UNIT-III

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.

Discuss about the software design process with approaches?

The design process essentially transforms the SRS document into a design document.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 it is still evolving.

Function-oriented

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 subfunctions:

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

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. The system state is centralised and shared among different functions.

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

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

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

ADT is an important concept that forms an important pillar of objectorientation. 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.

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. what is the advantage of developing an application using ADTs?

Let us examine the three main advantages of using ADTs in programs:

1. 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.
2. An ADT-based design displays high cohesion and low coupling. Therefore, object-oriented designs are highly modular.
3. Since the principle of abstraction is used, it makes the design solution easily understandable and helps to manage complexity.

How to characterize the good software design?

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 optimisation 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

These two principles are exploited by design methodologies to make a design modular and layered. 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. A design solution should be modular and layered to be

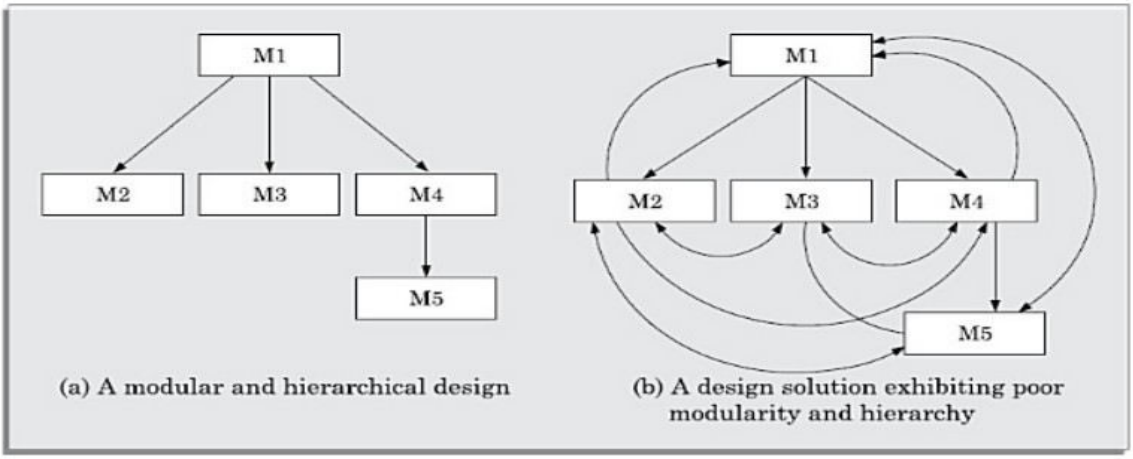


Figure 5.2: Two design solutions to the same problem.

understandable.

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.

For example, consider two alternate design solutions to a problem that are represented in Figure 5.2, 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 5.2(a) would be easier to understand since the interactions among the different modules is low.

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, since a module at a lower layer is unaware of (abouthow 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

Explain about the cohesion method?

Cohesion is a measure of the functional strength of a module,If the functions of the module do

then the module has very poor cohesion.

Functional independence

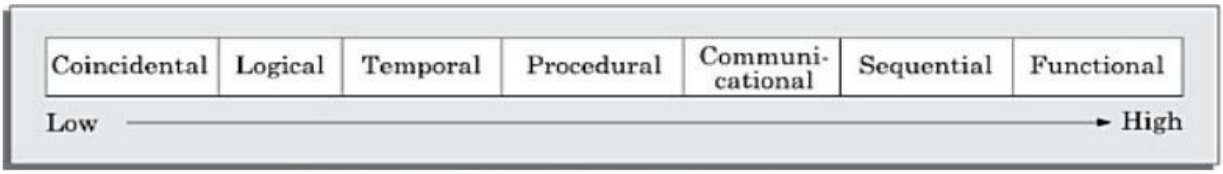


Figure 5.3: Classification of cohesion.

By the term functional independence, we mean that a module performs a single task and needs very little interaction with other modules. A module that is highly cohesive and also has low coupling with other modules is said to be functionally independent of the 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.

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.

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.

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 classes of cohesion are elaborated below.

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 at all. In this case, we can say that the module contains a random collection of functions.

An example of a module with coincidental cohesionhas been shown in Figure 5.4(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

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

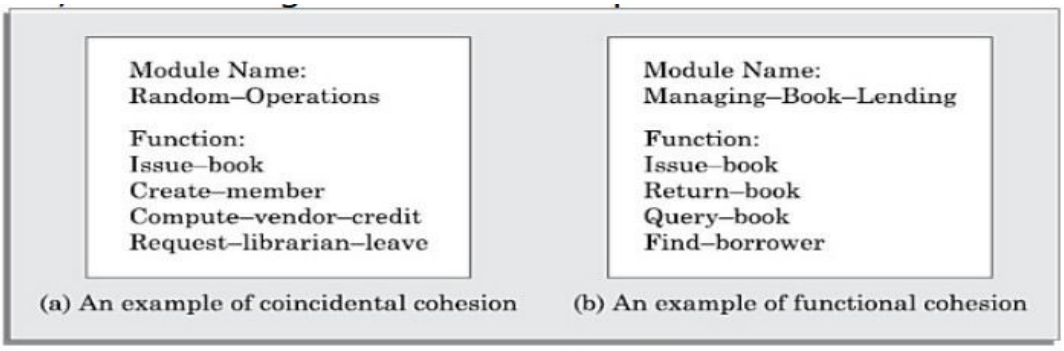


Figure 5.4: Examples of cohesion.

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 initialisation of memory and devices, loading the operating system, etc

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(), printbill(), place-order-on-vendor(), update-inventory(), and logout() all do different thing and operate on different data.

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(), placeorder-on-vendor() are placed in a single module

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.

Briefly write about the coupling method?

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. The interface complexity is determined based on the number of parameters and the complexity of the parameters that are interchanged while

These different types of coupling,

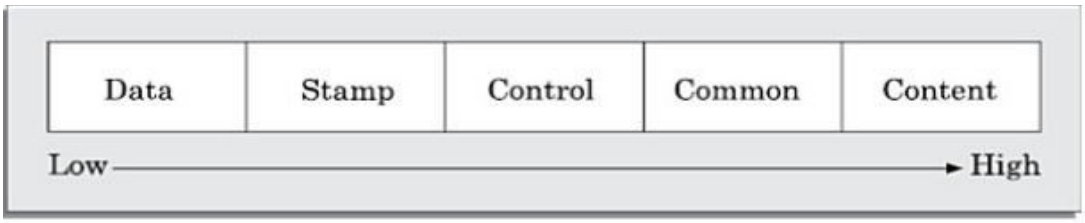


Figure 5.5: Classification of coupling.

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.

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)

AGILE DEVELOPMENT

WHAT IS AGILITY?

Agile is a software development methodology to build software incrementally using short iterations of 1 to 4 weeks so that the development process is aligned with the changing business needs.

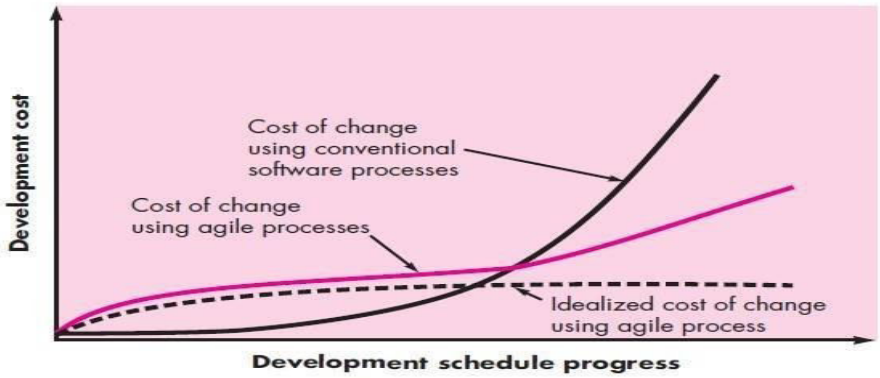
An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.

AGILITY AND THE COST OF CHANGE

An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment.

Agility argue that a well-designed agile process "flattens" the cost of change curve

software project without dramatic cost and time impact. When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming, the cost of making a change is attenuated. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence to suggest that a significant reduction in the cost of change can be achieved.



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 in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. 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 as predictable

Agility Principles

Agility principles for those who want to achieveagility:

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. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers 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. The sponsors, developers, andusers should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances 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, then tunes and adjusts its behavior accordingly.

Human Factors

Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that *the process molds to the needs of the people and team*

- **Competence.** In an agile development context, "competence" encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.
- **Common focus.** Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.

- **Collaboration.** Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.
- **Decision-making ability.** Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.

Fuzzy problem-solving ability. Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted bychange.

- **Mutual trust and respect.** The agile team must become what DeMarco and Lister call a "jelled" team. A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of the parts."

Self-organization. In the context of agile development, self-organization implies **three** things: (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. Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale.

EXTREME PROGRAMMING (XP)

Extreme Programming **(XP),** the most widely used approach to agile software development, emphasizes business results first and takes an incremental, get-something-started approach to building the product, using continual testing and revision.

XP Values

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**.

In order to achieve effective ***communication*** between software engineers and other stakeholders, XP emphasizes close, yet informal collaboration between customers and developers, the establishment of effective metaphors3 for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.

To achieve ***simplicity,*** XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code). If the design must be improved, it can be *refactored* at a later time.

Feedback is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy the software provides the agile team with feedback. XP makes use of the ***unit test*** as its primary testing tactic. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality.

Beck argues that strict adherence to certain XP practices demands ***courage.*** A better word might be ***discipline****.* An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.

By following each of these values, the agile team inculcates ***respect*** among it members, between other stakeholders and team members, and indirectly, for the software

respect for the XP process.

The XP Process

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Following figure illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity.

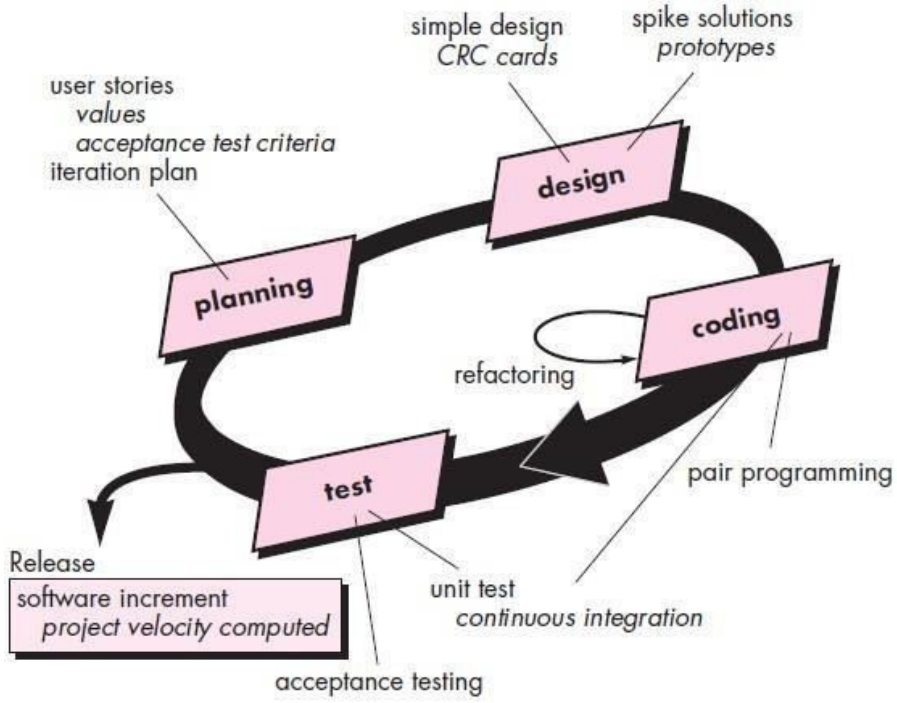


Fig : The Extreme Programming process

Key XP activities are

- **Planning.** The planning activity (also called *the planning game*) begins with *listening*—a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality.
- **Design.** XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality 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***, the design prototype is implemented and evaluated. XP encourages ***refactoring***—a construction technique that is also a method for design optimization.

  Fowler describes r**efactoring** in the following manner: Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure. It is a disciplined way to clean up code [that minimizes the chances of introducing bugs].

  - **Coding.** After stories are developed and preliminary design work is done, the team does *not* move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers

people work together at one computer workstation to create code for a story. This provides a mechanism for real time problem solving (two heads are often better than one) and real-time quality assurance.

- **Testing.** The creation of unit tests before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated. This encourages a regression testing strategy whenever code is modified. 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. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells states: "Fixing small problems every few hours takes less time than fixing huge problems just before the deadline."

XP ***acceptance tests***, also called ***customer tests***, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. 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) in the following manner: "IXP is an organic evolution of XP. It is imbued with XP's minimalist, customer-centric, test-driven spirit. IXP differs most from the original XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices." IXP incorporates six new practices that are designed to help ensure that an XP project works successfully for significant projects within a large organization.

- **Readiness assessment.** Prior to the initiation of an IXP project, the organization should conduct a *readiness assessment.* The assessment ascertains whether (1) an appropriate development environment exists to support IXP, (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, and (5) the broader project community will be populated appropriately.
- **Project community.** Classic XP suggests that the right people be used to populate the agile team to ensure success. The implication is that people on the team must be well- trained, adaptable and skilled, and have the proper temperament to contribute to a self- organizing team. When XP is to be applied for a significant project in a large organization, the concept of the "team" should morph into that of a *community.* A community may have a technologist and customers who are central to the success of a project as well as many other stakeholders (e.g., legal staff, quality auditors, manufacturing or sales types) who "are often at the periphery of an IXP project yet they may play important roles on the project". In IXP, the community members and their roles should be explicitly defined and mechanisms for communication and coordination between community members should be established.
- **Project chartering.** The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the organization. Chartering also examines the context of the project to determine how it complements, extends, or replaces existing systems or processes.
- **Test-driven management.** An IXP project requires measurable criteria for assessing the state of the project and the progress that has been made to date. Test-driven management establishes a series of measurable "destinations" and then defines mechanisms for determining whether or not these destinations have been reached.

increment is delivered. Called a *retrospective,* the review examines "issues, events, and lessons-learned" across a software increment and/or the entire software release. The intent is to improve the IXP process.

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

OTHER AGILE PROCESS MODELS

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 Highsmith as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization.

High smith argues that an agile, adaptive development approach based on collaboration is "as much a source of *order* in our complex interactions as discipline and engineering." He defines an ASD "life cycle" that incorporates three phases, speculation, collaboration, and learning.

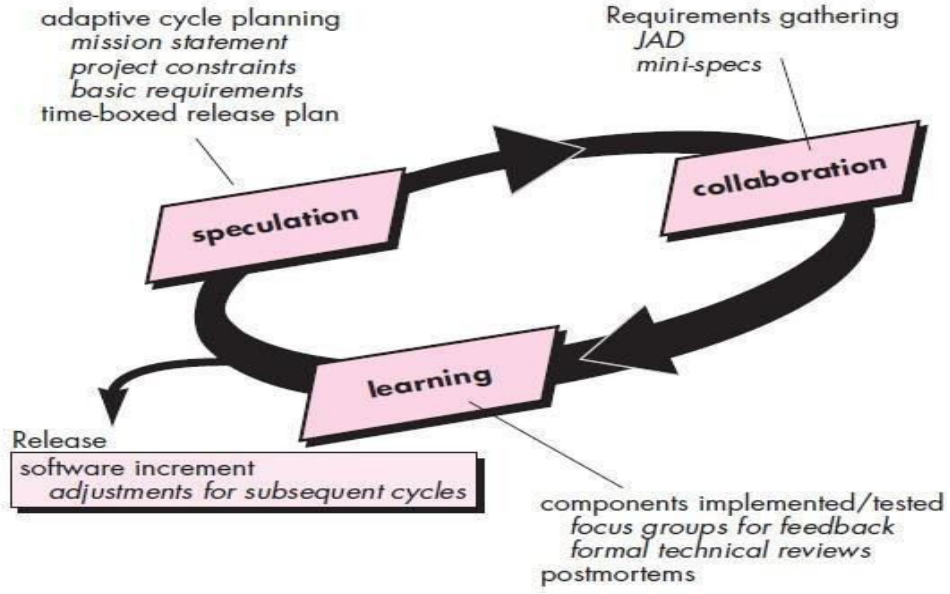


Fig : Adaptive software development

During ***speculation***, the project is initiated and ***adaptive cycle planning*** is

mission 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*** in a way that multiplies their talent and creative output beyond their absolute numbers. This approach is a recurring theme in all agile methods. But collaboration is not easy. It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking. It is, above all, a matter of trust. People working together must trust one another to (1) criticize without animosity, (2) assist without resentment, (3) work as hard as or harder than they do, (4) have the skill set to contribute to the work at hand, and (5) communicate problems or concerns in a way that leads to effective action.

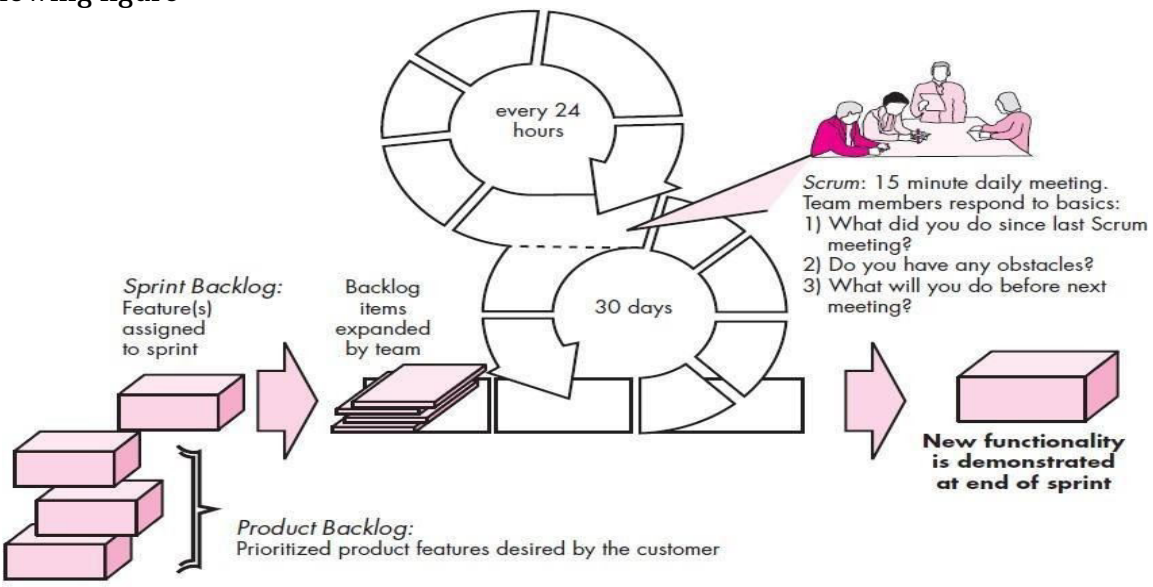
As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on **"learning"** as much as it is on progress toward a completed cycle.

ASD teams learn in **three** ways: **focus groups, technical reviews , and project postmortems**. ASD's overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning yield software project teams that have a much higher likelihood of success.

Scrum

Scrum is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks occur within a process pattern called a ***sprint.*** The work conducted within a sprint is adapted to the problem at hand and is defined and often modified

in real time by the Scrum team. The overall flow of the Scrum process is illustrated in following figure



Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns 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. The product manager assesses the backlog and updates priorities as required.

the backlog that must be fit into a predefined time-box (typically 30 days). Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.

- ***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. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to "**knowledgesocialization**"

- ***Demos***—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

Dynamic Systems Development Method (DSDM)

The *Dynamic Systems Development Method* (DSDM) is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment" The DSDM philosophy is borrowed from a modified version of the **Pareto principle—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 *DSDM life cycle* that defines **three** different iterative cycles, preceded by **two** additionallife cycle activities:

- ***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; also, defines the basic application architecture and identifies the maintainability requirements for the application.
- ***Functional model iteration***—produces a set of incremental prototypes that demonstrate functionality for the customer.
- ***Design and build iteration***—revisits prototypes built during ***functional model iteration*** to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, *functional model iteration* and *design and build iteration* occur concurrently.
- ***Implementation***—places the latest software increment 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. In either case, DSDM development work continues by returning to the functional model iteration activity.

Crystal

Alistair Cockburn and Jim Highsmith created the *Crystal family of agile methods* in

maneuverability" during what 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"

The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects. The intent is to allow agile teams to select the member of the crystal family that is most appropriate for their project and environment.

Feature Driven Development (FDD)

Feature Driven Development (FDD) was originally conceived by Peter Coad and his colleagues as a practical process model for object-oriented software engineering. Stephen Palmer and John Felsing have extended and improvedCoad's work, describing an adaptive, agile process that can be applied to moderately sized and larger softwareprojects.

Like other agile approaches, FDD adopts a philosophy that (1) emphasizes collaboration among people on an FDD team; (2) manages problem and project complexity using feature- based decomposition followed by the integration of software increments, and (3) communication of technical detail using verbal, graphical, and text-based means.

FDD emphasizes software quality assurance activities by encouraging an incremental development strategy, the use of design and code inspections, the application of software quality assurance audits, the collection of metrics, and the use of patterns (for analysis, design, and construction).

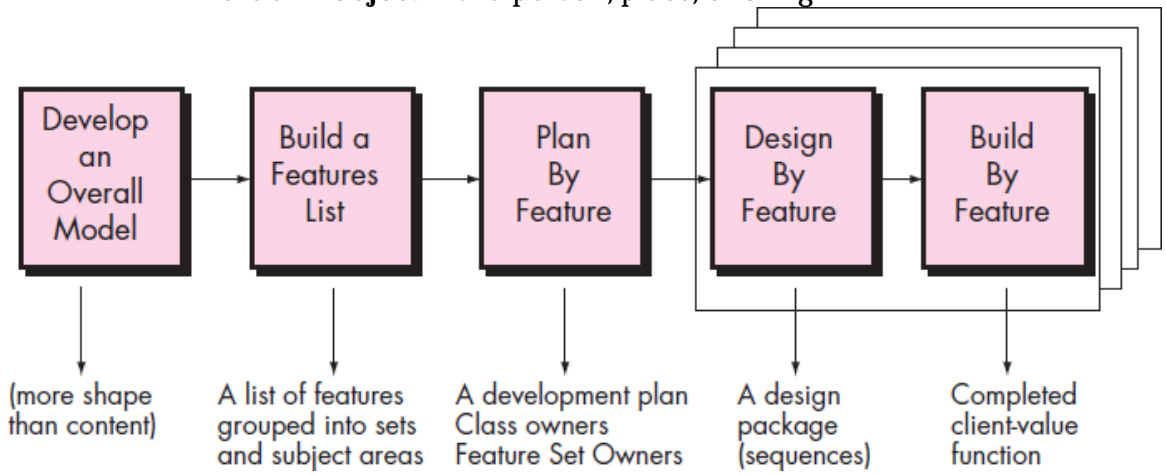
In the context of FDD, a *feature* "is a client-valued function that can be implemented in two weeks or less" The emphasis on the definition of features provides the followingbenefits:

- Because features are small blocks of deliverable functionality, users can describe them more easily; understand how they relate to one another more readily; and better review them for ambiguity, error, or omissions.
- Features can be organized into a hierarchical business-related grouping.
- 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 to inspect effectively.
- Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

Coad and his colleagues suggest the following template for defining a feature:

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

where an **<object>** is "a person, place, or thing



FDD provides greater emphasis on project management guidelines and techniques than many other agile methods. 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. The lean principles that inspire the LSD process can be summarized as ***eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people,*** **and** ***optimize the whole***. Each of these principles can be adapted to the software process.

Agile Modeling (AM)

Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. Agile models are more effective than traditional models because they are just barely good, they don't have to be perfect.

Agile modeling adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refactor. The team must also have the humility to recognize that technologists do not have all the answers and that business experts and other stakeholders should be respected and embraced.

Agile Modeling suggests a wide array of "core" and "supplementary" modeling principles, those that make AM unique are :

- **Model with a purpose.** A developer who uses AM should have a specific goal in mind before creating the model. Once the goal for the model is identified, the type of notation to be used and level of detail required will be more obvious.
- **Use multiple models.** There are many different models and notations that can be used to describe software. Only a small subset is essential for most projects. AM suggests that to provide needed insight, 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 and jettison the rest. Every work product that is kept must be maintained as changes occur. This represents work that slows the team down. Ambler notes that "Every time you decide to keep a model you trade-off agility for the convenience of having that information available to your team in an abstract manner
- **Content is more important than representation.** Modeling should impart information to its intended audience. A syntactically perfect model that imparts little useful content is not as valuable as a model with flawed notation that nevertheless provides valuable content for its 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. By adopting the classic UP phased activities

enables a team to visualize the overall process flow for a software project. However, within each of the activities, the team iterates to achieve agility and to deliver meaningful software increments to end users as rapidly as possible. 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.*** Like the generic process activity deployment in this context focuses on the delivery of a software increment and the acquisition of feedback from end users.
- ***Configuration and project management.*** In the context of AUP, 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.

A TOOL SET FOR THE AGILE PROCESS

Some proponents of the agile philosophy argue that automated software tools (e.g., design tools) should be viewed as a minor supplement to the team's activities, and not at all pivotal to the success of the team. However, Alistair Cockburn [Coc04] suggests that tools can have a benefit and that "agile teams stress using tools that permit the rapid flow of understanding. Some of those tools are social, starting even at the hiring stage. Some tools are technological, helping distributed teams simulate being physically present. Many tools are physical, allowing people to manipulate them in workshops."

Because acquiring the right people (hiring), team collaboration, stakeholder communication, and indirect management are key elements in virtually all agile process models, Cockburn argues that "tools" that address these issues are critical success factors for agility. For example, a hiring "tool" might be the requirement to have a prospective team member spend a few hours pair programming with an existing member of the team. The "fit" can be assessed immediately.

Collaborative and communication "tools" are generally low tech and incorporate any mechanism ("physical proximity, whiteboards, poster sheets, index cards, and sticky notes" [Coc04]) that provides information and coordination among agile developers.

Active communication is achieved via the team dynamics (e.g., pair programming), while passive communication is achieved by "information radiators" (e.g., a flat panel display that presents the overall status of different components of an increment). Project management tools deemphasize the Gantt chart and replace it with earned value charts or "graphs of tests created versus passed . . . other agile tools are used to optimize the environment in which the agile team works (e.g., more efficient meeting areas), improve the team culture by nurturing social interactions (e.g., collocated teams), physical devices (e.g., electronic whiteboards), and process enhancement (e.g., pair programming or time-boxing)" [Coc04].

Function-Oriented Software Design: Overview of SA/SD Methodology, Structured Analysis, Structured Design, Detailed Design, Design Review.

Introduction to Function-Oriented Software Design:

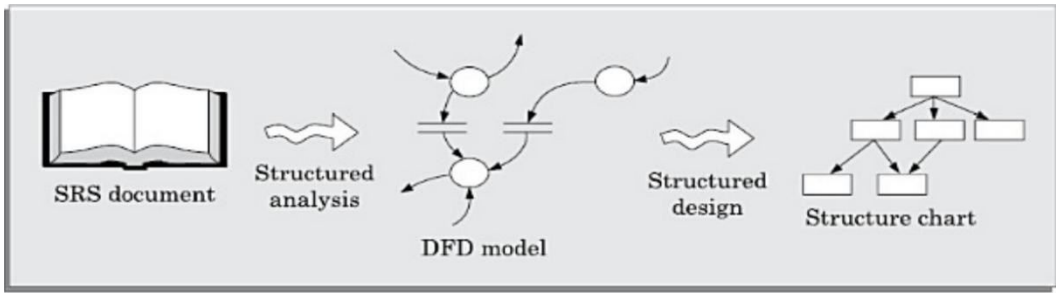
- Function-oriented design techniques were proposed nearly four decades ago.
- still very popular and are currently being used in many software development organizations. These techniques, to start with, view a system as a black-box that provides a set of services to the users of the software.
- These 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 i s 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 b e 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

to a module structure. This module structure is also called the high level design of 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:

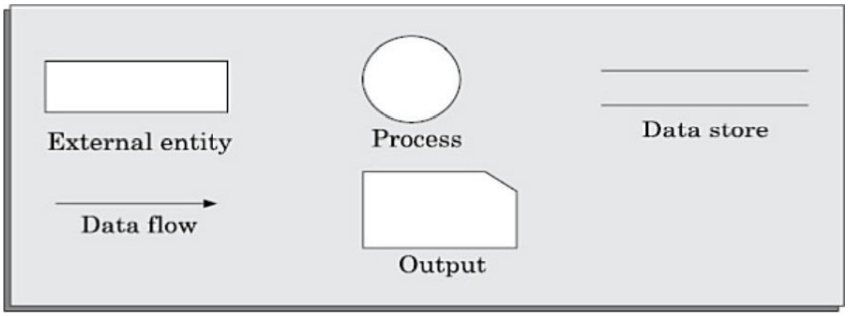
- Top-down decomposition approach.
- Application of divide and conquer principle.
- Through this each high level function is independently decomposed into detailed functions. Graphical representation of the analysis results using data flow diagrams (DFDs).

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 subfunctions performed by the functions using a hierarchy of diagrams.

Primitive symbols used for constructing DFDs

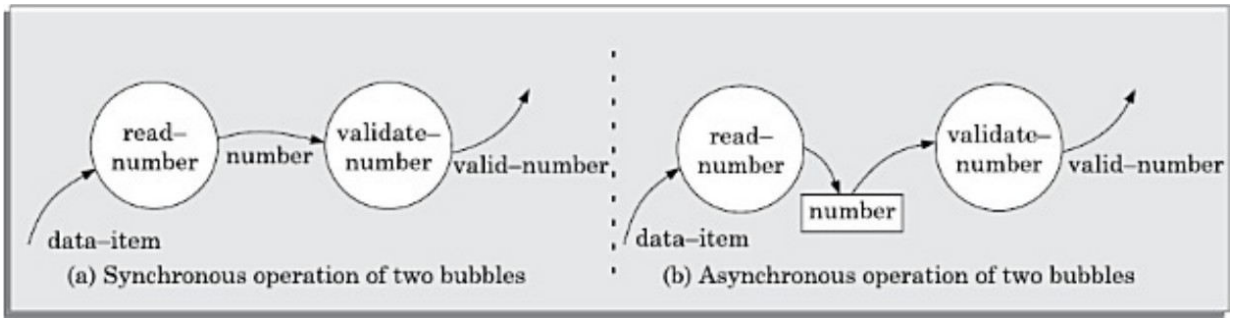
There are essentially five different types of symbols used for constructing DFDs.



- **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 i s 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 are 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:

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.

Self Study:

- ☐ ***Data Definition***

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 subprocesses and the data flow among these subprocesses are 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 subfunctions at the successive levels of the DFD model. Decomposition of a bubble is also known as factoring o r 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.

Self Study: Examples of DFD Models ☐ **RMS Calculator**

☐ ***Trading House Automation System***

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 i s inherent to a flow chart is suppressed in a 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.
- In the third step of transform analysis, the structure chart is refined by adding sub functions required by each of the high-level functional components.

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:

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.

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.

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

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-

- **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

- 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, They have become affordable.

Types of User Interfaces:

- Broadly speaking, 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.

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 recognised 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.

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

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.

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