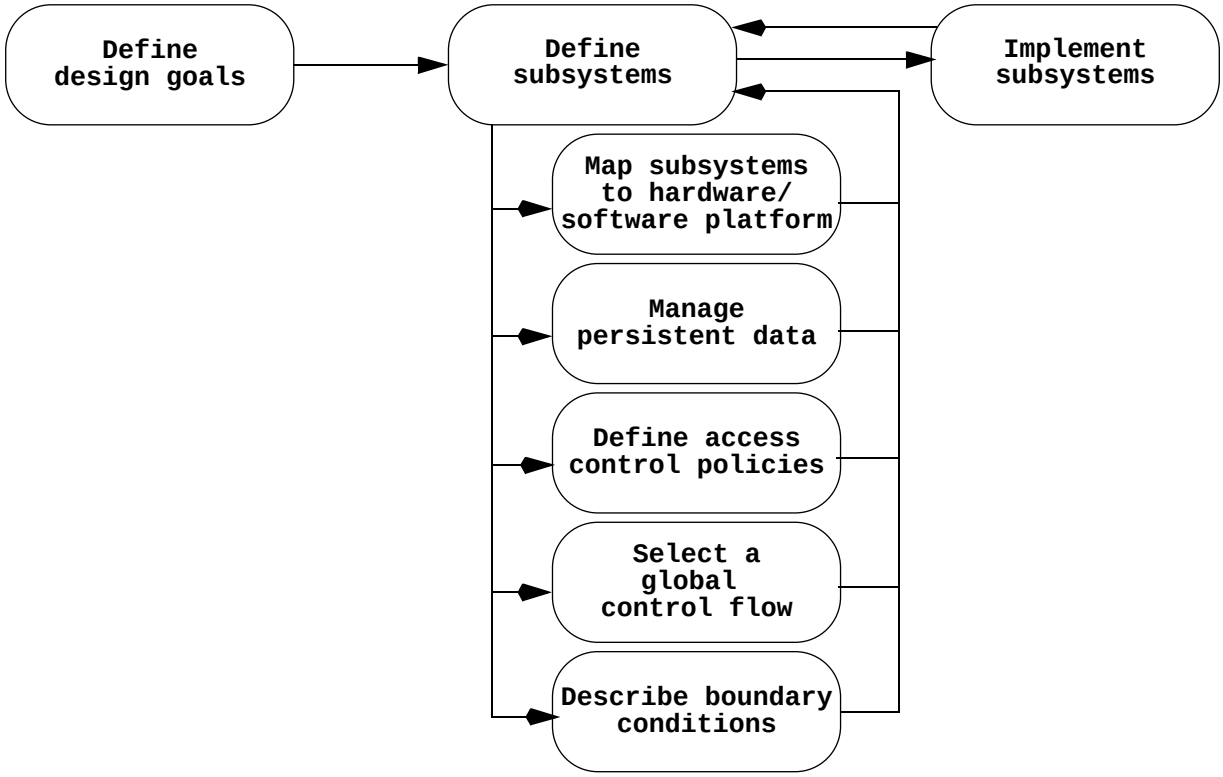


FIGURE 107. The activities of system design (UML activity diagram).

8.3. Subsystem decomposition

In this section, we describe in more detail subsystem decompositions and their properties. First, we define the concept of ***subsystem*** and their relationship to classes (Section 8.3.1). Next, we look at the interface of subsystems (Section 8.3.2): subsystems provide ***services*** to other subsystems. A service is a set of related operations that share a common purpose. During system design, we define the subsystems in terms of the services they provide. Later, during object design, we define the subsystem interface in terms of the operations they provide. Next, we look at two properties of subsystems: ***coupling*** and ***coherence*** (Section 8.3.3). Coupling measures the dependencies between two subsystems while coherence measures the dependencies among classes within a subsystem. Ideal subsystem decomposition should minimize coupling and maximize coherence. Next, we look at ***layering*** and ***partitioning***, two techniques for relating subsystems to each other (Section 8.3.4). Layering allows a system to be organized as a hierarchy of subsystems, each providing higher level services to the subsystem above it using lower level services from the subsystems below it. Partitioning organizes subsystems as peers that mutually provide different services to each other. Finally, in Section 8.3.5, we describe a number of typical software architectures that are found in practice.

8.3.1. Subsystems and classes

In Chapter 7, *Requirements Analysis* we introduced the distinction between application domain and solution domain. 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. In the case of complex subsystems, we recursively apply this principle and decompose a subsystem into simpler subsystems (see Figure 108).

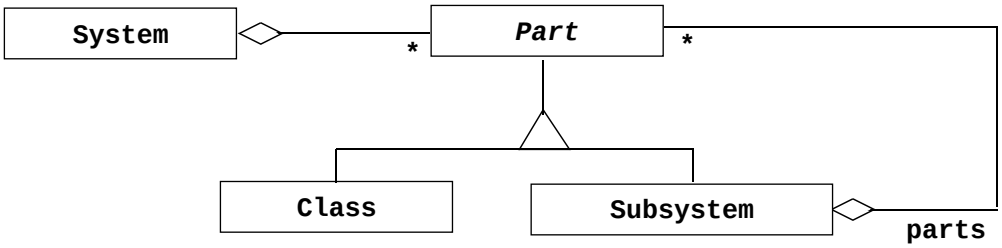


FIGURE 108. Subsystem decomposition (UML class diagram).

For example, the accident management system we previously described can be decomposed into a **`DispatcherInterface`** subsystem implementing the user interface for the **`Dispatcher`**, a **`FieldOfficerInterface`** subsystem implementing the user interface for the **`FieldOfficer`**, an **`IncidentManagement`** subsystem implementing the creation, modification, and storage of **`Incidents`**, and a **`Notification`** subsystem implementing the communication between **`FieldOfficer`** terminals and **`Dispatcher`** stations. This subsystem decomposition is depicted in Figure 109 using UML packages.

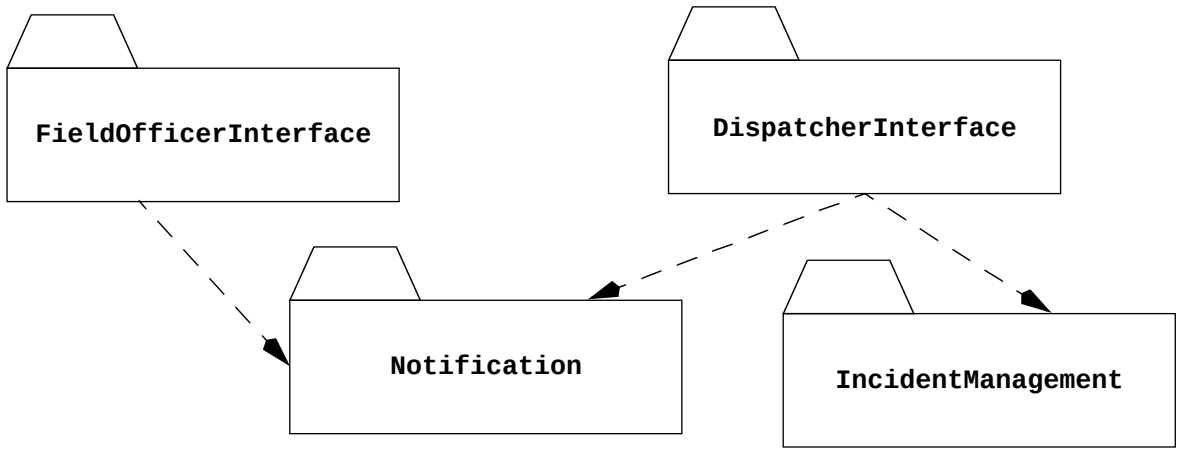


FIGURE 109. Subsystem decomposition for an accident management system (UML class diagram, collapsed view). Subsystems are shown as UML packages. Dashed arrows indicate dependencies between subsystems.

Several programming languages (e.g., Java and Modula-2) provide constructs for modeling subsystems (packages in Java, modules in Modula-2). In other languages, such as C or C++, subsystems are not explicitly modeled, in which case, developers use conventions for grouping classes (e.g., a subsystem can be represented as a directory containing all the files implementing the subsystem). Whether or not subsystems are explicitly represented in the programming language, developers need to carefully document the subsystem decomposition as subsystems are usually realized by different teams.

8.3.2. 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, lookup notification channels, subscribe and unsubscribe to a channel.

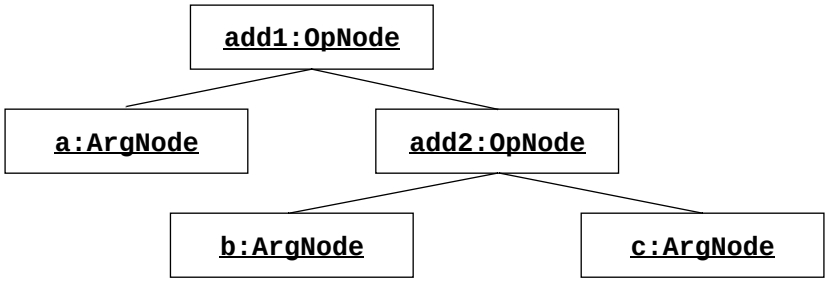
The set of operations of a subsystem that are available to other subsystems form the ***subsystem interface***. The subsystem interface, also referred to as the application programmer interface (API), includes the name of the operations, their parameters, their types, and their return values. System 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 defining the subsystem interfaces, that is, the type of the parameters and the return value of each operation.

The definition of a subsystem in terms of the services it provides help us focus on its interface as opposed to its implementation. A good subsystem interface should provide as little information about its implementation. This allows us to minimize the impact of change when we revise the implementation of a subsystem. More generally, we want to minimize the impact of change by minimizing the dependencies among subsystems.

8.3.3. Coupling and coherence

Coupling is the strength of dependencies between two subsystems. If two subsystems are loosely coupled, they are relatively independent, and thus, modifications to one of the subsystem will have little impact on the other. If two subsystems are strongly coupled, modifications to one subsystem is likely to have impact on the other. A desirable property of a subsystem decomposition is that subsystems are as loosely coupled as possible. This minimizes the impact that errors or future changes have on the correct operation of the system.

Binary tree representation



Sharing through attributes

```
class OpNode {
   ArgNode left;
   ArgNode right;
   String name;
}
class ArgNode {
   String name;
}
```

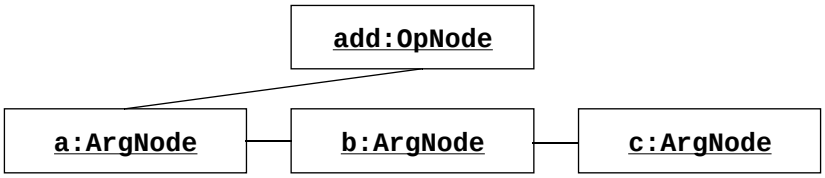
Sharing through operations

```
class OpNode {
   Enumeration getArguments();
   String getName();
}

class ArgNode {
   String getName();
}
```

FIGURE 110. Example of coupling reduction (UML object diagram and Java declarations). This figure shows a parse tree for the expression “a + b + c”. The left column shows the interface of the **OpNode** class when sharing through attributes. The right column shows the interface of **OpNode** when sharing through operations. Figure 111 shows the changes for each case when a linked list is selected instead.

Linked list representation



Sharing through attributes

```
class OpNode {
   ArgNode first;
   ArgNode left;
   ArgNode right;
   String name;
}
class ArgNode {
   String name;
   ArgNode next;
}
```

Sharing through operations

```
class OpNode {
   Enumeration getArguments();
   String getName();
}

class ArgNode {
   String getName();
}
```

FIGURE 111. Example of coupling reduction (UML object diagram and Java declarations). This figure shows the impact of changing the parse tree representation of Figure 110 to a linked list. In the left column, where data is shared through attributes, four attributes need to change (changes indicated in *italics*). In the right column, where data is shared through operations, the interface remains unchanged.

Consider, for example, a compiler in which a parse tree is produced by the syntax analysis subsystem and handed over to the semantic analysis subsystem. Both subsystems access and modify the parse tree. An efficient way for sharing large amounts of data is to allow both subsystems to access the nodes of the tree via attributes. This introduces, however, a tight coupling: both subsystems need to know the exact structure of the parse tree and its invariants. Figure 110 and Figure 111 show the effect of changing the parse tree data structure for two cases: the left column shows the class interfaces when attributes are used for sharing data; the right column shows the class interfaces when operations are used. Since both the syntactic analyzer and the semantic analyzer depend on these classes, both subsystems would need to be modified and re-tested in the case depicted by the left column. In general, sharing data via attributes increases coupling and should be avoided.

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

For example, consider a decision tracking system for recording design problems, discussions, alternative evaluations, decisions, and their implementation in terms of tasks (see Figure 112).

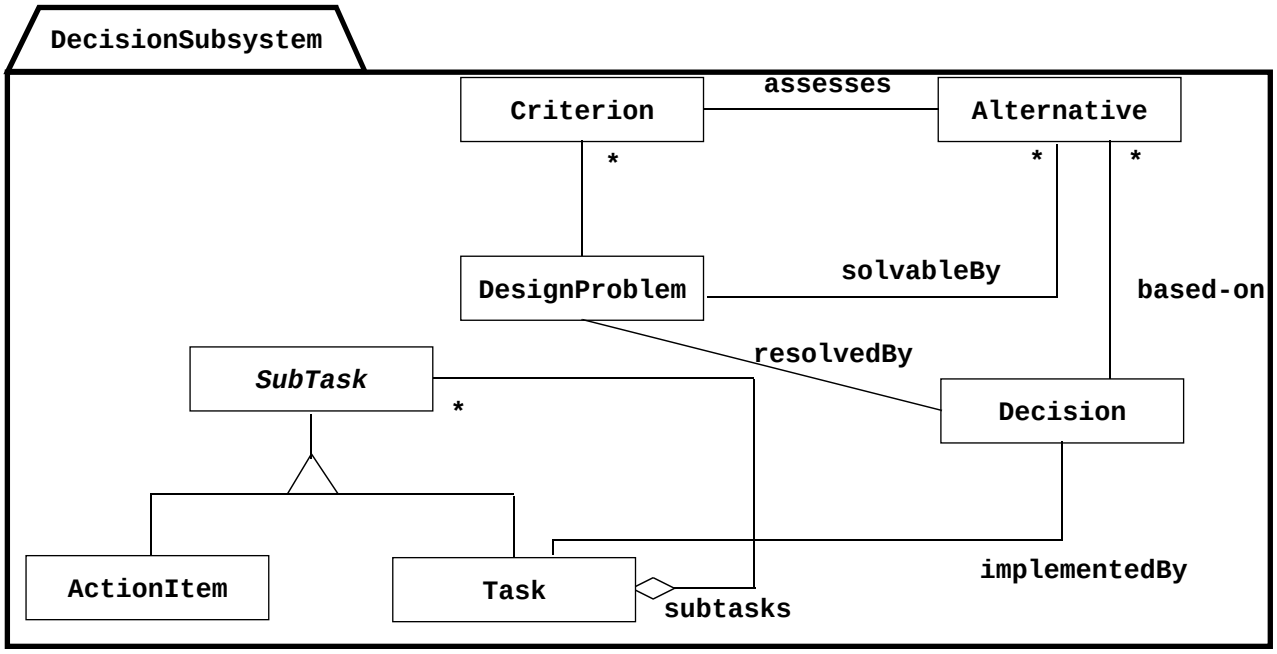


FIGURE 112. Decision tracking system (UML class diagram). The **DecisionSubsystem** has a fairly low coherence: the classes **Criterion**, **Alternative**, and **DesignProblem** have no relationships with **Subtask**, **ActionItem**, and **Task**. Figure 113 depicts a better subsystem decomposition which increases coherence.

DesignProblem and **Alternative** represent the exploration of the design space: we formulate the system in terms of a number of **DesignProblems**, document each **Alternative** they explore. The **Criterion** class represents the qualities in which we are interested. Once we assessed the explored **Alternatives** against desirable **Criteria**, we take **Decisions** and implement them in terms of **Tasks**. **Tasks** are recursively decomposed

into **Subtasks** small enough to be assigned to individual developers. We call atomic tasks **ActionItems**.

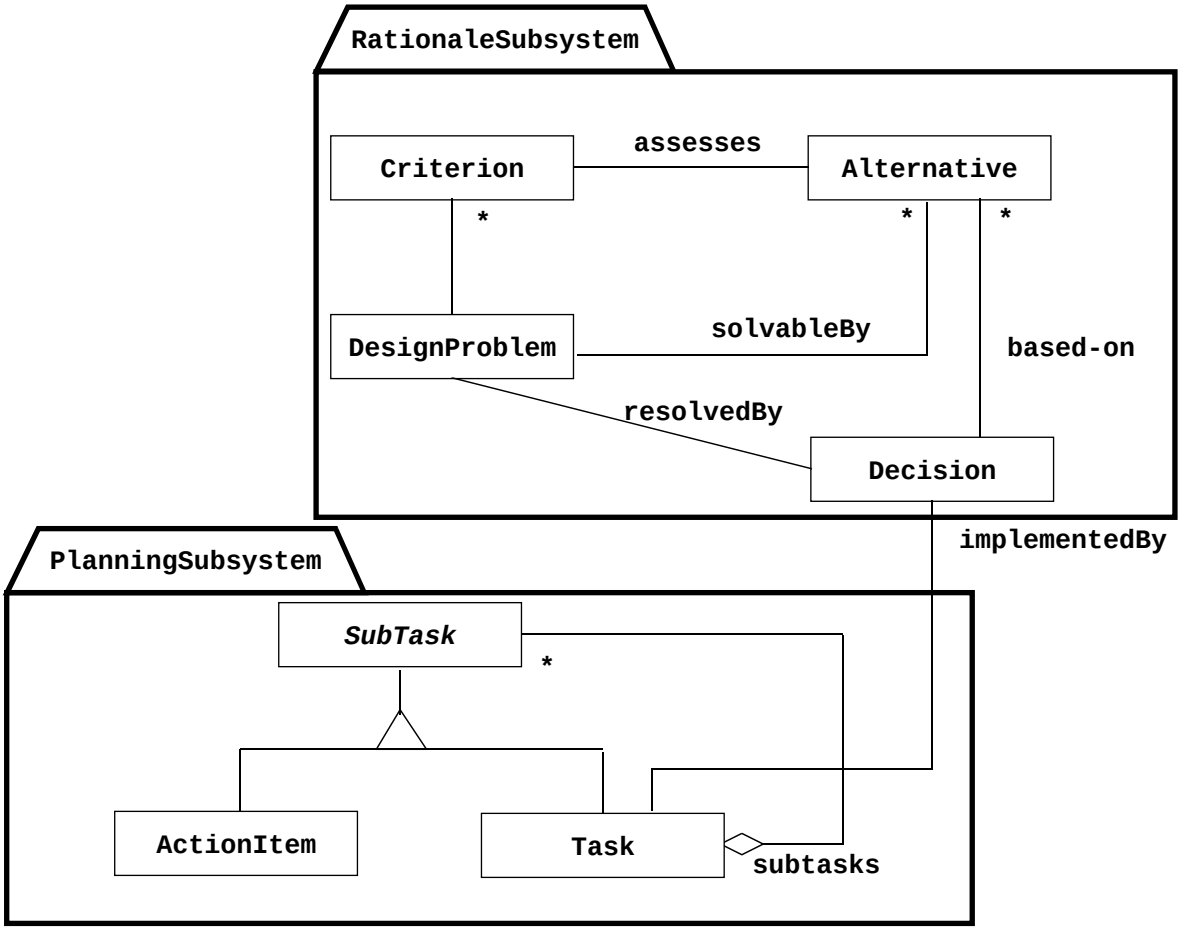


FIGURE 113. Alternative subsystem decomposition for the decision tracking system of Figure 112 (UML class diagram). The coherence of the **RationaleSubsystem** and the **PlanningSubsystem** are higher than the coherence of the original **DecisionSubsystem**. Note also that we also reduced the complexity by decomposition the system into smaller subsystems.

The decision tracking system is small enough that we could lump all these classes into one subsystem called **DecisionSubsystem** (see Figure 112). However, we observe that the class model can be partitioned into two subgraphs, one, called the **RationaleSubsystem**, containing the classes **DesignProblem**, **Alternative**, **Criterion**, and **Decision**, the other, called the **PlanningSubsystem** containing **Task**, **Subtask**, and **ActionItem** (see

Figure 113). Both subsystems have a better coherence than the original **`DecisionSubsystem`**. Moreover, the resulting subsystems are smaller than the original subsystem: we reduced complexity. The coupling between the new subsystems is relatively low, as there is only one association between the two subsystems.

In general, there is a trade-off between coherence and coupling. We can always increase coherence by decomposing the system into smaller subsystems. However, this also increases coupling as the number of interfaces increases. A good heuristic is that developers can deal with 7±2 concepts at any one level of abstraction. If there are more than 9 subsystems at any given level of abstraction or if there is a subsystem providing more than 9 services, you should consider a revision of the decomposition. By the same token, the number of layers should not be more than 7±2. In fact, many good systems design can be done with just 3 layers.

8.3.4. Layers and partitions

The goal of system design is to manage complexity by dividing the system into smaller, manageable pieces. This can be done by a divide and conquer approach, where we recursively divide parts until they are simple enough to be handled by one person or one team. Applying this approach systematically leads to a hierarchical decomposition in which each subsystem, or ***layer***, provides higher level services using services provided from lower level subsystems (see Figure 114). Each layer can only depend on lower level layers and has no knowledge of the layers above it. In a ***closed architecture***, each layer can only depend on the layers immediately below it. In an ***open architecture***, a layer can also access layers at deeper levels.

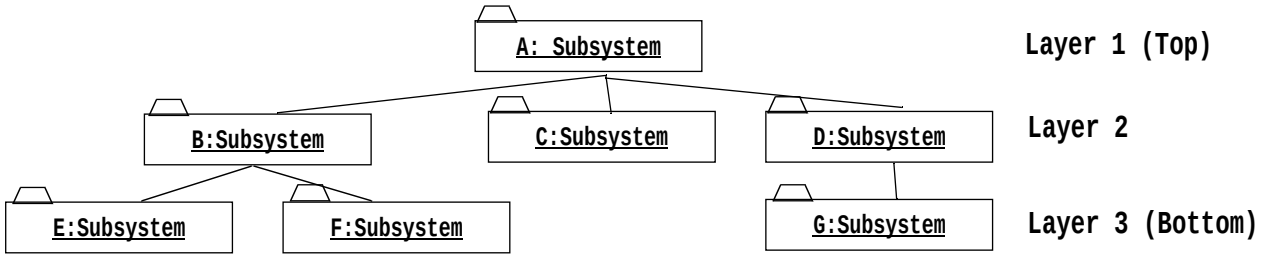


FIGURE 114. Subsystem decomposition of a system into 3 layers (UML object diagram). A subset from a layered decomposition that includes at least one subsystem from each layer is called a vertical slice. For example, the subsystems A, B and E constitute a vertical slice, whereas the subsystems D and G do not.

An example of a closed architecture is the Reference Model of Open Systems Interconnection (in short, the OSI model) is composed of seven layers [Tanenbaum, 1996]. Each layer is responsible for performing a well defined function. In addition, each layer provides its services by using services by the layer below. The **Physical** layer represents the

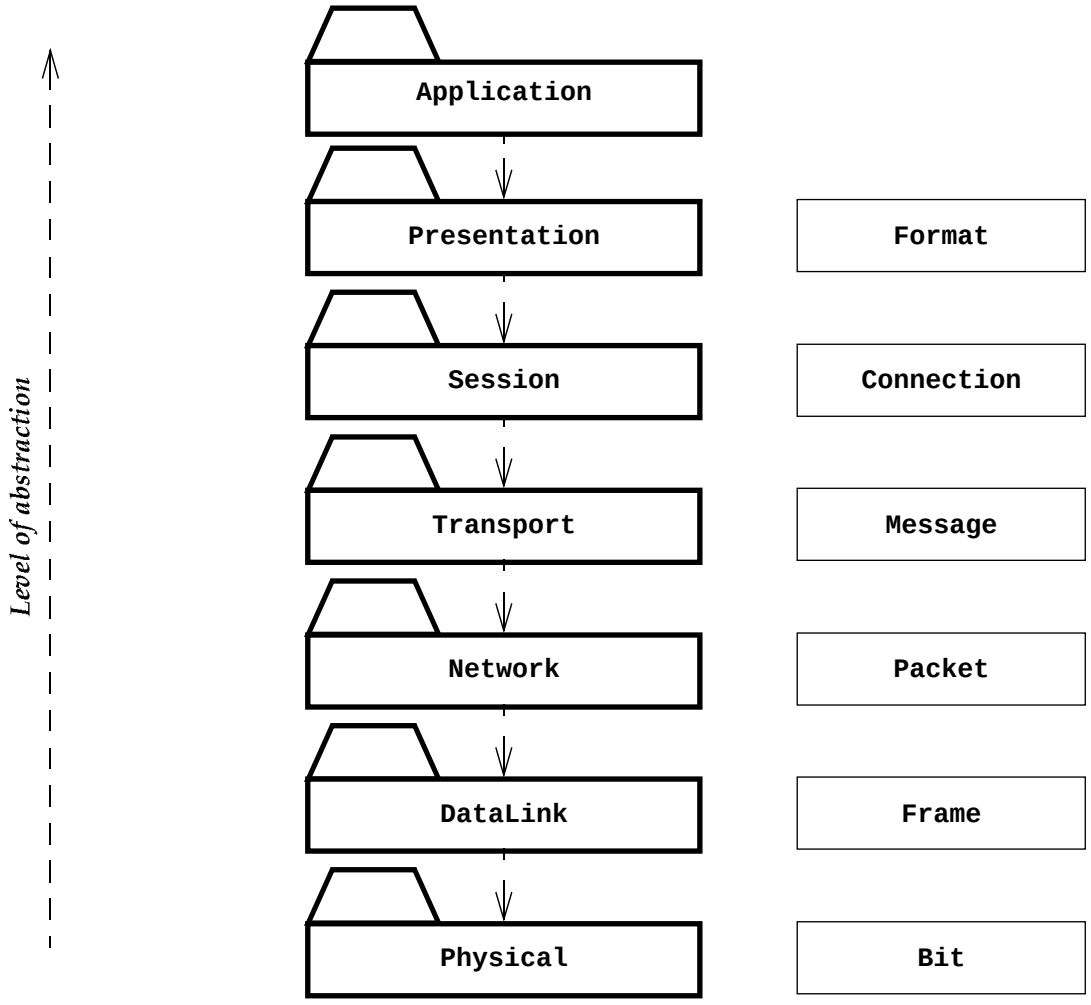


FIGURE 115. An example of closed architecture: the OSI model (UML class diagram). The OSI model decomposes network services into seven layers, each responsible for a different level of abstraction.

hardware interface to the network. It is responsible for transmitting bits over a communication channels. The **DataLink** layer is responsible for transmitting data frames without error using the services of the **Physical** layer. The **Network** layer is responsible for

transmitting and routing packets within a network. The **`Transport`** layer is responsible for ensuring that the data is reliably transmitted from end to end. The **`Transport`** layer is the interface Unix programmers see when transmitting information over TCP/IP sockets between two processes. The **`Session`** layer is responsible for the initialization of a connection, including authentication. The **`Presentation`** layer performs data transformation services, such as byte swapping or encryption. The **`Application`** layer is the system you are designing (unless you are building an operating system or protocol stack). The application layer can also consist of layered subsystems.

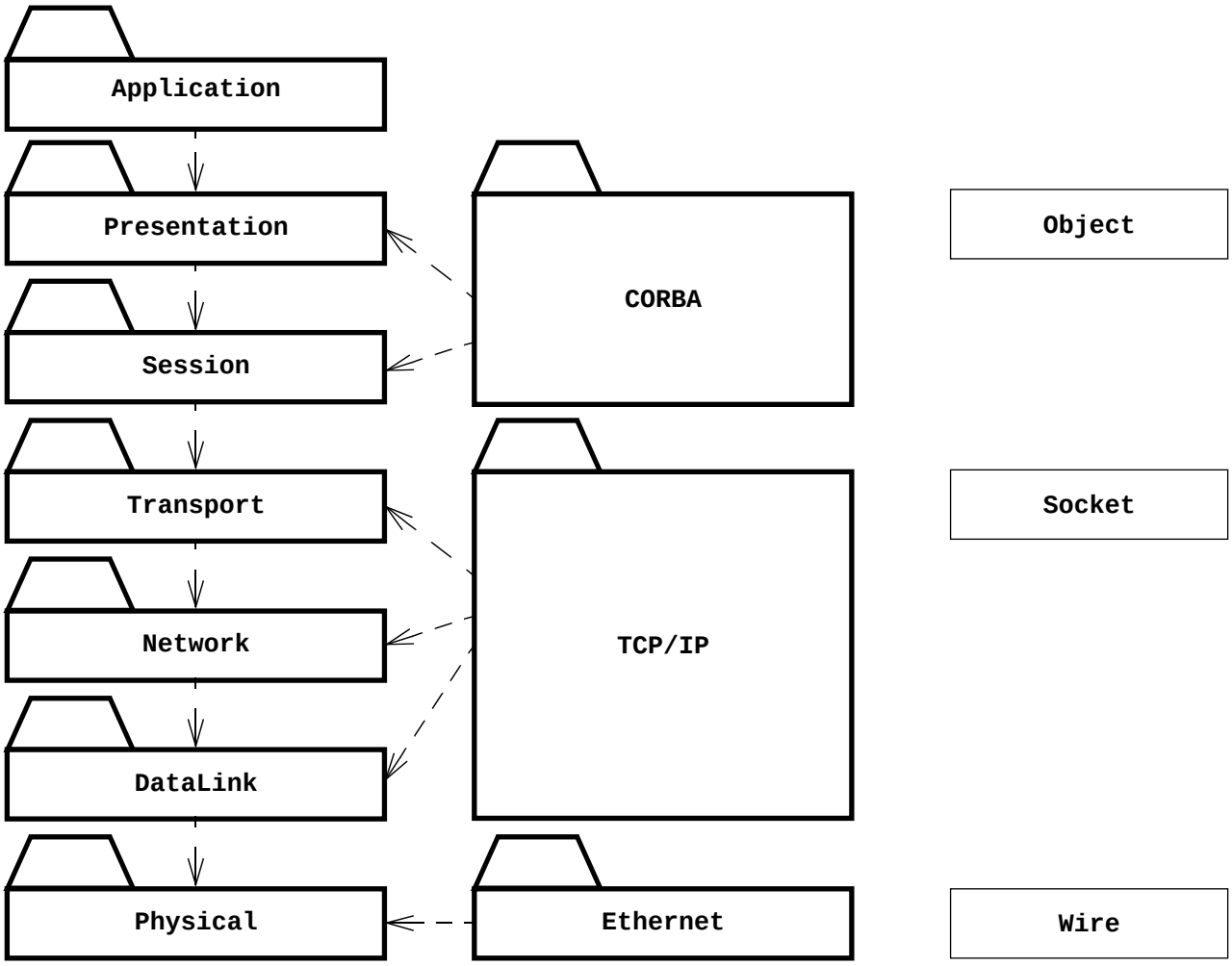


FIGURE 116. An example of closed architecture (UML class diagram). CORBA enables the access of objects implemented in different languages on different hosts. CORBA effectively implements the **`Presentation`** and **`Session`** layers of the OSI stack.

Until recently, only the four bottom layers of the OSI model were well standardized. Unix and many desktop operating systems, for example, provide interfaces to TCP/IP which implements the **`Transport`**, **`Network`**, and **`Datalink`** layers. As an application developer, you still needed to fill the gap between the **`Transport`** layer and the **`Application`** layer. With the growing number of distributed applications, this gap motivated the development of middleware such as CORBA [OMG, 1995] and Java RMI [RMI, 1998]. CORBA and Java RMI enable you to access remote objects transparently, by sending messages to them as you send messages to local objects, effectively implementing the **`Presentation`** and **`Session`** layers (see Figure 116).

An example of an open architecture is the Motif user interface toolkit for X11 [Nye et. al, 1992]. The lowest layer, **`Xlib`** provides basic drawing facilities and defines the concept of window. **`Xt`** provides tools for manipulating user interface objects, called widgets, using services from **`Xlib`**. **`Motif`** is a widget library which provides a wide range of facilities, from buttons to geometry management. Motif is built on top of **`Xt`** but also accesses **`Xlib`** directly. Finally, an application using **`Motif`**, such as a window manager, can access all three layers. **`Motif`** has no knowledge of the window manager and **`Xt`** has no knowledge of **`Motif`** or of the application. Many other user interface toolkits for X11 have open architectures. The openness of the architecture allows developers to bypass the higher level layers in case of performance bottleneck.

Closed layered architectures have desirable properties: they lead to low coupling between subsystems and subsystems can be integrated and tested incrementally. Each level, however, introduces a speed and storage overhead which may make it difficult to meet nonfunctional requirements. Also, adding functionality to the system in later revisions may prove difficult, especially when the additions were not anticipated. In practice, a system is rarely decomposed into more than three to five layers.

Another approach to dealing with complexity is to ***partition*** the system into peer subsystems, each responsible for a different class of services. For example, an onboard system for a car could be decomposed into a travel service, giving real time directions to the driver, an individual preferences service, remembering a driver's seat position and favorite radio station, and vehicle service, keeping track of the car's gas consumption, repairs, and scheduled maintenance. Each subsystem depends loosely on each other but could often operate in isolation.

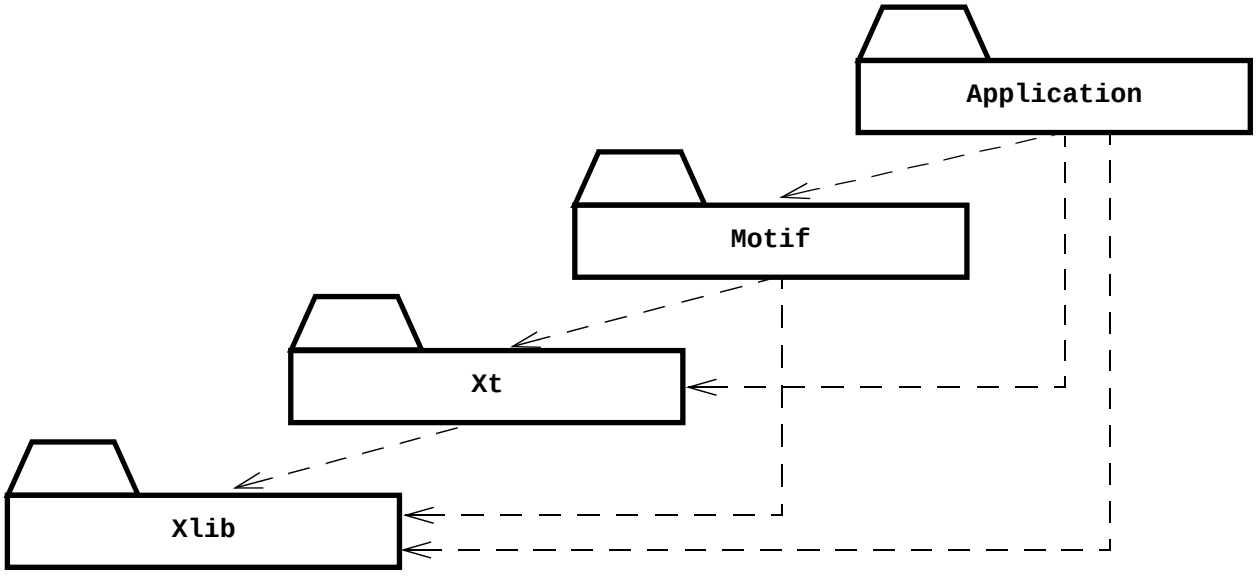


FIGURE 117. An example of open architecture: the OSF/Motif library (UML class diagram, packages collapsed). **`Xlib`** provides low level drawing facilities. **`Xt`** provides basic user interface widget management. **`Motif`** provides a large number of sophisticated widgets. The **`Application`** can access each of these layers independently.

In general, a subsystem decomposition is the result of both partitioning and layering. We first partition the system into top level subsystems which are responsible for specific functionality or which run on a specific hardware node. Each of the resulting subsystems are, if complexity justifies it, decomposed into lower and lower level layers until they are simple enough to be implemented by a single developer. Each subsystem adds a certain processing overhead due to its interface with other systems. Excessive partitioning or layering can lead to increased complexity.

8.3.5. Software architecture

As the complexity of systems increases, the specification of the system decomposition is critical. On the one hand, it is often used as a management tool for distributing work to different teams and team reorganization is expensive. On the other hand, it is difficult to

modify or correct a weak decomposition once development has started as most subsystem interfaces have to change. In recognition of the importance of this problem, the concept of ***software architecture*** has emerged. A software architecture includes the system decomposition, the global control flow, error handling policies and inter subsystem communication protocols [Shaw & Garlan, 1996].

In this section, we describe a few sample architectures that can be used for different types of systems. This is by no means a systematic or thorough exposition of the subject. Rather, we aim to provide you with a few representative examples and refer you to the literature for more details.

Repository architecture

In the repository architecture (see Figure 118), subsystems access and modify data from a single data structure called the central ***repository***. Subsystems are relatively independent and interact only through the central data structure. Control flow can be dictated either by the central repository (e.g., triggers on the data invoke peripheral systems) or by the subsystems (e.g., independent flow of control and synchronization through locks in the repository).

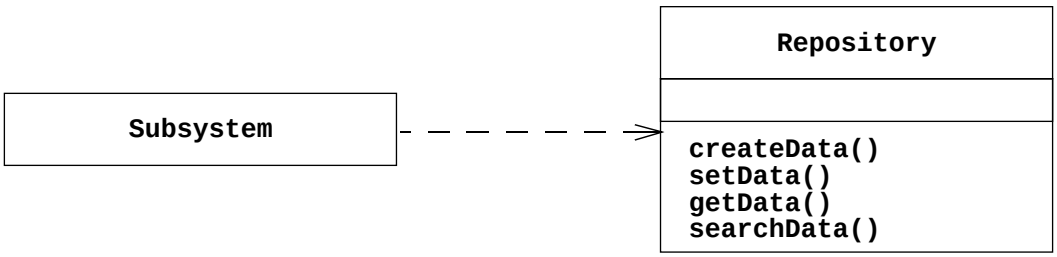


FIGURE 118. Repository architecture (UML class diagram). Every subsystem only depends on a central datastructure called the repository. The repository in turn, has no knowledge of the other subsystems.

The repository architecture is typical for database management systems, such as a payroll system or a bank system. The central location of the data makes it easier to deal with concurrency and integrity issues between subsystems. Modern compilers and software development environments also follow a repository architecture (see Figure 119). The

different subsystems of a compiler access and update a central parse tree and a symbol table. Debuggers and syntactical editors access the symbol table as well.

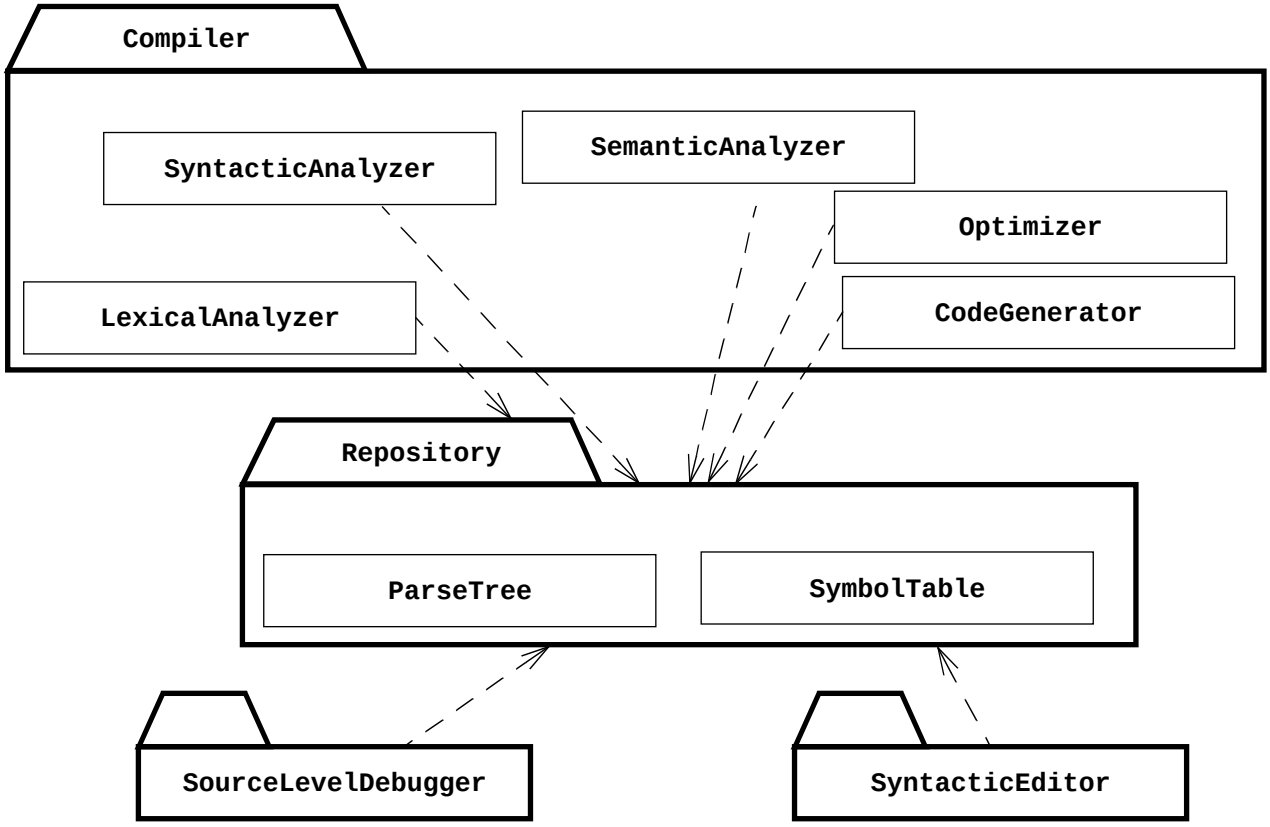


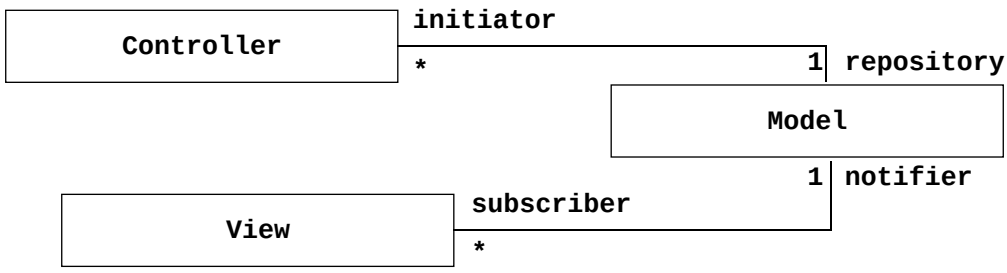
FIGURE 119. An instance of the repository architecture (UML Class diagram). A modern compiler incrementally generates a parse tree and a symbol table that can be later used by debuggers and syntactic editors.

The repository subsystem can also be used for implementing the global control flow. In the compiler example of Figure 119, each individual tool (e.g., the compiler, the debugger, and the editor) is invoked by the user. The repository only ensures that concurrent accesses are serialized. Conversely, the repository can be used to invoke the subsystems based on the state of the central datastructure. These systems are called blackboard systems. The HEARSAY II Speech understanding system [Erman 1980], one of the first blackboard system, selected tools to invoke based on the current state of the blackboard.

Repository architectures are well suited for applications with constantly changing complex data processing tasks. Once a central repository is well defined, we can easily add new services in the form of additional subsystems. The main disadvantage of repository systems is that the central repository can quickly become a bottleneck, both from a performance aspect and a modifiability aspect.

Model/View/Controller

In the Model / View / Controller (MVC) architecture (see Figure), subsystems are classified into three different kinds: ***model subsystems*** are responsible for maintaining domain knowledge, ***view subsystems*** are responsible for displaying it to the user, and ***controller subsystems*** are responsible for managing the sequence of interactions with the user. The model subsystems are developed such that they do not depend on any view or controller subsystem. Changes in their state is propagated to the view subsystem via a subscribe / notify protocol. The MVC architecture is a special case of repository architecture where Model implements the central datastructure and control objects dictate the control flow.



Model / View / Controller architecture (UML class diagram). The **`Controller`** gathers input from the user and sends messages to the **`Model`**. The **`Model`** maintains the central datastructure. The **`View`**(s) display the **`Model`** and is notified (via a subscribe / notify protocol) whenever the **`Model`** is changed.

For example, Figure 120 and Figure 121 illustrate the sequence of events that occur in a MVC architecture. Figure 120 displays two views of a file system. The bottom window lists the content of the **`Comp-Based Software Engineering`** folder, including the file **`9DesignPatterns2.ppt`**. The top window displays information about this file. The name of

the file **`9DesignPatterns2.ppt`** appears in three places: in both windows and in the title of the top window.

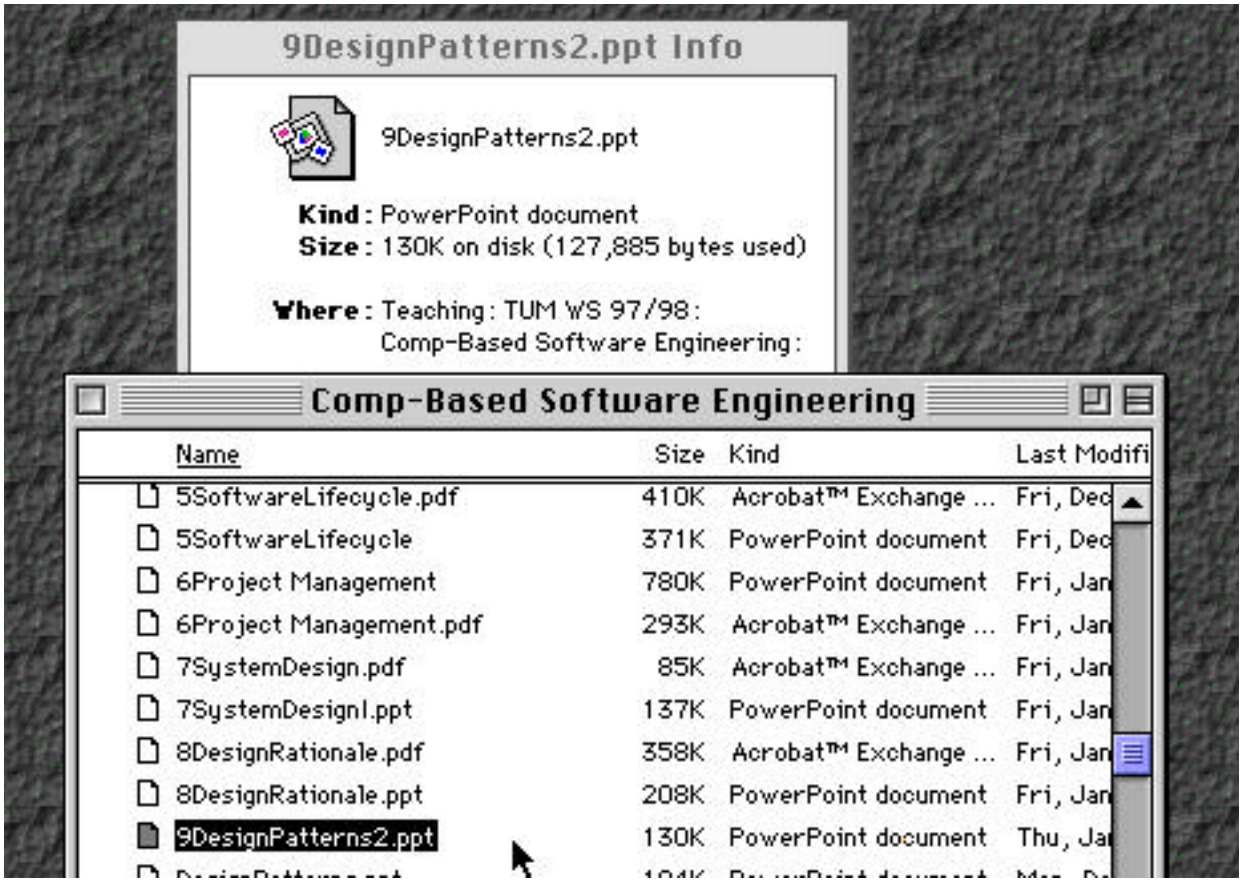


FIGURE 120. Use of the MVC architecture in the Macintosh file system. The “model” is the filename **`9DesignPAtterns2.ppt`**. One “view” is a window titled The **`Comp-Based Software Engineering`** which displays the content of a folder containing the file **`9DesignPatterns2.ppt`**. The other “view” is window called **`9DesignPatterns2.ppt Info`** which displays information related to the file. If the file name is changed, both views will immediately be updated by the “controller” (the Macintosh file system).

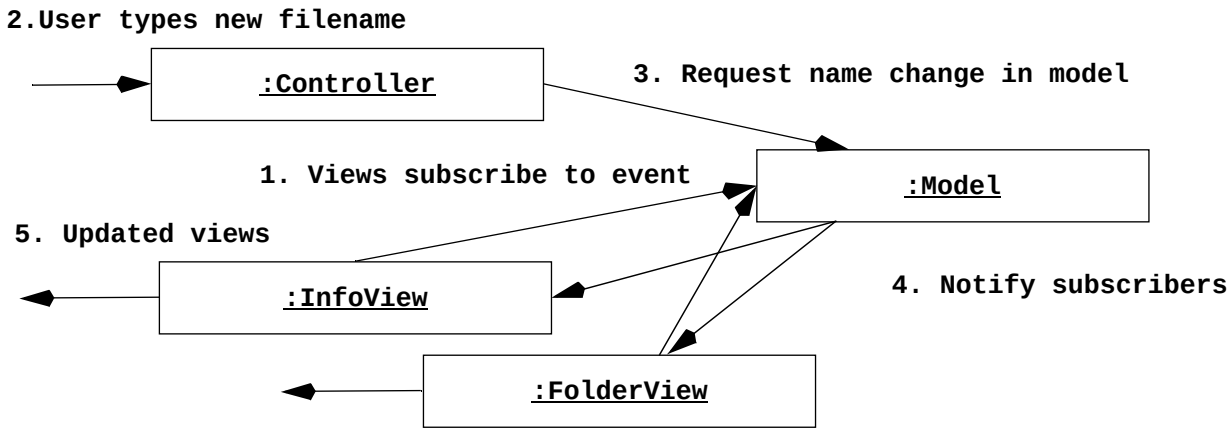


FIGURE 121. Sequence of events in the Model / View / Control architecture (UML collaboration diagram).

Assume now that we change the name of the file to **`9DesignPatterns.ppt`**. Figure 121 shows the sequence of events:

1. The **`InfoView`** and the **`FolderView`** both subscribe for changes to the File models they display (when they are created).
2. The user types the new name of the file.
3. The **`Controller`**, the object responsible for interacting with the user during file name changes, sends a request to the Model.
4. The **`Model`** changes the file name and notifies all subscribers of the change.
5. Both **`InfoView`** and **`FolderView`** are updated, the user sees a consistent change.

The rationale between the separation of Model, View, and Controller is that user interfaces, i.e., the View and the Controller, are much more often subject to change than domain knowledge, i.e., the Model. Moreover, by removing any dependency from the Model on the View with the subscription / notification protocol, changes in the views (user interface) do not have any effect on the model subsystems. In the example of Figure 120, we could add a unix-style shell view of the file system without having to modify the file system. We described a similar decomposition in Chapter 7, *Requirements Analysis* when we identified entity, interface, and control objects. This decomposition is also motivated by the same considerations about change.

MVC architectures are well suited for interactive systems, especially when multiple views of the same model are needed. MVC can be used for maintaining consistency across

distributed data, however, it introduces the same performance bottleneck as for other repository architectures.

Client/server architecture

In the client / server architecture (see Figure 122), a subsystem, the ***server***, provides services to instances of other subsystems called the ***clients***, which are responsible for interacting with the user. The request for a service is usually done via a remote procedure call mechanism or with the help of middleware such as CORBA and Java RMI. Control flow in the clients and the servers is independent except for synchronization to manage requests or receiving results.

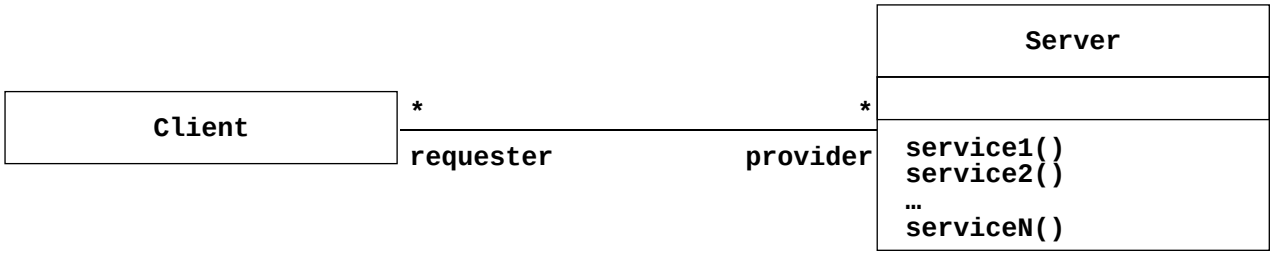


FIGURE 122. Client / server architecture (UML class diagram). Clients request services from one or more servers. The server has no knowledge of the client. The client / server architecture is a generalization of the repository architecture.

An information system with a central database is an example of a client / server architecture. The clients are responsible for receiving inputs from the user, performing range checks, and initiating a database transaction once all necessary data is collected. The server is then responsible for performing the transaction and guaranteeing the integrity of the data. In this case, a client / server architecture is a special case of the repository architecture where the central datastructure is managed by a process. Client / server systems, however, are not

restricted to a single server. In the world wide web, a single client can easily access data from thousands of different servers.

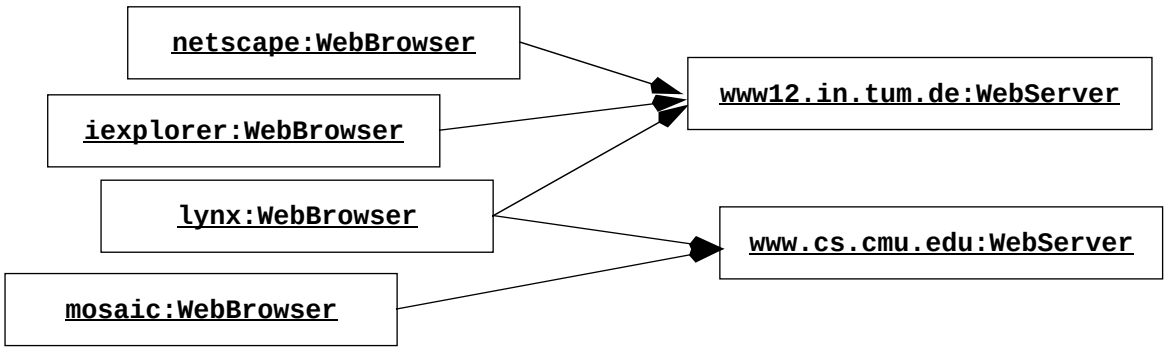


FIGURE 123. An instance of the client/server architecture (UML object diagram). Web browsers (i.e., clients) can access any number of http servers

Client/server architectures are well suited for distributed systems which manage large amounts of data.

Peer-to-peer architecture

A peer-to-peer architecture (see Figure 124) is a generalization of the client/server architecture in which subsystems can act both as client or as servers, in the sense that each subsystem can request and provide services. The control flow within each subsystem is independent from the others except for synchronizations on requests.

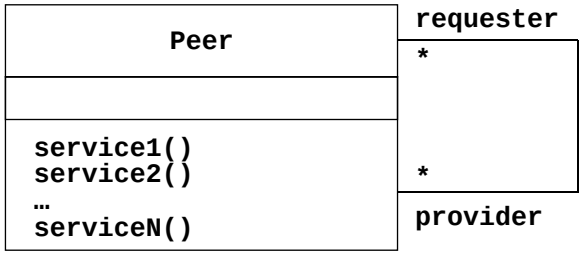


FIGURE 124. Peer-to-peer architecture (UML class diagram). Peers can request services from and provide services to other peers.

An example of a peer-to-peer architecture is a database which, on the one hand, accepts requests from the application and, on the other hand, sends notifications to the application whenever certain data are changed. Peer-to-peer systems are much more difficult to design than client/server systems. They introduce the possibility of deadlocks and complicate the control flow.

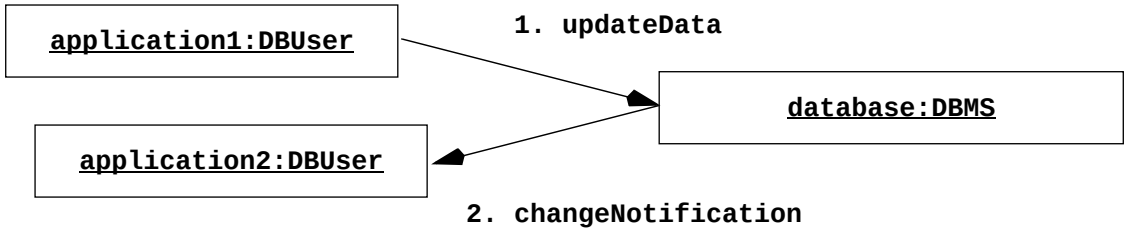


FIGURE 125. An example of peer-to-peer architecture (UML collaboration diagram). The database server can both process requests from and send notifications to applications.

Pipe and filter architecture

In the pipe and filter architecture (see Figure 126), subsystems process data received from a set of inputs and send results to other subsystems via a set of outputs. The subsystems are called ***filters*** and the associations between the subsystems are called ***pipes***. Each filter only knows the content and the format of the data received on the input pipes, not the filters that produced them. Each filter is executed concurrently and synchronization is done via the

pipes. The pipe and filter architecture is modifiable: filters can be substituted for others or reconfigured to achieve a different purpose.

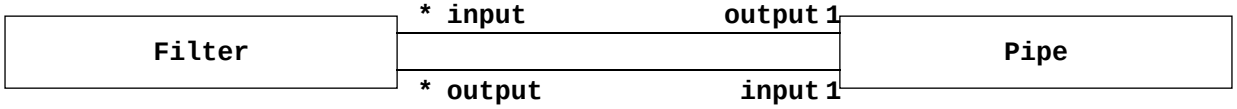


FIGURE 126. Pipe and filter architecture (UML class diagram). A **`Filter`** can have many inputs and outputs. A **`Pipe`** connects one of the outputs of a **`Filter`** to one of the inputs of another **`Filter`**.

The best known example of pipe and filter architecture is the Unix shell[]. Most filters are written such that they read their input and write their results on standard pipes. This enables a Unix user to combine them in many different ways. Figure 127 shows an example made out of four filters. The output of **`ps`** (process status) is fed into **`grep`** (search for a pattern) to get rid off all the processes that are not owned by a specific user. The output of **`grep`** (i.e., the processes owned by the user) is then sorted lexicographically by **`sort`** and then sent to **`more`**. **`more`** is a filter that displays its input to a terminal, one screen at a time.

```
% ps auxwww | grep dutoit | sort | more

dutoit   19737  0.2  1.6 1908 1500 pts/6    O 15:24:36  0:00 -tcsh
dutoit   19858  0.2  0.7  816  580 pts/6    S 15:38:46  0:00 grep dutoit
dutoit   19859  0.2  0.6  812  540 pts/6    O 15:38:47  0:00 sort
```

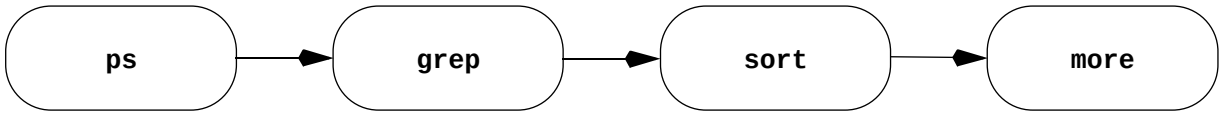


FIGURE 127. An instance of the pipe and filter architecture (unix command and UML activity diagram).

Pipe and filter architectures are suited for systems that apply transformations to streams of data without intervention by users. They are not suited for systems which require more complex interactions between components, such as an information management system or an interactive system.

8.4. From objects to subsystems

System design consists of transforming the requirements analysis model into the design model that takes into account the nonfunctional requirements and constraints described in the problem statement and the requirements analysis document. In Section 8.3, we focused on subsystem decompositions and their properties. In this section, we describe the activities that are needed to ensure that a subsystem decomposition addresses all the nonfunctional requirements and prepares for taking into accounts constraints during the implementation phase. We illustrate these activities with an example, MyTrip, a route planning system for car drivers, throughout this section. This will provide you with more concrete knowledge of system design concepts.

We start with the requirements analysis model from MyTrip. We then

- identify design goals from the nonfunctional requirements (Section 8.4.2)
- design an initial subsystem decomposition (Section 8.4.3)
- map subsystems to processors and components (Section 8.4.4)
- decide storage (Section 8.4.5)
- access control policies (Section 8.4.6)
- select a control flow mechanism (Section 8.4.7), and
- identify boundary conditions (Section 8.4.8).

In Section 8.4.9, we examine issues related to stabilizing the system design while anticipating change. Finally, in Section 8.4.10, we describe how the system design model is reviewed.

But first, we describe the requirements analysis model we use as a starting point for the system design of MyTrip.

8.4.1. Starting point: requirements analysis model for a route planning system

Using MyTrip, a driver can plan a trip from a home computer by contacting a trip planning service on the web (**`PlanTrip`** in Figure 128). The trip is saved for later retrieval on the server. The trip planning service must support more than one driver.

Use case name	**`PlanTrip`**
Entry condition	1. The **`Driver`** activates her home computer and logs into the trip planning web service.
Flow of events	2. Upon successful login, the **`Driver`** enters constraints for a trip as a sequence of destinations. 3. Based on a database of maps, the planning service computes the shortest way visiting the destinations in the specified order. The result is a sequence of segments binding a series of crossings and a list of directions. 4. The **`Driver`** can revise the trip by adding or removing destinations.
Exit condition	5. The **`Driver`** saves the planned trip by name in the planning service database for later retrieval.

FIGURE 128. **`PlanTrip`** use case of the MyTrip system.

The driver then goes to the car and starts the trip, while the onboard computer gives directions based on trip information from the planning service and her current position indicated by an onboard GPS system (**`ExecuteTrip`** in Figure 129).

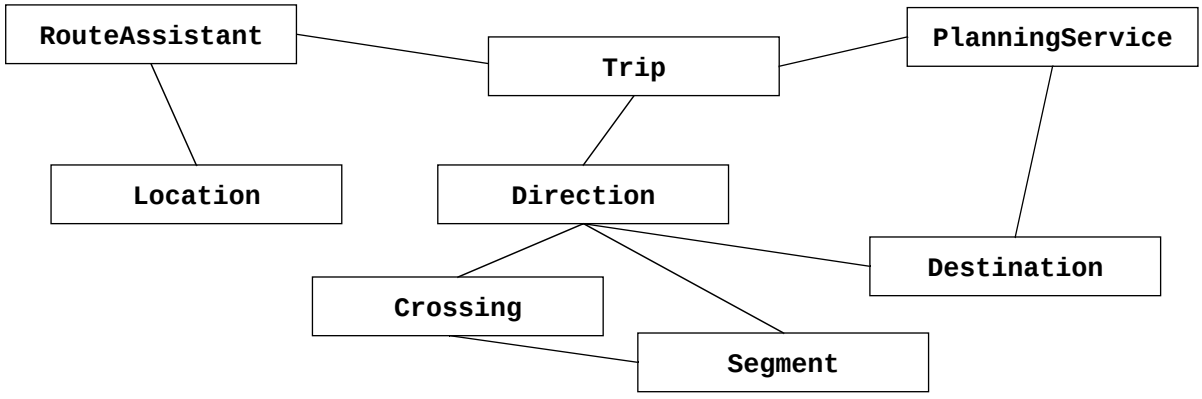
Use case name	**`ExecuteTrip`**
Entry condition	1. The **`Driver`** turns on her car and logs into the onboard route assistant.
Flow of events	2. Upon successful login, the **`Driver`** specifies the planning service and the name of the trip to be executed. 3. The onboard route assistant obtains the list of destinations, directions, segments, and crossings from the planning service.

FIGURE 129. **`ExecuteTrip`** use case of the MyTrip system.

	4. Given the current position, the route assistant provides the driver with the next set of directions.
Exit condition	5. The **`Driver`** arrives to destination and shuts down the route assistant.

FIGURE 129. **`ExecuteTrip`** use case of the MyTrip system.

We perform the requirements analysis for the MyTrip system following the techniques outlined in Chapter 7, *Requirements Analysis* and obtain the model in Figure 130.



`Crossing`	A **`Crossing`** is a geographical point were a driver can choose between several **`Segments`**.
`Destination`	A **`Destination`** represents a location where the driver wishes to go.
`Direction`	Given a **`Crossing`** and an adjacent **`Segment`**, a **`Direction`** describes in natural language terms how to steer the car onto the given **`Segment`**.
`Location`	A **`Location`** is the position of the car as known by the onboard GPS system or the number of turns of the wheels.
`PlanningService`	A **`PlanningService`** is a web server that can supply a trip linking a number of destinations in the form of a sequence of crossings and segments.

FIGURE 130. Requirements analysis model for the MyTrip route planning and execution.

RouteAssistant	A **RouteAssistant** gives **Directions** to the driver given the current **Location** and upcoming **Crossing**.
Segment	A **Segment** represents the road between two **Crossings**.
Trip	A **Trip** is a sequence of **Directions** between two **Destinations**.

FIGURE 130. Requirements analysis model for the MyTrip route planning and execution.

In addition, during requirements elicitation, our client specified the following nonfunctional requirements for MyTrip.

Nonfunctional requirements for MyTrip

1. MyTrip is in contact with the PlanningService via a wireless modem. It can be assumed that the wireless modem functions properly at the initial destination.
2. Once the trip has been started, MyTrip should give correct directions even if modem fails to maintain a connection with the PlanningService.
3. MyTrip should minimize connection time to reduce operation costs.
4. Replanning is possible only if the connection to the PlanningService is possible.
5. The PlanningService can support at least 50 different drivers, and 1000 trips

8.4.2. 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.

For example, in the light of the nonfunctional requirements for MyTrip described in Section 8.4.1, we identify ***reliability*** and ***fault tolerance to connectivity loss*** as design goals. We then identify ***security*** as a design goal as numerous drivers will have access to the same trip planning server. We add ***modifiability*** as a design goal as we want to provide the ability

for drivers to select a trip planning service of their choice. The following box summarizes the design goals we identified.

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 another driver's trips [deduced from application domain].
- **Modifiability:** MyTrip should be modifiable to use different routing services [anticipation of change by developers].

In general, we can select design goals from a long list of highly desirable qualities. Tables 33 through 37 list a number of possible design criteria. These criteria are organized into five groups: *performance*, *robustness*, *cost*., *maintenance*, and end *user criteria.* Performance, robustness, and end user criteria are usually specified in the requirements or inferred from the application domain. Cost and maintenance criteria are dictated by the customer and the supplier.

Performance criteria (Table 33) include the speed and space requirements imposed on the system. Should the system be responsive or should it accomplish a maximum number of tasks? Is memory space available for speed optimizations or should memory be used sparingly?

Table 33 Performance criteria

Design criteria	Definition
Response time	How soon is a user request acknowledged after the request has been issued?
Throughput	How many tasks can the system accomplish in a fixed period of time.
Memory	How much space is required for the system to run?

Dependability criteria (Table 34) determine how much effort should be expended in minimizing system crashes and their consequences. How often can the system crash? How available to the user should the system be? Are there safety issues associated with system crashes? Are there security risks associated with the system environment?

Table 34 Dependability criteria

Design criteria	Definition
Robustness	Ability to survive invalid user input
Reliability	Difference between specified and observed behavior.

Table 34 Dependability criteria

Design criteria	Definition
Availability	Percentage of time system can be used to accomplish normal tasks.
Fault tolerance	Ability to operate under erroneous conditions.
Security	Ability to stand malicious attacks
Safety	Ability to not endanger human lives, even in the presence of errors and failures.

Cost criteria (Table 35) include the cost to develop the system, to deploy it, and to administer it. Note that cost criteria not only include design considerations but managerial ones as well. When the system is replacing an older one, the cost of ensuring backward compatibility or transitioning to the new system has to be taken into account. There are also trade-offs between different types of costs such as development cost, end user training cost, transition costs and maintenance costs. Maintaining backward compatibility with a previous system can add to the development cost while reducing the transition cost.

Table 35 Cost criteria

Design criteria	Definition
Development cost	Money is required to develop the system.
Deployment cost	Money is required to install the system and train the users.
Development cost	Cost for developing the initial system
Backward compatibility	Ability to handle data from previous revisions or from different systems.
Training cost	Training the end user in the use of the new system
Maintenance cost	Cost required for bug fixes and enhancements to the system
Administration cost	Money required to administer the system.

Maintenance criteria (Table 36) determine how difficult it is to change the system after deployment. How easily can new functionality be added? How easily can existing functions be revised? Can the system be adapted to a different application domain? How much effort will be required to port the system to a different platform? These criteria are harder to optimize and plan for as it is seldom clear how long the system will be operational and how successful the project will be.

End user criteria (Table 37) include qualities that are desirable from a users' point of view that have not yet been covered under the performance and dependability criteria. These include usability (how difficult is the software to use and to learn?) and utility (how well

Table 36 Maintenance criteria

Design criteria	Definition
Extensibility	How easy is it to add the functionality or new classes of the system?
Modifiability	How easy is it to change the functionality of the system?
Adaptability	How easy is it to port the system to different application domains?
Portability	How easy is it to port the system to different platforms?
Readability	How easy is it to understand the system from reading the code?
Traceability of requirements	How easy is it to map the code to specific requirements?

does the system the user's work?). Often these criteria do not receive much attention especially when the client contracting the system is distinct from the users of the system.

Table 37 End user criteria

Design criteria	Definition
Utility	How well does the system support the work of the user?
Usability	How easy is it for the user to use the system?

When defining design goals, only a small subset of these criteria can be simultaneously taken into account. It is, for example, unrealistic to develop software that is safe, secure, and cheap. Typically, developers need to prioritize design goals and trade them off against each other as well as against managerial goals as the project runs behind schedule or over budget. Table 38 lists several possible trade-offs.

Table 38 Examples of design goal trade-offs

Trade-off	Rationale
Space vs. speed	If the software does not meet response time or throughput requirements, more memory can be expended to speed up the software (e.g., caching, more redundancy, etc.). If the software does not meet memory space constraints, data can be compressed at the cost of speed.
Delivery time vs. functionality	If the development runs behind schedule, a project manager can deliver less functionality than specified and deliver on time, or deliver the full functionality at a later time. Contract software usually puts more emphasis on functionality while off-the-shelf software projects put more emphasis on delivery date.

Table 38 Examples of design goal trade-offs

Trade-off	Rationale
Delivery time vs. quality	If the testing runs behind schedule, a project manager can deliver the software on time with known bugs (and possible providing a later patch to fix any serious bugs) or to deliver the software later with more bugs fixed.
Delivery time vs. staffing	If development runs behind schedule, a project manager can add resources to the project in order to increase productivity. In most cases, this option is only available early in the project: adding resources usually decreases productivity while new personnel is being trained brought up to date. Note that adding resources will also raise the cost of development.

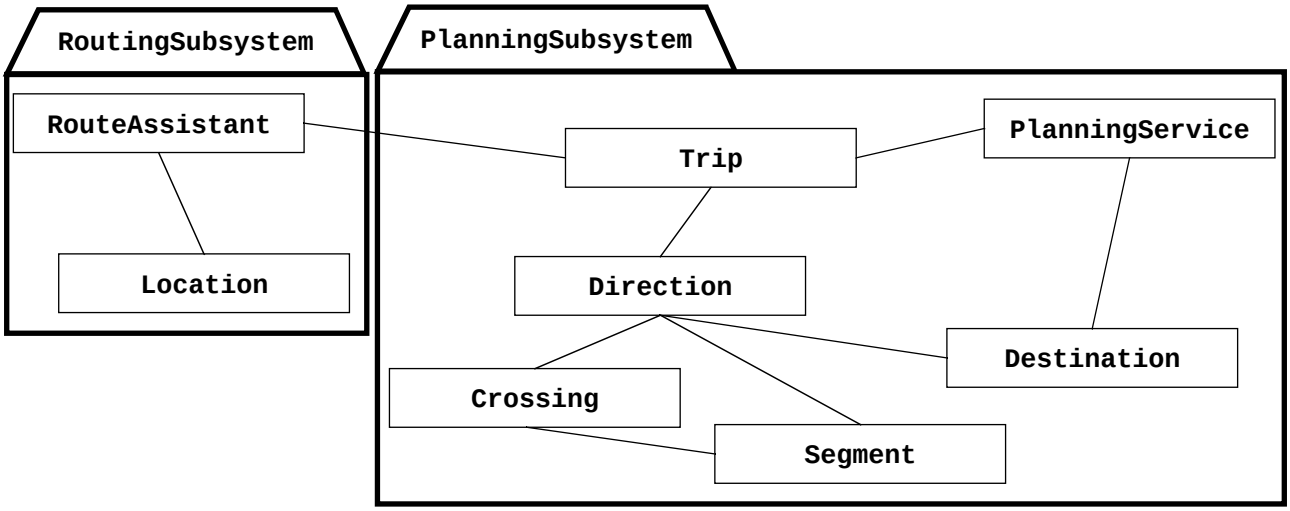
Managerial goals can be traded off against technical goals (e.g., delivery time vs. functionality). Once we have a clear idea of the design goals, we can proceed to design an initial subsystem decomposition.

8.4.3. Identifying subsystems

Finding subsystems during system design has many similarities to finding objects during requirements analysis: it is a volatile activity driven by heuristics. As a result, the object identification techniques we described in Chapter 7, *Requirements Analysis*, such as Abbots lexical rules, are applicable to subsystem identification. Moreover, subsystem decomposition is constantly revised whenever new issues are addressed: subsystems are merged into one subsystem, a complex subsystem is split into parts, some subsystems are added to take care of new functionality. The first iterations over the subsystem decomposition can introduce drastic changes in the system design model. These are often best handled through brainstorming.

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 cases, 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 they are used as a whole to represent a **`Trip`**. We decide to assign them with **`PlanningService`** to the **`PlanningSubsystem`**, while the remainder of the classes are assigned to the **`RoutingSubsystem`**. This leads to only one association crossing subsystem boundaries.

Note that this subsystem decomposition follows a repository architecture in which the **`PlanningSubsystem`** is responsible for the central datastructure.



`PlanningSubsystem`	The **`PlanningSubsystem`** is responsible for constructing a **`Trip`** connecting a sequence of **`Destinations`**. The **`PlanningSubsystem`** is also responsible for responding to replan requests from **`RoutingSubsystems`**.
`RoutingSubsystem`	The **`RoutingSubsystem`** is responsible for downloading a **`Trip`** from the **`PlanningService`** and executing it by giving **`Directions`** to the driver based on its **`Location`**.

FIGURE 131. Initial subsystem decomposition for MyTrip (UML class diagram).

Another heuristic for subsystem identification is to keep functionally related objects together. A starting point is to take the use cases and assign the participating objects that have been identified in each of them 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.

Encapsulating subsystems

Subsystem decomposition reduces the complexity of the solution domain by minimizing dependencies among classes. The Facade pattern [Gamma et al., 1994] allows us to further reduce dependencies between classes by encapsulating a subsystem with a simple, unified interface. For example, in Figure 132, the **`Compiler`** class is a Facade hiding the classes **`CodeGenerator`**, **`Optimizer`**, **`ParseNode`**, **`Parser`**, and **`Lexer`**. The Facade provides access only to the public services offered by the subsystem and hides all other details, effectively reducing coupling between subsystems.

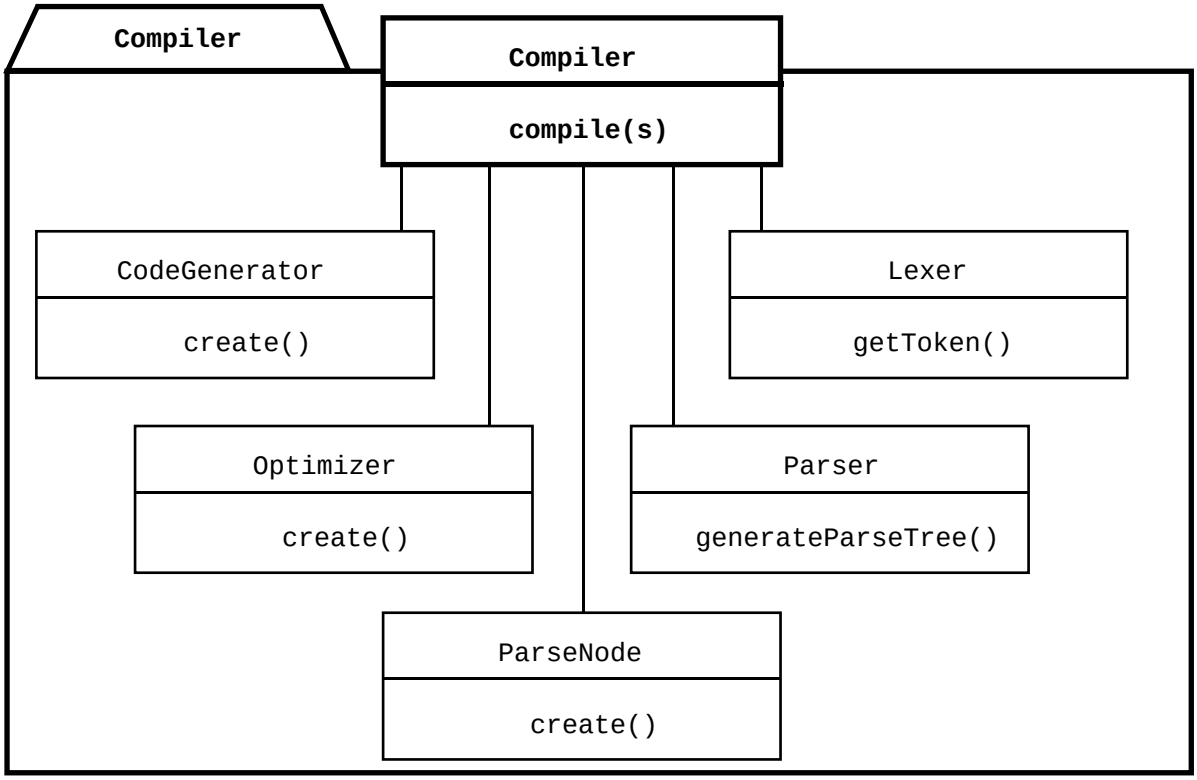


FIGURE 132. An example of Facade pattern (UML class diagram).

Subsystems identified during the initial subsystem decomposition often result from grouping several functionally related classes. These subsystems are good candidates for the Facade patterns and should be encapsulated under one class.

8.4.4. Mapping subsystems to processors and components

Selecting a hardware configuration and a platform

Many systems run on more than one computer and depend, to various extent, on access to an intranet or to the internet. The use of multiple computers can address high performance needs or to interconnect multiple distributed users. Consequently, we need to examine carefully the allocation of subsystems to computers and the design of the infrastructure for supporting communication between subsystems. These computers are modeled as nodes in UML deployment diagrams described in Chapter 2, *Modeling with UML*. Nodes can either represent a specific instances (e.g., myMac) or a class of computers (e.g., **WebServer**).[1] Since the hardware mapping activity has significant impact on the performance and complexity of the system, we perform it early in system design.

Select a hardware configuration also include selecting a virtual machine onto which the system should be built. The virtual machine includes the operating system and any software component that are needed, such as a database management system or a communication package. The selection of a virtual machine reduces the distance between the system and the hardware platform on which it will run. The more functionality the components provide, the less work involved. The selection of the virtual machine, however, may be constrained by the client who often acquired hardware before the start of the project. The selection of a virtual machine may also be constrained by cost considerations: in some cases, it is difficult to estimate whether building a component costs more than buying it.

In MyTrip, we deduce from the requirements that **PlanningSubsystem** and **RoutingSubsystem** run on two different nodes: the former is a web-based service on an Internet host while the second runs on the onboard computer. Figure 133 illustrates the hardware allocation for MyTrip with two nodes called :OnboardComputer and :WebServer.

1. These two cases are distinguished with the usual UML naming convention: underlined names for instances and non underlined names for classes

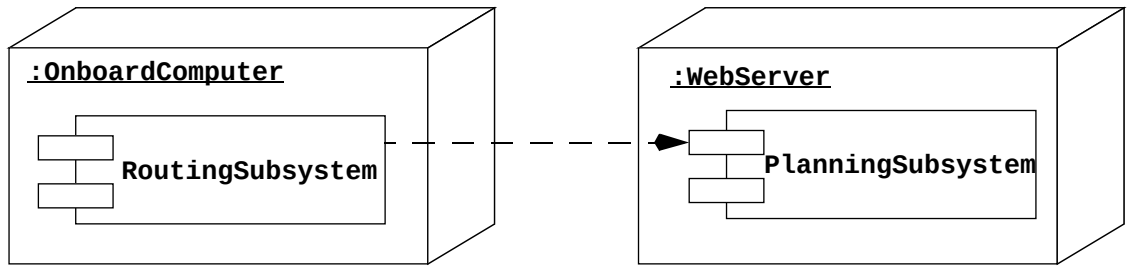


FIGURE 133. Allocation of MyTrip subsystems to hardware (UML deployment diagram). **RoutingSubsystem** runs on the **OnboardComputer** while **PlanningSubsystem** runs on a **WebServer**.

We select a Unix machine as the virtual machine for the :WebServer and the web browsers Netscape and Internet Explorer as the virtual machines for the :OnBoardComputer.

Allocating objects and subsystems to nodes

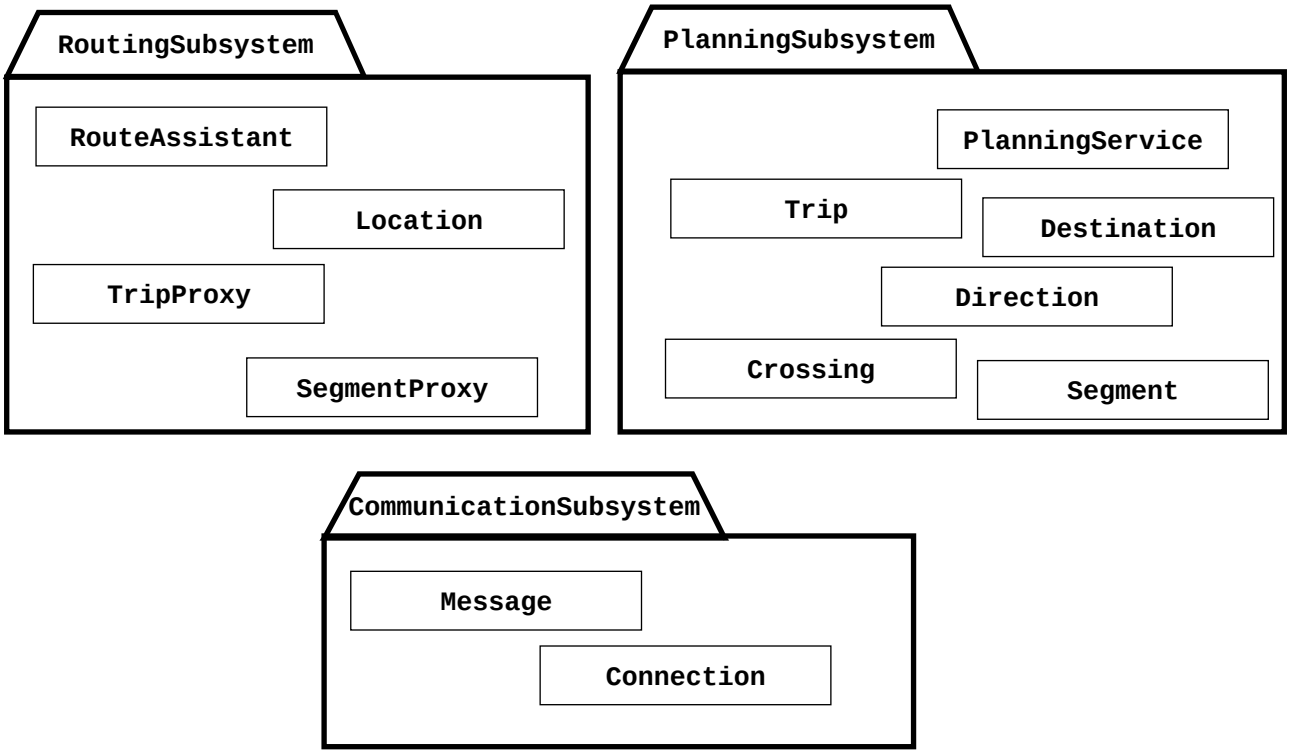
Once the hardware configuration has been defined and the virtual machines selected, objects and subsystems are assigned to nodes. This often triggers the identification of new objects and subsystems for transporting data among the nodes.

In the MyTrip system, both **RoutingSubsystem** and **PlanningSubsystem** share the objects **Trip**, **Destination**, **Crossing**, **Segment**, and **Direction**. Instances of these classes need to communicate via a wireless modem using some communication protocol. We create a new subsystem to support this communication: **CommunicationSubsystem**, a subsystem located on both nodes for managing the communication between the two.

We also notice, that only segments constituting the planned trip are stored in **RoutingSubsystem**. Adjacent segments not part of the trip are stored only in the **PlanningSubsystem**. To take this into account, we need objects in the **RoutingSubsystem** that can act as a surrogates to **Segments** and **Trips** in the **PlanningSubsystem**. An object that acts on the behalf of another one is called a proxy. We therefore create two new classes called **SegmentProxy** and **TripProxy** and make them part of the **RoutingSubsystem.** These proxies are examples of the Proxy design pattern [Gamma et al., 1994].

In case of replanning by the driver, this class will transparently request the **CommunicationSubsystem** to retrieve the information associated with its corresponding

Segments on the **PlanningSubsystem**. Finally, the **CommunicationSubsystem** is used for transferring a complete trip from **PlanningSubsystem** to **RouteAssistant**. The revised design model and the additional class descriptions are depicted in Figure 134.



CommunicationSubsystem	The **CommunicationSubsystem** is responsible for transporting objects from the **PlanningSubsystem** to the **RoutingSubsystem**.
Connection	A **Connection** represents an active link between the **PlanningSubsystem** and the **RoutingSubsystem**. A **Connection** object handles exceptional cases associated with loss of network services.
Message	A **Message** represents a **Trip** and its related **Destinations**, **Segments**, **Crossings**, and **Directions**, encoded for transport.

FIGURE 134. Revised design model for MyTrip (UML Class diagram, associations omitted for clarity).

In general, allocating subsystems to hardware nodes enables us to distribute functionality and processing power where it is most needed. Unfortunately, it also introduces issues related to storing, transferring, replicating, and synchronizing data among subsystems. For this reason, developers also select the components they will use for developing the system.

Encapsulating components

As the complexity of systems increases and the time to market shortens, developers have strong incentives to reuse code and to rely on vendor supplied components. Interactive systems, for example, are now rarely built from scratch, but rather, are developed with user interface toolkits that provide a wide range of dialogs, windows, buttons, or other standard interface objects. Other projects focus on redoing only part of an existing system. For example, corporate information systems, costly to design and build, need to be updated to new client hardware. Often, only the client side of the system is upgraded to new technology and the backend of the system left untouched.[1] Whether dealing with off-the-shelf component or legacy code, developers have to deal with existing code which they cannot modify and which has not been designed to be integrated into their system.

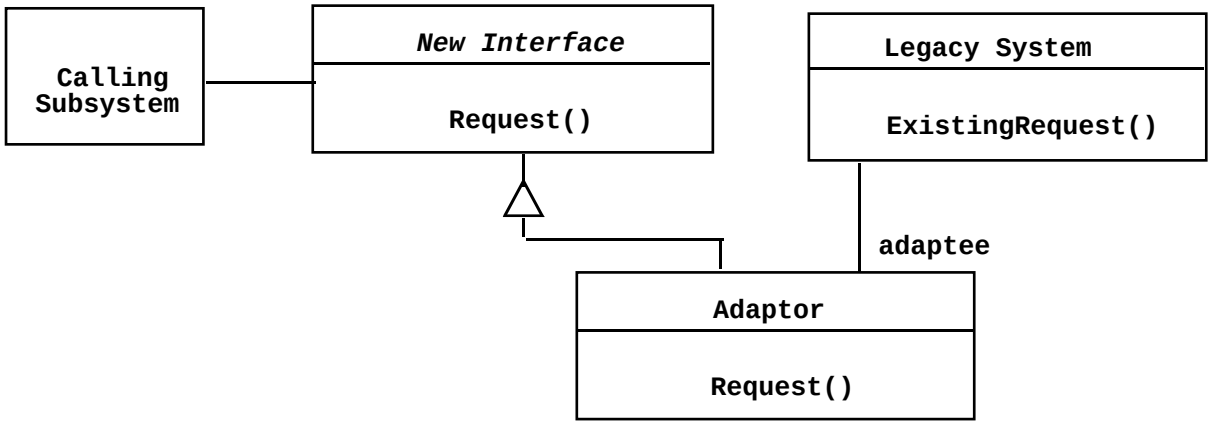


FIGURE 135. Adapter pattern (UML class diagram). The adapter pattern is used to provide a different interface (*New Interface*) to an existing component (**Legacy System**).

We can deal with existing components such as code by encapsulating them. This approach has the advantage of decoupling the system from the encapsulated code, thus minimizing

1. Such projects are called interface engineering projects (see Chapter 3, Software Life Cycle)

the impact of existing software on the design. When the encapsulated code is written in the same language as the new system, this can be done using an Adapter pattern.

```
// Existing target interface
interface Comparator {
   int compare(Object o1, Object o2);
   //…
}

// Existing client
class Array {
   static void sort(Object [] a, Comparator c);
   //…
}

// Existing adaptee class
class MyString extends String {
   boolean equals(Object o);
   boolean greaterThan(MyString s);
   //…
}

// New adaptor class
class MyStringComparator implements Comparator {
   //…
   int compare(Object o1, Object o2) {
      int result;
      if (o1.greaterThan(o2)) {
         result = 1
      } else if (o1.equals(o2)) {
         result = 0;
      } else {
         result = -1;
      }
      return result;
   }
}
```

FIGURE 136. Adapter pattern example (Java). The static **`sort()`** method on **`Arrays`** takes two arguments: an arrays of **`Objects`** to be sorted and a **`Comparator`** defining the relative order of the elements. To sort an array of **`MyStrings`**, we need to define a comparator called **`MyStringComparator`** with the proper interface. **`MyStringComparator`** is an Adaptor.

The Adapter pattern (see Figure 135) is used to convert the interface of an existing piece of code into an interface, called the **`New Interface`**, that a calling subsystem expects. An **`Adaptor`** class, also called a wrapper, is introduced to provide the glue between **`New Interface`** and **`Legacy System`**. For example, assume the client is the static **`sort()`** method of the Java **`Array`** class (see Figure 136). This method expects two arguments **`a`**, an **`Array`** of objects, and **`c`**, a **`Comparator`** object, which provides a **`compare()`** method defining the relative order between elements. Assume we are interested in sorting strings of the class **`MyString`**, which defines the **`greaterThan()`** and an **`equals()`** methods. To sort an **`Array`** of **`MyStrings`**, we need to define a new comparator, **`MyStringComparator`**, which provides a **`compare()`** method using **`greaterThan()`** and **`equals()`**. **`MyStringComparator`** is an Adaptor class. [1]

When encapsulating legacy code that is written in a different language than the system under development, we need to deal with language differences. Although integrating code from two different compiled languages can be done, it can present major problems, especially when one or both of the languages are object-oriented and implement different message dispatching semantics. This motivated standards such as CORBA which defines protocols for allowing the interoperabilty of distributed objects written in different languages. In the case of client/server architectures, other solutions include developing wrappers around communication protocols between processes.

Protocols for interprocess communication (e.g., pipes and sockets) are usually provided by the operating system, and thus, are language independent. In the cases of CORBA and interprocess communication, the cost of invoking a service becomes much higher than the cost of sending messages among objects in the same process. You need to carefully evaluate the impact on performance by wrappers around legacy code when response time and other performance design goals have been selected.

Technology decisions become obsolete quickly. The system you are building is likely to survive many platforms and will be ported and upgrade several times during maintenance. These tasks, when performed during maintenance, are usually costly because a large amount of the design information is lost. Why was the original hardware configuration used? Which features of the database management system does this system rely on? Developers can preserve this information by documenting the design rationale of their system, including hardware and component decisions. We describe techniques for doing this in Chapter 9, *Design Rationale*.

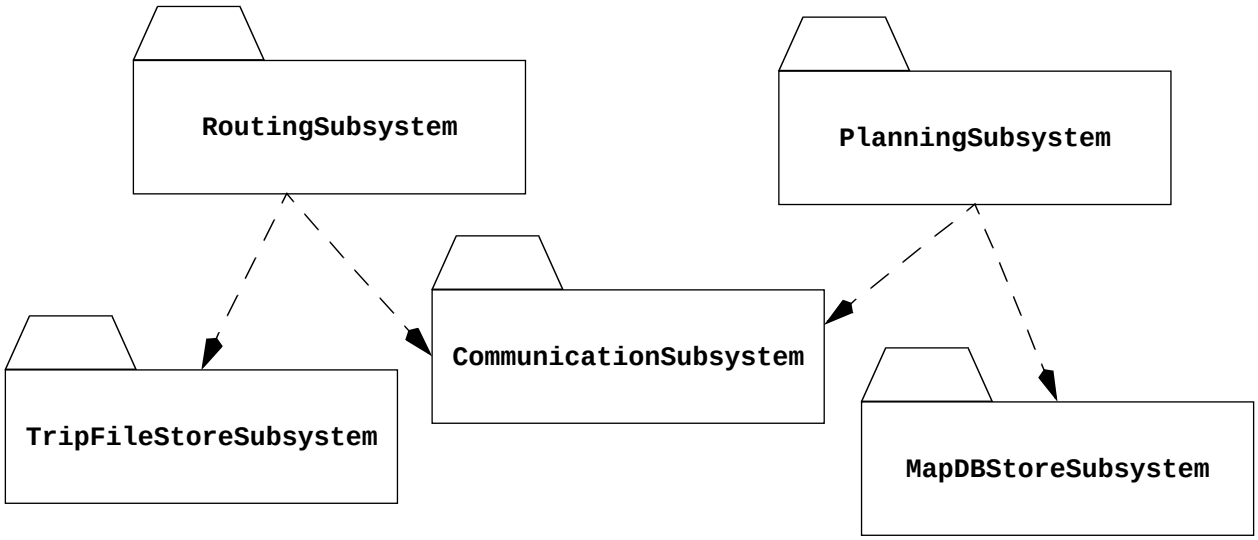
1. When designing a new system, adapters are seldom necessary as the interface of new classes can be defined such that they comply with the necessary interfaces

8.4.5. Defining persistent data stores

Persistent data outlive a single execution of the system. For example, an author may save his work into a file when using a word processor. The file can then be re-opened several days or weeks later. The word processor need not to run for the file to exist. Similarly, information related to employees, their employment status, and their paychecks live in a database management system. This allows all the programs that operate on employee data to do so consistently. Moreover, storing data in a database enables the system to perform complex queries on a large data set (e.g., the records of several thousands of employees).

Where and how data is stored in the system impacts the system decomposition. In some cases, for example in a repository architecture (see Section 8.3.5), a subsystem can be completely dedicated to the storage of data. The selection of a specific database management system can also have implications on the overall control strategy and concurrency management.

For example, in MyTrip, we decide to store the current **`Trip`** in a file on a small removable disk in order to allow the recovery of the **`Trip`** in case the driver shuts off the car before reaching the final **`Destination`**. Using a file is the simplest and most efficient solution in this case, given that the **`RoutingSubsystem`** will only store complete trips to the file before shutdown and load the file at start-up. In the **`PlanningSubsystem`**, however, the trips will be stored in a database. This subsystem can then be used to manage all **`Trips`** for many drivers as well as the maps needed to generate the trips. Using a database for this subsystem allows us to perform complex queries on these data. We add the **`TripFileStoreSubsystem`** and the **`MapDBStoreSubsystem`** subsystems to MyTrip to reflect these decisions, as illustrated in Figure 137.



`TripFileStoreSubsystem`	The **`TripFileStoreSubsystem`** is responsible for storing trips in files on the onboard computer. Since this functionality is only used for storing trips when the car shuts down, this subsystems only supports the fast storage and loading of whole trips.
`MapDBStoreSubsystem`	The **`MapDBStoreSubsystem`** is responsible for storing maps and trips in a database for the **`PlanningSubsystem`**. This subsystem supports multiple concurrent drivers and planning agents.

FIGURE 137. Subsystem decomposition of MyTrip after deciding on the issue of data stores (UML class diagram, packages collapsed for clarity).

In general, we first need to identify which objects need to be persistent. The persistency of objects is directly inferred from the application domain. In MyTrip, only **`Trips`** and their related classes need to be stored. The location of the car, for example, need not be persistent since it needs to be recalculated constantly. Then, we need to decide how these objects should be stored (e.g., file, relational database, or object database). The decision for the storage management is more complex and is usually dictated by nonfunctional requirements: should the objects be retrieved quickly? Is there a need for complex queries? Do objects take a lot of space (e.g., are there images to store)? Database management systems provide mechanisms for concurrency control and efficient queries over large datasets.

There are currently three realistic options for storage management:

- ***Flat files.*** Files are the storage abstractions provided by operating systems. The application stores its data as a sequence of bytes and defines how and when data should be retrieved. The file abstraction is relatively low level and enables the application to perform a variety of size and speed optimizations. Files, however, require the application to take care of many issues, such as concurrent accesses, and loss of data in case of crash.
- ***Relational database.*** A relational database provides an abstraction of data that is higher than flat files. Data are stored in tables which comply with a predefined type called schema. Each column in the table represents an attribute. Each row represents a data item as a tuple of attribute values. Several tuples in different tables are used to represent the attributes of an individual object. Relational databases have been used for a while and are a mature technology. The use of a relational database introduces a high cost and, often, a performance bottleneck.
- ***Object-oriented database.*** An object-oriented database provides services similar to a relational database. Unlike a relational database, it stores data as objects and associations. In addition to providing a higher level of abstraction (and thus reducing the need to translate between objects and storage entities), object-oriented databases provide developers with inheritance and abstract datatypes. Object-oriented databases are usually slower than relational databases for typical queries.

The following box summarizes the trade-offs when selecting storage management system.

Trade-off between files and databases
When should you choose a file?
- Voluminous data (e.g., images)
- Temporary data (e.g., core file)
- Low information density (e.g., archival files, history logs)

When should you choose a data base?
- Concurrent accesses
- Access at fine levels of details
- Multiple platforms
- Multiple applications over the same data

When should you choose a relational database?
- Complex queries over attributes.
- Large dataset.

When should you choose an object-oriented database?
- Extensive use of associations to retrieve data.
- Medium size data set.
- Irregular associations among objects.

Encapsulating data stores

Once we select a storage mechanism (say, a relational database), we can encapsulate it into a subsystem and define a high-level interface that is vendor independent. For example, the ***Bridge*** pattern (see Figure 138 and [Gamma et al., 1994]) allows the interface and the implementation of a class to be decoupled. This allows the substitution of different implementations of a given class, sometimes even at run-time. The **`Abstraction`** class defines the interface visible to the client. The **`Implementor`** is an abstract class which defines the lower level methods available to **`Abstraction`**. An **`Abstraction`** instance maintains a reference to its corresponding **`Implementor`** instance. **`Abstraction`** and **`Implementor`** can be refined independently.

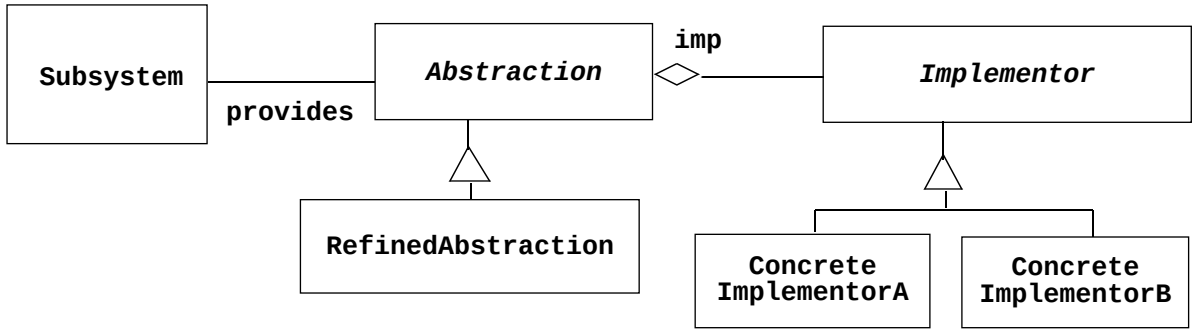


FIGURE 138. Bridge pattern (UML class diagram).

Database connectivity standards such as ODBC [Microsoft, 1995] and JDBC [JDBC, 1998] provide such abstractions for relational databases (see ODBC Bridge pattern in Figure 139). Note, however, that even if most relational databases provide similar services, providing

such an abstraction reduces performance. The design goals we defined at the beginning of the system design phase help us trade-off performance vs. modifiability.

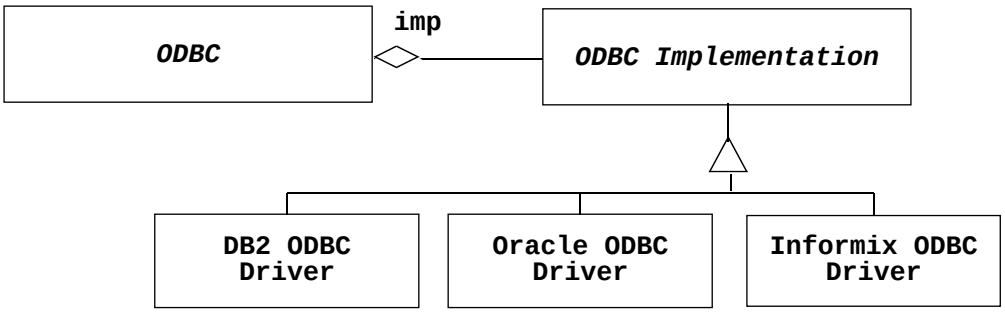


FIGURE 139. Bridge pattern for abstracting database vendors (UML class diagram). Removing the dependency from database vendors from the systems enables more flexibility in configuring the system. Some clients already have a site license with a database vendor. Other clients might be interested in cutting price and using a free database management system.

8.4.6. Defining access control

In multi user systems, different actors have access to different functionality and data. For example, an everyday actor may only access the data it creates while a system administrator actor may have unlimited access to system data and to other users' data. During requirements analysis, we modeled these distinctions by associating different use cases to different actors. During system design, we model access by examining the object model, by determining which objects are shared among actors, and by defining how actors can control access. Depending on the security requirements on the system, we also define how actors are authenticated to the system (i.e., how actors prove to the system who they are) and how selected data in the system should be encrypted.

`CommunicationSubsystem`	The **`CommunicationSubsystem`** is responsible for transporting ***`Trips`*** from the **`PlanningSubsystem`** to the **`RoutingSubsystem`**. *The* ***`CommunicationSubsystem`*** *uses the* ***`Driver`*** *associated with the* ***`Trip`*** *being transported for selecting a key and encrypting the communication traffic.*

FIGURE 140. Revisions to the design model stemming from the decision to authenticate Drivers and encrypt communication traffic (revisions indicated in *italics*).

PlanningSubsystem	The **PlanningSubsystem** is responsible for constructing a **Trip** connecting a sequence of **Destinations**. The **PlanningSubsystem** is also responsible for responding to replan requests from **RoutingSubsystems**. *Prior to processing any requests, the **PlanningSubsystem** authenticates the **Driver** from the **RoutingSubsystem**. The authenticated **Driver** is used to determine which **Trips** can be sent to the corresponding **RoutingSubsystem**.*
Driver	*A **Driver** represents an authenticated user. It is used by the **CommunicationSubsystem** to remember keys associated with a user and by the **PlanningSubsystem** to associate **Trips** with users.*

FIGURE 140. Revisions to the design model stemming from the decision to authenticate Drivers and encrypt communication traffic (revisions indicated in *italics*).

For example, in MyTrip, storing maps and **Trips** for many drivers in the same database introduces security issues. We must ensure that **Trips** are sent only to the driver that created them. This is also consistent with the security design goal we defined in Section 8.4.2 for MyTrip. Consequently, we model a driver with the **Driver** class and associate it with the **Trip** class. The **PlanningSubsystem** becomes also responsible for authenticating **Drivers** before sending **Trips**. Finally, we decide to encrypt the communication traffic between the **RoutingSubsystem** and the **PlanningSubsystem**. This will be done by the **CommunicationSubsystem**. The descriptions for the **Driver** class and the revised descriptions for the **PlanningSubsystem** and the **CommunicationSubsystem** are displayed in Figure 140.

Defining access control for a multi user system is usually more complex than in MyTrip. In general, we need to define for each actor which operations they can access on each shared object. For example, in a bank information system, a teller may credit or debit money from local accounts up to a pre-defined amount. If the transaction exceeds the pre-defined amount, a manager needs to approve the transaction. Moreover, managers and tellers can only access accounts in their own branch, that is, they cannot access accounts in other branches. Analysts on the other hand, can access information across all branches of the corporation, but cannot post transactions on individual accounts.

We model access on classes with an access matrix. The rows of the matrix represents the actors of the system. The columns represent classes whose access we control. An entry **(class, actor)** in the access matrix is called an ***access right*** and lists the operations (e.g., **postSmallDebit()**, **postLargeDebit()**, **examineBalance()**, **getCustomerAddress()**)

that can be executed on instances of the **`class`** by the **`actor`**. Figure 141 is an example of access matrix for our bank information system.

Objects **Actors**	**`Corporation`**	**`LocalBranch`**	**`Account`**

`Teller`		`lookupLocalAccount()`	`postSmallDebit()`
			`postSmallCredit()`
			`lookupBalance()`

`Manager`		`lookupLocalAccount()`	`postSmallDebit()`
			`postSmallCredit()`
			`postLargeDebit()`
			`postLargeCredit()`
			`examineBalance()`
			`examineHistory()`

`Analyst`	`examineGlobalDebits()` `examineGlobalCredits()`	`examineLocalDebits()` `examineLocalCredits()`	

FIGURE 141. Access matrix for a banking system. **`Tellers`** can only lookup local accounts, perform small transactions on accounts, and inquire balances. **`Managers`** can perform larger transactions and access account history in addition to the operations accessible to the tellers. **`Analysts`** can access statistics for all branches but not perform operations at the account level.

We can represent the access matrix using one of three different approaches, global access table, access control list and capabilities:

- A ***global access table*** represents explicitly every cell in the matrix as a **`(actor, class, operation)`** tuple. Determining if an actor has access to a specific object requires looking up the corresponding tuple. If no such tuple is found, access is denied.
- An ***access control list*** associates a list of **`(actor, operation)`** pairs with each **`class`** to be accessed. Empty cells are discarded. Every time an object is accessed, its access list is checked for the corresponding actor and operation. An example of an access control list is the guest list for a party. A butler checks the arriving guests by comparing their names against names on the guest list. If there is a match, the guests can enter, otherwise they are turned back.
- A ***capability*** associates a **`(class, operation)`** pair with an **`actor`**. A capability provides an actor to gain control access to an object of the class described in the capability. Denying a capability is equivalent to denying access. An example of a capability is an invitation card for a party. In this case, the butler checks if the arriving

guests hold an invitation for the party. If the invitation is valid, the guests are admitted, otherwise, they are turned back. No other checks are necessary.

The representation of the access matrix is also a performance issue. Global access tables are rarely used as they require a lot of space. Access control lists make it faster to answer the question “Who has access to this object?”, while capability lists make it faster to answer the question “Which objects has this actor access to?”.

Each row in the access matrix represents a different access view of the classes listed in the columns. All of these access views should be consistent. Usually, however, access views are implemented by defining a subclass for each different type of actor, operation tuple. For example, in our banking system, we would implement an **`AccountViewedByTeller`** and **`AccountViewedByManager`** class as subclasses of Account. Only the appropriate classes are available to the corresponding actor. For example, the **`Analyst`** client software would not include an **`Account`** class since the **`Analyst`** has no access to any operation in this class. This reduces the risk that an error in the system results in the possibility of unauthorized access.

An access matrix only represents ***static access control***. This means, that access rights can be modeled as attributes of the objects of the system. In the bank information system example, consider a broker actor who is assigned dynamically a set of portfolios. By policy, a broker cannot access portfolios managed by another broker. In this case, we need to model access rights dynamically in the system, and hence, this type of access is called ***dynamic access control***. For example, Figure 142 shows how this access can be implemented with a protection proxy pattern [Gamma et al., 1994]. For each **`Portfolio`**, we create a **`PortfolioProxy`** to protect the **`Portfolio`** and check for access. An **`Access`** association between a legitimate **`Broker`** and a **`PortfolioProxy`** indicates which **`Portfolio`** the **`Broker`** has access to. To access a **`Portfolio`**, the **`Broker`** sends a message to the corresponding **`PortfolioProxy`**. The **`PortfolioProxy`** first checks if the invoking **`Broker`**

has the appropriate association with the **PortfolioProxy**. If access is granted, the **PortfolioProxy** delegates the message to the **Portfolio**. Otherwise, the operation fails.

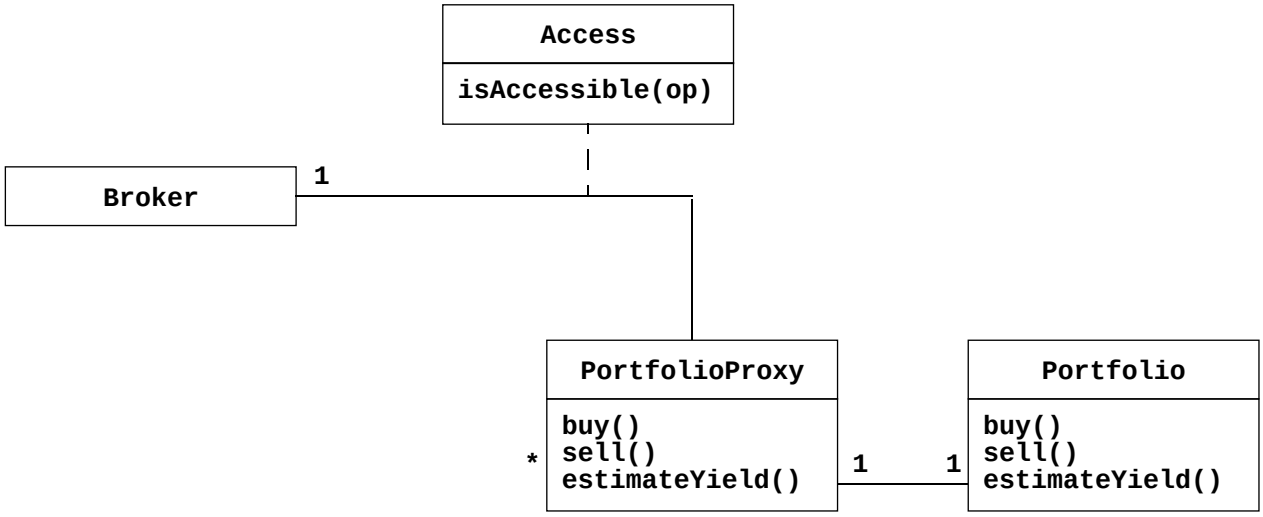


FIGURE 142. Dynamic access implemented with a protection proxy. The **Access** association class contains a set of operations that **Broker** can use to access a **Portfolio**. Every operation in the **PortfolioProxy** first checks with the **isAccessible()** operation if the invoking **Broker** has legitimate access. Once access has been granted, **PortfolioProxy** delegates the operation to the actual **Portfolio** object. One **Access** association can be used to control access to many **Portfolios**.

In both types of access control, we assume that we know the actor, either the user behind the keyboard or the calling subsystem. This process of verifying the association between the identity of the user or subsystem and the system is called ***authentication***. A widespread authentication mechanism, for example, is for the user to specify a user name, known by everybody, and a corresponding password, only known to the system and stored in an access control list. The system protects its users password by encrypting them before storing or transmitting them. If only a single user knows this user name/ password combination, that we can assume that the user behind the keyboard is legitimate. Although password authentication can be made secure with current technology, it suffers from many usability disadvantages: users choose passwords that are easy to remember, and thus, easy to guess. They also tend to write their password on notes which they keep close to their monitor, and thus, visible to many other, unauthorized, users. Fortunately, other, more secure, authentication mechanisms are available. For example, a smart card can be used in conjunction with a password: an intruder would need both the smart card and the password to gain access to the system. Better, we can use a biometric sensor for analyzing patterns of

blood vessels in a person's fingers or eyes. An intruder would then need the physical presence of the legitimate user to gain access to the system, which is much more difficult than just stealing a smart card.

In an environment where resources are shared among multiple users, authentication is usually not sufficient. In the case of a network, for example, it is relatively easy for an intruder to find tools to snoop the network traffic, including packets generated by other users (see Figure 143). Worse, protocols such as TCP/IP were not designed with security in mind: an intruder can forge packets such that they appear as if they were coming from legitimate users.

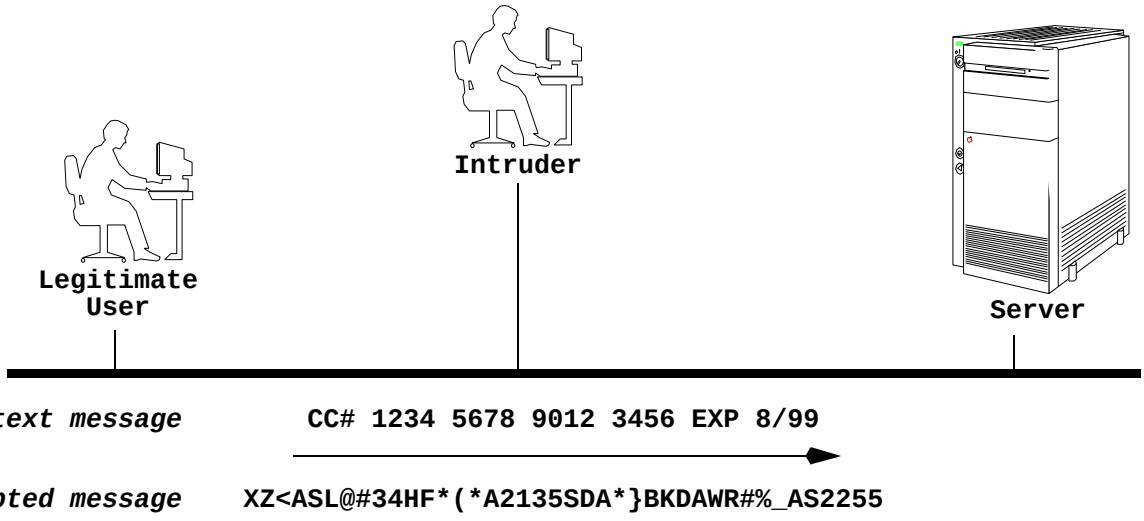


FIGURE 143. Passive attack. Given current technology, it is relatively easy for a passive intruder to listen to all network traffic. To prevent this kind of attack, encryption makes the information an intruder sees difficult to understand.

Encryption is used to prevent such unauthorized accesses. Using an encryption algorithm, we can translate a message, called ***plaintext***, into a encrypted message, called a ***cyphertext***, such that even if an intruder intercepts the message, it cannot be understood. Only the receiver has sufficient knowledge to correctly decrypt the message, that is, for reversing the original process. The encryption process is parameterized by a ***key***, such that the method of encryption and decryption can be switched quickly in case the intruder manages to obtain sufficient knowledge to decrypt the message.

Secure authentication and encryption are fundamentally difficult problems. You should always select one or more off-the-shelf algorithms or package instead of designing your

own (unless you are in the business of building such packages). Many such packages are based on public standards that are widely reviewed by the academia and the industry, thus ensuring a relatively high level of reliability and security.

Encapsulating access control

The use of vendor supplied software introduces a security problem: how can we be sure that the supplied software does not include a trap door? Moreover, once a vulnerability is found in a widely used package, how do we protect the system until a patch is available? We can use redundancy to address both issues. For example, the Java Cryptographic Architecture [JCA, 1998] allows multiple implementations of the same algorithms to coexist in the same system, thus reducing the dependency on a specific vendor. More generally, we can use the ***Strategy*** pattern [Gamma et al., 1994] to encapsulate multiple implementation of the same algorithm. In this pattern (see Figure 144), the *Strategy* abstract class defines the generic interface all implementations of the encapsulated algorithm should have. **ConcreteStrategy** classes provide implementations of the algorithm by subclassing *Strategy*. A **Context** class is responsible for managing the data structure on which **ConcreteStrategies** operate. **Context** and a **ConcreteStrategy** class cooperate to provided the needed functionality.

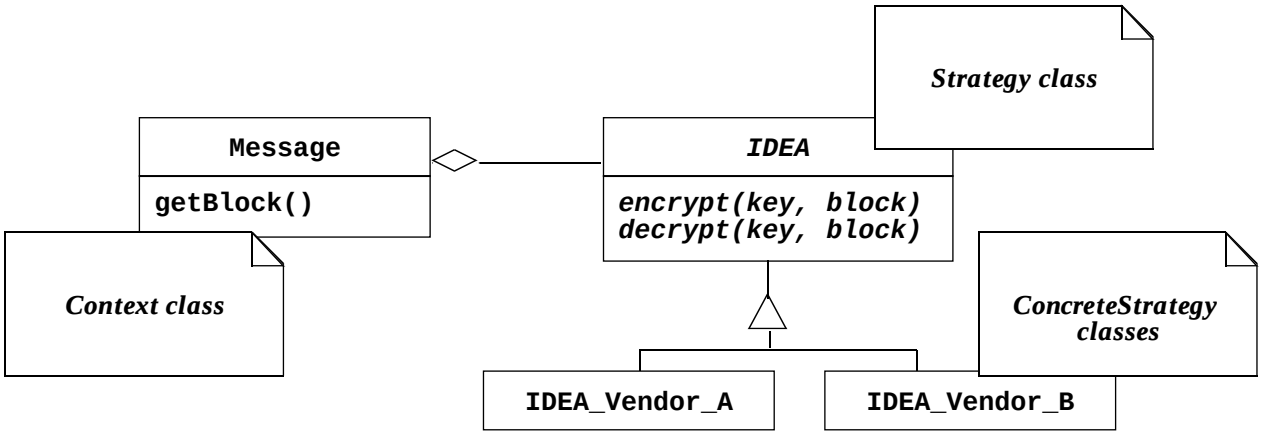


FIGURE 144. An example of Strategy pattern encapsulating multiple implementation of the IDEA encryption algorithm (UML class diagram). The **Message** and *IDEA* classes cooperate to realize the encryption of plain text. The selection of an implementation can be done dynamically.

Once authentication and encryption are provided, application specific access control can be more easily implemented on top of these building blocks. In all cases, addressing security issues is a difficult topic. When addressing these issues, developers should record their assumptions and describe the intruder scenarios they are considering. When several alternatives are explored, developers should state the design problems they are attempting to solve and record the results of the evaluation. We describe in the next chapter how to do this systematically using issue modeling.

8.4.7. Designing the global control flow

Control flow is the sequencing of actions in a system. In object-oriented systems, sequencing actions includes deciding which operations should be executed and in which order. These decisions are based on external events generated by an actor or on the passage of time.

Control flow is a typical design problem. During requirements analysis control flow is not an issue, because we simply assume that all objects are running simultaneously, executing operations any time they need to execute them. During system design we need to take into account that not every object has the luxury of running on its on processor.

There are three possible control flow mechanisms:

- ***Procedure-driven control.*** Operations wait for input whenever they need data from an actor. This kind of control flow is mostly used in legacy systems and systems written in procedural languages. It introduces difficulties when used with object-oriented languages. As the sequencing of operations is distributed among a large set of objects, it becomes increasingly difficult to determine the order of inputs by looking at the code.

```
    Stream in, out;
    String
// … initialization omitted …
    out.println(“Login:”);
    in.readln(userid);
    out.println(“Password:”);
    in.readln(passwd);
    if (!security.check(userid, passwd)) {
       out.println(“Login failed.”);
       system.exit(-1);
    }
// …
```

FIGURE 145. An example of procedure driven control (Java). The code prints out messages and waits for input from the user.

- ***Event-driven control.*** A main loop waits for an external event. Whenever an event becomes available, it is dispatched to the appropriate object, based on information associated with the event. This kind of control flow has the advantage of leading to a simpler structure and to centralize all input in the main loop. However, it makes the implementation of multi step sequences more difficult to implement.

```
Enumeration subscribers, eventStream;
Subscriber subscriber;
Event event;
EventStream eventStream;
//…
while (eventStream.hasMoreElements) {
   event = eventStream.nextElement();
   subscribers = dispatchInfo.getSubscribers(event);
   while (subscribers.hasMoreElements()) {
      subscriber = subscribers.nextElement()) {
      subscriber.process(event);
   }
}
//…
```

FIGURE 146. An example of main loop for event-driven control (Java). An event is taken from an event queue and sent to objects interested in it.

- ***Threads.*** Threads (also called light weight threads to distinguish them from processes which require more computing overhead) are the concurrent variation of procedure-driven control: the system can create an arbitrary number of threads, each responding to a different event. If a thread needs additional data, it waits for input from a specific actor. This kind of control flow is probably the most intuitive of the three mechanisms. However, debugging thread software requires good debugging tools: preemptive thread schedulers introduce non-determinism in the system and, thus, make it harder to find repeatable test cases.

```
Thread thread;
Event event;
EventHandler eventHandler;
boolean done;
// …
while (!done) {
   event = eventStream.getNextEvent();
   eventHandler = new EventHandler(event)
   thread = new Thread(eventHandler);
   thread.start();
}
// …
```

FIGURE 147. An example of event processing with threads (Java). **`eventHandler`** is an object dedicated to handling **`event`**. It implements the **`run()`** operation which is invoked when **`thread`** is started.

Procedure-driven control is useful for testing subsystems. A driver makes specific calls to methods offered by the subsystem. For the control flow of the final system, though, procedure-driven control should be avoided.

The trade-off between event-driven control and threads is more complicated. Event-driven control is more mature than threads. Modern languages have only recently started to provide support for thread programming. As more debugging tools are becoming available and experience is accumulated, developing thread-based systems will become easier. Also, many user interface packages supply the infrastructure for dispatching events and impose this kind of control flow on the design. Although threads are more intuitive, they currently introduce many problems during debugging and testing. Until more mature tools and infrastructures are available for developing with threads, event-driven control flow is preferred.

Once a control flow mechanism is selected, we can realize with a set of one or more control objects. The role of control objects is to record external events, store temporary state about them, and issue the right sequence of operation calls on the interface and entity objects associated with the external event. On the one hand, localizing control flow decisions for a use case into a single objects results into more understandable code, on the other hand, it makes the system more resilient to changes in control flow implementation.

Encapsulating control flow

An example of encapsulation of control is the ***Command*** pattern [Gamma et al., 1994] (see Figure 148). In interactive systems it is often desirable to execute, undo, or store user requests without knowing the content of the request. The key to decoupling requests from their handling is to turn requests into command objects which inherit from an abstract **Command** class. The **Command** class defines how the command is executed, undone, or stored, while the concrete class implements specific requests.

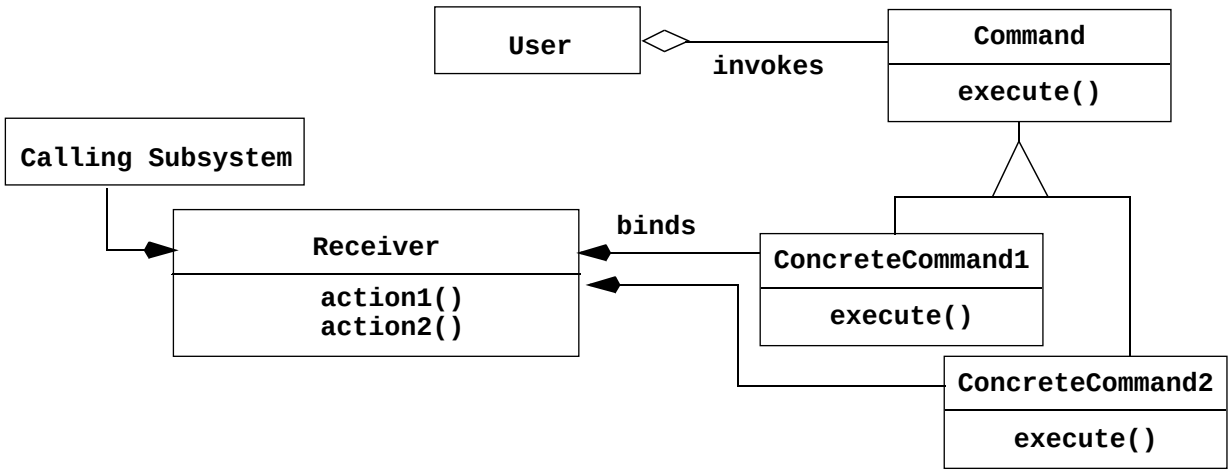


FIGURE 148. Command pattern (UML class diagram).This pattern enables the encapsulation of control such that user requests can be treated uniformly, independent of the specific request.

We can use the command pattern to decouple menu items from actions (see Figure 149). Decoupling menu items from actions has the advantage of centralizing control flow (e.g., dialog sequencing) into control objects instead of spreading it between interface and entity objects. A **Menu**, composed of **MenuItems**, creates a **Command** object of the appropriate class whenever the corresponding **MenuItem** is selected by the user. The **Application** invokes the **execute()** operation of the newly created **Command** object. If the user wishes to undo the last request, the **undo()** operation of the last **Command** object is executed. Different **Command** objects implement different requests (e.g., **CopyCommand** and **PasteCommand**).

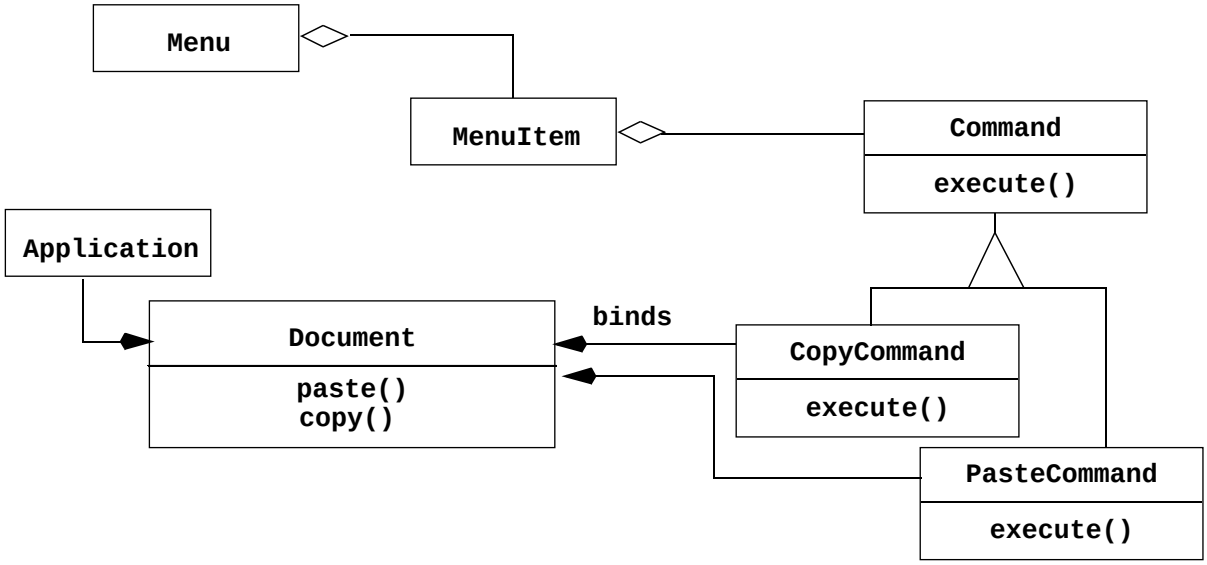


FIGURE 149. An example of a command pattern (UML class diagram). In this example, menu items and operations on documents are decoupled. This enables us to centralize control flow in the command objects (**`CopyCommand`** and **`PasteCommand`**) instead of spreading it across interface objects (**`MenuItem`**) and entity objects (**`Document`**).

8.4.8. Identifying boundary conditions

In previous sections, we dealt with designing and refining the system decomposition. We now have a better idea of how to decompose the system, how to distribute use cases among subsystems, where to store data, and how to achieve access control and to ensure security. We still need to examine the boundary conditions of the system, that is, to decide how the

system is started, initialized, shut down and we need to define how we deal with major failures, such as data corruption, whether it is caused by a software error or a power outage.

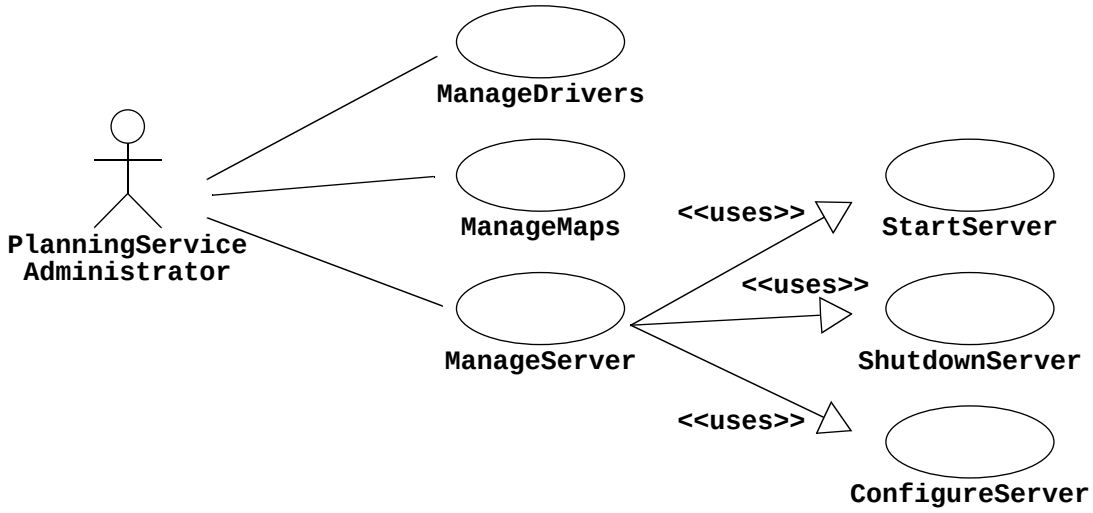


FIGURE 150. Administration use cases for MyTrip (UML use case diagram). **`ManageDrivers`** is invoked to add, remove, modify, or read data about drivers (e.g., user name and password, usage log, encryption key generation). **`ManageMaps`** is invoked to add, remove, or update maps that are used to generate trips. **`ManageServer`** includes all the functions necessary to start-up and shutdown the server.

For example, we now have a good idea of how MyTrip should work in steady state. We have, however, not yet addressed defining how MyTrip is initialized. For example, how are maps loaded into the **`PlanningService`**? How is MyTrip installed in the car? How does MyTrip know which **`PlanningService`** to connect to? How are drivers added to the **`PlanningService`**? We quickly discover a set of use cases that has not been specified. We call these the system administration use cases. *System administration use cases* specify the behavior of a system during the start-up and shutdown phase.

It is common that system administration use cases are not specified during requirements analysis or that they are treated separately. On the one hand, many system administration functions can be inferred from the everyday user requirements (e.g., registering and deleting users, managing access control), on the other hand, many functions are consequences of design decisions (e.g., cache sizes, location of database server, location of backup server), and not of requirement decisions.

We now modify the analysis model for MyTrip to include the administration use cases. In particular, we add three uses cases: **`ManageDrivers`**, to add, remove, and edit drivers, **`ManageMaps`**, to add, remove, and update maps used to generate trips, and **`ManageServer`**, to perform routine configuration, start-up, and shutdown (see Figure 150). **`StartServer`**, part of **`ManageServer`**, is provided as an example in Figure 151.

Use case name	**`StartServer`**
Entry condition	1. The **`PlanningServiceAdministrator`** logs into the server machine.
Flow of events	2. Upon successful login, the **`PlanningServiceAdministrator`** executes the **`startPlanningService`** command. 3. If the **`PlanningService`** was previously shutdown normally, the server reads the list of legitimate **`Drivers`** and the index of active **`Trips`** and **`Maps`**. If the **`PlanningService`** had crashed, it notifies the **`PlanningServiceAdministrator`** and performs a consistency check on the **`MapDBStore`**.
Exit condition	4. The **`PlanningService`** is available and waits for connections from **`RoutingAssistants`**.

FIGURE 151. **`StartServer`** use case of the MyTrip system.

In this case, adding three use cases, that is, revising the use case model, does not impact the subsystem decomposition. We added, however, new use cases to existing subsystems: the **`MapDBStoreSubsystem`** needs to be able to detect whether it was properly shut down or not, and needs to be able to perform consistency checks and repair corrupted data, if necessary. We revise the description of **`MapDBStoreSubsystem`**.

`MapDBStoreSubsystem`	The **`MapDBStoreSubsystem`** is responsible for storing maps and trips in a database for the **`PlanningSubsystem`**. This subsystem supports multiple concurrent drivers and planning agents. *When starting up, the **`MapDBStoreSubsystem`** detects if it was properly shutdown. If not, it performs a consistent check on the **`Maps`** and **`Trips`** and repairs corrupted data if necessary.*

FIGURE 152. Revised description for **`MapDBStoreSubsystem`** based on the additional **`StartServer`** use case of Figure 151 (Changes indicated in *italics*).

When examining boundary conditions, we also need to investigate exceptional cases. For example, the nonfunctional requirements of MyTrip specify that the system needs to tolerate connection failures. For this reason, the **`RouteAssistant`** downloads the planned **`Trip`** at the initial **`Destination`**. We also decide to download **`Segments`** that are close to the **`Trip`**, to enable minimum replanning even though a connection might not be available.

In general, an ***exception*** is an unexpected event or error that occurs during the execution of the system. Exceptions are caused by one of three different sources:

- *A user error*. The user mistakenly or deliberately input data that is about of bounds. For example, a negative amount in a banking transaction could lead to transferring money in the wrong direction if the system does not protect against such errors.
- *A hardware failure*. Hardware ages and fails. The failure of a network link, for example, can momentarily disconnect two nodes of the system. A hard disk crash can lead to the permanent loss of data.
- *A software bug*. An error can occur either because the system or one of its components contains a design error. Although writing bug free software is difficult, individual subsystem can anticipate errors from other subsystems and protect against them.

Exception handling is the mechanism by which a system treats an exception. In the case of a user error, the system should display a meaningful error message to the user, such that she can correct her input. In the case of a network link failure, the system needs to save its temporary state in order to recover once the network comes back on line.

Developing reliable systems is a difficult topic. Often, trading-off some functionality can make it easier on the design of the system. In MyTrip, we assumed that the connection is always possible at the source destination, and that replanning could be impacted by communication problems along the trip.

8.4.9. Anticipating change

System design introduces a strange paradox in the development process. On the one hand, we want to construct solid walls between subsystems, to manage complexity by breaking the system into smaller pieces, to prevent changes from one subsystem to impact others. On the other hand, we want the software architecture to be modifiable to minimize the cost of later change. These are conflicting goals that cannot be reconciled: we have to define an architecture early to deal with complexity and we have to pay the price of change later in the development process. We can, however, anticipate change and design for it, as sources of later changes tend to be the same for most systems:

- *New vendor or new technology*. When components are used to build the system, anticipate that the component will be replaced by an equivalent one from a different

vendor. This change is frequent and generally difficult to cope with. The software marketplace is dynamic and many vendors will start up and go out of business before your project is completed.

- *New implementation.* When subsystems are integrated and are tested together, the overall system response time is, more often than not, above stated or implicit performance requirements: posting a debit on a bank information system may take two minutes, a flight reservation system takes five minutes to book a flight. System wide performance is difficult to predict and usually not optimized before integration: developers focus on their subsystem first. This triggers the need for more efficient datastructures and algorithms, and better interfaces, often under time pressure.
- *New views.* Testing the software with real users uncovers many usability problems. These translate often into the creation of 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. Other times, the domain itself increases in complexity: previously, flight numbers were associated with one plane and one plane only. With the advent of carrier alliances, one plane can now have multiple flight numbers from different companies.
- *Errors.* Unfortunately, many requirements errors are discovered only when real users start using the system.

Modern object-oriented languages provide mechanisms that can minimize the impact of change when anticipated. The use of inheritance in conjunction with abstract classes decouples the interface of a subsystem from its actual implementation. In this chapter, we have provided you with selected examples of design patterns [Gamma et al., 1994] that deal with the above changes. Figure 153 summarizes the patterns and the type of change they protect against.

Adapter (see example in Section 8.4.4)	*New vendor, new technology, new implementation.* This pattern encapsulates a piece of legacy code that was not designed to work with the system. It also limits the impact of substituting the piece of legacy code for a different component.
Bridge (see example in Section 8.4.5)	*New vendor, new technology, new implementation.* This pattern decouples the interface of a class from its implementation. It serves the same purpose than the adapter pattern except that the developer is not constrained by an existing piece of code.

FIGURE 153. Selected design patterns and the changes they anticipate.

Command (see example in Section 8.4.7)	*New functionality.* This patterns decouples the objects responsible for command processing from the commands themselves. This pattern protects these objects from changes due to new functionality.
Observer (see example in Section 8.3.5)	*New views.* This pattern decouples entity objects from their views. Additional views can be added with entity objects being modified.
Strategy (see example in Section 8.4.6)	*New vendor, new technology, new implementation.* This pattern decouples an algorithm from its implementation(s). It serves the same purpose than the adapter and bridge patterns except that the encapsulated unit is a behavior.

FIGURE 153. Selected design patterns and the changes they anticipate.

A reason for the high cost of change late in the process is the loss of design context. Developers forget very quickly the reasons that pushed them to design complicated workarounds or complex datastructures during early phases of the process. When changing code late in the process, the likelihood of introducing errors into the system is high. To protect against such situations, as many assumptions should be recorded. For example, when using a design pattern to anticipate a certain change (from Figure 153), you should record which change they are anticipating. In Chapter 9, *Design Rationale*, we describe several techniques for recording the design alternatives and decisions.

8.4.10. Reviewing system design

Like requirements analysis, system design is an evolutionary and iterative activity. Unlike requirements analysis, there is no external agent, such as the client, to review the successive iterations and ensure better quality. This quality improvement activity, however, is still necessary, and project managers and developers need to organize a review process to substitute for it. Several alternatives exist, such as using the developers who were not involved in system design to act as independent reviewer, or to use developers from another project to act as a peer review. These review processes work only if the reviewers have an incentive in discovering and reporting problems.

In addition to meeting the design goals that were identified during system design, we need to ensure that the system design model is correct, complete, consistent, realistic, and readable. The system design model is ***correct*** if the requirements analysis model can be mapped to the system design model. You should ask the following questions to determine if the system design is correct:

- Can every subsystem be traced back to a use case or a nonfunctional requirement?
- Can every use case be mapped to a set of subsystems?
- Can every design goal be traced back to a nonfunctional requirement?
- Is every nonfunctional requirement addressed in the system design model?
- Has each actor an access policy?
- Is it consistent with the nonfunctional security requirement?

The model is ***complete*** if every requirement and every system design issue has been addressed. You should ask the following questions to determine if the system design is complete:

- Have the boundary conditions been handled?
- Was there a walkthrough of the use cases to identify missing functionality in the system design?
- Have all use cases been examined and assigned a control object?
- Have all aspects of system design (i.e., hardware allocation, persistent storage, access control, legacy code, boundary conditions) been addressed?
- Do all subsystems have definitions?

The model is ***consistent*** if it does not contain any contradiction. You should ask the following questions to determine if a system design is consistent:

- Are conflicting design goals prioritized?
- Are there design goals that violate a non functional requirement?
- Are there multiple subsystems or classes with the same name?
- Are collections of objects exchanged among subsystems in a consistent manner?

The model is ***realistic*** if the corresponding system can be implemented. You ask the following questions to determine if a system design is realistic:

- Are there any new technologies or components in the system? Were there any studies to evaluate the appropriateness or robustness of these technologies or components?
- Have performance and reliability requirements been reviewed in the context of the subsystem decomposition? For example, is there a network connection on the critical path of the system?
- Have concurrency issues (e.g., contention, deadlocks, mutual exclusion) been addressed?

The model is ***readable*** if developers not involved in the system design can understand the model. You should ask the following questions to ensure that the system design is readable:

- Are subsystem names understandable?
- Do entities (e.g., subsystems, classes, operations) with similar names denote similar phenomena?
- Are all entities described at the same level of detail?

In many projects, you will experience that system design and implementation overlap quite a bit. Parts of the implementation begin before the completion of the system design enabling developers to test risky design decisions early. This leads to many partial reviews instead of an encompassing review followed by a client sign-off, as for requirements analysis. Although this process yields greater flexibility, it also requires developers to track open issues more carefully. Many difficult issues tend to be resolved late not because they are difficult, but because they fell through the cracks of the organization.

8.5. Managing system design

In this section we discuss issues related to managing the system design activities in the context of a multi team project. As in requirements analysis, the primary challenge in managing the system design in such a project is to maintain consistency while using as many resources as possible. In the end, the software architecture and the system interfaces should describe a single coherent system understandable to a single person.

We first describe a document template that can be used to document the results of system (Section 8.5.1). Next, we describe the role assignment during system design (Section 8.5.2) and address communication issues during system design (Section 8.5.3). Next, we address management issues related to the iterative and in MyTrip nature of system design (Section 8.5.4).

8.5.1. Documenting system design

The product of the system design phase is the System Design Document (SDD). It describes the software architecture, including the design goals set by the project, the subsystem decomposition (UML class diagrams), hardware software mapping (UML deployment diagrams), data management, access control, legacy code, and boundary conditions. The SDD is used to define interfaces between teams of developers and as 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. The following template is an example of a SDD:

System Design Document (SDD)

Revision history

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

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 an overview 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 subsystems. 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 conditions* 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 subsystems in terms of operations. Although this section is usually empty or incomplete in the first versions of the SDD, this section servers 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 should be 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, should be 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.

8.5.2. Assigning responsibilities

Unlike requirements analysis, system design is mostly the realm of developers. The client and the end user fades to the background. Note, however, that many activities in system design trigger revisions to the requirements analysis model. The client and the user should be brought back into the process for such revisions. System design in complex systems is centered around the architecture team. This is a cross functional team made of architects (who define the subsystem decomposition) and selected developers (who will take part in the implementation of the subsystem). It is critical that system design include persons that are exposed to the consequences of system design decisions. The architecture team should start to work at full force right after the requirements analysis phase and continue to function until the end of the integration phase. This creates an incentive for the architecture team to anticipate problems encountered during integration. Below are the main roles of system design:

- The ***architect*** is the main role of system design. The architect ensures consistency in design decisions and interface styles. The architect ensures the consistency of the design in the configuration management and testing teams, in particular in the formulation of the configuration management policy as well as the system integration strategy This is mainly an integration role consuming information from

each subsystem team. The architect is the leader of the cross-functional architecture team.

- ***Architecture liaisons*** are the members of the architecture team. They are representatives from the subsystem teams. They convey information from and to their teams and negotiate interface changes. During system design they focus on the subsystem services, during the implementation phase focus on the consistency of the APIs.
- The ***document editor***, ***configuration manager***, and ***reviewer*** roles are the same as for requirements analysis.

The number of subsystems determines the size of the architecture team. For complex systems, an architecture team is introduced for each level of abstraction. In all cases, there should be one integrating role on the team to ensure consistency and the understandability of the architecture by a single individual.

8.5.3. Communicating about system design

Communication during system design should be less challenging than during requirements analysis: the functionality of the system has mostly been defined, project participants have similar backgrounds and by now, should know each other better. Communication is still difficult, due to new sources of complexity:

- *Size*. The number of issues to be dealt with increases as developers start designing. The number of items developers manipulate increases: each piece of functionality requires many operations on many objects. Moreover, developers investigate, often concurrently, multiple designs and multiple implementation technologies.
- *Change*. The subsystem decomposition and the interfaces of the subsystems are in constant flux. Terms used by developers to name different parts of the system evolve constantly. If the change is rapid, developers may not be discussing the same version of the subsystem, which can lead to much confusion.
- *Level of abstraction*. Discussions about requirements can be made concrete by using interface mock-ups and analogies with existing systems. Discussions about implementation become concrete when integration and test results are available. System design discussions are seldom concrete as consequences of design decisions are felt only later, during implementation and testing.
- *Reluctance to confront problems.* The level of abstraction of most discussions can also make it easy to delay the resolution of difficult issues. A typical resolution of control issues is often: “let us revisit this issue during implementation.” While it is usually desirable to delay certain design decisions, such as the internal datastructures and algorithms used by each subsystem for example, any decision that has an impact on the system decomposition and the subsystem interfaces should not be delayed.

- *Conflicting goals and criteria.* Individual developers often optimize different criteria. A developer experienced in user interface design will be biased towards optimizing response time. A developer experienced in databases might optimize throughput. These conflicting goals, especially when implicit, result in developers pulling the system decomposition in different directions, and lead to inconsistencies.

The same techniques we discussed in requirements analysis (see Section 7.4.3) can be applied during system design:

- *Identify and prioritize the design goals for the system and make them explicit* (see Section 8.4.2). If the developers concerned with system design have an input in this process, they will have an easier time committing to these design goals. Design goals also provide an objective framework against which decisions can be evaluated.
- *Make the current version of the system decomposition available to all concerned.* A live document distributed via the internet is one way to achieve rapid distribution. Using a configuration management tool to maintain the system design documents helps developers identifying recent changes.
- *Maintain an up-to-date glossary*. As in requirements analysis, defining terms explicitly reduces misunderstandings. When identifying and modeling subsystems, provide definitions in addition to names. A UML diagram with only subsystem names is not sufficient for supporting effective communication. A brief and substantial definition should accompany every subsystem and class name.
- *Confront design problems.* Delaying design decisions can be beneficial, but this does not prevent design problems to be confronted. If an issue will be revisited during implementation, several possible alternatives should be explored and described. This ensures that issues that are delayed can be delayed without serious impact on the system decomposition.
- *Iterate.* Selected excursions into the implementation phase can improve the system design. For example, new features in a vendor supplied component can be evaluated by implementing a vertical prototype (see Section 8.5.4) for the functionality most likely to benefit from the feature.

Finally, no matter how much effort is expended on system design, the system decomposition and the subsystem interfaces will almost certainly change during implementation. As new information about implementation technologies becomes available, developers have a clearer understanding of the system, and design alternatives are discovered. Developers should anticipate change and reserve some time to update the SDD before system integration.

8.5.4. Iterating over the system design

As in the case of requirements, system design occurs through successive iteration and change. Change, however, should be controlled to prevent chaos, especially in complex projects including many participants. We distinguish three types of iterations during system design. First major decisions early in the system design impact subsystem decomposition as each of the different activities of system design are initiated. Second, revisions to the interfaces of the subsystems occur when evaluation prototypes are done to evaluate specific issues. Third, errors and oversights discovered late trigger changes to the subsystem interfaces and sometimes to the system decomposition itself.

The first set of iterations is best handled through face-to-face and electronic brainstorming. Definitions are still in flux, developers do not have yet a grasp of the whole system, and communication should be maximized at the expense of formality or procedure. Often in team-based projects, the initial system decomposition is designed before the requirements analysis are complete. Decomposing the system early enables the responsibility of different subsystems to be assigned to different teams. Change and exploration should be encouraged if only to broaden the developers shared understanding or to generate supporting evidence for the current design. For this reason, there should not be a bureaucratic formal change process during this phase.

The second set of iterations aims at solving difficult and focused issues, such as the choice of a specific vendor or technology. The subsystem decomposition is stable (since it is should be independent of vendors and technology, see Section 8.4.9) and most of these explorations aim at identifying whether a specific package is appropriate for the system or not. During this period, developers can also realize a vertical prototype[1] for a critical use case to test the appropriateness of the decomposition. This enables control flow issues to be discovered early and addressed. Again, a formal change process is not necessary. A list of pending issues and their status can help developers propagate the result of a technology investigation quickly.

The third set of iterations remedies design problems discovered late. Although developers would much rather avoid any such iterations as they tend to incur a high cost and introduce many new bugs in the system, they should anticipate them. Anticipating late iterations includes documenting dependencies among subsystems, the design rationale for subsystem interfaces, and any work around that is likely to break in case of change. Change should be carefully managed and a change process, similar to the one tracking requirements changes, should be put in place.

1. A vertical prototype implements completely a restricted functionality (e.g., interface, control, and entity objects for one use case) while a horizontal prototype implements partially a broad range of functionality (e.g., interface objects for a number of use cases).

We can achieve the progressive stabilization of the subsystem decomposition using the concept of design window. In order to encourage change while controlling it, critical issues are left open only during a specified time. For example, the hardware/software platform on which the system is targeted should be resolved early in the project, such that purchasing decisions for the hardware can be done in time for the developers. Internal datastructures and algorithms, however, can be left open until after integration, allowing developers to revise them based on performance testing. Once the design window is past, the issue is resolved and can only be reopened in a subsequent iteration.

With the pace of technology innovation quickening, many changes can be anticipated when a dedicated part of the organization is responsible for technology management. Technology managers scan new technologies, evaluate them, and accumulate knowledge that is used during the selection of components. Often, change happens fast enough that companies are not aware of which technologies they provide themselves.

8.6. Exercises

1. Consider an on-line airline reservation system allowing airlines to offer tickets and travelers to purchase tickets. Reservations and seat assignments can be made when tickets are purchased. Multiple competing airlines can make offers and travelers from different countries can purchase tickets. Design a subsystem decomposition for this system taking into account hardware/software mapping, data storage, and access control issues (that is, ignore global software control and boundary conditions). State your design goals.
2. Assuming multiple redundant servers and a goal of no downtime, describe the following boundary conditions for the above airline reservation system:

- server down,
- server software upgrade, and
- plane ticket price upgrade.

3. Consider an existing, fax-based, problem reporting system for an aircraft manufacturer. You are part of a re-engineering project replacing the core of the system by a computer-based system, including a database and a notification system. The client insists on keeping the fax as an entry point for problem reports. You propose to replace it by email. Discuss the technical and managerial evaluation, pros, and cons of three possible solutions: fax only, email only, fax and email. Describe how you shield the rest of the system from this decision.
4. You are designing the access control policies for a web-based retail store. Customers access the store via the web, browse product information, input their address and payment information, and purchase products. Suppliers can add new products, update its information, and receive orders.The store owner sets the retail prices, makes tailored offers to customers based on their purchasing profiles, and provides marketing services. You have to deal with three actors: **`StoreAdministrator`**, **`Supplier`**, and **`Customer`**. Design an access control policy for all three actors. **`Customers`** can be created via the web while **`Suppliers`** are created by the **`StoreAdministrator`**. Include your rationale for every decision.

8.7. References

[Booch, 1994] G. Booch, *Object-Oriented Analysis and Design with Applications,* Second Edition, Benjamin / Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[Fowler, 1997] M. Fowler, *Analysis Patterns: Reusable Object Models,* Addison-Wesley, Reading, MA, 1997.

[Erman 1980] Erman, L. D., F. Hayes-Roth, et al. “The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty.” Computing Surveys 12(2): 213-253.

[Gamma et al., 1994] E. Gamma, R. Helm, R. Johnson, & J. Vlissides, *Design Patters: Elements of Reusable Object-Oriented Software,* Addison Wesley, Reading, MA, 1994.

[JCA, 1998] Java Cryptography Architecture, JDK Documentation, Javasoft, 1998.

[JDBC, 1998] JDBC^{TM} - Connecting Java and Databases, JDK Documentation, Javasoft, 1998.

[Microsoft, 1995] “Chapter 9: Open Database Connectivity (ODBC) 2.0 Fundamentals", *Microsoft Windows Operating Systems and Services Architecture,* Microsoft Corporation, 1995.

[Mowbray & Malveau, 1997] T.J. Mowbray & R.C. Malveau. *CORBA Design Patterns.* Wiley and Sons. 1997.

[Nye et. al, 1992] A. Nye & T. O'Reilly. *X Toolkit Intrinsics Programming Manual: OSF/Motif 1.1 Edition for X11 Release 5 The Definitive Guides to the X Windows Systems,* Vol. 4, O Reilly & Associates, Inc., Sebastopol, CA, 1992.

[OMG, 1995] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* John Wiley & Sons, New York, 1995.

[RMI, 1998] Java Remote Method Invocation, JDK Documentation, Javasoft, 1998.

[Rumbaugh et al., 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1991.

[Shaw & Garlan, 1996] M. Shaw & D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[Siewiorek & Swarz, 1992] *Reliable Computer Systems: Design and Evaluation.* Second edition. Digital Press, Burlington MA, 1992.

[Silberschatz et al, 1991] A. Silberschatz, J. Peterson, & P. Galvin. Operating System Concepts. Third Edition. Addison Wesley, Reading, MA, 1991.

[Tanenbaum, 1996] A.S. Tanenbaum. *Computer Networks.* Third Edition. Prentice Hall Inc., Upper Saddle River, NJ, 1996.