Chapter 1

Software Design

|  |
| --- |
| Objectives:   * To understand why need design * To know how to elicit design goals * To know how to specify design output * To know how to decompose a system |

S ystem design is the transformation of an analysis model into a system design model. During system design, developers define the design goals of the project and decompose the system into smaller subsystems that can be realized by individual teams. Developers also select strategies for building the system, such as the hardware/software strategy, the persistent data management strategy, the global control flow, the access control policy, and the handling of boundary conditions. The result of system design is a model that includes a subsystem decomposition and a clear description of each of these strategies. System design is not algorithmic. Developers have to make trade-offs among many design goals that often conflict with each other. They also cannot anticipate all design issues that they will face because they do not yet have a clear picture of the solution domain. System design is decomposed into several activities, each addressing part of the overall problem of decomposing the system:

* Identify design goals. Developers identify and prioritize the qualities of the system that they should optimize.
* Design the initial subsystem decomposition. Developers decompose the system into smaller parts based on the use case and analysis models. Developers use standard architectural styles as a starting point during this activity.
* Selection of off-the-shelf and legacy components. Off-the-shelf or legacy components realize specific subsystems more economically. The initial subsystem decomposition is adjusted to accommodate them.
* Mapping of subsystem to hardware. When the system is deployed on several nodes, additional subsystems are required for addressing reliability or performance issues.
* Design of a persistent data management infrastructure. Managing the states that outlives a single execution of the system has an impact on overall system performance and leads to the identification of one or more storage subsystems.
* Specification of an access control policy. Shared objects are protected so that user access to them is controlled. Access control impacts how objects are distributed within subsystems.
* Design of the global control flow. Determining the sequence of operations impacts the interface of the subsystems.
* Handling of boundary conditions. Once all subsystems have been identified, developers decide on the order in which individual components are started and shutdown.

We focus on the first two activities then refine the system decomposition. We then describe the management issues related to system design, such as documentation, responsibilities, and communication. We conclude this chapter by discussing in more detail system design issues and trade-offs using the ARENA case study.

# Introduction

System design, object design, and implementation constitute the construction of the system. During these three activities, developers bridge the gap between the requirements specification, produced during requirements elicitation and analysis, and 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 elicitation and analysis, we concentrated on the purpose and the functionality of the system. During system design, we focus on the processes, data structures, and software and hardware components necessary to implement it. The challenge of system design is that many conflicting criteria and constraints must 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 floor plan, 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. Note, however, that the architect does not need to know the exact contents of each room and the layout of the furniture; on the contrary, these decisions should be delayed and left to the client.

## Design Example



**FIGURE** 11-0 Example of residential house.

Figure 6-1 shows three successive revisions to a floor plan for a residential house. We set out to satisfy the following constraints:

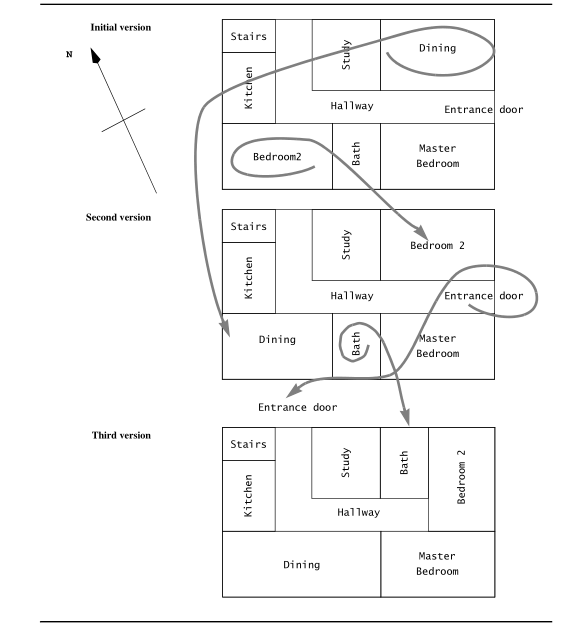
1. This house should have two bedrooms, a study, a kitchen, and a living room area.

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 living/dining area and in the master bedroom.

In the first version of our floor plan (at the top of Figure 6-1), 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 6-1). This also has 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 reorient bedroom 2 and move the bathroom closer to both bedrooms. The living area is increased, and we satisfied all original constraints.



**FIGURE** 11-1 Example of floor plan design. Three successive versions show how we minimize walking

distance and take advantage of sunlight.

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 without detailed knowledge of the layout of each individual room. Plans for plumbing, electrical lines, and heating ducts can proceed.

The design of a floor plan in architecture is similar to system design in software engineering (Table 6-1). The whole is divided into simpler components and interfaces, while taking into account nonfunctional and functional requirements. System design impacts implementation activities and results in costly rework if changed later. The design of individual components is delayed until later.

**Table** 11-1 Mapping of architectural and software engineering concepts.

|  |  |  |
| --- | --- | --- |
|  | Architectural concept | Software engineering concept |
| Components | Rooms | Subsystems |
| Intefaces | Doors and windows | Services |
| Nonfunctional requirements | Light, Space, Distance | Response time |
| Functional requirements | Sleep, Eat, Relax | Use cases |
| Costly rework | Moving walls | Change of subsystem interfaces |

Section2 provides a bird’s-eye view of system design and its relationship to analysis. Section 2 describes the concept of subsystems and subsystem decomposition. Section 4 describes system design activities and uses an example to illustrate how these building blocks can be used together.

## Analysis Result

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 ***functional model***, describing the system functionality from the actors’ point of view
* an ***structural model***, describing the entities manipulated by the system
* ***behavioral models*** for each use case, showing the sequence of interactions among objects participating in the use case.

The 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 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:

* ***design goals***, describing the qualities of the system that developers should optimize
* ***software architecture***, describing the subsystem decomposition in terms of subsystem responsibilities, dependencies among subsystems, subsystem mapping to hardware, and major policy decisions such as control flow, access control, and data storage
* ***boundary use cases***, describing the system configuration, startup, shutdown, and exception handling issues.

The design goals are derived from the nonfunctional requirements. Design goals guide the decisions to be made by the developers when trade-offs are needed. The subsystem decomposition constitutes the bulk of system design. Developers divide the system into manageable pieces to deal with complexity: each subsystem is assigned to a team and realized independently. For this to be possible, developers need to address system-wide issues when decomposing the system. In this chapter, we describe the concept of subsystem decomposition and discuss examples of generic system decompositions called “architectural styles.” In the next chapter, we describe how the system decomposition is refined to meet specific design goals. Figure 11-2 depicts the relationship of system design with other software engineering activities.

## Design Types

There are many different aspects of software design, including:

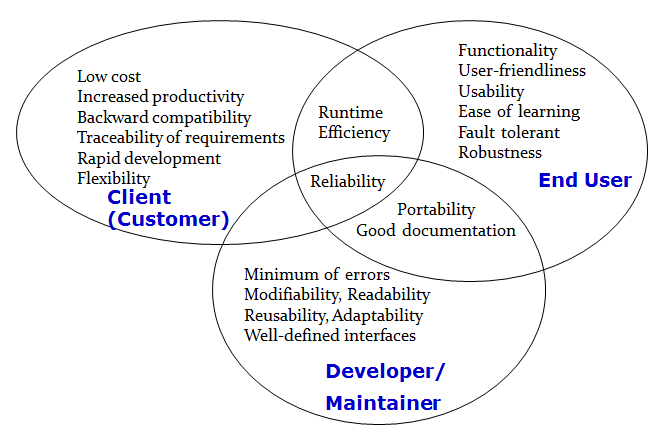
* System design: the division of software into subsystems and components, as well as the process of deciding how these will be connected and how they will interact, including determining their interfaces. We will talk more about this later in the chapter. Architecture design is commonly referred to as “software architecture”, although the latter term can also refer to the documentation produced, and the whole field of study.
* Object design: the design of the various features of classes such as associations, attributes, interactions and states.
* User interface design, focuses on the design of Web-based interfaces and graphical user interfaces (GUI) that use windows, menus, icons and a mouse.
* Database design: the design of how data is persistently stored so that it may be accessed by many programs and users, over an indefinite period of time.
* Physical architecture design: Describes the system’s hardware, software, and network environment.

# Identify Design Goals

In general, we can select design goals from a long list of highly desirable qualities. There are some examples of design goals, such as: Reliability, Modifiability, Maintainability, Understandability, Adaptability, Reusability, Efficiency, Portability, Traceability of requirements, Fault tolerance, Backward-compatibility, Cost-effectiveness, Robustness, High-performance, Good documentation, Well-defined interfaces, User-friendliness, Reuse of components, Rapid development, Minimum number of errors, Readability, Ease of learning, Ease of remembering, Ease of use, Increased productivity, Low-cost, Flexibility, et al.

## Design Goals

Sometimes, different stakeholders may have different design goals. For the client(customer), Low cost, Increased productivity, Backward compatibility, Traceability of requirements, Rapid development, Flexibility, Runtime Efficiency, Reliablity are the important design goals. For the end user, Functionality, User-friendliness, Usability, Ease of learning, Fault tolerant, Robustness, Runtime Efficiency, Portability, Good documentation, Reliablity are the important design goals. For the developer or maintainer, Minimum of errors, Modifiability, Readability, Reusability, Adaptability, Well-defined interfaces, Portability, Good documentation, Reliablity are the important design goals as Figure 11-1.



**FIGURE** 11-3 Stakeholders have different Design Goals.

Tables 6-2 through 6-6 list a number of possible design criteria. These criteria are organized into five groups: performance, dependability, cost, maintenance, and end user criteria. Performance, dependability, 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

**Performance criteria** (Table 6-2) 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** 11-1 Performance criteria.

|  |  |
| --- | --- |
| Design criterion | 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

**Dependability criteria** (Table 6-3) 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? Should the system tolerate errors and failures? Are security risks associated with the system environment? Are safety issues associated with system crashes?

**Table** 11-1 Dependability criteria.

|  |  |
| --- | --- |
| Design criterion | Definition |
| Robustness | Ability to survive invalid user input |
| Reliability | Difference between specified and observed behavior |
| Availability | Percentage of time that system can be used to accomplish normal tasks |
| Fault tolerance | Ability to operate under erroneous conditions |
| Security | Ability to withstand malicious attacks |
| Safety | Ability to avoid endangering human lives, even in the presence of errors and failures |

### Cost criteria

**Cost** **criteria** (Table 6-4) 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** 11-1 Cost criteria.

|  |  |
| --- | --- |
| Design criterion | Definition |
| Development cost | Cost of developing the initial system |
| Deployment cost | Cost of installing the system and training the users |
| Upgrade cost | Cost of translating data from the previous system. This criteria results in backward compatibility requirements |
| Maintenance cost | Cost required for bug fixes and enhancements to the system |
| Administration cost | Cost required to administer the system |

### Maintenance criteria

**Maintenance** **criteria** (Table 6-5) 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 successful the project will be and how long the system will be operational.

**Table** 11-1 Maintenance criteria.

|  |  |
| --- | --- |
| Design criterion | Definition |
| Extensibility | How easy is it to add functionality or new classes to 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? |

### End user criteria

**End** **user** **criteria** (Table 6-6) include qualities that are desirable from a users’ point of view, but have not yet been covered under the performance and dependability criteria. Is the software difficult to use and to learn? Can the users accomplish needed tasks on the system? Often these criteria do not receive much attention, especially when the client contracting the system is different from its users.

**Table** 11-1 End user criteria.

|  |  |
| --- | --- |
| Design criterion | 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? |

### Design goal trade off

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 6-7 lists several possible trade-offs.

**Table** 11-1 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). If the software does not meet memory space constraints, data can be compressed at the cost of speed. |
| Backward Compatibility vs Readability | In the past you would guarantee backward compatibility by introducing special switches. Nowadays, again with the use of design patterns, for example, the bridge pattern, you can keep a system backward compatibile and still keep it readable! |
| Functionality vs usability | Is a system with 100 functions usable? How about a large scale menu? |
| Low Cost vs Robustness | A low cost system does not check for errors when the user is entering wrong data |
| Efficiency vs Portability | Can you build a portable real-time game? How do you get high framerates. Special graphics routines that access the display buffer! That is usually not portable. If you write portable graphics code, say with the OpenGL, then you sometimes might not get the response times you are looking for. All special graphics code is still hand-tailored for a specific machine or graphics processor. |
| Rapid development vs. functionality | Let’s say your development time is 5 weeks, you have 5 programmers, your design window is 2 weeks, after design 3 programmers are leaving your company, and your delivery deadline cannot be moved. You are going to reduce the functionality. Not all the use cases in your model can be implemented nor delivered. |
| Cost vs. Reusability | This is an interesting trade-off, whose validity is changing right now. In the past, if you tried to make your design reusable you had to add extra effort. Recode your data structures (move from array of int to array of Generic). Moving from a 1-1 association to a many-many association involved more coding and more testing. Nowadays, with design patterns, this trade-off is changing a little bit. You can get reusability pretty cheap if you use design patterns! |
| Delivery time vs. functionality | If development runs behind schedule, a project manager can deliver less functionality than specified on time, or deliver the full functionality at a later time. Contract software usually puts more emphasis on functionality, whereas off-the-shelf software projects put more emphasis on delivery date. |
| Delivery time vs. quality | If testing runs behind schedule, a project manager can deliver the software on time with known bugs (and possibly provide a later patch to fix any serious bugs), or deliver the software later with fewer bugs. |
| Delivery time vs. staffing | If development runs behind schedule, a project manager can add resources to the project to increase productivity. In most cases, this option is only available early in the project: adding resources usually decreases productivity while new personnel are trained or 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.

## Robustness

We hava talked about many kinds of design goals, among them, most important design goals mainly include:

* Correctness: it means that the design meets the requirements and the code is executed according to the requirements. Generally, there may be multiple correct designs for a given requirement.
* Robustness: if the application can execute in case of errors, the design is robust; There are many kinds of errors. Users will make some errors when operating applications, and software developers will also make some errors when designing and coding.
* Reusability: refers to the code of the system that can be easily reused in the development of other systems.
* Maintainability: a maintainability design means that it can be easily modified.
* Efficiency: it includes time efficiency and space efficiency.

In software design, the primary goal is to meet the requirements, but also to meet some reasonable changes in the requirements.

Let's take a look at an example. A calculator program is implemented in Java language. The user inputs two numbers and divides them to get the result.

import java.io.\*;

public class Calculator {

public static void main(String[] args) throws IOException{

BufferedReader b = new BufferedReader(new InputStreamReader(System.in));

System.out.print("Input number A： ");

String A = b.readLine(); //How about input error？

System.out.print("Input number B： ");

String B = b.readLine();

int C = (new Integer(A).intValue())/(new Integer(B)).intValue();

//How about B=0？

System.out.println("Result is： "+C);

}

}

Analyze the characteristics of the implementation method and the problems to be considered:

First, the required functions are realized, so the correctness is satisfied. For example, enter 12 and 3, and the result is 4.

Secondly, if the user input error, the second number is 0, the system will crash or have unpredictable results, so there is a problem with robustness.

Then, the calculator class is specially designed for special problems, so it cannot be reused as a whole. Therefore, the reusability is not high.

In addition, it is difficult to modify the calculator class. Adding a subtraction function, you will find that adding the function requires modifying the original class, so there is a problem with maintainability.

Finally, how fast is the execution? How much memory is needed? This is a question of efficiency. Applications should run on real computers and be used by people. The memory of the computer is limited, and users are not willing to wait for the application to complete the operation for a long time.

If the design or implementation can handle all kinds of exceptions, such as data errors, user errors and environmental conditions, the design is robust. In order to improve robustness, it is necessary to prevent wrong input, including user input, data communication, method call of other applications and other non user input; We should also prevent development errors, including wrong design and wrong implementation.

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

public class CommandLineCalculator {

private int accumulatedValue = 0;

public CommandLineCalculator(){

super();

}

private static String getAnInputFromUser(){

try {

BufferedReader b = new BufferedReader(new InputStreamReader(System.in));

return (b.readLine());

}catch(IOException e){

System.out.println(e+"Input taken to be a single blank.");

return " ";

}

}

public static void main(String[] args){

System.out.print("Input number A： ");

String A = getAnInputFromUser();

System.out.print("Input number B： ");

String B = getAnInputFromUser();

int amountAdded = 0;

while(!A.equals("stop")&!B.equals("stop")){

try{

int a = (new Integer(A)).intValue(); // Throw error when it is not an integer

int b = (new Integer(B)).intValue(); // Throw error when it is not an integer

int c = a/b; //Throw error when b=0

System.out.println("The result is： "+c);

}catch(Exception e){

System.out.println("Sorry -- incorrect entry: Try again.");

}

System.out.print("Input number A： ");

A = getAnInputFromUser();

System.out.print("Input number B： ");

B = getAnInputFromUser();

}

System.out.println("Application ends.");

}

}

The code of calculator class can be modified. When the user inputs unreasonable data, the program prompts the user to input again, and the program can still continue to execute to obtain a robust interaction mode. Then the modified program has a certain robustness.

If the application saves data to files periodically to avoid application and system errors, the application will be more robust. If the data is sent to the remote memory at the cost of speed, the robustness can be further increased.

However, the above program only meets the current needs. The program is not easy to maintain, expand and reuse.

## Reusability

A good design should be easy to modify and reuse, which needs to consider maintainability and reusability. The so-called reusability means that the components of a software can be reused in different places of the same project or even in another project. One way to reduce costs and maximize productivity is to use the original work, that is, reuse. The application of Java API is a typical example of reuse. Java API is a collection of reusable classes.

Reuse can be realized from the aggregation of code, classes and related classes, such as design patterns, components, frameworks, software architecture and so on.

For a designed class, inheritance, aggregation, dependency and other technologies can be used to realize reuse. Specifically, the newly created class is directly designated as the subclass of the designed class, and the newly created class is defined by inheriting and modifying the properties and behavior of the parent class; Or, introduce the object of the designed class into the newly created class as the member variable of the newly created class. Then reuse the attributes and methods of the designed class through member variables in the newly created class; Or, introduce the object of the designed class into the newly created class as the parameter or return type of the method in the newly created class.

Reusability can be increased by reducing the coupling of classes. If class A is coupled with class B, class a cannot be used without class B, which reduces the reusability of class A.

Let's take a look at a customer class. You can use these three methods to realize reuse:

### Use inheritance to reuse

We know that when inheriting, the subclass will inherit the non private members in the parent class, which means that most of the methods and properties in the parent class need not be rewritten in the subclass. Here, for example, the compute () method in the customer class.

public class Customer {   
 Customer() {  
 }  
  
 int compute(){  
 //...  
 return 0;  
 }  
}

Then we can use inheritance to reuse the compute() method like following:

class SubCustomer extends Customer{  
 SubCustomer(){}  
   
 int Compute(){  
 int baseAmount = Compute();  
 //...  
 return baseAmount;  
 }  
}

### Use aggregation/composition to reuse

In addition to using inheritance, you can also use aggregation to reuse members in other classes. You can take other class objects as your members and call their properties and methods. For example, here the objcustomer class aggregates the customer class object, and then its properties and methods can be used.

class CustomerObj{  
 Customer customer = new Customer();  
 CustomerObj(){}  
  
 int Compute(){  
 int baseAmount = customer.Compute();  
 //...  
 return baseAmount;  
 }  
}

### Use dependency to reuse

Finally, dependency can also be used to realize reuse. The difference between dependency and aggregation is that the dependent object is not its own member, but by passing object parameters or creating an object in the method, and then reusing its members.

class CustomerOper{  
 SubCustomer(){}  
  
 int Compute(Customer customer){  
 int baseAmount = customer.Compute();  
 //...  
 return baseAmount;  
 }  
}

Or

class CustomerOper{  
 SubCustomer(){}  
  
 int Compute(){  
 Customer customer = new Customer();  
 int baseAmount = customer.Compute();  
 //...  
 return baseAmount;  
 }  
}

## Maintainability

One of the biggest challenges in software engineering is that the requirements change when the application has been completed; When you have decided how to do it, the goal of the application has changed. No matter how to improve the software requirements analysis, the requirements will still change in the process of project development. It's best to prevent customers from changing their needs after the design starts, but this is usually unlikely. Therefore, software design should be maintainable, and future changes should be considered in the design. This makes it easier for designers and programmers to modify the completed parts. In short, the reason for flexible design is that changes often occur.

The change of requirements is mainly reflected in the following aspects, and maintainability is also reflected in these aspects:

1. Add more functions of the same type. For example, in banking applications, more types of accounts can be processed without changing the existing design and modifying the existing code.
2. Add new functions, such as adding withdrawal function to the existing deposit function.
3. Modify functions, such as overdraft withdrawal.

For example, a calculator program realizes the basic addition operation. If the calculation code is mixed with other parts, such as the code for obtaining user input and output, there may be more changes when the requirements change in the future. In order to meet the maintainability, consider reconstructing the program and adding an abstract operation class to separate the calculation and display, that is, separate the business logic from the interface logic, so that the coupling between them is reduced and easy to expand and maintain. For example, if you want to write a calculator application with windows interface, you can reuse the operation calculation class. It can also be used for web programs, mobile phone and mobile system software.

public class Operation{   
 public int getResult(int numberA, int numberB){  
 int result = 0;  
 if(numberB != 0)  
 result = numberA/numberB;  
 return result;  
 }  
}

The problem with the above method is that if you want to add an addition operation, you can only change the operation class. Add an addition operation. The division operation that has been written should also be compiled. It is difficult to avoid accidentally changing the division operation into subtraction. It was originally to add a function, but the original function code running well has changed. Therefore, operations such as addition, subtraction, multiplication and division should be separated. Modifying one of them will not affect the others, and adding operation algorithms will not affect other codes, which requires inheritance and polymorphism. Simple factory design pattern uses inheritance and polymorphism to achieve better maintainability.

public class client {  
 public static void main(String[] args){  
 int intNumberA = 0, intNUmberB = 0;  
 try{  
 BufferedReader bufR = new BufferedReader(new InputStreamReader(System.in));  
 System.out.print("Input number A：");  
 try{  
 intNumberA = new Integer(bufR.readLine()).intValue();  
 }catch(Exception e){  
 System.out.println(e);  
 }  
 System.out.print("Input number B：");  
 try{  
 intNUmberB = new Integer(bufR.readLine()).intValue();  
 }catch(Exception e){  
 System.out.println(e);  
 }  
 int intResult = 0;  
 intResult = new Operation().getResult(intNumberA, intNUmberB);  
 System.out.println("Result is："+intResult);  
 }catch(Exception e){  
 System.out.println(e);  
 }  
 }  
}

The advantage of simple factory mode is that the factory class contains necessary logical judgment, but we still need to modify the factory class when we need to add new operations. The factory method design pattern can solve this problem. Looking at its class diagram, we can see that each calculation method has a corresponding factory class. If you want to add a new calculation method, you only need to add the corresponding engineering class and calculation implementation class, which conforms to the opening and closing principle and improves maintainability. We will introduce design patterns later.

## Efficiency

The goal of efficiency is to use the available memory to complete the work as soon as possible. Efficiency is considered from both time and space. The most ideal is to achieve efficiency in both time and space at the same time. In practice, they usually restrict each other and need time-space compromise.

### Design efficiency

In terms of simplicity, using convenient tools and languages will save a lot of design time, but programmers also lose control of the time and space of this system. For example, in order to facilitate programmers, the database management system provides a large number of built-in functions. However, in early military computing, database management system was rarely used because of the lack of control over space-time efficiency. With the development of database management system, this situation has changed, allowing the use of more existing tools.

If we consider the size of source code, we will not only reduce the cost of development and maintenance, but also sacrifice space-time efficiency.

### Implementation efficiency

Execution efficiency means that the application must complete specific functions within a specified time. Some real-time systems, such as aerospace control system, have the highest requirements for execution speed. It requires to complete the required functions in microsecond or even shorter time. For non real-time systems, execution speed is also important. If the application cannot complete the operation in time, the user will be patient quickly. For example, if a web page opens too slowly, users will no longer pay attention to it when it is fully opened. We can improve the execution efficiency from the aspects of remote call, loop, function call, object creation and so on.

### Storage efficiency

Storage efficiency is an important evaluation index for some applications that have certain requirements for memory capacity or hard disk space. Generally, there are three storage problems: the size of run-time ram, the size of code itself and the size of auxiliary memory such as disk. To achieve storage efficiency, the following measures can be taken:

1. Only the required data is stored, which requires a compromise between storage efficiency and data extraction and reorganization time.
2. Compress data, which requires a compromise between storage efficiency and data compression and decompression time.
3. Store data according to the relevant access frequency, which requires a compromise between storage efficiency and the time to determine the storage location.

# Package Diagrams

In UML, collaborations, partitions, and layers can be represented by a higher-level construct: a package. A package is a general construct that can be applied to any of the elements in UML models. In Chapter 5, we introduced the idea of packages as a way to group use cases together to make the use-case diagrams easier to read and to keep the models at a reasonable level of complexity. In Chapters 6 and 7, we did the same thing for class and communication diagrams, respectively.

Package diagram, a kind of structural diagram, shows the arrangement and organization of model elements in middle to large scale project. Package diagram can show both structure and dependencies between sub-systems or modules, showing different views of a system, for example, as multi-layered (aka multi-tiered) application - multi-layered application model.

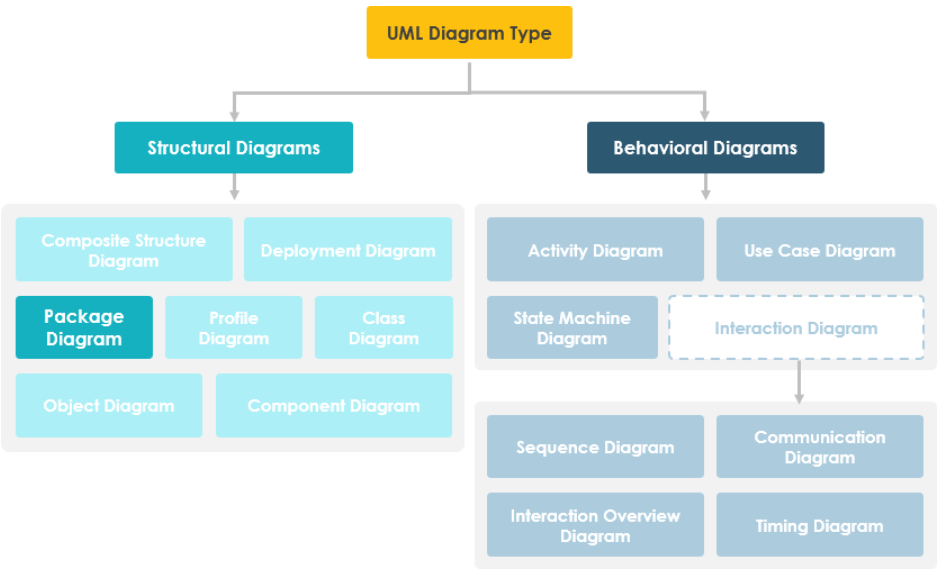


FIGURE UML Diagrams

Package diagrams are used to structure high level system elements. Packages are used for organizing large system which contains diagrams, documents and other key deliverables. Package Diagram can be used to simplify complex class diagrams, it can group classes into packages. A package is a collection of logically related UML elements. Packages are depicted as file folders and can be used on any of the UML diagrams.

## Purpose of Package Diagrams

Package diagrams are used to structure high level system elements. Packages are used for organizing large system which contains diagrams, documents and other key deliverables.

* Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.
* A package is a collection of logically related UML elements.
* Packages are depicted as file folders and can be used on any of the UML diagrams.

## Syntax for Package Diagram

Package diagram is used to simplify complex class diagrams, you can group classes into packages. A package is a collection of logically related UML elements.

The diagram below is a business model in which the classes are grouped into packages:

* Packages appear as rectangles with small tabs at the top.
* The package name is on the tab or inside the rectangle.
* The dotted arrows are dependencies.
* One package depends on another if changes in the other could possibly force changes in the first.

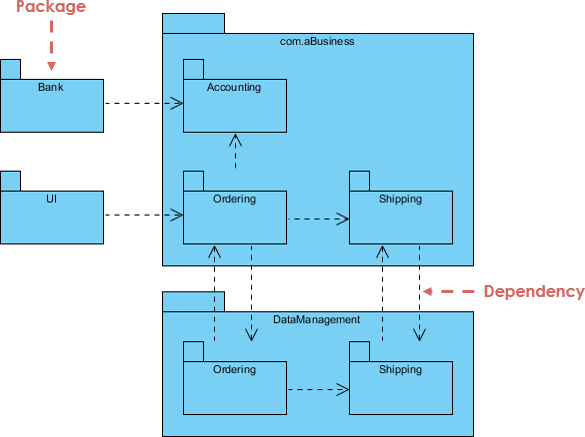


FIGURE Syntax for Package Diagram

In a package diagram, it is useful to depict a new relationship, the dependency relationship. A dependency relationship is portrayed by a dashed arrow. A dependency relationship represents the fact that a modification dependency exists between two packages. The diagram can be grouped into packages which appears as rectangles with small tabs at the top. And its name is on the tab or inside the rectangle. The dotted arrows are dependencies. One package depends on another if changes in the other could possibly force changes in the first.

### Package

The package is the UML mechanism for grouping items (including other packages). Package diagram follows hierarchal structure of nested packages. Atomic module for nested package are usually class diagrams. There are few constraints while using package diagrams:

* Package name should not be the same for a system, however classes inside different packages could have the same name.
* Packages can include whole diagrams, name of components alone or no components at all.
* Each package has its own namespace within which all names must be unique.
* Every model element is owned by one package.
* The packages form a hierarchy.
* The set of UML building blocks consists of things, relationships, and diagrams.

At the class level, there could be many causes for dependencies among classes. For example, if the protocol for a method is changed, then this causes the interface for all objects of this class to change. Therefore, all classes that have objects that send messages to the instances of the modified class may have to be modified. Capturing dependency relationships among the classes and packages helps the organization in maintaining object-oriented information systems.

As already stated, collaborations, partitions, and layers are modeled as packages in UML. Furthermore, collaborations are normally factored into a set of partitions, which are typically placed on a layer. In addition, partitions can be composed of other partitions.

Also, it is possible to have classes in partitions, which are contained in another partition, which is placed on a layer. All these groupings are represented using packages in UML.

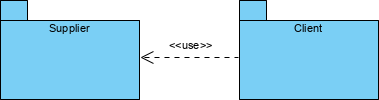
Remember that a package is simply a generic, grouping construct used to simplify UML models through the use of composition.

### Dependency

They are different types of dependency relationship represented by using stereotypes, perhaps we can define some new stereotype to serve for your specific purpose.

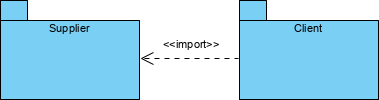
1. **Use dependency**

An element in the client package uses a public element in the supplier package in some way – the client depends on the supplier. If a package dependency is shown without a stereotype, then <<use>> should be assumed.



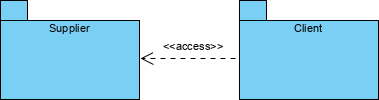
1. **Import dependency**

Public elements of the supplier namespace are added as public elements to the client namespace. Elements in the client can access all public elements in the supplier using unqualified names.



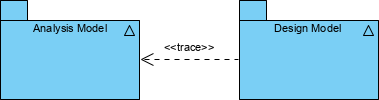
1. **Access dependency**

Public elements of the supplier namespace are added as private elements to the client namespace. Elements in the client can access all public elements in the supplier using unqualified names.



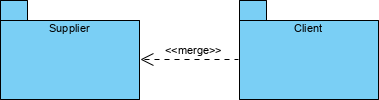
1. **Trace dependency**

<<trace>> usually represents a historical development of one element into another more developed version – it is usually a relationship between models rather than elements (an extra-model relationship),



1. **Merge dependency**

Public elements of the supplier package are merged with elements of the client package. This dependency is only used in metamodeling – you should not encounter it in ordinary OO analysis and design.



### Examples

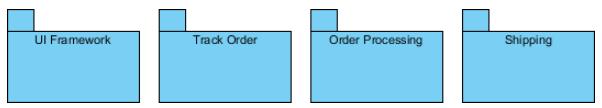
We are going to design package diagram for "Track Order" scenario for an online shopping store. Track Order module is responsible for providing tracking information for the products ordered by customers. Customer types in the tracking serial number, Track Order modules refers the system and updates the current shipping status to the customer.

Based on the project Description we should first identify the packages in the system and then related them together according to the relationship:

**Identify the packages of the system**

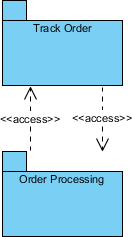
There is a track order module, it has to talk with other module to know about the order details, let us call it "Order Details".

Next after fetching Order Details it has to know about shipping details, let us call that as "Shipping".

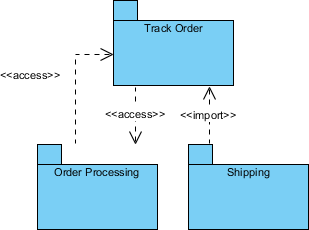


Identify the dependencies in the System

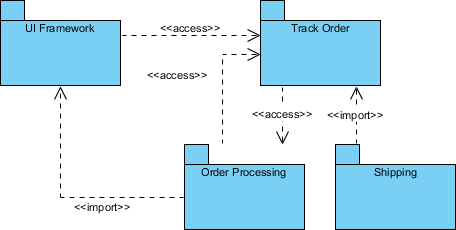
Track order should get order details from "Order Details" and "Order Details" has to know the tracking info given by the customer. Two modules are accessing each other which suffices <<access>> dual dependency



To know shipping information, "Shipping" can import "Track Order" to make the navigation easier.



Finally, Track Order dependency to UI Framework is also mapped which completes our Package Diagram for Order Processing subsystem.



## Identifying Packages and Creating Package Diagrams

In this section, we describe a simple five-step process to create package diagrams：

1. Set the context.

2. Cluster classes together based on shared relationships.

3. Model clustered classes as a package.

4. Identify dependency relationships among packages.

5. Place dependency relationships between packages.

The first step is to set the context for the package diagram. Remember, packages can be used to model partitions and/or layers. Revisiting the appointment system again, let’s set the context as the problem domain layer.

The second step is to cluster the classes together into partitions based on the relationships that the classes share. The relationships include generalization, aggregation, the various associations, and the message sending that takes place between the objects in the system. To identify the packages in the appointment system, we should look at the different analysis models [e.g., the class diagram (see Figure 6-2), the communication diagrams (see Figure 7-5)], and the CRUD matrix (see Figure 7-14). Classes in a generalization hierarchy should be kept together in a single partition.

The third step is to place the clustered classes together in a partition and model the partitions as packages. Figure 8-17 portrays five packages: PD Layer, Person Pkg, Patient Pkg, Appt Pkg, and Treatment Pkg.

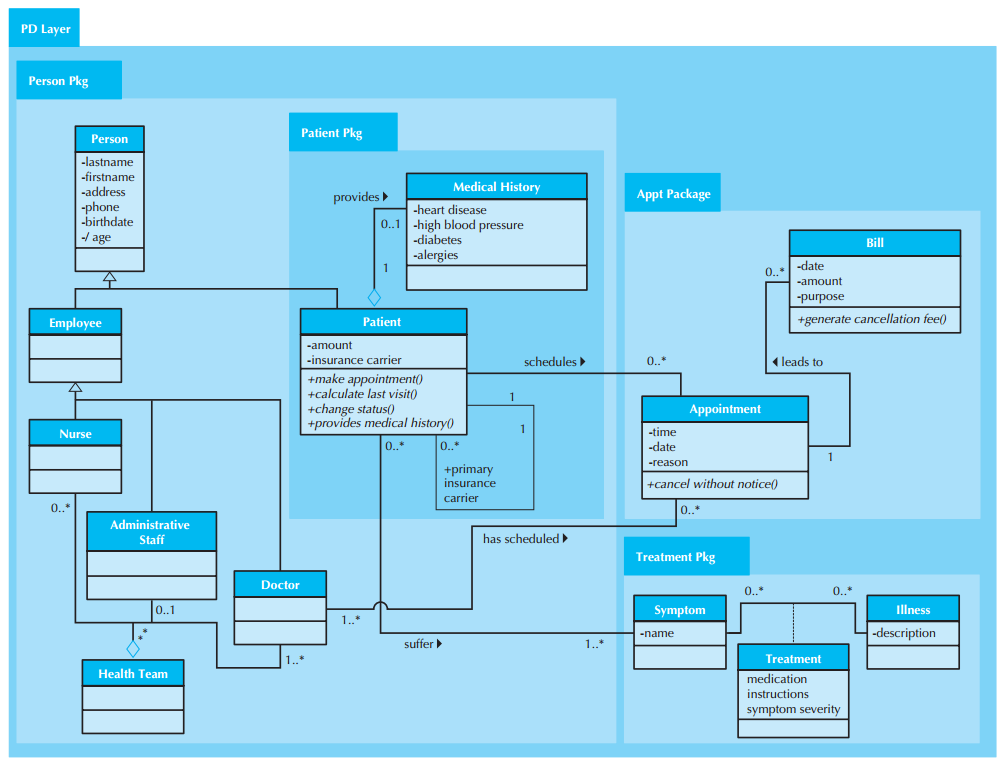


FIGURE Package Diagram of the PD Layer for the Appointment System

The fourth step is to identify the dependency relationships among the packages. In this case, we review the relationships that cross the boundaries of the packages to uncover potential dependencies. In the appointment system, we see association relationships that connect the Person Pkg with the Appt Pkg (via the association between the Doctor class and the Appointment class), and the Patient Pkg, which is contained within the Person Pkg, with the Appt Pkg (via the association between the Patient and Appointment classes) and the Treatment Pkg (via the association between the Patient and Symptom classes).

The fifth step is to place the dependency relationships on the evolved package diagram. In the case of the Appointment system, there are dependency relationships between the Person Pkg and the Appt Pkg and the Person Pkg and the Treatment Pkg. To increase the understandability of the dependency relationships among the different packages, a pure package diagram that shows only the dependency relationships among the packages can be created (see Figure 8-18).

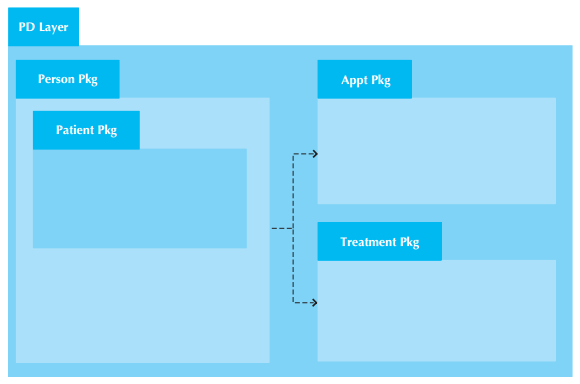


FIGURE Overview Package Diagram of the PD Layer for the Appointment System

## Verifying and Validating Package Diagrams

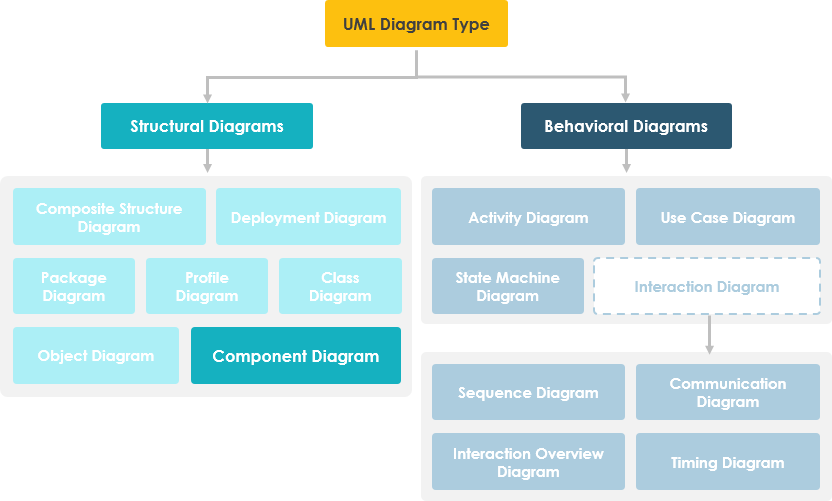
Like all the previous models, package diagrams need to be verified and validated. In this case, the package diagrams were derived primarily from the class diagram, the communications diagrams, and the CRUD matrix. Only two areas need to be reviewed.

First, the identified packages must make sense from a problem domain point of view. For example, in the context of an appointment system, the packages in Figure 8-18 (Person, Patient, Appt, and Treatment) seem to be reasonable.

Second, all dependency relationships must be based on message-sending relationships on the communications diagram, associations on the class diagram, and cell entries in the CRUD matrix. In the case of the appointment system, the identified dependency relationships are reasonable.

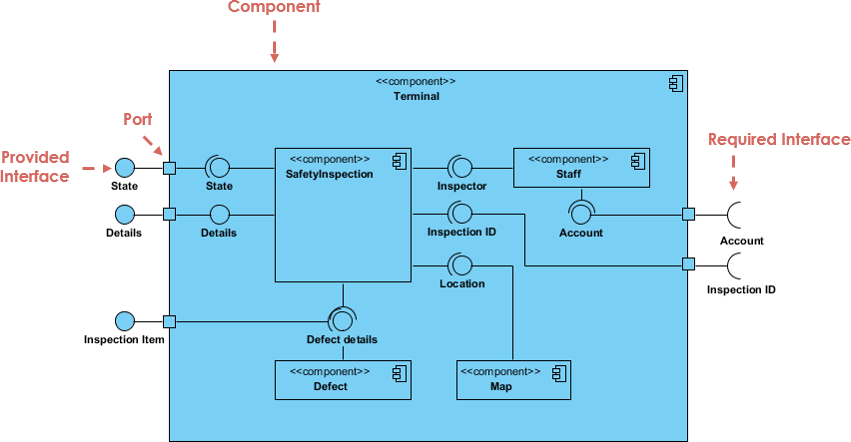
# Component Diagram

UML Component diagrams are used in modeling the physical aspects of object-oriented systems that are used for visualizing, specifying, and documenting component-based systems and also for constructing executable systems through forward and reverse engineering. Component diagrams are essentially class diagrams that focus on a system's components that often used to model the static implementation view of a system.



## Overview

A component diagram breaks down the actual system under development into various high levels of functionality. Each component is responsible for one clear aim within the entire system and only interacts with other essential elements on a need-to-know basis.



The example above shows the internal components of a larger component:

The data (account and inspection ID) flows into the component via the port on the right-hand side and is converted into a format the internal components can use. The interfaces on the right are known as required interfaces, which represents the services the component needed in order to carry out its duty.

The data then passes to and through several other components via various connections before it is output at the ports on the left. Those interfaces on the left are known as provided interface, which represents the services to deliver by the exhibiting component.

It is important to note that the internal components are surrounded by a large 'box' which can be the overall system itself (in which case there would not be a component symbol in the top right corner) or a subsystem or component of the overall system (in this case the 'box' is a component itself).

## Elements

### Component

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. In UML 2, a component is drawn as a rectangle with optional compartments stacked vertically. A high-level, abstracted view of a component in UML 2 can be modeled as:

* A rectangle with the component's name
* A rectangle with the component icon
* A rectangle with the stereotype text and/or icon

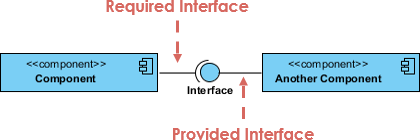
Looks of a Component

Figure Looks of a Component

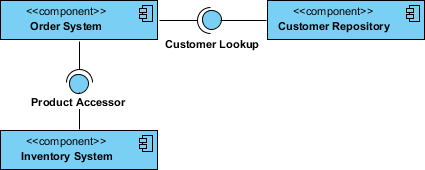
### Interface

In the example below shows two type of component interfaces:

* Provided interface symbols with a complete circle at their end represent an interface that the component provides - this "lollipop" symbol is shorthand for a realization relationship of an interface classifier.
* Required Interface symbols with only a half circle at their end (a.k.a. sockets) represent an interface that the component requires (in both cases, the interface's name is placed near the interface symbol itself).

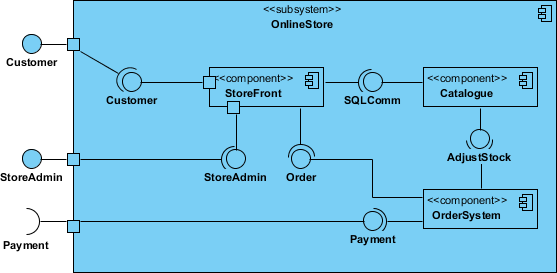


For example，here is the using interface of order system:



### Subsystem

The subsystem classifier is a specialized version of a component classifier. Because of this, the subsystem notation element inherits all the same rules as the component notation element. The only difference is that a subsystem notation element has the keyword of subsystem instead of component.



### Port

Ports are represented using a square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component.



### Relationships

Graphically, a component diagram is a collection of vertices and arcs and commonly contain components, interfaces and dependency, aggregation, constraint, generalization, association, and realization relationships. It may also contain notes and constraints.

|  |  |
| --- | --- |
| Relationships | Notation |
| Association:   * An association specifies a semantic relationship that can occur between typed instances. * It has at least two ends represented by properties, each of which is connected to the type of the end. More than one end of the association may have the same type. | Component Diagram Notation: Association |
| Composition:   * Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. * If a composite is deleted, all of its parts are normally deleted with it. | Component Diagram Notation: Composition |
| Aggregation   * A kind of association that has one of its end marked shared as kind of aggregation, meaning that it has a shared aggregation. | Component Diagram Notation: Aggregation |
| Constraint   * A condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element. | Component Diagram Notation: Constraint |
| Dependency   * A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. * This means that the complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s). | Component Diagram Notation: Dependency |
| Links:   * A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. * Each instance of the specific classifier is also an indirect instance of the general classifier. * Thus, the specific classifier inherits the features of the more general classifier. | Component Diagram Notation: Generalization |

## Examples

### Modeling Source Code

Either by forward or reverse engineering, identify the set of source code files of interest and model them as components stereotyped as files.

For larger systems, use packages to show groups of source code files.

Consider exposing a tagged value indicating such information as the version number of the source code file, its author, and the date it was last changed. Use tools to manage the value of this tag.

Model the compilation dependencies among these files using dependencies. Again, use tools to help generate and manage these dependencies.

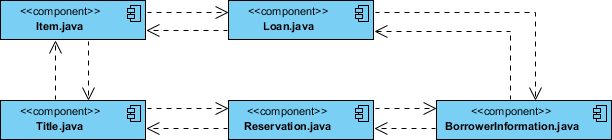


Figure Component Example - Java Source Code

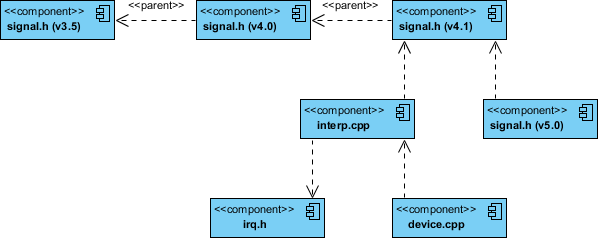


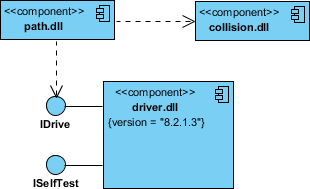
Figure Component Diagram Example - C++ Code with versioning

### Modeling an Executable Release

Identify the set of components you'd like to model. Typically, this will involve some or all the components that live on one node, or the distribution of these sets of components across all the nodes in the system.

Consider the stereotype of each component in this set. For most systems, you'll find a small number of different kinds of components (such as executables, libraries, tables, files, and documents). You can use the UML's extensibility mechanisms to provide visual cues(clues) for these stereotypes.

For each component in this set, consider its relationship to its neighbors. Most often, this will involve interfaces that are exported (realized) by certain components and then imported (used) by others. If you want to expose the seams in your system, model these interfaces explicitly. If you want your model at a higher level of abstraction, elide these relationships by showing only dependencies among the components.



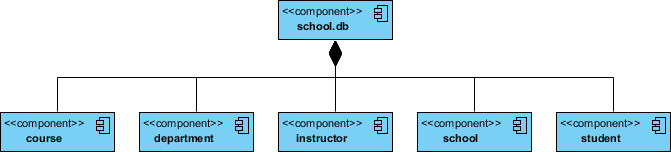
### Modeling a Physical Database

Identify the classes in your model that represent your logical database schema.

Select a strategy for mapping these classes to tables. You will also want to consider the physical distribution of your databases. Your mapping strategy will be affected by the location in which you want your data to live on your deployed system.

To visualize, specify, construct, and document your mapping, create a component diagram that contains components stereotyped as tables.

Where possible, use tools to help you transform your logical design into a physical design.



# System Decomposition

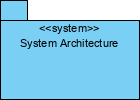
In this section, we describe subsystem decompositions and their properties in more detail. First, we define the concept of subsystem and its relationship to classes. Next, we look at the interface of **subsystems**: subsystems provide services to other subsystems. A **service** is a set of related operations 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 it provides. Next, we look at two properties of subsystems, coupling and cohesion.

## System

A system is represented as a package with the stereotype of <<system>> as shown in Figure below. The system represents all the model elements that pertain to the particular project. You can also break a system into <<business systems>> and <<application systems>> when building more detailed models to make them smaller and more workable. In the UML, packages are represented as folders.

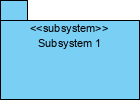
Stereotypes are a high-level classification of an object that gives you some indication of the kind of object it is. Classes can be grouped under stereotypes, whose name is written between matched guillemots (<< >>), over the class name.

A stereotype enables you to extend the UML to fit your modeling needs more specifically. A stereotype is a UML modeling element that extends the existing elements. Stereotyping a UML element causes it to act as something else that has specific properties. A stereotype is represented as <<stereotype>> on the element being stereotyped.

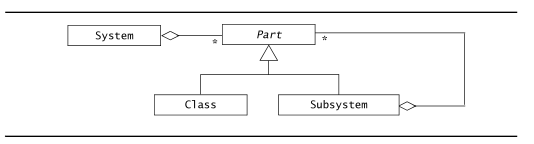


## Subsystems and Classes

A subsystem is a grouping of model elements that are part of the overall system. Subsystems, like systems, are stereotyped packages with the stereotype of <<subsystem>> as shown in the Figure below.



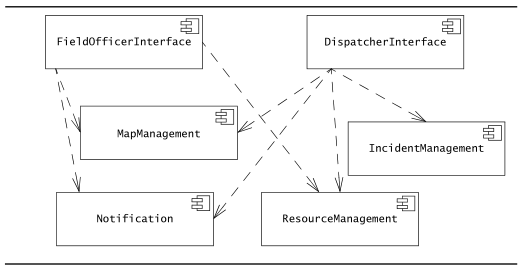
In order to reduce the complexity of the application domain, we identified smaller parts called “classes” and organized them into packages. Similarly, to reduce the complexity of the solution domain, we decompose a system into simpler parts, called “subsystems,” which are made of a number of solution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



**FIGURE** 11-3 Subsystem decomposition (UML class diagram).

For example, the accident management system we previously described can be decomposed into a *DispatcherInterface* subsystem, realizing the user interface for the Dispatcher ; a *FieldOfficerInterface* subsystem, realizing the user interface for the FieldOfficer ; an *IncidentManagement* subsystem, responsible for the creation, modification, and storage of Incidents ; a *ResourceManagement* subsystem, responsible for tracking available Resources (e.g., FireTrucks and Ambulances ); a *MapManagement* for depicting Maps and Locations; and a *Notification* subsystem, implementing the communication between FieldOfficer terminals and Dispatcher stations.

This subsystem decomposition is depicted in Figure 6-4 using UML components. Components are depicted as rectangles with the component icon in the upper right corner. Dependencies among components can be depicted with dashed stick arrows. In UML, components can represent both logical and physical components. A **logical component** corresponds to a subsystem that has no explicit run-time equivalent, for example, individual business components that are composed together into a single run-time application logic layer. A **physical component** corresponds to a subsystem that as an explicit run-time equivalent, for example, a database server.



**FIGURE** 11-4 Subsystem decomposition for an accident management system (UML component diagram).

Subsystems are shown as UML components. 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, so developers use conventions for grouping classes (e.g., a subsystem can be represented as a directory containing all the files that implement the subsystem). Whether or not subsystems are explicitly represented in the programming language, developers need to document carefully the subsystem decomposition as subsystems are usually realized by different teams.

## Services and Subsystem Interface

A subsystem is characterized by the services it provides to other subsystems. A **service** is a set of related operations that share a common purpose. A subsystem providing a notification service, for example, defines operations to send notices, look up notification channels, and subscribe and unsubscribe to a channel. The set of operations of a subsystem that are available to other subsystems form the **subsystem interface**. The subsystem interface includes the name of the operations, their parameters, their types, and their return values. System design focuses on defining the services provided by each subsystem, that is, enumerating the operations, their parameters, and their high-level behavior. Object design will focus on the **application programmer interface** (API), which refines and extends the subsystem interfaces. The API also includes the type of the parameters and the return value of each operation.

Provided and required interfaces can be depicted in UML with **assembly connectors**, also called **ball-and-socket connectors**. The provided interface is shown as a ball icon (also called lollipop) with its name next to it. A required interface is shown as a socket icon. The dependency between two subsystems is shown by connecting the corresponding ball and socket in the component diagram.

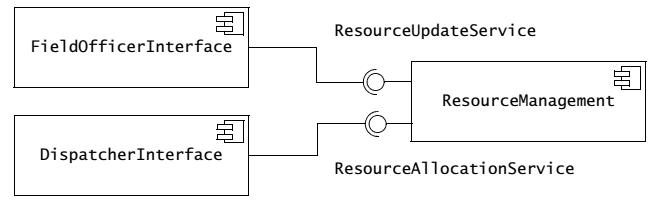


Figure 6-5 depicts the dependencies among the *FieldOfficerInterface*, *DispatchterInterface* and *ResourceManagement* subsystems. The *FieldOfficerInterface* requires the *ResourceUpdateService* to update the status and location of the *FieldOfficer* . The *DispatcherInterface* requires the *ResourceAllocationService* to identify available resources and allocating them to new Incidents . The *ResourceManagement* subsystem provides both services. Note that we use the ball-and-socket notation when the subsystem decomposition is already fairly stable and that our focus has shifted from the identification of subsystems to the definition of services. During the early stages of system design, we may not have such a clear understanding of the allocation of functionality to subsystems, in which case we use the dependency notation (dashed arrows) of Figure 6-4.

The definition of a subsystem in terms of the services it provides helps us focus on its interface as opposed to its implementation. When writing a subsystem interface, one should strive to minimize the amount of information provided about the implementation. For example, a subsystem interface should not refer to internal data structures, such as linked lists, arrays, or hash tables. 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.

## Component vs Subsystem vs Package

Component is a physical component that follows a set of interfaces and provides implementation in the system. It usually refers to the physical implementation of development and runtime classes. Components generally represent actual and physical objects. Components can be: program source code, subsystem, dynamic link library, etc. components generally contain many classes and implement many interfaces. Components are classified, and configuration components are the basis for forming executable files, such as dynamic link library and binary executable; Work product components are class elements of configuration components, such as data files and program source code; The execution component is the running result produced by the final runnable program.

A subsystem is a component, usually including many smaller components, which is a large component.

A package is a grouping of elements, which forms a higher-level unit logically. Used to group classes. Similar to namespace, the name is unique in the same package.

In general, the similarity between components and packages is that they are a large "aggregate". The component is relatively complete. It is a complete individual with independent functions. For the outside, the internal components are not important, but the interfaces it can provide. The package is just a collection of multiple individuals, like a class. There is no independent function, and the internal individuals have functions. For the outside, the package itself is not important, but the classes inside the package play a role.

## Design Method

There are several fundamentally different sequences with which you can make design decisions. In top-down design, you start with the very high-level structure of the system. You then gradually work down towards detailed decisions about low-level constructs. Examples of high-level issues that are approached first in top-down design include the software architecture and the kind of database that will be used. After many higher-level decisions are made, you finally arrive at detailed decisions such as the format of particular data items, and the individual algorithms that will be used.

The inverse approach, bottom-up design, involves first making decisions about reusable low-level utilities and then deciding how these will be put together to create high-level constructs.

A mix of top-down and bottom-up design is normally used. Top-down design is almost always needed to give the system a good structure. On the other hand, some bottom-up design helps ensure that you create reusable components that can be used in several places in the overall system.

# System Design Principles

In this section we introduce you to general philosophies you should apply whenever you are designing software. Applying these principles diligently will result in designs that have many advantages over designs in which the principles were not applied.

Some overall goals we want to achieve when doing good design are:

* Increasing profit by reducing cost and increasing revenue. For most organizations, this is the central objective. However, there are a number of ways to reduce cost, and also many different ways to increase the revenue generated by software.
* Ensuring that we actually conform to the requirements, thus solving the customers’ problems.
* Accelerating development. This helps reduce short-term costs, helps ensure the software reaches the market soon enough to compete effectively, and may be essential to meet some deadline faced by the customer.
* Increasing qualities such as usability, efficiency, reliability, maintainability and reusability. These can help reduce costs and also increase revenues.

## Divide and Conquer

The divide and conquer principle dates back to the earliest days of organized human activity. Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things. Military campaigns are waged this way: commanders try to avoid fighting on all fronts at once. Cars are also built using the divide and conquer strategy: some people design the engines while others design the body, etc. Furthermore, the task of assembling the car is also divided into smaller, more manageable chunks – each assembly-line worker will focus on one small task.

In software engineering, the divide and conquer principle is applied in many ways. We have already seen how the process of development is divided into activities such as requirements gathering, design and testing. In this section we will look at how software systems themselves can be divided.

Dividing a software system into pieces has many advantages:

■ Separate people can work on each part. The original development work can therefore be done in parallel.

■ An individual software engineer can specialize in his or her component, becoming expert at it. It is possible for someone to know everything about a small part of a system, but it is not possible to know everything about an entire system.

■ Each individual component is smaller, and therefore easier to understand.

■ When one part needs to be replaced or changed, this can hopefully be done without having to replace or extensively change other parts.

■ Opportunities arise for making the components reusable.

A software system can be divided in many ways:

■ A distributed system is divided up into clients and servers.

■ A system is divided up into subsystems.

■ A subsystem can be divided up into one or more packages.

■ A package is composed of classes.

■ A class is composed of methods.

## High Cohesion

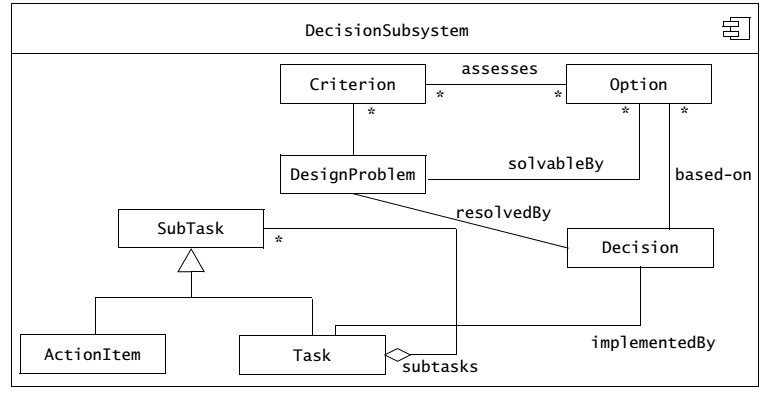
The cohesion principle is an extension of the divide and conquer principle –divide and conquer simply says to divide things up into smaller chunks.

Cohesion says to do it intelligently: yes, divide things up, but keep things together that belong together.

A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things. This makes the system as a whole easier to understand and change.

Listed below are several important types of cohesion that designers should try to achieve. Table 9.1 summarizes these types of cohesion, starting with the most desirable.

High cohesion can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries. Here are some questions to ask: Does one subsystem always call another one for a specific service? If the answer is yes, consider moving them together into the same subystem. Another question is which of the subsystems call each other for services? Can this be avoided by restructuring the subsystems or changing the subsystem interface? Can the subsystems even be hierarchically ordered (in layers)? For example, following diagram show us a system called DecisionSubsystem contains many clalsses.



However, there are classes such as SubTask and subclass AtionItem, Task has low relationship with other classes. So we consider separate this subsystem into two smaller subsystems, which looks like following diagram.

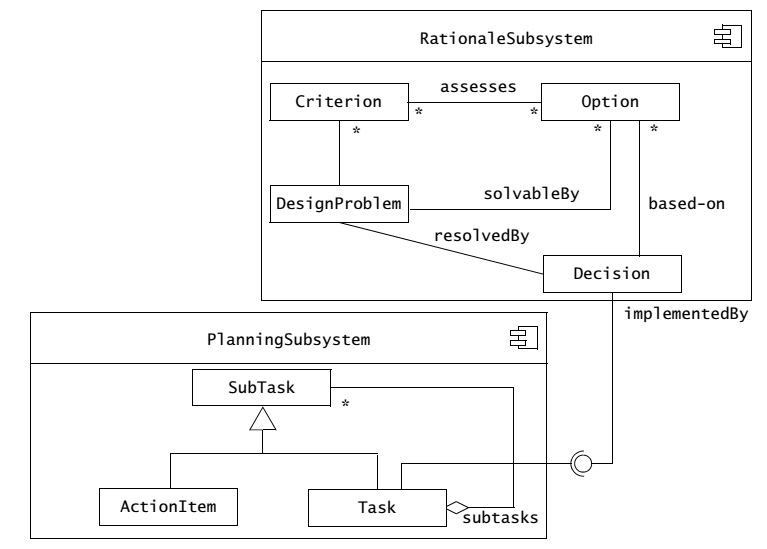


Table. The different types of cohesion, ordered from highest to lowest in terms of the precedence you should normally give them when making design decisions

|  |  |
| --- | --- |
| **Coupling type** | **Comments** |
| Functional | Facilities are kept together that perform only one computation with no side effects. Everything else is kept out. |
| Layer | Related services are kept together, everything else is kept out, and there is a strict hierarchy in which higher-level services can access only lower-level services. Accessing a service may result in side effects. |
| Communicational | Facilities for operating on the same data are kept together, and everything else is kept out. Good classes exhibit communicational cohesion. |
| Sequential | A set of procedures, which work in sequence to perform some computation, is kept together. Output from one is input to the next. Everything else is kept out. |
| Procedural | A set of procedures, which are called one after another, is kept together. Everything else is kept out. |
| Temporal | Procedures used in the same general phase of execution, such as initialization or termination, are kept together. Everything else is kept out. |
| Utility | Related utilities are kept together, when there is no way to group them using a stronger form of cohesion. |

### Functional cohesion

This is achieved when a module only performs a single computation, and returns a result, without having side effects.

A module lacks side effects if performing the computation leaves the system in the same state it was in before performing the computation. The result computed by the module is the only thing that should have an effect on subsequent computations.

The inputs to a functionally cohesive module typically include function parameters, but they can also include files or some other stream of data.

Whenever exactly the same inputs are provided, the module will always compute the same result. The result is often a simple return value, but can also be a more complex data structure.

Modules that update a database or create a new file are not functionally cohesive since they have side effects in the database or file-system respectively.

Similarly, a module that interacts with the user is not functionally cohesive: prompting the user is a kind of output; therefore it violates the rule that the only output of a functionally cohesive module is the result returned at the end of execution.

The following are some examples of modules that can be designed to be functionally cohesive:

■ A module that computes a mathematical function such as sine or cosine.

■ A module that takes a set of equations and solves for the unknowns.

■ A module in a chemical factory that takes data from various monitoring devices and computes the yield of a chemical process as a percentage of the theoretical maximum.

A functionally cohesive module can call the services of other modules, but the called modules must preserve the functional cohesion. For example, a module that computes a mathematical function can certainly call modules that perform other mathematical functions.

■ It is easier to understand a module when you know that all it does is generate one specific output and has no side effects.

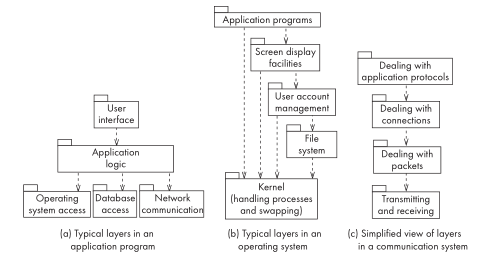
■ Due to its lack of side effects, a functionally cohesive module is much more likely to be reusable.

■ It is easier to replace a functionally cohesive module with another that performs the same computation. Being able to make such easy replacements greatly assists maintenance. In the case of a non-functionally cohesive module that has side effects, you would have to verify that any replacement also has precisely the same side effects. Even if the side effects were carefully documented, doing such verification is time-consuming and error-prone. Furthermore, maintainers often fail to pay attention to the presence of side effects.

### Layer cohesion

This is achieved when the facilities for providing a set of related services to the user or to higher-level layers are kept together, and everything else is kept out.

To have proper layer cohesion, the layers must form a hierarchy. Higher layers can access services of lower layers, but it is essential that the lower layers do not access higher layers. This is illustrated in Figure 9.3.



**FIGURE** 6-1 Examples of the use of layers. Higher layers can call on the services of lower layers, but not the other way around.

An individual service in a layer may have functional cohesion. However, this is not necessary – side effects are allowed, and are often essential.

The set of related services that could form a layer might include:

■ services for computation;

■ services for transmission of messages or data;

■ services for storage of data;

■ services for managing security;

■ services for interacting with users;

■ services to access the operating system;

■ services to interact with the hardware.

For example, if a system is to interface with a particular sound card, then a module should be created specifically to interact with that card. Furthermore, all the code for directly accessing the card should be in this module and the module should do nothing else except interact with the card.

The set of procedures or methods through which a layer provides its services is commonly called an application programming interface (API). The specification of the API must describe the protocol that higher-level layers use to access it, as well as the semantics of each service, including the side effects.

Advantages of layer cohesion are:

■ You can replace one or more of the top-level layers without having any impact on the lower-level layers.

■ You know you can replace a lower layer with an equivalent layer, because you know it does not access higher layers. To do this, however, you have to replicate all aspects of the API, so that upper layers will continue to work the same way.

### Communicational cohesion

This is achieved when modules that access or manipulate certain data are kept together (e.g. in the same class) – and everything else is kept out.

The term ‘communicational’ is used for historical reasons. You can remember it by thinking of the following: All the procedures that ‘communicate’ with the data are kept together.

For example, a class called Employee would have good communicational cohesion if all the system’s facilities for storing and manipulating employee data were contained in this class, and if the class did not do anything other than manage employee data.

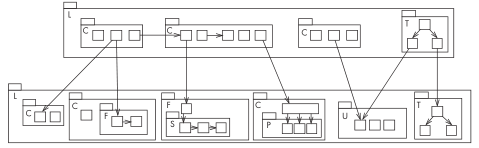
As another example of communicational cohesion, imagine a module that updates a database, and a second module that keeps a history log of the changes to the database. Since both database and log file are representations of the same data, both modules should be kept together in a higher-level module or subsystem.

A communicational cohesive module can be embedded in a layer. In other words, part of a layer’s API can involve manipulating a particular class of data. The objects manipulated by the layer may be returned to higher layers in response to calls to the API.

The big advantage of communicational cohesion is the same key advantage we ascribed earlier in this book to object orientation: when you need to make changes to the data, you will find all the code in one place.

You should not sacrifice layer cohesion to achieve communicational cohesion: for example, even though objects may be stored in a database or on a remote host, a class must only load and save objects using the services in the API of lower layers.

Figure 9.4 shows several examples of communicational cohesive modules (marked with a ‘C’). These exist inside layers (marked ‘L’) and call on services in their own layer as well as lower layers. The services they call on may be in modules with other types of cohesion.



**FIGURE** 6-1 Cohesive modules, nested inside each other, using the services of other modules.

The modules are labeled using the first letter of the type of cohesion they represent.

### Sequential cohesion

This is achieved when a series of procedures, in which one procedure provides input to the next, are kept together – and everything else is kept out. This is illustrated in Figure 9.4 by the module marked ‘S’.

Your objective should be to achieve sequential cohesion, once you have already achieved the other types of cohesion listed above. Methods in two different classes might provide inputs to each other and be called in sequence; but they would each be kept in their own class, since communicational cohesion is more important than sequential cohesion.

As an example of sequential cohesion, imagine a text recognition subsystem. One module is given a bitmap as input and divides it up into areas that appear to contain separate characters. The output from this is fed into a second module that recognizes shapes and determines the probability that each area corresponds to a particular character. The output from that is fed into a third module that uses the probabilities to determine the sequence of words embedded in the input. If all these modules were grouped together, then the result would have sequential cohesion.

### Procedural cohesion

This is achieved when you keep together several procedures that are used one after another, even though one does not necessarily provide input to the next. It is therefore weaker than sequential cohesion. In Figure 9.4, the module marked ‘P’ is procedurally cohesive.

For example, in a university registration system, there would be a module to perform all the steps required to register a student in a course. The facilities for doing separate activities, such as adding a new course, would be in other modules.

### Temporal cohesion

This is achieved when operations that are performed during the same phase of the execution of the program are kept together, and everything else is kept out. This is weaker than procedural cohesion and is illustrated in Figure 9.4 by the modules marked ‘T’.

For example, a designer would achieve temporal cohesion by placing together the code used during system start-up or initialization, so long as this did not violate one of the other forms of cohesion listed above. Similarly, all the code for system termination, or for certain occasionally used features, could be kept together to achieve temporal cohesion.

There may be a temporally cohesive module in a layer whose job is to initialize the services of that layer. The module would be called at startup time, and not at any other time.

Although it would be temporally cohesive, it would be a violation of communicational cohesion to create a module that directly initializes the static variables of several different classes or the services of different layers. However, it would be permissible to have a temporally cohesive module that calls the initialization procedures of other modules.

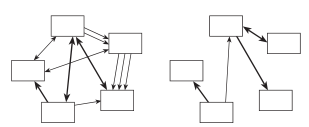
### Utility cohesion

This is achieved when related utilities that cannot be logically placed in other cohesive units are kept together. A utility is a procedure or class that has wide applicability to many different subsystems and is designed to be reusable. A utility module is marked ‘U’ in Figure 9.4.

For example, the *java.lang.Math* class has utility cohesion. Where possible, it would be better to put mathematical functions in classes on whose instances they are applied; however, *java.lang.Math* allows the grouping together of functions that have no obvious single home.

## Low Coupling

Coupling occurs when there are interdependencies between one module and another. Figure 9.5 illustrates the concept of a tightly coupled and loosely coupled system.



**FIGURE** 6-1 Abstract examples of a tightly coupled system (left) and a loosely coupled system (right). The boldness of the arrows indicates the strength of the coupling

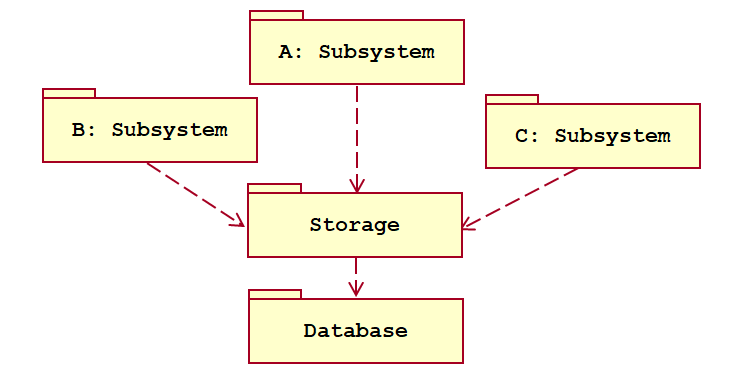
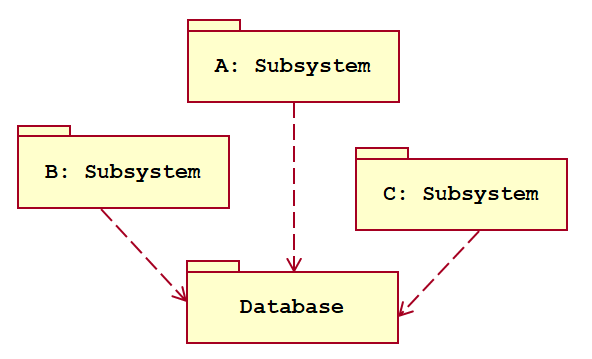
In general, the more tightly coupled a set of modules is, the harder it is to understand and, hence, change the system. Two reasons for this are:

■ When interdependencies exist, changes in one place will require changes somewhere else. Requiring changes to be made in more than one place is problematic since it is time-consuming to find the different places that need changing, and it is likely that errors will be made.

■ A network of interdependencies makes it hard to see at a glance how some component works.

Additionally, coupling implies that if you want to reuse one module, you will also have to import those with which it is coupled. This is because the coupled components need each other in order to work properly.

How to achieve low coupling? For example, in a software system, three subsystems A, B and C all need to access a relational database. In the following design scheme, the three subsystems directly access the database subsystem. A storage subsystem is added between the subsystem and the database to shield the impact of the change of the underlying database on the upper subsystem.



Below, we list some of the many different ways by which modules can be coupled, and some of the ways to reduce the coupling. The types of coupling are summarized in Table 9.2.

To reduce coupling, you have to reduce the number of connections between modules and the strength of the connections. Some of the types of coupling listed in Table 9.2 are particularly strong and should always be avoided.

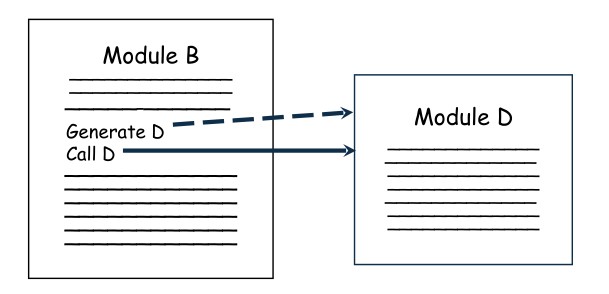
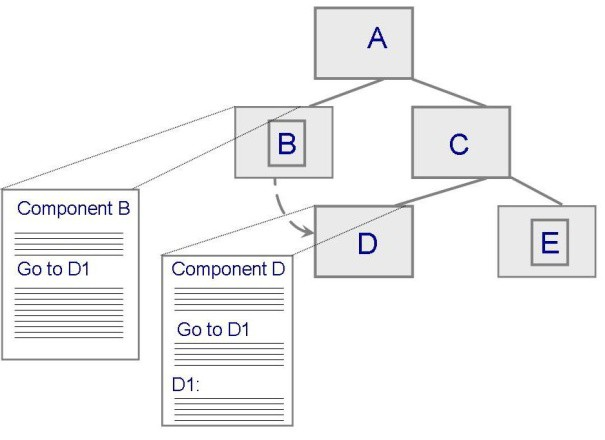
Table. Different types of coupling. You should reduce coupling where possible, but the types at the top are the strongest and hence the most important to avoid.

|  |  |
| --- | --- |
| **Coupling type** | **Comments** |
| **Content** | A component surreptitiously modifying internal data of another component. Always avoid this. |
| **Common** | The use of global variables. Severely restrict this. |
| **Control** | One procedure directly controlling another using a flag. Reduce this using polymorphism. |
| **Stamp** | One of the argument types of a method is one of your application classes. If it simplifies the system, replace each such argument with a simpler argument (an interface, a superclass or a few simple data items). |
| **Data** | The use of method arguments that are simple data. If possible, reduce the number of arguments. |
| **Routine call** | A routine calling another. Reduce the total number of separate calls by encapsulating repeated sequences. |
| **Type use** | The use of a globally defined data type. Use simpler types where possible (superclasses or interfaces). |
| **Inclusion/import** | Including a file or importing a package. Eliminate when not necessary. |
| **External** | A dependency exists to elements outside the scope of the system, such as the operating system, shared libraries or the hardware. Reduce the total number of places that have dependencies on such external elements. |

### Content coupling

This occurs when one component surreptitiously modifies data that is internal to another component. Content coupling should always be avoided since any modification of data should be easy to find and easy to understand. Usually content coupling has following situation:

* One module modifies the internal data items of another module
* One module modifies the code of another module
* Branches within one module are transferred to another module



Java is designed so that the worst kinds of content coupling (e.g. those involving manipulation of pointers) cannot be easily achieved. However, there are still some unfortunate tricks that Java programmers can play.

A form of content coupling occurs whenever you modify a public instance variable in a way that designers did not intend. To reduce content coupling you should therefore encapsulate all instance variables by declaring them private, and providing get and set methods. If you do this, you then have confidence that the only places where the variable is accessed and modified are in these methods. The set methods can ensure that only valid changes are made to the variables.

A worse form of content coupling, which is much harder to detect, occurs when you directly modify an instance variable of an instance variable. For example, in the following code, class Arch has a method called slant; this surreptitiously modifies the y value of the Point at the end of its baseline instance variable.

*public class Line{*

*private Point start, end;*

*...*

*public Point getStart() {return start;}*

*public Point getEnd() {return end;}*

*}*

*public class Arch{*

*private Line baseline;*

*void slant(int newY){*

*Point theEnd = baseline.getEnd();*

*theEnd.setLocation(theEnd.getX(),newY);*

*}*

*}*

The content coupling occurs here even though the instance variables are private, and baseline, an instance of *Line*, is supposedly immutable (Line has no *setStart* or *setEnd* methods). It is surreptitious because the *Line* is changed without “knowing” it is changing.

Part of the problem is that this code does not adhere to the delegation pattern (and the law of Demeter): the slant method is not accessing a neighboring object (the *Line*) but a more distant object (the *Point*).

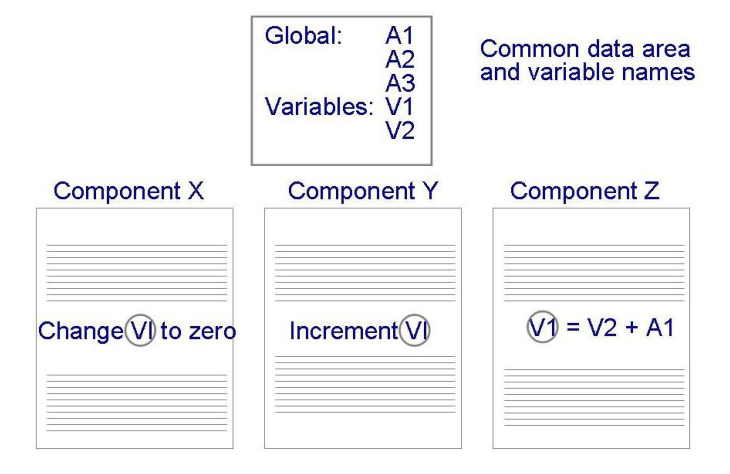
Two things must be done to combat this form of content coupling:

1. Make moving the end of a Line explicit, by adding a *moveEnd* method to it. The slant method should call this. However, this is not enough since programmers could still bypass the *moveEnd* method.

2. Make the Line class truly immutable. To do this it is necessary to use immutable classes for its instance variables. If you do this, then you eliminate the possibility of surreptitious modification.

### Common coupling

This occurs whenever you use a global variable–all the modules using the global variable become coupled to each other, and to the module that declares the variable. The coupling occurs because changes to the variable’s declaration will affect all the code that uses the variable. Also, changes to the way one module uses a variable will often have an effect on how the other modules should interpret the variable.



The word “global”, as used here, can mean that the variable is visible to all procedures and objects in the system. However, a weaker form of common coupling occurs any time a variable can be accessed by all instances of a subset of the system’s classes (e.g. a Java package).

In older programming languages, the use of global variables was widespread; the name ‘common’ comes from the FORTRAN language in which it is the keyword used to declare global data. In Java, public static variables serve as global variables.

The use of common coupling should be minimized, since it shares many of the disadvantages of content coupling. Occasionally, a case can be made to create global variables that represent system-wide default values – the argument for this is that it would be more complex to force a large number of routines to pass around such information as their parameters. However, most of these system-wide values are actually constants (i.e. declared final), and not variables.

For example, the *java.lang.Math* package has the constants *PI* and *E*. As is the case with content coupling, common coupling can be reduced by encapsulation. For each global variable, create a module that has specially designated public methods that can be called to get or set the data. The internal representation of the data can then be more easily changed and it can be protected from inappropriate changes made by ‘rogue’ code; also, the set method can verify that changes are valid.

Encapsulation reduces the harm of global variables, but there is still some undesirable coupling, therefore avoid having too many such encapsulated variables. Note that the Singleton pattern, provides encapsulated global access to an object; therefore avoid having too many singletons.

### Control coupling

This occurs when one procedure calls another using a ‘flag’ or ‘command’ that explicitly controls what the second procedure does. The following is an example:

*public routineX(String command){*

*if (command.equals("drawCircle"){*

*drawCircle();*

*}else{*

*drawRectangle();*

*}*

*}*

The method *routineX* will have to change whenever any of its callers adds a new command. It should also probably be changed if any of its callers deletes a command, otherwise it will have code that is said to be ‘dead’.

Control coupling can often be reduced by simply having the callers of routine directly call methods such as drawCircle or *drawRectangle*. But the use of polymorphic operations is normally the best way to reduce control coupling. In the example above, there could be two separate classes *Circle* and *Rectangle*; routine could then just call *draw*, with the system choosing the appropriate method to run.

There are cases when control coupling cannot or should not be completely avoided. For example, the *SimpleChat* server has the method *handleMessageFromClient*. This is tightly coupled to the methods in the

*SimpleChat* client generates the commands. One way to reduce the control coupling in this case would be to have a look-up table that mapped a command to a method that should be called when that command is issued.

There is still some coupling, since the look-up table must be modified when commands are changed; however, look-up tables are simpler in structure than nested if-then-else statements.

### Stamp coupling

This occurs whenever one of your application classes is declared as the type of a method argument. Some stamp coupling is necessary; however, the following situation illustrates why it is best to try to reduce it.

Imagine a class Employee that has many instance variables such as name, address, email, salary, manager, etc., and many methods to manipulate these variables. Any method that is passed an instance of Employee is given the ability to call any of its public methods. The method sendEmail in the following Emailer class, for example, has this ability.

*public class Emailer{*

*public void sendEmail(Employee e, String text) {...}*

*...*

*}*

The problem here is that the sendEmail method does not need to be given access to the full Employee object; it really only needs access to email and name. Giving it full access represents unnecessary stamp coupling. Any time a maintainer changes the Employee class he or she will have to check the sendEmail method to see if it needs to be changed. The Emailer class is also not reusable – it can only be used in applications that use the Employee class.

There are two ways to reduce stamp coupling, a) using an interface as the argument type, and b) passing simple variables. The following illustrates the first way:

*public interface Addressee{*

*public abstract String getName();*

*public abstract String getEmail();*

*}*

*public class Employee implements Addressee {...}*

*public class Emailer{*

*public void sendEmail(Addressee e, String text) {...}*

*...*

*}*

The stamp coupling is reduced since the sendEmail method now has access only to the name and email data that it truly needs. Changes to the Employee class will be far less likely to impact it. The sendEmail method will still be impacted if the Addressee interface is changed, although that is probably unlikely to occur. Given that the Addressee interface is easy to reuse, the Emailer class now becomes reusable.

Instead of creating a new Addressee interface, you might have considered using a superclass of Employee (e.g. Person) as the type of the sendEmail method. This can sometimes effectively reduce the stamp coupling; but using an interface is usually a more flexible solution.

The second way to reduce stamp coupling is illustrated as follows:

*public class Emailer{*

*public void sendEmail(Employee e, String text)*

*{...}*

*...*

*}*

In this case the stamp coupling has been replaced with data coupling, discussed below.

### Data coupling

This occurs whenever the types of method arguments are either primitive or else simple class such as *String*. Methods must obviously have arguments; therefore some data coupling or stamp coupling is unavoidable. However, you should reduce coupling by not giving methods unnecessary arguments.

*public class Emailer{*

*public void sendEmail(String name, String email, String text)*

*{...}*

*...*

*}*

The more arguments a method have, the higher the coupling. This is because each caller to the method must have code to prepare the data for each argument; and any changes to how the method declares or interprets each argument may require changes to each caller’s code.

There is a trade-off between data coupling and stamp coupling. In the case of a single argument, data coupling is considered looser and therefore better, than stamp coupling.

However, if you replace a single complex argument (stamp coupling) with many simple arguments (data coupling), the total resulting coupling will be higher. In the above code, it was acceptable to eliminate stamp coupling at the expense of adding one extra argument to the *sendEmail* method.

It would not have been acceptable to add three or four extra arguments; in such a case, sticking with the stamp coupling (using the *Addressee* interface) would have been better.

### Routine call coupling

This occurs when one routine (or method in an object-oriented system) calls another. The routines are coupled because they depend on each other’s behavior, and the caller depends on the interface of the called routine.

Routine call coupling is always present in any system. However, if you use a sequence of two or more methods to compute something, and this sequence is used in more than one place, then you can reduce routine call coupling by writing a single routine that encapsulates the sequence.

For example, imagine that to use a graphics package; you had to write the following sequence of code over and over again:

*aShape.drawBackground();*

*aShape.drawForeground();*

*aShape.drawBorder();*

You would be better off creating a new method that encapsulated this sequence. Should the arguments of the above three methods ever change; the maintainer would now only have to change your encapsulated method.

### Type use coupling

This occurs when a module uses a data type defined in another module. Type use coupling naturally occurs in typed languages such as Java. It occurs any time a class declares an instance variable or a local variable as having another class for its type.

Type use coupling is similar to common coupling, but instead of data being shared, only data types are shared. The impact of sharing data types is normally less than the impact of sharing data, hence type use coupling is considered less problematic than common coupling.

The consequence of type use coupling is that if the type definition changes, then the users of the type may well have to change.

Stamp coupling is closely related to type use coupling, therefore the techniques for reducing stamp coupling can also be applied to type use coupling.

In particular, you should declare the type of a variable to be the most general possible class or interface that contains the required operations. For example, when creating a variable that is to contain a collection, you should normally declare its type to be List, that is, any class that implements the *java.util.List* interface. The actual instance stored in the variable could be an *ArrayList*, *LinkedList* or *Vector*, or perhaps some other class to be defined later. However, declaring the type to be List is sufficient since all the important operations are defined in that interface. The benefit is that your code would be less likely to need to change were you to later decide to use a different type of collection.

### Inclusion or import coupling

Import coupling occurs when one component imports a package (as in Java); inclusion coupling occurs when one component includes another (as in C++).

Doing this means that the including or importing component is now exposed to everything in the included or imported component–even if it is not actually using the facilities of that component. If the included or imported component changes something on which the *includer* relies, or adds something that raises a conflict with something in the *includer*, then the *includer* must change.

The bigger the imported or included component, the worse the coupling. However, importing a standard package (e.g. one delivered with the programming language) is better than importing a homemade package.

Some inclusion or import coupling is necessary–since it enables you to use the facilities of libraries or other subsystems. However, it is important not to import packages or classes that you do not need: in addition to having to worry about changes to the things you are using, you then also have to worry about changes to things you don’t use. For example, your system might suddenly fail if a new item is added to an imported file, and this new item has the same name as something you have already defined in your subsystem.

### External coupling

This occurs when a module has a dependency on such things as the operating system, shared libraries or the hardware. It is best to reduce the number of places in the code where such dependencies exist.

The Façade design pattern can reduce external coupling by providing a very small interface to external facilities.

## High Abstraction

You should ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity. The general term given to this property of designs is abstraction. Abstractions are needed because the human brain can process only a limited amount of information at any one time.

We have discussed many types of abstractions in earlier chapters. In Chapter 2 we introduced procedural abstraction and data abstraction – hiding the details of procedures and data, respectively. In Section 2.7 we discussed several types of abstraction present in object-oriented programs.

Some abstractions, like classes and methods, are supported directly by the programming language. Others, like associations, are present purely in models used by the designer.

Abstractions work by allowing you to understand the essence of something and make important decisions without knowing unnecessary details. The details can be provided in several ways:

■ At a later stage of design. For example, when creating class diagrams, you often initially leave out the data types of attributes, and you do not show the implementation details of associations.

■ By the compiler or run-time system. For example, dynamic binding takes care of which methods will run.

■ By the use of default values. For example, a draw operation that always makes the background white unless some explicit action is taken to change the default.

## Increase Reusability

There are two complementary principles that relate to reuse; the first is to design for reuse, and the second is to design with reuse.

Designing for reusability means designing various aspects of your system so that they can be used again in other contexts, both in your system and in other systems. As discussed in Chapter 3, you can build reusability into algorithms, classes, procedures, frameworks and complete applications. Mechanisms whereby components can be reused include calling procedures and inheriting a superclass.

Important strategies for increasing reusability are as follows:

■ **Generalize your design as much as possible**. As you design a potentially reusable component, imagine several other systems that could use this component. Then design your component so that it could work with the other systems too. For example, if you are creating a facility to draw a particular kind of diagram, why not design it so that it could be used to draw other kinds of diagrams for other applications? Better yet, forget the specific application and focus on the reusable component alone. For example, if you need to create a method to save instances of *Employee* to a binary file, instead consider the problem of saving instances of any class to a binary file.

■ **Follow the preceding three design principles**. Increasing cohesion increases reusability since the component has a well-defined purpose. Reducing coupling increases reusability because the component can stand alone. Increasing abstraction increases reusability since abstractions are naturally more general.

■ **Design your system to contain hooks.** A hook is an aspect of the design deliberately added to allow other designers to add additional functionality. One of the barriers to reuse occurs when a component does most of what someone else needs, but not quite everything. If a component has effective hooks, then other people can easily extend it to do what they want. For example, the OCSF system has hooks such as *connectionClosed* that allow application designers to choose to do something interesting when a connection is closed.

■ **Simplify your design as much as possible.** The more complex the component, the less it is likely to be reusable in novel contexts. The most reusable components are those that do one simple thing but do it very well. Basic Unix commands such as grep, cat, head, tail, sort, uniq, awk, and sed are considered classic examples of reusable components because they are very powerful yet relatively simple. Their simplicity comes from the fact that they all input and output the same data type: streams of characters. Their power comes from the fact that they can be strung together in a large variety of combinations.

## Reuse existing designs and code

Designing with reuse is complementary to designing for reusability. Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components.

Cloning should normally not be seen as an effective form of reuse. Cloning involves copying code from one place to another; it should be avoided since, when there are two or more occurrences of the same or similar code in the system, any changes made (e.g. to fix defects) will have to be made in all clones.

Unfortunately, maintainers are often not aware of all the clones that exist, and hence only make the change in one place. The bug thus remains, even though the maintainer thinks it is fixed.

In general, it can be acceptable to clone a single line of code; perhaps a line that contains a complicated call to a method with many arguments. However, any time you are tempted to clone more than a couple of lines of code, it is normally best to encapsulate the code in a separate method and call it from all the places it is needed.

## Design for flexibility

Designing for flexibility (also known as adaptability) means actively anticipating changes that a design may have to undergo in the future and preparing for them.

Such changes might include changes in implementation (e.g. to improve efficiency or to handle larger volumes of data) or changes in functional requirements.

Ways to build flexibility into a design include:

■ Reducing coupling and increasing cohesion. This allows you to more readily replace part of a system. For example, if your current application saves data to a file, you might anticipate that in future you will want to use a commercial database package.

Placing the data-saving parts of your system in a subsystem that has layer cohesion will greatly facilitate such a change.

■ Creating abstractions. In particular, try to create interfaces or superclasses with polymorphic operations. Doing this allows new extensions to be easily added.

■ Not hard-coding anything. Constants should be banished from code. For example, if you want to limit the number of clients that can connect to a server to 25, do not have a line of code that says: if(numConnections <= 25)… . Instead, read the maximum value from a configuration file when the server starts.

Better yet, make such values preferences that users can change through a preferences dialog.

■ Leaving all options open. For example, when a method encounters an exception, it is best that the method throws the exception rather than taking a definite action to handle it. The caller of the method then has the flexibility to decide what to do with the exception.

■ Using reusable code and making code reusable. The techniques discussed in the previous two design principles, such as adding hooks, tend also to make designs more flexible.

## Anticipate obsolescence

Anticipation of obsolescence is a special case of design for flexibility. Changes will inevitably occur in the technology a software system uses and in the environment in which it runs. Anticipating obsolescence means planning for evolution of the technology or environment so that the software will continue to run or can be easily changed.

The following are some rules that designers can use to better anticipate obsolescence:

■ Avoid using early releases of technology. The immediate problem is that early releases are likely to have more defects than later releases. However, even if it is possible to work around a defect, a secondary problem may then arise: if the provider of the technology fixes the defect in a subsequent release of the technology, the original work-around may no longer work. Even where no defect exists, improvements to the technology, which are especially likely in the first few releases, can render designs that use the technology in need of change.

For example, early adopters of Java, in the 1995–1997 time frame, were later required to make many changes to their code because some of the classes and methods they used became deprecated. The Java designers declared components to be deprecated when they developed improved designs – they do not intend to support the older components indefinitely, therefore users are forced to update their software.

■ Avoid using software libraries that are specific to particular environments. For example, software that makes use of specific features found only in one operating system, or one particular type of hardware, is less likely to be supported in the distant future.

■ Avoid using undocumented features or little-used features of software libraries. The little-used features are not only more likely to have defects but, more importantly, the manufacturers may feel that little harm will be done if they are removed or changed. On the other hand, if the technology provider makes changes to heavily used features there will be loud protests from many users; therefore such changes are less likely to be made.

■ Avoid using reusable software or special hardware from smaller companies, or from those that are less likely to provide long-term support. A smaller company is more likely to go out of business or not to have the resources to support older versions. It may seem harsh to suggest that we should only trust larger companies – they can go out of business too. However, the probability is higher that a small company will have to abandon a product.

■ Use standard languages and technologies that are supported by multiple vendors. Doing this gives you some confidence that important technology will not be orphaned. Many software systems still in existence today are based on obscure proprietary languages. However, standards are not a panacea: they can change, and there may be subtle differences in implementations of a standard that make it difficult to switch to a competing vendor, even if the new vendor ostensibly supports the same standard.

## Design for portability

Designing for portability shares many things in common with anticipating obsolescence, although the objective is different. Anticipating obsolescence has, as its primary objective, the survival of the software. Design for portability has, as its prime objective, the ability to have the software run on as many platforms as possible, although sometimes this might also be a necessity for survival.

An important guideline for achieving portability is to avoid the use of facilities that are specific to one particular environment. Some programming languages, such as Java, make this easy because the language itself is designed to allow software to run on different platforms unchanged. Nevertheless, even with Java, there can be subtle differences regarding how some features work on different platforms – knowing about these and avoiding them is important. One such difference is class libraries; some companies have produced special Java libraries that work only with that company’s compiler, which in turn runs on only one platform. Attempting to port software that uses that library to another platform can be difficult.

Other languages such as C++ have many features that are very much dependent on the particular hardware architecture. You have to be aware, for example, of the order of characters within a word (so-called big-endian versus little-endian), and the number of bits in an integer.

Another important portability issue has to do with text files: the characters used to terminate lines differ from platform to platform.

## Design for testability

During design you can take steps to make testing easier. Testingcan be performed both manually and automatically.

Automatic testing involves writing a program that will provide various inputs to the system in order to test it thoroughly. Therefore it pays to design a system so that automatic testing is made easy.

The most important way to design for testability is to ensure that all the functionality of the code can be executed without going through the graphical user interface. You can achieve this by carefully separating the UI from the functional layer of the system. A test harness can then be written that calls the API of the functional layer. Another good strategy is to provide a command-line version of your system, such as the command-line version of *SimpleChat* that we presented at the beginning of this book. This will allow you to write a test program that automatically issues commands to your application.

In order to design a Java class for testability you can create a *main* method in each class. Such *main* methods simply exercise the other methods of a class and report any problems.

## Design defensively

You should never trust how others will try to use a component you are designing. Just like automobile drivers are taught not to trust other drivers, and therefore to drive defensively, a software designer should not trust other designers or programmers, and so should design defensively. In other words, in order to increase the reliability of your system, you not only need to make sure you don’t add any defects yourself, but you must also properly handle all cases where other code attempts to use your component inappropriately.

The most important way to design defensively is to check that all of the inputs to your component are valid. Or, more accurately, check the preconditions of each component.

For example, imagine you have a method that determines whether a certain date is a working day. The first thing this method would do is check that the date is valid.

Unfortunately, over-zealous defensive design can result in unnecessarily performing the same validity checks over and over again. For example, imagine the following method:

*public boolean isWorkingDay(String aDate)throws InvalidDateException{*

*if(!isValidDate(aDate)) throw new InvalidDateException();*

*return !(isWeekEnd(aDate) || isHoliday(aDate));*

*}*

The first line of the method body validates the date. However, due to defensive design, *isWeekEnd* and *isHoliday* may also validate the date. It is a waste of computing power to check the date up to three times like this.

Design by contract is a technique that allows you to design defensively in an efficient and systematic way. The key idea behind design by contract is that each method has an explicit contract with its callers. The contract has a set of assertions that state:

■ What preconditions the called method requires to be true when it starts executing. The caller has the responsibility to make these preconditions true before making the call.

■ What postconditions the called method agrees to ensure are true when it finishes executing. The called method has the responsibility to make these postconditions true, before returning.

■ What invariants the called method agrees will not change as it executes. Preconditions, postconditions and invariants are all Boolean expressions. If they ever evaluate to false, this indicates that there is a failure. They are similar to the OCL expressions we discussed in Chapter 5; in fact, OCL can be used to write assertions.

Performing assertion checking inside a program is one of the most effective ways to detect and correct errors. Many languages incorporate different mechanisms to write assertions. The ANSI macro assert(expression) can be used to this end in C++. In Java, the keyword *assert* has been introduced in version 1.4. For example, in the code below, an explicit precondition assertion has been added, meaning that the method must always be called with a valid date. Not fulfilling this condition would be an error.

*public boolean isWeekEnd(String aDate){*

*assert isValidDate(aDate); // precondition.*

*return (dayOfTheWeek(aDate)==SUNDAY || dayOfTheWeek(aDate)==SATURDAY);*

*}*

Under normal operation, the assertions should not be explicitly evaluated in each method since they have always to be true. However, in the testing phase it is useful to have the assertions explicitly executed, so that you can quickly identify any methods that do not fulfill their contract. In Java, assertions are enabled at compile time by using the *–ea* switch of *javac*. In this mode, assert will throw an *AssertionError* if the expression evaluates to false. When assertion checking is disabled (default mode), the assert statements are simply ignored.

You should therefore always switch on assertion checking during development and testing, and turn it off when the system is finally released.

Note that design by contract implies a level of trust. It is like having a driving instructor with you, clarifying the rules of the road, and stopping you from having an accident while you are learning. However, if you make mistakes after getting your license (the assertion checking is off) you can still have an accident.

Therefore at the boundaries between major components, such as layers, you should always rigorously check inputs. At these boundaries, you would therefore not use an assertion mechanism that can be turned off.

# Managing System Design

In this section, we discuss issues related to managing the system design activities. As in analysis, the primary challenge in managing the system design 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 cohesive system understandable by a single person.

## Reviewing System Design

Like analysis, system design is an evolutionary and iterative activity. Unlike analysis, there is no external agent, such as the client, to review the successive iterations and ensure better quality. This quality improvement activity 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 reviewers, or to use developers from another project to act as a peer review. These review processes work only if the reviewers have an incentive to discover and report 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 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?
* Does each actor have an access policy?
* Is every access policy 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 contradictions. You should ask the following questions to determine if a system design is consistent:

* Are conflicting design goals prioritized?
* Does any design goal violate a nonfunctional 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 any new technologies or components included in the system? Was the appropriateness or robustness of these technologies or components evaluated? How?
* Have performance and reliability requirements been reviewed in the context of subsystem decomposition?
* Have concurrency issues (e.g., contention, deadlocks) 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) with similar names denote similar concepts?
* Are all entities described at the same level of detail?

In many projects, you will find that system design and implementation overlap quite a bit. For example, you may build prototypes of selected subsystems before the architecture is stable in order to evaluate new technologies. This leads to many partial reviews instead of an encompassing review followed by a client sign-off, as for 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 process.

## Assigning Responsibilities

Unlike analysis, system design is the realm of developers. The client and the end user fade into the background. Note, however, that many activities in system design trigger revisions to the analysis model. The client and the user are 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 up of architects who define the subsystem decomposition and selected developers who will implement the subsystem. It is critical that system design include people who are exposed to the consequences of system design decisions. The architecture team starts work as soon as the analysis model is stable and continues 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** takes the main role in 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 and 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, they focus on the consistency of the APIs.
* The **document** **editor**, **configuration** **manager**, and **reviewer** roles are the same as for 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.

## Communicating about System Design

Communication during system design should be less challenging than during analysis: the functionality of the system has 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 that 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.” Whereas it is usually desirable to delay certain design decisions, such as the internal data structures and algorithms used by each subsystem, 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 toward 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 analysis (see Section 5.5.3) can be applied during system design:

* *Identify and prioritize the design goals for the system and make them explicit* (see Section 6.4.2). If the developers concerned with system design have 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 in identifying recent changes.
* *Maintain an up-to-date glossary*. As in 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 when more information is needed before committing to the design decision. This approach, however, can prevent the confrontation of difficult design problems. Before tabling an issue, several possible alternatives should be explored and described, and the delay justified. This ensures that issues 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 7.5.4) 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.

## 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 system design affect subsystem decomposition as each of the different activities of system design is initiated. Second, revisions to the interfaces of the subsystems occur when evaluation prototypes are created to evaluate specific issues. Third, errors and oversights that are discovered late trigger changes to the subsystem interfaces and sometimes to the system decomposition itself.

The first set of iterations is best handled in brainstorming sessions (either face-to-face or electronic). 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 analysis is complete. Decomposing the system early allows 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, a bureaucratic formal change process should not be used 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 (ideally, it should be independent of vendors and technology), and most of these explorations aim at identifying whether a specific package is appropriate for the system. During this period, developers can also create a vertical prototype 1 for a critical use case to test the appropriateness of the decomposition. This enables control flow issues to be discovered and addressed early. Again, a formal change process is not necessary. A list of pending issues and their status can help developers quickly propagate the results of a technology investigation.

The third set of iterations remedies design problems discovered late in the process. Although developers would much rather avoid these iterations, as they tend to be costly and introduce many new bugs in the system, they should anticipate changes late in development. Anticipating late iterations includes documenting dependencies among subsystems, the design rationale for subsystem interfaces, and any workaround that is likely to fail 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.

We can achieve the progressive stabilization of subsystem decomposition by using the concept of a design window. 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 so that purchasing decisions for the hardware can be done in time for development. Internal data structures and algorithms, however, can be left open until after integration, allowing developers to revise them based on performance testing. Once the design window is closed, the issue must be 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 so fast that companies are not aware of which technologies they themselves provide.