UNIT - I
SOFTWARE AND SOFTWARE ENGINEERING

The nature of software:

Software differs in important ways from the types of artifacts produced by other types of engineers:

1. Software is largely intangible
2. The mass-production of duplicate pieces of software is trivial.
3. The software industry is labor intensive.
4. It is all too easy for an inadequately trained software developer to create a piece of software that is difficult to understand and modify.
5. Software is physically easy to modify; however, as a side effect of theirmodifications, new bugs appear.
6. Software does not *wear out with use* like other engineering artefacts, but instead its *design deteriorates* as it is changed repeatedly

Types of software and their differences

Custom software is developed to meet the specific needs of a particularcustomer and tends to be of little use to others.

Ex: web sites, air-traffic control systems and software for managing the specialized finances of large organizations.

Generic software, on the other hand, is designed to be sold on the openmarket, to perform functions that many people need, and to run on generalpurposecomputers. Generic software is often called *Commercial Off-The-Shelf* software (COTS), and it is sometimes also called *shrink-wrapped* software

Ex: word processors, spreadsheets, compilers, web browsers, operating systems, computer games and accounting packages for small businesses.

Embedded software runs specific hardware devices which are typically soldon the open market. users cannot usually replace embedded software or upgrade it without also replacing the hardware.

EX: washing machines, DVD players, microwave ovens and automobiles.
Another important way to categorize software in general is whether it is *real time* or*data processing* software.

- The most distinctive feature of real-time software is that it has to react immediately (i.e. in real time) to stimuli from the environment (e.g. the pushing of buttons by the user, or a signal from a sensor).
- Data processing software is used to run businesses. It performs functionssuch as recording sales, managing accounts, printing bills etc.

- Differences among custom, generic and embedded software

	Custom	Generic	Embedded
Number of *copies* in use	low	medium	high
Total *processing power* devoted to running this type of software	low	high	medium
Worldwide annual *development effort*	high	medium	low

What is software engineering?

Definition:

Software engineeringis the process of solving customers' problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints.

Software engineering as a branch of the engineering profession

People have talked about software engineering since 1968 when the term wascoined at a NATO conference.

Prior to the 1940s, very few jurisdictions required engineers to be licensed. Since engineering has become a licensed profession, adherence to *codes of ethics* and taking *personal responsibility* for work have also become essential characteristics. In the 1970s the discipline of *computer science* developed, and educated many of the current generation of software developers.

In the mid-1990s the first jurisdictions started to recognize software engineering as a distinct branch of engineering. For example, in the United Kingdom those who study software engineering in computer science departments at universities have been able to achieve the status of Chartered Engineer. Since considerable numbers of these graduates are now entering the workforce, software engineering has become firmly established as a branch of engineering.

Stakeholders in software engineering

We will classify these *stakeholders* into **four** major categories

1. **Users**. These are the people who will use the software.
2. **Customers** (also known as *clients*). These are the people who make decisions about ordering and paying for the software.
3. **Software developers**. These are the people who develop and maintain the software, many of whom may be called software engineers.
4. **Development managers**: These are the people who run the organization that is developing the software; they often have an educational background in business administration.

Software quality

Attributes of software quality

Five of the most important attributes of software quality (External)

1. **Usability**. The higher the usability of software, the easier it is for users to work with it.
2. **Efficiency**. The more efficient software is, the less it uses of CPU-time, memory, disk space, network bandwidth and other resources.
3. **Reliability**. Software is more reliable if it has fewer failures.
4. **Maintainability**. This is the ease with which you can change the software. The more difficult it is to make a change, the lower the maintainability.
5. **Reusability**. A software component is reusable if it can be used in several different systems with little or no modification.

Software engineers improve one quality at the expense of another.

Example:

Improving efficiency may make a design less easy to understand.

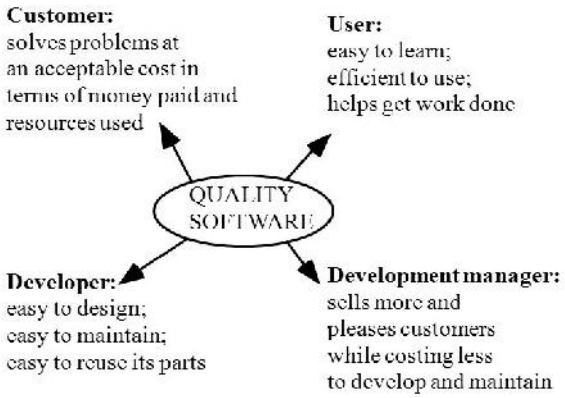
Achieving high reliability often entails repeatedly checking for errors and adding redundant computations;

Improving usability may require adding extra code, which might in turn reduce overall efficiency and maintainability.

There are also many *internal quality criteria.Ex:* The amount of commenting of the code.

The complexity of the code measured in terms of the nesting depth.

Software Quality and the Stakeholders



Quality for the short term vs. quality for the long term

It is human nature to worry more about short-term needs and ignore the longer-term consequences of decisions. This can have severe consequences. Examples of short-term quality concerns are: Does the software meet the customer's immediate needs? Is it sufficiently efficient for the volume of data we have today?

Software engineering projects

Software projects into **three** major categories:

1. Evolutionary projects(those that involve modifying an existing system)
   Evolutionary or maintenance projects can be of several different types:

Corrective projects involve fixing defects.
Adaptive projects involve changing the system in response to changes in the environment in which the software runs.
Enhancement projects involve adding new features for the users.
Re-engineering or *perfective* projects involve changing the system internally so that it is more maintainable

2. Greenfield projects(involve starting to develop a system from scratch) Developers often enjoy such brand new, or *Greenfield*, projects because they have a wider freedom to be creative about the design. In a Greenfield project you are not constrained by the design decisions and errors made by predecessors.

3. Projects building on a framework or a set of existing components
This type of project, which is becoming increasingly common, starts with a *framework*, or involves plugging together several *components* that are already developed and provide significant functionality. The code that you write to connect the two component packages is called *glue*.

Activities common to software projects

1. Requirements and specification you must first understand the problems, the customer's business environment, and the available technology which can be used to solve the problems. Overall process may include
   Domain analysis: understanding the background needed
   Defining the problem: precise problem that needs solving.
   Requirements gathering: obtaining all the ideas people have about what the software should do.
   Requirements analysis: organizing the information that has been gathered, and making decisions
   Requirements specification: writing a *precise* set of instructions that define what the software should do.

2. Design
Important activities during design include:
   Deciding what requirements should be implemented in hardware and what in software. This is called *systems engineering*
   Deciding how the software is to be divided into subsystems. This process is often called *software architecture* ;(*architectural patterns* or *styles*.)
   Deciding how to construct the details of each subsystem. This process is often called *detailed design*.
   Deciding in detail how the user is to interact with the system, and the *look and feel* of the system. This is called *user interface design*
   Deciding how the data will be stored on disk in databases or files.

3. Modeling
Modeling is the process of creating a representation of the domain or the software.
Ex: visual language called *UML*.These includes:

Use case modeling.
Structural modeling.
Dynamic and behavioral modeling

4. Programming

   Programming is an integral part of software engineering. It should be thought of as the final stage of design. People who limit their work to programming are often todaycalled 'coders'.

5. Quality assurance

   It includes the following:
   Reviews and inspections. These are formal meetings organized to discuss requirements, designs or code to see if they are satisfactory.
   Testing. This is the process of systematically executing the software to see if it behaves as expected.

6. Deployment

   Deployment involves distributing and installing the software and any other components of the system such as databases, special hardware etc.

7. Managing the process

   The manager has to undertake the following tasks:
   Estimating the cost of the system. This involves studying the requirements and determining how much effort they will take to design and implement.
   Planning. This is the process of allocating work to particular developers, andsetting a schedule with deadlines.

Difficulties and risks in software engineering as a whole

Complexity and large numbers of details. Software systems tend to becomecomplex.
Resolution. Design the system for flexibility right from the start. Divide the system into smaller subsystems

Uncertainty about technology. Technology on which a system depends will work as expected or not.
Resolution. Avoid technology sold by just a single vendor

Uncertainty about requirements. Whether it meets the customer's needs or not
Resolution. Understand the application domain so you can communicate effectively with clients and users

Uncertainty about software engineering skills. Software engineering is heavily labour-intensive;

Resolution. Make sure software engineers have sufficient general education, plus training in the technology to be used.

Constant change. Both technology and requirements can be expected tochange regularly.

Resolution. Design for flexibility to accommodate potential changes. Stay aware of things that may change.

Deterioration of software design. Software deteriorates due to successive changes that introduce bugs.

Resolution. Build flexibility and other aspects of maintainability into t he software from the start

'Political' risks. Not everybody will be happy with the requirements. Not everybody may want the system.

Resolution. Participate in promoting and marketing the project.

Review of Object Orientation

What is object orientation?

Object-oriented systems combine procedural abstraction with dataabstraction.Procedural abstraction and the procedural paradigm

From the earliest days of programming, software has been organized around the notion of *procedures* (also in some contexts called *functions* or *routines*). In the so-called procedural paradigm, the entire system is organized into a setof procedures. One 'main' procedure calls several other procedures, which inturn call others. Procedural paradigm is complex if eachprocedure works with many types of data, or if each type of data has many different procedures that access and modify it.

Data abstraction

Data abstractions can help reduce some of a system's complexity. *Records* and*structures* were the first data abstractions to be introduced. The idea is to grouptogether the pieces of data that describe some entity, so that programmers can manipulate that data as a unit.

Example:

```
if account is of type checking then
do something
else if account is of type savings then
do something else
else
do yet another thing
endif
```

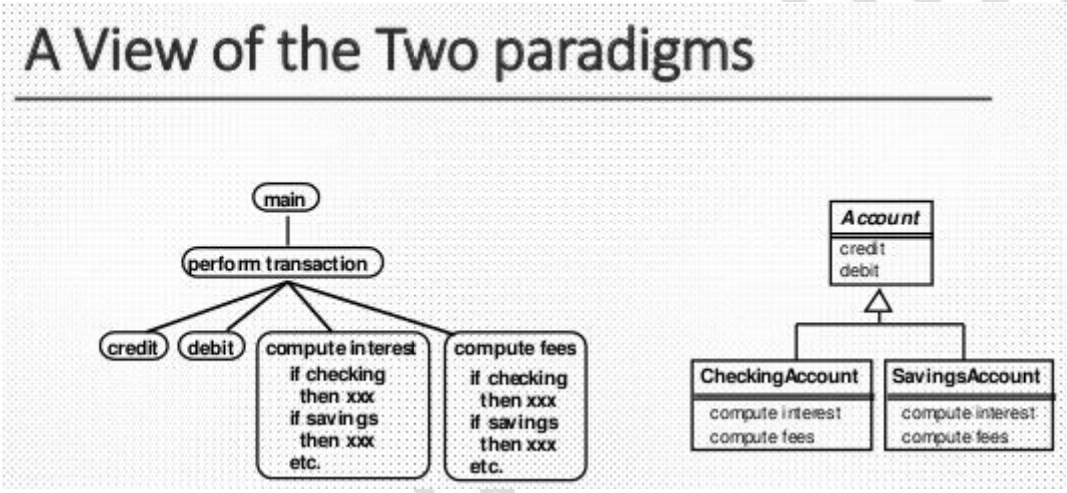
Imagine also that clients can hold several accounts of different types, and some accounts can be held jointly; also the different account holders might havedifferent rights. Rules to deal with issues like these would be scattered throughout the code, making change very difficult.

The object-oriented paradigm: organizing procedural abstractions in the context of data abstractions

Definition:

The *object-oriented paradigm* is an approach to the solution of problems in which allcomputations are performed in the context of objects. The objects are instances of programming constructs, normally called classes, which are data abstractions and which contain procedural abstractions that operate on the objects.The difference between the object-oriented and procedural paradigms.

Procedural Paradigm	**Object-oriented Paradigm**
1. Code is organized into procedures 2. Each manipulate different types of data	1.Code is organized into classes 2.Each contain procedures for manipulating instances of thatclass



Classes and objects

Objects

An object is a chunk of structured data in a running software system. It canrepresent anything with which you can associate *properties* and *behavior*.Properties characterize the object, describing its current *state*. Behavior is theway an object acts and reacts, possibly changing its state.

EX:
Margaret:
Date ofBirth="1984/03/03"
address="150 C++ Rd."
position="Teller"

Jane:
Date ofBirth="1955/02/02"
address="99 UML St."

position="Manager"

Mortgage account 29865:
balance=198760.00
opened="2003/08/12"
property="75 Object Dr."

Classes and their instances

Classes are the units of data abstraction in an object-oriented program. Ex:
Employee
name
dateofBirth
address
position

Instance variables

A variable is a place where you can put data. Each class declares a list of variables corresponding to data that will be present in each instance; such variables are called *instance variables*. There are **two** groups of instance variables,

Attributes

An *attribute* is a simple piece of data used to represent the properties of an object. For example, each instance of class Employee might have the followingattributes:

name
dateOfBirth
socialSecurityNumber
telephoneNumber
addres

Associations

An *association* represents the relationship between instances of one class andinstances of another. For example, class Employee in a business application mighthave the following relationships:

supervisor (association to class Manager)
tasksToDo (association to class Task)

Variables versus objects

Difference between *variables* and *objects*

At any given instant, a variable can refer to a particular object or to no object at all. Variables that refer to objects are therefore often called references.

Variables can be local variables in methods; these are created when a method runs and are destroyed when a method returns.

Instance variables versus class variables

If you declare that a class has an instance variable called var, then you are sayingthat*each instance* of the class will have its own slot named var. Therefore, forexample, each Employee has a supervisor.

Sometimes, however, you want to create a variable whose value is shared by all instances of a class. Such a variable is known as a *class variable* or *static variable.*

Methods, operations and polymorphism

- Methods: are procedural abstractions used to implement the behavior of a class.
- An *operation:* is a higher-level procedural abstraction. It is used to discuss and specify a type of behavior
- *Polymorphism:* is a property of object-oriented software by which an abstract operation may be performed in different ways, typically in different classes.

Concepts that define object orientation

To be called object oriented, a language needs to have the following features:

Identity: Every object has a unique identity;
Classes: Classes, which describes the structure and function of a set of objects.
Inheritance: Features inherit from super classes to subclasses.
Polymorphism:several methods can have the same name and implement the same abstract operation.
Abstraction: There are many abstractions in an object-oriented program:

An **object**is an abstraction of something of interest to the program,
A**class**is an abstraction of a set of objects;
A**superclass**is an abstraction of a set of subclasses:
A**method**is a procedural abstraction that hides its implementation
An**operation**is an abstraction of a set of methods.
Attributes and **associations** are abstractions of the underlying instance variables used to implement them.

Modularity

An object-oriented system can be constructed *entirely* from a setof classes, where each class takes care of a particular subset of the functionality

Encapsulation

A class acts as a container to hold its features (variables and methods). Abstraction, modularity and encapsulationeach help provide **informationhiding**.

Difficulties and risks in programming lang. choice and OO programming

Language evolution and deprecated features

Every programming language evolves, such that code written for earlier versions will not run or gives warning messages threatening that it will not run in the future.

Resolution

Pay careful attention to the documentation describing which featuresof Java are deprecated.

Efficiency can be a concern in some object-oriented systems

Most implementations of Java run using a virtual machine. This means that Java code tends not to be as efficient as code written in a language such as C++.Java's exception handling and safety checking also can consume considerable CPU time.

Resolution

Learn about the different programming strategies that make a Java program run faster. Consider languages other than Java for number-crunching applications.

UNIT - II
DEVELOPING REQUIREMENTS

Domain analysis

Domain analysis is the process by which a software engineer learns backgroundinformation. The word 'domain' in this case means the general field of business or technology in which the customers expect to be using the software.

Benefits:

Faster development. You will be able to communicate with the stakeholdersmore effectively

Better system. Knowing the subtleties of the domain will help ensure that thesolutions you adopt will more effectively solve the customer's problem.

Anticipation of extensions. Armed with domain knowledge, you will obtain insights into emerging trends and you will notice opportunities for future development.

A domain analysis document should be divided into sectionssuch as the following:

A. **Introduction**. Name the domain, and give the motivation for performing theanalysis.

B. **Glossary**. Describe the meanings of all terms used in the domain that are eithernot part of everyday language or else have special meanings.

C. **General knowledge about the domain**. Summarize important facts or rules. Such knowledge includes scientific principles, business processes, analysis techniques

D. **Customers and users**. Describe who will or might buy the software, and inwhat industrial sectors they operate.

E. **The environment**. Describe the equipment and systems used.

F. **Tasks and procedures currently performed**. Make a list of what the variouspeople do as they go about their work. It is important to understand both theprocedures people are supposed to follow as well as the shortcuts they tend totake.

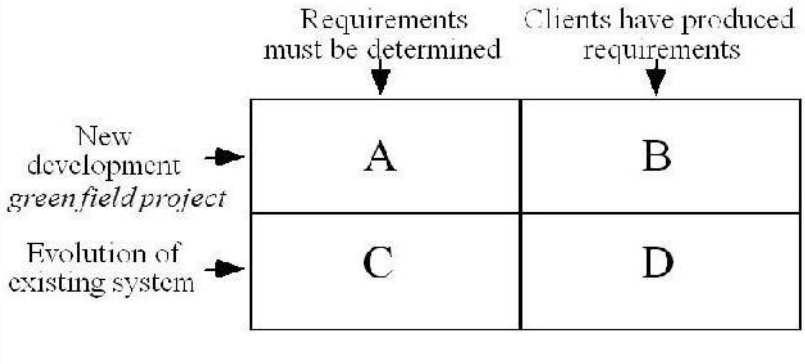
G. **Competing software**. Describe what software is available to assist the usersand customers. Discuss its advantages and disadvantages.

H. **Similarities across domains and organizations**. Determine what distinguishes this domain and the customer's organization from others, as well as what they have in common.

The starting point for software projects

When a development team starts work on a software project, their starting point can vary considerably. We can distinguish different types of project, based on whether or not

software exists at the outset, and whether or not requirements exist at the outset. The four broad categories of starting point are



In projects of type A or B, the development team starts to develop n wsoftware from scratch – this is sometimes called *green-field* development, alluding to co structing a new building where none existed before.

In cases C and D the team evolves an existing system, a rather more common situation. In cases A and C, the development team has to determine the requirements for the software – they either have a bright idea for something that might sell, or else they are asked to solve a problem and have to work out the best way to solve it.

In cases B and D, on the other hand, the development team is contracted todesign and implement a very specific set of requirements.

Defining the problem and the scope

Once you have learned enough about the domain, you can begin to determinethe requirements. The first step in this process is to work out an initial definition of the problem to be solved. Problem can be expressed as a difficulty the users or customers are facing, or as an opportunity that will result in some benefit such as improved productivity or sales.

You should write the problem as a simple statement. Careful attention to the problem statement is important since, later on, the requirements will be evaluated based on the question: 'are we adequately solving the problem? A good problem statement is short and succinct – one or two sentences are best.

What is a requirement?

Definition:

A requirement is a statement describing either 1) an aspect of what the proposed system must do, or 2) a constraint on the system's development. In either case, it must contribute in some way towards adequately solving the customer's problem; the set of requirements as a whole represents a negotiated agreement among all stakeholders.

Types of requirements

Requirements can be divided into **four** major types: functional, quality, platformandprocess. Requirements documents normally include at least the **first two types**.

Functional requirements

Functional requirements describe what the system should do; in other words, they describe the *services* provided for the users and for other systems. The functional requirements shouldinclude

1) Everything that a *user* of the system would need to know regarding what the system does, and

2) Everything that would concern any *other system* that has to interface to this system.

The functional requirements can be further categorized as follows:

Whatinputs the system should accept
Whatoutputs the system should produce
What data the system should store that other systems might use
Whatcomputations the system should perform.
Thetiming and synchronization of the above

Some techniques for gathering requirements

We list some structured techniques that are particularly effective at gathering (also known as *eliciting*) requirements.

1) Observation

You can read documents and discuss requirements extensively with users, butoften only the process of observing the users at work will bring to light subtledetails that you might otherwise miss. Observation means taking a notebook and 'shadowing'important potential users as they do their work, writing down everything they do. You can also ask users to talk as they work, explaining what they are doing.

2) Interviewing

Interviewing is a widely used technique. However, a well-conducted series ofinterviews can elicit much more information than poorly planned ad-hocinterviews. Firstly, plan to have as many members of the software engineering teaminterview as many stakeholders as possible. Spread out the interviews over time, and allow yourself several hours for eachinterview, even if you do not expect to use that much time. Prepare an extensive list of questions, although do not be disappointed if thereis not enough time to have them all answered in a given interview.

Ask about **specific details**
Ask about the stakeholder's **vision for the future**
Ask if they have any **alternative ideas**
Ask what would be a **minimally acceptable** solution to the problem.
Ask for **other sources of information**.
Have the interviewee draw **diagrams**.

3) Brainstorming

Brainstorming is an effective way to gather information from a group of people.The general idea is that the group sits around a table and discusses some topic with the goal of generating ideas. The following is a suggested;

1. Call a meeting with representation from all stakeholders. Effective brainstorming sessions can be run with five to 20 people.

2. Appoint an experienced moderator (also known as a facilitator) – that is,someone who knows how to run brainstorming meetings, and will lead theprocess. The moderator may participate in the discussions if he or she wishes.

3. Arrange the attendees around the periphery of a table and give them plenty ofpaper to work with.

4. Decide on a 'trigger question'. This is a key step in the process. A trigger question is one for which the participants can provide simple one-line answersthat are more than just numbers or yes/no responses.

5. Ask each participant to follow these instructions:

(a) Think of an answer to the trigger question, no matter how trivial or Questionable the answer is!

(b) Write the answer down in one or two lines on a sheet of paper, *one idea per sheet*.

(c) Pass the paper to the neighbor on your left (i.e. clockwise) to stimulate his or her thoughts.

(d) Look at the answers passed from your neighbor to the right and pass these on to your left as well. Use the ideas you have read to stimulate your ownideas.

6.Continue step 5 until ideas stop flowing or a fixed time (5–15 minutes) passes.

7.Moving around the table, ask everybody to read out one of the ideas on the sheets that happen to be in front of them. If anyone seeks an explanation, the originator of the idea may comment briefly (although he or she may choose not to say anything in order to remain anonymous). The moderator, or a secretary, writes each idea on a flip-chart. Then, optionally, the whole group may briefly discuss the idea.

8. After a fixed time period, or after all ideas have recorded on the flip-chart, the group may take a series of votes to prioritize them. For example, every person may be given a fixed number of votes that they can allocate to the answers they think are the most important.

4) Prototyping

A prototype is a program that is rapidly implemented and contains only a smallpart of the anticipated functionality of a complete system. Its purpose is to gather requirements by allowing software engineer to obtain early feedback about their ideas. The simplest kind of prototype is a *paper prototype* of the user interface. Thisis a set of pictures of the system that are shown to customers and users in sequence to explain what would happen when the system runs.

Managing changing requirements

One of the most important things to realize about requirements is that they change. Just because you have written a requirements document, and have obtained approval of it by all the stakeholders, does not mean that you can confidently design and implement the system as specified.

The following are some of the changes to anticipate:

Business process changes

Businesses regularly adjust the way they do things in order to better compete in the market or merely because they gain experience and decide that an alternative approach is better. Changes to business processes can also be prompted by such things as changes in laws, as well as growth or rearrangement of the company.

Technology changes

A new release of the operating system, or some other system with which your system interacts, may force you to reassess the requirements.

Better understanding of the problem

Even though everybody might be confident about the requirements when they are first approved, various stakeholders may discover problems when looking at them again several months later.When dealing with changes to requirements it is very important to avoid *requirements creep*. This is what occurs when the changes are really enhancements in disguise.

UNIT - III
MODELING WITH CLASSES

What is UML?

The Unified Modeling Language (UML) is a standard graphical language for modeling object-oriented software. It was developed in the mid-1990s as a collaborative effort by **James Rumbaugh, Grady Booch and Ivar Jacobson**, each of whom had developed their own notation in the early 1990s. The 'U' in UML stands for 'unified', since its three developers combined the best features of the languages they had each previously developed. The custodian of the UML standard is the Object Management Group (OMG). In 2004 the OMG approved version 2.0 of UML.

UML contains a **variety** of diagram types, including:

Class diagrams, which describe classes and their relationships.
Interaction diagrams, which show the behavior of systems in terms of how objects interact with each other. **Two types** of interaction diagrams: sequence diagrams and communication diagrams.
State diagrams and activity diagrams, which show how systems behave.
Component and deployment diagrams, which show how the various components of systems are arranged logically and physically.

Additional interesting features:

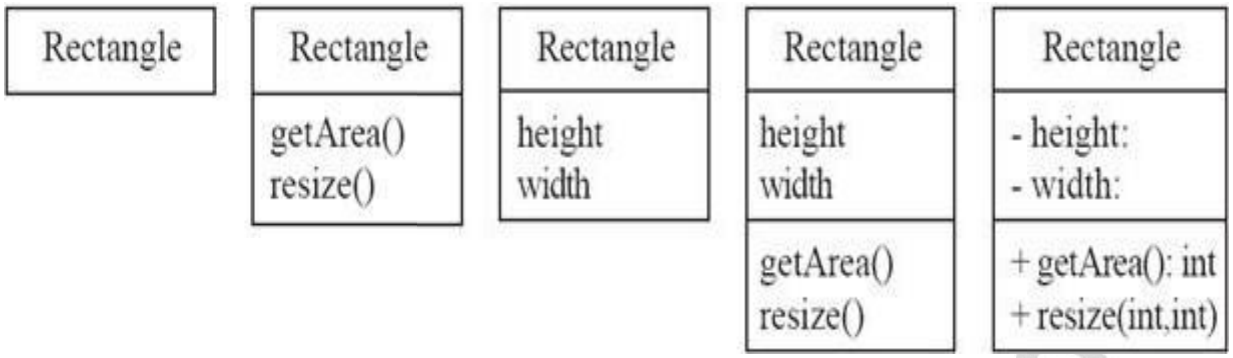
The diagrams you create with it are intended to be interconnected to form a unified *model*; we will discuss this more in the next subsection.
It has a detailed *semantics*, describing mathematically the meaning of many aspects of its notations.
It has *extension* mechanisms, which allow software designers to represent concepts that are not part of the core of UML. We will show some examples of these mechanisms.
It has an associated textual language called *Object Constraint Language* (OCL) that allows you to formally state various facts about the elements of the diagrams.

Essentials of UML class diagrams

The main symbols shown on class diagrams are:
Classes, which represent the types of data themselves.
Associations, which show how instances of classes reference instances of other classes.
Attributes, which are simple data found in instances.
Operations, which represent the functions performed by the instances.
Generalizations, which are used to arrange classes into inheritance hierarchies.

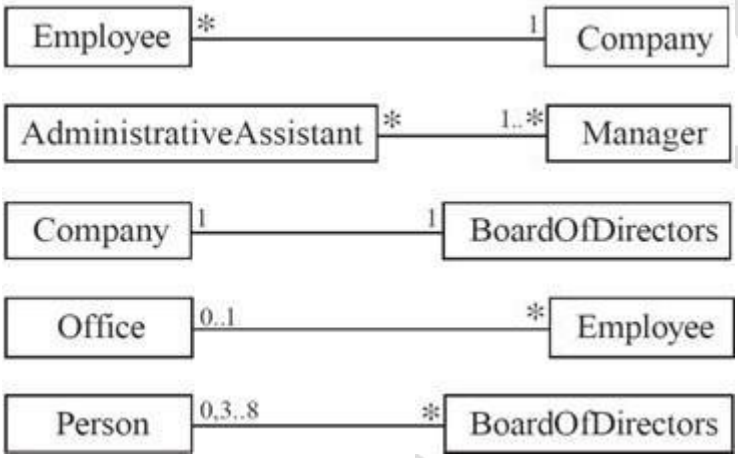
Classes

A class is represented as a box with the name of the class inside. The name should always be singular and start with a capital letter. The class diagram may also show the attributes and operations contained in each class. This is done by dividing a class box into two or three smaller boxes: the top box contains the class name, the next box lists attributes, and the bottom box lists operations.



Associations and multiplicity

An *association* is used to show how instances of two classes will reference each other. The association is drawn as a line between the classes. The multiplicity indicates how many instances of the class at this end of the association can be linked to an instance of the class at the other end of the association.



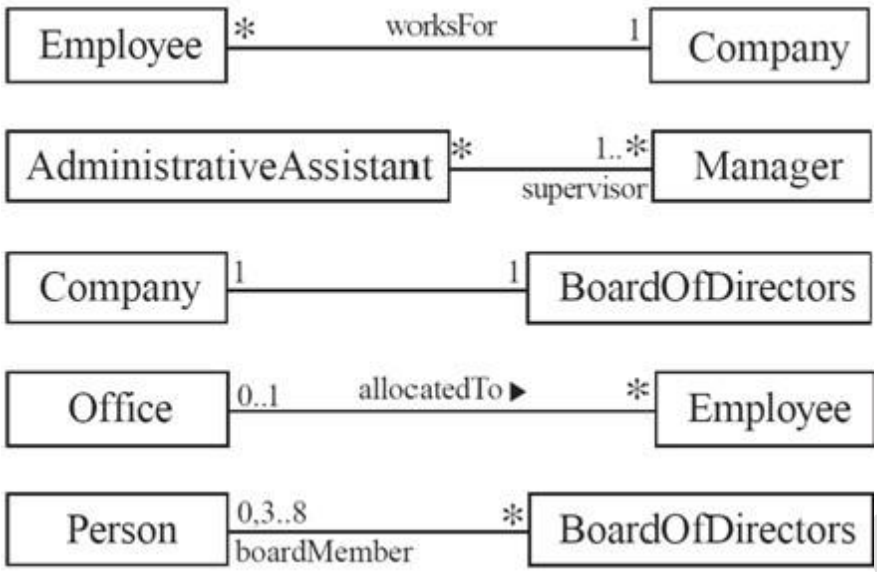
A multiplicity of 1 indicates that there must be exactly one instance linked to each object at the other end of the association A very common multiplicity is *, which is normally read as 'many', and means any integer greater than or equal to zero.

Labeling associations

Each association can be labeled, to make explicit the nature of the association. There are **two** types of labels, **association names and role names.**

An *association name* should be a verb or verb phrase, and is placed next to the middle of the association. One class becomes the subject and the other class becomes the object of the verb.

Another way of labeling an association is to use a *role name*. Role names can be attached to either or both ends of an association. A role name acts, in the context of the association, as an alternative name for the class to which it isattached.



Analyzing and validating associations
Three of the most common patterns of multiplicity

One-to-many. A company has many employees, but an employee can only work for one company.

Many-to-many. An administrative assistant can work for many managers, and a manager can have many administrative assistants.

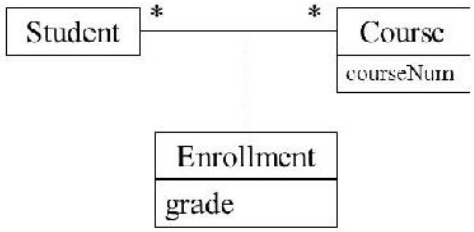
One-to-one. For each company, there is exactly one board of directors. Also, a board is the board of only one company.

Association classes

In some circumstances, an attribute that concerns two associated classes cannot be placed in either of the classes.

For example, imagine the association in which a student can register in any number of course sections, and a course section can have any number of students. In which class should the student's grade be put?

The solution to this problem is to create an *association class* to hold the grade.



Reflexive associations

It is possible for an association to connect a class to itself.

Links as instances of associations

In the same way that we say an object is an instance of a class, we say that a *link* is an instance of an association. Each link connects two objects – an instance of each of the two classes involved in the association.

Directionality in associations

Associations and links are by default *bi-directional*. That is, if a Driver object is linked to a Car object, then the Car is also implicitly linked to that Driver. If youknow the car, you can find out its driver – or if you know the driver, you can findout the car.

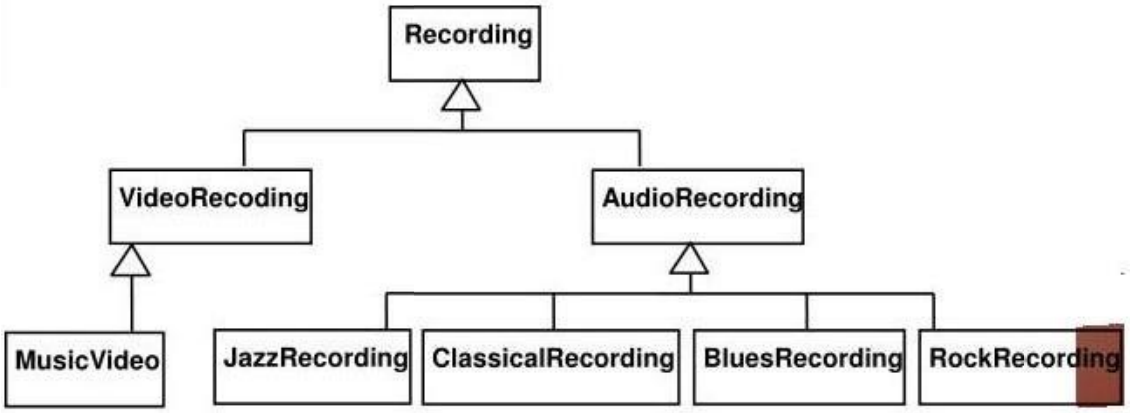
It is possible to limit the navigability of an association's links by adding an arrow at one end.

Generalization

They must follow the "is a" rule, and several other rules as well.

1) Avoiding unnecessary generalizations

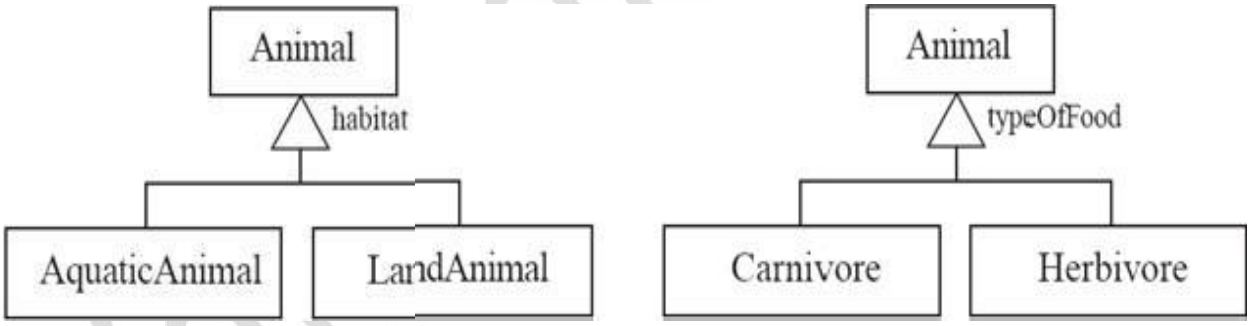
A common mistake made by beginners is to overdo generalization.



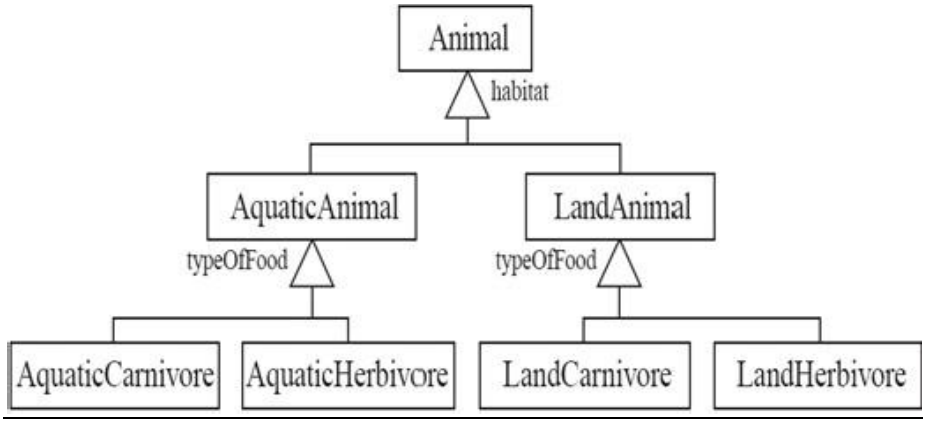
A hierarchy of classes in which there would not be any differences in operations. This should be avoided

2) Handling multiple generalization sets

A *generalization set* is a labeled group of generalizations with a common superclass; the label describes the criteria used to specialize the superclass into two or more subclasses. It is clearest to unite all the generalizations in a set using a *single* open triangle. You place the label next to the open triangle.



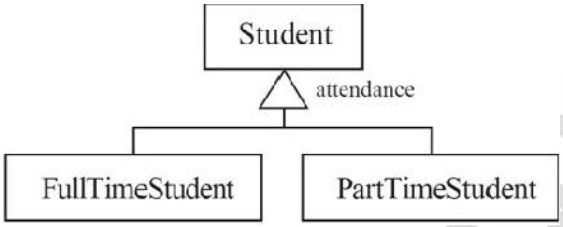
Another possible solution, using *multiple inheritances*.This approach uses even more classes and generalizations but avoids duplication of features.



Allowing different combinations of features by duplicating a generalization set label at a lower level of the hierarchy. Duplication like this should be avoided

3) Avoiding having objects change class

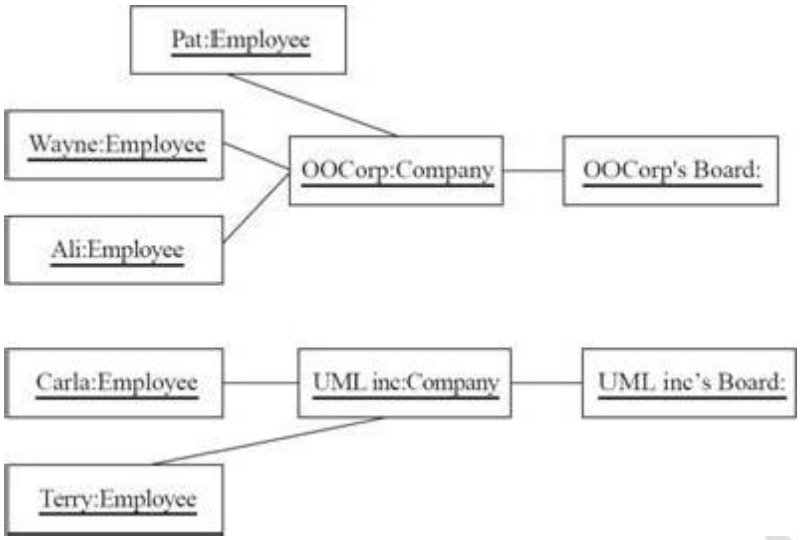
Another issue that can arise when creating generalizations is avoiding the need for objects to change class. In general, an object should never need to change class



A situation in which objects will need to change class from time to time. Generalizations of this type should be avoided

Instance diagrams

An *object diagram* shows an example configuration of objects and links that may exist at a particular point during execution of a program. Objects are shown as rectangles, just like classes; the difference is that the name of the class is underlined and preceded by a colon,:Employee, for example. You can also give a name to each instance before the colon, as in Pat:Employee, or even omit the class name entirely if it is clear from the context, such as Pat



Object diagrams generated from class diagrams

It is a common mistake for beginners to think of generalizations as special associations. This misconception arises because both generalizations and associations connect classes together in a class diagram.

However, the differences between the two concepts are profound.
An association describes a relationship that will exist between instances at runtime.
A generalization describes a relationship between classes in a class diagram.

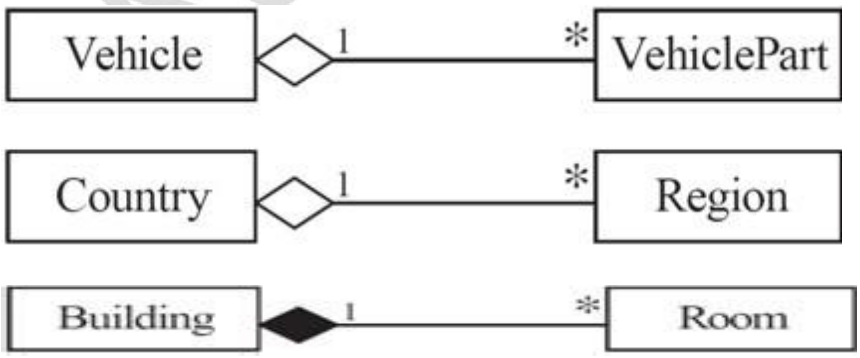
More advanced features of class diagrams

We describe additional features for adding more specific information to the diagrams.

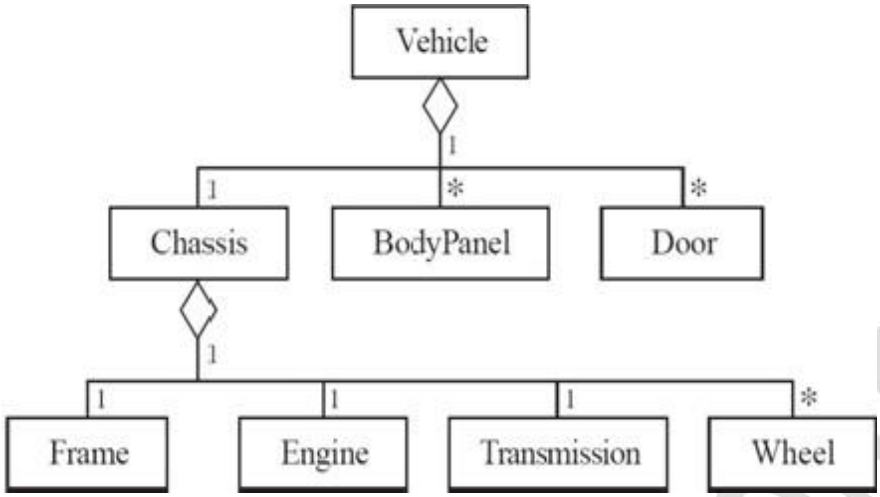
Aggregation

Aggregations are special associations that represent 'part–whole' relationships. The 'whole' side of the relationship is often called the assembly or the aggregate.

A composition is a strong kind of aggregation in which if the aggregate is destroyed, then the parts are destroyed as well. A composition is shown using a solid (filled-in) diamond, as opposed to an open one.

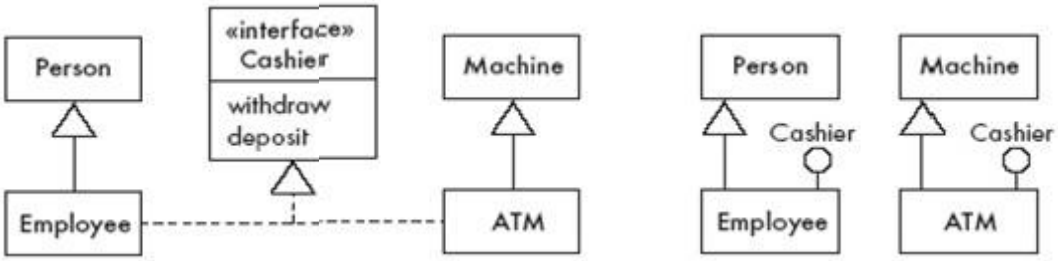


Unlike other associations, UML allows aggregations to be drawn as a hierarchy, as shown .The use of such hierarchies in valid models is quite rare, however. It is a much less flexible way to model vehicle parts than the first diagram.



Interfaces

An *interface* is similar to a class, except it lacks instance variables and implemented methods. It normally contains only abstract methods although it may also contain class variables. We can say that an interface describes a *portion of the visible behavior*of a set of objects.



Two ways of showing the cashier interface

Constraints, notes and descriptive text

Very often, in a class diagram, you want to say more than the graphical UMLnotation readily allows. There are three ways in which you can add additional information to a UML diagram:

Descriptive text and other diagrams

It is highly recommended to embedyour diagrams in a larger document that describes the system more fully.

Notes. In contrast to the descriptive text described above, a note is a smallblock of text embedded in a UML diagram. The box has a 'bent corner'.

Constraints

A constraint is like a note, except that it is written in a formallanguage that can be interpreted by a computer. In a UML diagram, a constraint is shown in curly brackets (also called 'braces').

Modeling interactions and behavior

We look at how to model system dynamics, focusing on two aspects: **interactions and behavior.**

An **interaction model** shows a set of actors and objects interacting by exchanging messages.

A **behavior model** shows how an object or system changes state in reaction to a series of events.

Twotypes of UML interaction diagram used to model detailed sce narios of system execution: sequence diagrams and communication diagrams.State and activi ty diagrams, two other UML diagram types that are used to model the possible behavior of a system.

Interaction diagrams

Interaction diagrams are used to model the dynamic aspects of a software system – they help to visualize how the system runs. They show how a set of actors and objects communicate with each other to perform the steps of a use case, or of some other piece of functionality. The set of steps, taken together, is called an interaction.Interaction diagrams can show several different types of communication.

These include messages exchanged over a network, simple procedure calls, and commands issued by an actor through the user interface. Collectively, these are referred to as messages.Elements found in an interaction diagram:

Instances of classes or actors

Instances of classes (i.e. objects) are shown as boxes with the class and object identifier underlined. Actors are shown using the same stick-person symbol as in use case diagrams.

Messages

These are shown as arrows from actor to object or from object to object. One of the main objectives of drawing interaction diagrams is to betterunderstand the sequence of messages.

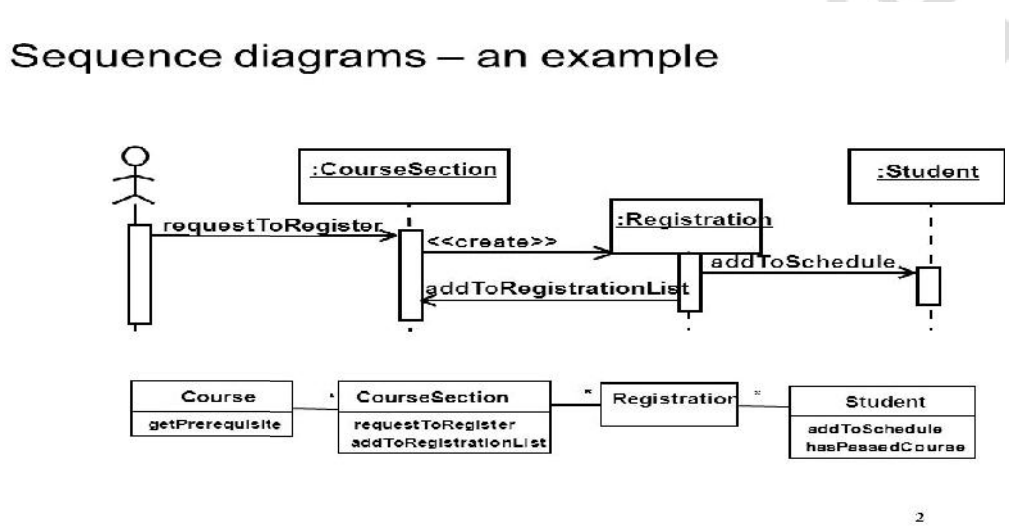
Two kinds of diagrams are used to show interactions:

- *sequence diagrams*
- *communication* /collaboration *diagrams*

Sequence diagrams

- A sequence diagram shows the sequence of messages exchanged by the set of objects (and optionally an actor) performing a certain task.
- The objects are arranged from left to right across the diagram – an actor that initiates the interaction is often shown on the left.
- The vertical dimension represents time.

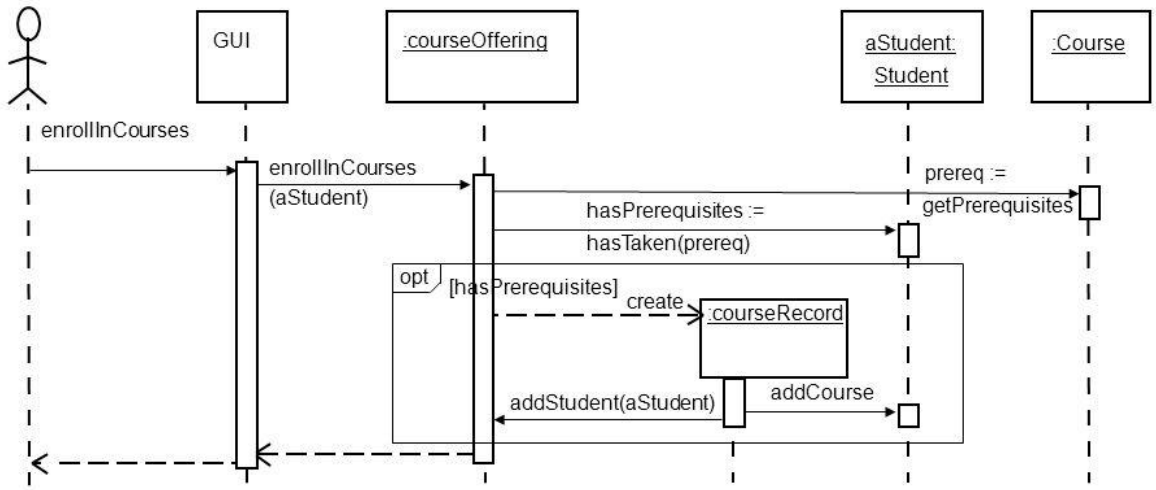
- The top of the diagram is the starting point, and time progresses downwards towards the bottom of the diagram.
- A vertical dashed line, called a *lifeline*, is attached to each object or actor.
- The lifeline becomes a box, called an *activation box*, during the period of time that the object is performing computations. The object is said to have *live activation* during these times.
- A message is represented as an arrow between activation boxes of the sender and receiver. You give each message a label; it can optionally have an argument list and a response. The complete syntax is as follows:



response:=message(arg,...)

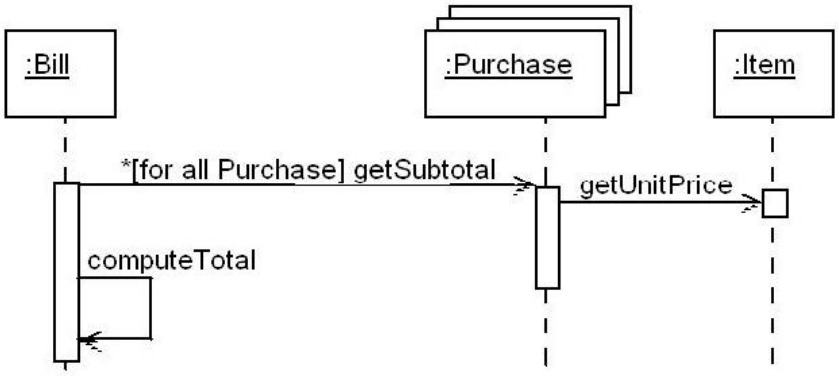
As with class diagrams, ir teraction diagrams can be drawn at various levels of detail. The level of detail you choose depends on what you wish to communicate.

A Sequence Diagram showing more detail

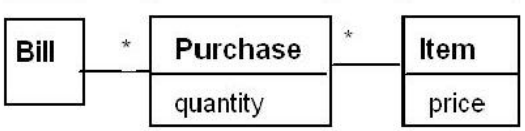


 Chapter 8: Modelling Interactions and Behaviour 6

A sequence diagram showing more detail about the student registration processincluding an optional combined fragment

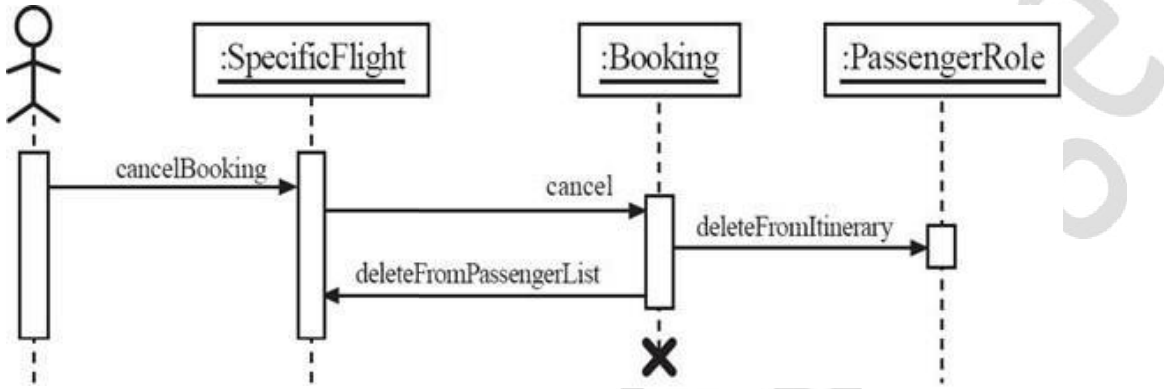


A sequence diagram showing a loop fragment



Class diagram

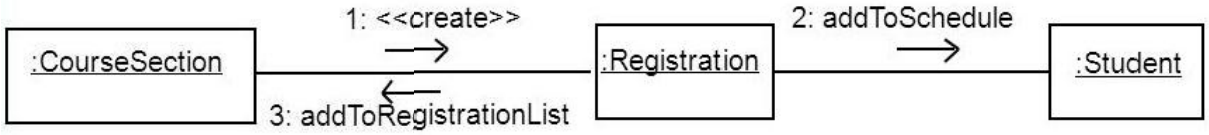
A sequence diagram can show the destruction of an object using a big **X** symbol on a lifeline.



Communication diagrams/Collaboration diagram

A communication diagram shows several objects working together. It appears asa graph with a set of objects and actors as the vertices.

A communication diagram is very much like an object diagram except that, as we will discuss below, it shows *communication* links instead of links of associations. It also has much in common with a sequence diagram, except that lifelines, activation boxes and combined fragments are absent. Instead, you draw a communication link between each pair of objects involved in the sending of a message; the messages themselves are attached to this link. You represent a message using an arrow, labeled with the message name and optional arguments. You specify the order in which messages are sent by prefixing each message using some numbering scheme.



Communication links can exist between two objects whenever it is possible for one object to send a message to the other one. Several situations can make this possible:
The classes of the two objects are joined by an association. This is the most common case.
The receiving object is stored in a *local* variable of the sending method (but theobjects are not yet joined by an association). we tag such a message with thestereotype 《local》.
A reference to the receiving object has been received as a parameter of anearlier message to the sender.

The receiving object is *global*. This is the case when a reference to an object canbe obtained using a public static method (e.g. using the Singleton pattern). Thestereotype《global》
The objects communicate over a network. The stereotype《network》 could be used to show this.

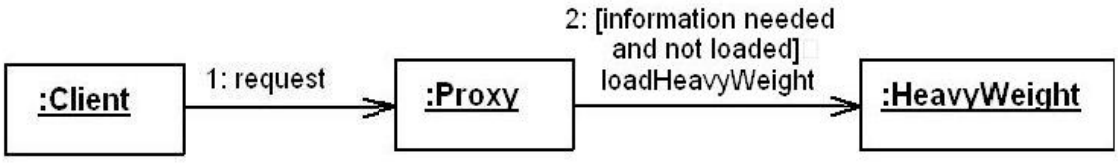
How to choose between using a sequence or a communication diagram?

Since sequence and communication diagrams contain much the sameinformation, you have to decide which of the two you should draw. Sequencediagrams are often the better choice in the following four situations:

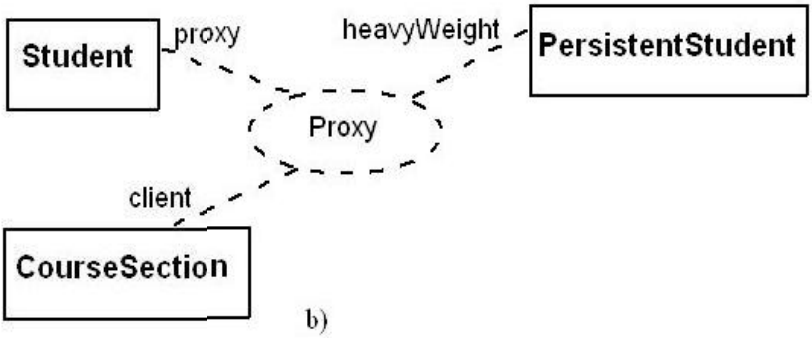
You want the reader to be able to easily see the order in which messages occur.
You want to build an interaction model from a use case. Use cases already hav a sequence of steps; sequence diagrams expand on these to show which objectsare involved.
You need to show details of messages, such as parameters and return values. Doing so on communication diagrams can result in clutter.
You need to show loops, optional sequences and other things that can only beproperly expressed using combined fragments.

Communication diagrams, patterns and collaborations

A communication diagram can be used to represent aspects of a design pattern the two steps involved in the main interaction of the Proxy pattern. First, a client object makes a request to a<<Proxy>>object. Then, if the <<HeavyWeight>> is needed and is not already loaded, the <<Proxy》causes it to be loaded, before returning the result to the client.



a)



b)

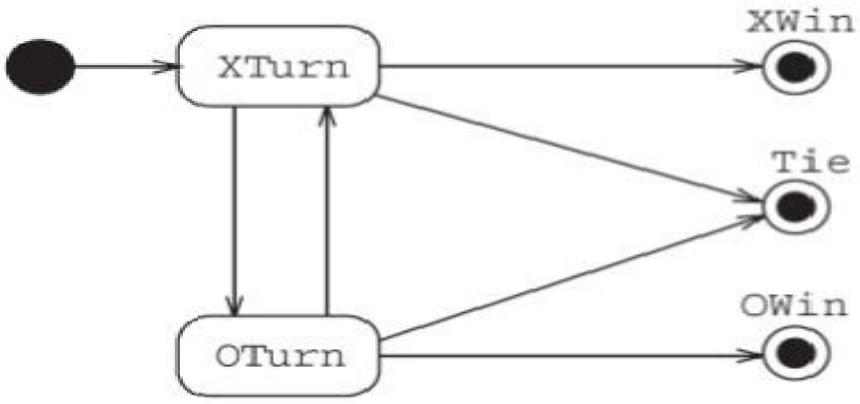
State diagrams

A state diagram, also known as a state machine diagram, is another way of expressing dynamic information about a system. It is used to describe the externally visible behavior of a system or of an individual object.

At any given point in time, the system or object is said to be in a certain state.
It remains in this state until an event occurs that causes it to change state. Being in a state means that it behaves in a specific way in response to any events thatoccur. You represent a state using a rounded rectangle that contains the name ofthe state.

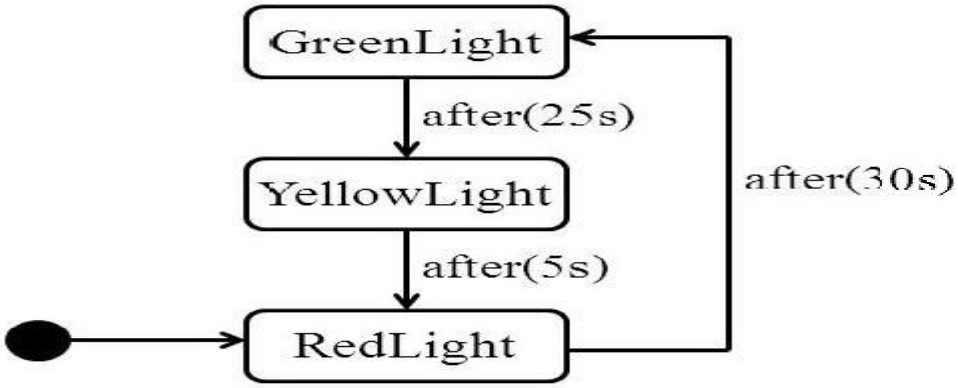
A transition represents a change of state in response to an event, and is considered to occur instantaneously – that is, to take no time. You draw a transition using an arrow connecting two states. You can also show a label on a transition; this represents the event that causes the change of state.

There are **two** other special symbols that can appear on a state diagram:
A black circle represents the *start state*.
A black circle with a ring around it represents an *end state*.



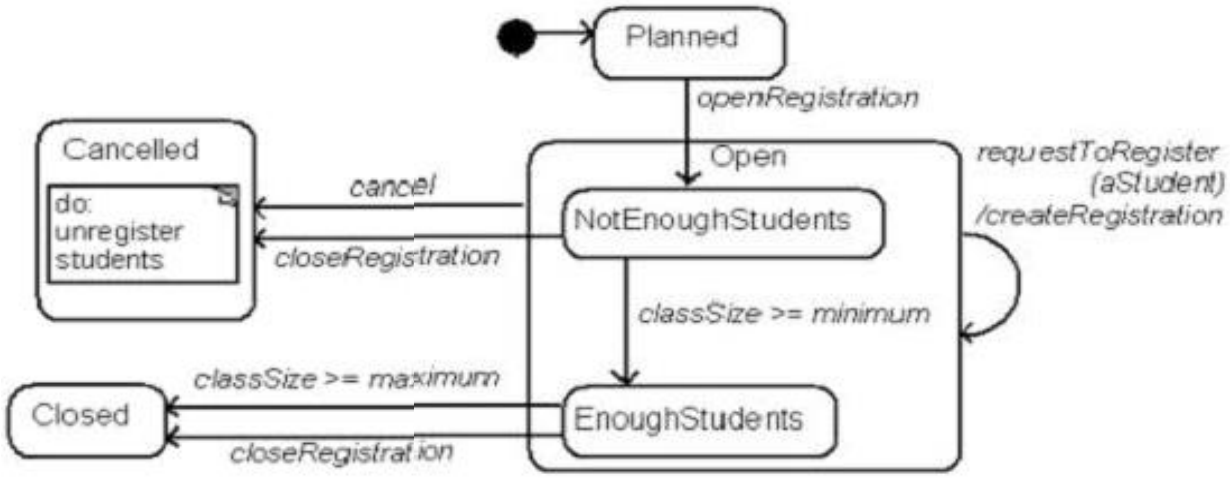
• tic-tac-toe game
(also called noughts and crosses)

Elapsed-time transitions
The event that triggers a transition can be a certain amount of elapsed time.



Transitions triggered by a condition becoming true

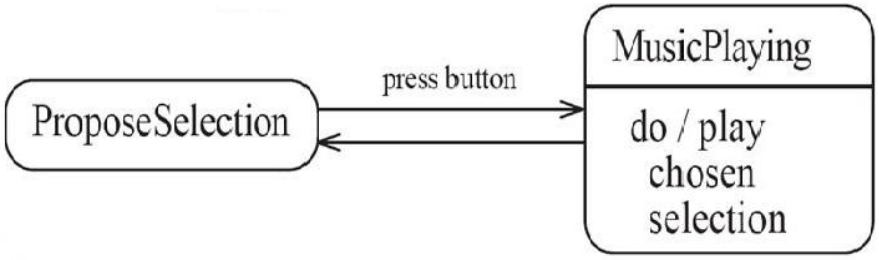
A condition can be distinguished from an event name since it contains a Boolean operator. The two conditions in this figure are classSize>= minimum, and classSize>= maximum.



Activities and actions in state diagrams

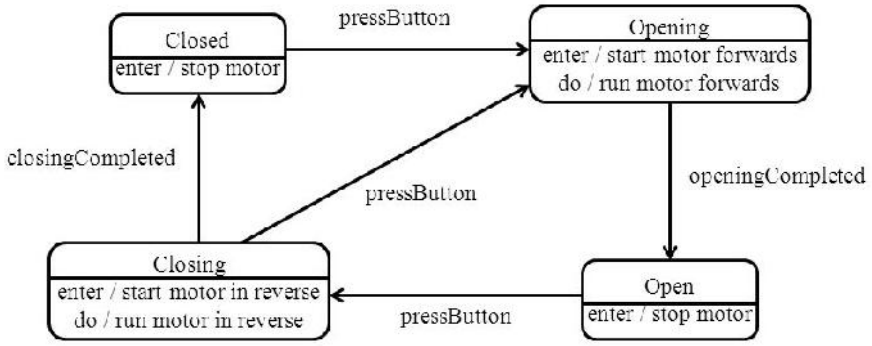
You can represent **two kinds of computations** using state diagrams.

An activity is something that occurs over a period of time and takes place while the system is in a state. An activity is shown textually within a state box by the word 'do' followed by a '/' symbol, and a description of what is to be done. When you have details such as actions in a state, you draw a horizontal line above them to separate them from the state name.



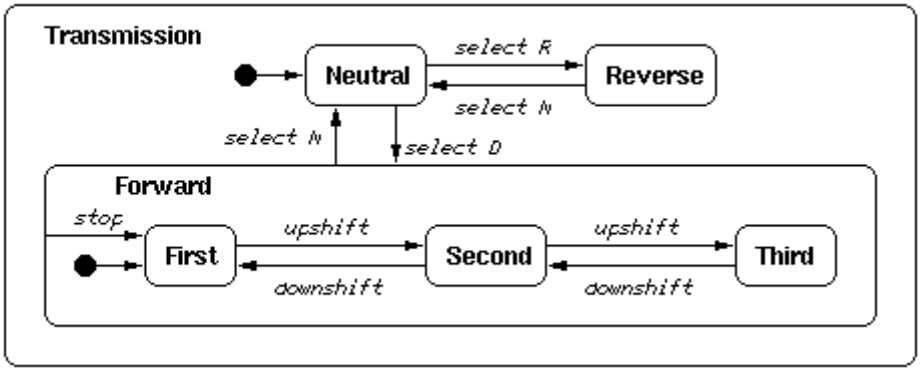
An action is something that takes place effectively instantaneously in any of the following situations:
When the system takes a particular transition.
Upon entry into a particular state, no matter which transition causes entry into that state.
Upon exit from a particular state, no matter which transition is being taken.

An action should take place with no noticeable consumption of time; therefore itshould be something simple, such as sending a message, starting a hardware d vice orsetting a variable.An action is always shown preceded by a slash ('/') symbol. If the action is to beperformed during a transition, then the syntax is event/action. If the action is to beperformed when entering or exiting a state, then it is written in the state boxwith the notation enter/action or exit/action.



Nested substates and guard co ditions

A state diagram can be nested inside a state. The states of the inner diagram arecalled *substates*.



Activity diagrams

An activity diagram is like a state diagram, except that it has a few additionalsymbols and is used in a different context. In a state diagram, most transitionsare caused by *external* events; however, in an activity diagram, most transitionsare caused by *internal* events, such as the completion of an activity.

One of the strengths of activity diagrams is the representation of concurrentactivities. Concurrency is shown using forks, joins and rendezvous, all three of which are represented as short lines, at which transitions can start and end.

A***fork***has one incoming transition and multiple outgoing transitions. The result is that execution splits into two concurrent threads.

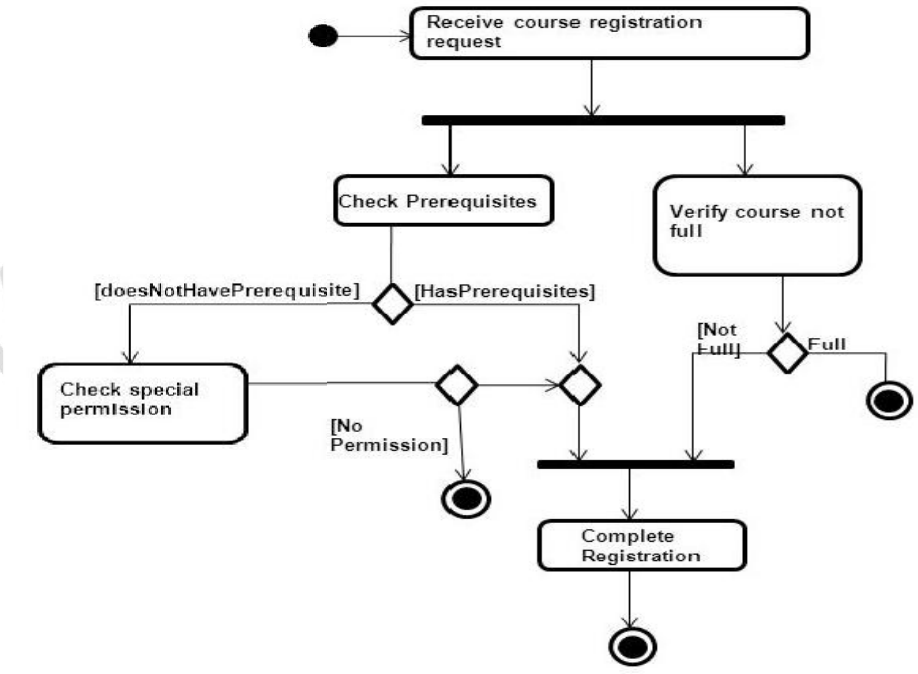
A***join***has multiple incoming transitions and one outgoing transition. The outgoing transition will be taken only when all incoming transitions have been triggered. The incoming transitions must be triggered in separate threads.

A***rendezvous***has multiple incoming and multiple outgoing transitio s. Once all the incoming transitions are triggered, the system takes all the outgoing transitions, each in a separate thread.

An activity diagram also has two types of nodes for branching within a singlethread. These are represented as small diamonds:

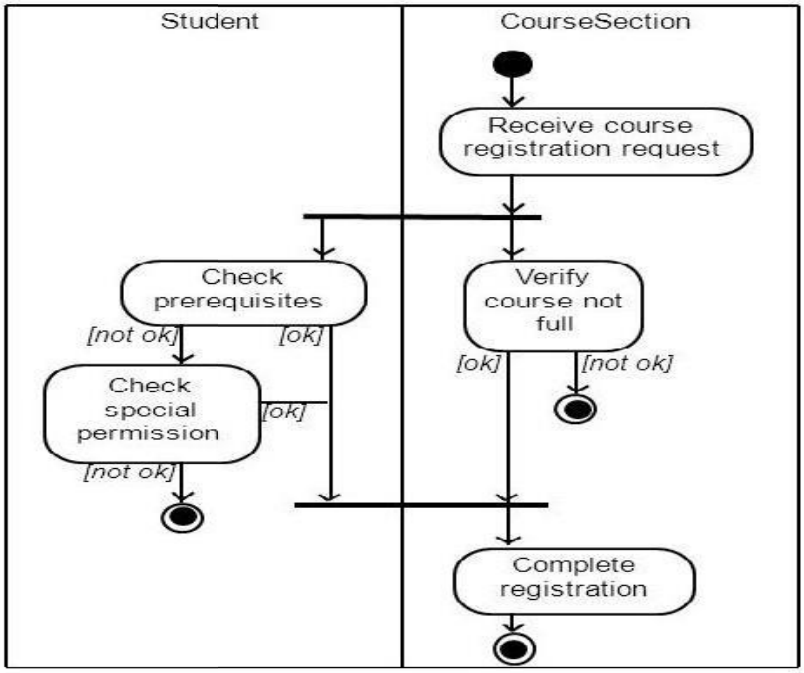
A*decision node* has one incoming transition and multiple outgoing transitions,each with a Boolean guard in square brackets. Exactly one of the outgoing transitions will be taken.

A*merge node* has two incoming transitions and one outgoing transition. It isused to bring together paths that had been split by decision nodes.



While state diagrams typically show states and events concerning only oneclass, activity diagrams are most often associated with several classes. Thepartition of activities among the

existing classes can be explicitly shown in anactivity diagram by the introduction of *swimlanes*. These are boxes that form columns, each containing activities associated with one or more classes.



UNIT - IV
ARCHITECTING AND DESIGNING SOFTWARE

The process of design

Design as a series of decisions

Definition:

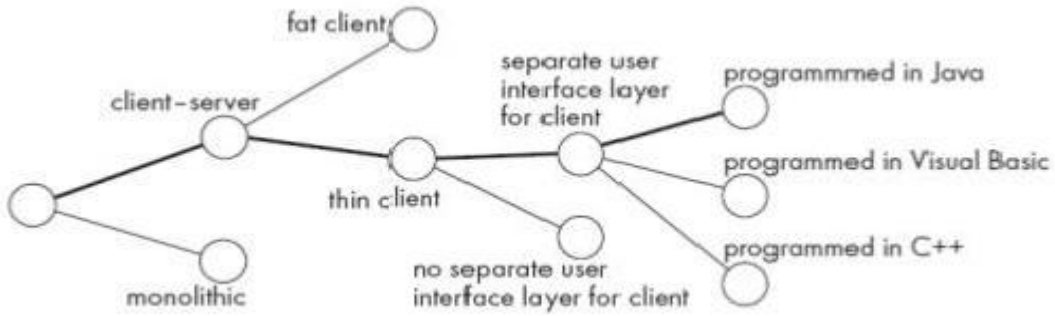
Design is a problem-solving process whose objective is to find and describe a way:

- To implement the system's functional requirements...
- While respecting the constraints imposed by the non-functional requirements... - including the budget
- And while adhering to general principles of good quality

Each issue normally has several alternative solutions, also known as design *options*. The designer makes a *design decision* to resolve each issue. To make each design decision, the software engineer uses all the knowledge including:

- Knowledge of the requirements;
- Knowledge of the design as created so far;
- Knowledge of the technology available;
- Knowledge of software design principles and 'best practices'; and
- Knowledge about what has worked well in the past.

Once a decision is made, new issues are raised. Several different alternatives may have opposite advantages and disadvantages, with no clear 'winner'. The space of possible designs that could be achieved by choosing different sets of alternatives is often called the *design space*. Each design decision should be recorded, along with the reasoning that went into making the decision (known as the *design rationale*).



Parts of a system: subsystems, components and modules

Component: any piece of software or hardware that has a clear role and can be isolated, allowing you to replace it with a different component with equivalentfunctionality.

Module:a component that is defined at the programming language level.
For example, methods, classes and packages are modules in Java.

System :a logical entity, having a set of definable responsibilities or objectives, and consisting of hardware, software or both.

Subsystem:a system that is part of a larger system, and which has a definite interface. Java uses packages to implement subsystems ;

Top-down versus bottom-up design

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.

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.

Special types of design

There are many different aspects of software design, including:

Architecture design: the division of software into subsystems and components, as well as the process of deciding how these will be connected and how theywill interact, are including determining their interfaces.

Class design: the design of the various features of classes such as associations, attributes, interactions and states.

User interface design.

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.
Algorithm design: the design of computational mechanisms.
Protocol design: the design of communications protocols – the languages with which processes communicate with each other over a network.

Principles leading to good design

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 thesoftware 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.

Design Principle 1: Divide and conquer

Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things. We have already seen how the process of development is divided into activities such as requirements gathering, design and testing.

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 notpossible to know everything about an entiresystem.

Each individual component is smaller, and therefore easier to understand.
When one part needs to be replaced or changed, this can hopefully be donewithout 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.

Design Principle 2: Increase cohesion where possible

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 tryto achieve.

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-levelservices 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

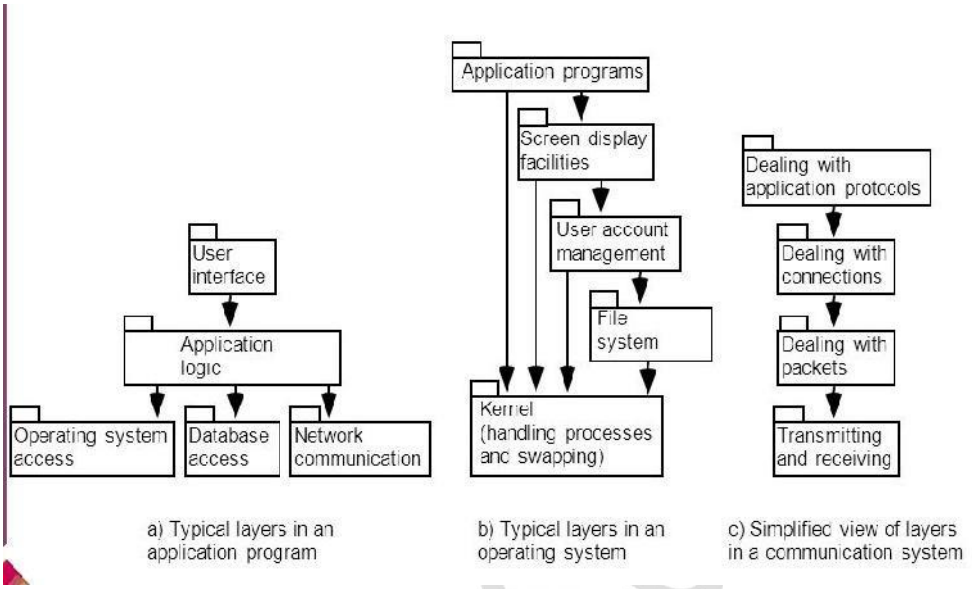
Functional cohesion

The following are some examples of modules that can be designed to befunctionally 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 monitoringdevices and computes the yield of a chemical process as a percentage of thetheoretical maximum.

There are several reasons why it is good to achieve functional cohesion:
It is easier to understand a module when you know that all it does is generateonespecific output and has no side effects.
Due to its lack of side effects, a functionally cohesive module is much morelikely to be reusable.
It is easier to replace a functionally cohesive module with another thatperforms the same computation.

Layer cohesion

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.



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

Services for computation;
Services s 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.

Advantages of layer cohesion are:

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

You know you can replace a lower layer with an equivalent layer, because youknow it does not access higher layers.

Communicational cohesion

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

A class would have good communicational cohesion if

- all the system's facilities for storing and manipulating its data are contained in this class.
- the class does not do anything other than manage its data.

Main advantage:

When you need to make changes to the data, you find all the code in one place

Sequential cohesion

Procedures, in which one procedure provides input to the next, are kept together and everything else is kept out

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 modulethat 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 woul d 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.

Temporal cohesion

This is achieved when operations that are performed during the same phase ofthe execution of the program are kept together, and everything else is kept out. This is weaker than procedural cohesion

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.

Design Principle 3: Reduce coupling where possible

Coupling occurs when there are interdependencies between one module andanother. 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

A network of interdependencies makes it hard to see at a glance how somecomponent works.

Coupling type	**Comments**
Content	A component *surreptitiously modifying internal data* of another component. Always avoid this

Common Control	The use of *global variables*. Severely restrict this One procedure *directly controlling another* using a flag. Reduce this using polymorphism
Stamp	One of the *argument types* of a method is one of you *application classes*. If it simplifies the system, replace each such argument with a simpler argument (an interface, super class or a few simple data items)
Data	The use of *method arguments that are simple data*.Ifpossible, 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 (super classes 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

Design Principle 4: Keep the level of abstraction as high as possible

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.

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

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

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

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

Design Principle 5: Increase reusability where possible

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

Important strategies for increasing reusability are as follows:

Generalize your design as much as possible.
Follow the preceding three design principles.
Design your system to contain hooks. A hook is an aspect of the design deliberately added to allow other designers to add additional functionality.
Simplify your design as much as possible.

Design Principle 6: Reuse existing designs and code where possible

Designing with reuse is complementary to designing for reusability. Activelyreusing designs or code allows you to take advantage of the investment you orothers 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;

Design Principle 7: Design for flexibility

Designing for *flexibility* (also known as *adaptability*) means actively anticipatingchanges that a design may have to undergo in the future and preparing for them. Such changes might include changes in implementation

Ways to build flexibility into a design include:

Reducing coupling and increasing cohesion.
Creating abstractions.
Not hard-coding anything.
Leaving all options open.
Using reusable code and making code reusable.

Design Principle 8: Anticipate obsolescence

Anticipating obsolescence means planning forevolution of the technology or environment so that the software will continue torun or can be easily changed.

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

Avoid using early releases of technology.
Avoid using software libraries that are specific to particular environments.
Avoid using undocumented features or little-used features of software libraries.
Avoid using reusable software or special hardware from smaller companies, or from those that are less likely to provide long-term support.
Use standard languages and technologies that are supported by multiplevendors.

Design Principle 9: Design for portability

Designing for portability shares many things in common with anticipatingobsolescence, 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.

Design Principle 10: Design for testability

During design you can take steps to make testing easier. Testing can 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.

Design Principle 11: 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.

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 eachmeth od has an explicit contract with its callers. The contract has a set of assertions that state:

Whatpreconditions the called method requires to be true when it sta rtsexecuting. The caller has the responsibility to make these preconditions truebefore making the call.

Whatpost conditions the called method agrees to ensure are true when itfinishes executing. The called method has the responsibility to make thesepost conditions true, before returning.

Whatinvariants the called method agrees will not change as it executes.

Techniques for making good design decisions

Two approaches that will help you to make decisions.

1) Using priorities and objectives to decide among alternatives

Before you start design, you should have established priorities and objectives for various aspects of quality. An objective is a measurable value you wish to attain. A priority states which qualities override others in those cases where you must make compromises. The qualities to consider when setting priorities and objectives include memory efficiency, CPU efficiency, maintainability, portability and usability. In general, the priorities and objectives should be obtained from the non-functional requirements.

In order to make a design decision, you can perform the following steps:

Step 1 List and describe the alternatives for the design decision.

Step 2 List the advantages and disadvantages of each alternative with respect to yourobjectives and priorities.

Step 3 Determine whether any of the alternatives prevents you from meeting one ormore of the objectives.

Step 4 Choose the alternative that helps you to best meet your objectives.

Step 5 Adjust your priorities for subsequent decision making.

2) Using cost–benefit analysis to choose among alternatives

An important consideration when you do design is finding ways to reduce costs and increase benefits. To estimate the costs of a new feature or design alternative, you should add up estimates of the following:

The incremental cost of doing the *software engineering* work, including ongoing maintenance for the life of the system.The incremental costs of any *development technology* that you will have to buy such as programming languages, reusable components, databases etc.

The incremental costs that *end-users and product support personnel* will experience.To estimate the *benefits* of a new feature or design alternative, you should add up the following:The incremental software engineering time saved. The incremental benefits measured in terms of either increased sales or elsefinancial benefit to users.

Software architecture

Definition:

Software architecture is the process of designing the global organization of asoftware system, including dividing software into subsystems, deciding how these will interact, and determining their interfaces.

The importance of developing an architectural model

Software engineers discuss all aspects of a system's design in terms of the architectural model. Decisions made while this model is being developed therefore have a profound impact on the rest of the design process. The architectural model is the core of the design; therefore all software engineers need to understand it.

There are **four main reasons** why you need to develop an architectural model:

To enable everyone to better understand the system.
To allow people to work on individual pieces of the system in isolation.
To prepare for extension of the system.
To facilitate reuse and reusability.

Contents of a good architectural model

A system's architecture will often be expressed in terms of several different *views*. These can include:

The logical breakdown into subsystems. This is often shown using package diagrams, which we will describe later. The interfaces among the subsystems must also be carefully described.

The dynamics of the interaction among components at run time, perhaps expressed using interaction or activity diagrams.

The data that will be shared among the subsystems, typically expressed using class diagrams.

The components that will exist at run time, and the machines or devices on whichthey will be located. This information can be expressed using component and deployment diagrams.

To ensure the maintainability and reliability of a system, an architecturalmodel must be designed to be *stable*. Being stable means that the new featurescan be easily added with only small changes to the architecture.

How to develop an architectural model

The basis for the architectural model will be the system domain model and the use cases. The first draft of the architectural model should be created at the same time as these. These give the architect an idea about which components will be needed and how they will interact. At the same time, the early architecture will give use case modelers guidance about the steps the user will need to perform.
The following are some steps that you can use iteratively as you refine the architecture.

1. Start by sketching an outline of the architecture, based on the principalrequirements, including the domain model and use cases.
2. Refine the architecture by identifying the main ways in which the componentswill interact, and by identifying the interfaces among them.
3. Consider each use case, adjusting the architecture to make it realizable.
4. Mature the architecture as you define the final class diagrams and interactiondiagrams.

Describing an architecture using UML

All UML diagrams can be useful to describe aspects of the architectural model. Three other types of UML diagram are particularly important for architecture modeling: **package diagrams, component diagrams and deployment diagrams.**

Packages

Breaking a large system into subsystems is a fundamental principle of softwaredevelopment. A good decomposition helps make the system more understandable and therefore facilitates its maintainability. In UML, a *package* is a collection of modeling elements that are grouped together because they are logically related.

Component diagrams

A component diagram shows how a system's components – that is, the physical elements such as files, executables, etc. – relate to each other. The UML symbolfor a component is a box with a little 'plug' symbol in the top-right corner.
Various relationships can exist among components, for example:

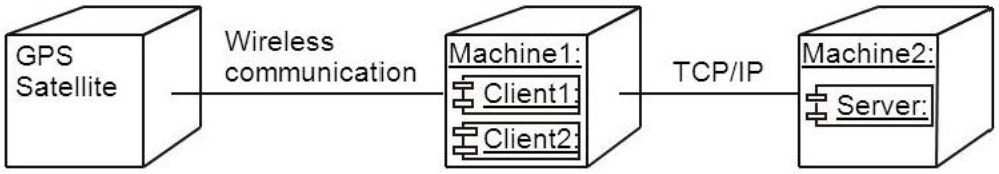
A component may *execute* another component, or a method in the othercomponent.
A component may *generate* another component.
Two components may *communicate* with each other using a network.

The difference is that package diagrams show *logical groupings* of design elements, whereas component diagrams show relationships among types of *physical* components.

Deployment diagrams

A deployment diagram describes the hardware where various instances of components reside at run time.

Deployment diagrams



Architectural patterns

We present several of the most important *architectural patterns*, which are also often called *architectural styles*.

The Multi-Layer architectural pattern

In a layered system, each layer communicates only with the layer immediately below it.

Each layer has a well-defined interface used by the layer immediately above.

- The higher layer sees the lower layer as a set of services.

A complex system can be built by superposing layers at increasing levels of abstraction.

- It is important to have a separate layer for the UI.
- Layers immediately below the UI layer provide the application functions determined by the use-cases.
- Bottom layers provide general services.
- e.g. network communication, database access

Design Principles:

1. Divide and conquer: The layers can be independently designed.
2. Increase cohesion: Well-designed layers have layer cohesion.
3. Reduce coupling: Well-designed lower layers do not know about the higher layers and the only connection between layers is through the API.
4. Increase abstraction: you do not need to know the details of how the lower layers are implemented.
5. Increase reusability: The lower layers can often be designed generically
6. Increase reuse: You can often reuse layers built by others that provide the services you need.
7. Increase flexibility: you can add new facilities built on lower-level services, or replace higher-level layers.
8. Anticipate obsolescence: By isolating components in separate layers, the system becomes more resistant to obsolescence.
9. Design for portability: All the dependent facilities can be isolated in one of the lower layers.
10. Design for testability: Layers can be tested independently.
11. Design defensively: The APIs of layers are natural places to build in rigorous assertion-checking.

The Client–Server and other distributed architectural patterns

There is at least one component that has the role of server, waiting for and then handling connections.

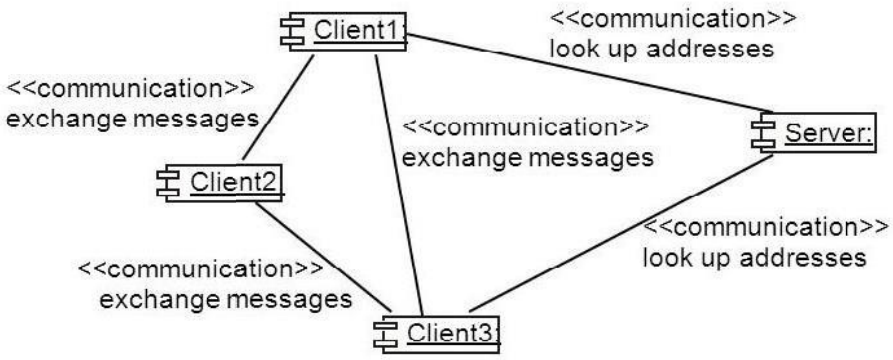
There is at least one component that has the role of client, initiating connections in order to obtain some service.

A further extension is the Peer-to-Peer pattern. —A system composed of various software
components that are distributed over several hosts.

Its basic principles are: a) there is at least one component that has the role of server, waiting for and then handling connections, and b) there is at least onecomponent that has the role of client, initiating connections in order to obtainsome service.

An important variant of the client–server architecture is the three-tier model under which a server communicates with both a client (usually through the Internet) and a database server (usually within an intranet, for security reasons).

The server acts as a client when accessing the database server.



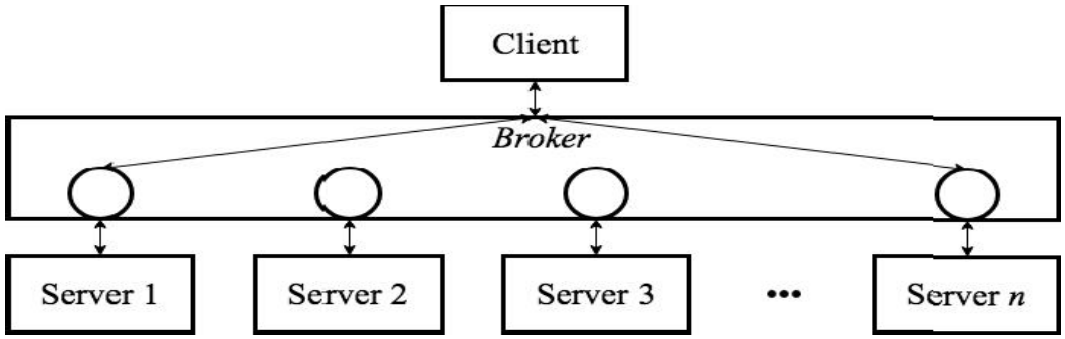
Distributed architectures help you adhere to design principles such as thefollowing:

1. *Divide and conquer*. Dividing the system into client and server processes is a very strong way to divide the system. Each can be separately developed.
2. *Increase cohesion*. The server can provide a cohesive service to clients
3. *Reduce coupling*. There is usually only one communication channel between distributed components, and the data being passed is usually simple messages.
4. *Increase abstraction*. Separate distributed components are often good abstractions. For example, you do not need to understand the details of how a server operates.
5. *Increase reuse*. It is often possible to find suitable frameworks on which to build good distributed systems (e.g. OCSF). However, reusability may not be high since client–server systems are often very application specific.
6. *Design for flexibility*. Distributed systems can often be easily reconfigured by adding extra servers or clients.
7. *Design for portability*. You can write clients for new platforms without having to port the server.
8. *Design for testability*. You can test clients and servers independently.
9. *Design defensively*. You can put rigorous checks in the message handling code to ensure that no matter what messages you receive.

The Broker architectural pattern

The idea of the Broker architectural pattern is to distribute aspects of the software system *transparently* to different nodes. Using the Broker architecture, an object can call methods of another object without knowing that this object is remotely located. The use of the Proxy design pattern can help achieve this goal.

CORBA is a well-known open standard that allows you to build this kind of architecture – it stands for Common Object Request Broker Architecture.



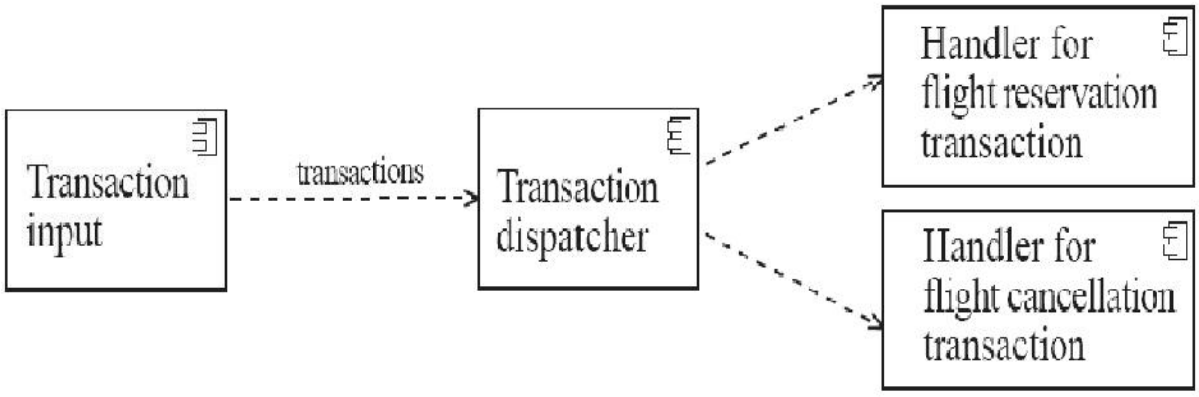
The Broker pattern is particularly useful in helping you follow these designprinciples:

1. *Divide and conquer*. The remote objects can be independently designed.
2. *Increase reusability*. It is often possible to design the remote objects so that other systems can use them too.
3. *Increase reuse*. You may be able to reuse remote objects that others have created.
4. *Design for flexibility*. The broker objects can be updated as required, or you can redirect the proxy to communicate with a different remote object.
5. *Design for portability*. You can write clients for new platforms while still accessing brokers and remote objects on other platforms.
6. *Design defensively*. You can provide careful assertion checking in the remote objects

The Transaction Processing architectural pattern

A process reads a series of inputs one by one.

- Each input describes a transaction – a command that typically some change to the data stored by the system
- There is a transaction dispatcher component that decides what to do with each transaction
- This dispatches a procedure call or message to one of a series of component that will handle the transaction

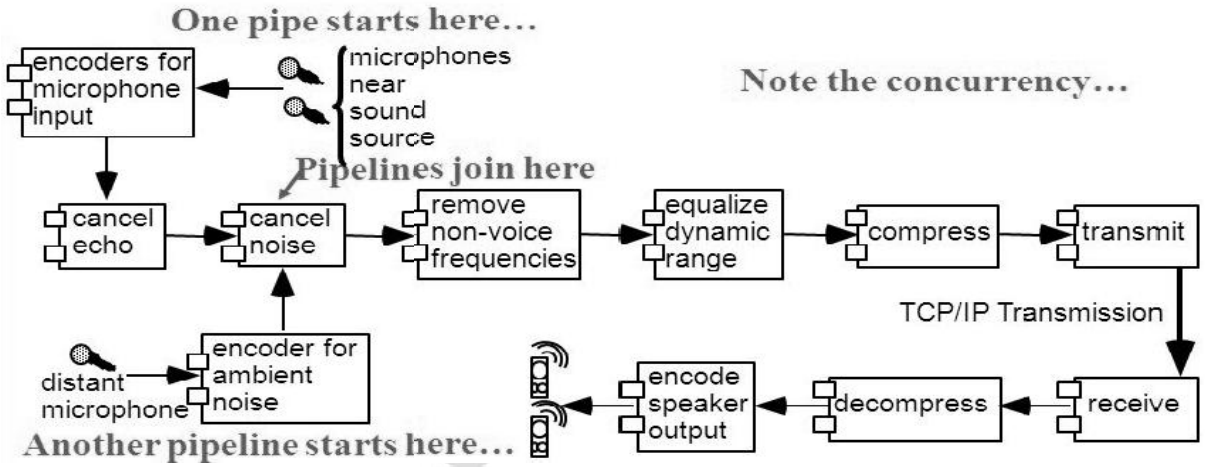


1. *Divide and conquer*. The transaction handlers are suitable system divisions that you can give to separate software engineers.

2. *Increase cohesion*. Transaction handlers are naturally cohesive units. They may exhibit functional, sequential or procedural cohesion. However, they tend not to exhibit communicational cohesion.
3. *Reduce coupling*. Separating the dispatcher from the handlers tends to reduce coupling. However, you have to be careful that the coupling among the transaction handlers is kept under control.
4. *Design for flexibility*. You can readily add new transaction handlers.
5. *Design defensively*. You can add assertion checking in each transaction handler and/or in the dispatcher.

The Pipe-and-Filter architectural pattern

The Pipe-and-Filter architectural pattern is also often called the *transformational* architectural pattern. It works as follows. A stream of data, in a relatively simple format, is passed through a series of processes, each of which transforms it in some way. The series of processes is called a *pipeline*. Data is constantly fed into the pipeline; the processes work concurrently (conceptually at least) so that data is also constantly emerging from the pipeline.



A pipe-and-filter system provides fulfills the following principles:

1. *Divide and conquer*. The separate processes can be independently designed.
2. *Increase cohesion*. The processes generally have functional cohesion.
3. *Reduce coupling*. The processes have only one input and one output, normally using a standard format, therefore coupling is very low. Type use coupling can become an issue if the format of the data needs to change.
4. *Increase abstraction*. The pipeline components are often good abstractions, hiding their internal details.
5. *Increase reusability*. The processes can often be used in many different contexts.
6. *Increase reuse*. It is often possible to find reusable components to insert into a pipeline.
7. *Design for flexibility*. There are several ways in which the system is flexible:
   Almost all the components could be removed.
   Components could be replaced with different implementations.
   New components could be inserted, for example to perform encryption.
   Certain components could be reordered.

The encoders for microphone input and for ambient noise could be instances of the same component type.

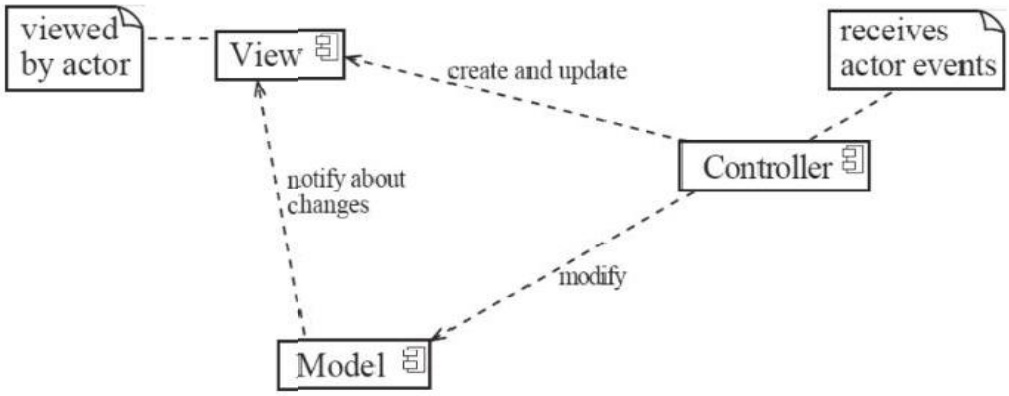
8. *Design for testability*. It is normally easy to test the individual processes.
9. *Design defensively*. You can check the inputs of each component, or you can use design by contract, writing careful preconditions and post conditions for each component.

The Model–View–Controller (MVC) architectural pattern

Model–View–Controller, or MVC, is an architectural pattern used to help separate the user interface layer from other parts of the system. Not only does MVC help enforce layer cohesion of the user interface layer, but it also helps reduce the coupling between that layer and the rest of the system, as well as between different aspects of the UI itself. The MVC pattern separates the functional layer of the system (the *model*) from two aspects of the user interface, the *view* and the *controller*.

The MVC architectural pattern allows us to adhere to the following design prin iples:

1. *Divide and conquer*. The three components can be somewhat independently designed.
2. *Increase cohesion*. The components have stronger layer cohesion than if the view and controller were together in a single UI layer.
3. *Reduce coupling*. The communication channels between the three components are minimal and easy to find.
4. *Increase reuse*. The view and controller normally make extensive use of reusable components for various kinds of UI controls. The UI, however will become application specific, therefore it will not be easily reusable.
5. *Design for flexibility*. It is usually quite easy to change the UI by changing the view, the controller, or both.
6. *Design for testability*. You can test the application separately from the UI.



Writing a good design document

Design documents serve two main purposes.

Firstly, they help you, as a designer or a design team, to *make good design decisions*. The process of writing down your design helps you to think more clearly about it and to find flaws in it.

Secondly, they help you *communicate the design* to others.

- Design documents as an aid to making better designs

- Design documents help you, as a designer, because they force you to be explicit and to consider the important issues before starting implementation. They also allow a group of people to review the design and therefore to improve it.
- There has been a tendency among software developers to omit design documentation or to document the design only *after* it is complete.
- Design documents as a means of communication when writing anything, it is important to know the *audience* for your work.

Design documents are used to communicate with three groups of individuals. Ingeneral; you can expect most documents to be read by all **three groups**:
Those who will be *implementing* the design, that is, the programmers
Those who will need, in the future, to *modify* the design
Those who need to create systems or subsystems that *interface* with the systembeing designed.

Contents of a design document

We suggest that a design document should contain the following information.

A. **Purpose**. Specify what system or part of the system this design documentdescribes
B. **General priorities**. Describe the priorities used to guide the design process.
C. **Outline of the design**. Give a high-level description of the design that allowsthe reader to get a general feeling for it quickly.
D. **Major design issues**. Discuss the important issues that had to be resolved.
E. **Details of the design**. Give any other details the reader will need to know that have not yet been mentioned.

At the same time, remember that there is no point writinginformation that would never be read because the reader already knows it or caneasily find it from some other source. In particular:

Avoid documenting information that would be readily obvious to a skilledprogrammer or designer.

Avoid writing details in a design document that would be better placed ascomments in the code.

Avoidwriting details that can be extracted automatically from the code, such asthe list of public methods.

UNIT - V
TESTING AND INSPECTING TO ENSURE HIGH QUALITY

Basic definitions

Definition: a *failure* is an unacceptable behavior exhibited by a system.
Definition: a *defect* is a flaw in any aspect of the system including the requirements, the design and the code, that contributes, or may potentially contribute, to the occurrence of one or more failures. A defect is also known as a *fault*.

Definition: an *error* is a slip-up or inappropriate decision by a software developer that leads to the introduction of a defect into the system.

Effective and efficient testing

Testing is the processes of deliberately trying to cause failures in a system in orderto detect any defects that might be present. As with all engineering activities, efficiency and effectiveness are both important.

To test *effectively*, you must use a strategy that uncovers as many defects as possible.
To test *efficiently*, you must find the largest possible number of defects using the fewest possible tests. An effective and efficient testing strategy is often called a *high-yield* strategy.

Black-box and glass-box testing

Most of the time, testers treat a system as a *black box*. This means they providethe system with inputs and observe the outputs, but they cannot see what isgoing on inside. In particular, they can see neither the source code, the internaldata, nor any of the design documentation describing the system's internals.
An alternative approach to testing is to treat the system as a *glass box*. In glass boxtesting, the tester can examine the design documents and the code, as wellas observe at run time the steps taken by algorithms and their internal data.

Glass-box testing is widely referred to as *white-box* testing orstructural testing;
Once you have a flow graph you must then choose from the followingstrategies for glass-box testing:
Covering all possible paths. This is infeasible in graphs with loops since therewould be an infinite number of paths (i.e. repeatedly looping).
Covering all possible edges. This is probably the most efficient strategy. Youdevise a sufficient set of tests to make sure that each of the outgoing edges of allnodes is taken.
Covering all nodes. This is a less exhaustive and therefore less effectivestrategy.

Testing is like detective work .The job of the tester has certain similarities with that of the detective:

A detective must try to understand the criminal mind. Similarly, the testermust try to understand how programmers, designers and users think, so as tobetter find defects.

Detective work is painstaking. The detective must not leave anythinguncovered, and must be suspicious of everything. Similar suspicion andattention to detail are the hallmark of an effective tester.

Equivalence classes: a strategy for choosing what to testin order to test efficiently, you should divide the possible inputsinto groups that you believe will be treated similarly by reasonable algorithms. Such groups are called *equivalence classes.* A tester needs only torun one test per equivalence class, using a *representative* member of thatequivalence class as input.

Combinations of equivalence classes

To decide how to divide this system into equivalence classes for testing. You are told that the user can enter the following data (ex: land vehicles)

1. Whether the manufacturers give data about the vehicle in metric or traditional (Imperial or US) units.
2. Maximum speed, an integer ranging from 1 to 750 km/h or 1 to 500 mph (four equivalence classes: [−∞..0], [1..500], [501..750], [751..∞]).
3. Type of fuel, one of a set of 10 possible strings that the user explicitly types
4. Average fuel efficiency, a fixed-point value with one decimal place, ranging from 1 to 240 L/100 km or 1 to 240 mpg (three equivalence classes).
5. Time to accelerate to 100 km/h or 60 mph. This is a fixed-point value with one decimal place, ranging from 1 to 100s (three equivalence classes).
6. Range, an integer from 1 to 5000 km or 1 to 3000 miles (four equivalenceclasses: [−∞..0], [1..3000], [3001..5000], [5001..∞]).

The set of equivalence classes for the system as a whole is the set of all possible *combinations* of inputs. Most systems have many distinct inputs; the total number of system equivalence classes can become very large. This is called a *combinatorial explosion* of the space of required tests.

Testing at boundaries of equivalence classesmore errors in software occur at the boundaries of equivalence classes than inthe 'middle'. Therefore we should expand the idea of equivalence class testing tospecifically test values at the extremes of each equivalence class.

Detecting specific categories of defects

As mentioned earlier, a tester must act like a detective, trying to uncover anydefects that other software engineers might have introduced. This means notonly testing at equivalence classes and their boundaries, but also designing teststhat explicitly try to catch a range of specific types of defects that commonlyoccur.

Defects in ordinary algorithms

The following subsections list some of the most common kinds of defects found in all types of algorithms.

Incorrect logical conditions

Defects

The logical conditions that govern looping and if-then-else statements are wrongly formulated. reversing comparison operators (e.g. > becomes <), or mishandling the equality case (e.g. >= becomes > or vice versa).

Testing strategy

Use equivalence class and boundary testing. To compute the equivalence classes, consider each variable used in the logical condition as an input.

Ex:

```
if(!landingGearDeployed&&
(min(now-takeoffTime,estLandTime-now))<
(visibility< 1000 ? 180 :120) ||
relativeAltitude<
(visibility< 1000 ? 2500 :2000)
)
{
throw new LandingGearException();
}
```

Unfortunately, this type of bad style that gives rise to defects is rather a commonpractice. There is at least one critical defect in this code – see if you canunderstand the code and find the defect. It is likely that a programmer might notnotice it due to the nested parentheses and the overall complexity of thecondition.

Performing a calculation in the wrong part of a control construct

Defect

The program performs an action when it should not, or does not perform anaction when it should.

Testing strategies

Design tests that execute each loop *zero* times, exactly *once*, and *more than once*. Also, ensure that anything 'bad' or 'unusual' that could happen while looping ismade to occur on the first iteration and the last iteration. This kind of defect is not always reliably caught using black-box testing; insuch cases glass-box testing or inspections may be more effective.

Examples

The following Java code illustrates a typical case:

```
while(j<maximum)
{
k=someOperation(j);
j++;
}
if(k==-1) signalAnError();
```

In this case, signalAnError was supposed to be called if any of the calls tosomeOperation resulted in a value of -1. Unfortunately, here it can only be calledfollowing the last call to someOperation. The final line should therefore have been placed inside a loop. This kind of defect may be missed if the loop normally executes only once.

Not terminating a loop or recursion

Defect testing

A loop or a recursion does not always terminate, that is, it is infinite'.

Strategies

Although the programmer should have analyzed all loops or recursions toensure they reach a terminating case, a tester should nevertheless assume that the programmer has made an error.

Example

Imagine that a program is supposed to count the total number of atoms in acomplex organic molecule. It might do this by starting at an arbitrary moleculeand traversing it from bond to bond, walking down each branch.

Not setting up the correct preconditions for an algorithm

Defect

When specifying an algorithm, one specifies *preconditions* that state what must be true before the algorithm should be executed. A defect would exist if a program proceeds to do its work, even when the preconditions are not satisfied.

Testing strategy

Run test cases in which each precondition is not satisfied. Preferably its input values are just beyond what the algorithm can accept.

Example

In the organic chemistry program, a precondition might be that the input is a single molecule. The tester should therefore try to test by giving as input two disjoint molecules.

Not handling null conditions

Defect

It is defect when a program behaves abnormally when a null condition is encountered. In these situations, the program should 'do nothing, gracefully'.

Testing strategy

Determine all possible null conditions and run test cases that would highlight any inappropriate behavior.

Examples

Imagine you want to calculate the average sales of members of each division in an organization. But what if some division has no members?

As a related example, imagine you are asked to find the salesperson who has sold the most in the above division. In attempting to perform this calculation, an algorithm might loop zero times and hence never actually set the maximum value, or leave it set to some arbitrary, but incorrect value.

Not handling singleton or non-singleton conditions

Defect

A singleton condition is like a null condition. It occurs when there is normally *more than one* of something, but sometimes there is only one. A non-singleton condition is the inverse – there is almost always *one* of something, but occasionally there can be more than one. Defects occur when the unusual case is not properly handled.

Testing strategy

Brainstorm to determine unusual conditions and run appropriate tests.

Examples

The following are two examples of these conditions:

Imagine that in a web browser you can set up a series of 'personal profiles'. Each user of the computer has their own personal profile that includes their name, bookmarks list, and 'cookies'. Imagine you created a personal profile under your name, 'John Smith'. Later on you accidentally created another profile, using the same name. Then you decided to get rid of one of the two profiles; you therefore selected a profile and issued the 'delete' command. Unfortunately, the deletion operation might assume that there can be only one profile using a given name, so that it simply traverses the list of profiles, deleting*all* the profiles by that name. The result? Both of the profiles are deleted. Anticipating this kind of defect, a tester can enter several identically-named profiles and make sure that only one is deleted.

Imagine that a program is designed to randomly assign members of a sports club into pairs who will play against each other. Does the program do something intelligent with the left-over person when there is an odd number of a member? And what happens if there is only one member?

Off-by-one errors

Defect

A program inappropriately adds or subtracts 1, or inappropriately loops one too many times or one too few times. This is a particularly common type of defect.

Testing strategy

Develop boundary tests in which you verify that the program computes the correct numerical answer, or performs the correct number of iterations. Since graphical applications are common places where off-by-one errors are found, study the display to see if objects slightly overlap or have slight gaps.

Examples

Assuming 0-based indexing, as is the case in Java, then the following loop would always skip the first element, and loop one too few times.

for (i=1; i<arrayname.length; i++) { /* do something */ }

Operator precedence errors

Defect

An operator precedence error occurs when a programmer omits needed parentheses, or puts parentheses in the wrong place.

Testing strategy

In software that computes formulae, run tests that anticipate defects such as those described in the example below.

Example

A program may compute z+(x*y), when it was supposed to compute (z+x)*y. In this case, the programmer probably wrote z+x*y and forgot that multiplication takes precedence over addition. If z is normally zero, or all three variables are normally 1, then this defect could remain hidden. Testing for errors like this therefore means thinking carefully about which values of x, y and z to use.

Use of inappropriate standard algorithms

Defect

An inappropriate standard algorithm is one that is unnecessarily inefficient or has some other property that is widely recognized as being bad.

Testing strategies

The tester has to know the properties of algorithms and design tests, such as those in the following examples that will determine whether any undesirable algorithms have been implemented.

Examples

The following are some bad choices of algorithms that testers should try to detect:
An inefficient sort algorithm. The most classical ‘bad’ choice of algorithm is sorting using a so-called ‘bubble sort’

An inefficient search algorithm. Ensure that the search time does not increase unacceptably as the list gets longer. Also check that the position in the list of the item you are looking for does not have a noticeable impact on search time.

A non-stable sort. A non-stable sort will take equal elements and sometimes switch their order after the sorting process.

A search or sort that is case sensitive when it should not be, or vice versa.
The tester should test algorithms with mixed-case data to see if the algorithm behaves as expected.

Defects in numerical algorithms

Numerical computation defects are a special class of algorithmic defect. They can occur in any software that performs mathematical calculations, especially calculations involving floating point values.

Not using enough bits or digits to store maximum values

A system does not use variables that are capable of representing the largestpossible values that could be stored. When the capacity of a data type is exceeded, an unexpected exception might be thrown, or else the data may bestored incorrectly.

Testing strategies

Test using very large numbers to ensure that the system has a wide enough margin of error.

Example

Imagine that you were going to be storing the monthly salary of an employee in a short integer (whose value ranges up to 32765).

Using insufficient places after the decimal point or too few significant figures
Defects

This problem occurs with floating point or fixed-point values. A floating-point value might not be 'wide' enough to store enough significant figures.

Testing strategies

Perform calculations that involve many significant figures, and large differences in magnitude. Verify that the calculated results are correct in such cases.

Example

Imagine an investment application that tracks a portfolio of shares. Typical shares are quoted using three or four significant digits, hence, prices might be$135.5 or $33.16. However, if a share 'crashes' in value for some reason, it might drop to only a few cents. In such a case, your system might have to record its value as $0.0344. If your system were only able to record values to two decimal places after the point, then it could not correctly manipulate such stocks.

Ordering operations poorly so that errors build up

Defects

This defect occurs when you do small operations on large floating-point numbers, and excessive rounding or truncation errors build up.

Testing strategies If a numerical application is designed to work with floating-point numbers then make sure it works with inputs that vary widely in magnitude, including both large positive and large negative exponents

Example

Imagine you are adding a large number of small credits to an account. If the account's total were in the thousands of dollars, you might always round it to the nearest dollar. However, in this situation, small transactions of a few cents would never affect the account balance.

Assuming a floating-point value will be exactly equal to some other value

Defect

If you perform an arithmetic calculation on a floating-point value, then the result will very rarely be computed exactly. It is therefore a defect to test if floating-point value is exactly equal to some other value. You should instead test if it is within a small range around that value.

Testing strategies

Standard boundary testing should detect this type of defect.

Example

The following is at risk of resulting in an infinite loop, since d may never equal precisely 10.0, but may instead equal 9.99999999999 after 10 iterations.

```
for (double d = 0.0; d != 10.0; d+=2.0) {...}
```

The correct expression should have been:

```
for (double d = 0.0; d < 10.0; d+=2.0) {...}
```

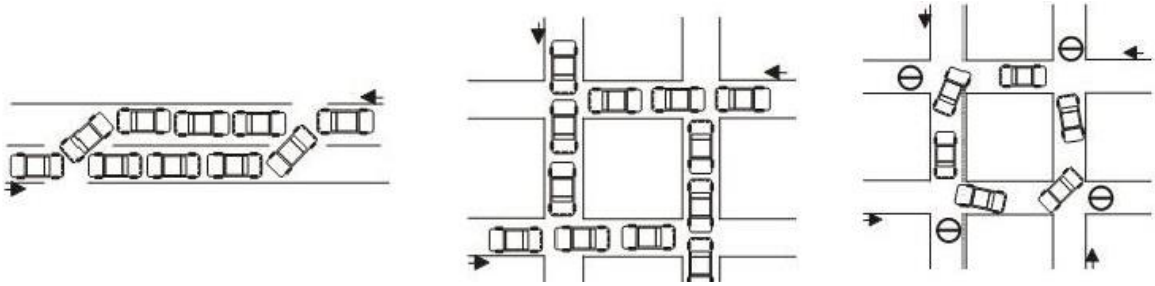
Defects in timing and co-ordination: deadlocks, livelocks and critical races

The three most important kinds of timing and co-ordination defects aredeadlocks, livelocksand critical races.

Deadlock and livelock

Defects

A deadlock is a situation where two or more threads or processes are stopped, waiting for each other to do something before either can proceed. Since neither can do anything, they permanently stop each other from proceeding. A classic example of real-life deadlock is the 'gridlock' sometimes encountered in busy cities.



Testing strategies

Deadlocks and livelocks tend to occur as a result of unusual combinations of conditions that are hard to anticipate or reproduce. It is often most effective to use*inspection* to detect such defects, rather than testing alone. If black-box testing is the only possibility, then you can try some of the following tactics:

Vary the time consumption of different threads by giving them differing amounts of input, or running them on hardware that varies in speed.

Run a large number of threads concurrently.

Deliberately deny resources to one or more threads (e.g. temporarily cut a network connection, or make a file unreadable).

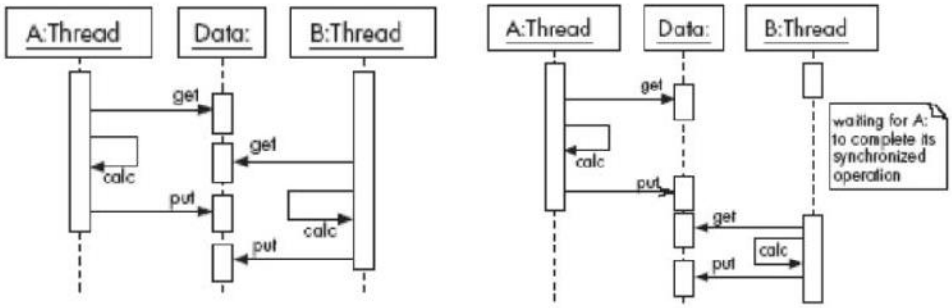
Critical races

Defects

A critical race is a defect in which one thread or process can sometimes experience a failure because another thread or process interferes with the 'normal' sequence of events. The defect is not that the other thread tries to do something, but that the system allows interference to occur.

One type of critical race occurs when two processes or threads normally work together to achieve some outcome; however, if one is sped up or slowed down then the outcome is incorrect.

A second type of critical race occurs when one thread unexpectedly changes data that is being operated on by another thread, resulting in incorrect results.



a) Abnormal: The value put by thread A is immediately overwritten by the value put by thread B.

b) The problem has been solved by accessing the data using synchronized methods

Designers can prevent critical races by using various mechanisms that allow data items to be locked so that they cannot be accessed by other threads when they are not ready. One widely used locking mechanism is called a *semaphore*.

Testing strategies

Testing for critical races is done using the same strategies as testing for deadlocks and livelocks. Once again, inspection is often a better solution. One possible, although invasive, strategy is to deliberately slow down one of the threads by adding a call to the sleep method.

Managing the software process

What is project management?

Project management encompasses all the activities needed to plan and execute a project. The following are specific activities often done by a project manager:

Deciding what needs to be done

Finding customers; working with customers to determine their problem and the scope of the project; prioritizing the work; selecting the overall processes that will be followed, and negotiating contracts.

Estimating costs

The most important aspect of this is estimating the amount of elapsed time and effort that will be required to complete the project.

Ensuring there are suitable people to undertake the project

This includesfinding people, and ensuring that people have appropriate training. It can also include firing people who are not performing adequately.

Defining responsibilities

Determining how people will work together in teams and who will be responsible for what

Scheduling

Determining the sequence of tasks, plus setting deadlines for when tasks must be complete. Major deadlines are called *milestones*.

Making arrangements for the work

Initiating the paperwork involved in hiring or subcontracting; setting up training courses; finding office space; ensuring that hardware and software is available; ensuring that people have the requisite security clearance, etc.

Directing

Telling subordinates and contractors what to do. Many of the other activities in this list involve making decisions; but acting on those decisions by ordering people to do things is a distinct activity. Directing is not as simple as issuing orders – you have to get people to commit to deliver what they promise.

Being a technical leader

Giving advice about engineering problems; helping people solve problems by leading discussions; pointing people to appropriate sources of information; acting as a mentor, and making high-level decisions about requirements and design.

Reviewing and approving decisions made by others

In certain types of projects, the project manager will have to take the ultimate legal responsibility for declaring that proper engineering practice has been followed, and that the manager believes the resulting system will be safe. However, a certain amount of reviewing and approving is a part of every project.

Building morale and supporting staff

Helping resolve interpersonalconflicts; ensuring that people feel rewarded, respected and motivated; giving people feedback to help them improve their work; and ensuring that people always have somebody to talk to about problems.

Monitoring and controlling

Finding out what is going on, determining how the plans need to change, and taking action to keep the project on track.

Coordinating the work with managers of other projects

Reporting

Telling customers and higher-level managers what they need and want to know.

Continually striving to improve the process.

Software process models

Software process models are general approaches for organizing a project into activities. The models should be seen as *aids to thinking.*

In the opportunistic approach, developers keep on modifying the software until they or their users are satisfied. This approach has several important problems:

The system might satisfy certain user needs, but reaching a high level of user satisfaction will require many changes.

The design of software deteriorates faster if it is not well designed.

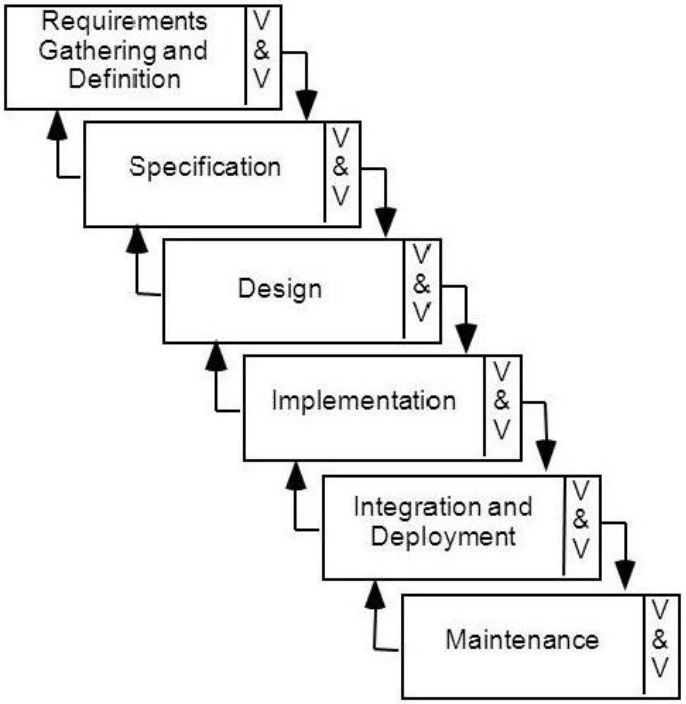
Since there are no plans, there is nothing to aim towards. Since there is nothing to aim towards, you can never know if you are doing well or poorly. Therefore there is no *control* of costs or schedule in an opportunistic project.

Many undetected defects therefore remain, giving rise to never-ending changes that make the system worse and worse.

The above problems make the cost of developing and maintaining software very high.

The waterfall model

The *waterfall* model is a significant improvement over the opportunistic approach. It is a classic way of looking at software engineering that accounts for the importance of requirements, design and quality assurance. The model is so named because it tend to look like cascading waterfalls.



It has some limitations and, if followed too strictly, can lead to the following types of problems:

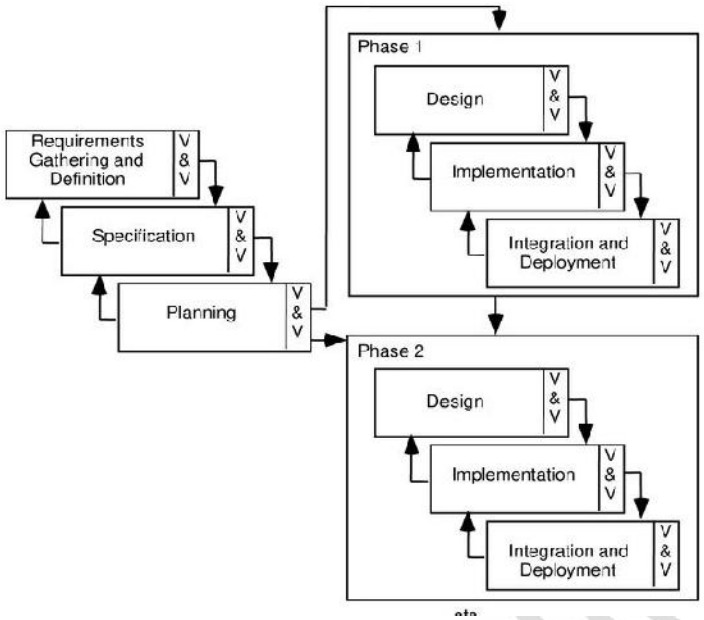
The model implies that you should attempt to complete the entire specificationbefore moving on to the design, and the entire design before moving on to implementation.

The waterfall model makes no allowances for prototyping and implies that you can get the requirements right by simply writing them down and reviewing them.

The model implies that once the product is finished, everything else is maintenance.

The phased-release model

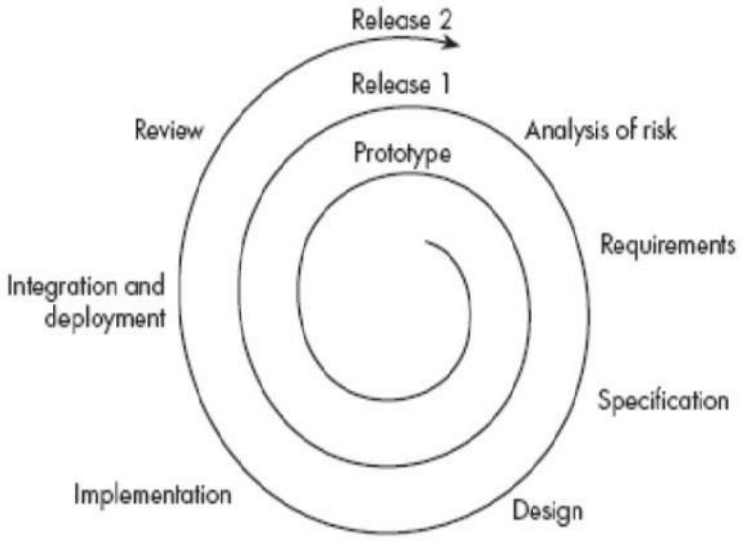
The phased-release model of software development rectifies some, but not all, of the problems of the waterfall model. The most important change is that it introduces the notion of incremental development.



The spiral model

The *spiral model*, as shown in Figure 11.4, is another view of incremental development that explicitly embraces prototyping and an *iterative* approach to software development. This model takes the position that you should start to develop software by developing a small prototype (innermost loop of the spiral).

This first prototype follows a mini-waterfall process, but is very quickly developed and serves primarily to gather requirements.



The spiral model also adds the notion of *risk analysis* to process modeling. When following the spiral model, a project undergoes a large number of cycles. The cycling only ends when the system is finally retired.

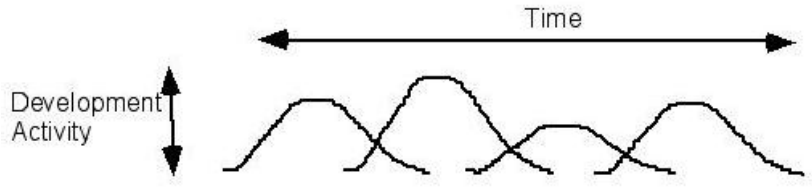
The evolutionary model

The *evolutionary model* (Figure 11.5) shows software development as a series ofhills, each representing a separate loop of the spiral. This is a third way of thinking about incremental development. This model shows two things that are not always clear from the spiral model.

First, it shows that loops, or releases, tend to overlap each other. As testing andpreparations for deployment of one release are under way, planning for the next release has already started.

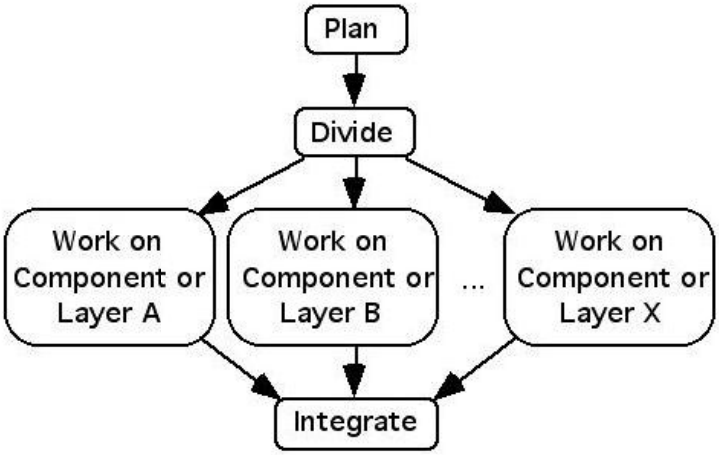
Secondly, the evolutionary model makes it clear that development work tends to reach a peak, at around the time of the deadline for completion of implementation.

Finally, the model shows that each prototype or release can take diff rent amounts of time to deliver, and can take differing amounts of effort.



The concurrent engineering model

The *concurrent engineering model* (Figure 11.6) explicitly accounts for the divide and conquer principle. Each team works on its own component, typically following a spiral or evolutionary approach.



Choosing a process model

When planning a particular project, the important thing to recognize is that you can combine the features of the models that apply best to your current project.

From the **waterfall model**, you will always incorporate the notion of stages,From the **phased-release model**, you can incorporate the notion of doing some initial high-level analysis, and then dividing the project into releases.

You can also incorporate the notions of prototyping and risk analysis from the **spiral model**.Next, from the **evolutionary model** you can incorporate the notion of varying amounts of time and work, with overlapping releases.

And from the **concurrent engineering model**, you can break the system down into components and develop them in parallel.

Re-engineering

No matter what process model you use, in any large or long-lived project, the design will deteriorate. Periodically, therefore, project managers should set aside some time to re-engineer part or all of the system. The extent of this work can vary considerably: it could include cleaning up the code to make it more readable, completely replacing a layer, or refactoring part of the design.

Cost estimation

To estimate how much software-engineering time will be required to do some work.

- Elapsed time
  The difference in time from the start date to the end date of a task or project
- Development effort
  The amount of labour used in person-months or person-days.
  To convert an estimate of development effort to an amount of money:
- You multiply it by the weighted average cost (burdened cost) of employing a software engineer for a month (or a day).

Principles of effective cost estimation

Principle 1:

Divide and conquer.

- To make a better estimate, you should divide the project up into individual subsystems.
- Then divide each subsystem further into the activities that will be required to develop it.
- Next, you make a series of detailed estimates for each individual activity.

And sum the results to arrive at the grand total estimate for the project

Principle 2:

Include all activities when making estimates.

The time required for *all* development activities must be taken into account.

Including:

- Prototyping
- Design
- Inspecting
- Testing
- Debugging
- Writing user documentation
- Deployment.

Principle 3:

Base your estimates on past experience combined with knowledge of the current project.

- If you are developing a project that has many similarities with a past project: You can expect it to take a similar amount of work.
- Base your estimates on the *personal judgement* of your experts

or

- Use *algorithmic models* developed in the software industry as a whole by analyzing a wide range of projects.

They take into account various aspects of a project's size and complexity, and provide formulas to compute anticipated cost.

Algorithmic models

Based on an estimate of some other factor that you can measure, or that is easier to estimate:

- The number of use cases
- The number of distinct requirements
- The number of classes in the domain model
- The number of widgets in the prototype user interface
- An estimate of the number of lines of code

Algorithmic models

A typical algorithmic model uses a formula like the following:

- COCOMO:
- Functions Points:
- $S = W_1F_1 + W_2F_2 + W_3F_3 + \ldots$

Principle 4:

Be sure to account for *differences* when extrapolating from other projects.

- Different software developers
- Different development processes and maturity levels
- Different types of customers and users
- Different schedule demands

- Different technology
- Different technical complexity of the requirements
- Different domains
- Different levels of requirement stability

Principle 5:
Anticipate the worst case and plan for contingencies.

- Develop the most critical use cases first

  If the project runs into difficulty, then the critical features are more likely to have been completed
- Make three estimates:

  Optimistic (O)
  - Imagining a everything going perfectly

  Likely (L)
  - Allowing for typical things going wrong

  Pessimistic (P)
  - Accounting for everything that could go wring

Principle 6:
Combine multiple independent estimates.

- Use several different techniques and compare the results.
- If there are discrepancies, analyse your calculations to discover what factors causing the differences.
- Use the Delphi technique.
  - Several individuals initially make cost estimates in private.
  - They then share their estimates to discover the discrepancies.
  - Each individual repeatedly adjusts his or her estimates until a consensus is reached.

Principle 7:
Revise and refine estimates as work progresses

- As you add detail.
- As the requirements change.
- As the risk management process uncovers problems.

Building software engineering teams

Software engineering is a human process. Choosing appropriate people for a team, and assigning roles and responsibilities to the team members, is therefore an important project management skill.

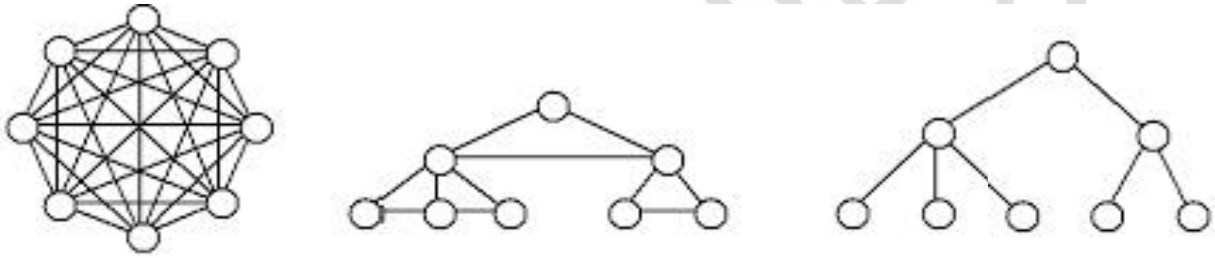
Strict hierarchy versus more flexible arrangements

Software engineering teams can be organized in many different ways. One approach is to use a hierarchical manager–subordinate structure.

In the *egoless approach*, decisions are made by consensus; this can lead to more creative solutions, since group members spontaneously get together to solve problems when they arise. In general, the egoless approach is most suited to difficult projects with many technical challenges.

The *hierarchical approach* is reminiscent of the military or large bureau critic organizations. It is suitable for large projects with a strict schedule and whereeverybody is well trained and has a well-defined role.

The '*chief programmer team*' is a model that is midway between egoless andhierarchical. It works very much like the surgical team in an operating room.



a) Egoless b) Chief programmer c) Strict hierarchy

Choosing an effective size for a team

For a given project or project iteration, the number of people on a team will not be constant. Initially, just a few people will be involved in defining the scope;later on, additional people will become involved as requirements, design,implementation and testing get under way. As the project or iteration nears completion, people will start moving on to the next iteration or to other work, leaving only a few people to undertake deployment.

It is important to remember the following rule, however: if your team has an appropriate number of people to start with, then you cannot generally add people if you get behind schedule, in the hope of catching up.

Skills needed on a team

No matter how the team is organized, individual people are often assigned specific roles based on their particular skills. The following are some of the more common roles found on a development team:

Architect

This person is responsible for leading the decision making about the architecture, and maintaining the architectural model.

Project manager

Responsible for doing the project management tasks described in this chapter. Even in an egoless team, somebody has to be the custodian of the cost estimates and the schedule.

Configuration management and build specialist

This person ensures that, as changes are made, no new problems are introduced. Everyone relies on builds as the baselines for quality assurance and subsequent development. This person also makes sure that documentation for each change is properly updated.

User interface specialist

Although everybody should interact with users, this person has the particular responsibility to make sure that usability is kept at forefront of the design process.

Technology specialist

Such a person has specialized knowledge and expertise in a technology such as databases, networking, operating systems etc.

Hardware and third-party software specialist

This person makes sure that the development team has appropriate types of hardware and third-party software on which to develop and test the software. This person will install and perform acceptance testing on any reusable components the team plans to use.

User documentation specialist

This person, who should have a technical writing background, ensures that online help and user manuals are well written.

Tester

Even though there should be an independent test group, the development group may have a person who is responsible for the first stage of testing.

Project scheduling and tracking

Scheduling is the process of deciding in what sequence a set of activities will be performed, as well as when they should start and be completed. Tracking is the process of determining how well you are sticking to the cost estimate and schedule.

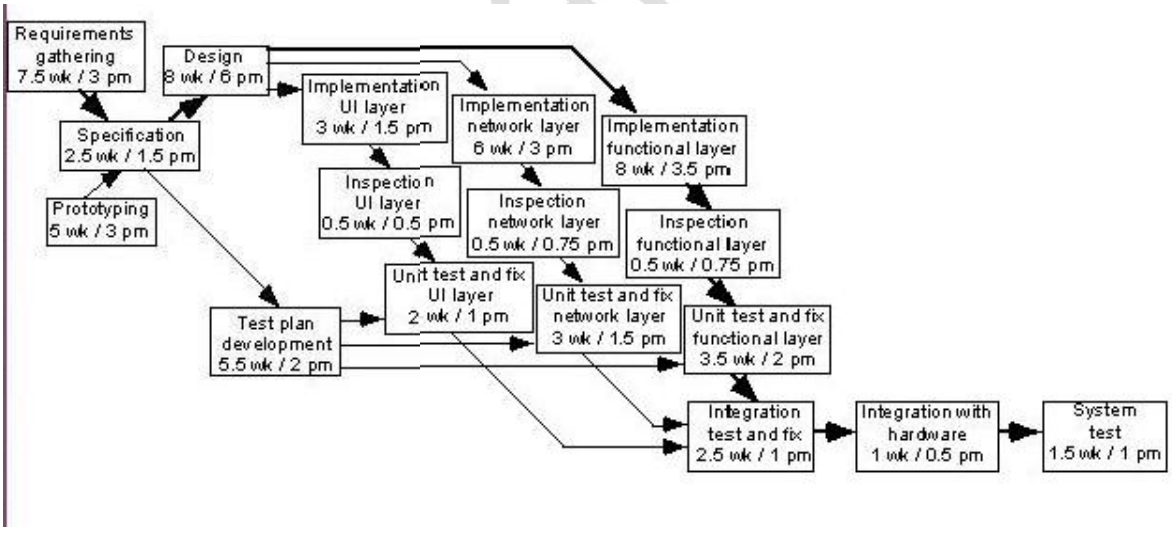
Two types of diagram are particularly important in scheduling: ***PERT charts*** **and** ***Gantt charts*****.** ***Earned value charts*** are useful for tracking.

PERT charts

A PERT chart shows the sequence in which tasks must be completed. Each task has zero or more predecessors on which it depends, and zero or more successors, which depend on it. The whole diagram therefore forms a graph, whose nodes are tasks, and whose arcs are dependencies.

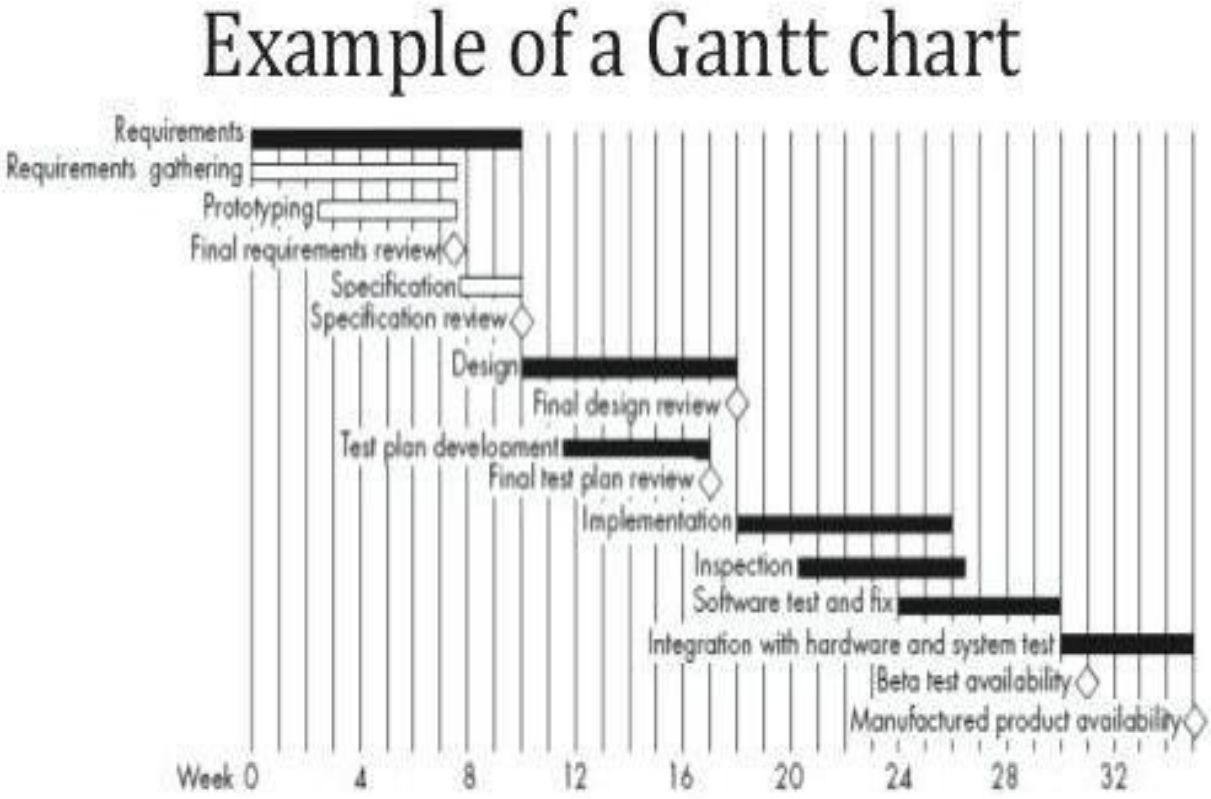
In each node of a PERT chart, you typically show the elapsed time and effort estimates. You can also show optimistic, likely and pessimistic estimates.

One of the most important uses of a PERT chart is to determine the *critical path*. The critical path indicates the minimum time in which it is possible to complete the project.



Gantt charts

A Gantt chart is used to graphically present the start and end dates of each software engineering task. A Gantt chart is like a UML sequence chart: one axis shows time and the other axis shows the activities that will be performed.



Earned value charts

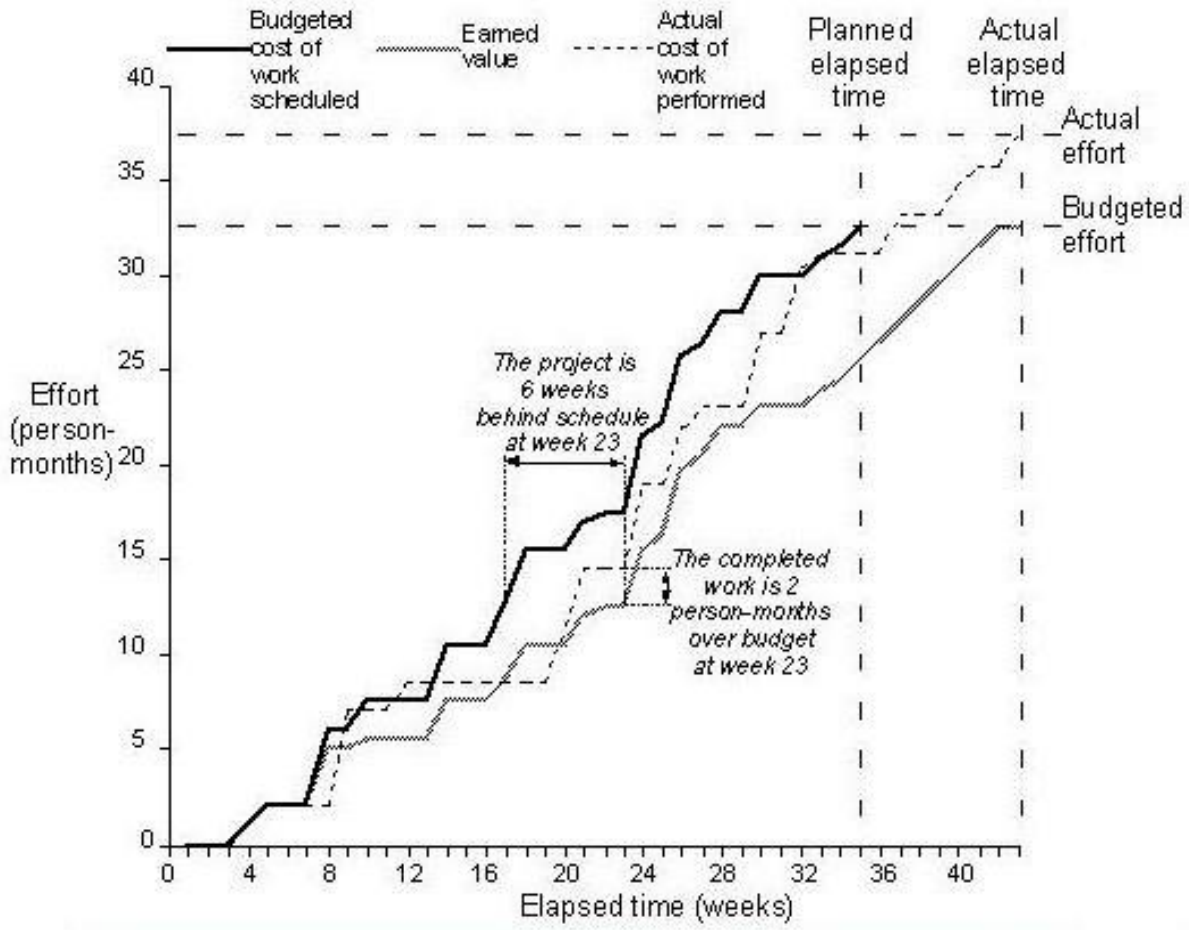
Earned value is the amount of work completed, measured according to the *budgeted*effort that the work was supposed to consume. It is also called the *budgeted cost of work performed*. As each task is completed, the number of person-months originally planned for that task is added to the earned value of the project.

An earned value chart has **three curves**:

The budgeted cost of work scheduled. This is the planned amount of effort that was supposed to have been expended by any point in time. It is computed by examining at the cost estimates and the Gantt chart. It is shown here as the solid black curve.

The earned value – that is, the budgeted cost of the work performed. This is shown here as the dotted curve.

The actual cost of the work performed so far. This is shown as a dashedcurve.


Budgeted cost of work scheduled
Earned value
Actual cost of work performed
Planned elapsed time
Actual elapsed time
Actual effort
Budgeted effort
The project is 6 weeks behind schedule at week 23
The completed work is 2 person-months over budget at week 23
Effort (person-months)
Elapsed time (weeks)
0
5
10
15
20
25
30
35
40
0
4
8
12
16
20
24
28
32
36
40