Unit –3

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 *(Text Book 2)*
Function-Oriented Software Design: Overview of SA/SD methodology, Structured analysis, Developing the DFD model of a system, Structured design, Detailed design, and Design Review.
User Interface Design: Characteristics of a good user interface, Basic concepts, Types of user interfaces, Fundamentals of component-based GUI development, and user interface design methodology.

SOFTWARE DESIGN

In software design phase, the design document is produced, based on the customer requirements as documented in the SRS document. The design document produced at the end of the design phase should be implementable using a programming language in the subsequent (coding) phase.

1. OVERVIEW OF THE DESIGN PROCESS:

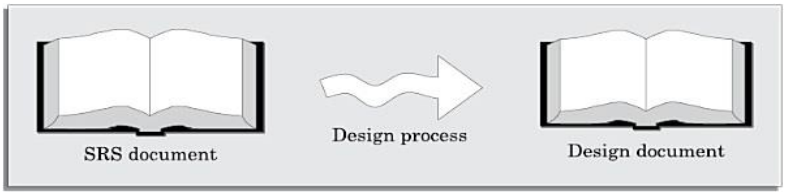


Figure 1: The design process

1.1 Outcome of the Design Process:

1. **Different modules required:**Each module is a collection of functions and the data shared by the functions of the module. Module should be named according to the task it performs. Ex: An academic automation s/w - Student registration.
2. **Control relationships among modules:** A control relationship between two modules essentially arises due to function calls across the two modules.
3. **Interfaces among different modules**: The interfaces between two modules identifies the exact data items that are exchanged between the two modules when one module invokes a function of the other module.
4. **Data structures of the individual modules**: Data structures for storing and managing the data of a module need to be properly designed and documented.
5. **Algorithms required to implement the individual modules:** The algorithms are designed and documented with the accuracy of the results, space and time complexities.

The design documents are reviewed by the members of the development team to ensure that the design solution conforms to the requirements specification.

1.2 Classification of Design Activities: Classify the design activities into 2 parts-

- Preliminary (or high-level) design - The outcome is called the *program structure or the s/w architecture*
- Detailed design - The outcome is usually documented in the form of a *module specification document*

1.3 Classification of Design Methodologies

Classify these Methodologies into Procedural and Object-oriented approaches. Design technique requires the designer to make many decisions and 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.

Analysis versus Design: The goal of analysis technique is to elaborate the customer requirements through careful thinking and the exact way the system is to be implemented.

2. HOW TO CHARACTERISE A GOOD SOFTWARE DESIGN?

Definition: A Good Software Design can vary depending on the application being designed.

Example: "*Memory size used up by a program*" in embedded applications are often required to work under severely limited memory sizes due to cost, space, or power consumption considerations.

The characteristics are listed below:

Correctness: Implement all the functionalities of the system.
Understandability: Easily understandable.
Efficiency: Should address resource, time, and cost optimisation issues.
Maintainability: It should be easy amenable to change.

Understandability of a Design: A Major Concern

- Need to choose the best one among a large number of design solutions. All incorrect designs have to be discarded first.
- A good design should help overcome the human cognitive limitations that arise due to *limited short-term memory.* A design that is easy to understand is also easy to develop and maintain. A complex design would lead to tremendous effort to implement, test, debug, and maintain it. 60% of the product cost is spent on maintenance. If the software is not easy to understand, development costs increases, the effort required to maintain the product also increases.

An understandable design is modular and layered: A designs should have the features

- It should assign consistent and meaningful names to various design components.
- It should be a *modular design*.
- It should arrange neatly the modules ina hierarchy. *E.g: Tree like diagrams*.

Modularity

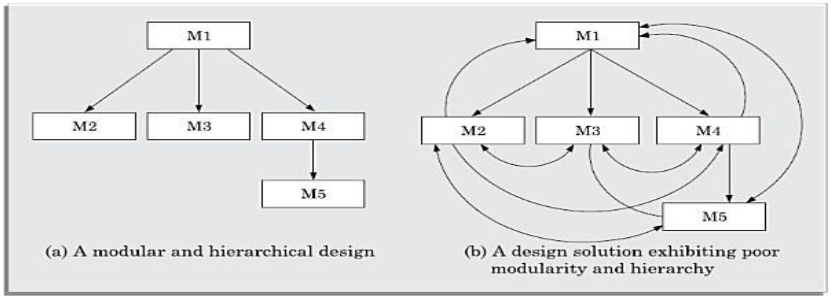


Figure 2: Two design solutions to the same problem

- A modular design (*Figure 2: a&b*), 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 with each other, then each module can be understood separately. This reduces *complexity* of the design solution greatly.

Layered design

- A layered design, modules are arranged in a hierarchy of layers and represented graphically and easily understandable, it would result in a *tree-like diagram.* A module (managers) can only invoke functions of the modules in the layer immediately below it. A layered design is a control abstraction, since a module at a lower layer is unaware of the higher layer modules.
- When a failure is detected, the modules below it can possibly be the source of the error. This greatly simplifies debugging and detect the error.

3. 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* between the two modules.
- **Functional independence:** A module that is *highly cohesive and low coupling* with other modules is said to be functionally independent of the other modules.
- **Error isolation:** Functional independence reduces the chances of the error propagation to the other modules. If a module is functionally independent, its interaction with other modules is low. If a module is not functionally independent, once a failure is detected the error can be potentially in any of the large number of modules.
- **Scope of reuse:** In a functionally independent module, the interfaces of the module with other modules are very few and simple. So the module can be reused in a different program.
- **Understandability:** When modules are functionally independent, complexity of the design is greatly reduced. Modules can be understood in isolation.

3.1 CLASSIFICATION OF COHESIVENESS:

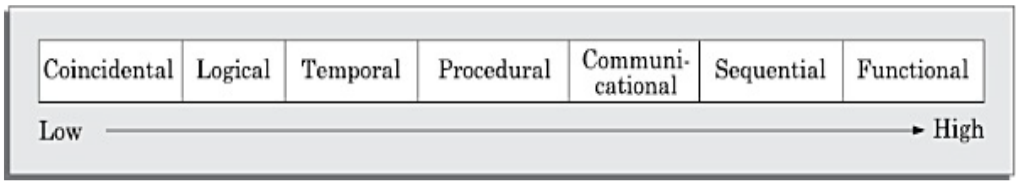


Figure 3.1: Classification of cohesion

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 classes of cohesion that modules can possess are depicted in Figure 3.1 These different classes of cohesion are elaborated below.

1. **Coincidental cohesion:** A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely.

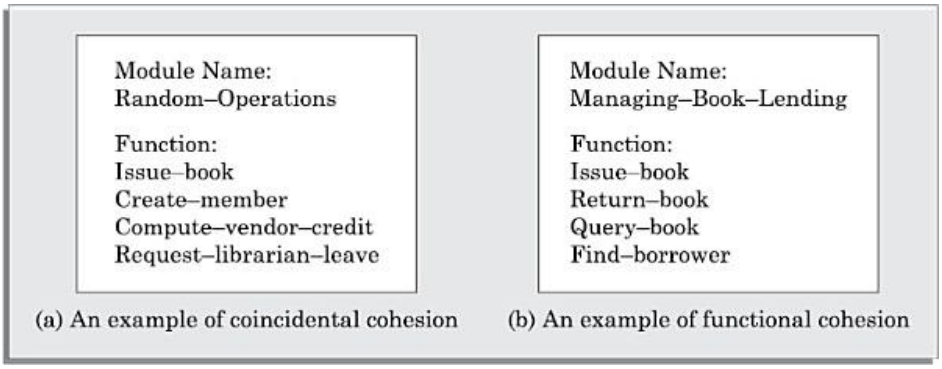


Figure 3.2: Examples of cohesion

*Figure 3.2(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.

2. **Logical Cohesion:** If all elements of the module perform similar operations, such as error handling, data input, data output, etc. Example: Consider a set of print functions for reports such as grade sheets, salary slips, annual reports, etc.
3. **Temporal Cohesion:** Functions are executed in the same time span, then the module is said to possess temporal cohesion. Example: When a computer is booted, initialisation of memory and devices, loading the operating system, etc.
4. **Procedural Cohesion:** If the set of functions of the module are executed one after the other. The functions login(), place-order(), check-order(), printbill().
5. **Communicational cohesion:** If all functions of the module refer to or update the same data structure. Example: Consider a module 'student' in which the different functions such as admitStudent, enterMarks, printGradeSheet, etc. access and manipulate data stored.
6. **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. Example: If the functions create-order(), check-item-availability(), placeorder_vendor() are placed in a single module, then the module would exhibit sequential cohesion.
7. **Functional cohesion:** If different functions of the module co-operate to complete a single task. Example: *Figure 3.2(b).* The functions issue-book(), return-book(), query-book(), & find-borrower(), together manage all activities concerned with book lending.

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.

3.2 CLASSIFICATION OF COUPLING:

- The coupling between two modules indicates the degree of interdependence between them. If two modules interchange large data, then they are highly interdependent or coupled. The *degree of coupling* between two modules depends on their interface complexity.

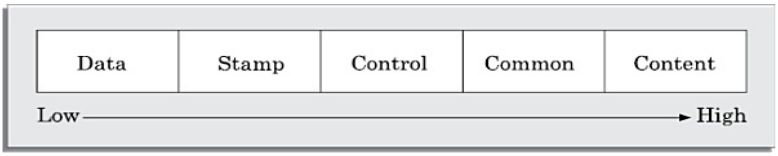


Figure 4: Classification of coupling.

1. **Data coupling:** Communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc.
2. **Stamp coupling:** Communicate using a composite data item such as a record in a structure in C.
3. **Control coupling:** Data from one module is used to direct the order of instruction execution in another. *Example:* Control coupling is a flag set in one module and tested in another module.
4. **Common coupling:** Two modules are common coupled, if they share some global data items.
5. **Content coupling:** Content coupling exists between two modules, if they share code. That is, a jump from one module into another module.

Figure 4: The degree of coupling increases from data coupling to content coupling. High coupling among modules makes a design solution difficult to understand and maintain.

4 LAYERED ARRANGEMENT OF MODULES:

- The *control hierarchy or program structure* represents the organisation of program components. The common notation is a *tree-like diagram* known as a *structure chart.* Warnier-Orr and Jackson's notations are not widely used.

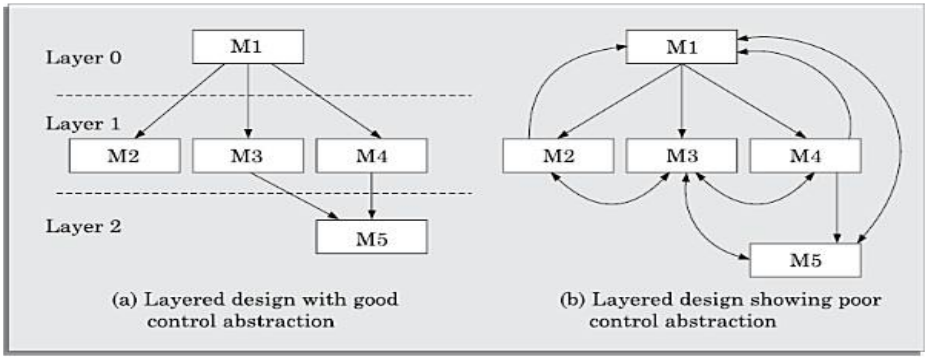


Figure 5: Examples of good and poor control abstraction.

- **Layering:** The modules are arranged in layers. A module is allowed to call only the modules that are at a lower layer. *Figure 5(a):* layered design, *(b)* design that is not layered. *Figure 5(b):* is actually not layered since all the modules can be considered to be in the same layer.
- A layered design achieves *control abstraction* and is easier to understand and debug. In a control hierarchy, a module that controls another module is said to be **Superordinate** to it. 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 visible to a module.
- **Control abstraction:** In a layered design, a module should only invoke the functions of the modules that are immediately below it.
- **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. Figure 5(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 5(a), the fan-out of the module M1 is 3. A design having high fan-out numbers is not a good design.
- **Fan-in:** Fan-in indicates the number of modules that directly invoke a given module. Figure 5(a), the fan-in of the module M1 is 0, M2 is 1, and that of M5 is 2.

5 APPROACHES TO SOFTWARE DESIGN:

- There are two fundamentally different approaches to software design, function-oriented design, and object-oriented design - used for development of large programs.

5.1 Function-oriented Design:

- In top-down decomposition, starting at a high-level view of the system, is successively refined into more detailed functions. *Example,* consider a function - create-new-library member. This high-level function may be refined into the following subfunctions:
  - assign-membership-number
  - create-member-record
  - print-bill
- Centralised system state: The system state is centralised and shared among different functions. Example, in the library management system, share data such as member-records for reference and updation:
  - create-new-member
  - delete-member
  - update-member-record
- Few well-established function-oriented design approaches are
  - Jackson's structured design by Jackson [1975]
  - Warnier-Orr methodology [1977, 1981]

5.2 Object-oriented Design:

- In the object-oriented design (OOD) approach, a system is viewed as a collection of objects (i.e. entities). Each object is associated with a set of functions that are called its methods. *Example*, in a 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

OBJECT - ORIENTED DESIGN VERSUS FUNCTION-ORIENTED DESIGN APPROACHES:

- Unlike FOD methods in OOD, the basic abstraction is *not the services* available to the users such as issuebook, display-book-details etc., but real-world entities such as member, book, etc.
- In OOD, state information exists in the form of *data distributed among several objects* of the system. In an OOD, these data are distributed among different employee objects of the system.
- FOD techniques group functions together if, as a group, they constitute a higher level function. OOD techniques group functions together on the basis of the data they operate on.

FOD approach:	**OOD approach:**
interrogate_detectors();	class detector
get_detector_location();	*attributes:* status, location, neighbours
determine_neighbour_alarm();	*operations:* create, sense-status, get-location,find-neighbours
determine_neighbour_sprinkler();	class alarm
ring_alarm();	*attributes:* location, status
activate_sprinkler();	*operations:* create, ring-alarm, get_location, resetalarm
reset_alarm();	class sprinkler
reset_sprinkler();	*attributes:* location, status
report_fire_location();	*operations:* create, activate-sprinkler, get_location, reset-sprinkler

FUNCTION-ORIENTED SOFTWARE DESIGN

- The term top-down decomposition is often used to denote the successive decomposition of a set of high-level functions into more detailed functions.

6.1 OVERVIEW OF SA/SD METHODOLOGY:

- SA/SD methodology involves carrying out two distinct activities:
  - Structured Analysis (SA)
  - Structured Design (SD)
- During SA, the SRS document is transformed into a data flow diagram (DFD) model.
- During SD, the DFD model is transformed into a structure chart.

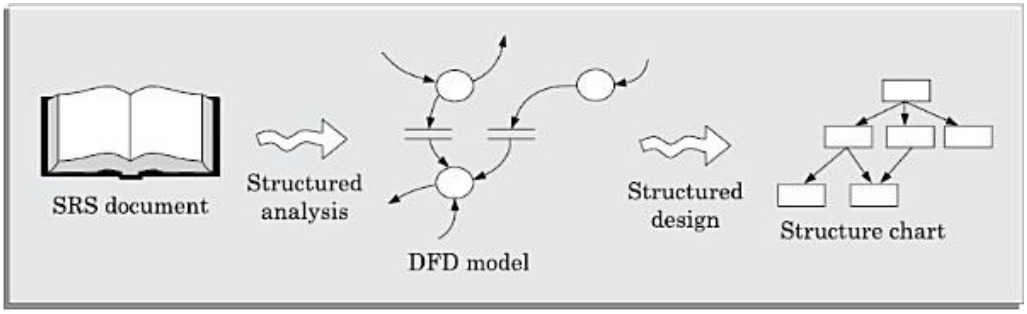


Figure 6.1: Structured analysis and structured design methodology

- *Figure 6.1:* The SA activity transforms the SRS document into a graphic model called the DFD model. During SA, functional decomposition of the system is achieved. SD, all functions identified during SA are mapped to a module structure or highlevel design or the s/w architecture for the given problem. This is represented using a structure chart.
- SA captures the detailed structure of the system as perceived by the user, whereas SD defines the structure of the solution that is suitable for implementation in some programming language

6.2 STRUCTURED ANALYSIS

- The major processing tasks (high-level functions) of the system are analysed, and the data flow among these processing tasks are represented graphically. This is based on the following principles:
  - Top-down decomposition approach.
  - Divide and conquer principle. Each function is decomposed into detailed functions.
  - Graphical representation of the analysis results using data flow diagrams (DFDs).
- 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.

6.2.1 Data Flow Diagrams (DFDs):

- The DFD terminology, each function is called a ***process or a bubble***. Each function as a processing station (or process) that consumes some i/p data and produces o/p data. The DFD (also known as the bubble chart) is a simple graphical formalism.
- DFD is a very simple to understand and use. A DFD model uses a primitive symbols (Figure 6.2) to represent the functions performed by a system and the data flow among these functions.

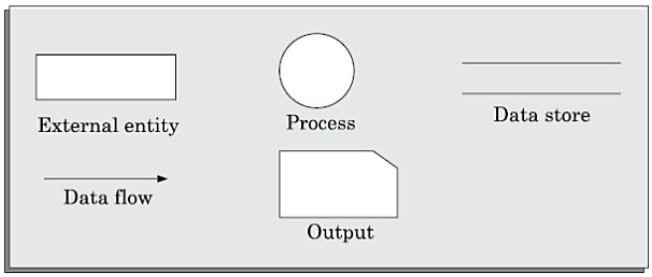


Figure 6.2: Symbols used for designing DFDs.

Primitive symbols used for constructing DFDs:

- **Function symbol:** A function is represented using a circle called a process or a bubble. Bubbles are annotated with the names of the corresponding functions.
- **External entity symbol:** An external entity is represented by a rectangle. The external entities interact with the system by inputting data or by consuming the data produced by the system.
- **Data flow symbol:** An arrow is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes in the direction of the data flow arrow. Data flow symbols are usually annotated with the corresponding data names.
- **Data store symbol:** A data store is represented using two parallel lines. Each data store is 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. Annotated with the name of the corresponding data items.
- **Output symbol:** The output symbol is used when a hard copy is produced.

Synchronous and asynchronous operations:

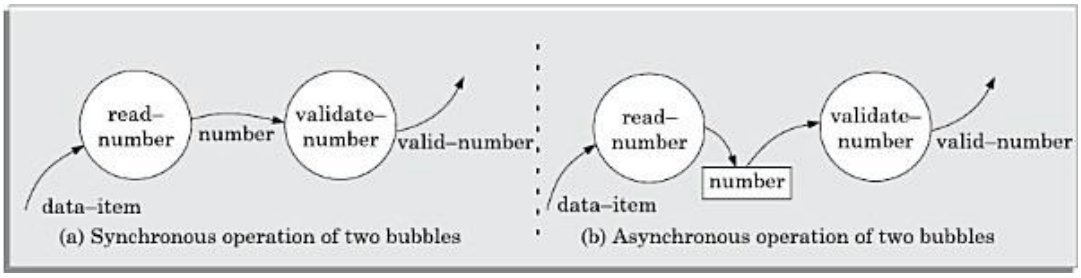


Figure 6.3: Synchronous and asynchronous data flow.

- If two bubbles are directly connected by a data flow arrow, then they are *synchronous* and they operate at the same speed shown in Figure 6.3(a). Here, the validate-number bubble can start processing only after the readnumber bubble has supplied data to it; and the read-number bubble has to wait until the validate-number bubble has consumed its data.
- However, if two bubbles are connected through a data store, as in Figure 6.3(b) then the speed of operation of the bubbles are independent. The data gets stored in the data store. The producer bubble stores data items, even before the consumer bubble consumes any of them.

Data dictionary:

- 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.
- A data dictionary lists all data items that appear in a DFD model. The data items listed include all *data flows* and the *contents of all data stores* appearing on all the DFDs in a DFD model. DFD model of a system consists of several DFDs, that is *level 0 DFD*, *level 1 DFD*, *level 2 DFDs, etc*., as shown in Figure 6.4

The dictionary plays a very important role in:

- 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.
- Most CASE tools usually capture the data items appearing in a DFD as the DFD is drawn, and automatically generate the data dictionary

Data definition:

- Composite data items can be defined in terms of primitive data items:
  +: denotes composition of two data items, e.g. a+b represents data a and b.
  [, ,]: represents selection e.g: [a,b] represents either **a** occurs or **b** occurs.
  ( ): bracket represent optional data which may or may not appear.
  e.g: a+(b) -- either a or a+b occurs.
  { }: represents iterative data definition,
  e.g. {name}5 represents five name data.
  {name}* represents zero or more instances of name data.
  = : represents equivalence, e.g. a=b+c means that a represents b and c.
  /* */: Anything appearing within /* and */ is considered as comment

6.3 DEVELOPING THE DFD MODEL OF A SYSTEM

- A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs.
- First the most abstract representation (highest level) of the problem is to be worked out. Subsequently, the lower level DFDs are developed. **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. All the data names appearing in all DFDs are in the data dictionary.

6.3.1 Context Diagram or Level 0 DFD

- The DFD model of a system is constructed by using a hierarchy of DFDs (Figure 6.4). The top level DFD is called the ***level 0 DFD*** or ***the context diagram***. It represents the entire system as **a single bubble**.
- Annotated with the name of the software system being developed (**usually a noun**). The bubbles at all other levels are annotated with **verbs** according to the main function performed by the bubble.

- To develop a higher-level DFD model, processes are decomposed into their subprocesses and the data flow among these subprocesses are identified.
- To develop the context diagram of the system, we have to analyse the SRS document to identify the different types of users who would be using the system and the kinds of data they would be inputting to the system and the data they would be receiving from the system. Users includes any external systems which supply data to or receive data.

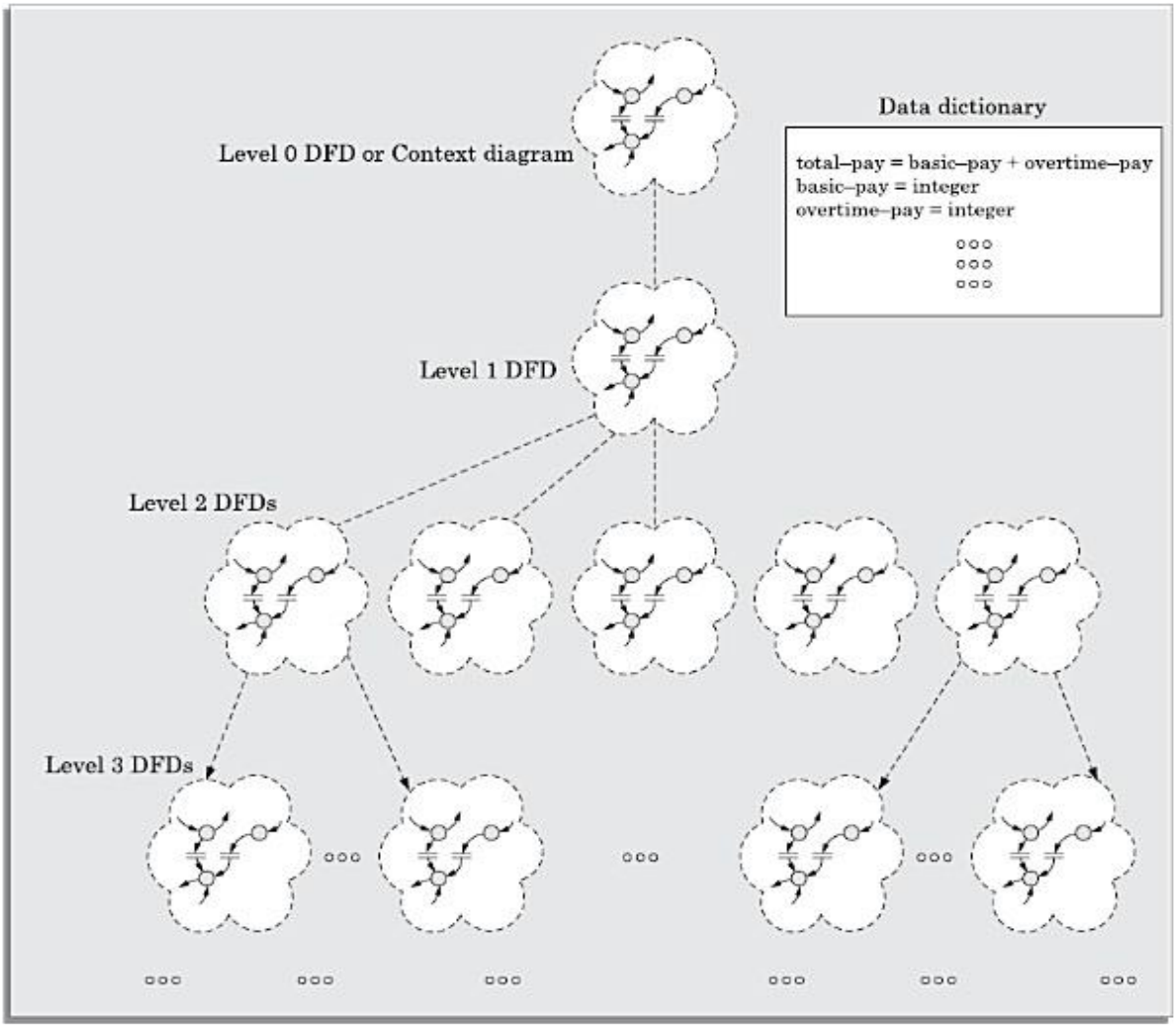


Figure 6.4: DFD model of a system consists of a hierarchy of DFDs and a single data dictionary

6.3.2 Level 1 DFD:

- The level 1 DFD usually contains 3 to 7 high-level functional requirements represented as bubbles in the level 1 DFD. Next, examine the input data and the data output as documented in the SRS document and represent them appropriately in the diagram.
- What if a system has more than 7 high-level requirements then *combine and represented as a single bubble in the level 1 DFD*. These can be splitted in the lower DFD levels. If a system has less than 3 high-level functional requirements, then *split into their subfunctions* so that we have roughly about 5 to 7 bubbles represented on the diagram.

- **Decomposition:**The bubbles are decomposed into subfunctions 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 3 to 7 bubbles.
  Developing the DFD model of a system more systematically by *Construction of context diagram, Construction of level 1 diagram, Construction of lower-level diagrams*
- **Numbering of bubbles:** The bubble at the context level is usually assigned the **number 0** to indicate that it is the 0 level DFD. Bubbles at level 1 are numbered, **0.1, 0.2, 0.3, etc**. When a bubble numbered x is decomposed, its children bubble are numbered **x.1, x.2, x.3, etc**.
- **Balancing DFDs:** The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as *balancing a DFD*. In Figure 6.5. level 1 DFD, d1 and d3 flow out of the bubble 0.1 and d2 flows into the bubble 0.1 (dotted circle). In the next level, bubble 0.1 is decomposed into three DFDs (0.1.1,0.1.2,0.1.3). The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in. Please note that dangling arrows (d1,d2,d3) represent the data flows into or out of a diagram.

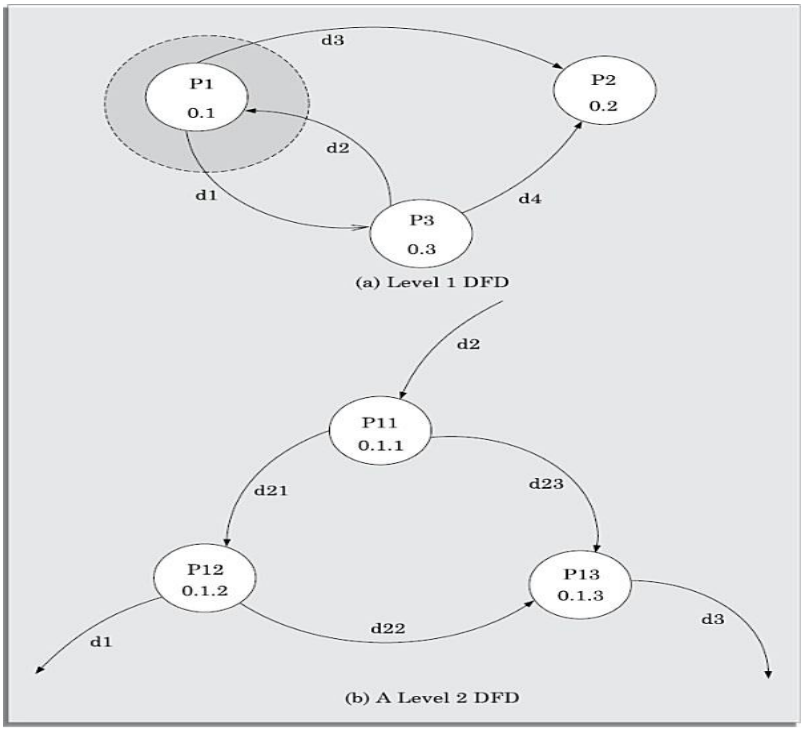


Figure 6.5: An example showing balanced decomposition

- **Commonly made errors while constructing a DFD model:** While modelling s/w problems using DFDs different types of mistakes that beginners usually make while constructing the DFD model.
  1. Many beginners commit the mistake of drawing more than one bubble in the context diagram. It should depict the system as a single bubble.
  2. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear in the DFDs at any other level.
  3. Only 3 to 7 bubbles per diagram should be allowed. DFD should be decomposed 3 to 7 bubbles in the next level. Many beginners leave the DFDs as unbalanced.

4. *Figure 6.6:* DFD represents only *data flow*, and it does not represent any *control information*.A data flow arrow should not connect two data stores. A data store is connected only to bubbles through data flow arrows.
5. All the functionalities of the system must be captured by the DFD model. The designer should not assume the functionalities which are not specified in the SRS document and then try to represent them in the DFD.
6. Incomplete data dictionary. The data and function names must be intuitive. Developers use meaningless symbolic data names.

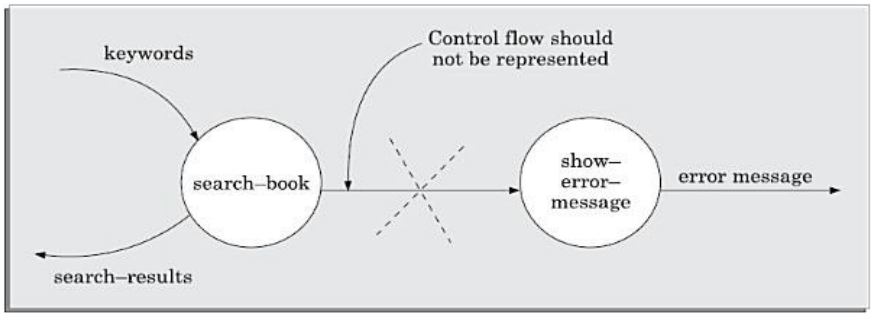
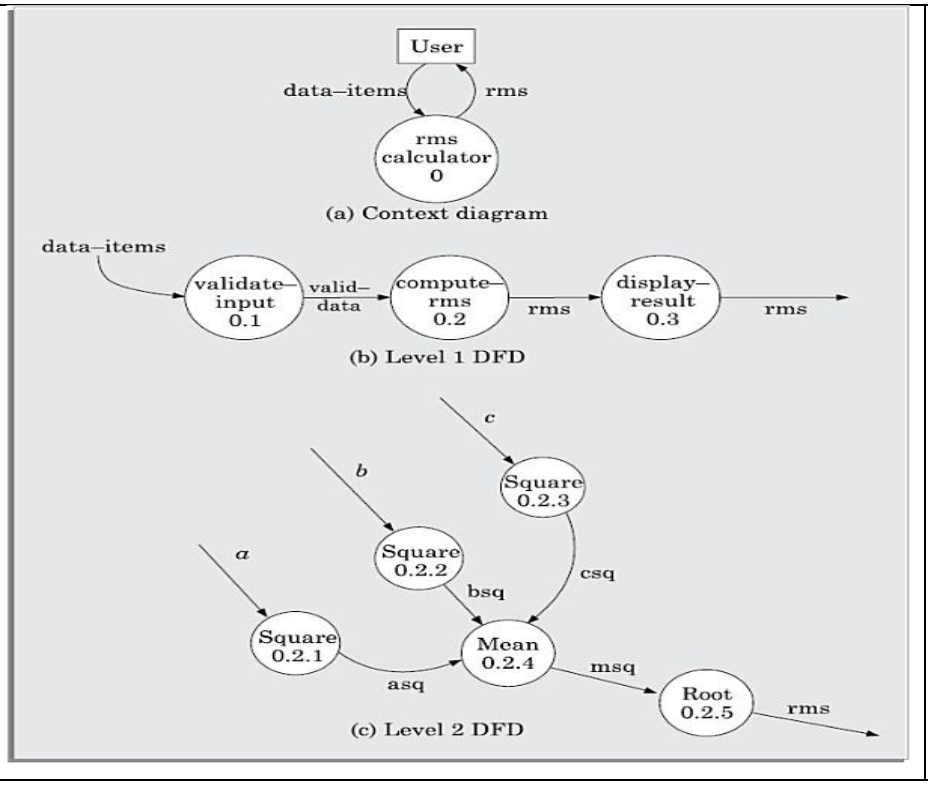


Figure 6.6: It is incorrect to show control information on a DFD.

- **Example 6.1 (RMS Calculating Software)** A software system called RMS calculating software would read three integral numbers from the user in the range of –1000 and +1000 and would determine the root mean square (RMS) of the three input numbers and display it.

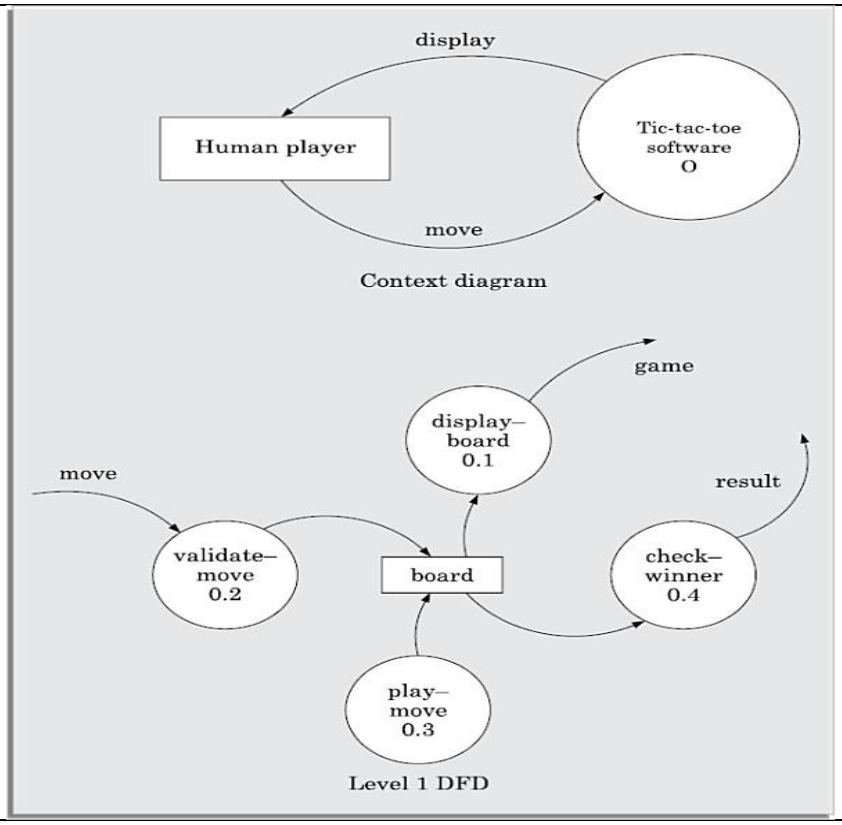


data-items: {integer}3
rms: float
valid-data: data-items
a: integer
b: integer
c: integer
asq: integer
bsq: integer
csq: integer
msq: integer

Figure 6.8: Context diagram, level 1, and level 2 DFDs for Example 6.1.

- **Example 6.2 (Tic-Tac-Toe Computer Game)** Tic-tac-toe is a computer game in which a human player and the computer make alternate moves on a 3 × 3 square. The player who is first to place

three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square wins. computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, and all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.



move: integer/*number b/w 1 to 9 */

display: game+result
game: board
board: {integer}9
result: [“computer won”,
“human won”,
“drawn”]

Figure 6.9: Context diagram and level 1 DFDs for Example 6.2.

Short comings of the DFD model:

- In the DFD model, a **short label** may not capture the entire functionality of a bubble..
- A DFD model does not **specify the order** in which the different bubbles are executed. Representation of such aspects is very important for modelling real-time systems.
- **Decomposition:** decomposition is carried out are highly subjective and depend on the choice and judgment of the analyst. Due to this reason, many times it is not possible to say which DFD representation is superior or preferable to another one.
- **Improper data flow diagram:** The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its subfunctions and we have to use subjective judgment to carry out decomposition.

6.4 STRUCTURED DESIGN

- The aim of **structured design** is to transform the results of the **structured analysis** into a **structure chart.** The structure chart representation can be implemented using some programming language. The basic building blocks used to design structure charts are:

- **Rectangular boxes:** represents a module. Annotated with the name of the module it represents.
- **Module invocation arrows:** An arrow connecting two modules implies that during program execution control is passed from one module to the other.
- **Data flow arrows:** These are small arrows appearing alongside the module invocation arrows. Annotated with the corresponding data name. Data flow arrows represent the fact that the named data passes from one module to the other in the direction of the arrow.
- **Library modules:** represented by a rectangle with double edges. when a module is invoked by many other modules, it is made into a library module.
- **Selection:** The diamond symbol represents that one module of several modules connected with the diamond symbol is 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.
- Structure Chart, there should be one and only one module at the top, called the **root**. There should be 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. As Lower-level modules can't invoke the high-level modules.

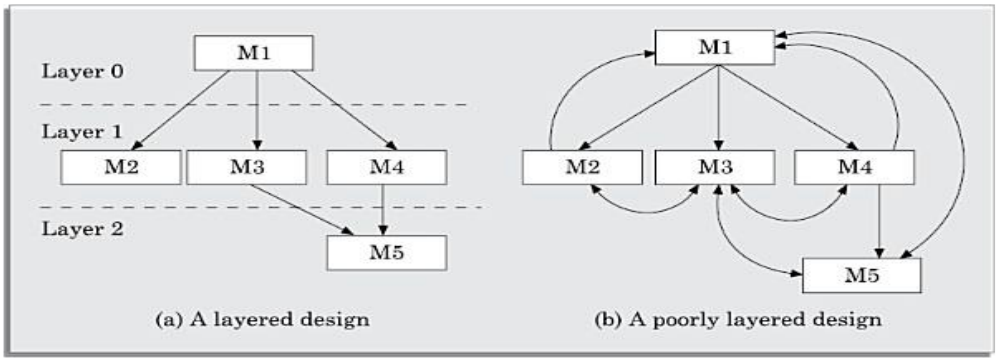


Figure 6.18: Examples of properly and poorly layered designs

6.4.1 Transformation of a DFD Model into Structure Chart

- Structured design provides two strategies for a DFD into a structure chart: First determine whether the *transform or the transaction analysis* is applicable to a particular DFD
- **Transform analysis** - The data input to the diagram are represented by **dangling arrows.** If all the data flow into the diagram are processed in similar ways then transform analysis is applicable. The first step in transform analysis is to divide the DFD into 3 types of parts: ***Input, Processing, Output***. Each input portion is called an *afferent branch.* Each output portion is called an *efferent branch.* The structure chart is derived by drawing one functional component each for the central transform, *the afferent and efferent branches* and refined by adding subfunctions required by each of the high-level functional components. This process of breaking functional components into subcomponents is called factoring. Factoring includes adding read and write modules, error-handling modules and initialisation.

Example 6.6 Draw the structure chart for the RMS software of Example 6.1.

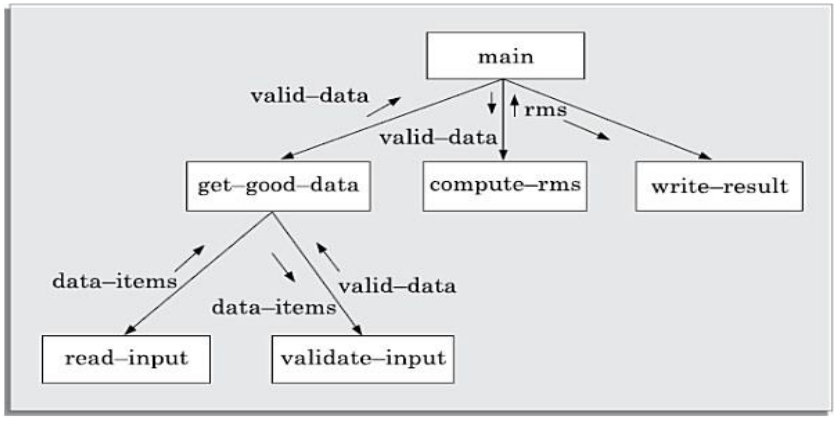


Figure 6.19: Structure chart for Example 6.6.

- **Transaction analysis:** This 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. In a transaction-driven system, different data items may pass through different computation paths through the DFD. A simple way to identify a transaction is the following. Check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. In the structure chart, draw a root module and below this module draw each identified transaction as a module.

- **Example 6.7** Draw the structure chart for the tic-tac-toe software of Example 6.2.

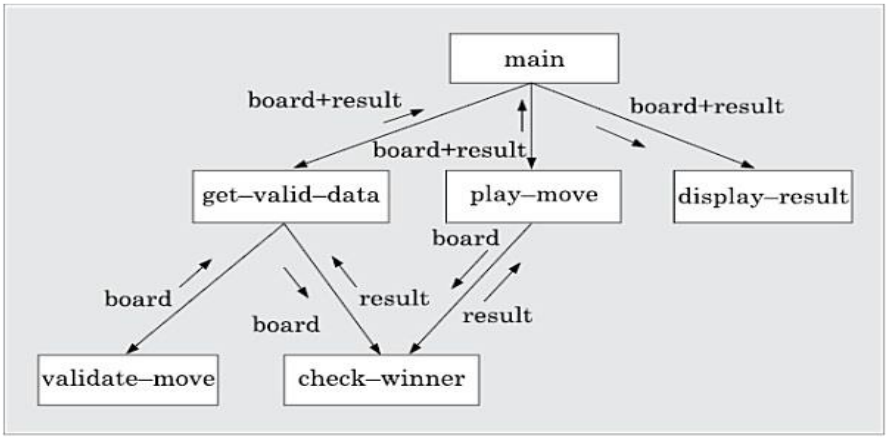


Figure 6.20: Structure chart for Example 6.7.

6.5 DETAILED DESIGN:

- During detailed design the pseudo code description and the data structures are designed for the different modules of the structure chart. These are usually described in the form of ***module specifications*** **(MSPEC).** The MSPEC for the non-leaf modules describe the conditions under which the responsibilities are delegated to the lowerlevel 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, refer to the DFD model and the SRS document to determine the functionality of the module.

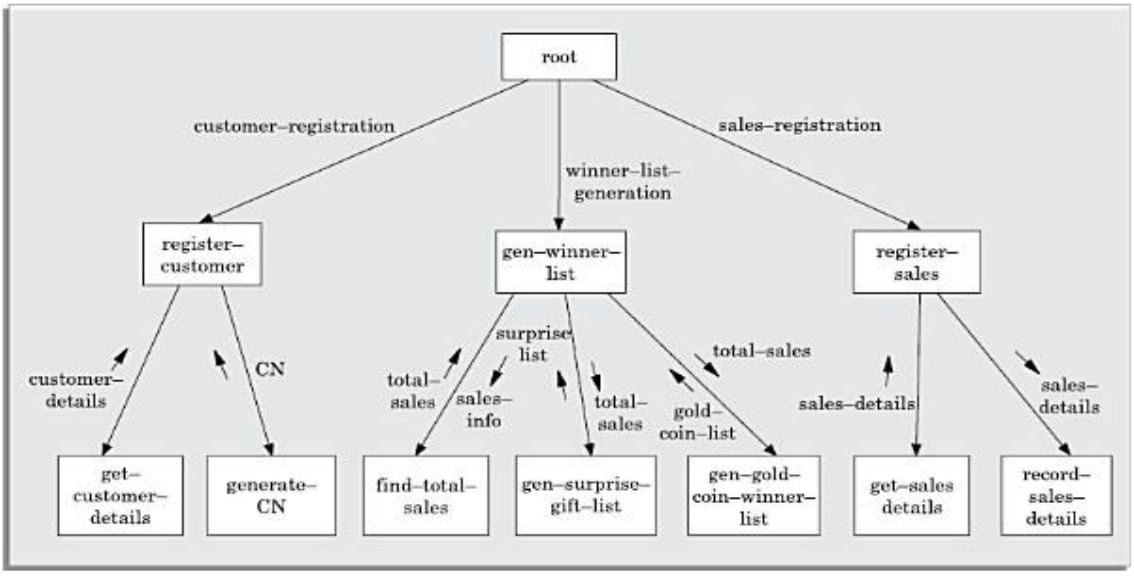


Figure 6.21: Structure chart for Supermarket Prize Scheme software

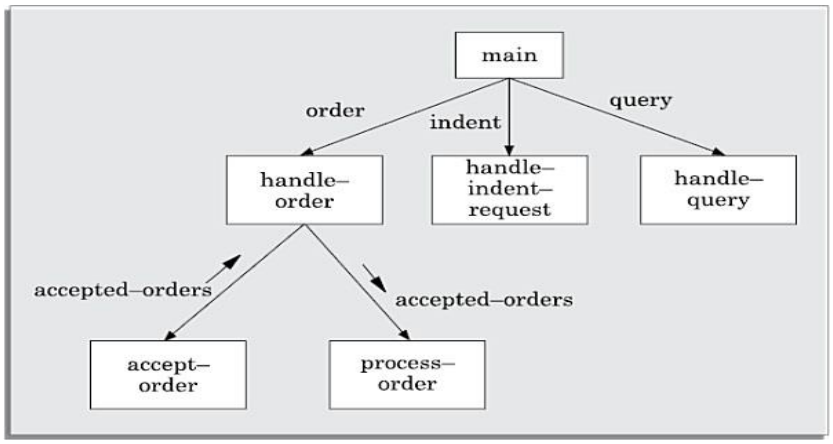


Figure 6.22: Structure chart for *trade-house automation system*.

6.6 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, who may or may not be the members of the development team. 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. They check whether each functional requirement in the SRS document can be traced to some bubble in the DFD model 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 be implemented.

USER INTERFACE DESIGN

The User Interface portion of a software product is responsible for all interactions with the user. An interface that is difficult to use leads to higher levels of user errors and ultimately leads to user dissatisfaction. User interface is designed and developed in a systematic manner and 50% of the total development effort is spent on developing the user interface part.

9.1 CHARACTERISTICS OF A GOOD USER INTERFACE

1. **Speed of learning:** A good user interface should not require its users to memorise commands. Three issues to enhance the speed of learning:
   - **Use of metaphors and intuitive command names:** Speed of learning an interface is facilitated if these are based on some day to- day real-life examples or physical objects with which the users are familiar. The abstractions of real-life objects used in UID are called metaphors. Popular metaphor is a shopping cart.
   - **Consistency:** A user learns a command, he should be able to use the similar commands in different circumstances for carrying out similar actions. The different commands supported by an interface should be consistent.
   - **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. Applications are developed using standard user interface components.
2. **Speed of use:** Speed of use of a UI is determined by the time and user effort to initiate and execute commands and referred to as productivity support of the interface. E.g, an interface that requires users to type in lengthy commands or involves mouse movements to different areas of the screen.
3. **Speed of recall:** Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures, and intuitive command names.
4. **Error prevention:** Consistency of names, issue procedures, and behaviour of similar commands and the simplicity of the command issue procedures minimise error possibilities.
5. **Aesthetic and attractive:** An attractive user interface catches user attention and fancy.
6. **Consistency:** The basic purpose of consistency is to allow users to generalise the knowledge about aspects of the interface from one part to another. Thus, consistency facilitates speed of learning, speed of recall, and also helps in reduction of error rate.
7. **Feedback:** A good user interface must provide feedback to various user actions. A novice user might even start recovery/shutdown procedures in panic.
8. **Support for multiple skill levels:** When someone uses an application for the first time, his primary concern is speed of learning. After periods of time, he becomes familiar with the operation of the software and his focus shifts from usability aspects to speed of command issue aspects. Experienced users look for options such as “hot-keys”, “macros”, etc.
9. **Error recovery (undo facility):** A good user interface should allow a user to undo a mistake committed by him while using the interface.
10. **User guidance and on-line help:** Users forget a command or are unaware of some features of the software then they seek guidance or help from the system.

9.2 BASIC CONCEPTS

- **9.2.1 User Guidance and On-line Help:** Users may seek guidance and error messages which are flashed automatically without the user asking for them. The guidance messages prompt the user regarding the options.

  On-line help system: 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 and advantage of any graphics and animation characteristics of the screen.

  Guidance messages: The guidance messages should be carefully designed to prompt the user about the next actions he might pursue. eg, a user using a command language interface might need a different type of guidance compared to a user using a menu or iconic interface .

  Error messages: Error messages are generated by a system either when the user commits some error or errors encountered by the system during processing due to some exceptional conditions. Error messages should be polite.
- **9.2.2 Mode-based versus Modeless Interface:**

  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. In a mode-based interface, different sets of commands can be invoked depending on the mode in which the system in. A mode-based interface can be represented using a state transition diagram.
- **9.2.3 Graphical User Interface (GUI) vs Text-based User Interface:**

  Let us compare various characteristics of a GUI with those of a textbased user interface: In a GUI multiple windows with different information can simultaneously be displayed on the user screen.

  Iconic information representation and symbolic information manipulation is possible in a GUI. Symbolic information manipulation such as dragging an icon representing a file to a trash

  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 special input devices such a mouse.

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

9.3 TYPES OF USER INTERFACES

User interfaces can be classified into three categories:

- Command language-based interfaces
- Menu-based interfaces
- Direct manipulation interfaces

9.3.1 Command Language-based Interface:

It is based on designing a command language which the user can use to issue the commands. command language-based interface allow users to compose complex commands by using a set of primitive commands , dramatically reduces the number of command names one would have to remember. Thus, it made concise requiring minimal typing by the user and allowing fast interaction with the computer. The command language interface allow most efficient command issue and can be implemented even on cheap alphanumeric terminals and it is easier to develop .

Drawbacks:Command language-based interfaces are difficult to learn. Most users make errors while formulating commands in the command language. For casual and inexperienced users, command language-based interfaces are not suitable.

Issues in designing a command language-based interface:

Command design issues are to reduce the number of primitive commands that a user has to remember and to minimise the total typing required.

- The designer has to decide what mnemonics (command names) to use for the different commands. eg, the shortest mnemonic should be assigned to the most frequently used commands.
- Letting a user define his own mnemonics for various commands is a useful feature, but it increases the complexity of user interface development.
- The ability to combine commands is a powerful facility in the hands of experienced users, but quite unnecessary for inexperienced users.

9.3.2 Menu-based Interface:

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. In this the typing effort is minimal as most interactions are carried out through menu selections using a pointing device. However, experienced users find the menu-based user interface to be slower than a command language-based interface because an experienced user can type fast and can get speed advantage by composing different primitive commands to express complex commands. If the number of choices is large, it is difficult to design a menu-based interfae. The following are some of the techniques available to structure a large number of menu items:

- Scrolling menu: When a full choice list is large and cannot be displayed within the menu area, scrolling of the menu items is required so that the user can easily locate a command that he needs.
- Walking menu: Walking menu used to structure a large collection of menu items. In this technique, 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 limited choices .
- Hierarchical menu: This type of menu is suitable for small screens. In this the menu items are organised in a hierarchy or tree structure. Selecting a menu item causes the current menu display to be replaced by an appropriate sub-menu.This can be used to manage large number of choices, but the users are likely to face navigational .

9.3.3 Direct manipulation interfaces:

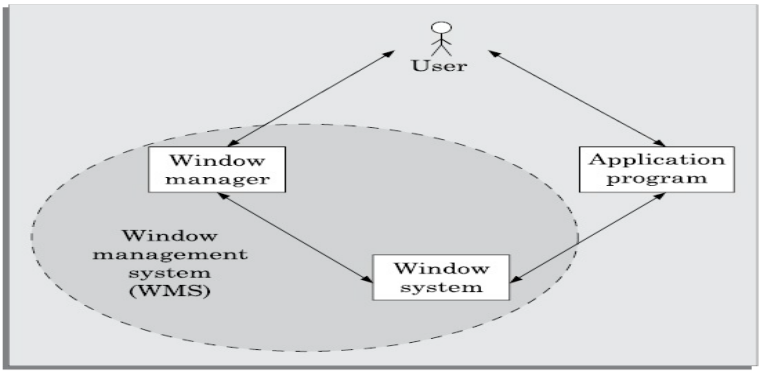
Direct manipulation interfaces present the interface to the user in theform of visual models (i.e., icons or objects). For this reason, directmanipulation interfaces are sometimes called as iconic interfaces. It is difficult to give complex commands using a direct manipulation interface.

9.4 COMPONENT-BASED GUI DEVELOPMENT:

- GUI became popular in the 1980s. There were very few GUI-based applications because the graphics terminals were too expensive. One of the first computers to support GUI-based applications was the Apple Macintosh computer. In those early days of GUI design, the user interface programmer typically started his interface development from the scratch. He would starting from simple pixel display routines, write programs to draw lines, circles, text, 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. The window system lets the application programmer create and manipulate windows without having to write the basic windowing functions.

- **9.4.1 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, it provides an interface to the user. A window is divided into two parts—client part, and non-client part. The client area is the area available to a client application for display. The non-client-part of the window determines the look and feel of the window. The window manager is responsible for managing and maintaining the non-client area of a window.
- **Window management system (WMS) :**
  A GUI consists of a large number of windows. it is necessary to manage these windows. These windows are managed through 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 and also provides the basic behaviour to the windows and provides several utility routines. 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.



A WMS consists of two parts a window manager and a window system.

Window Manager and Window System:

- The window manager, not the window system, determines how windows look and behave. Several window managers can be developed based on the same window system. The window manager is a special client that uses the services of the window system. The application programmer can also directly invoke these services to develop the user interface.
- The relationship between the window manager, window system, and application program shows that the end-user can interact either with the application or the window manager (for actions like resize and move), and both invoke services from the window system.
- The window manager is the component of the Window Management System (WMS) that the end-user interacts with. Developing user interfaces using the basic window system can be cumbersome,

so most systems provide a higher-level abstraction called widgets. A widget is a window object, which is a collection of related data and operations (such as size, location, background color) that can be manipulated (resize, move, draw, etc.).

- Widgets are the standard user interface components, and a user interface is made by integrating several widgets.

Component-based development:

A development style based on widgets is called component-based (or widget-based) GUI development style. It help users learn an interface fast. This style of development reduces the application programmer's work significantly.

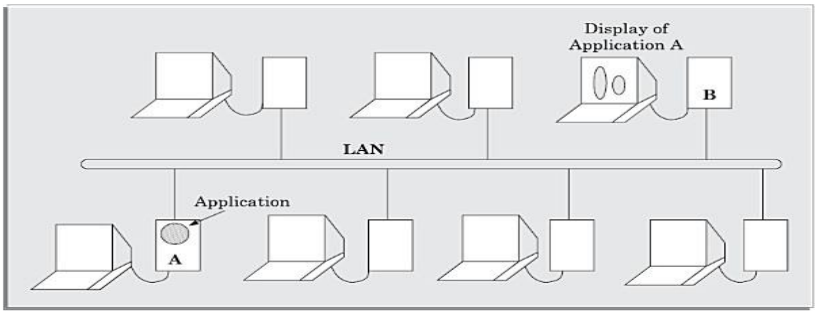
9.4.2 Types of Widgets:

Different interface programming packages support different widget sets. The following are generic widgets:

1. **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.
2. **Container widget:** These widgets are 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.
3. **Pop-up menu:** These are transient and task specific. A pop-up menu appears upon pressing the mouse button, irrespective of the mouse position.
4. **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.
5. **Dialog boxes:** A dialog box remains visible until explicitly dismissed by the user. A dialog box can include areas for entering text as well as values. Generally, these boxes ask you to read the information presented and then click OK to dismiss the box.
6. **Push button**: A push button contains key words or pictures that describe the action that is triggered when you activate the button.
7. **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.
8. **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.

9.4.3 An Overview of X-Window/MOTIF:

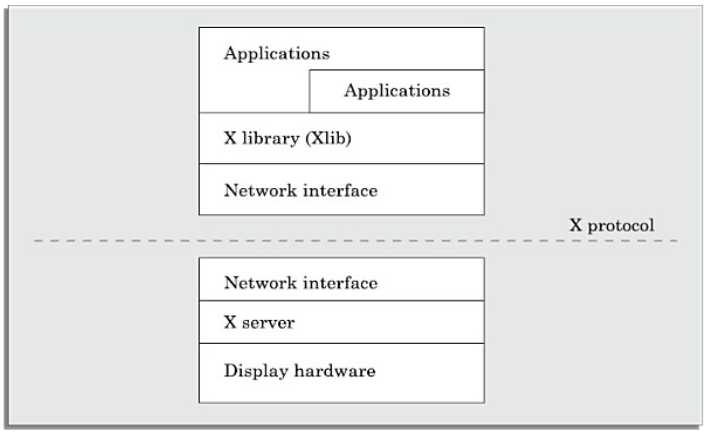
The applications developed using the X-window system are device independent and become network independent. Network-independent GUI operation has been schematically represented in below Figure:



- 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. 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.

9.4.4 X- Architecture:

The X architecture is pictorially depicted in below figure:



1. **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.
2. **X protocol:** The X protocol defines the format of the requests between client applications and display servers over the network.
3. **X library (Xlib):** The Xlib provides a set of about 300 utility routines for applications to call. Xlib provides low level primitives for developing an user interface.
4. **Xtoolkit (Xt):** The Xtoolkit consists of two parts: the intrinsics and the widgets. 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. Developing an interface using Xtoolkit is much easier than using X library.

9.4.5 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. The different components making up an interface might have been in written using code of drastically different sizes. 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. 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.

9.5 A USER INTERFACE DESIGN METHODOLOGY:

- GUI design methodologies are user-centered. User-centered does not mean design by users. One should not get the users to design the interface. Though users may have good knowledge but they may not know the GUI design issues.

9.5.1 Implications of Human Cognition Capabilities on User Interface Design:

HCI is how human cognitive capabilities & limitations influence the way an interface should be designed:

- **Limited memory:** Humans can remember at most seven unrelated items of information for short periods of time. It is the GUI designer's responsibility to anticipate what information the user will need at what point of each task and to ensure that the relevant information is displayed for the user to see.
- **Frequent task closure:** Doing a task requires doing several subtasks. the system should gives a clear feedback to the user that a task has been completed, so then user gets a sense of achievement and relief. The user can clear out information regarding the completed task from memory. This is known as task closure. When the overall task is big and complex, it should be divided into subtasks, each of which has a clear subgoal which can be a closure point.
- **Recognition rather than recall**: Information recall is memory burden, recognition of information from the alternatives shown is more acceptable.
- **Procedural versus object-oriented:** Procedural designs focus on tasks, giving them very few options for anything else. This approach is best applied where the tasks are narrow and well-defined or where the users are inexperienced, such as a bank ATM. An object-oriented focuses on objects. This allows the users a wide range of options. just remove redundent line and unnessary line, provde me same data as per author pov

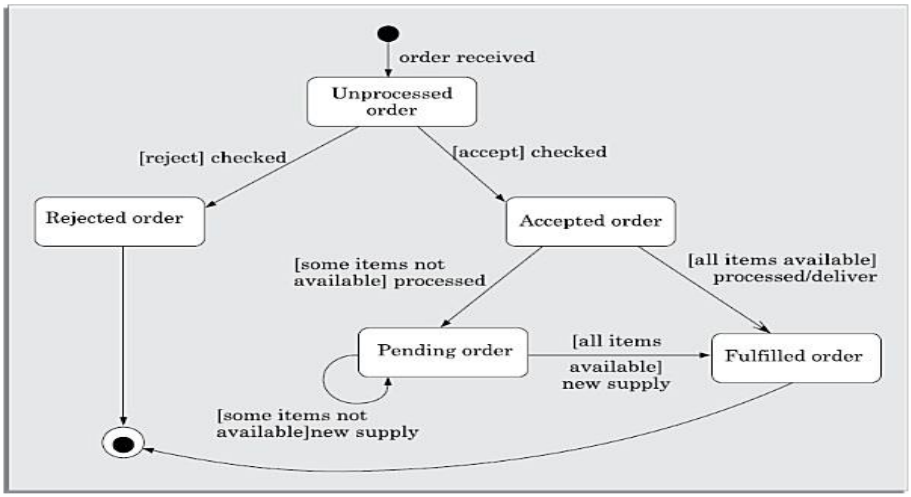
9.5.2 A GUI Design Methodology:

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:

1. **Examine the Use Case Model**: The design process begins by understanding the tasks users need to perform with the software through the use case model.
2. **User Engagement**: Interviewing and discussing GUI issues with end-users to gather input is essential for design.
3. **Task and Object Modelling**: Identifying tasks and objects that the interface must support is crucial.
4. **Metaphor Selection**: Metaphors play a vital role in GUI design by reducing training efforts and costs. They represent familiar solutions to commonly occurring situations, making the interface more intuitive.
5. **Interaction Design and Layout**: Developing the interaction flow and rough layout follows metaphor selection.
6. **Detailed Graphics Design**: Refining the presentation and graphics of the interface is the next step in making it visually appealing.
7. **GUI Construction and Usability Evaluation**: Finally, the actual construction of the GUI is done, followed by usability evaluations to ensure it meets user needs.

9.5.3 Task and object modelling:

- A Task model represents the structure of a task, showing how it is broken down into subtasks. Tasks are often modeled hierarchically, and can be illustrated using graphical notations, similar to an activity network. Tasks are represented as boxes, with lines indicating the breakdown into subtasks. An underlined task box indicates that no further decomposition is needed.
- A Object modeling is essential for designing object-based systems. This involves identifying the business objects that users interact with, such as books and journals in a library system, or items and bills in supermarket automation software. The user object model is key to understanding the objects the end-users engage with. A **state diagram** (similar to UML notation) is used to model the states of these objects. This diagram helps determine how interface elements, such as menu items, should behave depending on the state of the object. An example of this is the state chart diagram for an order object.



9.5.4 Metaphor selection:

- When identifying candidate metaphors for a user interface, the first place to look is the set of parallels to the objects, tasks, and terminology of the use cases. If no suitable metaphors are apparent, designers can consider metaphors from the physical world of concrete objects. The suitability of a metaphor should be tested by restating the user interface tasks and objects using that metaphor. Key criteria for judging metaphors include simplicity, clarity, coherence, and alignment with users' common sense. A metaphor should be intuitive and easy for users to understand—e.g., using scissors to glue items would be confusing.
- An example is provided for a web-based pay-order shop interface, where several metaphors could be applied to various parts of the process:
  - Items can be picked from racks and examined.
  - Users can click on items to view associated catalogs.
  - Related items can be stored in drawers of an item cabinet.
  - Information can be organized like a book, similar to how a semiconductor handbook organizes electronic components.
  - When users decide to purchase an item, they can place it in a shopping cart.

  These metaphors align with common real-world shopping experiences, making the interface easier for users to navigate.

9.5.5 Interaction design and rough layout:

- The interaction design involves mapping subtasks to appropriate controls and widgets, such as forms, text boxes, etc. This requires choosing the best components from a set of available options to suit each subtask. Rough layout focuses on how to organize these controls and widgets within windows.
- Detailed presentation and graphics design suggests that each window should represent either a single object or multiple objects that have a clear relationship. At one extreme, each object could have its own window, but this would lead to excessive opening, closing, moving, and resizing of windows. On the other hand, placing all views in one window side-by-side could create a very large window, forcing the user to move the cursor around to find different objects.
- GUI construction involves defining some windows as modal dialogs, which prevent access to other windows in the application until the current window is closed. After closing a modal dialog, the user is returned to the window that invoked it. Modal dialogs are commonly used when explicit confirmation or authorization is needed for an action (e.g., confirming a delete action). While modal dialogs are essential in some situations, overusing them reduces user flexibility, and sequences of modal dialogs should be avoided.

9.5.6 User interface inspection:

Nielsen's study on common usability problems and presents a checklist for evaluating interfaces based on Nielsen's work. The checklist includes the following principles:

1. **Visibility of the System Status**: The system should keep users informed about its status, check command has been received or if the system is functioning properly. E.g: the system should indicate when it's processing or if there's a delay.
2. **Match Between the System & the Real World**: The system should use language and concepts familiar to the user, avoiding system-oriented terminology. This makes the interface more intuitive for users.
3. **Undoing Mistakes**: Users should feel in control and be able to undo or redo actions. This empowers users to correct mistakes without feeling helpless.
4. **Consistency**: The interface should be consistent, meaning the same words, concepts, and operations should have the same meaning and behavior in different contexts, reducing confusion.
5. **Recognition Rather Than Recall**: Users should not have to remember information from previous screens. All necessary data and instructions should be visible on the screen for easy selection.
6. **Support for Multiple Skill Levels**: The system should accommodate users with varying levels of expertise by providing shortcuts for users, helping them perform tasks more efficiently.
7. **Aesthetic and Minimalist Design**: Screens and dialogs should avoid irrelevant or unnecessary information, as extra content can reduce the visibility and importance of the relevant details.
8. **Help and Error Messages**: These should be written in plain language, clearly describing the problem and providing constructive suggestions for resolving it.
9. **Error Prevention**: The system should minimize the likelihood of user errors. This can be achieved by limiting the options to valid choices (e.g., using dropdown lists for predefined values) and validating data before it's submitted, ensuring the correct format.

This checklist serves as a guide to improve usability by ensuring clarity, consistency, and user control, thereby enhancing the overall user experience.

AGILITY

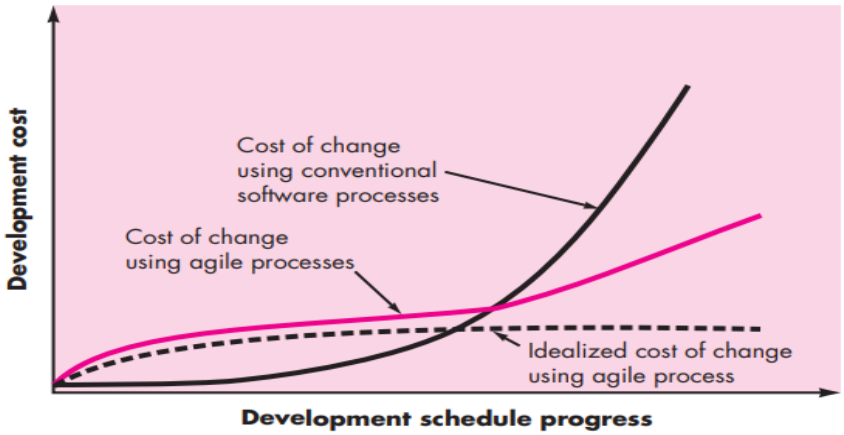
Agility: Agility and the Cost of Change, Agile Process, Extreme Programming (XP), Other Agile Process Models, Tool Set for the Agile Process

1. AGILITY:

Agility in software engineering refers to the ability to respond to change efficiently and effectively. Agile software engineering combines a philosophy and a set of development guidelines. The philosophy encourages customer satisfaction and early incremental delivery of software. The development guidelines are stress delivery over analysis and design and active and continuous communication between developers and customers.

2. AGILITY AND THE COST OF CHANGE:

The conventional wisdom in software development is that the cost of change increases non linearly as a project progresses. When a project is in its early phases changes are easy to make. If a major change is requested late in the project it can require major changes leading to high costs and delays.**The Agile methods aim is to reduce this cost.** Agile methods help "flatten" the cost curve, making it easier to accommodate changes even in later stages without extreme cost increases.



3. AGILE PROCESS:

An Agile software process addresses key assumptions about software projects:

- Requirements and customer priorities are unpredictable
- Design and construction are interleaved, meaning both should evolve together to validate design models.
- Development phases (analysis, design, construction, testing) are not always predictable.

To manage unpredictability, an agile process must be adaptable and incremental. Continual adaptation without progress is ineffective, so teams must make small, regular improvements. Customer feedback is crucial and should be gathered through working prototypes or system increments. These increments must be delivered in short cycles, allowing for continuous evaluation, feedback, and necessary adaptations.

3.1 Agility Principles:

The Agile Alliance ([Agi03], [Fow01]) defines 12 principles for achieving agility:

1. Customer satisfaction is the top priority
2. Accept changing requirements, even late in development, to provide a competitive advantage.
3. Deliver working software frequently, preferring shorter timescales (weeks to months).
4. Encourage daily collaboration between business stakeholders and developers.
5. Simplicity—focusing on essential work and minimizing waste—is key.
6. Build teams around motivated individuals, providing them with the necessary environment, support, and trust.
7. Face-to-face communication is the most effective way to share information.
8. Progress is measured by working software rather than documentation or plans.
9. Promote sustainable development with a consistent work pace for sponsors, developers, and users.
10. Prioritize technical excellence and good design to enhance agility.
11. Self-organizing teams produce the best architectures, requirements, and designs.
12. Regular reflection and adaptation help teams improve continuously.

Not all agile models apply these principles equally, but they define the core agile mindset present in various process models.

3.2 The Politics of Agile Development:

The benefits and applicability of agile development Vs traditional SE humorously captures the extremes:

- Agilists see traditional methodologists as prioritizing documentation over working systems.
- Traditionalists view agile practitioners as undisciplined hackers unable to scale their methods.

How to achieve agility effectively while ensuring software meets current and future customer needs. The solution is even within agile practices, multiple process models exist, each with unique approaches. Many agile concepts are simply adaptations of good software engineering principles. The best approach is to learn from both schools rather than dismiss either.

3.3 Human Factors:

Human factors highlight that agile processes should adapt to the team. The traits necessary for an effective agile team include:

1. *Competence* – Team members should have the required skills, knowledge, and talent for both software development and agile processes.
2. *Common Focus* – Everyone should be aligned towards delivering a working software increment within the promised time while adapting the process as needed.
3. *Collaboration* – Effective teamwork involves sharing information, communicating with stakeholders, and building software that delivers business value.
4. *Decision-making ability* – Agile teams should have autonomy to make their own technical and project-related decisions.
5. *Fuzzy problem-solving ability* – Teams must be comfortable with ambiguity and change, learning from each challenge, even if priorities shift.
6. *Mutual trust and respect* – A strong, "jelled" team is more than the sum of its parts, working cohesively towards success.

7. *Self-organization* – The team decides how to work, organizes its process, and sets its schedule, which improves morale and efficiency.

Agile teams thrive on adaptability, collaboration, and autonomy, making them more efficient and motivated in delivering quality software.

4. EXTREME PROGRAMMING (XP):

Extreme Programming is the most widely used approach for agile software development. It was Developed in the late 1980s,with significant contribution from Kent Beck. Later a variant called Industrial XP (IXP) was introduced, refining XP to better suit large organizations

4.1 XP Values:

Kent Beck defines a set of five values that establish a foundation for all work performed as part of XP—communication, simplicity, feedback, courage, and respect. Each of these values is used as a driver for specific XP activities, actions, and tasks.

1. Communication

- Encourages close, informal collaboration between developers and customers.
- Uses metaphors to simplify complex concepts.
- Reduces reliance on extensive documentation, favoring direct interaction.

2. Simplicity

- Developers design only for immediate needs, not future requirements.
- Keeps design simple and easy to implement.
- Uses refactoring to improve design when necessary.

3. Feedback: Comes from three sources:

  - Software tests (unit tests verify functionality).
  - Customers (via user stories & acceptance tests).
  - Team members (collaborative planning and iteration reviews).

- Helps refine design and estimate cost & schedule impact.

4. Courage (Discipline)

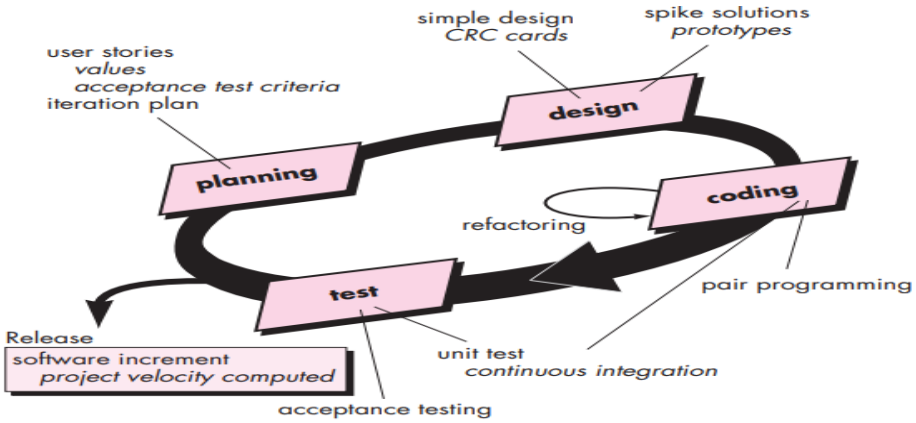
- Avoids over-engineering by resisting the urge to design for future needs.
- Accepts that future changes may require redesign, rather than making premature assumptions.

5. Respect

- Grows as the team delivers working software consistently.
- Encourages mutual respect among developers, stakeholders, and the process itself.

4.2 The XP Process:

- Extreme Programming (XP) follows an object-oriented approach and consists of four framework activities: planning, design, coding, and testing. Below is a summary of key XP activities:



Planning: The process starts with requirements gathering where the XP team understands the business context. Customers create user stories describing features and functionality, assigning priority to each. The development team estimates the cost in development weeks. If a story exceeds three weeks, it is split into smaller stories. Stories are prioritized based on immediate implementation, business value or risk. Once the first release is delivered, project velocity—the number of customer stories implemented—is calculated to refine estimates for future releases. Customers can modify stories, and the plan is adjusted accordingly.

Design: XP adheres to the "Keep It Simple" (KIS) principle, designing only what is needed for current requirements. CRC (Class-Responsibility-Collaborator) cards are used for object-oriented design. Complex design issues trigger the creation of spike solutions—prototypes used to mitigate risk and validate estimates. XP encourages refactoring, a disciplined approach to improving design without altering functionality. Since XP avoids heavy documentation, design evolves continuously through coding and refactoring.

Coding: Before coding begins, unit tests are created to validate the functionality of each user story. Developers write only the necessary code to pass these tests, maintaining simplicity. XP employs pair programming, where two developers work together at one workstation to improve problem-solving and code quality. Continuous integration ensures compatibility and early detection of issues.

Testing: Unit tests are automated to facilitate frequent execution and regression testing. Testing occurs daily, integrating unit tests into a universal testing suite to monitor progress. Acceptance tests, defined by the customer, validate overall system functionality and ensure implemented user stories meet requirements.
By maintaining simplicity, constant feedback, and close collaboration, XP promotes adaptability and high-quality software development.

4.3 Industrial XP:
Joshua Kerievsky describes IXP as an evolution of XP, maintaining its minimalist, customer-centric, test-driven approach while expanding management inclusion, customer roles, and technical practices. IXP introduces six key practices to support large-scale projects:

- **Readiness Assessment** – Evaluates the organization's preparedness, including development environment, stakeholder involvement, quality programs, cultural adaptability, and broader project community.
- **Project Community** – Expands the concept of a "team" into a "community," incorporating stakeholders like legal staff, quality auditors, and sales representatives. Clear roles and communication mechanisms are established.
- **Project Chartering** – Ensures business justification and alignment with organizational goals while assessing its impact on existing systems.
- **Test-Driven Management** – Defines measurable milestones ("destinations") and establishes mechanisms to track progress.
- **Retrospectives** – Conducts post-increment technical reviews to analyze lessons learned and improve the IXP process.
- **Continuous Learning** – Encourages team members to acquire new skills and techniques to enhance software quality.

Modified XP Practices in IXP:

- Story-Driven Development (SDD) – Acceptance test stories are written before coding begins.
- Domain-Driven Design (DDD) – Evolving domain models accurately represent expert knowledge.
- Pairing – Expands pair programming to managers and stakeholders for better knowledge sharing.
- Iterative Usability – Advocates evolving interface design based on real user feedback rather than upfront design.

4.4 The XP Debate:

Extreme Programming (XP) has sparked both discussion and debate. Stephens and Rosenberg argue that while many XP practices are valuable, some are overhyped or problematic.

Key concerns raised by critics include:

- **Requirements Volatility** – Frequent changes due to informal customer input can lead to scope creep and rework. Proponents argue this is inevitable in any process and that XP includes mechanisms to manage scope changes.
- **Conflicting Customer Needs** – Multiple customers may have differing requirements, and XP teams must resolve these conflicts, often beyond their authority.
- **Informal Requirements** – XP relies on user stories and acceptance tests, which critics argue lack formal models to catch omissions and inconsistencies early. Proponents claim rigid models become obsolete quickly.
- **Lack of Formal Design** – XP downplays architectural design, which critics argue is crucial for complex systems to ensure quality and maintainability. XP supporters believe incremental development reduces complexity and the need for extensive upfront design.

Every software process has flaws, but many organizations have successfully used XP by adapting it to their specific needs.

5. OTHER AGILE PROCESS MODELS:

Agile software development has introduced multiple process models, each competing for acceptance. While Extreme Programming (XP) is the most widely used, several other agile models are also in practice.

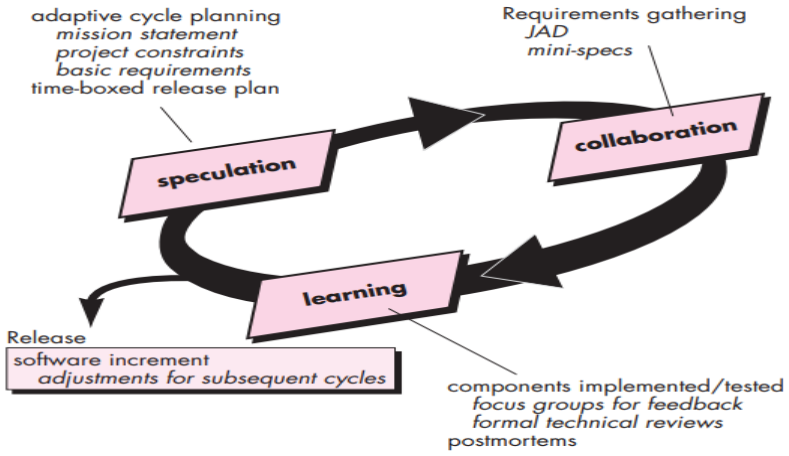
5.1 ADAPTIVE SOFTWARE DEVELOPMENT (ASD):

Proposed by Jim Highsmith, Adaptive Software Development (ASD) is designed for building complex software and systems, emphasizing human collaboration and self-organizing teams. Highsmith argues that an agile, adaptive approach based on collaboration provides structure in complex environments, similar to traditional discipline and engineering.

ASD follows a three-phase life cycle: speculation, collaboration, and learning.

- Speculation – The project is initiated with adaptive cycle planning, using the customer's mission statement, constraints, and basic requirements to define software increments (release cycles). Plans are continuously reviewed and adjusted based on progress.
- Collaboration – ASD relies on teamwork and trust, where individuals contribute creatively while ensuring:
    1. Constructive criticism without animosity.
    2. Mutual assistance without resentment.
    3. Shared work ethic and skill competency.
    4. Effective communication of issues for timely action.
- Learning – Teams focus on continuous learning alongside development to improve understanding of technology, process, and project goals. Learning is facilitated through focus groups, technical reviews, and project postmortems.

ASD's philosophy, emphasizing self-organizing teams, collaboration, and continuous learning, enhances the success of software projects, making it applicable to various agile methodologies



5.2 SCRUM:

Scrum, an agile software development method, was conceived by Jeff Sutherland and his team in the early 1990s, with further refinement by Schwaber and Beedle . It follows agile principles and consists of five framework activities: requirements, analysis, design, evolution, and delivery. Work within these activities is organized into sprints, adaptable iterations that respond to real-time project needs.

Scrum uses process patterns effective for projects with tight deadlines and evolving requirements:

- **Backlog** – A prioritized list of project requirements or features providing business value. Items can be added at any time, with priorities updated by the product manager.
- **Sprints** – Time-boxed iterations (typically 30 days) where team members complete specific backlog items. No new work is introduced during a sprint to maintain stability.
- **Scrum Meetings** – Daily 15-minute stand-ups where team members answer three key questions:
    1. What have you done since the last meeting?
    2. What obstacles are you encountering?
    3. What do you plan to accomplish before the next meeting?

    A Scrum Master leads the meeting, ensuring progress and identifying potential issues early. These meetings encourage team collaboration and self-organization.
- **Demos –** At the end of a sprint, the software increment is presented to the customer for evaluation. The demo may not include all planned features but showcases functional progress.

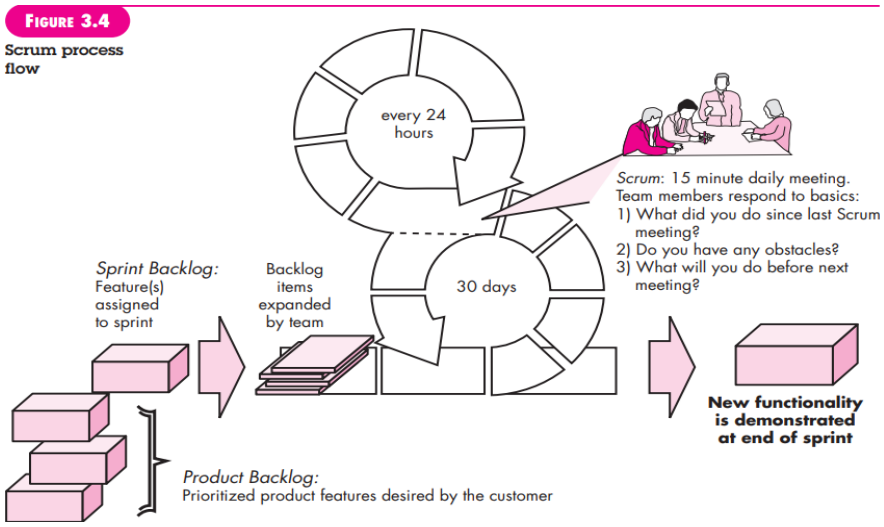


FIGURE 3.4 Scrum process flow

5.3 DYNAMIC SYSTEMS DEVELOPMENT METHOD (DSDM):

The Dynamic Systems Development Method (DSDM) is an agile software development approach that emphasizes incremental prototyping within controlled time constraints. Based on a modified Pareto principle, it suggests that 80% of an application can be delivered in 20% of the total development time, with remaining details refined later as requirements evolve.

The DSDM Consortium maintains the methodology and defines its life cycle, which includes two initial activities followed by three iterative cycles:

DSDM Life Cycle Stages

1. Feasibility Study – Determines business requirements and constraints, assessing the project's viability for DSDM.
2. Business Study – Defines functional and information requirements, application architecture, and maintainability needs.
3. Functional Model Iteration – Develops incremental prototypes demonstrating functionality, gathering user feedback for refinements.
4. Design and Build Iteration – Enhances prototypes to ensure engineering integrity and business value. Often runs concurrently with the previous stage.

5. Implementation – Deploys the latest software increment into production. The system may still evolve, requiring a return to the functional model iteration for further refinements.

Combining DSDM with Other Agile Models:

DSDM can be integrated with XP (Extreme Programming) to provide a structured process model alongside practical engineering practices. Additionally, collaboration and self-organizing teams from Adaptive Software Development (ASD) can be incorporated into DSDM for a flexible, hybrid approach.

5.4 CRYSTAL:

The Crystal family of agile methods was developed by Alistair Cockburn and Jim Highsmith to prioritize maneuverability in software development. Cockburn describes the process as a resource-limited, cooperative game focused on delivering useful, working software while preparing for future development cycles.

To achieve this flexibility, the Crystal family consists of multiple methodologies, each tailored to different project sizes, complexities, and team structures. While all Crystal methods share core principles, they differ in roles, process patterns, work products, and practices, allowing teams to choose the most suitable approach for their specific environment.

5.5 FEATURE-DRIVEN DEVELOPMENT (FDD):

Feature-Driven Development (FDD) was originally developed by Peter Coad as a practical model for object-oriented software engineering. It was later refined by Stephen Palmer and John Felsing into an adaptive, agile process suited for moderately sized and large projects.

FDD follows three key principles:

1. Collaboration – Encourages teamwork among developers, stakeholders, and customers.
2. Feature-Based Development – Breaks problems into small, manageable features, integrating them incrementally.
3. Clear Communication – Uses verbal, graphical, and text-based methods for technical clarity.

FDD defines five core activities (referred to as "processes" in FDD):

1. Develop an Overall Model
2. Build a Feature List
3. Plan by Feature
4. Design by Feature
5. Build by Feature

Project Management in FDD:

FDD places a strong emphasis on project management and tracking, more so than many agile methodologies. To ensure proper scheduling and monitoring, FDD defines six milestones for each feature:

1. Design Walkthrough
2. Design
3. Design Inspection
4. Code
5. Code Inspection
6. Promote to Build

FDD ensures that large-scale projects remain organized, manageable, and trackable, making it ideal for structured yet flexible development environments.

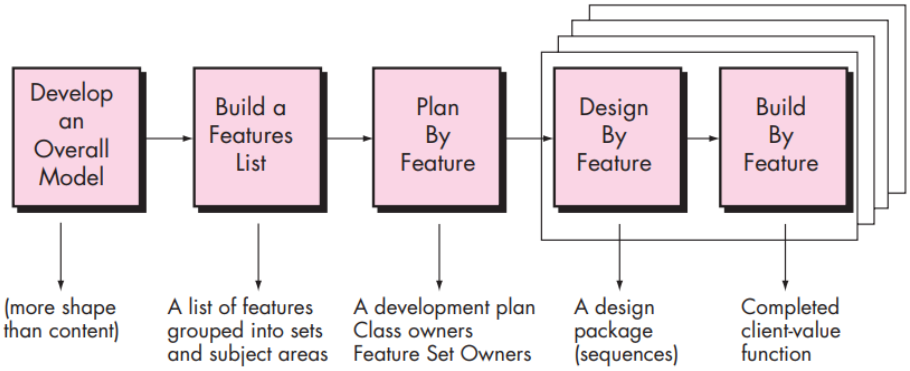


FIGURE 3.5
Feature Driven Development [Coa99] (with permission)

5.6 LEAN SOFTWARE DEVELOPMENT (LSD):

Lean Software Development (LSD) applies lean manufacturing principles to software engineering. The core principles of LSD, as outlined by Poppendieck
include:

1. Eliminate Waste – Avoid unnecessary features, streamline processes, and improve efficiency.
2. Build Quality In – Integrate quality assurance throughout development rather than relying on late-stage testing.
3. Create Knowledge – Encourage continuous learning and improvement.
4. Defer Commitment – Delay irreversible decisions until sufficient data is available.
5. Deliver Fast – Prioritize rapid delivery of valuable software increments.
6. Respect People – Empower teams and value stakeholder collaboration.
7. Optimize the Whole – Focus on the entire system rather than isolated parts.

Eliminating Waste in Agile Software Projects

According to Das ,waste reduction in software development involves:

- Avoiding unnecessary features.
- Evaluating cost and schedule impact of new requirements.
- Removing redundant steps in the development process.
- Improving information access for team members.
- Enhancing testing strategies to detect defects early.
- Reducing decision-making delays.
- Streamlining communication among stakeholders.

5.7 AGILE MODELING (AM):

Agile Modeling (AM), developed by Scott Ambler , offers a lightweight, practice-based approach to software modeling and documentation.

Ambler describes Agile Modeling (AM) as a collection of values, principles, and practices that help teams model software in an effective, streamlined manner. It follows agile principles, emphasizing adaptability and collaboration with business experts and stakeholders.

Key Principles of Agile Modeling

1. Model with a Purpose – Every model should serve a specific goal (e.g., communicate with customers or clarify software aspects). The notation and level of detail depend on this goal.

2. Use Multiple Models – Different models provide different insights. Only those that add real value should be used.
3. Travel Light – Keep only models that offer long-term value; discard unnecessary ones to avoid slowing down the team.
4. Content Over Representation – A model's value lies in the information it conveys, not its syntactical perfection.
5. Know Your Models and Tools – Understand the strengths and weaknesses of each modeling approach and tool.
6. Adapt Locally – Adjust modeling practices based on team needs and project requirements.

5.8 AGILE UNIFIED PROCESS (AUP):

The Agile Unified Process (AUP) follows a "serial in the large, iterative in the small" approach ,combining the phased structure of the Unified Process (UP) with agile iterations. It retains UP's four phases—inception, elaboration, construction, and transition—to provide a structured process flow while ensuring agility within each phase to deliver software increments quickly.

Each iteration of AUP includes the following activities :

- Modeling – Uses UML representations to define business and problem domains. Models are kept "just barely good enough" to stay agile.
- Implementation – Translates models into source code.
- Testing – Follows an XP-style approach with test design and execution to detect errors and ensure requirement fulfillment.
- Deployment – Delivers software increments and gathers user feedback.
- Configuration & Project Management – Manages changes, risks, and work products, tracking progress and coordinating team activities.
- Environment Management – Maintains standards, tools, and support infrastructure for the development process.

6.TOOL SET FOR THE AGILE PROCESS:

Alistair Cockburn suggests that agile teams benefit from tools that facilitate rapid understanding. These tools can be social, technological, or physical and support team collaboration, stakeholder communication, and management. Agile tools are not just software; they include any mechanism that enhances collaboration, efficiency, and adaptability within agile teams.

Types of Agile Tools:

- Hiring Tools – Methods like pair programming trials with prospective team members to assess fit.
- Collaboration & Communication Tools – Often low-tech, including whiteboards, index cards, sticky notes, and physical proximity for effective teamwork.
- Information Radiators – Visual displays (e.g., flat panels, dashboards) that show project status transparently.
- Project Management Tools – Shift focus from Gantt charts to earned value charts, test coverage graphs, and real-time tracking.
- Process & Culture Tools – Optimize the team's work environment, enhance social interaction (e.g., co-located teams), and improve efficiency through electronic whiteboards, pair programming, and time-boxing.