SOFTWARE ENGINEERING (CSE3005)

Prof. Anand Motwani
Faculty, SCSE
VIT Bhopal University

UNIT – III
(SOFTWARE DESIGN CONCEPTS)

- The Design Process 2
  - Design Concepts
  - Design Model
- Software Architecture 2
  - Architectural Styles
  - Alternatives
- Mapping To DFD 1
- Component Based Development and Design 1
- User Interface Design 1
  - Interface Analysis,
  - Interface Design.

SOFTWARE DESIGN CONCEPTS

- The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture.
- Software design involves identifying and describing the fundamental software system abstractions and their relationships.

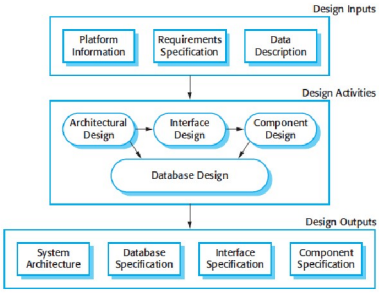
- A software design is a
  - description of the structure of the software to be implemented,
  - the data models and structures used by the system,
  - the interfaces between system components and, sometimes, the algorithms used.
  - **Note:**
  - Designers do not arrive at a finished design immediately but develop the design iteratively. They add formality and detail as they develop their design with constant backtracking to correct earlier designs.

GOAL OF DESIGN PROCESS

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

GENERAL MODEL OF DESIGN PROCESS

- Figure is an abstract model of this process showing the inputs to the design process, process activities, and the documents produced as outputs from this process.



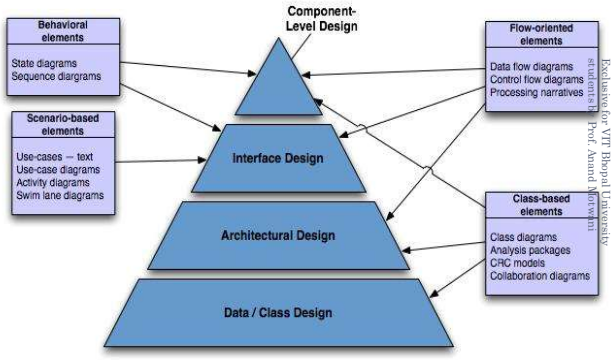
FOUR ACTIVITIES THAT MAY BE PART OF THE DESIGN PROCESS FOR INFORMATION SYSTEMS:

Design Model consists of 4 designs:

- 1. *Architectural design, where you identify the overall structure of the system, the* principal components (sometimes called sub-systems or modules), their relationships, and how they are distributed.
- 2. *Interface design, where you define the interfaces between system components.* This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.

- 3. *Component design, where you take each system component and design how it will* operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model. The design model may be used to automatically generate an implementation.
- 4. *Database design, where you design the system data structures and how these are* to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.

Translating Analysis → Design



Design Concepts

Design concepts

Design concepts provide the necessary framework for "to get the thing on right way".

- Abstraction
- Refinement
- Modularity
- Architecture
- Control Hierarchy
- Structural Partitioning
- Data Structure
- Software Procedure
- Information Hiding

Abstraction

- At the highest level of abstraction – a solution is stated in broad terms
- At lower level of abstraction – a more detailed description of the solution is provided.
- Two types of abstraction:
- *Procedural abstraction:* Sequence of instructions that have a specific and limited function.

Ex. Open a door

open implies long sequence of activities (e.g. walk to the door, grasp knob, turn knob and pull the door, etc).

- *Data abstraction*: collection of data that describes a data object.

Ex. Open a door. – **door** is data object.

- Data abstraction for ***door*** would encompass a set of attributes that describe the door. (E.g. door type, swing direction, opening mechanism, etc.)

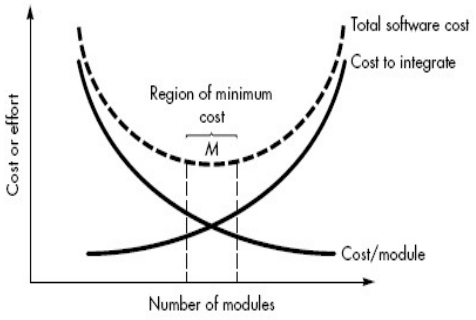
Refinement

- Refinement is actually a process of *elaboration.*
- begin with a statement of function (or description of information) that is defined at a high level of abstraction.
- That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information.
- Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.
- Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details.
- Refinement helps the designer to expose low-level details as design progresses.

Modularity

- Architecture and design pattern embody modularity.
- Software is divided into separately named and addressable components, sometimes called modules, which are integrated to satisfy problem requirement.
- modularity is the single attribute of software that allows a program to be intellectually manageable
- It leads to a "divide and conquer" strategy. – it is easier to solve a complex problem when you break into a manageable pieces.
- Refer fig. that state that effort (cost) to develop an individual software module does decrease if total number of modules increase.
- However as the no. of modules grows, the effort (cost) associated with integrating the modules also grows.

Modularity and Software Cost



- Undermodularity and overmodularity should be avoided. But how do we know the vicinity of M?
- We modularize a design so that development can be more easily planned.
- Software increments can be defined and delivered.
- Changes can be more easily accommodated.
- Testing and debugging can be conducted more efficiently and long-term maintained can be conducted without serious side effects.

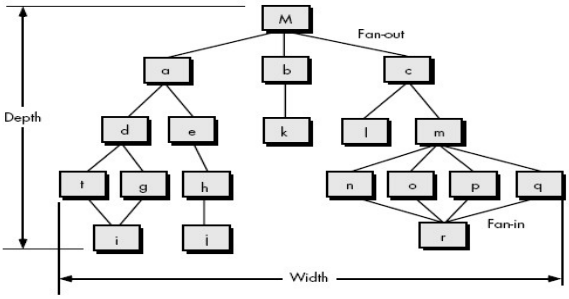
ARCHITECTURE

- Software architecture suggest " the overall structure of the software and the ways in which that structure provides conceptual integrity for a system.
- No. of different models can use to represent architecture.
  - Structural Model- represent architecture as an organized collection of components
  - Framework model – Increase level of design abstraction by identifying repeatable architectural design framework.
  - Dynamic model – address behavior of the program architecture and indicating how system or structure configuration may change as a function.
  - Process Model – focus on design of the business or technical process that the system must accommodate.
  - Functional models – used to represent the functional hierarchy of a system.

CONTROL HIERARCHY

- *Control hierarchy,* represents the organization of program components (modules) and implies a hierarchy of control.
- Different notations are used to represent control hierarchy for those architectural styles that are amenable to this representation.
- The most common is the treelike diagram that represents hierarchical control for call and return architectures.

CONTROL HIERARCHY

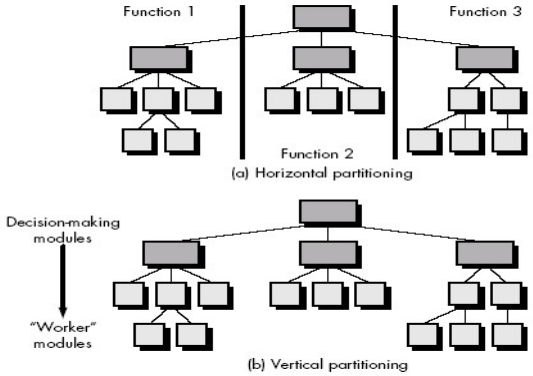


- In fig. *depth* and *width* provide an indication of the number of levels of control and overall *span of control*.
- *Fan-out* is a measure of the number of modules that are directly controlled by another module. *Fan-in* indicates how many modules directly control a given module.
- A module that controls another module is said to be *superordinate* to it, and conversely, a module controlled by another is said to be *subordinate* to the controller.

Ex. Module *M* is superordinate to modules *a, b,* and *c.*

Module *h* is subordinate to module *e*

STRUCTURAL PARTITIONING



STRUCTURE PARTITIONING

- Two types of Structure partitioning:
  - Horizontal Partitioning
  - Vertical Partitioning
- ***Horizontal partitioning*** defines separate branches of the modular hierarchy for each major program function.
- *Control modules*, represented in a darker shade are used to coordinate communication & execution between its functions.
- *Horizontal partitioning* defines three partitions—input, data transformation (often called *processing*) and output.
- Benefits of Horizontal partitioning:
  - software that is easier to test
  - software that is easier to maintain
  - propagation of fewer side effects
  - software that is easier to extend

- On the negative side (Drawback), horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow.
- ***Vertical partitioning***, often called *factoring*, suggests that control (decision making) and work should be distributed top-down in the program structure.
- Top-level modules should perform control functions and do little actual processing work.
- Modules that reside low in the structure should be the workers, performing all input, computation, and output tasks.

- A change in a control module (high in the structure) will have a higher probability of propagating side effects to modules that are subordinate to it.
- A change to a worker module, given its low level in the structure, is less likely to cause the propagation of side effects.
- For this reason vertically partitioned structures are less likely to be susceptible to side effects when changes are made and will therefore be more maintainable.

Data Structure

- *Data structure* is a representation of the logical relationship among individual elements of data
- A *scalar item* is the simplest of all data structures. It represents a single element of information that may be addressed by an identifier.
- When scalar items are organized as a list or contiguous group, a *sequential vector* is formed.
- When the sequential vector is extended to two, three, and ultimately, an arbitrary number of dimensions, an *n-dimensional space* is created. Most common n-dimensional space is the two-dimensional matrix
- A *linked list* is a data structure that organizes contiguous scalar items, vectors, or spaces in a manner (called *nodes*) that enables them to be processed as a list.
- A *hierarchical data structure* is implemented using multilinked lists that contain scalar items, vectors, and possibly, *n*-dimensional spaces.

Software Procedure

- Software procedure focuses on the processing details of each module individually.
- Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization and structure.

Information Hiding

- The principle of *information hiding* suggests that modules be "characterized by design decisions that (each) hides from all others modules."
- In other words, modules should be specified and designed so that information (algorithm and data) contained within a module is inaccessible to other modules that have no need for such information.
- The intent of information hiding is to hide the details of data structure and procedural processing behind a module interface.
- It gives benefits when modifications are required during testing and maintenance because data and procedure are hiding from other parts of software, unintentional errors introduced during modification are less.

EFFECTIVE MODULAR DESIGN

- Effective modular design consist of three things:
  - Functional Independence
  - Cohesion
  - Coupling

FUNCTIONAL INDEPENDENCE

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- In other words - each module addresses a specific sub-function of requirements and has a simple interface when viewed from other parts of the program structure.
- Independence is important –
  - Easier to develop
  - Easier to Test and maintain
  - Error propagation is reduced
  - Reusable module.

FUNCTIONAL INDEPENDENCE

- To summarize, functional independence is a key to good design, and design is the key to software quality.
- To measure independence, have two qualitative criteria: cohesion and coupling
- *Cohesion* is a measure of the relative functional strength of a module.
- *Coupling* is a measure of the relative interdependence among modules.

COHESION

- Cohesion is a natural extension of the information hiding concept
- A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program
- Simply state, a cohesive module should (ideally) do just one thing.
- We always strive for high cohesion, although the mid-range of the spectrum is often acceptable.
- Low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion.
- So. designer should avoid low levels of cohesion when modules are designed.

COHESION

- When processing elements of a module are related and must be executed in a specific order, *procedural cohesion* exists.
- When all processing elements concentrate on one area of a data structure, *communicational cohesion* is present.
- High cohesion is characterized by a module that performs one distinct procedural task.

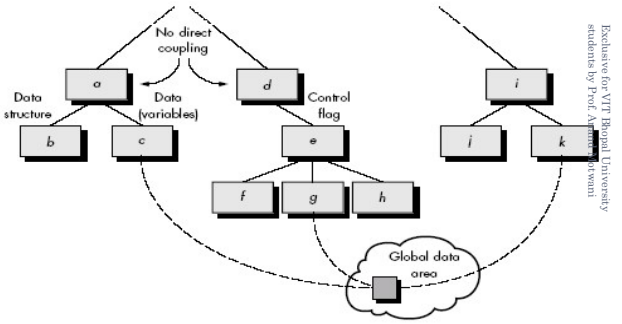
TYPES OF COHESION

- A module that performs tasks that are related logically is *logically cohesive*.
- When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits *temporal cohesion*.
- At the low-end of the spectrum, a module that performs a set of tasks that relate to each other loosely, called *coincidentally cohesive*.

COUPLING

- Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface
- In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect" caused when errors occur at one location and propagate through a system.
- It occur because of design decisions made when structure was developed.

COUPLING



COUPLING

- Coupling is characterized by passage of control between modules.
- "Control flag" (a variable that controls decisions in a subordinate or superordinate module) is passed between modules d and e (called control coupling).
- Relatively high levels of coupling occur when modules are communicate with external to software.
- External coupling is essential, but should be limited to a small number of modules with a structure.

- High coupling also occurs when a number of modules reference a global data area.
- Common coupling, no. of modules access a data item in a global data area
- So it does not mean "use of global data is bad". It does mean that a software designer must be take care of this thing.

EVOLUTION OF SOFTWARE DESIGN

- Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a top-down manner.
- Later work proposed methods for the translation of data flow or data structure into a design definition.
- Today, the emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures.

CHARACTERISTICS ARE COMMON TO ALL DESIGN METHODS

- A mechanism for the translation of analysis model into a design representation,
- A notation for representing functional components and their interfaces.
- Heuristics for refinement and partitioning
- Guidelines for quality assessment.

Design Quality Attributes

- Acronym FURPS –
  - Functionality
  - Usability
  - Reliability
  - Performance
  - Supportability

- Functionality – is assessed by evaluating the feature set and capabilities of the program.
  - Functions that are delivered and security of the overall system.
- Usability – assessed by considering human factors, consistency & documentation.
- Reliability – evaluated by
  - measuring the frequency and severity of failure.
  - Accuracy of output results.
  - Ability to recover from failure and predictability of the program.
- Performance - measured by processing speed, response time, resource consumption, efficiency.
- Supportability – combines the ability to extend the program (extensibility), adaptability and serviceability.

Software Architecture

"The software architecture of a program or computing system is:

- the structure or structures of the system, which comprise the software components,
- the externally visible properties of those components,
- and the relationships among them."

Software Architecture

- It is not operational software but it is representation that enables software engineer to
  - Analyze the effectiveness of design in meeting its stated requirement.
  - consider architectural alternatives at a stage when making design changes is still relatively easy,
  - reduce the risks associated with the construction of the software.

Importance of Software Architecture

- Architecture Representations is an enabler for communication between all parties (stakeholders) interested in the development
- It highlights early design decisions that will have a profound impact on all S. Engg. work that follows.
- It "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together"

Architectural Style

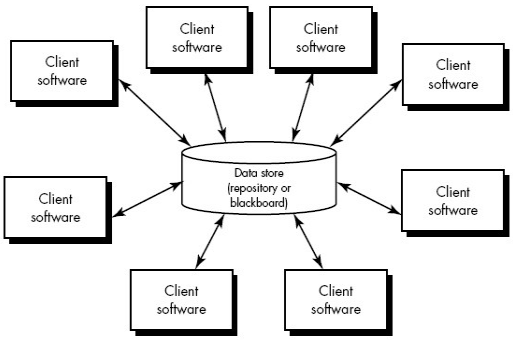
- Style describes a system category that encompasses

1. A set of *components* (e.g., a database, computational modules) that perform a function required by a system;
2. a set of *connectors* that enable "communication, co-ordinations and cooperation" among components;
3. *constraints* that define how components can be integrated to form the system
4. *semantic models* that enable a designer to understand the overall properties of a system

It can be represent by

- Data-centered architecture
- Data flow architecture
- Call and return architecture
- Object oriented architecture
- Layered architecture.

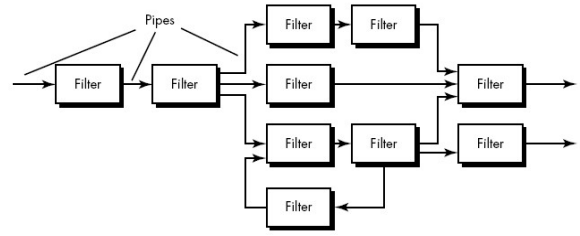
Data-centered architecture



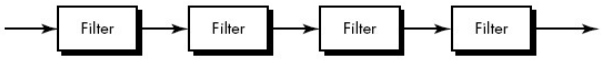
Data-centered architecture

- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.
- Client software accesses a central repository which is in passive state (in some cases).
- client software accesses the data independent of any changes to the data or the actions of other client software.
- So, in this case transform the repository into a "Blackboard".
- A blackboard sends notification to subscribers when data of interest changes, and is thus active.
- Data-centered architectures promote *integrability*.
- Existing components can be changed and new client components can be added to the architecture without concern about other clients.
- Data can be passed among clients using the blackboard mechanism. So Client components independently execute processes

DATA FLOW ARCHITECTURE



Pipes and filters

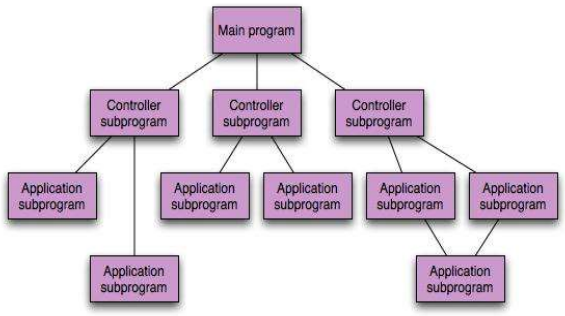


Batch Sequential

DATA FLOW ARCHITECTURE

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- A *pipe and filter pattern (Fig .)* has a set of components, called *filters,* connected by pipes that transmit data from one component to the next.
- Each filter works independently (i.e. upstream, downstream) and is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- the filter does not require knowledge of the working of its neighboring filters.
- If the data flow degenerates into a single line of transforms, it is termed *batch sequential.*

CALL AND RETURN ARCHITECTURE



CALL AND RETURN ARCHITECTURE

- Architecture style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale.
- Two sub-styles exist within this category:

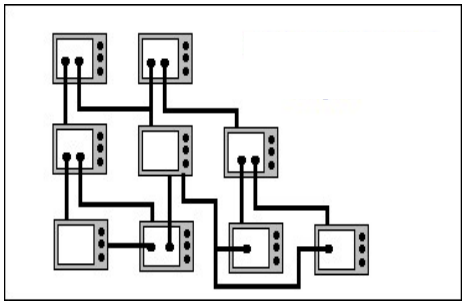
1. *Main/sub program architecture:*

- Program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components.

2. *Remote procedure Call architecture:*

- The components of a main program/subprogram architecture are distributed across multiple computers on a network

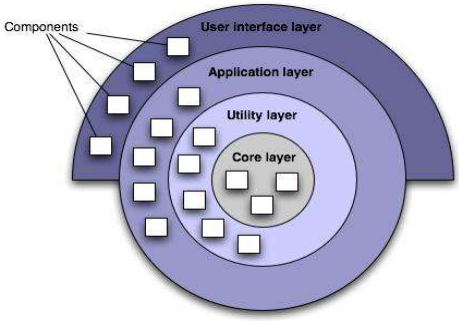
OBJECT-ORIENTED ARCHITECTURE



OBJECT-ORIENTED ARCHITECTURE

- The object-oriented paradigm, like the abstract data type paradigm from which it evolved, emphasizes the bundling of data and methods to manipulate and access that data (Public Interface).
- Components of a system summarize data and the operations that must be applied to manipulate the data.
- Communication and coordination between components is accomplished via message passing.

LAYERED ARCHITECTURE

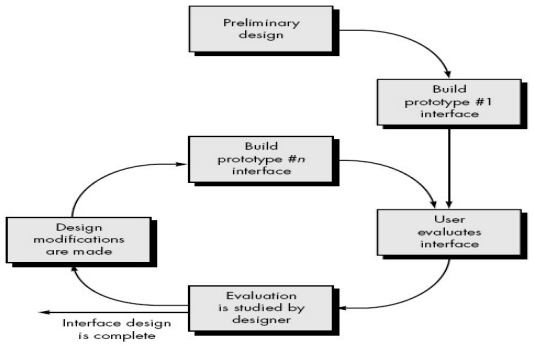


- A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components examine user interface operations.
- At the inner layer, components examine operating system interfacing.
- Intermediate layers provide utility services and application software functions.

User Interface Design

- User interface design creates an effective communication medium between a human and a computer.
- Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

Design Evaluation



user interface evaluation cycle

- After the design model has been completed, a first-level prototype is created.
- The prototype is evaluated by the user, who provides the designer with direct comments about the efficiency of the interface.
- In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), the designer may extract information from these data.
- Design modifications are made based on user input and the next level prototype is created.
- The evaluation cycle continues until no further modifications to the interface design are necessary.
- The prototyping approach is effective, but is it possible to evaluate the quality of a user interface before a prototype is built?
- If potential problems uncovered and corrected early, the number of loops through the evaluation cycle will be reduced and development time will shorten.

- Evaluation criteria can be applied during early design reviews:

1. The length and complexity of the written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
2. The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.
3. The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.
4. Interface style, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

- Once the first prototype is built, the designer can collect a variety of qualitative and quantitative data that will assist in evaluating the interface.
- To collect qualitative data, questionnaires can be distributed to users of the prototype.
- Questions can be all
  - simple yes/no response,
  - numeric response,
  - scaled (subjective) response,
  - percentage (subjective) response.
  - Likert scale (e.g. strongly disagree, somewhat agree).
  - Open-minded.

EXAMPLE

1. Were the icons self-explanatory? If not, which icons were unclear?
2. Were the actions easy to remember and to invoke?
3. How many different actions did you use?
4. How easy was it to learn basic system operations (scale 1 to 5)?
5. Compared to other interfaces you've used, how would this rate—top 1%, top 10%, top 25%, top 50%, bottom 50%?

QUIZ

1. What is outside in design?

- a) Interaction design works mainly from the computational goals mandated in requirements specifications , to the realization as interactions between cooperating program units
- b) Many program interactions do not give rise to interesting or complicated program component interactions
- c) All of the mentioned
- d) None of the mentioned

2. What is controller ?

- a) It is a program component that makes decisions and directs other components
- b) It is a way that decision making is distributed among program components
- c) All of the mentioned
- d) None of the mentioned

3. Match the following

GROUP I	GROUP II
(P) Service oriented computing	(1) Interoperability
(Q) Heterogeneous communicating systems	(2) BPMN
(R) Information representation	(3) Publish-find-bind
(S) Process description	(4) XML

- **(A)** P-1, Q-2, R-3, S-4
  (B) P-3, Q-4, R-2, S-1
  (C) P-3, Q-1, R-4, S-2
  (D) P-4, Q-3, R-2, S-1

Mapping To DFD

- Already discussed and performed practical in class.
- For sample example refer the following link:
- https://study.com/academy/lesson/software-architecture-design-transform-mapping.html

Component Based Development and Design

- Component-based development (CBD) is a procedure that accentuates the design and development of computer-based systems with the help of reusable software components.
- techniques involve:
  - procedures for developing software systems by choosing ideal off-the-shelf components
  - then assembling them using a well-defined software architecture.

Component Based Development and Design

- Key Goals
- The CBD intends to deliver better quality and output. The key goals are as follows:
- Save time and money when building large and complex systems: Developing complex software systems with the help of off-the-shelf components helps reduce software development time substantially. Function points or similar techniques can be used to verify the affordability of the existing method.
- Enhance the software quality
- Detect defects within the systems: The CBD strategy supports fault detection by testing the components; however, finding the source of defects is challenging in CBD.

CBD

CBD: specific routines:

- Component development
- Component publishing
- Component lookup as well as retrieval
- Component analysis
- Component assembly

CBD

- There are many standard component frameworks such as COM/DCOM, JavaBean, EJB, CORBA, .NET, web services, and grid services.
- These technologies are widely used in local desktop GUI application design such as graphic JavaBean components, MS ActiveX components, and COM components which can be reused by simply drag and drop operation.

USER INTERFACE DESIGN

Interface design

- It defines a set of **interface** objects, actions, and their screen representations that enable a **user** to perform all defined tasks in a manner that meets every usability objective defined for the system.
- Interface design is concerned with specifying the detail of the interface to an object or to a group of objects.

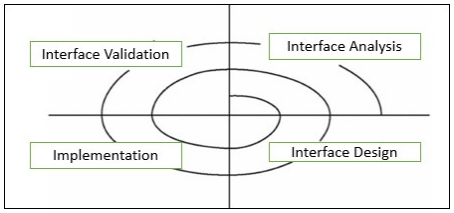
FOUR LEVELS IN UI DESIGN

- **The conceptual level** – It describes the basic entities considering the user's view of the system and the actions possible upon them.
- **The semantic level** – It describes the functions performed by the system i.e. description of the functional requirements of the system, but does not address how the user will invoke the functions.
- **The syntactic level** – It describes the sequences of inputs and outputs required to invoke the functions described.
- **The lexical level** – It determines how the inputs and outputs are actually formed from primitive hardware operations.

Stages in development of UI

User Interface Development Process

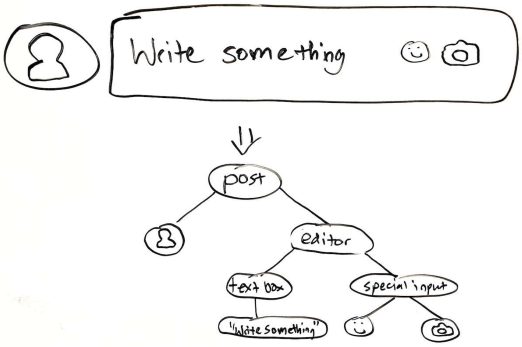
- It follows a spiral process as shown in the following diagram –
- It is iterative process.



Design & Analysis User Interface

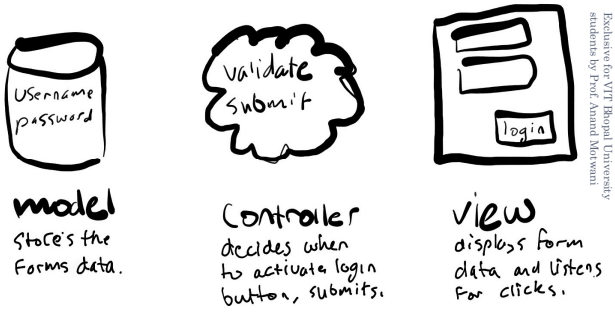
- It starts with task analysis which understands the user's primary tasks and problem domain. It should be designed in terms of User's terminology and outset of user's job rather than programmer's.
- To perform user interface analysis, the practitioner needs to study and understand four elements –
  - The **users** who will interact with the system through the interface
  - The **tasks** that end users must perform to do their work
  - The **content** that is presented as part of the interface
  - The **work environment** in which these tasks will be conducted

Example of FB UI



Model View Controller (MVC)

- A nearly ubiquitous way to implement the application is to follow a model-view-controller architecture.
- This architecture divides a user interface into three parts:



- The **MODEL** stores the *data* that a user interface is presenting. For example, in the figure above, the model stores the username and password a user is currently entering. (In many user interface toolkits, this wouldn't be stored in a separate database, but in the text field object in memory).
- The **VIEW** visualizes the data in the model. For example, in the figure above, the view is the form itself and the controls on it, including the text fields and login button. The view's job is to render the controls, listen for input from the user (e.g., clicking the login button), and display any output the controller decides to provide (e.g., form validation feedback).
- The **CONTROLLER** mediates implements the logic of the user interface. In our login form example above, that includes validating the username and password (e.g., neither can be empty), and submitting the information when the login button is pressed. The controller gets and stores data when necessary, and tells the view to update itself as the model changes.

77