UNIT-III

SOFTWARE DESIGN

Software Design: Overview of the design process, How to characterize a good software design? Layered arrangement of modules, Cohesion and Coupling. Approaches to software design.
Agility: Agility and the Cost of Change, Agile Process, Extreme Programming (XP), Other Agile Process Models, Tool Set for the Agile Process.

Function-Oriented Software Design: Overview of SA/SD Methodology, Structured Analysis, Structured Design, Detailed Design, Design Review.
User Interface Design: Characteristics of Good User Interface, Basic Concepts, Types of user interfaces, Fundamentals of component-based GUI development, and user interface design methodology.

Introduction: During the software design phase, the design document is produced, basedon the customer requirements as documented in the SRS(Software Requirements Specification)document. We can state the main objectives of the design phase, in other words, as follows.

This view of a design process has been shown schematically in Figure 3 .1. As shown in Figure 3.1, the design process starts using the SRS document andcompletes with the production of the design document. The design document produced at the end of the design phase should be implementable using a programming language in the subsequent (coding) phase.

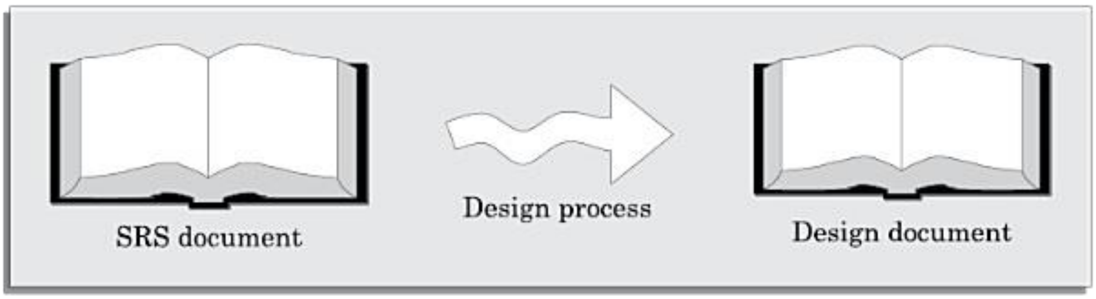


Figure 3.1: The design process.

3.1 OVERVIEW OF THE DESIGN PROCESS

The design process essentially transforms the SRS document into a design document. In the following sections and subsections, we will discuss a few important issues associated with the design process.

3.1.1Outcome of the Design Process

The following items are designed and documented during the design phase.

- **Different modules required:** The different modules in the solution should be clearly identified. Each module is a collection of functions and the data shared by the functions of the module. Each module should accomplish some well-defined task out of the overall responsibility of the software. Each module should be named according to the task it performs. For example, inacademic automation software, the module consisting of the functions and data necessary to accomplish the task of registration of the students should be named handle student registration.

- **Control relationships among modules:** A control relationship between two modules essentially arises due to function calls across the two modules. The control relationships existing among various modules should be identified in the design document.
- **Interfaces among different modules:** The interfaces between two modules identify the exact data items that are exchanged between the two modules when one module invokes a function of the other module.
- **Data structures of the individual modules:** Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module. Suitable data structures for storing and managing the data of a module need to be properly designed and documented.
- **Algorithms required implementing the individual modules:** Each function in a module usually performs some processing activity. The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due considerations given to the accuracy of the results, space and time complexities.

Starting with the SRS document (as shown in Figure 3.1), the designdocuments are produced through iterations over a series of steps that we are going to discuss in this chapter and the subsequent three chapters. The design documents are reviewed by the members of the development team to ensure that the design solution conforms to the requirements specification.

3.1.1 Classification of Design Activities

A good software design is seldom realized by using a single step procedure, rather it requires iterating over a series of steps called the design activities. Let us first classify the design activities before discussing them in detail. Depending on the order in which various design activities are performed, we can broadly classify them into two important stages.

- Preliminary (or high-level) design, and
- Detailed design.

The meaning and scope of these two stages can vary considerably from one design methodology to another. However, for the traditional function-oriented design approach, it is possible to define the objectives of the high-level design as follows:

The outcome of high-level design is called the program structure or the software architecture. High-level design is a crucial step in the overall design of software. When the high-level design is complete, the problem should have been decomposed into many small functionally independent modules that are cohesive, have low coupling among them and are arranged in a hierarchy. Many different types of notations have been used to represent a high-level design. A notation that is widely being used for procedural development is a tree-like diagram called the structure chart. Another popular design representation techniques called UML that is being used to document object-oriented design, involves developing several types of diagrams to document the object-oriented design of a systems. Though other notations such as Jackson diagram [1975] or Warnier-Orr [1977, 1981] diagram are available to document a software design, we confine our attention in this text to structure charts and UML diagrams only.

Once the high-level design is complete, detailed design is undertaken.

The outcome of the detailed design stage is usually documented in the form of a module specification (MSPEC) document. After the high-level design is complete, the problem would have been decomposed into small modules, and the data structures and algorithms to be used described using MSPEC and can be easily grasped by programmers for initiating coding. In this text, we do not discuss MSPECs and confine our attention to high-level design only.

3.1.2 Classification of Design Methodologies

The design activities vary considerably based on the specific design methodology being used. A large number of software design methodologies are available. We can roughly classify these methodologies into procedural and object-oriented approaches. These two approaches are two fundamentally different design paradigms. In this chapter, we shall discuss the important characteristics of these two fundamental design approaches. Over the next three chapters, we shall study these two approaches in detail.

Do design techniques result in unique solutions?

Even while using the same design methodology, different designers usually arrive at very different design solutions. The reason is that a design technique often requires the designer to make many subjective decisions and work out compromises to contradictory objectives. As a result, it is possible that even the same designer can work out many different solutions to the same problem. Therefore, obtaining a good design would involve trying out several alternatives (or candidate solutions) and picking out the best one. However, a fundamental question that arises at this point is—how to distinguish superior design solution from an inferior one? Unless we know what a good software design is and how to distinguish a superior design solution from an inferior one, we cannot possibly design one. We investigate this issue in the next section.

Analysis versus design

Analysis and design activities differ in goal and scope.

- The analysis results are generic and do not consider implementation or the issues associated with specific platforms. The analysis model is usually documented using some graphical formalism. In case of the function-oriented approach that we are going to discuss, the analysis model would be documented using data flow diagrams (DFDs), whereas the design would be documented using structure chart. On the other hand, for object-oriented approach, both the design model and the analysis model will be documented using unified modeling language (UML). The analysis model would normallybe very difficult to implement using a programming language.
- The design model is obtained from the analysis model through transformations over a series of steps. In contrast to the analysis model, the design model reflects several decisions taken regarding the exact way system is to be implemented. The design model should be detailed enough to be easily implementable using a programming language.

3.2 HOW TO CHARACTERISE A GOOD SOFTWARE DESIGN?

Coming up with an accurate characterization of a good software design that would hold across diverse problem domains is certainly not easy. In fact, the definition of a "good" software design can vary depending on the exact application being designed. For example, "memory size used up by a program" may be an important issue to characterize a good solution for embedded software development—since embedded applications are often required to work under severely limited memory sizes due to cost, space, or power consumption considerations. For embedded applications, factors such as design comprehensibility may take a back seat while judging the goodness of design. Thus for embedded applications, one may sacrifice design comprehensibility to achieve code compactness. Similarly, it is not usually true that a criterion that is crucial for some application needs to be almost completely ignored for another application. It is therefore clear that the criteria used to judge a design solution can vary widely across different types of applications. Not only do the criteria used to judge a design solution depend on the exact application being designed, but to makethe matter worse, there is no general agreement among software engineers and researchers on the exact criteria to use for judging a design even for a specific category of application. However, most researchers and software engineers agree on a few desirable characteristics that every good software design for general applications must possess. These characteristics are listed below:

- **Correctness:** A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.
- **Understandability:** A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.
- **Efficiency:** A good design solution should adequately address resource,time, and cost optimization issues.
- **Maintainability:** A good design should be easy to change. This is an important requirement, since change requests usually keep coming from the customer even after product release.

3.2.1 Understandability of a Design: A Major Concern

While performing the design of a certain problem, assume that we have arrived at a large number of design solutions and need to choose the best one. Obviously all incorrect designs have to be discarded first. Out of the correct design solutions, how can we identify the best one? the human cognitive limitations that arise due to limited short-term memory. A large problem overwhelms the human mind, and a poor design would make the matter worse. Unless a design solution is easily understandable, it could lead to an implementation having a large number of defects and at the same time tremendously pushing up the development costs. Therefore, a good design solution should be simple and easily understandable. A design that is easy to understand is also easy to develop and maintain. A complex design would lead to severely increased life cycle costs. Unless a design is easily understandable, it would require tremendous effort to implement, test, debug, and maintain it. We had already pointed out in Chapter 2 that about 60 per cent of the total effort in the life cycle of a typical product is spent on maintenance. If the software is not easy to understand, not only would it lead to increased development costs, the effort required to maintain the product would also increase manifold. Besides, a design solution that is difficult to understand would lead to a program that is full of bugs and is unreliable. Recollect that we had already discussed in Chapter 1 that understandability of a design solution can be enhanced through clever applications of the principles of abstraction and decomposition.

An understandable design is modular and layered

How can the understandability of two different designs be compared, so that we can pick the better one? To be able to compare the understandability of two design solutions, we should at least have an understanding of the general features that an easily understandable design should possess. A design solution should have the following characteristics to be easily understandable. It should assign consistent and meaningful names to various design components. It should make use of the principles of decomposition and abstractionin good measures to simplify the design.
These two principles are exploited by design methodologies to make a design modular and layered. (Though there are also a few other forms in which the abstraction and decomposition principles can be used in the design solution, we discuss those later). We can now define the characteristics of an easily understandable design as follows: A design solution is understandable, if it is modular and the modules are arranged in distinct layers.

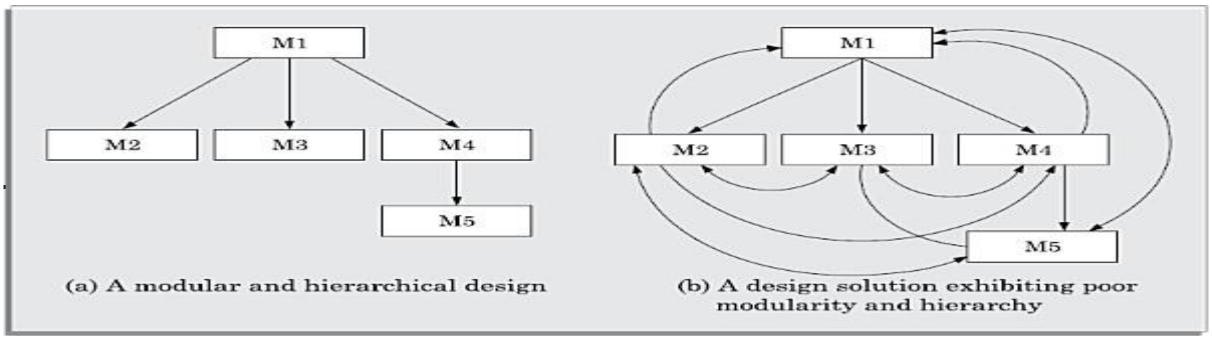
Modularity

A modular design is an effective decomposition of a problem. It is a basic characteristic of any good design solution. A modular design, in simple words, implies that the problem has been decomposed into a set of modules that have only limited interactions with each other. Decomposition of a problem into modules facilitates taking advantage of the divide and conquer principle. If different modules have either no interactions or little interactions with each other, then each module can be understood separately. This reduces the perceived complexity of the design solution greatly. To understand why this is so, remember that it may be very difficult to break a bunch of sticks which have been tied together, but very easy to break the sticks individually.

It is not difficult to argue that modularity is an important characteristic of a good design solution. But, even with this, how can we compare the modularity of two alternate design solutions? From an inspection of the module structure, it is at least possible to intuitively form an idea as to which design is more modular For example, consider two alternate design solutions to a problem that are represented in Figure 3.2, in which the modules M1 , M2 etc. have been drawn as rectangles. The invocation of a module by another module has been shown as an arrow. It can easily be seen that the design solution of Figure 3.2(a) would be easier to understand since the interactions among the different modules is low. But, can we quantitatively measure the modularity of a design solution? Unless we are able to quantitatively measure the modularity of a design solution, it will be hard to say which design solution is more modular than another. Unfortunately, there are no quantitative metrics available yet to directly measure the modularity of a design. However, we can quantitatively characterize the modularity of a design solution based on the cohesion and coupling existing in the design.

A software design with high cohesion and low coupling among modules is the effective problem decomposition we discussed in Chapter 1. Such a design would lead to increased productivity during program development by bringing down the perceived problem complexity.

Figure 3.2: Two design solutions to the same problem.


M1
M2
M3
M4
M5
(a) A modular and hierarchical design
M1
M2
M3
M4
M5
(b) A design solution exhibiting poor modularity and hierarchy

Based on this classification, we would be able to easily judge the cohesion and coupling existing in a design solution. From knowledge of the cohesion and coupling in a design, we can form our own opinion about the modularity of the design solution. We shall define the concepts of cohesion and coupling and the various classes of cohesion and coupling in Section 3.3. Let us now discuss the other important characteristic of a good design solution—layered design.

Layered design

A layered design is one in which when the call relations among different modules are represented graphically, it would result in a tree-like diagram with clear layering. In a layered design solution, the modules are arranged in a hierarchy of layers. A module can only invoke functions of the modules in the layer immediately below it. The higher layer modules can be considered to be similar to managers that invoke (order) the lower layer modules to get certain tasks done. A layered design can be considered to be implementing control abstraction, sincea module at a lower layer is unaware of (about how to call) the higher layer modules.

A layered design can make the design solution easily understandable, since to understand the working of a module, one would at best have to understand how the immediately lower layer modules work without having to worry about the functioning of the upper layer modules.

When a failure is detected while executing a module, it is obvious that the modules below it can possibly be the source of the error. This greatly simplifies debugging since one would need to concentrate only on a few modules to detect the error. We shall elaborate these concepts governing layered design of modules in Section 3.4.

3.3 COHESION AND COUPLING

We have so far discussed that effective problem decomposition is an important characteristic of a good design. Good module decomposition is indicated through high cohesion of the individual modules and low coupling of the modules with each other. Let us now define what is meant by cohesion and coupling.

Cohesion is a measure of the functional strength of a module, whereas the coupling between two modules is a measure of the degree of interaction (or interdependence) between the two modules.

Coupling: Intuitively, we can think of coupling as follows. Two modules are said to be highly coupled, if either of the following two situations arise:

If the function calls between two modules involve passing large chunks of shared data, the modules are tightly coupled.

If the interactions occur through some shared data, then also we saythat they are highly coupled.

If two modules either do not interact with each other at all or at best interact by passing no data or only a few primitive data items, they are said to have low coupling.

Cohesion: To understand cohesion, let us first understand an analogy. Suppose you listened to a talk by some speaker. You would call the speech to be cohesive, if all the sentences of the speech played some role in giving the talk a single and focused theme. Now, we can extend this to a module in a design solution. When the functions of the module co-operate with each other for performing a single objective, then the module has good cohesion. If the functions of the module do very different things and do not co-operate with each other to perform a single piece of work, then the module has very poor cohesion.

Functional independence

By the term functional independence, we mean that a module performs a single task and needs very little interaction with other modules.

Functional independence is a key to any good design primarily due to the following advantages it offers:

Error isolation: Whenever an error exists in a module, functional independence reduces the chances of the error propagating to the other modules. The reason behind this is that if a module is functionally independent, its interaction with other modules is low. Therefore, an error existing in the module is very unlikely to affect the functioning of other modules.

Further, once a failure is detected, error isolation makes it very easy to locate the error. On the other hand, when a module is not functionally independent, once a failure is detected in a functionality provided by the module, the error can be potentially in any of the large number of modules and propagated to the functioning of the module.

Scope of reuse: Reuse of a module for the development of other applications becomes easier. The reasons for this is as follows. A functionally independent module performs some well-defined and precise task and the interfaces of the module with other modules are very few and simple. A functionally independent module can therefore be easily taken out and reused in a different program. On the other hand, if a module interacts with several other modules or the functions of a module perform very different tasks, then it would be difficult to reuse it. This is especially so, if the module accesses the data (or code) internal to other modules.

Understandability: When modules are functionally independent, complexity of the design is greatly reduced. This is because of the fact that different modules can be understood in isolation, since the modules are independent of each other. We have already pointed out in Section 3.2 that understandability is a major advantage of a modular design. Besides the three we have listed here, there are many other advantages of a modular design as well. We shall not list those here, and leave it as an assignment to the reader to identify them.

3.1.2 Classification of Cohesiveness

Cohesiveness of a module is the degree to which the different functions of the module co-operate to work towards a single objective. The different modules of a design can possess different degrees of freedom. However, the different classes of cohesion that modules can possess are depicted in Figure 3.3. The cohesiveness increases from coincidental to functional cohesion. That is, coincidental is the worst type of cohesion and functional is the best cohesion possible. These different classes of cohesion are elaborated below.

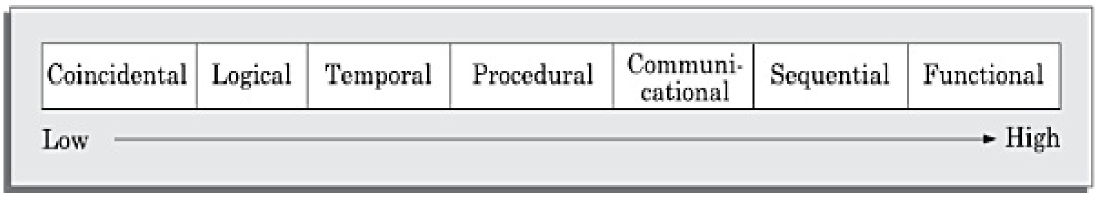


Figure 3.3: Classification of cohesion.

Coincidental cohesion: A module is said to have coincidental cohesion,if it performs a set of tasks that relate to each other very loosely, if atall. In this case, we can say that the module contains a random collection of functions. It is likely that the functions have been placed in the module out of pure coincidence rather than through some thought or design. The designs made by novice programmers often possess this category of cohesion, since they often bundle functions to modules rather arbitrarily. An example of a module with coincidental cohesion has been shown in Figure 3.4(a).Observe that the different functions of the module carry out very different and unrelated activities starting from issuing of library books to creating library member records on one hand, and handling librarian leave request on the other.

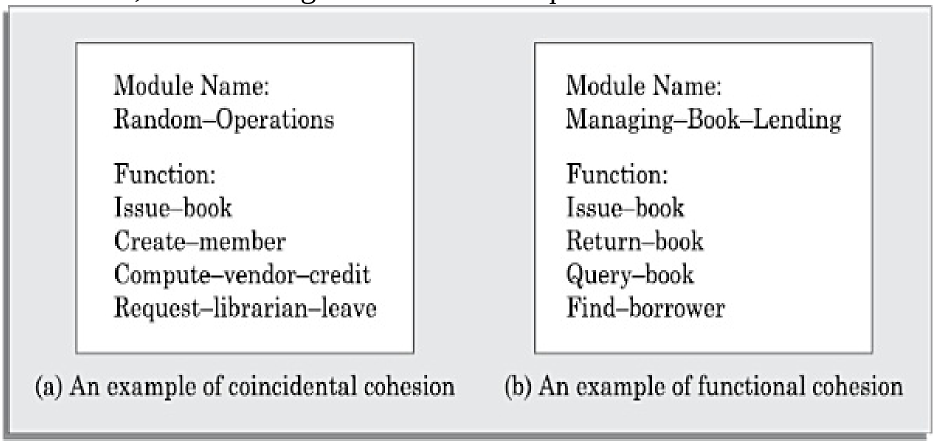


Figure 3.4: Examples of cohesion.

Logical cohesion: A module is said to be logically cohesive, if all elements of the module perform similar operations, such as error handling, data input, data output, etc. As an example of logical cohesion, consider a module that contains a set of print functions to generate various types of output reports such as grade sheets, salary slips, annual reports, etc.

Temporal cohesion: When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion. As an example, consider the following situation. When a computer is booted, several functions need to be performed. These include initialization of memory and devices, loading the operating system, etc. When a single module performs all these tasks, then the module can be said to exhibit temporal cohesion. Other examples of modules having temporal cohesion are the following. Similarly, a module would exhibit temporal cohesion, if it comprises functions for performing initialization, or start-up, or shut-down of some process.

Procedural cohesion: A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data. Consider the activities associated with order processing in a trading house. The functions login(), place-order(), check-order(), print- bill(), place-order-on-vendor(), update-inventory(), and logout() all do different thing and operate on different data. However, they are normally executed one after the other during typical order processing by a sales clerk.

Communicational cohesion: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure. As an example of procedural cohesion, consider a module named student in which the different functions in the module such as admitStudent, enterMarks, printGradeSheet, etc. access and manipulate data stored in an array named studentRecords defined within the module.

Sequential cohesion: A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence. As an example consider the following situation. In an on-line store consider that after a customer requests for some item, it is first determined if the item is in stock. In this case, if the functions create-order(), check-item-availability(), place- order-on-vendor() are placed in a single module, then the module would exhibit sequential cohesion. Observe that the function create-order() creates an order that is processed by the function check-item-availability() (whether the items are available in the required quantities in the inventory) is input to place-order-on-vendor().

Functional cohesion: A module is said to possess functional cohesion, if different functions of the module co-operate to complete a single task. For example, a module containing all the functions required to manage employees' pay-roll displays functional cohesion. In this case, all the functions of the module (e.g., computeOvertime(), computeWorkHours(), computeDeductions(), etc.) work together to generate the payslips of the employees. Another example of a module possessing functional cohesion has been shown in Figure 3.4(b). In this example, the functions issue-book(), return-book(), query-book(), and find-borrower(), together manage all activities concerned with book lending. When a module possesses functional cohesion, then we should be able to describe what the module does using only one simple sentence. For example, for the module of Figure 3.4(a), we can describe the overall responsibility of the module by saying "It manages the book lending procedure of the library."

A simple way to determine the cohesiveness of any given module is as follows. First examine what do the functions of the module perform. Then, try to write down a sentence to describe the overall work performed by the module. If you need a compound sentence to describe the functionality of the module, then it has sequential or communicational cohesion. If you need words such as "first", "next", "after", "then", etc., then it possesses sequential or temporal cohesion. If it needs words such as "initialise", "setup", "shut down", etc., to define its functionality, then it has temporal cohesion.

We can now make the following observation. A cohesive module is one in which the functions interact among themselves heavily to achieve a single goal. As a result, if any of these functions is removed to a different module, the coupling would increase as the functions would now interact across two different modules.

3.1.2 Classification of Coupling

The coupling between two modules indicates the degree of interdependence between them. Intuitively, if two modules interchange large amounts of data, then they are highly interdependent or coupled. We can alternately state this concept as follows.

The interface complexity is determined based on the number of parameters and the complexity of the parameters that are interchanged while one module invokes the functions of the other module.

Let us now classify the different types of coupling that can exist between two modules. Between any two interacting modules, any of the following five different types of coupling can exist. These different types of coupling, in increasing order of their severities have also been shown in Figure 3.5.

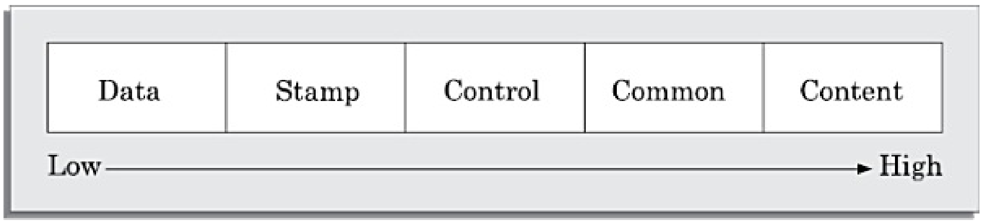


Figure 3.5: Classification of coupling.

Data coupling: Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for control purposes.

Stamp coupling: Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

Control coupling: Control coupling exists between two modules, if data from one module is used to direct the order of instruction execution in another. An example of control coupling is a flag set in one module and tested in another module.

Common coupling: Two modules are common coupled, if they share some global data items.

Content coupling: Content coupling exists between two modules, if they share code. That is, a jump from one module into the code of another module can occur. Modern high-level programming languages such as C do not support such jumps across modules.

The different types of coupling are shown schematically in Figure 5.5. The degree of coupling increases from data coupling to content coupling. High coupling among modules not only makes a design solution difficult to understand and maintain, but it also increases development effort and also makes it very difficult to get these modules developed independently by different team members.

3.4 LAYERED ARRANGEMENT OF MODULES

The control hierarchy represents the organization of program components in terms of their call relationships. Thus we can say that the control hierarchy of a design is determined by the order in which different modules call each other. Many different types of notations have been used to represent the control hierarchy. The most common notation is a tree-like diagram known as a structure. However, other notations such as Warnier-Orr [1977, 1981] or Jackson diagrams [1975] may also be used. Since, Warnier-Orr and Jackson's notations are not widely used.

In a layered design solution, the modules are arranged into several layers based on their call relationships. A module is allowed to call only the modules that are at a lower layer. That is, a module should not call a module that is either at a higher layer or even in the same layer. Figure 3.6(a) shows a layered design, whereas Figure 3.6(b) shows a design that is not layered. Observe that the design solution shown in Figure 3.6(b), is actually not layered since all the modules can be considered to be in the same layer. In the following, we state the significance of a layered design and subsequently we explain it.

In a layered design, the top-most module in the hierarchy can be considered as a manager that only invokes the services of the lower level module to discharge its responsibility. The modules at the intermediate layers offer services to their higher layer by invoking the services of the lower layer modules and also by doing some work themselves to a limited extent. The modules at the lowest layer are the worker modules. These do not invoke services of any module and entirely carry out their responsibilities by themselves.

Understanding a layered design is easier since to understand one module, one would have to at best consider the modules at the lower layers (that is, the modules whose services it invokes). Besides, in a layered design errors are isolated, since an error in one module can affect only the higher layer modules. As a result, in case of any failure of a module, only the modules at the lower levels need to be investigated for the possible error. Thus, debugging time reduces significantly in a layered design. On the other hand,if the different modules call each other arbitrarily, then this situation would correspond to modules arranged in a single layer. Locating an error would be both difficult and time consuming. This is because, once a failure is observed, the cause of failure (i.e. error) can potentially be in any module, and all modules would have to be investigated for the error. In the following, we discuss some important concepts and terminologies associated with a layered design:

Superordinate and subordinate modules: In a control hierarchy, a module that controls another module is said to be superordinate to it. Conversely, a module controlled by another module is said to be subordinate to the controller.

Visibility: A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visibleto a module.

Control abstraction: In a layered design, a module should only invoke the functions of the modules that are in the layer immediately below it. In other words, the modules at the higher layers, should not be visible (that is, abstracted out) to the modules at the lower layers. This is referred to as control abstraction.

Depth and width: Depth and width of a control hierarchy provide an indication of the number of layers and the overall span of control respectively.For the design of Figure 3.6(a), the depth is 3 and width is also 3.

Fan-out: Fan-out is a measure of the number of modules that are directly controlled by a given module. In Figure 3.6(a), the fan-out of the module M1is 3. A design in which the modules have very high fan-out numbers is not a good design. The reason for this is that a very high fan-out is an indication that the module lacks cohesion. A module having a large fan-out (greaterthan 7) is likely to implement several different functions and not just a single cohesive function.

Fan-in: Fan-in indicates the number of modules that directly invoke a given module. High fan-in represents code reuse and is in general, desirable in a good design. In Figure 3.6(a), the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.

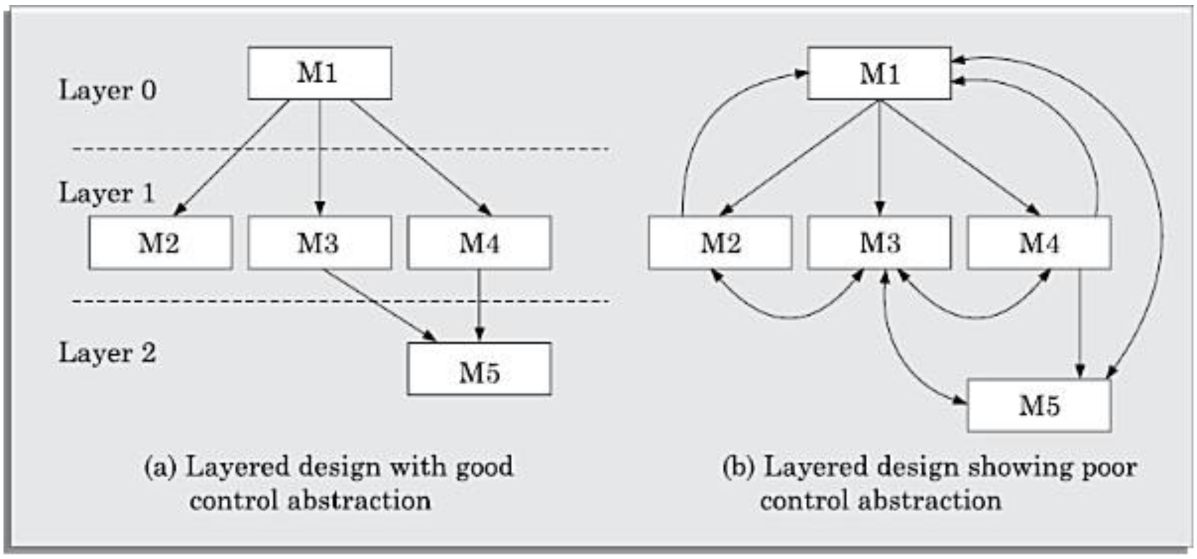

Layer 0
Layer 1
Layer 2
M1
M2
M3
M4
M5
(a) Layered design with good control abstraction
(b) Layered design showing poor control abstraction

Figure 3.6: Examples of good and poor control abstraction.

3.5 APPROACHES TO SOFTWARE DESIGN

There are two fundamentally different approaches to software design that are in use today— function-oriented design, and object-oriented design. Though these two design approaches are radically different, they are complementary rather than competing techniques. The object- oriented approach is a relatively newer technology and is still evolving. For development of large programs, the object- oriented approach is becoming increasingly popular due to certain advantages that it offers. On the other hand, function-oriented designing is a mature technology and has a large following.

3.5.1 Function-oriented Design

The following are the salient features of the function-oriented design approach:

Top-down decomposition: A system, to start with, is viewed as a black box that provides certain services (also known as high-level functions) to the users of the system.

In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed functions.

For example, consider a function create-new-library member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge. This high-level function may be refined into the following sub functions:

- assign-membership-number
- create-member-record
- print-bill

Each of these sub functions may be split into more detailed sub functions andso on.

Centralised system state: The system state can be defined as the values of certain data items that determine the response of the system to a user action or external event. For example, the set of books (i.e. whether borrowed by different users or available for issue) determines the state of a library automation system. Such data in procedural programs usually have global scope and are shared by many modules.

For example, in the library management system, several functions such asthe following share data such as member-records for reference.

- create-new-member
- delete-member
- update-member-record

A large number of function-oriented design approaches have been proposedin the past. A

few of the well-established function-oriented design approaches are asfollowing:

- Structured design by Constantine and Yourdon, [1979]
- Jackson's structured design by Jackson [1975]
- Warnier-Orr methodology [1977, 1981]
- Step-wise refinement by Wirth [1971]
- Hatley and Pirbhai's Methodology [1987]

3.5.2 Object-oriented Design

In the object-oriented design (OOD) approach, a system is viewed as being made up of a collection of objects (i.e. entities). Each object is associated with a set of functions that are called its methods. Each object contains its own data and is responsible for managing it. The data internal to an object cannot be accessed directly by other objects and only through invocation of the methods of the object. The system state is decentralized since there is no globally shared data in the system and data is stored in each object. For example, in library automation software, each library member may be a separate object with its own data and functions to operate on the stored data. The methods defined for one object cannot directly refer to or change the data of other objects.

The object-oriented design paradigm makes extensive use of the principles of abstraction and decomposition as explained below. Objects decompose a system into functionally independent modules. Objects can also be considered as instances of abstract data types (ADTs). The ADT concept did not originate from the object-oriented approach. In fact, ADT concept was extensively used in the ADA programming language introduced in the 1970s. ADT is an important concept that forms an important pillar of object- orientation. Let us now discuss the important concepts behind an ADT. There are, in fact, three important concepts associated with an ADT—data abstraction, data structure, data type. We discuss these in the following subsection:

Data abstraction: The principle of data abstraction implies that how data is exactly stored is abstracted away. This means that any entity external to the object (that is, an instance of an ADT) would have no knowledge about how data is exactly stored, organised, and manipulated inside the object. The entities external to the object can access the data internal to an object only by calling certain well-defined methods supported by the object. Consider an ADT such as a stack. The data of a stack object may internally be stored in an array, a linearly linked list, or a bidirectional linked list. The external entities have no knowledge of this and can access data of a stack object only through the supported operations such as push and pop.

Data structure: A data structure is constructed from a collection of primitive data items. Just as a civil engineer builds a large civil engineering structure using primitive building materials such as bricks, iron rods, and cement; a programmer can construct a data structure as an organised collection of primitive data items such as integer, floating point numbers, characters, etc.

Data type: A type is a programming language terminology that refers to anything that can be instantiated. For example, int, float, char etc., are the basic data types supported by C programming language. Thus, we can say that ADTs are user defined data types.

In object-orientation, classes are ADTs. But, what is the advantage of developing an application using ADTs? Let us examine the three main advantages of using ADTs in programs: The data of objects are encapsulated within the methods. The encapsulation principle is also known as data hiding. The encapsulation principle requires that data can be accessed and manipulated only through the methods supported by the object and not directly. This localises the errors. The reason for this is as follows. No program element is allowed to change a data, except through invocation of one of the methods. So, any error can easily be traced to the code segment changing the value. That is, the method that changes a data item, making it erroneous can be easily identified.

1. An ADT-based design displays high cohesion and low coupling. Therefore, object- oriented designs are highly modular.

2. Since the principle of abstraction is used, it makes the design solution easily understandable and helps to manage complexity.

3. Similar objects constitute a class. In other words, each object is a memberof some class. Classes may inherit features from a super class. Conceptually, objects communicate by message passing. Objects have their own internal data. Thus an object may exist in different states depending the values of the internal data. In different states, an object may behave differently.

Object-oriented versus function-oriented designapproaches

The following are some of the important differences between the function-oriented and object-oriented design Unlike function-oriented design methods in OOD, the basic abstractionis not the services available to the users of the system such as issue-book, display-book-details, find-issued-books, etc., but real-world entities such as member, book, book-register, etc. For example in OOD, an employee pay-roll software is not developed by designing functions such as update-employee-record, get-employee-address, etc., but by designing objects such as employees, departments, etc.

- In OOD, state information exists in the form of data distributed among several objects of the system. In contrast, in a procedural design, the state information is available in a centralized shared data store. For example, while developing an employee pay-roll system, the employee data such as the names of the employees, their code numbers, basic salaries, etc., are usually implemented as global data in a traditional programming system; whereas in an object-oriented design, these data are distributed among different employee objects of the system. Objects communicate by message passing. Therefore, one object may discover the state information of another object by sending a message to it. Of course, somewhere or other the real-world functions must be implemented.
- Function-oriented techniques group functions together if, as a group, they constitute a higher level function. On the other hand, object- oriented techniques group functions together on the basis of the data they operate on.

To illustrate the differences between the object-oriented and the function- oriented design approaches, let us consider an example—that of an automated fire-alarm system for a large building.

Automated fire-alarm system—customer requirements

The owner of a large multi-storied building wants to have a computerized fire alarm system designed, developed, and installed in his building. Smoke detectors and fire alarms would be placed in each room of the building. The fire alarm system would monitor the status of these smoke detectors. Whenever a fire condition is reported by any of the smoke detectors, the fire alarm system should determine the location at which the fire has been sensed and then sound the alarms only in the neighboring locations. The fire alarm system should also flash an alarm message on the computer console. Firefighting personnel would man the console round the clock. After a fire condition has been successfully handled, the fire alarm system should support resetting the alarms by the firefighting personnel.

Function-oriented approach: In this approach, the different high-level functions are first identified, and then the data structures are designed.

```
/* Global data (system state) accessible by various functions */
        BOOL   detector_status[MAX_ROOMS];
        int    detector_locs[MAX_ROOMS];
        BOOL   alarm-status[MAX_ROOMS]; /* alarm activated when status is set */
        int    alarm_locs[MAX_ROOMS]; /* room number where alarm is located */
        int    neighbour-alarms[MAX-ROOMS][10]; /* each detector has at most */
                                            /* 10 neighbouring alarm locations */
        int    sprinkler[MAX_ROOMS];
```

The functions which operate on the system state are:

```
interrogate_detectors();
get_detector_location();
determine_neighbour_alarm();
determine_neighbour_sprinkler();
ring_alarm();
activate_sprinkler();
reset_alarm();
reset_sprinkler();
report_fire_location();
```

Object-oriented approach: In the object-oriented approach, the different classes of objects are identified. Subsequently, the methods and data for each object are identified. Finally, an appropriate number of instances of each class is created.

class detector
attributes: status, location, neighbours operations: create, sense-status, get-location,
find-neighbours

class alarm
attributes: location, status
operations: create, ring-alarm, get_location, reset-alarm

class sprinkler

attributes: location, status
operations: create, activate-sprinkler, get_location,reset-sprinkler

We can now compare the function-oriented and the object-oriented approaches based on the two examples discussed above, and easily observethe following main differences:

- In a function-oriented program, the system state (data) is centralised and several functions access and modify this central data. In case of an object-oriented program, the state information (data) is distributed among various objects.
- In the object-oriented design, data is private in different objects and these are not available to the other objects for direct access and modification.

The basic unit of designing an object-oriented program is objects, whereas it is functions and modules in procedural designing. Objects appear as nouns in the problem description; whereas functions appear as verbs.

An object- oriented design be implemented by using an object-oriented language only. However, an object-oriented language such as C++ and Java support the definition of all the basic mechanisms of class, inheritance, objects, methods, etc. and also support all key object-oriented concepts that we have just discussed. Thus, an object-oriented language facilitates the implementation of an OOD. However, an OOD can as well be implemented using a conventional procedural languages—though it may require more effort to implement an OOD using a procedural language as compared to the effort required for implementing the same design using an object-oriented language.

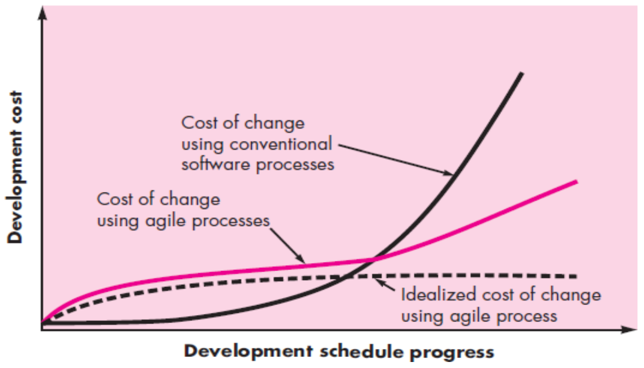
PART-II

Agility:

- Agility means effective (rapid and adaptive) response to change, effective communication among all stakeholder.
- Drawing the customer onto team and organizing a team so that it is in control of work performed. The Agile process is light-weight methods and People-based rather than plan-based methods.
- The agile process forces the development team to focus on software itself rather than design and documentation.
- The agile process believes in iterative method.
- The aim of agile process is to deliver the working model of software quickly to the customer For example: Extreme programming is the best known of agile process.

AGILITY AND THE COST OF CHANGE

The cost of change increases as the project progresses.(Figure 3.1, solid black curve). It is relatively easy to accommodate a change when a software team is gathering requirements (early in a project).



The team is in the middle of validation testing (something that occurs relatively late in the project), and stakeholder is requesting a major functional change. The change requires a modification to the architectural design of the software, the design and construction of three new components, modifications to another five components, the design of new tests, and so on. Costs and time increases quickly.

A well-designed agile process "flattens" the cost of change curve (Figure 3.1, shaded, solid curve), allowing a software team to accommodate changes late in a software project without cost and time impact. The agile process encompasses incremental delivery.

Agile Process

Any agile software process is characterized in a manner that addresses a number of key assumptions about the majority of software projects:

1. It is difficult to predict (estimate for future change) in advance which software requirements will persist and which will change.
2. For many types of software, design and construction are performed simultaneously. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not expected. (from a planning point of view)

Agility Principles

12 agility principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development.
3. Deliver working software frequently, from a couple of weeks to a couple of Months.
4. Stake holders and software engineers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development.
9. Continuous attention to technical excellence and good design enhances (increases) agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self– organizing teams.
12. At regular intervals, the team reflects on how to become more effective and then

adjusts its behavior accordingly.
Not every agile process model applies these 12 principles with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more of the principles.

The Politics of Agile Development

- There is debate about the benefits and applicability of agile software development as opposed to more conventional software engineering processes (produces documents rather than working product).
- Even within the agile, there are many proposed process models each with a different approach to the agility.

Human Factors

Agile software development takes the importance of "people factors". Number of different talents must exist among the people on an agile team and the team itself:
Competence: "competence" encompasses talent, specific software-related skills, and overall knowledge of the process.

Common focus: Members of the agile team may perform different Tasks and all should be focused on one goal—to deliver a working software increment to the customer within the time promised.
Collaboration: Team members must collaborate with one another and all other Stakeholders to complete the their task.
Decision-making ability: Any good software team (including agile teams) must be allowed the freedom to control its own destiny.
Fuzzy problem-solving ability: The agile team will continually have to deal with ambiguities (confusions or doubts).

Mutual trust and respect: The agile team exhibits the trust and respect.

Self-organization:

(1) The agile team organizes itself for the work to be done

(2) The team organizes the process to best accommodate its local environment

(3) The team organizes the work schedule to best achieve delivery of the software increment.

Extreme Programming

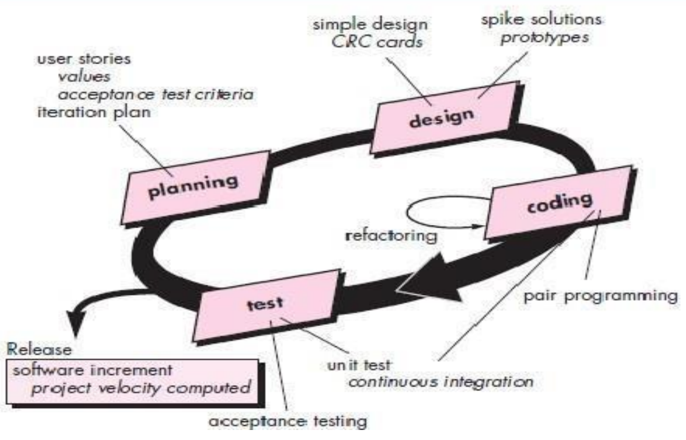
Extreme Programming (XP), the most widely used approach to agile software development. XP proposed by kent beck during the late 1980's.

XP Values

Beck defines a set of five *values* —communication, simplicity, feedback, courage, and respect. Each of these values is used in XP activities, actions, and tasks.

- Effective ***communication*** between software engineers and other stakeholders.
- To achieve ***simplicity***, XP restricts developers to design only for immediate needs, rather than future needs.
- ***Feedback*** is derived from three sources: the implemented software itself, the customer, and other software team members
- ***courage: (discipline)*** An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically.
- The agile team inculcates ***respect*** among it members, between other stakeholders and team members.

FIGURE 3.2
The Extreme Programming process



The XP(Extreme Programming) Process

XP Process have four framework activities: planning, design, coding, and testing.

Planning: The planning activity begins with *listening*—a requirements gathering activity.

- Listening leads to the creation of a set of "stories" (also called *user stories*) that describe required output, features, and functionality for software to be built.
- Each *story* is written by the customer and is placed on an index card. The customer assigns a *value* (i.e., a priority) to the story based on the overall business value of the feature or function.
- Members of the XP team then assess each story and assign a *cost*—measured in development weeks—to it.
- If the story is estimated to require more than three development weeks, the story into smaller stories and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time.
- The stories with highest value will be moved up in the schedule and implemented first.

Design: XP design follows the KIS (keep it simple) principle.

- If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a *spike solution.*
- XP encourages *refactoring*—a construction technique that is also a method for design optimization.
- Refactoring is the process of changing a software system in a way that it does not change the external behavior of the code and improves the internal structure.

Coding: Design work is done, the team does *not* move to code, develops a series of unit tests for each of the stories that is to be included in the current release (software increment).

- Once the unit test has been created, the developer is better able to focus on what must be implemented to pass the test.
- Once the code is complete, it can be unit-tested immediately, and providing feedback to the developers.
- A key concept during the coding activity is *pair programming*. i.e two people work together at one computer workstation to create code for a story.
- As pair programmers complete their work, the code they develop is integrated with the work of others.

Testing:

- As the individual unit tests are organized into a "universal testing suite" integration and validation testing of the system can occur on a daily basis.
- XP *acceptance tests*, also called *customer tests,* are specified by the customer and focus on overall system features and functionality. Acceptance tests are derived from user stories that have been implemented as part of a software release.

Industrial XP

Joshua Kerievsky describes *Industrial Extreme Programming* (IXP)

IXP has six new practices that are designed to help XP process works successfully for projects within a large organization.

Readiness assessment: the organization should conduct a *readiness assessment.*

(1) Development environment exists to support IXP(Industrial Extreme Programming).

(2) The team will be populated by the proper set of stakeholders.

(3) The organization has a distinct quality program and supports continuous improvement.

(4) The organizational culture will support the new values of an agile Team.

Project community:

- Classic XP suggests that the right people be used to populate the agile team to ensure success.
- The people on the team must be well-trained, adaptable and skilled.
- A community may have a team members and customers who are central to the success of the project.

Project chartering:

- Chartering examines the context of the project to determine how it complements, extends, or replaces existing systems or processes.

Test-driven management:

- Test-driven management establishes a series of measurable "destinations" and then defines mechanisms for determining whether or not these destinations have been reached.

Retrospectives.

- An IXP team conducts a specialized technical reviews after a software increment is delivered. Called a *retrospective.*
- The review examines "issues, events, and lessons-learned" across a software increment and/or the entire software release.

Continuous learning.

- learning is a vital part of continuous process improvement, members of the XP team are encouraged to learn new methods and techniques that can lead to a higher quality product.

The XP Debate

Requirements volatility. The customer is an active member of the XP team changes to requirements is requested informally. As a consequence,
the scope of the project can change and earlier work may have to be modified to accommodate current needs.

• ***Conflicting customer needs.*** Many projects have multiple customers, each with his own set of needs.

• ***Requirements are expressed informally.*** User stories and acceptance tests are the only explicit manifestation of requirements in XP. Specification is often needed to remove inconsistencies, and errors before the system is built.

• ***Lack of formal design***: when complex systems are built, design must have the overall structure of the software then it will exhibit quality.

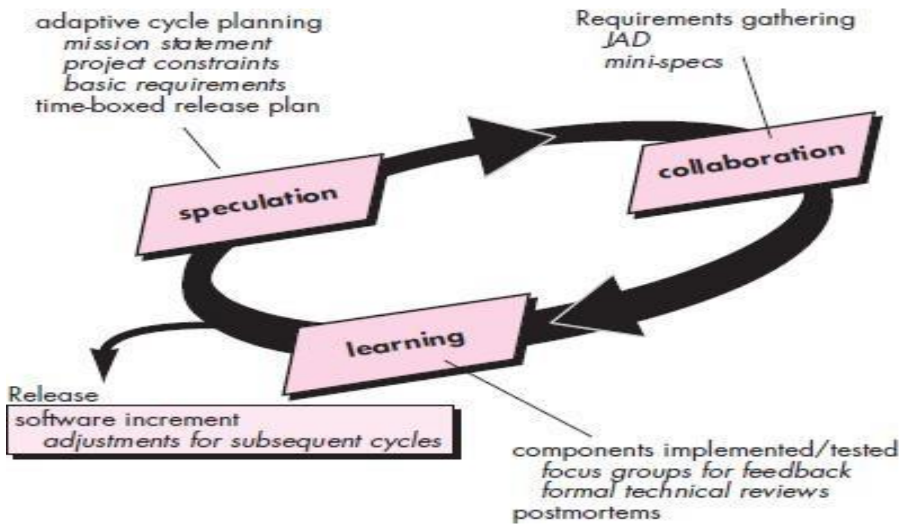
OTHER AGILE PROCESS MODELS

The most widely used of all agile process model is Extreme Programming (XP). But many other agile process models have been proposed and are in use across the industry. Among the most common are:

- Adaptive Software Development (ASD)
- Scrum
- Dynamic Systems Development Method (DSDM)
- Crystal
- Feature Drive Development (FDD)
- Lean Software Development (LSD)
- Agile Modeling (AM)
- Agile Unified Process (AUP)

Adaptive Software Development (ASD)*Adaptive Software Development* (ASD) has been proposed by Jim High smith as a technique for building complex systems. ASD focus on human collaboration and team self- organization.

- ASD "life cycle" (Figure 3.3) has three phases:- speculation, collaboration, and learning.
- ***speculation:*** The project is initiated and *adaptive cycle planning* is conducted. Adaptive cycle planning uses project initiation information—the customer's statement, project constraints (e.g., delivery dates or user descriptions), and basic requirements—to define the set of release cycles (software increments) that will be required for the project.



- Motivated people use ***collaboration***
- People working together must trust one another to (1) without criticize, (2) work as hard as or harder than they do, (3) have the skill set to contribute to the work and (4) communicate problems in a way that leads to effective action.

- **learning** will help them to improve their level of real understanding.

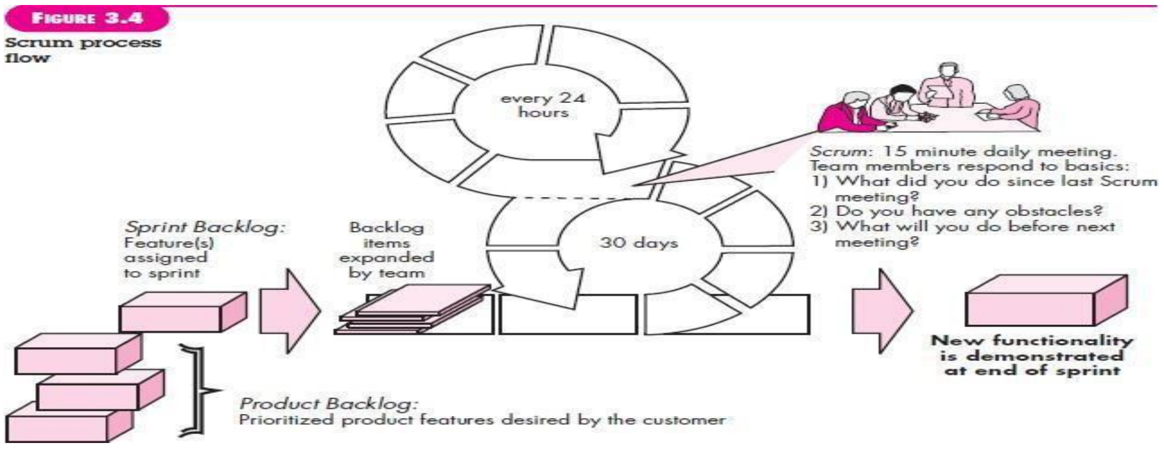
Scrum

- Scrum is an agile software development method that was coined by Jeff Sutherland and his development team in the early 1990's.
- Scrum has the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, actions and work tasks occur within a process called a *sprint.*
- scrum defines a set of development actions:

Backlog—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time (this is how changes are introduced). The product manager assesses the backlog and updates priorities as required.

Sprints—consist of work units that are required to achieve a requirement defined in the backlog .

FIGURE 3.4
Scrum process flow


every 24 hours
30 days
Scrum: 15 minute daily meeting.
Team members respond to basics:
1) What did you do since last Scrum meeting?
2) Do you have any obstacles?
3) What will you do before next meeting?
Sprint Backlog: Feature(s) assigned to sprint
Backlog items expanded by team
New functionality is demonstrated at end of sprint
Product Backlog: Prioritized product features desired by the customer

Scrum meetings—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members.

- What did you do since the last team meeting?
- What obstacles are you encountering?
- What do you plan to accomplish by the next team meeting?

A team leader, called a *Scrum master,* leads the meeting and assesses the responses from each person.

Demos—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer.

Dynamic Systems Development Method (DSDM)

- The *Dynamic Systems Development Method* (DSDM) is an agile software development approach.
- The DSDM—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete (100 percent) application.
- DSDM is an iterative software process in which each iteration follows the 80 percent rule. That is, only enough work is required for each increment to facilitate movement to the next increment.
- The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.
- The activities of DSDM are:

Feasibility study—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.
Business study—establishes the functional and information requirements that will allow the application to provide business value.
Functional model iteration—produces a set of incremental prototypes that demonstrate functionality for the customer.

- The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.

Design and build iteration—prototypes built during *functional model Iteration* to ensure that each has been engineered in a manner that it will provide operational business value for end users.
Implementation—places the latest software increment (an "operationalized" prototype) into the operational environment. It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place.

Crystal

- Alistair Cockburn and Jim Highsmith created the *Crystal family of agile methods* .
- Cockburn characterizes as "a resource limited, cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game".
- Cockburn and Highsmith have defined a set of methodologies, each with core elements that are common to all, and roles, process patterns, work products, and practice that are unique to each.
- The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects.

Feature Driven Development (FDD)

- *Feature Driven Development* (FDD) was originally coined by Peter Coad and his colleagues as a process model for object-oriented software engineering.
- A *feature* "is a client-valued function that can be implemented in two weeks or less".
- the definition of features provides the following benefits:
  - ➢ features are small blocks of deliverable functionality, users can describe them more easily.
  - ➢ Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks.
  - ➢ Because features are small, their design and code representations are easier.
- the following template for defining a feature:

<action> the **<result>** **<by for of to>** a(n) **<object>**

Where

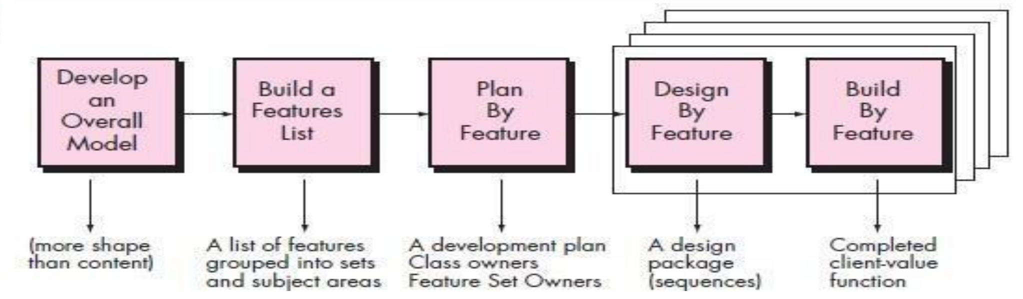
<object> is "a person, place, or thing."

Examples of features for an **e-commerce application** might be:

1) Add the product to shopping cart

2) Store the shipping-information for the customer

FIGURE 3.5
Feature Driven Development [Coa99] (with permission)


Develop an Overall Model
Build a Features List
Plan By Feature
Design By Feature
Build By Feature
(more shape than content)
A list of features grouped into sets and subject areas
A development plan Class owners Feature Set Owners
A design package (sequences)
Completed client-value function

- A feature set groups related features into business-related categories and is defined as:

  <action><-ing> a(n) **<object>**

For example: *Making a product sale* is a feature set that would encompass the features noted earlier and others.

- The FDD approach defines five "collaborating" framework activities as shown in **Figure 3.5.**
- It is essential for developers, their managers, and other stakeholders to understand project status.
- For that FDD defines six milestones during the design and implementation

of a feature: "design walkthrough, design, design inspection, code,

code inspection, promote to build".

Lean Software Development (LSD)

- *Lean Software Development* (LSD) has adapted the principles of lean manufacturing to the world of software engineering.
- LSD process can be summarized as *eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people,* and *optimize the whole.*
- For example, *eliminate waste* within the context of an agile software project as

(1) adding no extraneous features or functions

(2) assessing the cost and schedule impact of any newly requested requirement,

(3) removing any superfluous process steps,

(4) establishing mechanisms to improve the way team members find information,

(5) ensuring the testing finds as many errors as possible,

Agile Modeling (AM)

Agile Modeling (AM) suggests a wide array of "core" and "supplementary" modeling principles,

Model with a purpose: A developer who uses AM should have a specific goal (e.g., to communicate information to the customer or to help better understand aspect of the software) in mind before creating the model.

Use multiple models: There are many different models and notations that can be used to describe software.

- Each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.

Travel light: As software engineering work proceeds, keep only those models that

will provide long-term value.
Content is more important than representation: Modeling should impart information to its intended audience.
Know the models and the tools you use to create them: Understand the strengths and weaknesses of each model and the tools that are used to create it.
Adapt locally: The modeling approach should be adapted to the needs of the agile team.

Agile Unified Process (AUP)

The *Agile Unified Process* (AUP) adopts a "serial in the large" and "iterative in the small" philosophy for building computer-based systems. Each AUP iteration addresses the following activities:

• ***Modeling:*** UML representations of the business and problem domains are created.

• ***Implementation:*** Models are translated into source code.

• ***Testing:*** Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.

• ***Deployment:*** focuses on the delivery of a software increment and the acquisition of feedback from end users.

• ***Configuration and project management:*** Configuration management addresses change management, risk management, and the control of any persistent work products that are produced by the team.
Project management tracks and controls the progress of the team and coordinates team activities.

• ***Environment management:*** Environment management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

TOOL SET FOR AGILE DEVELOPMENT PROCESS

- Automated software tools (e.g., design tools) should be viewed as a minor supplement to the team's activities, and not to the success of the team.
- Collaborative and communication "tools" are generally low technology and incorporate any mechanism (whiteboards, poster sheets) that provides information and coordination among agile developers.
- other agile tools are used to optimize the environment in which the agile team works ,improve the team culture by social interactions, physical devices (e.g., electronic whiteboards), and process enhancement (e.g., pair programming or time-boxing)"

PART-III

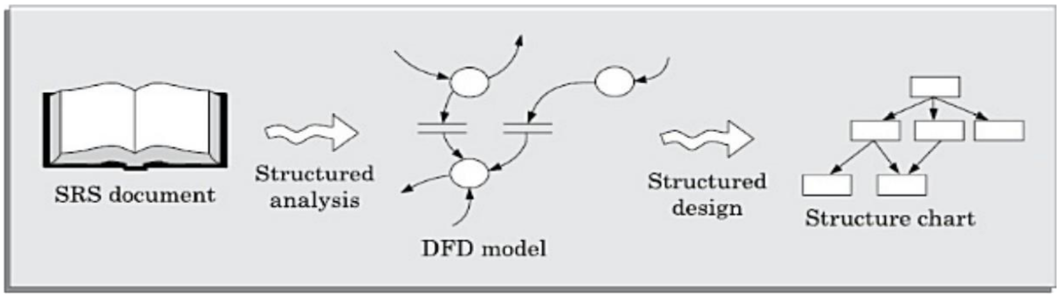
INTRODUCTION TO FUNCTION-ORIENTED SOFTWARE DESIGN:

- Function-oriented design techniques were proposed nearly four decades ago.
- very popular and are currently being used in many software development organizations. These techniques, to start with, view a system as a black-box that provides a set of services to the users of the software.
- These services are also known as the high-level functions supported by the software. During the design process, these high-level functions are successively decomposed into more detailed functions

- The term top-down decomposition is often used to denote the successive decomposition of a set of high-level functions into more detailed functions.
- Different identified functions are mapped to modules and a module structure is created.
- We shall discuss a methodology that has the essential features of several important function-oriented design methodologies.
- The design technique discussed here is called structured analysis/structured design (SA/SD) methodology.
- The SA/SD technique can be used to perform the high-level design of a software.

OVERVIEW OF SA/SD METHODOLOGY

- As the name itself implies, SA/SD methodology involves carrying out two distinct activities:
- Structured analysis (SA)
- Structured design (SD).
- The roles of structured analysis (SA) and structured design (SD) have been shown schematically in below figure.



- The structured analysis activity transforms the SRS document into a graphic model called the DFD model. During structured analysis, functional decomposition of the system is achieved. The purpose of structured analysis is to capture the detailed structure of the system as perceived by the user
- During structured design, all functions identified during structured analysis are mapped to a module structure. This module structure is also called the high level design or the software architecture for the given problem. This is represented using a structure chart. The purpose of structured design is to define the structure of the solution that is suitable for implementation
- The high-level design stage is normally followed by a detailed design stage.

STRUCTURED ANALYSIS

During structured analysis, the major processing tasks (high-level functions) of the system are analyzed, and t h e data flow among these processing tasks are represented graphically.

The structured analysis technique is based on the following underlying principles:

- Top-down decomposition approach.
- Application of divide and conquer principle.
- Through this each high level function is independently decomposed into detailed functions. Graphical representation of the analysis results using data flow

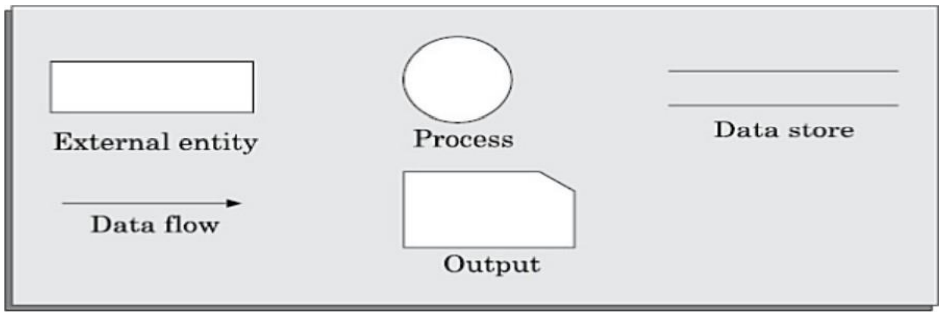
diagrams (DFDs).

WHAT IS DFD?

- A DFD is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among those functions.
- Though extremely simple, it is a very powerful tool to tackle the complexity of industry standard problems.
- A DFD model only represents the data flow aspects and does not show the sequence of execution of the different functions and the conditions based on which a function may or may not be executed.
- In the DFD terminology, each function is called a process or a bubble. Each function as a processing station (or process) that consumes some input data and produces some output data.
- DFD is an elegant modeling technique not only to represent the results of structured analysis but also useful for several other applications.
- Starting with a set of high-level functions that a system performs, a DFD model represents the sub functions performed by the functions using a hierarchy of diagrams.

Primitive symbols used for constructing DFDs

There are essentially five different types of symbols used for constructing DFDs.

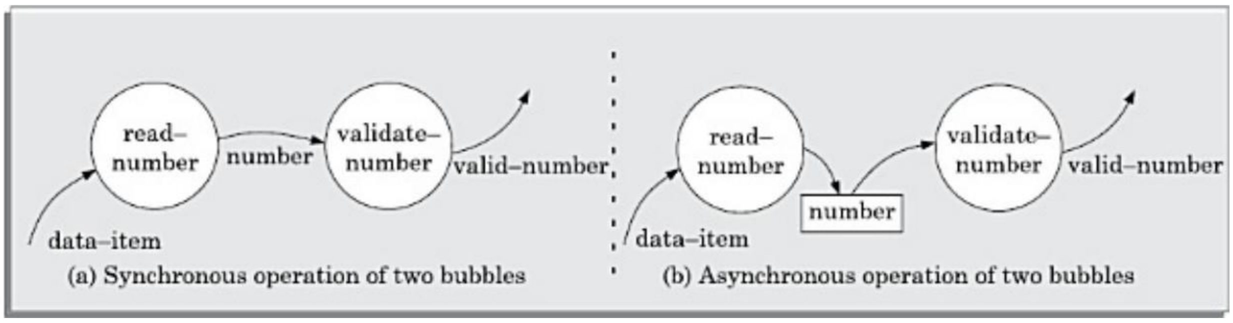


- Function symbol: A function is represented using a circle. This symbol is called a process or a bubble. Bubbles are annotated with the names of the corresponding functions
- External entity symbol: represented by a rectangle. The external entities are essentially those physical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system.
- Data flow symbol: A directed arc (or an arrow) is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow.
- Data store symbol: A data store is represented using two parallel lines. It represents a logical file. That is, a data store symbol can represent either a data structure or a physical file on disk. Connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store.

- Output symbol: The output symbol is as shown in Figure 6.2. The output symbol is used when a hard copy is produced.

Important concepts associated with constructing DFD models Synchronous and asynchronous operations

- If two bubbles are directly connected by a data flow arrow, then they are synchronous. This means that they operate at the same speed.
- If two bubbles are connected through a data store, as in Figure (b) then the speed of operation of the bubbles is independent.



Data Dictionary

- Every DFD model of a system must be accompanied by a data dictionary. A data dictionary lists all data items that appear in a DFD model.
- A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items.
- It includes all data flows and the contents of all data stores appearing on all the DFDs in a DFD model.
- For the smallest units of data items, the data dictionary simply lists their name and their type.
- Composite data items are expressed in terms of the component data items using certain operators.

● The dictionary plays a very important role in any software development process, especially for the following reasons:
○ A data dictionary provides a standard terminology for all relevant data for use by the developers working in a project.
○ The data dictionary helps the developers to determine the definition of different data structures in terms of their component elements while implementing the design.
○ The data dictionary helps to perform impact analysis. That is, it is possible to determine the effect of some data on various processing activities and vice versa.
● For large systems, the data dictionary can become extremely complex and voluminous.
● Computer-aided software engineering (CASE) tools come handy to overcome this problem.
● Most CASE tools usually capture the data items appearing in a DFD as the DFD is drawn, and automatically generate the data dictionary.

Developing the DFD model of a system:

- The DFD model of a problem consists of many DFDs and a single data dictionary. The DFD model of a system i s constructed by using a hierarchy of DFDs.
- The top level DFD is called the level 0 DFD or the context diagram.
  - This is the most abstract (simplest) representation of the system (highest level). It is the easiest to draw and understand.
- At each successive lower level DFDs, more and more details are gradually introduced.
- To develop a higher-level DFD model, processes are decomposed into their sub processes and the data flow among these sub processes is identified.
- Level 0 and Level 1 consist of only one DFD each. Level 2 may contain up to 7 separate DFDs, and level 3 up to 49 DFDs, and so on.
- However, there is only a single data dictionary for the entire DFD model.

Context Diagram/Level 0 DFD:

- The context diagram is the most abstract (highest level) data flow representation of a system. It represents the entire system as a single bubble.
- The bubble in the context diagram is annotated with the name of the software system being developed.
- The context diagram establishes the context in which the system operates that is, who are the users, what data do they input to the system, and what data they received by the system.
- The data input to the system and the data output from the system are represented as incoming and outgoing arrows.

Level 1 DFD:

- The level 1 DFD usually contains three to seven bubbles.
- The system is represented as performing three to seven important functions.
- To develop the level 1 DFD, examine the high-level functional requirements in the SRS document.
- If there are three to seven high level functional requirements, then each of these can be directly represented as a bubble in the level 1 DFD.
- Next, examine the input data to these functions and the data output by these functions as documented in the SRS document and represent them appropriately in the diagram.

Decomposition:

- Each bubble in the DFD represents a function performed by the system.
- The bubbles are decomposed into sub functions at the successive levels of the DFD model. Decomposition of a bubble is also known as factoring or exploding a bubble.

Each bubble at any level of DFD is usually decomposed to anything from three to seven bubbles. Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm

STRUCTURED DESIGN

- The aim of structured design is to transform the results of the structured analysis into a structure chart.

- A structure chart represents the software architecture.
- The various modules making up the system, the module dependency (i.e. which module calls which other modules), and the parameters that are passed among the different modules.
- The structure chart representation can be easily implemented using some programming language.
- The basic building blocks using which structure charts are designed are as following:
    - ***Rectangular boxes:*** A rectangular box represents a module.
    - ***Module invocation arrows:*** An arrow connecting two modules implies that during program execution control is passed from one module to the other in the direction of the connecting arrow.
    - ***Data flow arrows:*** These are small arrows appearing alongside the module invocation arrows. Represent the fact that the named data passes from one module to the other in the direction of the arrow.
    - ***Library modules:*** A library module is usually represented by a rectangle with double edges. Libraries comprise the frequently called modules.
    - ***Selection:*** The diamond symbol represents the fact that one module of several modules connected with the diamond symbol i s invoked depending on the outcome of the condition attached with the diamond symbol.

○ Repetition: A loop around the control flow arrows denotes that the respective modules are invoked repeatedly.

● In any structure chart, there should be one and only one module at the top, called the root.

● There should be at most one control relationship between any two modules in the structure chart. This means that if module A invokes module B, module B cannot invoke module A.

Flow Chart vs Structure chart:

● Flow chart is a convenient technique to represent the flow of control in a program.

● A structure chart differs from a flow chart in three principal ways:

1. It is usually difficult to identify the different modules of a program from its flow chart representation.
2. Data interchange among different modules is not represented in a flow chart.
3. Sequential ordering of tasks that is inherent to a flow chart is suppressed in a structure chart.

Transformation of a DFD Model into Structure Chart:

● Systematic techniques are available to transform the DFD representation of a problem into a module structure represented as a structure chart.

● Structured design provides two strategies to guide transformation of a DFD into a structure chart:

○ Transform analysis
○ Transaction analysis

● Normally, one would start with the level 1 DFD, transform it into module representation using either the transform or transaction analysis and then proceed toward the lower level DFDs

● At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD.

Whether to apply transform or transaction processing?

Given a specific DFD of a model, one would have to examine the data input to the diagram.

If all the data flow into the diagram are processed in similar ways (i.e. if all the input data flow arrows are incident on the same bubble in the DFD) then transform analysis is

applicable. Otherwise, transaction analysis is applicable.
Transform Analysis:

- Transform analysis identifies the primary functional components (modules) and the input and output data for these components.
- The first step in transform analysis is to divide the DFD into three types of parts:
  - Input (afferent branch)
  - Processing (central transform)
  - Output (efferent branch)
- In the next step of transform analysis, the structure chart is derived by drawing one functional component each for the central transform, the afferent and efferent branches.
- These are drawn below a root module, which would invoke these modules.

Transaction Analysis:

- Transaction analysis is an alternative to transform analysis and is useful while designing transaction processing programs.
- A transaction allows the user to perform some specific type of work by using the software.
- For example, 'issue book', 'return book', 'query book', etc., are transactions.
- As in transform analysis, first all data entering into the DFD need to be identified.
- In a transaction-driven system, different data items may pass through different computation paths through the DFD.
- This is in contrast to a transform centered system where each data item entering the DFD goes through the same processing steps.
- Each different way in which input data is processed is a transaction. For each identified transaction, trace the input data to the output.
- All the traversed bubbles belong to the transaction.
- These bubbles should be mapped to the same module on the structure chart.
- In the structure chart, draw a root module and below this module draw each identified transaction as a module.

DETAILED DESIGN:

- During detailed design the pseudo code description of the processing and the different data structures are designed for the different modules of the structure chart.
- These are usually described in the form of module specifications (MSPEC). MSPEC is usually written using structured English.
- The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower level modules.
- The MSPEC for the leaf-level modules should describe in algorithmic form how the primitive processing steps are carried out.
- To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.

DESIGN REVIEW:

- After a design is complete, the design is required to be reviewed.
- The review team usually consists of members with design, implementation, testing, and maintenance perspectives.
- The review team checks the design documents especially for the following aspects:
- Traceability: Whether each bubble of the DFD can be traced to some module in the

structure chart and vice versa.

- Correctness: Whether all the algorithms and data structures of the detailed design are correct.
- Maintainability: Whether the design can be easily maintained in future.
- Implementation: Whether the design can be easily and efficiently implemented.
- After the points raised by the reviewers are addressed by the designers, the design document becomes ready for implementation.

PART-IV

USER INTERFACE DESIGN

● The user interface portion of a software product is responsible for all interactions with the user. Almost every software product has a user interface.
● User interface part of a software product is responsible for all interactions.
● The user interface part of any software product is of direct concern to the end-users.
● No wonder then that many users often judge a software product based on its user interface
● an interface that is difficult to use leads to higher levels of user errors and ultimately leads to user dissatisfaction.
● Sufficient care and attention should be paid to the design of the user interface of any software product.
● Systematic development of the user interface is also important.
● Development of a good user interface usually takes a significant portion of the total system development effort.
● For many interactive applications, as much as 50 per cent of the total development effort is spent on developing the user interface part.
● Unless the user interface is designed and developed in a systematic manner, the total effort required to develop the interface will increase tremendously.

CHARACTERISTICS OF A GOOD USER INTERFACE

The different characteristics that are usually desired of a good user interface. Unless we know what exactly is expected of a good user interface, we cannot possibly design one.
● **Speed of learning:**
○ A good user interface should be easy to learn.
○ A good user interface should not require its users to memorize commands.
○ Neither should the user be asked to remember information from one screen to another
■ **Use of metaphors and intuitive command names:** Speed of learning an interface is greatly facilitated if these are based on some dayto-day real-life examples or some physical objects with which the users are familiar with. The abstractions of real-life objects or concepts used in user interface design are called metaphors.
■ **Consistency:** Once a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions.
■ **Component-based interface:** Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar with.
○ The speed of learning characteristic of a user interface can be determined by measuring the training time and practice that users require before they can effectively use the software.
● **Speed of use:**

○ Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands.
○ It indicates how fast the users can perform their intended tasks.
○ The time and user effort necessary to initiate and execute different commands should be minimal.
○ This can be achieved through careful design of the interface.
○ The most frequently used commands should have the smallest length or be available at the top of a menu.

● **Speed of recall:**

○ Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximized.
○ This characteristic is very important for intermittent users.
○ Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures, and intuitive command names.

● **Error prevention:**

○ A good user interface should minimize the scope of committing errors while initiating different commands.
○ The error rate of an interface can be easily determined by monitoring the errors committed by an average user while using the interface.
○ The interface should prevent the user from entering wrong values.

● **Aesthetic and attractive:**

○ A good user interface should be attractive to use.
○ An attractive user interface catches user attention and fancy.
○ In this respect, graphics-based user interfaces have a definite advantage over text-based interfaces.

● **Consistency:**

○ The commands supported by a user interface should be consistent.
○ The basic purpose of consistency is to allow users to generalize the knowledge about aspects of the interface from one part to another.

● **Feedback:**

○ A good user interface must provide feedback to various user actions.
○ Especially, if any user request takes more than a few seconds to process, the user should be informed about the state of the processing of his request.
○ In the absence of any response from the computer for a long time, a novice user might even start recovery/shutdown procedures in panic.

● **Support for multiple skill levels:**

○ A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users.
○ This is necessary because users with different levels of experience in using an application prefer different types of user interfaces.

○ Experienced users are more concerned about the efficiency of the command issue procedure, whereas novice users pay importance to usability aspects.
○ The skill level of users improves as they keep using a software product and they look for commands to suit their skill levels.

● **Error recovery (undo facility):**

○ While issuing commands, even the expert users can commit errors.
○ Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface.
○ Users are inconvenienced if they cannot recover from the errors they commit while using a software.

○ If the users cannot recover even from very simple types of errors, they feel irritated, helpless, and out of control.

● **User guidance and on-line help:**

○ Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software.

○ Whenever users need guidance or seek help from the system, they should be provided with appropriate guidance and help.

BASIC CONCEPTS:

User Guidance and Online help:

Users may seek help about the operation of the software any time while using the software. This is provided by the on-line help system.

1. **Online help system:**

● Users expect the on-line help messages to be tailored to the context in which they invoke the "help system".

● Therefore, a good online help system should keep track of what a user is doing while invoking the help system and provide the output message in a context-dependent way.

2. **Guidance messages:**

● The guidance messages should be carefully designed to prompt the user about the next actions he might pursue, the current status of the system, the progress so far made in processing his last command, etc.

● A good guidance system should have different levels of sophistication.

3. **Error Messages:**

● Error messages are generated by a system either when the user commits some error or when some errors encountered by the system during processing due to some exceptional conditions, such as out of memory, communication link broken, etc.

● Users do not like error messages that are either ambiguous or too general such as "invalid input or system error". Error messages should be polite.

Mode-based and modeless interfaces:

● A mode is a state or collection of states in which only a subset of all user interaction tasks can be performed.

● In a modeless interface, the same set of commands can be invoked at any time during the running of the software.

● Thus, a modeless interface has only a single mode and all the commands are available all the time during the operation of the software.

● On the other hand, in a mode-based interface, different sets of commands can be invoked depending on the mode in which the system is.

● A mode-based interface can be represented using a state transition diagram.

Graphical User Interface versus Text-based User Interface:

● In a GUI multiple windows with different information can simultaneously be displayed on the user screen. user has the flexibility to simultaneously interact with several related items at any time

● Iconic information representation and symbolic information manipulation is possible in a GUI.

● A GUI usually supports command selection using an attractive and user-friendly menu selection system.

● In a GUI, a pointing device such as a mouse or a light pen can be used for issuing commands.

● A GUI requires special terminals with graphics capabilities for running and also requires special input devices such a mouse.

● A text-based user interface can be implemented even on a cheap alphanumeric display terminal.

- Graphics terminals are usually much more expensive than alphanumeric terminals.

TYPES OF USER INTERFACES:

- User interfaces can be classified into the following three categories:
  - Command language-based interfaces.
  - Menu-based interfaces.
  - Direct manipulation interfaces.
- Each of these categories of interfaces has its own characteristic advantages and disadvantages.

Command Language-based Interface

- A command language-based interface—as the name itself suggests, is based on designing a command language which the user can use to issue the commands.
- The user is expected to frame the appropriate commands in the language and type them appropriately whenever required.
- A simple command language-based interface might simply assign unique names to the different commands.
- However, a more sophisticated command language-based interface may allow users to compose complex commands by using a set of primitive commands.
- The command language interface allows for the most efficient command issue procedure requiring minimal typing.
- A command language-based interface can be implemented even on cheap alphanumeric terminals.
- A command language-based interface is easier to develop compared to a menu-based or a direct-manipulation interface because compiler writing techniques are well developed
- Command language-based interfaces suffer from several drawbacks.
- Command language-based interfaces are difficult to learn and require the user to memorize the set of primitive commands.
- Most users make errors while formulating commands.
- All interactions with the system are through a key-board and cannot take advantage of effective interaction devices such as a mouse.

Issues in designing a command language based interface:

- The designer has to decide what mnemonics (command names) to use for the different commands.
- The designer has to decide whether the users will be allowed to redefine the command names to suit their own preferences.
- The designer has to decide whether it should be possible to compose primitive commands to form more complex commands.

Menu-Based Interfaces:

- An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands.
- A menu-based interface is based on recognition of the command names, rather than recollection.
- In a menu-based interface the typing effort is minimal.
- A major challenge in the design of a menu-based interface is to structure a large number of menu choices into manageable forms.
- Techniques available to structure a large number of menu items:
  - **Scrolling menu:**

■ Sometimes the full choice list is large and cannot be displayed within the menu area, scrolling of the menu items is required.

■ In a scrolling menu all the commands should be highly correlated, so that the user can easily locate a command that he needs.

■ This is important since the user cannot see all the commands at any one time.

○ **Walking menu:**

■ Walking menu is very commonly used to structure a large collection of menu items.

■ When a menu item is selected, it causes further menu items to be displayed adjacent to it in a sub-menu.

■ A walking menu can successfully be used to structure commands only if there are tens rather than hundreds of choices.

○ **Hierarchical menu:**

■ This type of menu is suitable for small screens with limited display area such as that in mobile phones.

■ The menu items are organized in a hierarchy or tree structure.

■ Selecting a menu item causes the current menu display to be replaced by an appropriate sub-menu.

Direct Manipulation Interfaces:

● Direct manipulation interfaces present the interface to the user in the form of visual models.

● Direct manipulation interfaces are sometimes called iconic interfaces.

● In this type of interface, the user issues commands by performing actions on the visual representations of the objects, e.g., pull an icon representing a file into an icon representing a trash box, for deleting the file.

● Important advantages of iconic interfaces include the fact that the icons can be recognized by the users very easily, and that icons are language independent.

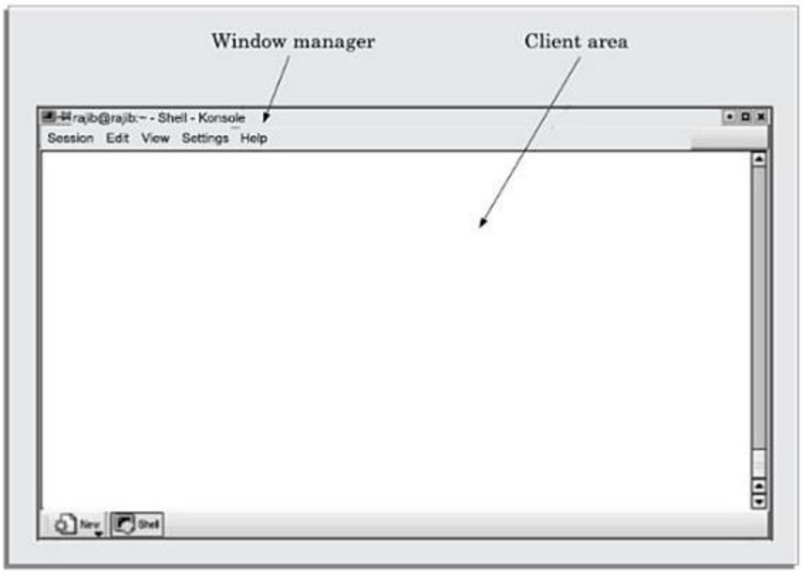
● However, experienced users find direct manipulation interfaces very useful too.

● Also, it is difficult to give complex commands using a direct manipulation interface.

FUNDAMENTALS OF COMPONENT-BASED GUI DEVELOPMENT

Graphical user interfaces became popular in the 1980s. The main reason why there were very few GUI-based applications prior to the eighties is that graphics terminals were too expensive. For example, the price of a graphics terminal those days was much more than what a high-end personal computer costs these days. Also, the graphics terminals were of storage tube type and lacked raster capability. One of the first computers to support GUI-based applications was the Apple Macintosh computer. In fact, the popularity of the Apple Macintosh computer in the early eighties is directly attributable to its GUI. In those early days of GUI design, the user interface programmer typically started his interface development from the scratch. Starting from simple pixel display routines, write programs to draw lines, circles, text, etc. He would then develop his own routines to display menu items, make menu choices, etc. The current user interface style has undergone a sea change compared to the early style. The current style of user interface development is component-based. It recognizes that every user interface can easily be built from a handfuls of predefined components such as menus, dialog boxes, forms, etc. Besides the standard components, and the facilities to create good interfaces from them, one of the basic support available to the user interface developers is the window system. The window system lets the application programmer create and manipulate windows without having to write the basic windowing functions. In the following subsections, we provide an overview of the window management system, the component-based development style, and visual programming. Below fig shows Window System Most

modern graphical user interfaces are developed using some window system. A window system can generate displays through a set of windows.
Window: A window is a rectangular area on the screen. A window can be considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent activities, e.g., one window can be used for editing a program and another for drawing pictures, etc.


Window manager
Client area
rajib@rajib:~ - Shell - Konsole
Session Edit View Settings Help
New
Shell
Figure: Window with client and user areas marked

A window can be divided into two parts

- client part :The client area makes up the whole of the window, except for the borders and scroll bars. The client area is the area available to a client application for display.
- non-client part: The non-client-part of the window determines the look and feel of the window. The look and feel defines a basic behavior for all windows, such as creating, moving, resizing, iconifying of the windows.
- The window manager is responsible for managing and maintaining the non-client area of a window. A basic window with its different parts is shown in Figure above

Window management system (WMS):-A graphical user interface typically consists of a large number of windows. Therefore, it is necessary to have some systematic way to manage these windows. Most graphical user interface development environments do this through a window management system (WMS).
A window management system is primarily a resource manager. It keeps track of the screen area resource and allocates it to the different windows that seek to use the screen. From a broader perspective, a WMS can be considered as a user interface management system (UIMS) —which not only does resource management, but also provides the basic behavior to the windows and provides several utility routines to the application programmer for user interface development. A WMS simplifies the task of a GUI designer to a great extent by providing the basic behaviour to the various windows such as move, resize, iconify, etc. as soon as they are created and by providing the basic routines to manipulate the windows from the application program such as creating, destroying, changing different attributes of the windows, and drawing text, lines, etc.
A WMS consists of two parts
• a window manager, and
• a window system.
Window manager and window system: The window manager is built on the top of the

window system in the sense that it makes use of various services provided by the window system. The window manager and not the window system determine how the windows look and behave. In fact, several kinds of window managers can be developed based on the same window system. The window manager can be considered as a special kind of client that makes use of the services (function calls) supported by the window system. The application programmer can also directly invoke the services of the window system to develop the user interface. The relationship between the window manager, window system, and the application program is shown in Figure. This figure shows that the end-user can either interact with the application itself or with the window manager (resize, move, etc.) and both the application and the window manger invoke services of the window manager.

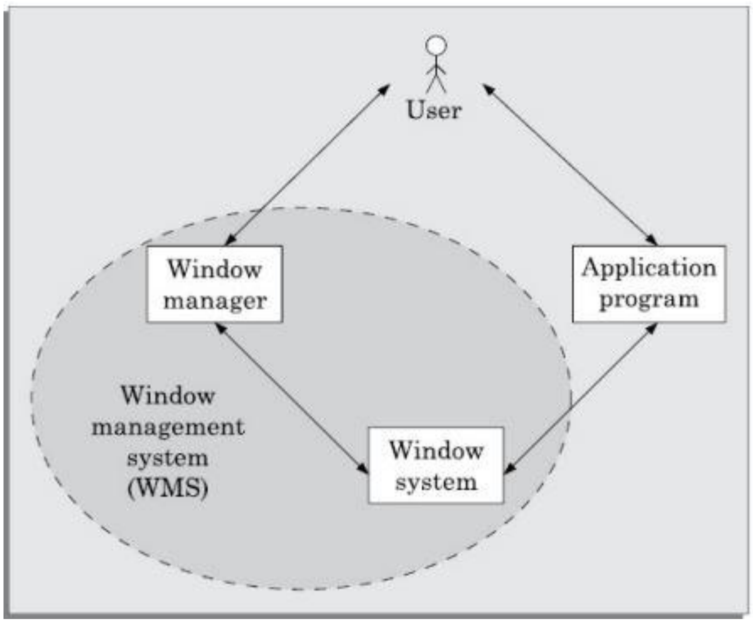


Figure: Window Management System

It is usually cumbersome to develop user interfaces using the large set of routines provided by the basic window system. Therefore, most user interface development systems usually provide a high-level abstraction called widgets for user interface development. A widget is the short form of a window object. We know that an object is essentially a collection of related data with several operations defined on these data which are available externally to operate on these data. The data of an window object are the geometric attributes (such as size, location etc.) and other attributes such as its background and foreground colour, etc. The operations that are defined on these data include, resize, move, draw, etc. Widgets are the standard user interface components. A user interface is usually made up by integrating several widgets.

Component-based development: A development style based on widgets is called component-based (or widget-based) GUI development style. There are several important advantages of using a widget-based design style. One of the most important reasons to use widgets as building blocks is because they help users learn an interface fast. In this style of development, the user interfaces for different applications are built from the same basic components. Therefore, the user can extend his knowledge of the behaviour of the standard components from one application to the other. Also, the component-based user interface development style reduces the application programmer’s work significantly as he is more of a user interface component integrator than a programmer in the traditional

sense.
Visual programming: Visual programming is the drag and drop style of program development. In this style of user interface development, a number of visual objects (icons) representing the GUI components are provided by the programming environment. The application programmer can easily develop the user interface by dragging the required component types (e.g., menu, forms, etc.) from the displayed icons and placing them wherever required. Thus, visual programming can be considered as program development through manipulation of several visual objects. Reuse of program components in the form of visual objects is an important aspect of this style of programming. Though popular for user interface development, this style of programming can be used for other applications such as Computer-Aided Design application (e.g., factory design), simulation, etc. User interface development using a visual programming language greatly reduces the effort required to develop the interface. Examples of popular visual programming languages are Visual Basic, Visual C++, etc. Visual C++ provides tools for building programs with windowbased user interfaces for Microsoft Windows environments. In visual C++ you usually design menu bars, icons, and dialog boxes, etc. before adding them to your program. These objects are called as resources. You can design shape, location, type, and size of the dialog boxes before writing any C++ code for the application.
Types of Widgets: Different interface programming packages support different widget sets. However, a surprising number of them contain similar kinds of widgets, so that one can think of a generic widget set which is applicable to most interfaces. The following widgets we have chosen as representatives of this generic class.
Label widget: This is probably one of the simplest widgets. A label widget does nothing except to display a label, i.e., it does not have any other interaction capabilities and is not sensitive to mouse clicks. A label widget is often used as a part of other widgets.
Container widget: These widgets do not stand by themselves, but exist merely to contain other widgets. Other widgets are created as children of the container widget. When the container widget is moved or resized, its children widget also get moved or resized. A container widget has no callback routines associated with it.
Pop-up menu: These are transient and task specific. A pop-up menu appears upon pressing the mouse button, irrespective of the mouse position.
Pull-down menu: These are more permanent and general. You have to move the cursor to a specific location and pull down this type of menu.
Dialog boxes: We often need to select multiple elements from a selection list. A dialog box remains visible until explicitly dismissed by the user. A dialog box can include areas for entering text as well as values. If an apply command is supported in a dialog box, the newly entered values can be tried without dismissing the box. Though most dialog boxes ask you to enter some information, there are some dialog boxes which are merely informative, alerting you to a problem with your system or an error you have made. Generally, these boxes ask you to read the information presented and then click OK to dismiss the box.
Push button: A push button contains key words or pictures that describe the action that is triggered when you activate the button. Usually, the action related to a push button occurs immediately when you click a push button unless it contains an ellipsis (. . .). A push button with an ellipsis generally indicates that another dialog box will appear. Radio buttons: A set of radio buttons are used when only one option has to be selected out of many options. A radio button is a hollow circle followed by text describing the option it stands for. When a radio button is selected, it appears filled and the previously selected radio button from the group is unselected. Only one radio button from a group can be selected at any time. This operation is similar to that of the band selection buttons that

were available in old radios.

Combo boxes: A combo box looks like a button until the user interacts with it. When the user presses or clicks it, the combo box displays a menu of items to choose from. Normally a combo box is used to display either one-of-many choices when space is limited, the number of choices is large, or when the menu items are computed at run-time.

An Overview of X-Window/MOTIF: One of the important reasons behind the extreme popularity of the X-window system is probably due to the fact that it allows development of portable GUIs. Applications developed using the X-window system are deviceindependent. Also, applications developed using the X-window system become network independent in the sense that the interface would work just as well on a terminal connected anywhere on the same network as the computer running the application is. Network-independent GUI operation has been schematically represented in Figure below. Here, A is the computer application in which the application is running. B can be any computer on the network from where you can interact with the application. Networkindependent GUI was pioneered by the X-window system in the mid-eighties at MIT (Massachusetts Institute of Technology) with support from DEC (Digital Equipment Corporation). Now-a-days many user interface development systems support network-independent GUI development, e.g., the AWT and Swing components of Java.

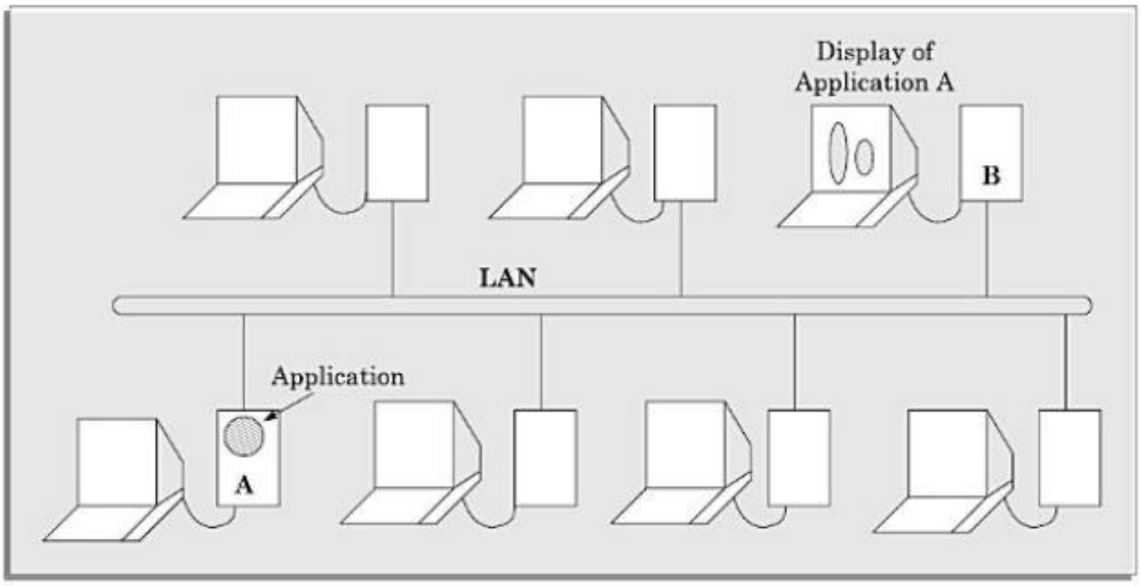


Figure Network-independent GUI

The X-window functions are low level functions written in C language which can be called from application programs. But only the very serious application designer would program directly using the X-windows library routines. Built on top of X-windows are higher level functions collectively called Xtoolkit, which consists of a set of basic widgets and a set of routines to manipulate these widgets. One of the most widely used widget sets is X/Motif. Digital Equipment Corporation (DEC) used the basic X-window functions to develop its own look and feel for interface designs called DECWindows. In the following, we shall provide a very brief overview of the X-window system and its architecture and the interested reader is referred to Scheifler et al. [1988] for further study on graphical user interface development using X-windows and Motif.

X Architecture: The X architecture is pictorially depicted in Figure below. The different terms used in this diagram are explained as follows:

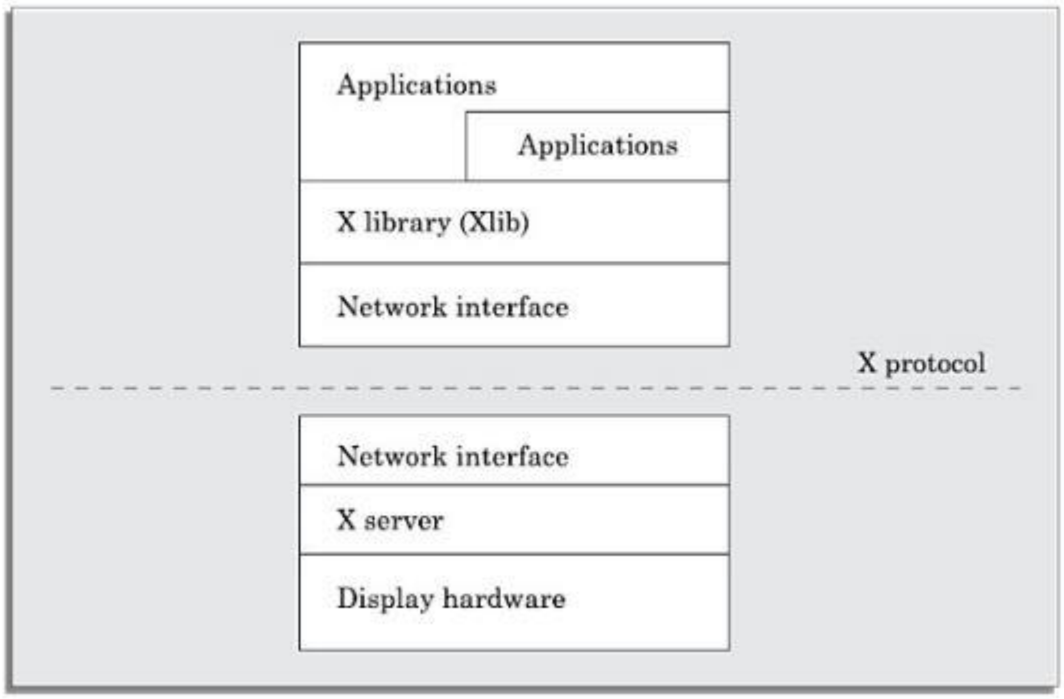


Figure: Architecture of the X System

Xserver: The X server runs on the hardware to which the display and the key board are attached. The X server performs low-level graphics, manages window, and user input functions. The X server controls accesses to a bit-mapped graphics display resource and manages it.
X protocol: The X protocol defines the format of the requests between client applications and display servers over the network. The X protocol is designed to be independent of hardware, operating systems, underlying network protocol, and the programming language used.
X library (Xlib): The Xlib provides a set of about 300 utility routines for applications to call. These routines convert procedure calls into requests that are transmitted to the server. Xlib provides low level primitives for developing an user interface, such as displaying a window, drawing characters and graphics on the window, waiting for specific events, etc.
Xtoolkit (Xt): The Xtoolkit consists of two parts: the intrinsics and the widgets. We have already seen that widgets are predefined user interface components such as scroll bars, menu bars, push buttons, etc. for designing GUIs. Intrinsics are a set of about a dozen library routines that allow a programmer to combine a set of widgets into a user interface. In order to develop a user interface, the designer has to put together the set of components (widgets) he needs, and then he needs to define the characteristics (called resources) and behaviour of these widgets by using the intrinsic routines to complete the development of the interface. Therefore, developing an interface using Xtoolkit is much easier than developing the same interface using only X library.
Size Measurement of a Component-based GUI: Lines of code (LOC) is not an appropriate metric to estimate and measure the size of a component-based GUI. This is because the interface is developed by integrating several pre- built components. The different components making up an interface might have been in written using code of drastically different sizes. However, as far as the effort of the GUI developer who develops an interface by integrating the components may not be affected by the code size of the components he integrates. A way to measure the size of a modern user interface is

widget points (wp). The size of a user interface (in wp units) is simply the total number of widgets used in the interface. The size of an interface in wp units is a measure of the intricacy of the interface and is more or less independent of the implementation environment. The wp measure opens up chances for contracts on a measured amount of user interface functionality, instead of a vague definition of a complete system. However, till now there are no reported results to estimate the development effort in terms of the wp metric. An alternate way to compute the size of GUI is to simply count the number of screens. However, this would be inaccurate since a screen complexity can range from very simple to very complex

USER INTERFACE DESIGN METHODOLOGY

- At present, no step-by-step methodology is available which can be followed by rote to come up with a good user interface.
- What we present in this section is a set of recommendations which you can use to complement your ingenuity.

A GUI Design Methodology:

- The GUI design methodology we present here is based on the seminal work of Frank Ludolph.
- Our user interface design methodology consists of the following important steps:
  - Examine the use case model of the software.
  - Interview, discuss, and review the GUI issues with the end-users.
  - Task and object modeling.
  - Metaphor selection.
  - Interaction design and rough layout.
  - Detailed presentation and graphics design.
  - GUI construction.
  - Usability evaluation.

Examining the use case model

- The starting point for GUI design is the use case model.
- This captures the important tasks the users need to perform using the software.
- Metaphors help in interface development at lower effort and reduced costs for training the users.

- Metaphors can also be based on physical objects such as a visitor’s book, a catalog, a pen, a brush, a scissor, etc.
- A solution based on metaphors is easily understood by the users, reducing learning time and training costs.

Task and object Modeling:

- A task is a human activity intended to achieve some goals.
- Examples of task goals can be as follows:
  - Reserve an airline seat
  - Buy an item Transfer money from one account to another
  - Book a cargo for transmission to an address.
- A task model is an abstract model of the structure of a task.
- A task model should show the structure of the subtasks that the user needs to perform to achieve the overall task goal.
- Each task can be modeled as a hierarchy of subtasks.
- A task model can be drawn using a graphical notation similar to the activity network model.
- A user object model is a model of business objects which the end-users believe that they are interacting with.

- The objects in a library software may be books, journals, members, etc.

Metaphor selection:

- The first place one should look for while trying to identify the candidate metaphors is the set of parallels to objects, tasks, and terminologies of the use cases.
- If no obvious metaphors can be found, then the designer can fall back on the metaphors of the physical world of concrete objects.
- The appropriateness of each candidate metaphor should be tested by restating the objects and tasks of the user interface model in terms of the metaphor.

Interaction design and rough layout

- The interaction design involves mapping the subtasks into appropriate controls, and other widgets such as forms, text box, etc.
- This involves making a choice from a set of available components that would best suit the subtask.
- Rough layout concerns how the controls, and other widgets to be organized in windows.

Detailed presentation and graphics design

- Each window should represent either an object or many objects that have a clear relationship to each other.
- At one extreme, each object view could be in its own window. But, this is likely to lead to too much window opening, closing, moving, and resizing.
- At the other extreme, all the views could be placed in one window side-by-side, resulting in a very large window.
- This would force the user to move the cursor around the window to look for different objects.

GUI construction

- Some of the windows have to be defined as modal dialogs.
- When a window is a modal dialog, no other windows in the application are accessible until the current window is closed.
- When a modal dialog is closed, the user is returned to the window from which the modal dialog was invoked.
- Modal dialogs are commonly used when an explicit confirmation or authorization step is required for an action.

User interface inspection

- Nielson studied common usability problems and built a check list of points which can be easily checked for an interface. The following checklist is based on the work of Nielson:
  - Visibility of the system status
  - Match between the system and the real world
  - Undoing mistakes
  - Consistency
  - Recognition rather than recall
  - Support for multiple skill levels
  - Aesthetic and minimalist design
  - Help and error messages
  - Error prevention