SCHOOL OF COMPUTER SCIENCE AND IT

School of Computer Science & IT

Department of BCA

SOFTWARE ENGINEERING (22BCA3C01)

MODULE 4: Design Models QA

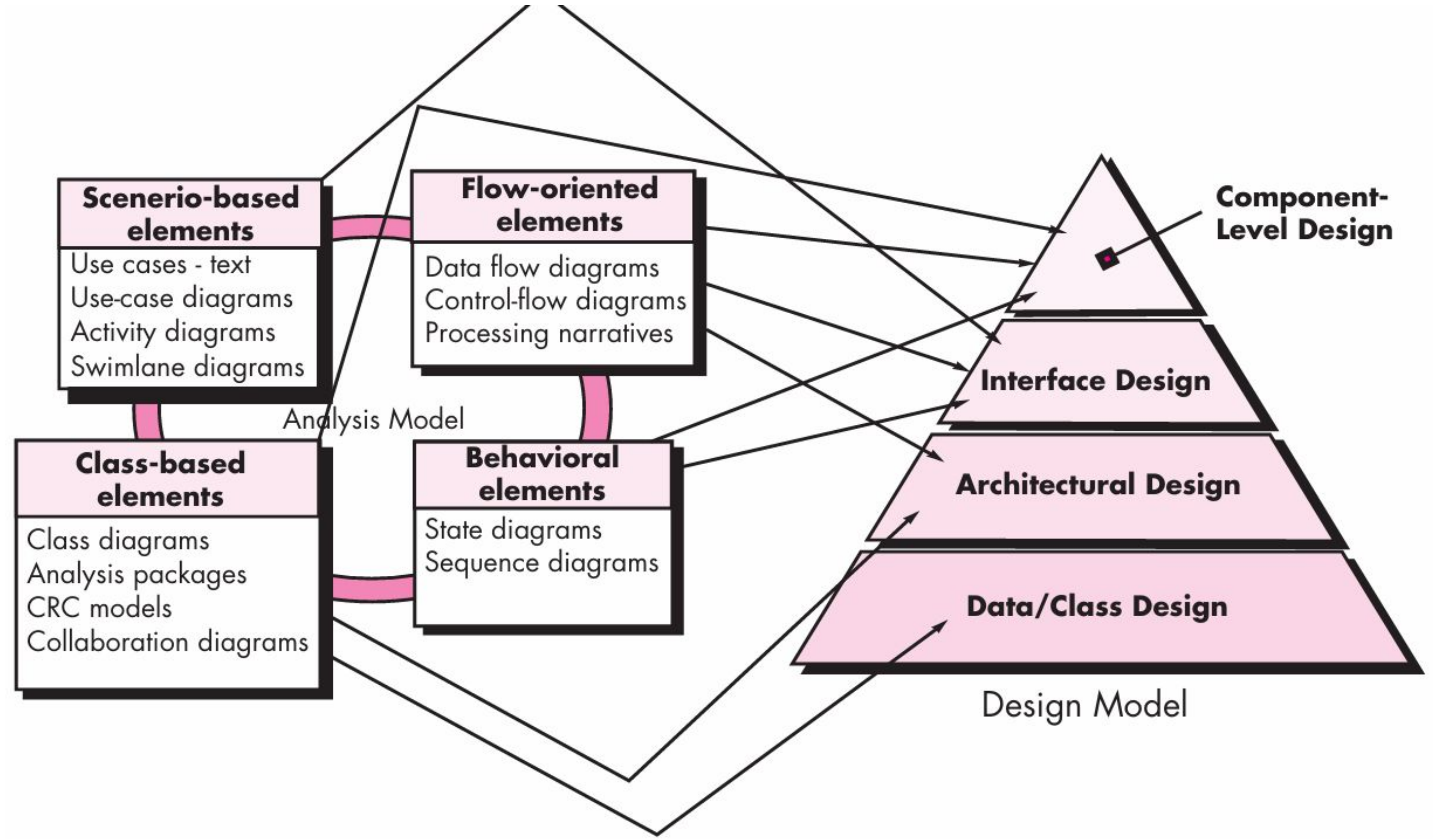
1. Map the components of Analysis model with the respective components of Software design Model?

- Analysis Model is a technical representation of the system. It acts as a link between the system description and the design model. In Analysis Modelling, information, behavior, and functions of the system are defined and translated into the architecture, component, and interface level design in the design modeling. An analysis model is a technical representation of a system that connects the system description to the design model. The design model builds on the analysis model by providing more detailed information about the system's structure and implementation.
- Here are some steps to translate an analysis model into a design model:
- **Define the system's information, behavior, and functions**
- **Transform the information into the design model's architecture, components, and interface level**
- **Refine the classes identified in the analysis model to include implementation constructs**
- **Create the four levels of design detail: data structure, system architecture, interface representation, and component level detail**
- In web engineering, analysis modeling helps to understand the customer's requirements and the shape of the web app that will address the problem. Design modeling helps to understand the internal structure of the web app.

2.What is Design?

- Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product.
- Design creates a representation or model of the software, but unlike the requirements model (that focuses on describing required data, function, and behavior), the **design model** provides **detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.**
- Design allows you to **model the system or product that is to be built**.
- Good software design should exhibit:
  - ❑ ***Firmness:*** A program should **not have any bugs** that inhibit its **function.**
  - ❑ ***Commodity:*** A program should be suitable for the **purposes** for which it was **intended.**
  - ❑ ***Delight:*** The experience of using the program should be **pleasurable** one.

3. Map or translate Analysis Model to Design Model?



- Once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).
- The flow of information during software design is illustrated in Figure 3.1. The requirements model, manifested by scenario-based, class-based, flow-oriented, and behavioral elements, feed the design task.
- The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action.
- The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.
- The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior .
- The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

Analysis modeling

- Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet.
- The requirements modeling action results in one or more of the following types of models:

• Scenario-based models of requirements from the point of view of various system "actors"

• Data models that depict the information domain for the problem

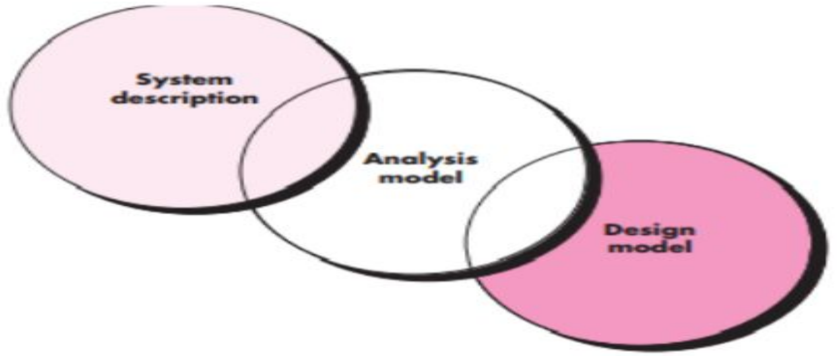
• Class-oriented models that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements

• Flow-oriented models that represent the functional elements of the system and how they transform data as it moves through the system

• Behavioral models that depict how the software behaves as a consequence of external "events" .

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs.

The requirements model as a bridge between the system description and the design model



4.Define Design and Quality?

- Design is the place where software quality is established.
- Design models can be assessed for quality and improved before code is generated, and tests are conducted.
- Software 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.

5.List the Quality Guidelines?

- A design should be modular; that is, the software should be logically partitioned into elements or subsystems
- A design should contain distinct representations of data, architecture, interfaces, and components.
- A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.

❖ These design guidelines are not achieved by chance. They are achieved through the application of fundamental design principles, systematic methodology, and thorough review.

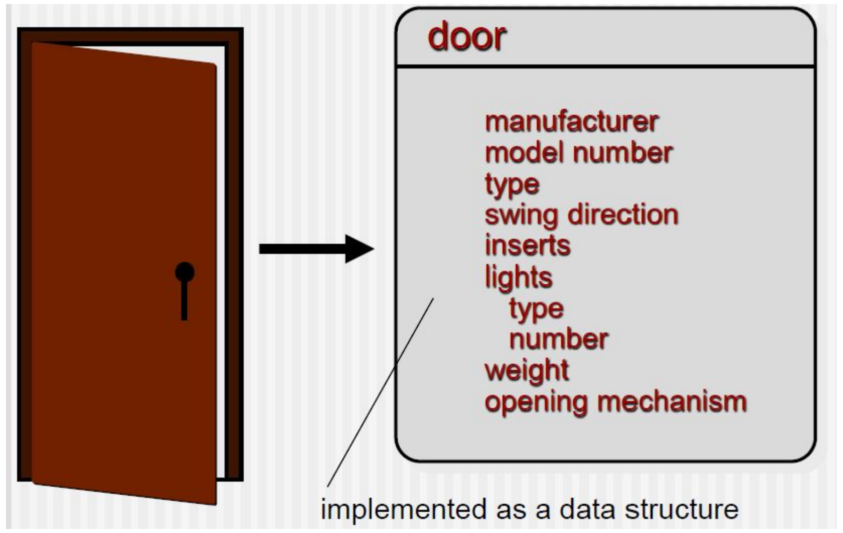
6. Explain Design Principles?

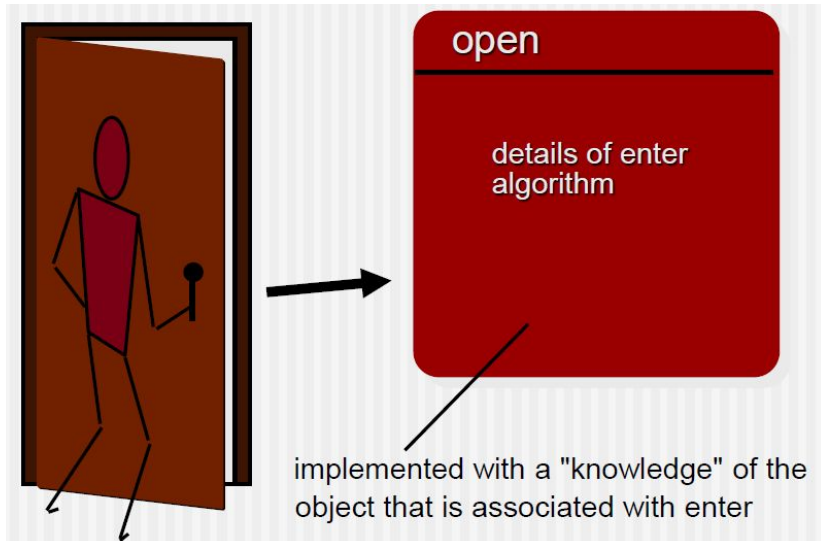
1. The design process should not suffer from 'tunnel vision.' i.e. A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job.
2. The design should be traceable to the analysis model.
3. The design should not reinvent the wheel. That means, it should not waste time or effort in creating things that already exist. For example, pattern-based design should be used.
4. The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world. That is, the structure of the software design should, whenever possible, mimic the structure of the problem domain. Design should reflect real-world closely.
5. The design should exhibit uniformity and integration. It should look as if developed by one person.
6. The design should be structured to accommodate change.
7. The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered. i.e. the software should display graceful degradation.
8. Design is not coding, coding is not design. Design is the description of the logic, which is used in solving the problem. Coding is the language specification which is implementation of the design.
9. The design should be assessed for quality as it is being created, not after the fact.
10. The design should be reviewed to minimize conceptual (semantic) errors.

1. Abstraction—data, procedure
2. Architecture—the overall structure of the software
3. Patterns—”conveys the essence” of a proven design solution
4. Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
5. Modularity—compartmentalization of data and function
6. Hiding—controlled interfaces
7. Functional independence—single-minded function and low coupling
8. Refinement—elaboration of detail for all abstractions
9. Refactoring—a reorganization technique that simplifies the design
10. OO design concepts—such as classes and objects, inheritance, messages, and polymorphism, among others
11. Design Classes—provide design detail that will enable analysis classes to be implemented

7.Explain various Fundamental Concepts of Design?

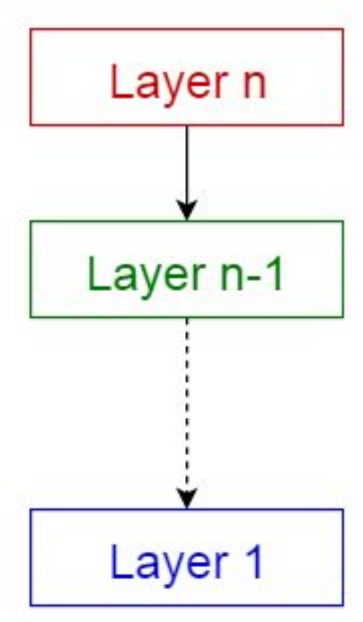
Abstraction




open
details of enter algorithm
implemented with a "knowledge" of the object that is associated with enter

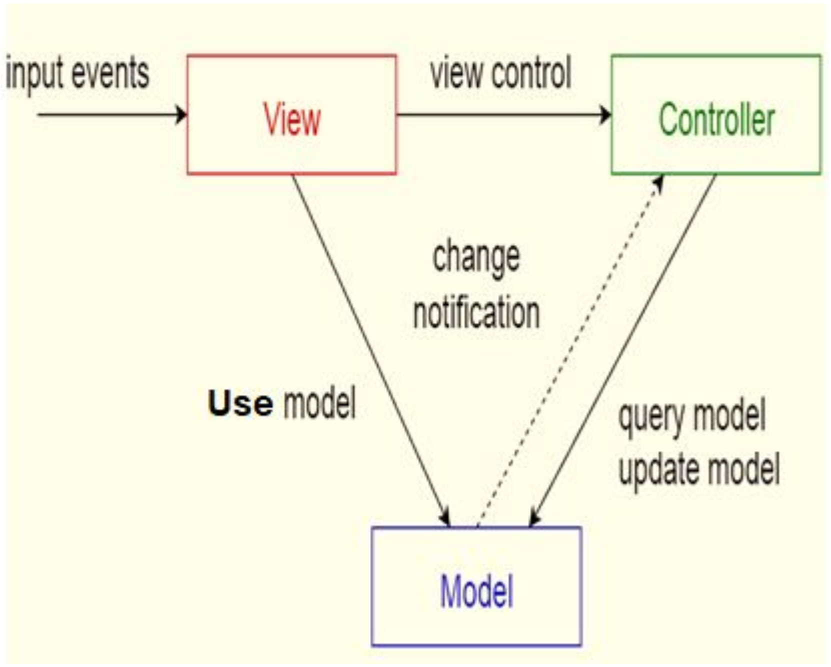
- **Abstraction is a mechanism to understand a complex thing while hiding the irrelevant details of it.**
- **Abstraction has been playing a major role in advancement of software engineering to simplify the design and development of software program.**
- **When you use better abstraction, you model the real-world more closely in the software, write less code, make less error, and solve more complex problem in less time.**
- **Class is an abstraction to model real-world objects in software.**
- **Functions in C program is an abstraction to model real-world processes / procedures in software.**
- **Data Abstraction and Procedural Abstraction**

Architecture



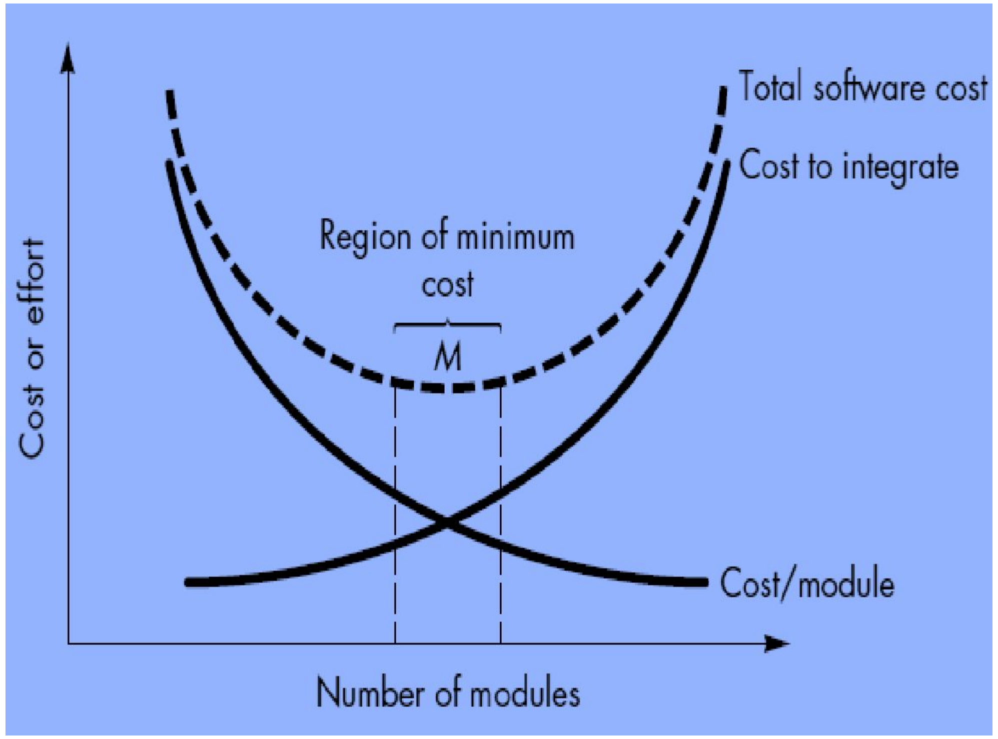
- Architecture defines the components of a system (e.g., modules, objects/classes) and the manner in which those components are packaged and interact with one another.
- The architectural design description should address requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems.

Design patterns



- In software engineering, a **design pattern** is a reusable solution to a commonly occurring problem in software design. It is a description or template for how to solve a problem that can be used in many different situations.
- Design patterns can speed up the development process by providing tested, proven design solutions.
- Effective software design requires use of design patterns that helps to prevent subtle issues that can cause major problems and improves code quality.
- Design patterns allow developers to communicate effectively among themselves as they are well-known, well understood names for software design.
- Example: MVC (Model-View-Controller): This pattern divides an user interface in to 3 parts:
  1. **model** — holds the data
  2. **view** — accepts inputs from the user and displays the information to the user
  3. **controller** — handles the input from the user and coordinates the interaction between the model and view
- It decouples components and allows efficient code reuse. Used heavily in Java Swing API
- Fundamental OO pattens (GoF) : Singleton, Builder, Factory, Adapter, Prototype etc.

Modularity: Trade-off



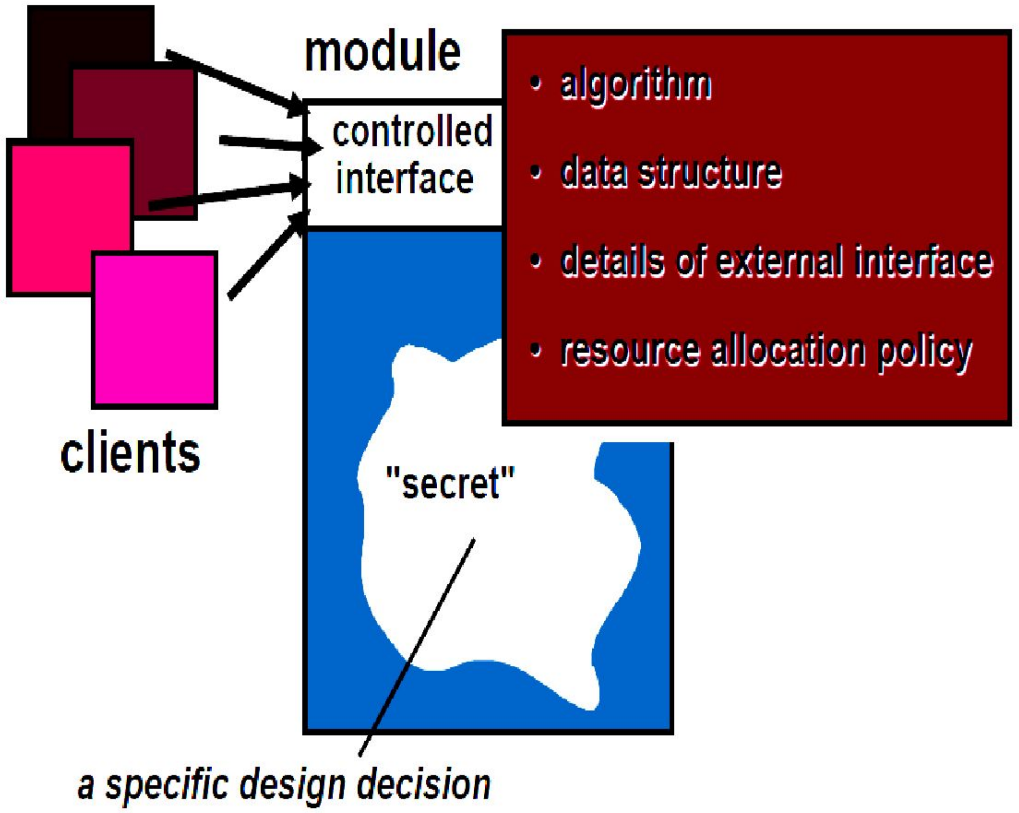
❑ Separation of Concerns

- Any complex problem can be more easily handled if it is subdivided into pieces such that each can be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

❑ Modularity

- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
- The overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

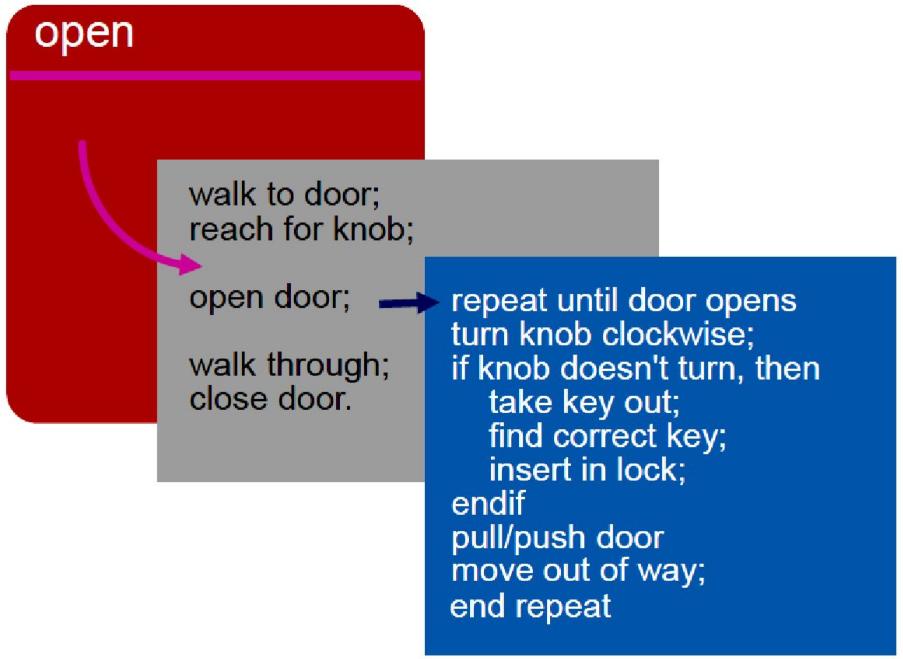
Information Hiding


module
controlled interface
• algorithm
• data structure
• details of external interface
• resource allocation policy
clients
"secret"
a specific design decision

Modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

- Why Information Hiding?
  - reduces the likelihood of "side effects" – accidental / deliberate modification.
  - limits the local design decisions
  - emphasizes communication through controlled interfaces
  - discourages the use of global data
  - leads to encapsulation—an attribute of high quality design
  - results in higher quality software

Stepwise Refinement



```
repeat until door opens
turn knob clockwise;
if knob doesn't turn, then
    take key out;
    find correct key;
    insert in lock;
endif
pull/push door
move out of way;
end repeat
```

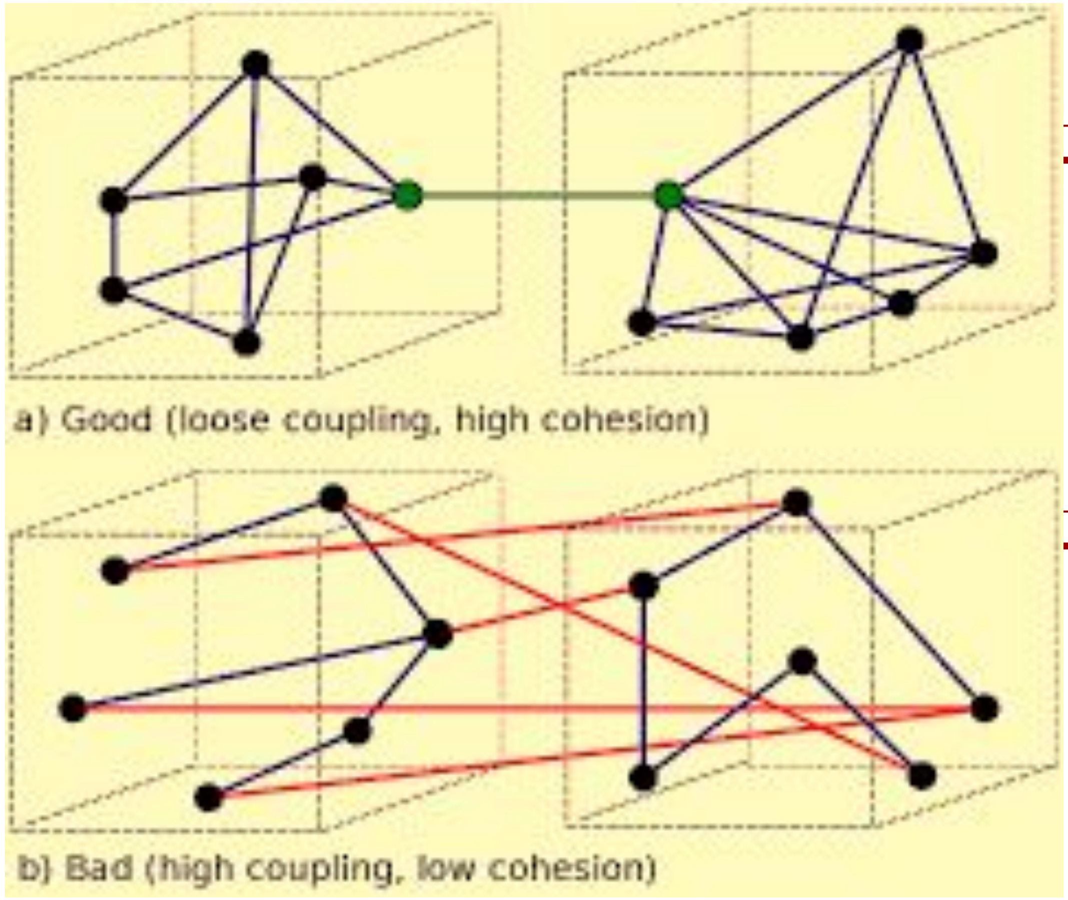
- *Stepwise refinement* is a top-down design strategy.
- A program is developed by successively refining levels of procedural detail.
- A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.
- Refinement is actually a process of *elaboration.* We begin with a statement of function (or description of information) that is defined at a high level of abstraction.
- 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 reveal low-level details as design progresses.
- Both concepts aid the designer in creating a complete design model as the design evolves.

Refactoring

- "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."

- When software is refactored, the existing design is examined for
  - redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures
  - or any other design failure that can be corrected to yield a better design

Functional Independence


a) Good (loose coupling, high cohesion)
b) Bad (high coupling, low cohesion)

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion“ to excessive interaction with other modules.

❑ *Cohesion* is an indication of the relative functional strength of a module.

- ❖ A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.

❑ *Coupling* is an indication of the relative interdependence among modules.

- ❖ 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.

- Cohesion in software engineering refers to how closely related the responsibilities and functions of a single module. In other words, it **measures how well the elements within a module work together to achieve a single purpose**. High cohesion means that a module has a clear, focused purpose and its parts are directly related to that purpose. This is desirable because highly cohesive modules are easier to understand, maintain, and test.
- **Example** : Consider a module that handles all the operations related to user authentication, such as logging in, logging out, and managing passwords. If these tasks are all contained within one module, that module has high cohesion.

Coupling

- Coupling in software engineering refers to the **degree of interdependence between two or more modules in a syste**m. It indicates how much one module relies on the details of another. Low coupling is preferable because it means changes in one module are less likely to impact others, making the system more flexible and easier to maintain.
- **Example** : If a user authentication module is designed in such a way that it can work independently of other modules, such as the user profile or the database, then it has low coupling. This independence means that changes to the authentication module won't necessitate changes to other parts of the system.

8.What is software architecture?

- The architecture of a system describes the structure of the software in terms of its components and how they fit together.
- It is the structure of a system and the manner in which data and procedural components collaborate with one another.
- The software architecture is the structure of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

9. **Expand the role of software architectural design? Or** Why Software Architecture?

- The software architecture is a representation that enables a software engineer to

  (1) analyze the effectiveness of the design in meeting its stated requirements,

  (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and

  (3) help reducing the risks associated with the construction of the software.

- Software Architecture is important because:
  - Representations of software architecture are an enabler for **communication between all parties (stakeholders) interested in the development of a computer-based system**.
  - The architecture highlights **early design decisions** that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
  - Architecture provides a **graspable model** of how the system is structured and how its components work together.

10. What are the different architectures required in architectural styles.

- The software that is built for computer-based systems also exhibits one of many architectural styles.
- An *architectural style* is used as a descriptive mechanism to differentiate the software architecture from other styles.
- Each 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, coordination and cooperation" among components;

  (3) *constraints* that define how components can be integrated to form the system; and

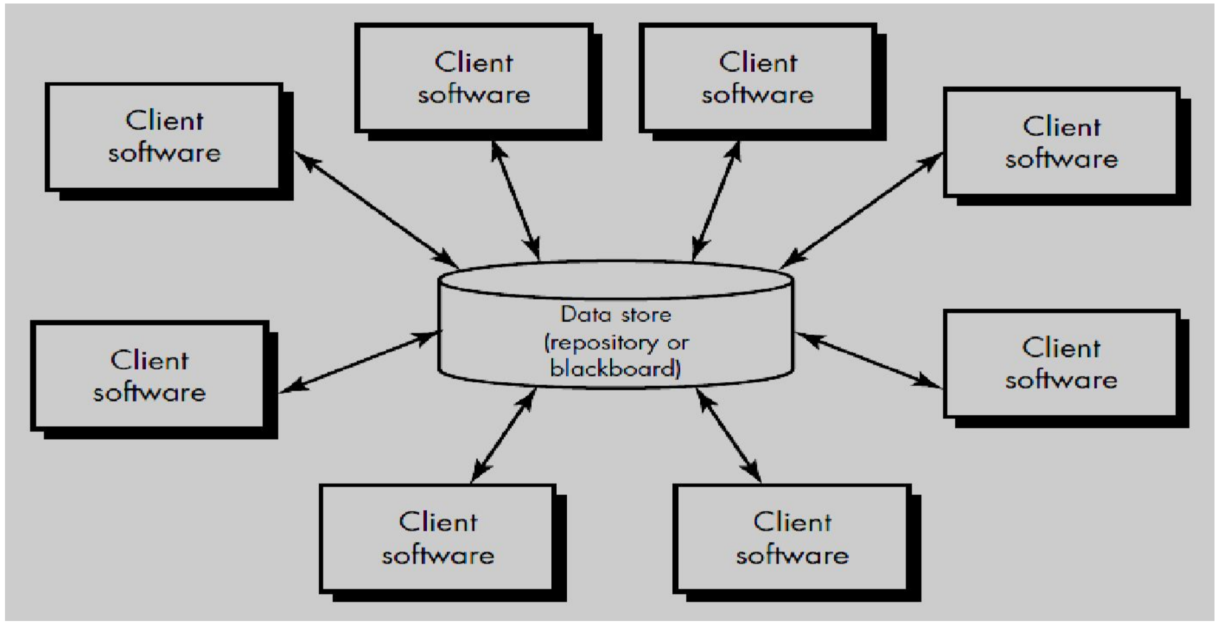
  (4) *semantic models* that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

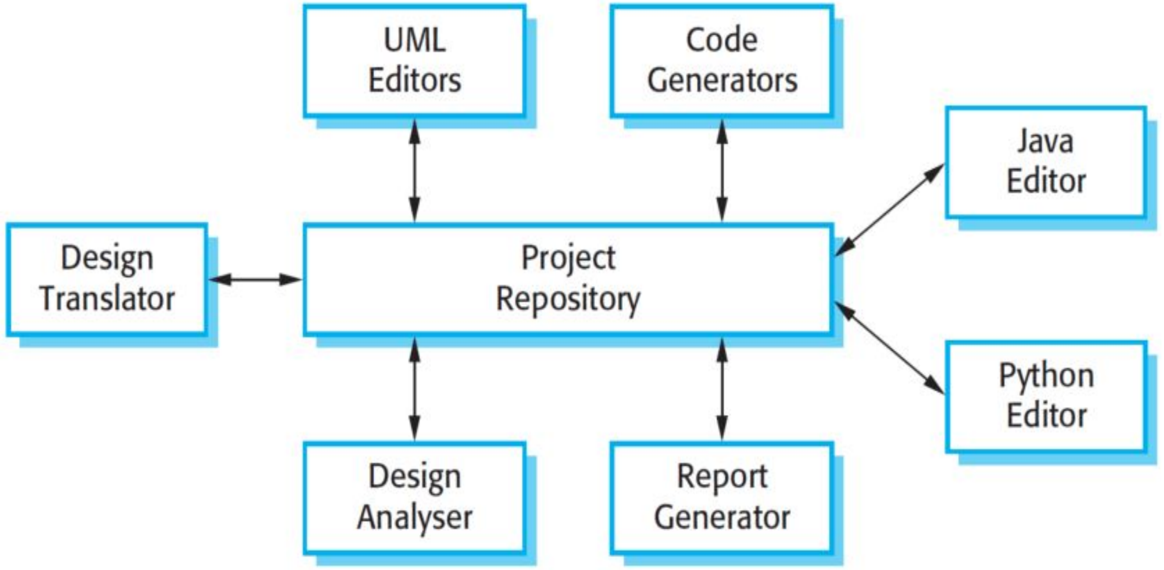
❑ Different Architectural styles:

  - Data-centered architectures
  - Data flow architectures
  - Call and return architectures
  - Layered architectures
  - Client-server architectures

  ❖ Sometime, a Hybrid Architecture uses multiple styles

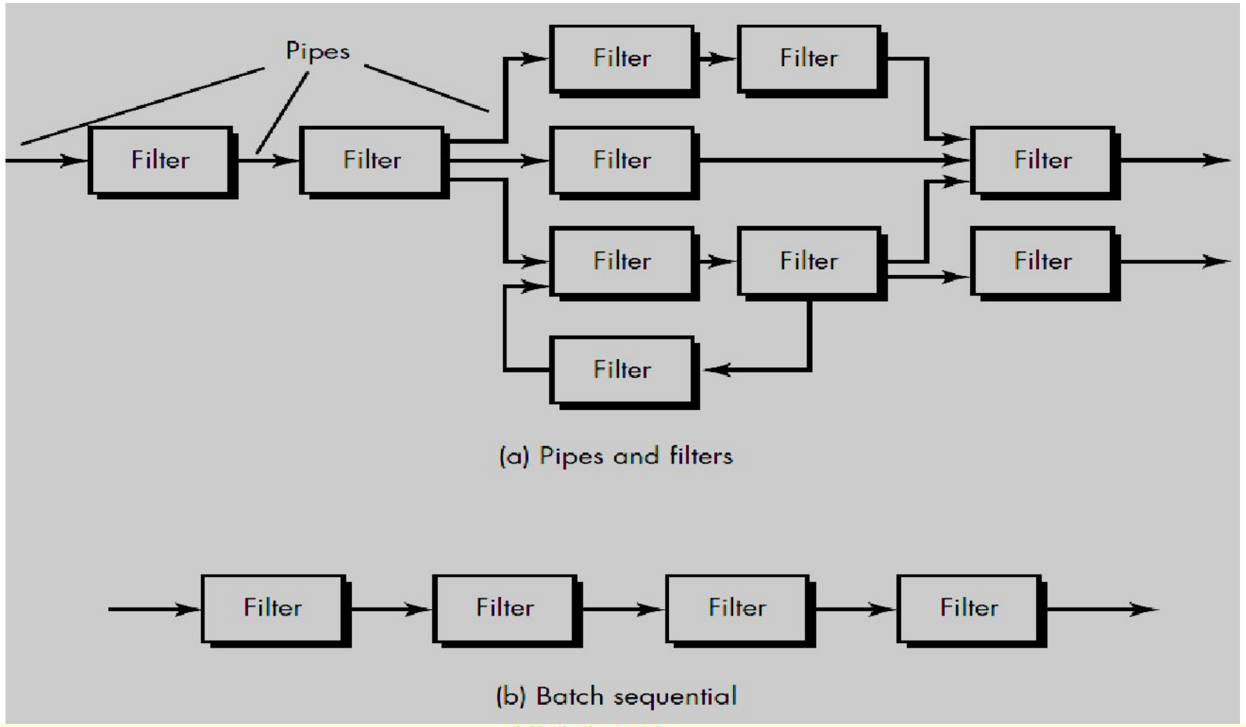
Data-Centered architectures





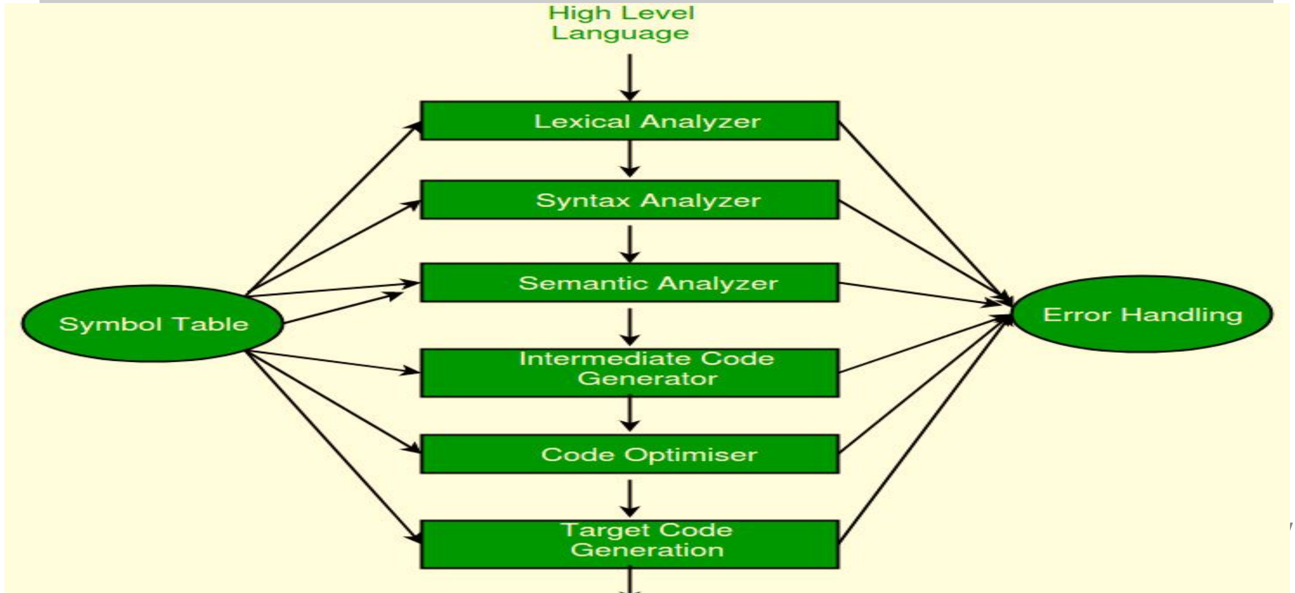
- A shared data store with multiple components interacting independently.
- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store.
- The client software/ component accesses the data independent of any changes to the data or the actions of other client software.
- Data-centered architectures promote ***integrability***. That is, existing components can be changed and new client components can be added to the architecture without concern about other clients (because the client components operate independently).
- Example: IDE

Data-flow architectures



(a) Pipes and filters

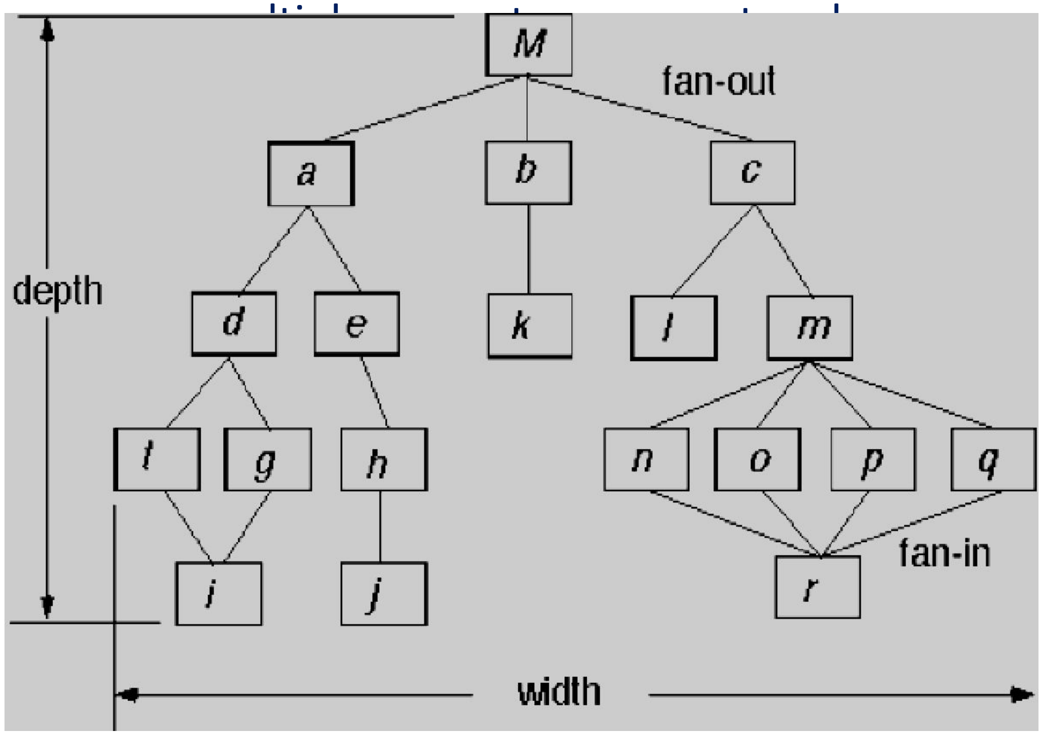
(b) Batch sequential

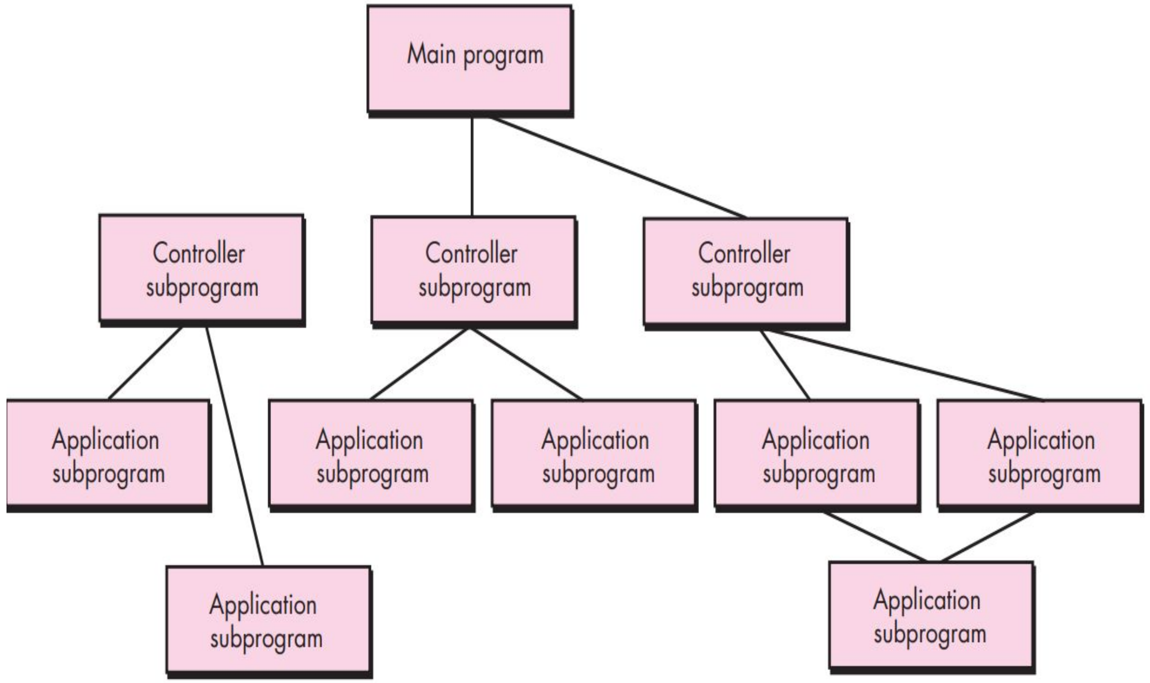


- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.
- A *pipe and filter pattern* has a set of components, called *filters,* connected by *pipes* that transmit data from one component to the next.
- Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.
- However, 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.*** This pattern accepts a batch of data and then applies a series of sequential components (filters) to transform it.
- Example: Text-based utilities (sort, awk, sed etc.) in the UNIX Shell. Compiler (lexical analysis, tokenization, syntax analysis, semantic analysis, code generation)

Call and return architectures

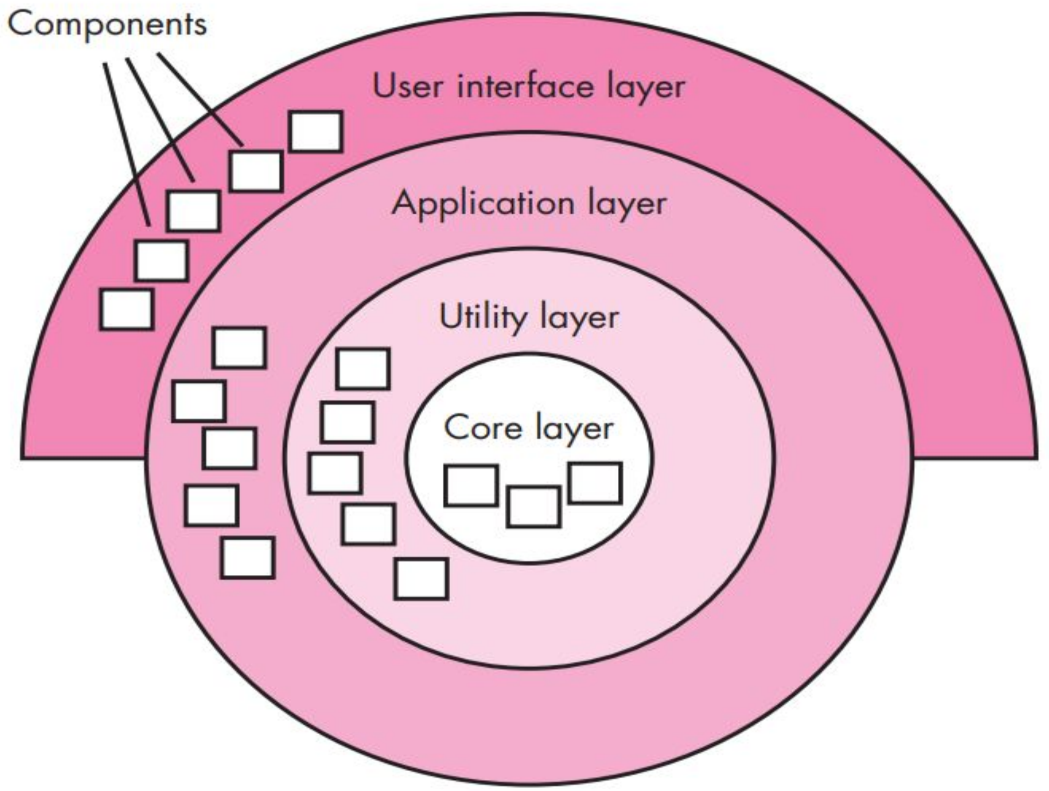
- This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. A number of substyles exist within this category:
- ***Main program/subprogram architectures*** : This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components.
- ***Remote procedure call architectures***:The components of a main program/ subprogram architecture are distributed



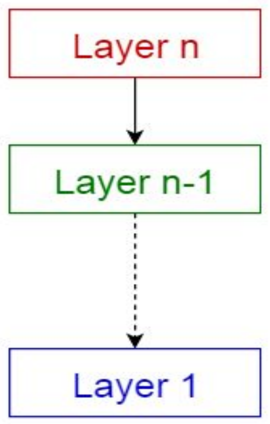


Main program/subprogram architecture

Layered architectures



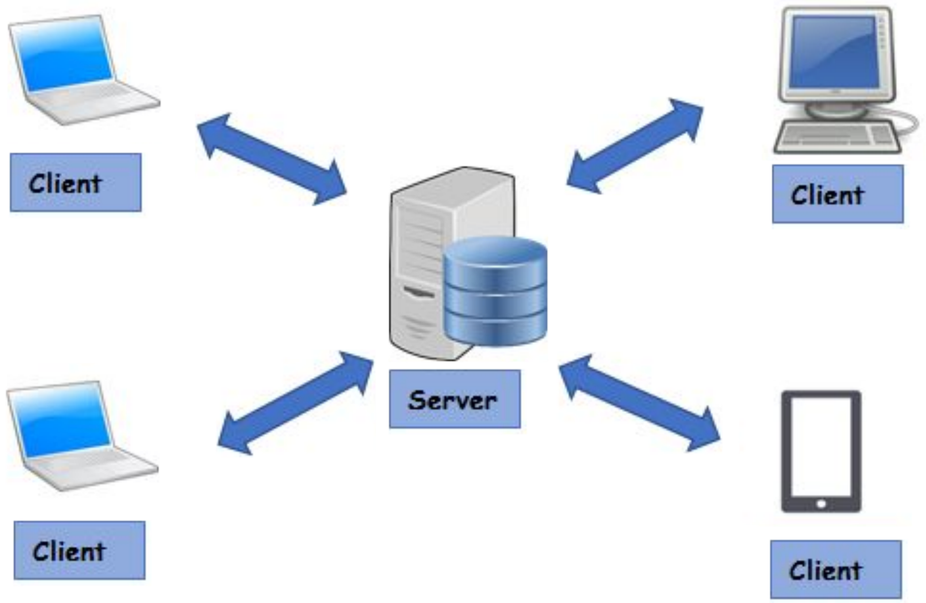
Layered architecture

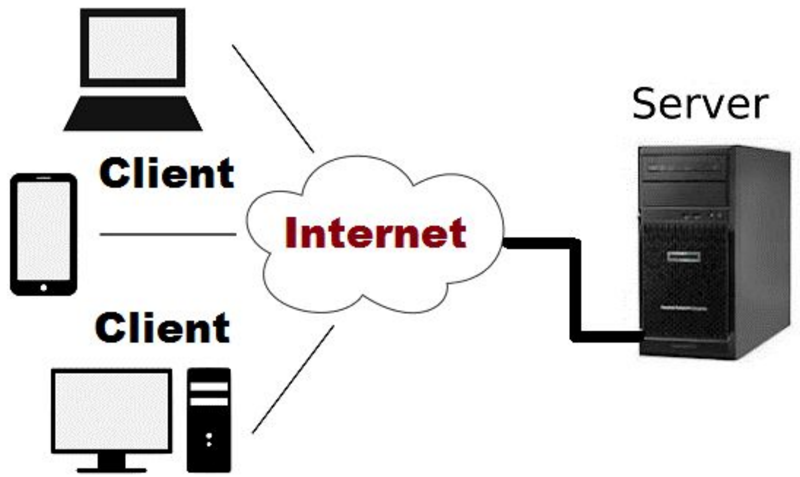


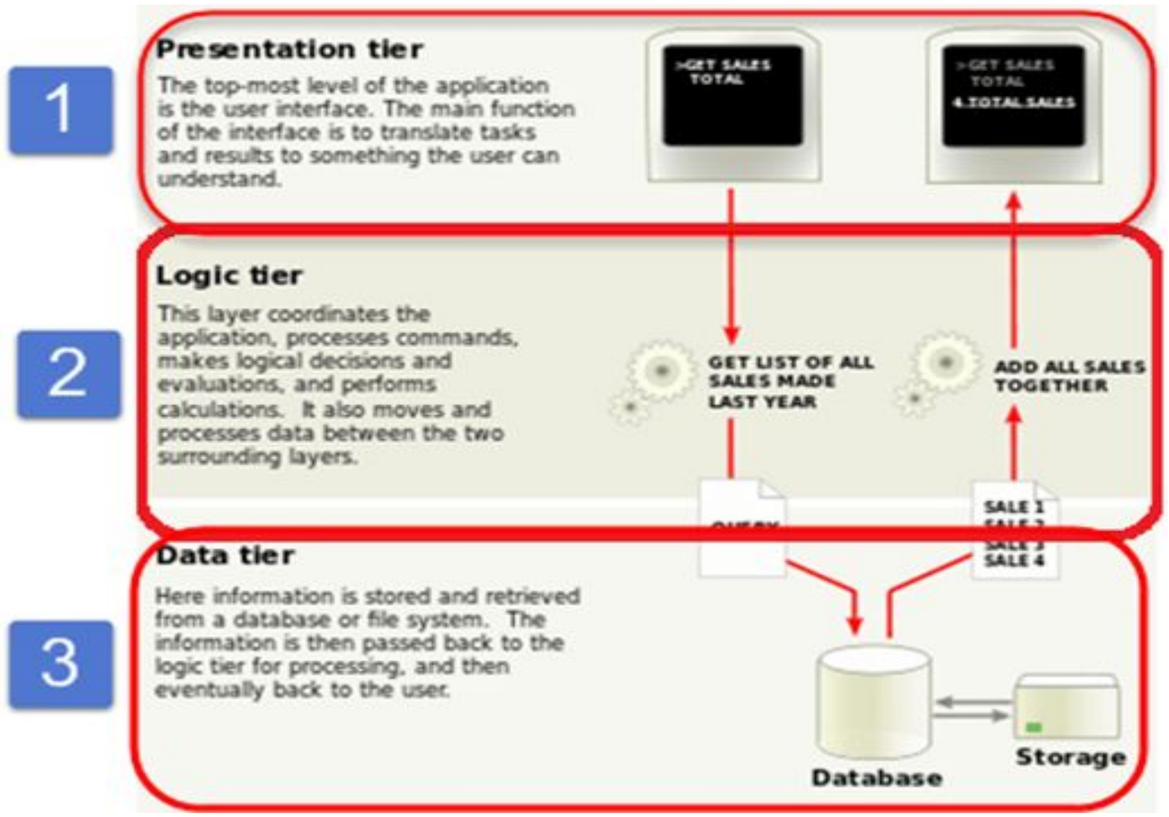
- A number of different layers are defined with each layer performing a well-defined set of operations that progressively become closer to the machine instruction set..
- At the outer layer, components provide user interface operations.
- At the inner layer, components perform operating system interfacing.
- Intermediate layers provide utility services and application software functions.
- Example: OS, Network s/w, Web apps

Client-Server Architectures

- This style consists of two components; a **server** and multiple **clients**. The server component will provide services to multiple client components. Clients request services from the server and the server provides relevant services to those clients. Furthermore, the server continues to listen to client requests.
- Clients can be : Thick-client (traditional apps) or Thin-client (e.g. web apps)







Presentation tier

The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

11. Explain the steps required for architectural mapping using data flow?

some steps for architectural mapping using data flow:

- **Review the system model**: Review the fundamental system model.
- **Refine data flow diagrams**: Review and refine the software's data flow diagrams (DFDs). Determine if the DFD has transaction or transform flow.
- **Isolate the transform center**: Specify the incoming and outgoing flow boundaries to isolate the transform center.
- **Perform factoring**: Perform first-level and second-level factoring.

Refine the architecture: Refine the first-iteration architecture using design heuristics to improve software quality.

- Data flow mapping is a high-level look at a system's architecture. It can provide insight into the security of the system's processes. Here are some tips for data flow mapping:

Ensure accuracy

- Accuracy is critical in data flow mapping. A single mistake can have serious consequences, especially when dealing with personal data.

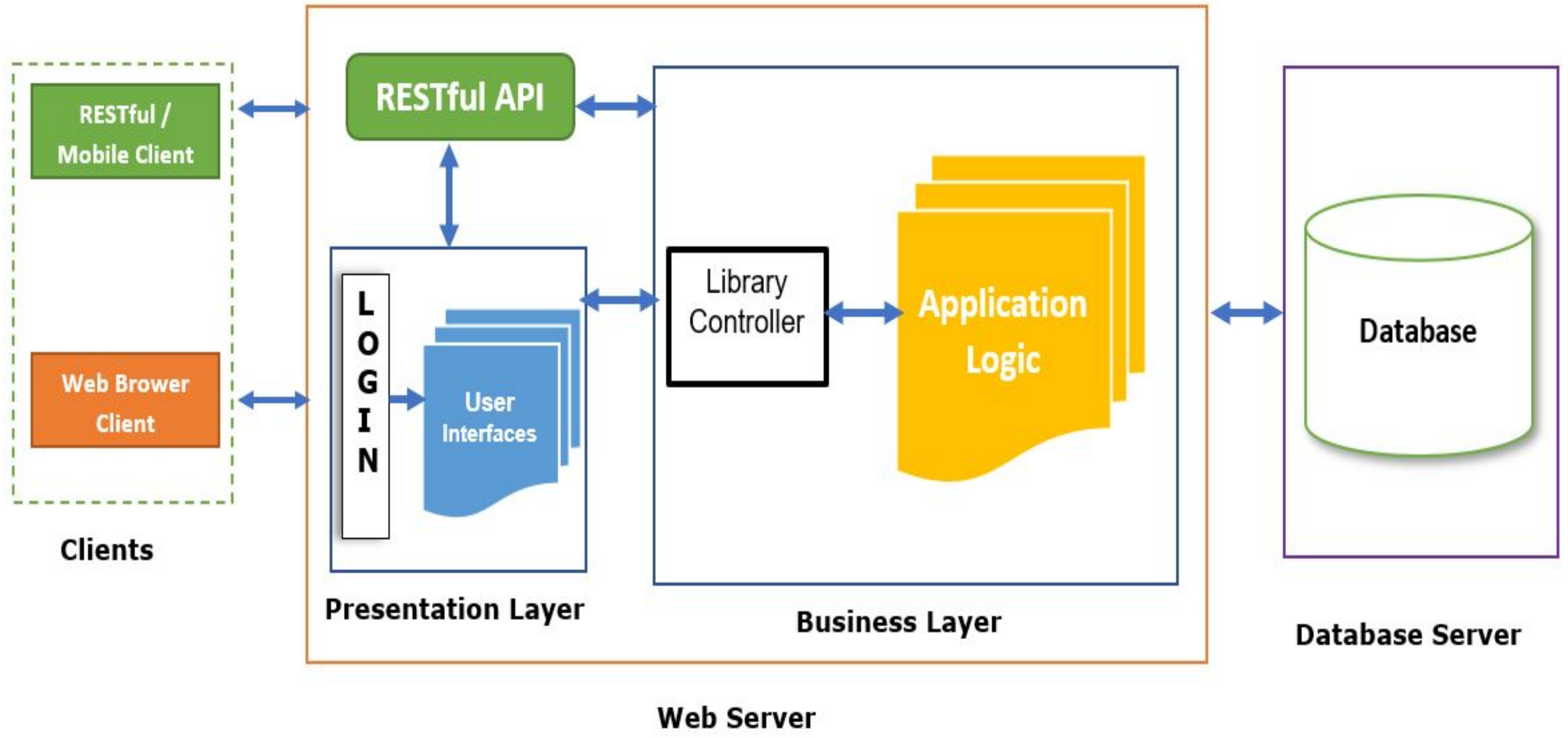
Validate and verify

- Validate and verify the DFD to enhance its accuracy and reliability. This iterative process leads to a robust representation of the data flow within the system

12.Explain object oriented view and traditional view in a component level design.

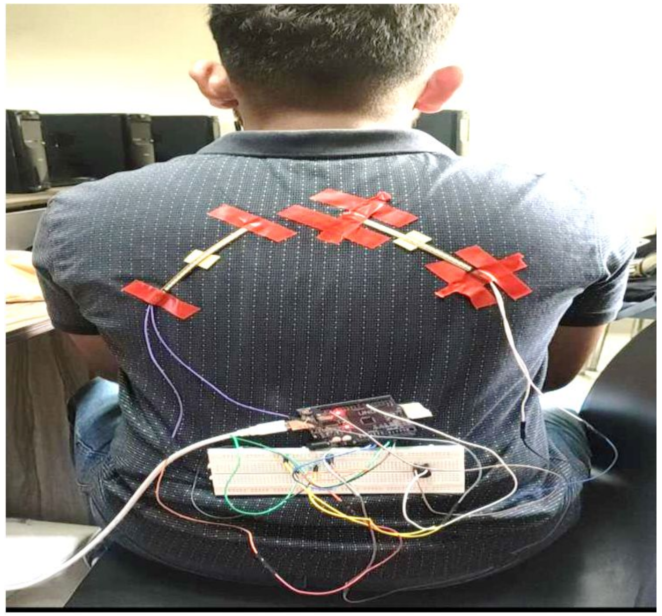
- In the object-oriented view, a component contains a set of collaborating classes. As a quick reminder, classes are groups of objects with common properties, operations, and relationships to other objects and meanings. Analysis classes can be thought of as classes that have to do with the real world. In component-level design, the object-oriented view and the traditional view differ in how they define a component:
- **Object-oriented view**: A component is a collection of classes that work together.
- **Traditional view**: A component is a functional element of a program that includes processing logic, internal data structures, and an interface.
- Object-oriented design is a methodology that uses object modeling techniques to design a system. It's best suited for projects that use emerging object technologies to construct, manage, and assemble objects into computer applications.
- Object-oriented analysis organizes requirements around objects, which integrate both behaviors and states. This is different from traditional analysis methodologies, which consider processes and data separately.

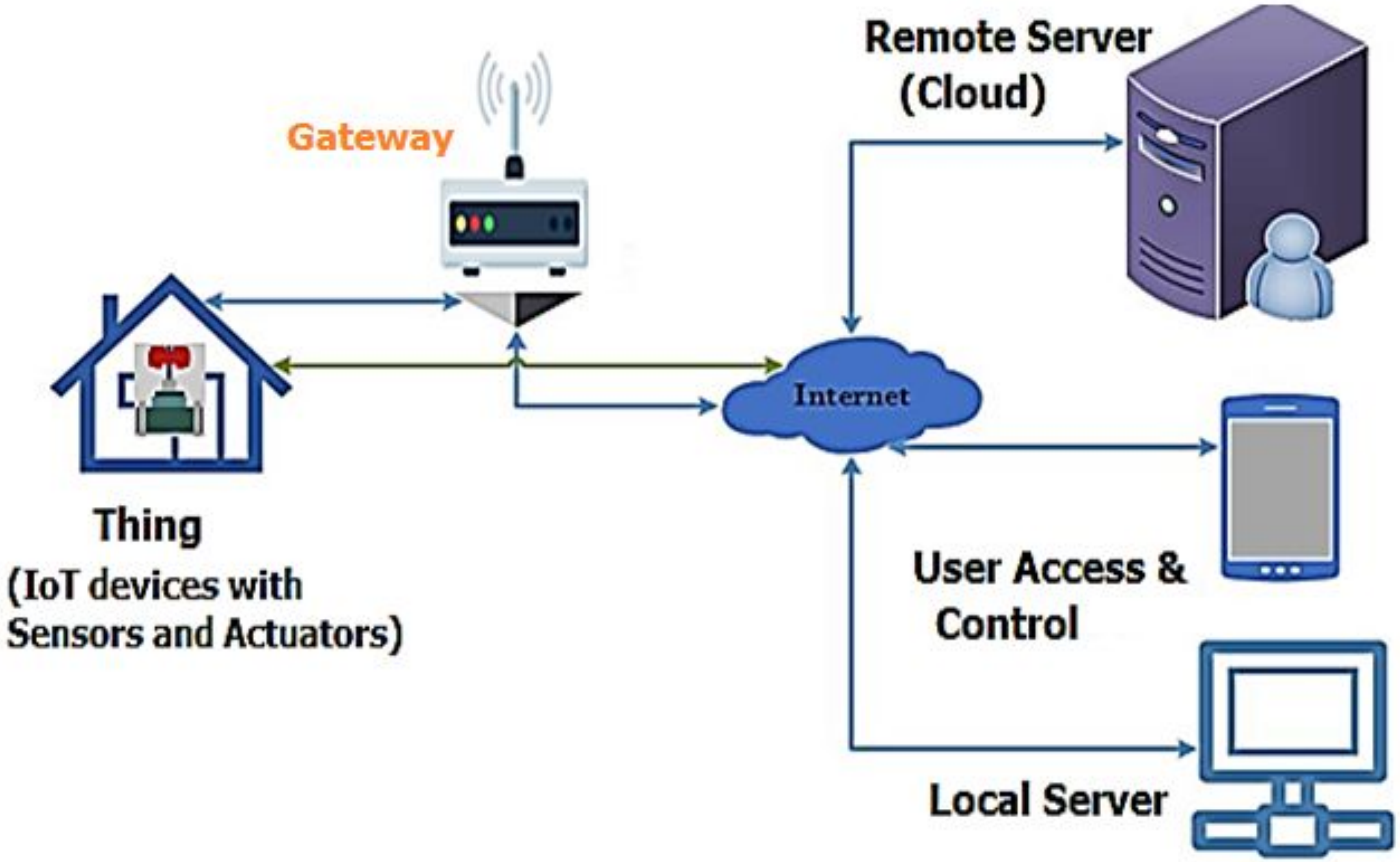
13.Example: Library Case Study- Architecture



Representational State Transfer (REST) is a software architecture that imposes conditions on how an API should work.

14.Example: IoT System Architecture





15.Define a Component?

- A component is a modular building block for computer software.
- More formally, the OMG Unified Modeling Language Specification(object management group) defines a component as "a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

An Object-Oriented View

- In the context of object-oriented software engineering, a component contains a set of

collaborating classes. Each class within a component has been fully elaborated to include all

attributes and operations that are relevant to its implementation. As part of the design

elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined. To accomplish this, you begin with the requirements model and elaborate analysis classes and infrastructure classes.

The Traditional View

- In the context of traditional software engineering, a component is a functional elementof a program that incorporates processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it. A traditional component, also called a module, resides within the
- software architecture and serves one of three important roles:

(1)A control component that coordinates the invocation of all other problem domain components,

(2)a problem domain component that implements a complete or partial function that is required by the customer, or

(3)an infrastructure component that is responsible for functions that support the processing required in the problem domain.

16.Explain Basic Design Principles?

- There are five object-oriented design (OOD) principles by Robert C. Martin. These principles enable good quality maintainable , extensible software.Popularly known as **SOLID**, it stands for:

- **S** - **Single-responsibility Principle**
  - A module / component / class **should have one and only one reason to change**, meaning that a class should have only one job.
- **O** - **Open-closed Principle**
  - A module / component should be **open for extension** but **closed for modification**. This means that a class **should be extendable without modifying the class itself**.
- **L** - **Liskov Substitution Principle**
  - Subclasses should be substitutable for their base classes.
  - LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it.

- I - Interface Segregation Principle
  - Many client-specific interfaces are better than one general purpose interface.
  - There are many instances in which multiple client components use the operations provided by a server class.
  - ISP suggests that you should create a specialized interface to serve each major category of clients.
  - Only those operations that are relevant to a particular category of clients should be specified in the interface for that client.
  - If multiple clients require the same operations, it should be specified in each of the specialized interfaces.
- D - Dependency Inversion Principle
  - **Depend on abstractions. Do not depend on concretions**. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

17.What are the Design Guidelines?

- Components
  - Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model.
- Interfaces
  - Interfaces provide important information about communication and collaboration. They should follow design principles.
- Dependencies and Inheritance
  - For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes). Following the philosophy of the OCP(Open-Closed Principle), this will help to make the system more maintainable.

Cohesion and Coupling

- Cohesion is the "single-mindedness" of a module.
  - Cohesion implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.
- Coupling is the degree to which a component is connected to other components and to the external world.
  - It is a qualitative measure of the degree to which classes are connected to one another.

❖ A good design needs high cohesive components and low coupled components.

❖ When components become highly cohesive they tend to become highly coupled.

❖ Hence a trad-off of cohesion and coupling is required.

18. What are the steps required for conducting component level design?

- Step 1. Identify all design classes that correspond to the problem domain.
- Step 2. Identify all design classes that correspond to the infrastructure domain.
- Step 3. Elaborate all design classes that are not acquired as reusable components.
- Step 3a. Specify message details when classes or component collaborate.
- Step 3b. Identify appropriate interfaces for each component.
- Step 3c. Elaborate attributes and define data types and data structures required to implement them.
- Step 3d. Describe processing flow within each operation in detail.
- Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.
- Step 5. Develop and elaborate behavioral representations for a class or component.
- Step 6. Elaborate deployment diagrams to provide additional implementation detail.
- Step 7. Factor every component-level design representation and always consider alternatives.

19.Define Component Diagram and how component diagram can be modelled?

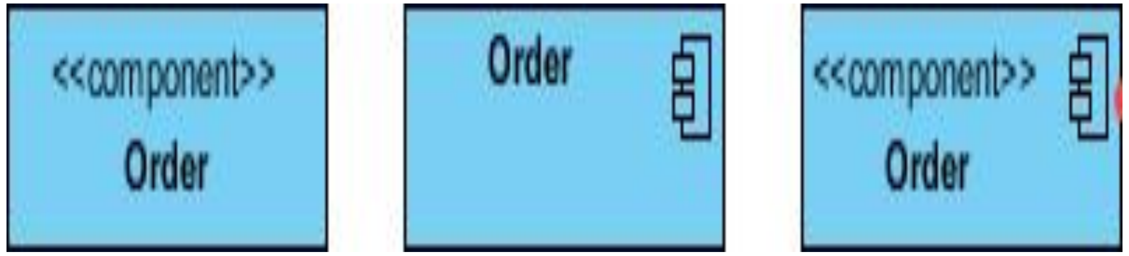
- The component diagram's main purpose is to show the structural relationships between the components of a system.

- Component diagrams allow an architect to verify that a system's required functionality is being implemented by components, thus ensuring that the eventual system will be acceptable.
- Developers find the component diagram useful because it provides them with a high-level, architectural view of the system that they will be building, which helps developers begin formalizing a roadmap for the implementation, and make decisions about task assignments and/or needed skill enhancements.
- A component diagram breaks down the actual system under development into various high levels of functionality.
- Each component is responsible for one clear aim within the entire system and only interacts with other essential elements on a need-to-know basis.

Basic Concepts of Component Diagram

- A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. In UML 2, a component is drawn as a rectangle with optional compartments stacked vertically. A high-level, abstracted view of a component in UML 2 can be modeled as:

- A rectangle with the component's name
- A rectangle with the component icon
- A rectangle with the stereotype text and/or icon

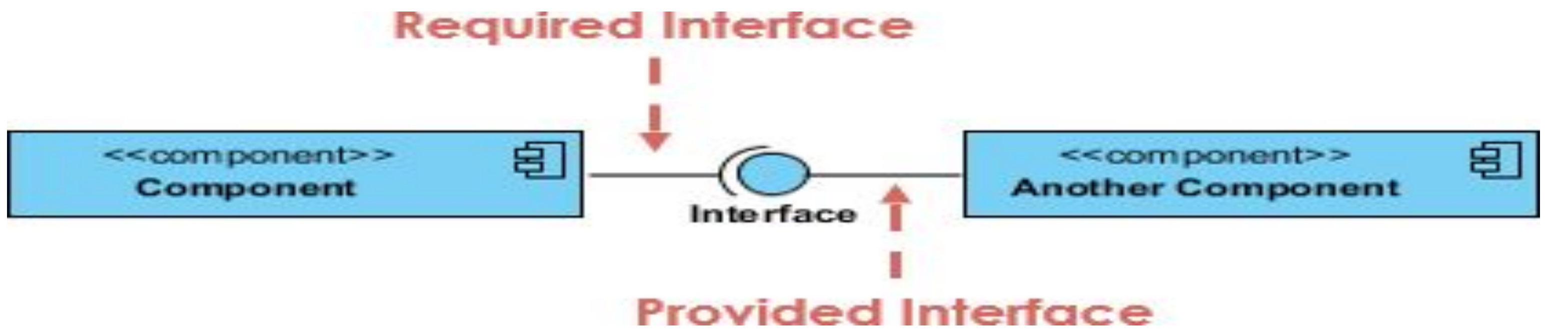


Interface

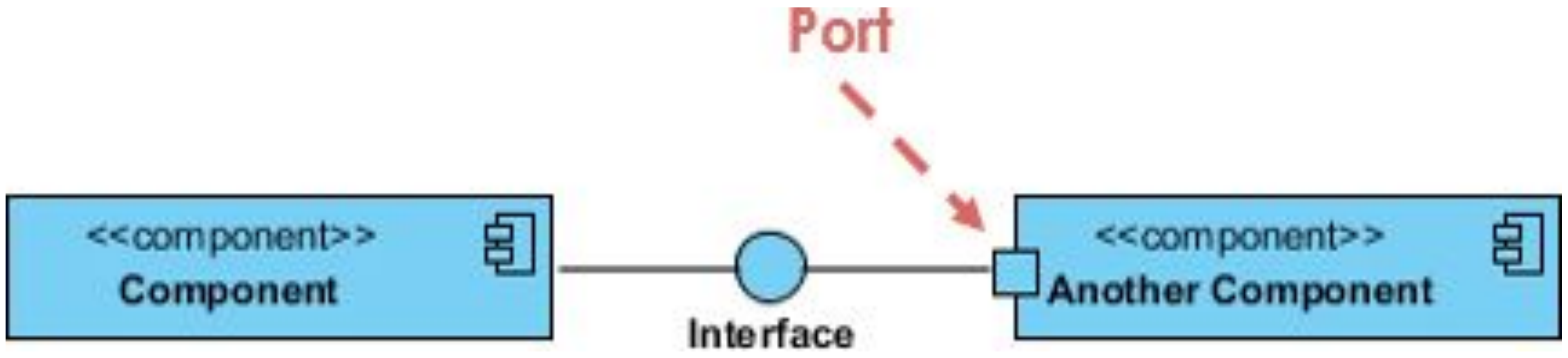
In the example below shows two type of component interfaces:

Provided interface symbols with a complete circle at their end represent an interface that the component provides - this "lollipop" symbol is shorthand for a realization relationship of an interface classifier.

Required Interface symbols with only a half circle at their end (a.k.a. sockets) represent an interface that the component requires (in both cases, the interface's name is placed near the interface symbol itself).

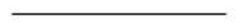
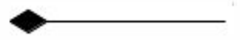
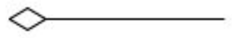
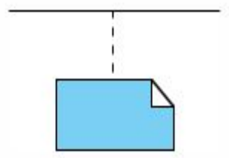
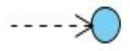
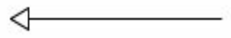


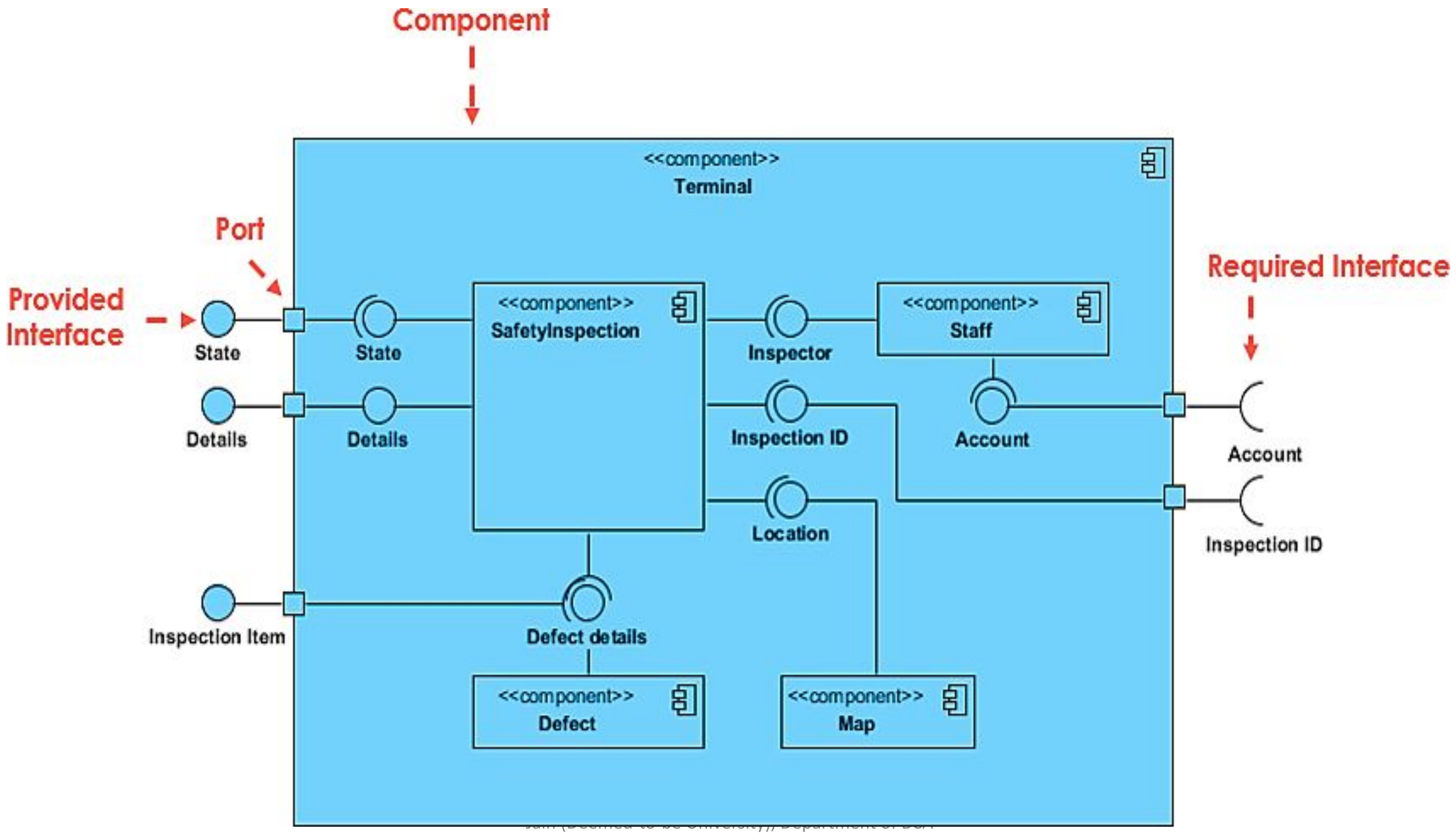
Port : Ports are represented using a square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component.



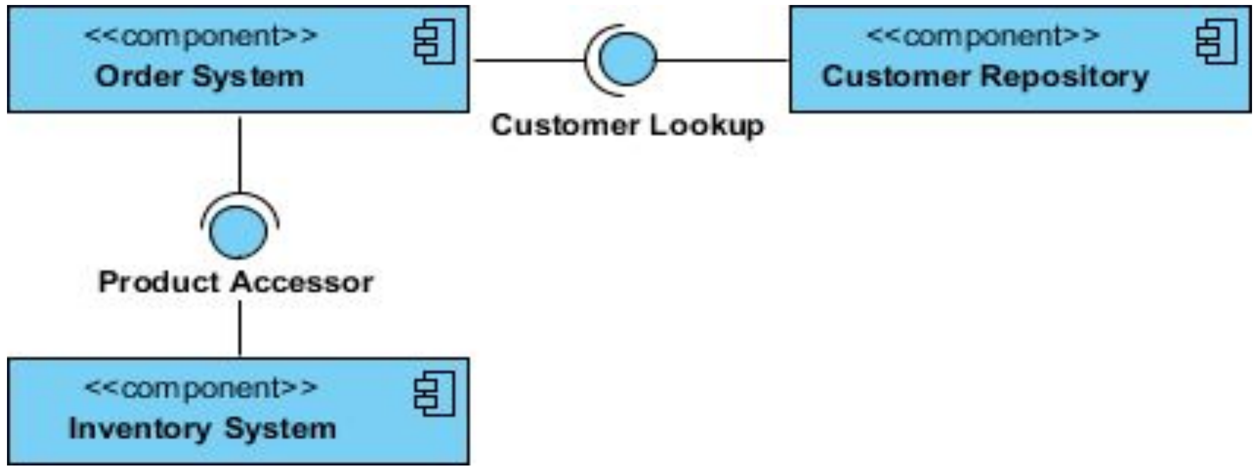
Relationships

Graphically, a component diagram is a collection of vertices and arcs and commonly contain components, interfaces and dependency, aggregation, constraint, generalization, association, and realization relationships. It may also contain notes and constraints.

Relationships	Notation
Association: • An association specifies a semantic relationship that can occur between typed instances. • It has at least two ends represented by properties, each of which is connected to the type of the end. More than one end of the association may have the same type.	
Composition: • Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. • If a composite is deleted, all of its parts are normally deleted with it.	
Aggregation • A kind of association that has one of its end marked shared as kind of aggregation, meaning that it has a shared aggregation.	
Constraint • A condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element.	
Dependency • A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. • This means that the complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s).	
Links: • A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. • Each instance of the specific classifier is also an indirect instance of the general classifier. • Thus, the specific classifier inherits the features of the more general classifier.	

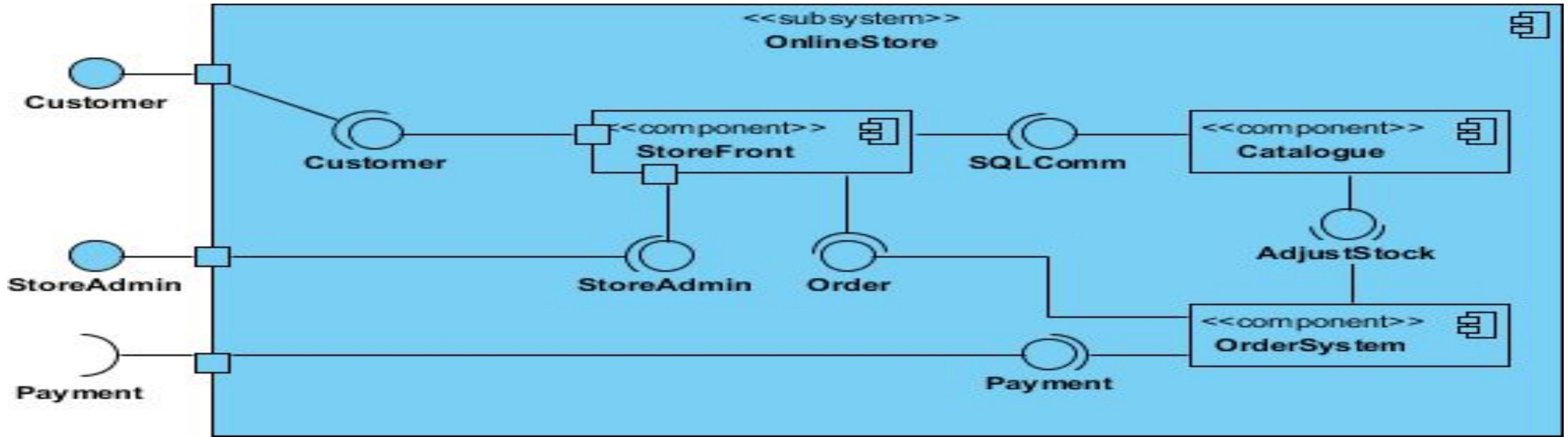

Component
Port
Provided Interface
Required Interface
<<component>>
Terminal
State
Details
Inspection Item
State
Details
<<component>>
SafetyInspection
Inspector
Inspection ID
Location
<<component>>
Staff
Account
Account
Inspection ID
Defect details
<<component>>
Defect
<<component>>
Map

20.Component Diagram Example - Using Interface (Order System)

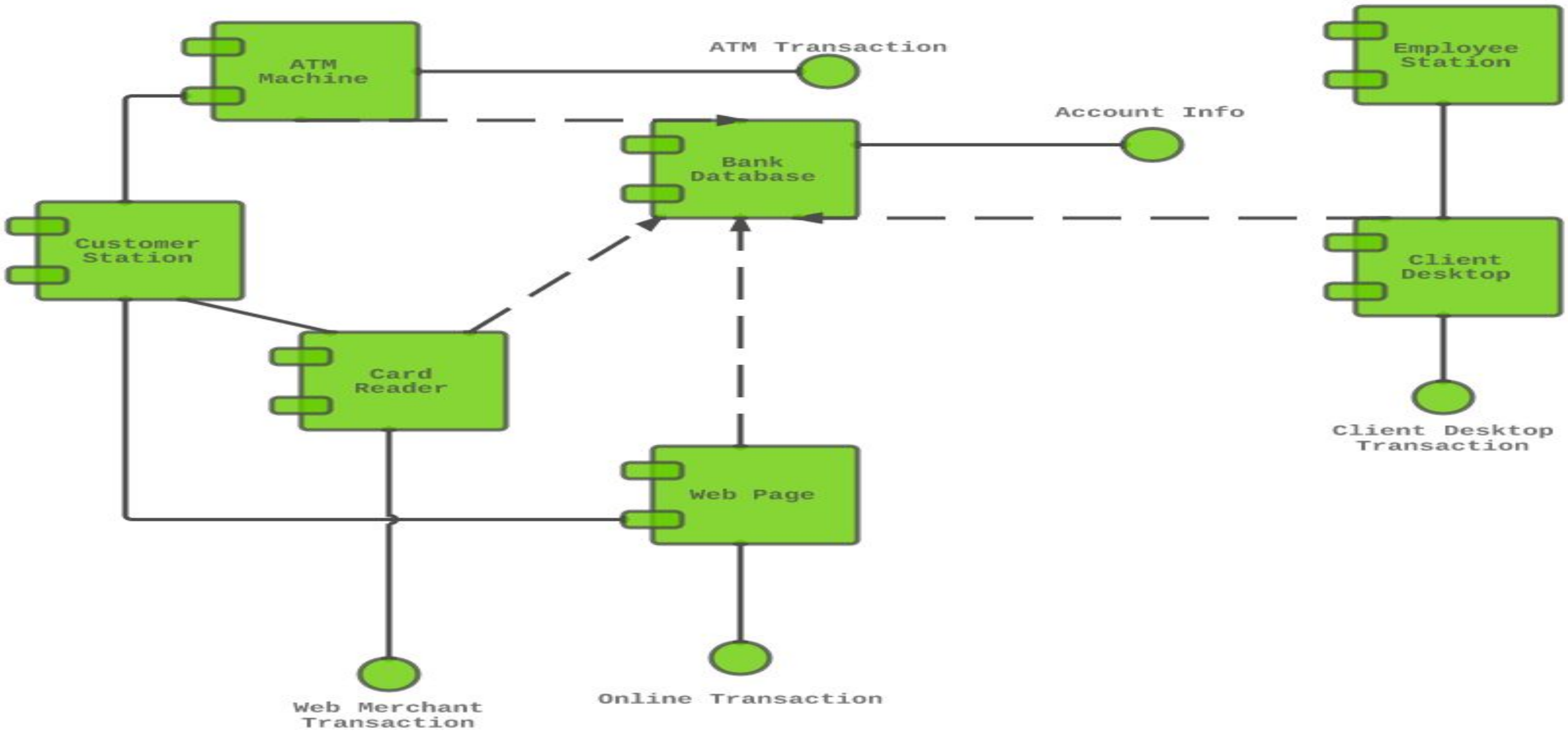


Subsystems

The subsystem classifier is a specialized version of a component classifier. Because of this, the subsystem notation element inherits all the same rules as the component notation element. The only difference is that a subsystem notation element has the keyword of subsystem instead of Component.



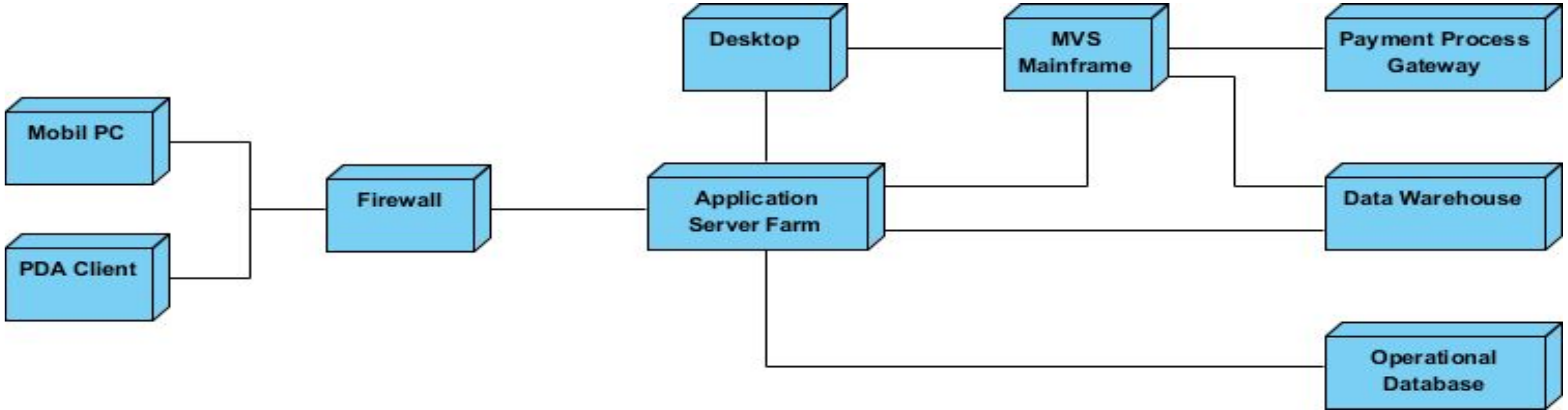
21.Component Diagram Example - ATM



22.Define Deployment Diagram with an example ? And explain its components?

- A UML Deployment diagrams is a kind of structure diagram used in modeling the physical aspects of an object-oriented system. They are often be used to model the static deployment view of a system (topology of the hardware).
- They show the **structure of the run-time system**. They capture the hardware that will be used to implement the system and the links between different items of hardware. They **model physical hardware elements and the communication paths between them**. They can be used to plan the architecture of a system. They are also useful for Document the deployment of software components or nodes.

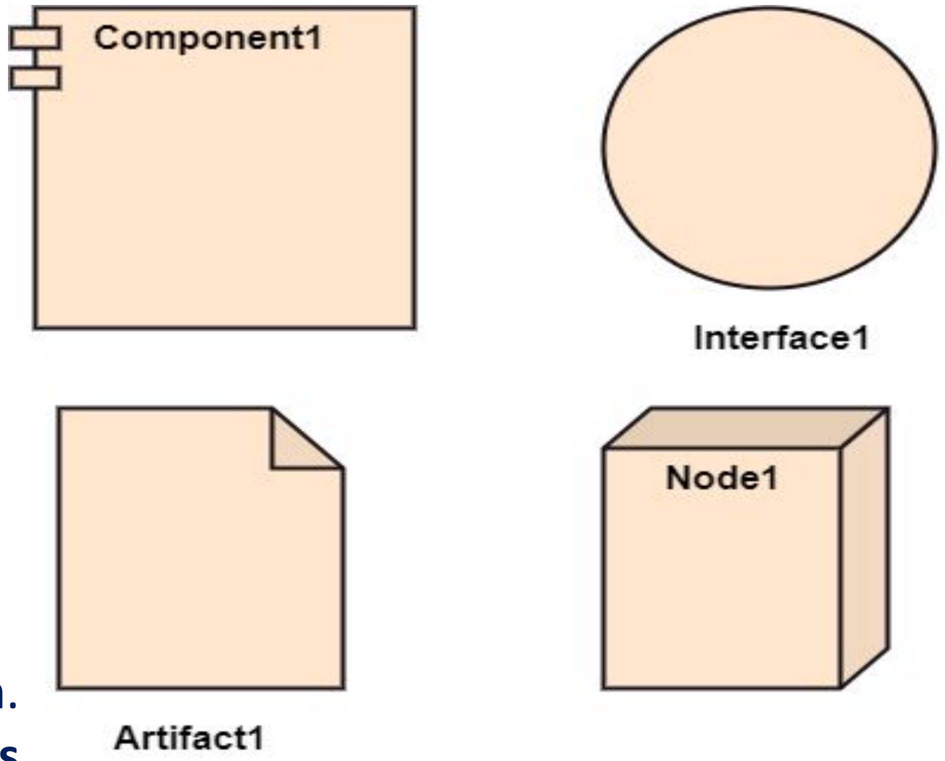
Deployment Diagram Example - Corporate Distributed System



- The deployment diagram does not focus on the logical components of the system, but it put its attention on the hardware topology.

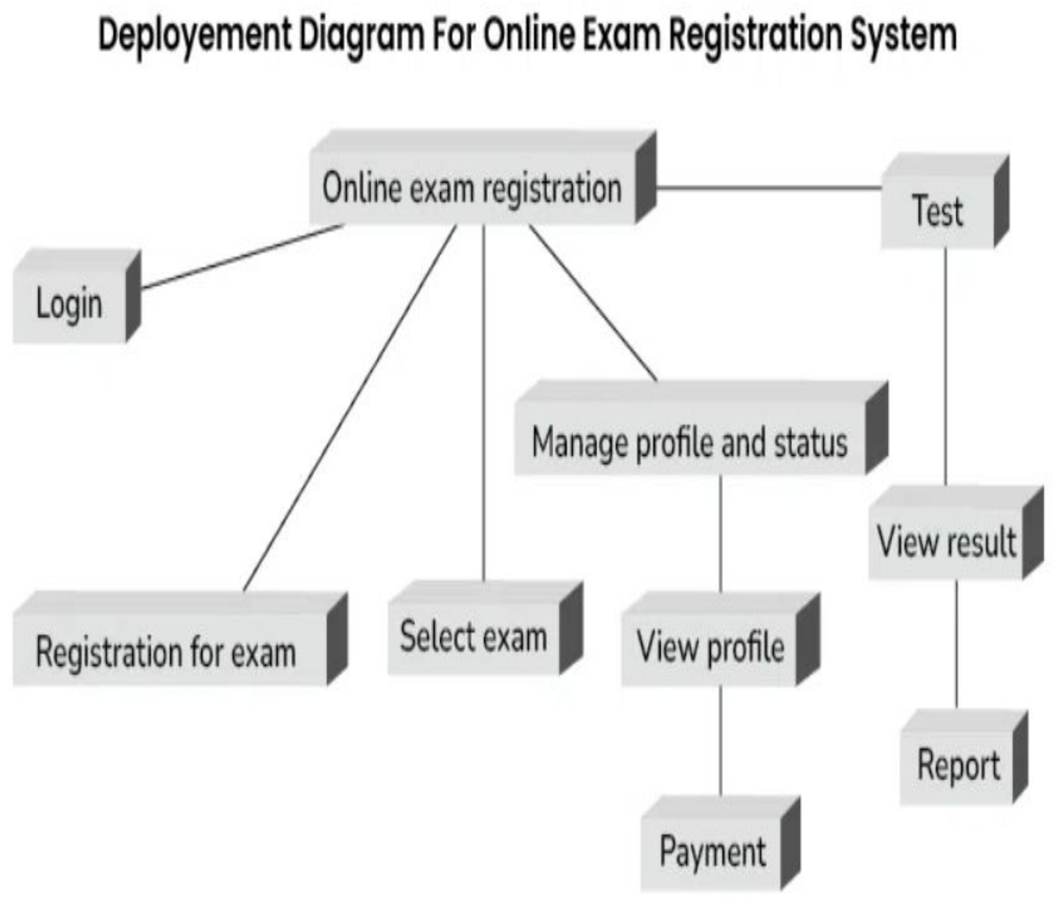
Symbol and notation of Deployment diagram

- The deployment diagram consist of the following notations:
- A component
- An artifact
- An interface
- A node



23.How to draw a Deployment Diagram?Design Deployment diagram for online exam Reservation ?

The deployment diagram portrays the deployment view of the system. It helps in **visualizing the topological view of a** system. It **incorporates nodes, which are physical hardware**. The nodes are **used to execute the artifacts**. The **instances of artifacts can be deployed on the instances of nodes.**



Below is the explanation of the above example:

Login or Registration: Users start by entering their credentials to access the exam registration system. New users must register by providing details like their name, email, and contact number.

Select Exam: After logging in or registering, users can choose the exam they want to sign up for. This involves picking from available exams, checking dates and locations, and ensuring they meet eligibility requirements.

Manage Profile and Status: Users can update their profiles, check their registration status, and print e-receipts.

View Profile: Users can access their registered information to confirm its accuracy, update contact details, print e-receipts, and save their profile.

Payment: After selecting an exam, users go to the payment page to pay the exam fee using options like credit/debit cards, net banking, or digital wallets.

Test: After registration and payment, users can access sample tests or practice papers, if available, to prepare for the exam.

View Result: Once the exam is finished, users can log in to their account to check their results.

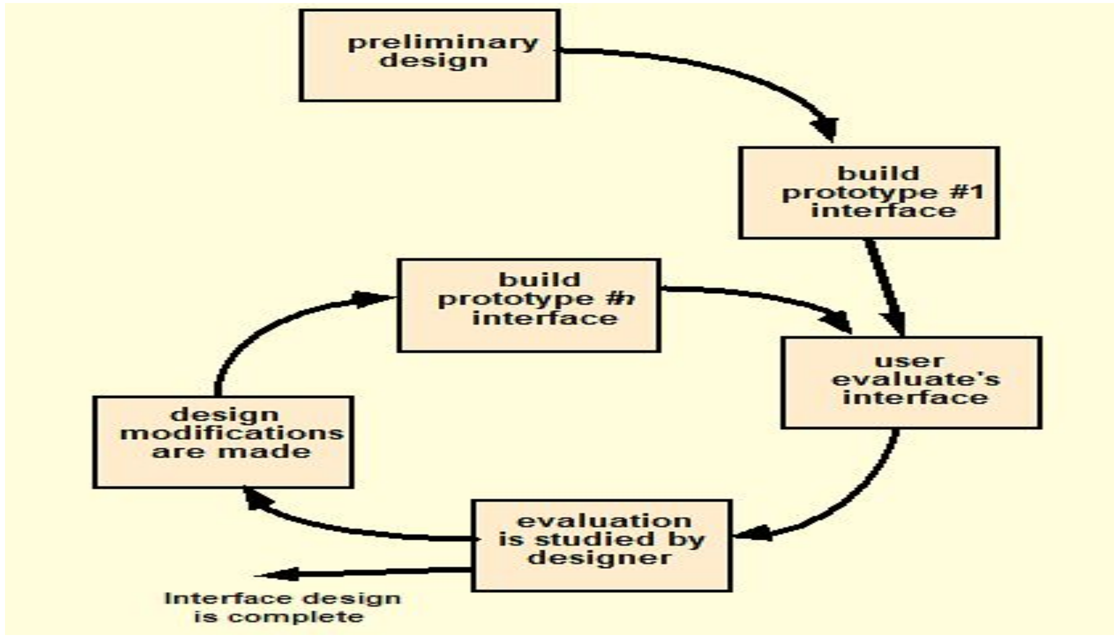
Report: Users may receive various reports, such as scorecards, response sheets, or feedback forms, depending on the exam and the policies of the examination board.

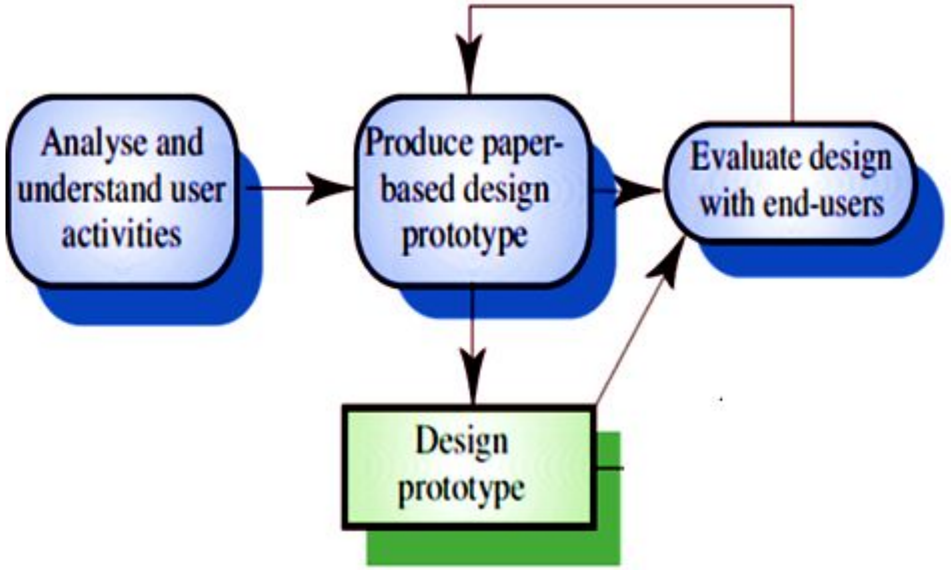
24.Write a short note on user interface analysis and design with example ?

- ❑ 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.
- ❑ Interface design focuses on three areas of concern:
  - The design of interfaces between software components,
  - The design of interfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities), and
  - The design of the interface between a human (i.e., the user) and the computer.

User-centred design process(interface design evaluation cycle)

- User-centred design is an approach to UI design where the needs of the user are paramount and where the user is involved in the design process
- UI design always involves the development of prototype interfaces.


preliminary design
build prototype #1 interface
build prototype #n interface
user evaluate's interface
design modifications are made
evaluation is studied by designer
Interface design is complete


Analyse and understand user activities
Produce paper-based design prototype
Evaluate design with end-users
Design prototype

The user interface evaluation cycle takes the form shown in figure above.

- 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 efficacy of the interface.
- In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), the designer may extract information from these data (e.g., 80 percent of all users did not like the mechanism for saving data files).
- 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.

25. State and explain Golden rules of User Interface?

1. Place the user in control
2. Reduce the user's memory load
3. Make the interface consistent

1. Place the user in control:

- **Define the interaction modes in such a way that does not force the user into unnecessary or undesired actions:** The user should be able to easily enter and exit the mode with little or no effort.

Example: login by id-password, OTP, Google Credentials.; Interaction using Mouse, or Keyboard.

- **Provide for flexible interaction:** Different people will use different interaction mechanisms, some might use keyboard commands, some might use mouse, some might use touch screen, etc., Hence all interaction mechanisms should be provided.
- **Allow user interaction to be interruptible and undoable**: When a user is doing a sequence of actions the user must be able to interrupt the sequence to do some other work without losing the work that had been done. The user should also be able to do undo operation.
- **Streamline interaction as skill level advances and allow the interaction to be customized:** Advanced or highly skilled user should be provided a chance to customize the interface as user wants which allows different interaction mechanisms so that user doesn't feel bored while using the same interaction mechanism.
- **Hide technical internals from casual users:** The user should not be aware of the internal technical details of the system. He should interact with the interface just to do his work.
- **Design for direct interaction with objects that appear on-screen:** The user should be able to use the objects and manipulate the objects that are present on the screen to perform a necessary task. By this, the user feels easy to control over the screen.

2.Reduce the User’s Memory Load :

- **Reduce demand on short-term memory:** When users are involved in some complex tasks the demand on short-term memory is significant. So the interface should be designed in such a way to reduce the remembering of previously done actions, given inputs and results.

- **Establish meaningful defaults:** Always an initial set of defaults should be provided to the average user, if a user needs to add some new features then he should be able to add the required features.
- **Define shortcuts that are intuitive:** Mnemonics should be used by the user. Mnemonics means the keyboard shortcuts to do some action on the screen.
- **The visual layout of the interface should be based on a real-world metaphor:** Anything you represent on a screen if it is a metaphor for a real-world entity then users would easily understand.
- **Disclose information in a progressive fashion:** The interface should be organized hierarchically i.e., on the main screen the information about the task, an object or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

3.Make the Interface Consistent

- **Allow the user to put the current task into a meaningful context:** Many interfaces have dozens of screens. So it is important to provide indicators consistently so that the user know about the doing work. The user should also know from which page has navigated to the current page and from the current page where it can navigate.

- **Maintain consistency across a family of applications:** In The development of some set of applications all should follow and implement the same design, rules so that consistency is maintained among applications.

- **If past interactive models have created user expectations do not make changes unless there is a compelling reason.**

User interface design is a crucial aspect of software engineering, as it is the means by which users interact with software applications. A well-designed user interface can improve the usability and user experience of an application, making it easier to use and more effective.

26. Write a short notes on user interface design steps?

- **User familiarity**: The interface should be based on user-oriented terms and concepts rather than computer concepts. For example, an office system should use concepts such as letters, documents, folders etc. rather than directories, file identifiers, etc.
- **Consistency:** The system should display an appropriate level of consistency. The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout) should be similar in all interfaces of the system
- **Recoverability:** The system should provide some resilience to user errors and allow the user to recover from errors. This might include an undo facility, confirmation of destructive actions,'soft' deletes, etc.
- **User guidance**: Some user guidance such as help systems, on-line manuals, etc. should be supplied
- **User diversity**: Interaction facilities for different types of user should be supported. For example, some users have seeing difficulties and so larger text should be available
- **Anticipation:** A WebApp should be designed so that it anticipates the use's next move.
- **Communication:** The interface should communicate the status of any activity initiated by the user.
- **Fitt's Law : "**The time to acquire a target is a function of the distance to and size of the target." A target should not be more than 3 clicks away.
- **Readability :** All information presented through the interface should be readable by young and old.
- **Track state:** When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.

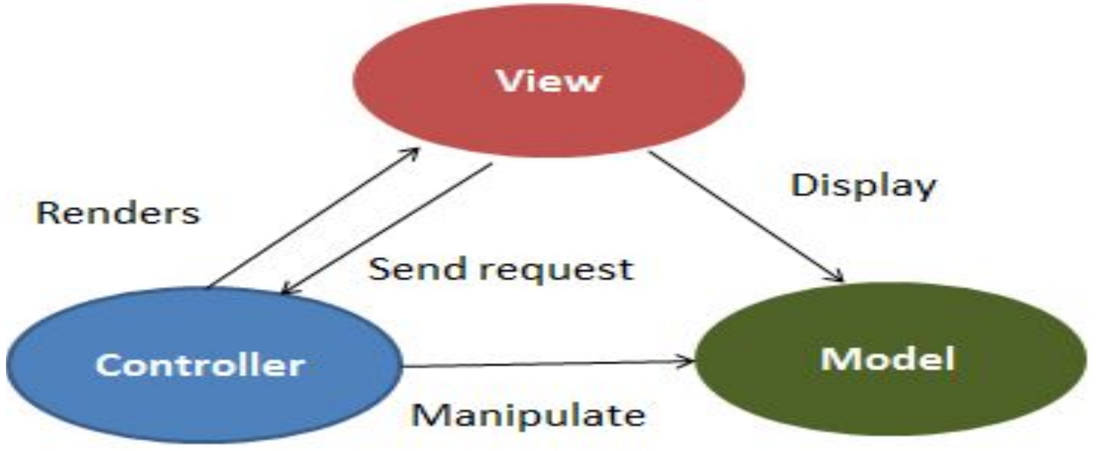
27.Define Design Patterns and its types?

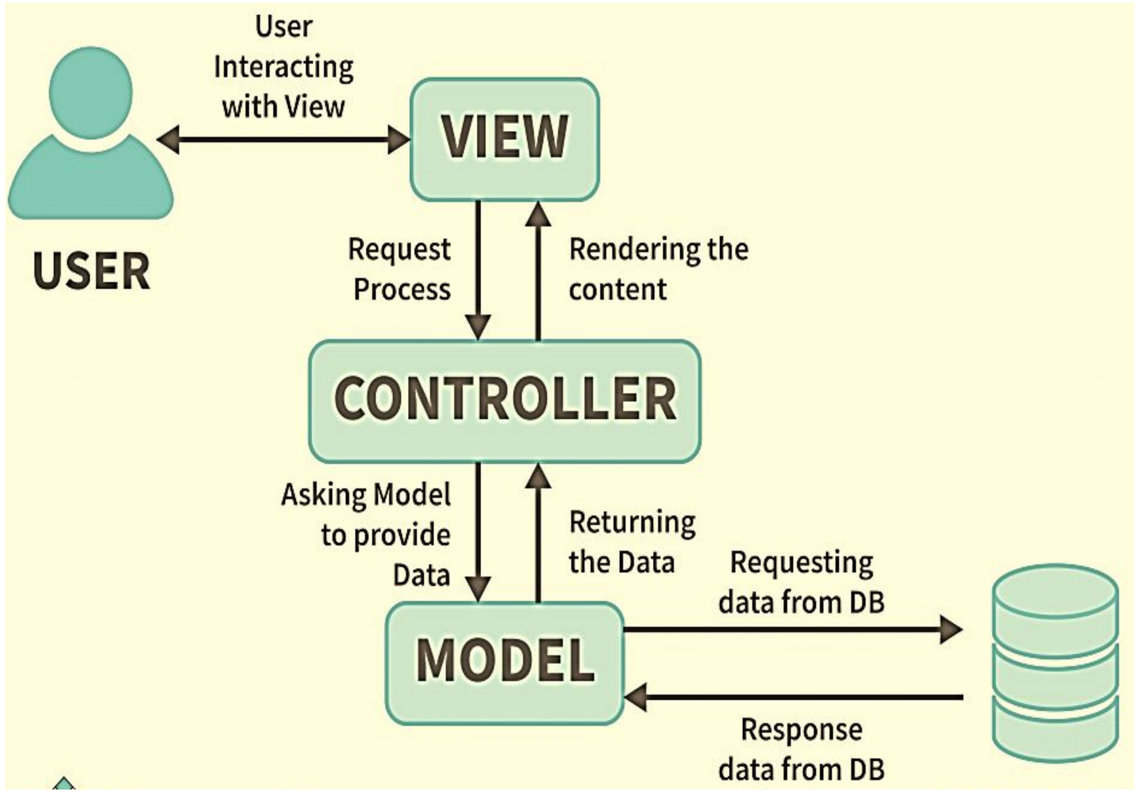
- When each of us has encountered a design problem and we might have silently thought: *Has anyone developed a solution to for this?*
  - What if there was a standard way of describing a problem (so you could look it up), and an organized method for representing the solution to the problem?
- *Design patterns* are a codified method for describing problems and their solutions. It allows the software engineering community to capture design knowledge in a way that enables it to be reused.
- A design pattern describes a problem that occurs over and over again in application contexts and then describes the solution to that problem in such a way that you can use the again and again.
- Design patterns are reusable proven solutions to design problems and pattern based design improves quality of software design.

Kinds of Patterns – a small list

- ***Creational patterns*** focus on the “creation, composition, and representation of objects, e.g.,
  - **Abstract factory pattern:** centralize decision of what factory to instantiate.
  - **Factory method pattern:** centralize creation of an object of a specific type choosing one of several implementations.
  - **Singleton patterns**: ensures creation of one and only one instance of a class.
- ***Structural patterns*** focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure, e.g.,
  - **Adapter pattern:** 'adapts' one interface for a class into one that a client expects
  - **Aggregate pattern:** a version of the Composite pattern with methods for aggregation of children / child classes.
- ***Behavioral patterns*** address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects, e.g.,
  - **Chain of responsibility pattern:** Command objects are handled or passed on to other objects by logic-containing processing objects
  - **Command pattern:** Command objects encapsulate an action and its parameters

28.Write a short notes on MVC architecture?





❖ Commonly used for User Interface design by separating application concerns into three parts.

❑ **Model:** Holds the data

- The model contains all application specific content or data and processing logic.

❑ **View**: Where user interacts with system

- The view contains all interface specific functions and enables the presentation of content to the user.

❑ **Controller:** Process the user request and send the response to view.

- The controller manages access to the model and the view and coordinates the flow of data between them.

1. First, the user will make the request through the view.
2. The request from the view goes to the controller, and the controller checks the route configuration.
3. After checking the configuration, it passes to the model, and the model checks the request.
4. In the model, there are two steps, page Initialization will get started and it will generate results.
5. Once the result is generated, it will unload to view through the view engine.

29. What are the major attributes of quality for WebApps?

Security : Protect from external attacks.

- Exclude unauthorized access.
- Ensure the privacy of users/customers.

- **Availability**
  - the measure of the percentage of time that a WebApp is available for use.
- **Scalability**
  - Can the WebApp and the systems with which it is interfaced handle significant variation in user or transaction volume.
- **Accuracy and Consistency**
  - Is the data / content presented accurate enough?
  - Are the webpages consistent in look and feel, presentation?
- **Response Time and Latency**
  - Does the Web site server respond to a browser request within certain parameters?
  - In an E-commerce context, how is the end to end response time after a SUBMIT?
  - Are there parts of a site that are so slow the user declines to continue working on it?
- **Performance**
  - Is the connection quick enough to load webpages?
  - How does the performance vary by time of day, by load and usage?
  - Is performance adequate for E-commerce applications?

30. Explain WebApp Design Goals?

- ❑ Identity
  - The aesthetic, interface, and navigational design of a WebApp must be consistent with the application domain for which it is to be built. A website for a hip hop group will undoubtedly have a different look and feel than a WebApp designed for a financial services company.
- ❑ Robustness
  - The user expects robust content and functions that are relevant to the user's needs. If these elements are missing or insufficient, it is likely that the WebApp will fail.
- ❑ Navigability

  Design web browsing in a manner that is intuitive and predictable. Information should be kept at three clicks away – a Golden Rule. The user should understand how to move about the WebApp without having to search for navigation links or instructions.
- ❑ Visual appeal
  - The look and feel of content, interface layout, color coordination, the balance (how much quantity) of text, graphics and other media, navigation mechanisms must appeal to end-users.
- ❑ Compatibility
  - WebApp should work with all major browser environments and configurations

31.Write about WebApp Design Qualities?

Design leads to high-quality product.

• General quality attributes which can be applied to WebApp are

1)Usability :-

- ✔ Global site understandability
- ✔ Online feedback and help features
- ✔ Interface and aesthetic features
- ✔ Special features

2) Functionality :-

- ✔ Navigation and browsing features
- ✔ Searching and retrieving capability
- ✔ Application domain-related features

3) Reliability :-

- ✔ Correct link processing
- ✔ Error Recovery
- ✔ User input validation and recovery

4) Efficiency :-

- ✔ Response time performance
- ✔ Page generation speed
- ✔ Graphics generation speed

5) Maintainability :-

- ✔ Ease of correction
- ✔ Adaptability
- ✔ Extendibility

32.write about WebApp Design Pyramid?

A Web application has

- Information Content to be presented
- Functionality / Operations to be performed

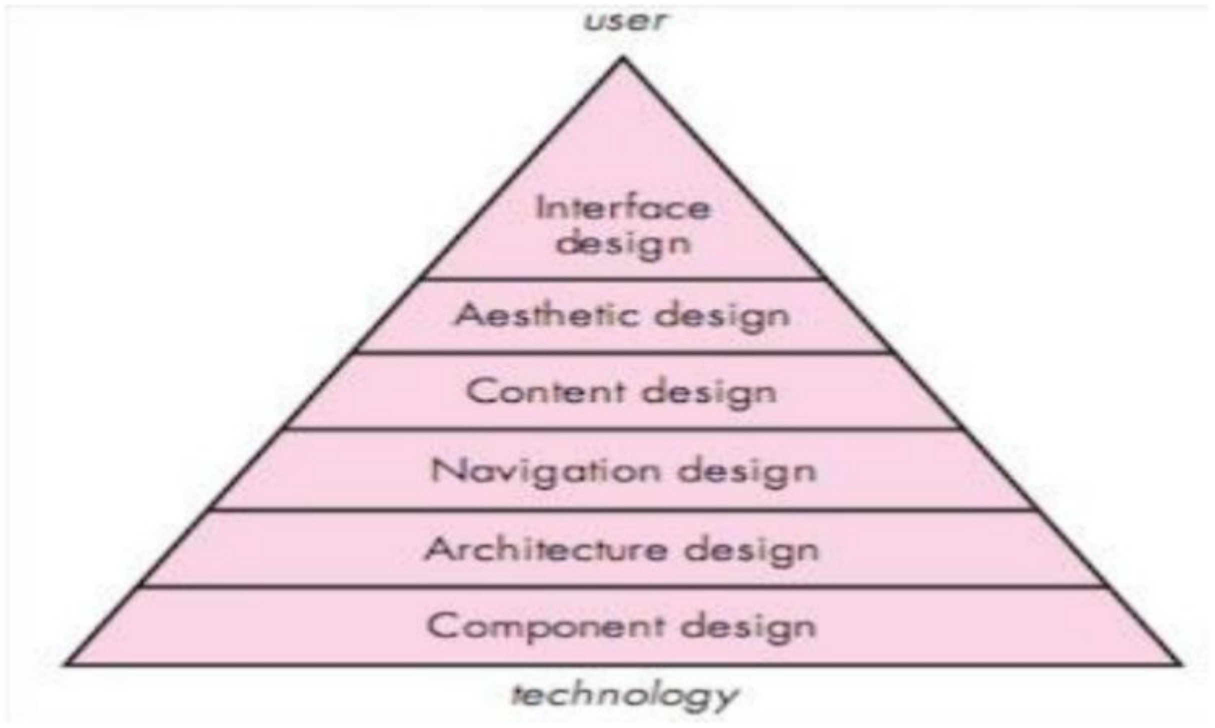
Design must take care of both the aspects.

- The creation of an effective design will typically require a diverse set of skills.
- Sometimes, for small projects, a single developer may need to be multi-skilled.

For larger projects, it may be advisable and/or feasible to draw on the expertise of specialists:

o Web engineers,o Graphic designers,
o Content developers,o Programmers,
o Database specialists,
o Information architects,
o Network engineers,
o Security experts,
o Testers.

- Information Contents Designs
  - Content Design
  - Navigation Design
  - Aesthetic Design
- Functionality Design
  - Architecture Design
  - Component Design
  - Interface Design



1. Interface Design:

- The interface should provide an indication of the WebApp that has been accessed.
- Inform the user of location in the content hierarchy.

2.Aesthetic design

- Aesthetic design, also called graphic design.

Layout Issues :

- Don’t be afraid of white space.
- Emphasize content.
- Organize layout elements from top-left to bottom right.
- Group navigation, content, and function geographically within the page.
- Don’t extend your real estate with the scrolling bar.
- Consider resolution and browser window size when designing layout.

3.Content Design:

- Develops a design representation for content objects
- For WebApps, a content object is more closely aligned with a data object for conventional software.
- Represents the mechanisms required to instantiate their relationships to one another.
- Analogous to the relationship between analysis classes and design components.
- A content object has attributes that include content-specific information
- Implementation-specific attributes that are specified as part of design.

4.Architecture Design:

• Content architecture focuses on the manner in which content objects (or composite objects such as Web pages) are structured for presentation and navigation.

o The term information architecture is also used to predict structures that lead to better organization, labeling, navigation, and searching of content objects.

5. Navigation Design:

• Begins with a consideration of the user hierarchy and related use cases.

• Each actor may use the WebApp somewhat differently and therefore have different navigation requirements.

• As each user interacts with the WebApp, it encounters a series of navigation semantic units (NSUs).

• NSU—"a set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements".

6. Component-Level Design

• Perform localized processing to generate content and navigation capability in a dynamic Fashion.

• Provide computation or data processing capability that are appropriate for the WebApp's business domain.

• Provide sophisticated database query and access.

• Data interfaces with external corporate systems.

33. Write a short notes on WebApp interface design And Aesthetic design?

- When designing a web application interface, aesthetics are important for creating a good first impression and making users enjoy the experience:

Aesthetics

- An attractive interface can make users trust the design, ignore minor usability issues, and spend more time on the app. Aesthetics include colors, fonts, icons, buttons, and menus.

User experience

- The user experience (UX) is how easy or complex it is for a user to interact with the app. A good UX design is simple and easy, and makes users feel like they've gotten what they want.

Efficiency

- An efficient interface makes users more productive by using shortcuts and good design. This can include proper flow between screens, effective transitions, and better shortcuts.

Responsiveness

- A web app should be responsive on different devices, adapting to the screen size and providing a good user experience.

Consistency

- A consistent design helps users identify patterns and use the app smoothly