Software Engineering

MODULE 3 : System Design Engineering

3.1 Design quality, Classification of Design Activities, Design Concepts: Modularity and Layering, Introduction to Pattern-Based Software Design

3.2 Software Architecture, Data Design, Object-Oriented versus Function-Oriented Design, Design of Software Objects, Methods, Cohesion and Coupling between Objects

3.3 User Interface Design: Rules, User Interface Analysis and Steps in Interface Design, Design Evaluation

3.4 Software Reuse, Component-Based Software Engineering

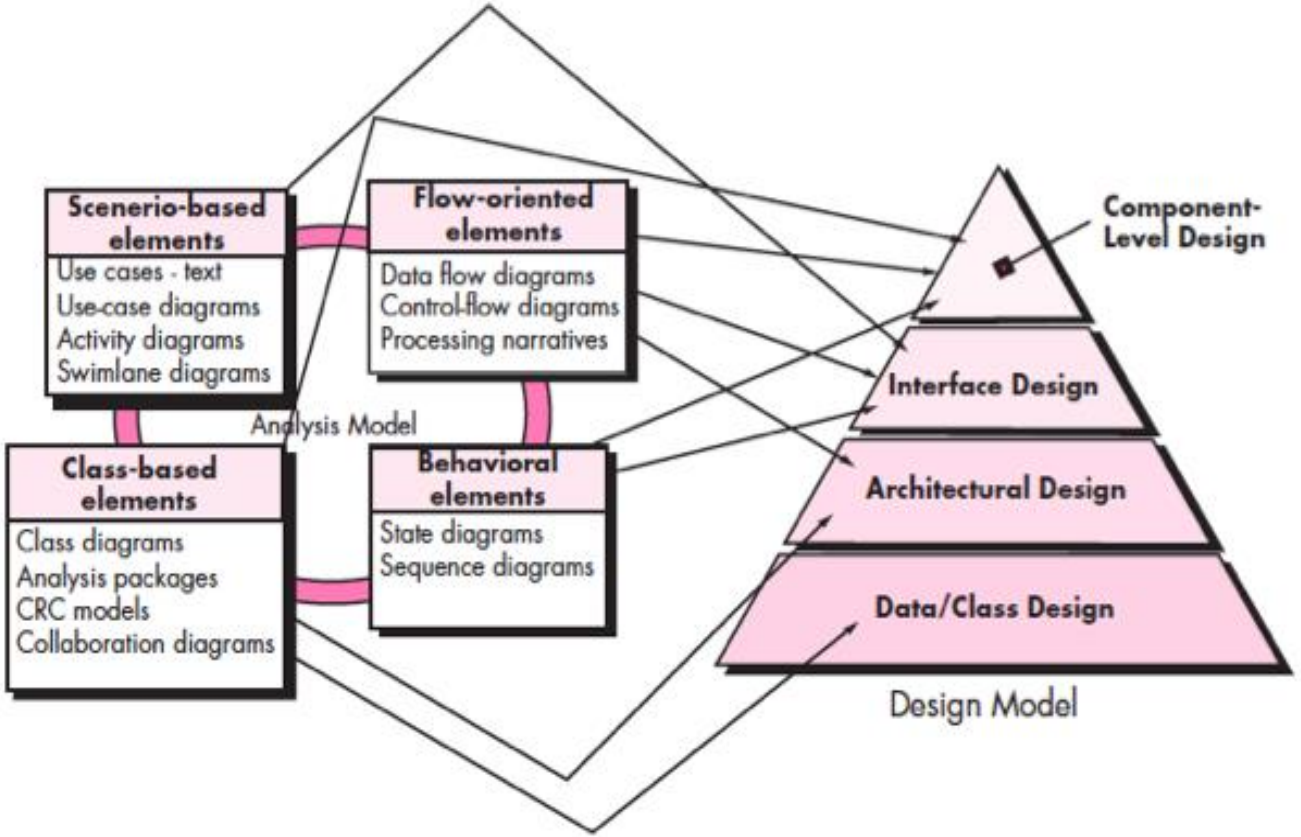
3.1 Software Design Quality

Requirements model focuses on describing required data, function, and behaviour

Design Model :

- Represents model of the software which provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system
- Developed by Software engineers to conduct each of the design tasks.
- Design allows you to model the system or product that is to be built.
- Can be assessed for quality and improved before code is generated, tests are conducted, and end users become involved in large numbers.

Translating the requirement model in to Design Model



Design Quality

Design Steps:

- The architecture of the system or product must be represented.
- The interfaces that connect the software to end users, to other systems and devices, and to its own constituent components are modelled.
- The software components that are used to construct the system are designed.
- Each of these views represents a different design action, but all must conform to a set of basic design concepts that guide software design work.

Outcome of Design Model

Architectural, interface, component level, and deployment representations

Design model is assessed by the software team to determine :

- Whether it contains errors, inconsistencies, or omissions
- Whether better alternatives exist
- Whether the model can be implemented within the constraints, schedule, and cost that have been established.

Design

- **Software design** is an **iterative process** through which **requirements are translated into a "blueprint"** for constructing the software.
- Developers identify and prioritize the qualities of the system that they should optimize.
- **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

- **Explicit Requirements:** The Things You Wrote Down
- **Implicit Requirements:** The Things Your Customers Will Expect
- **Latent Requirements:** Things That Will Delight Your Customers

- **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 behavioural domains from an implementation perspective

Quality Guidelines

1. A design should exhibit an architecture that:
   a. Is created using standard architectural patterns.
   b. Is composed of components that exhibit good design characteristics.
   c. Can be implemented in an evolutionary fashion
2. A design should be modular: logically partitioned into elements or subsystems
3. A design should contain **distinct representations** of data, architecture, interfaces, and components.
4. A design should lead to **data structures that are appropriate** for the classes to be implemented and are drawn from recognizable data patterns.

Quality Guidelines

5. A design should lead to components that **exhibit independent functional characteristics.**
6. A design should lead to interfaces that **reduce the complexity** of connections between components and with the external environment.
7. A design should be derived using a **repeatable method** that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that **effectively communicates its meaning**.

Implementation

The implementation phase of software development is the **process of converting a system specification into an executable system** through the **design of system**.

Design Processes

The **design proces**s is a sequence of steps that enable the designer to **describe all aspects of the software to be built.**

1. **The design process should not suffer from "tunnel vision."-** A good designer should consider alternative approaches.
2. **The design should be traceable to the analysis model.-** means for tracking how requirements have been satisfied by the design model.
3. **The design should not reinvent the wheel. -** Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
4. **The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world. -** structure of the software design should be replica of structure of the problem domain.

Design Processes

5. The design should exhibit uniformity and integration.

- format should be defined for a design team before design work begins.
- design is integrated if care is taken in defining interfaces between design components.

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.

- Well designed software should never “bomb.”
- It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

Design Processes

8. Design is not coding, coding is not design.

-the level of abstraction of the design model is higher than source code.

-only design decisions made at the coding level address the small implementation details.

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.

Design Principles

Software Design concepts are **fundamental concepts** which **provides** the software designer with a **foundation from which more sophisticated design methods** can be applied for software design process.

Design Activities

1. Architectural Design

- The architectural design for software is the equivalent to the floor plan(overview) of a house.
- Architectural design elements give us an overall view of the software.

Architectural model is derived from three sources:

1. **Information about the application** domain for the software to be built
2. **specific requirements**
3. The availability of architectural **Styles and patterns**

Design Activities

2. Interface Design

- Analogous to a set of detailed drawings.
- Specifications of software
- The interface design elements for software depict **information flows into and out of the system** and how it is communicated among the components defined as part of the architecture.

Important elements of interface design:

1. **User Interface (UI)**
2. **External interface**
3. **Internal interface**

Design Activities

3. Component Level Design

- Analogous to a set of detail drawing and specifications of each room in a house.
- The component-level design for software fully **describes the internal detail of each software component**.
- Component-level design **defines data structures** for all local data objects and **algorithmic detail**.
- Within the context of object-oriented software engineering, a **component is represented in UML diagrammatic form.**

Design Activities

4. Data structure Design

- Data structures used in the system implementation are designed in details and specified

5. Algorithm Design

- Algorithms used to provide the service are designed and specified

6. Deployment level Design

Indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software

Design Concepts

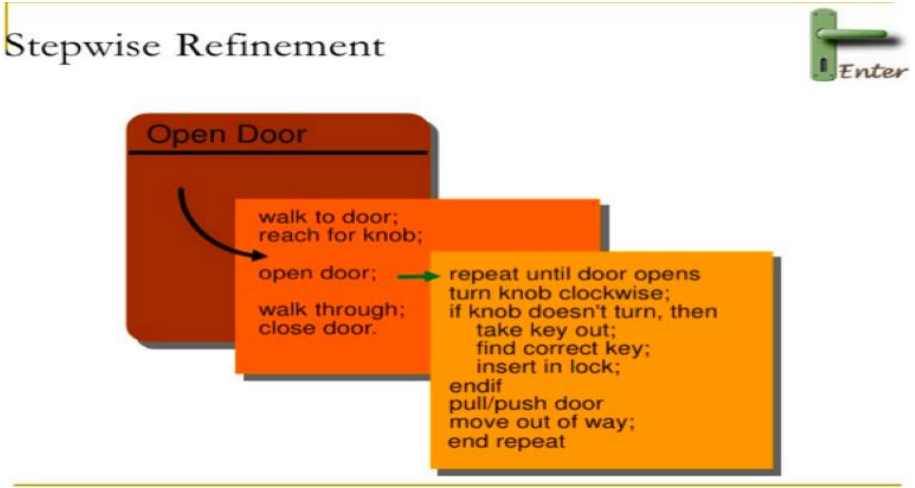
1. Abstraction

- Modular solution to any problem , many levels of abstraction can be posed.
- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- Lower levels of abstraction, a more procedural orientation is taken.
- Each step in the software process is a refinement in the level of abstraction of the software solution.
- A **procedural abstraction** is a named sequence of instructions that has a specific and limited function. (algorithm)
  - Example: word **DOOR** implies a long sequence of procedural steps
- A **data abstraction** is a named collection of data that describes a data object
  - Example: word **DOOR** would encompass a set of attributes that describe the door
  - **control abstraction** implies a program control mechanism for coordinating activities.

Design Concepts

2. Refinement

- Refinement is process of elaboration
- It's a top-down design strategy.
- A program is developed by successively refining levels of procedural detail.
- 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.



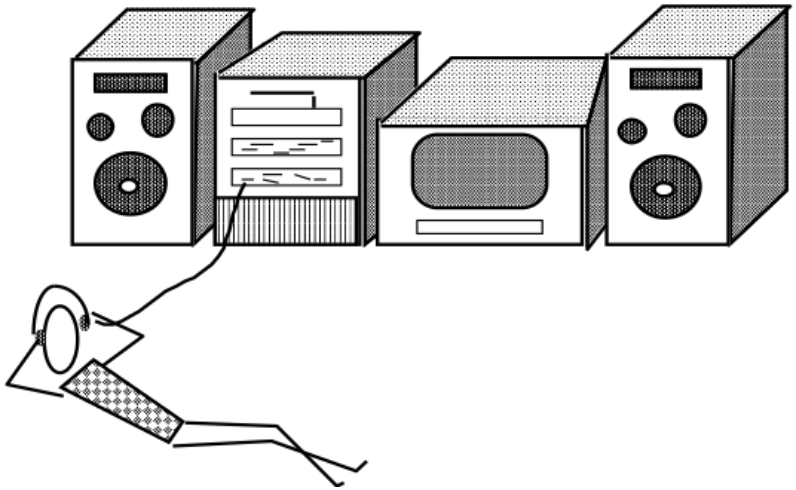
Design Concepts

3. Modularity

- **Software is divided** into separately named and addressable **components**, often called modules, that are **integrated to satisfy problem requirements.**

- **Modularity** is the single attribute of software that **allows** a program to be **intellectually manageable.**

easier to build, easier to change, easier to fix ...



Design Concepts

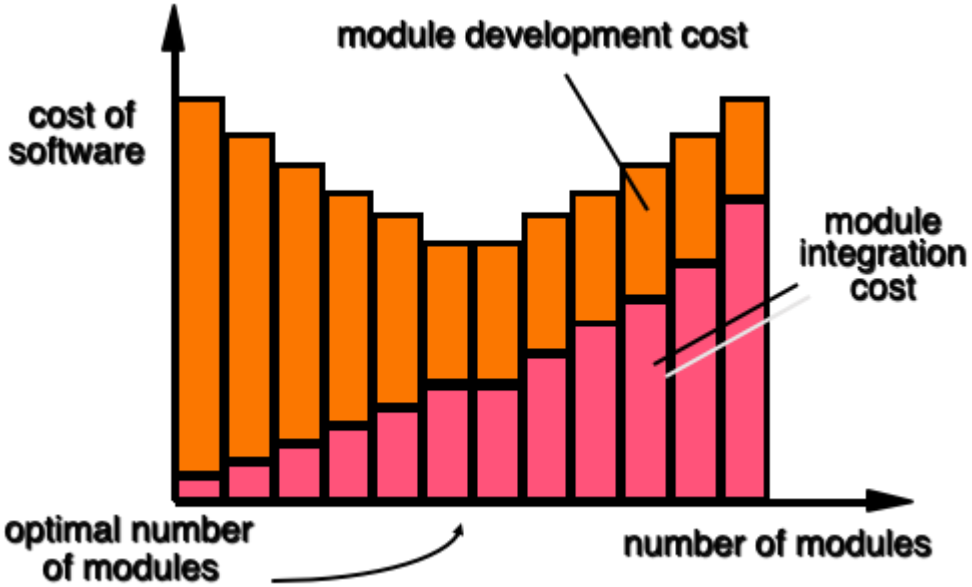
Modularity

"divide and conquer" an argument for modularity

- it's easier to solve a complex problem when you break it into manageable pieces

Design Concepts

Right number of modules for a specific software design?



Design Concepts

3. Modularity

Five criteria to define an effective modular system:

1. **Modular decomposability** – systematic mechanism for **decomposing** the problem into subproblems.
2. **Modular composability – reusing** existing system into new system yield modular solution and not reinvent wheel.
3. **Modular understandability** - understood as **standalone** unit , it will be easier to build and easier to change.
4. **Modular continuity** - **small changes** to the system requirements - changes to **individual modules** rather system – change induced side effects minimized.
5. **Modular protection** - **irregular** condition - error-induced side effects minimized.

Design Concepts

4. Software Architecture

- Architecture- the overall structure of the software
- Hierarchical structure of the software components (modules), the manner in which these components interact and the structure of data that are used by the components
- Well designed architecture serves as framework for detailed design activities which can also help in reuse of design level concepts

Design Concepts

4. Software Architecture

- **Properties of architectural design:**

1. **Structural properties** – defines the components of a system and their interaction with one another
2. **Extra-functional properties–** like performance, capacity, reliability, security, adaptability, and other system characteristics
3. **Families of related systems-** should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems..

Design Concepts

Architectural Models:

1. **Structural Models –** represent architecture as an organized collection of program components
2. **Framework Models–** identify repeatable architectural design frameworks (patterns)
3. **Dynamic Models-** address the behavioural aspects of the program architecture, **indicating changes** in configuration system
4. **Process Models-** focus on the design of the **business or technical process** that the system must accommodate
5. **Functional Models-** represent the functional hierarchy of a system

Design Concepts

5. Patterns

- A design pattern describes a design structure that solves a particular design problem within a specific context.
- The intent of each design pattern is to provide a description that enables a designer to determine:
  1. whether the pattern is applicable to the **current work**
  2. whether the pattern can be **reused**
  3. whether the pattern can serve as a guide for **developing a similar,** but f**unctionally** or **structurally different pattern**

Design Concepts

6. Information Hiding

- **Hide the details of data structures and procedural processing** behind a module interface.
- Knowledge of the **details need not be known by users**
- independent modules that **communicate with one another only that information necessary** to achieve software function
- Information hiding provides the **greatest benefits when modifications** are required.
- Because most data and procedural **detail are hidden** from other parts of the **software errors less likely to propagate.**

Design Concepts

7. Functional Independence:

- Software should be designed such that **each module addresses a specific subset of requirements** and has a **simple interface with other program structure.**
- Software with effective modularity, that is, independent modules, is easier to develop.

Independent modules

- ➢ easier to maintain (and test)
- ➢ effects caused by design or code modification limited
- ➢ error propagation is reduced
- ➢ reusable modules are possible.
- ➢ functional independence is a **key to good design**, and **design** is the **key to software quality**.

Design Concepts

Independence is assessed by qualitative criteria

COHESION - the degree to which a module performs one and only one function.

COUPLING - the degree to which a module is "connected" to other modules in the system.

Design Concepts

Cohesion:

- A cohesive module performs a **single task**, requiring **little interaction** with **other components** in other parts of a program.
- **High degree of cohesion is advisable but** it is also necessary and advisable for **components to perform multiple functions.**

Coupling:

- Coupling is an indication of **interconnection among modules** in a software structure.
- Coupling depends on the **interface complexity** between modules – entry point and type of data that passes through interface.
- In software design, one should strive for the lowest possible coupling

Design Concepts

Coupling:

- Software should be designed such that **each module addresses a specific subset of requirements** and has a **simple interface with other program structure.**
- Software with effective modularity, that is, independent modules, is easier to develop.

Independent modules

- ➢ easier to maintain (and test)
- ➢ effects caused by design or code modification limited
- ➢ error propagation is reduced
- ➢ reusable modules are possible.
- ➢ functional independence is a **key to good design**, and **design** is the **key to software quality**.

Design Concepts

8. Refactoring:

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

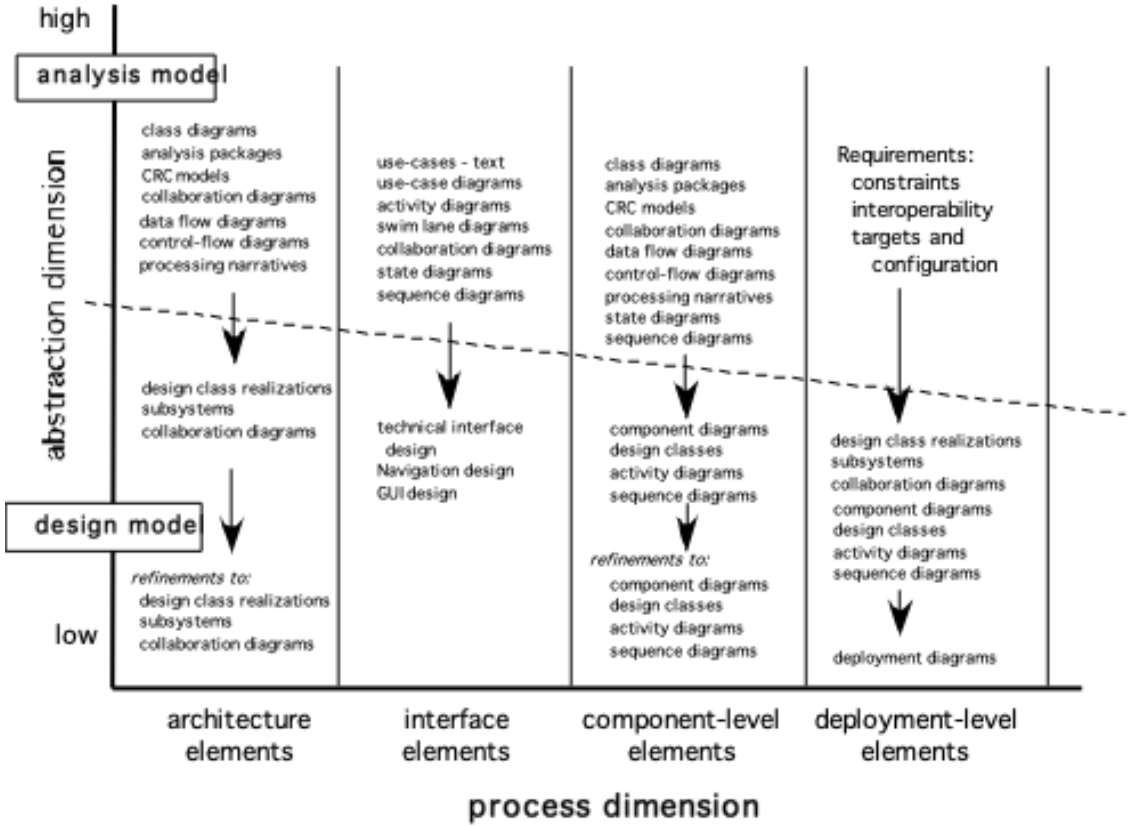
- When software is refactored:
  - the existing design is examined for redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures,
  - any other design failure that can be corrected to yield a better design.

Design Model

- Each element of analysis model is transformed into design model and refined iteratively.

- The **process dimension** as design tasks are executed as part of software process.
- The **abstraction dimension** represents the level of detail in each element

Design Model

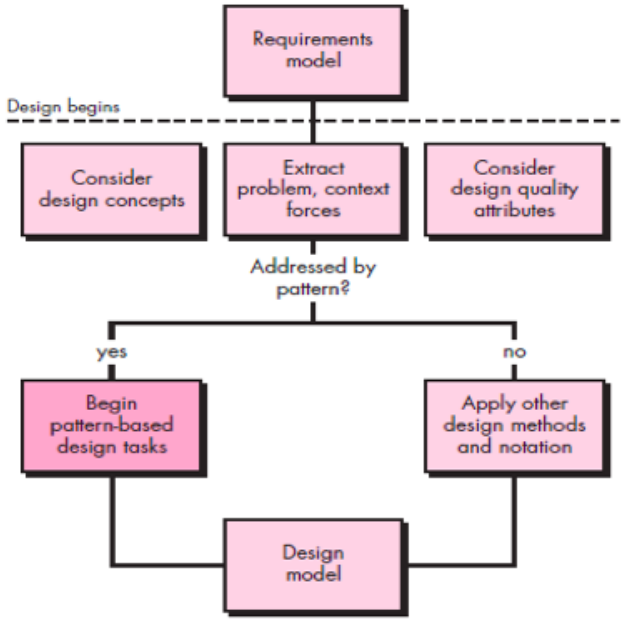


Pattern Based Software Design

- The best designers in any field have an uncanny **ability to see patterns that characterize a problem** and corresponding patterns that can be combined to create a solution.
- A description of a design pattern may also consider a set of design forces.
- ***Design forces*** **describe non-functional requirements** (e.g., ease of maintainability, portability) **associated with the software** for which the pattern is to be applied.
- The pattern characteristics (classes, responsibilities, and collaborations) indicate the attributes of the design that may be adjusted to enable the pattern to accommodate a **variety of problems (problem solving)**.
- Throughout the design process, one should look for every opportunity to **apply existing design patterns** rather than creating new ones.

Pattern Based Software Design

1. Pattern-Based Design in Context



Use methods and modelling tools available for architectural, component level, and interface design.

Pattern Based Software Design

2. Thinking in Patterns

- "**new way of thinking**" when one **uses patterns** as part of the design activity.
- Good design begins by considering context—the big picture.

Approach that enables a designer to think in patterns:

1. Context of **software build and requirements** to be understood.
2. extract the **patterns** that are **present** at **level of abstraction.**
3. Begin design with "**big picture**" patterns that establish a **context or skeleton** for further design work.
4. "**Work inward from the context**" lower levels of abstraction that contribute to the design solution.
5. **Repeat steps 1 to 4** until the complete design is fleshed out.
6. **Refine the design** by adapting each pattern to the specifics of the software

Design Model

3. Design Tasks

The following design tasks are applied when a pattern-based design philosophy is used:

1. Examine the requirements model and develop a problem hierarchy.
2. Determine if a **reliable pattern language** has been **developed** for the problem domain.
3. Beginning with a broad problem, determine **whether one or more architectural patterns** is **available** for it.
4. Using the collaborations provided for the architectural pattern , **examine subsystem or component-level problems and search for appropriate patterns** to address them.

Repeat steps until all broad problems have been addressed

Design Model

3. Design Tasks

The following design tasks are applied when a pattern-based design philosophy is used:

5. user interface design problems have been -**search** the many user interface **design pattern** repositories for appropriate patterns.

6. Compare various pattern against the existing ones for better designing.

7. Be certain to refine the design as it is derived from patterns using **design quality criteria** as a guide.

Design Model

4. Building a Pattern-Organizing Table

•As pattern based design proceeds organizing and categorizing problem may occur.

•Solution: A pattern-organizing table can be implemented as a spreadsheet model

	Database	Application	Implementation	Infrastructure
Data/Content				
Problem statement ...	PatternName(s)		PatternName(s)	
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...	PatternName(s)			PatternName(s)
Architecture				
Problem statement ...		PatternName(s)		
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...				
Component-level				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...				PatternName(s)
Problem statement ...		PatternName(s)	PatternName(s)	
User interface				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	

Module 3

3.1 Design Quality, Classification of Design activities. Design concepts: Modularity & Layering. Introduction to Pattern-based software design.

3.2 Software Architecture, Data design, Object -Oriented versus Function - oriented Design, Design of software object, methods, Cohesion & Coupling between objects.

3.3 User Interface design: Rules, User Interface Analysis Steps in Interface design, Design Evaluation

3.4 Software Reuse, Component-based software design

Software Architecture

•Why Architecture?

The architecture **is not the operational software**.

A representation that enables a software engineer to:

- **analyze the effectiveness of the design** in meeting its stated requirements; **Representations of software architecture are an enabler for communication** between all parties (stakeholders) interested in the development of a computer-based system.
- **consider architectural alternatives** at a stage when making design changes is still relatively easy, **•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.

Software Architecture

- **Why Architecture?**

The architecture **is not the operational software**.

A representation that enables a software engineer to:

- **reduce the risks** associated with the construction of the software.

  Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together".

Architectural Styles

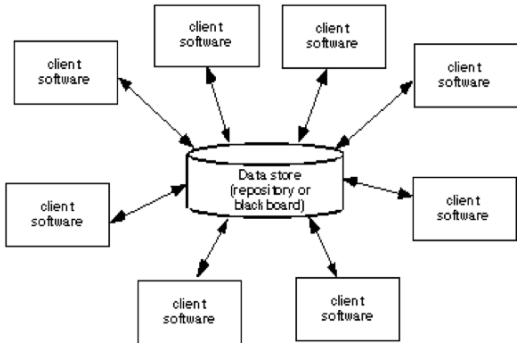
- **Each style describes a system category that encompasses:**

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

Architectural Styles

1 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.
- **Client software** accesses a **central repository.**
- Data-centered architectures promote **integration-ability.**
- New client components can be added easily.
- Client components independently execute processes.



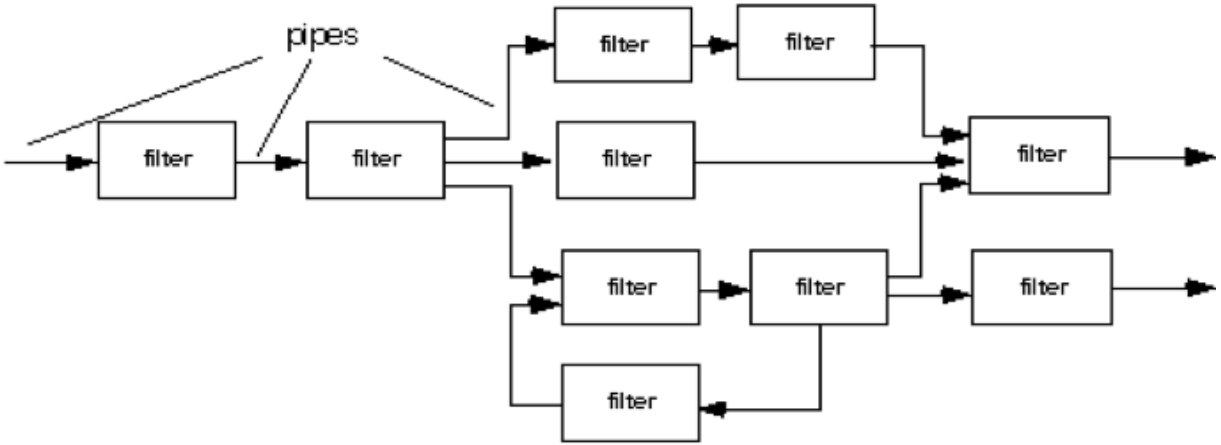
Architectural Styles

2. 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.**
- Pipe and filter pattern.
- Each filter works independently.
- If the data **flow degenerates into a single line** of transforms, it is termed **batch sequential.**

Architectural Styles

2 Data flow architecture:



(a) pipes and filters

(b) batch sequential

Architectural Styles

3. Call and return architecture:

This architectural style enables you to achieve a **program structure that is relatively easy to modify and scale.**

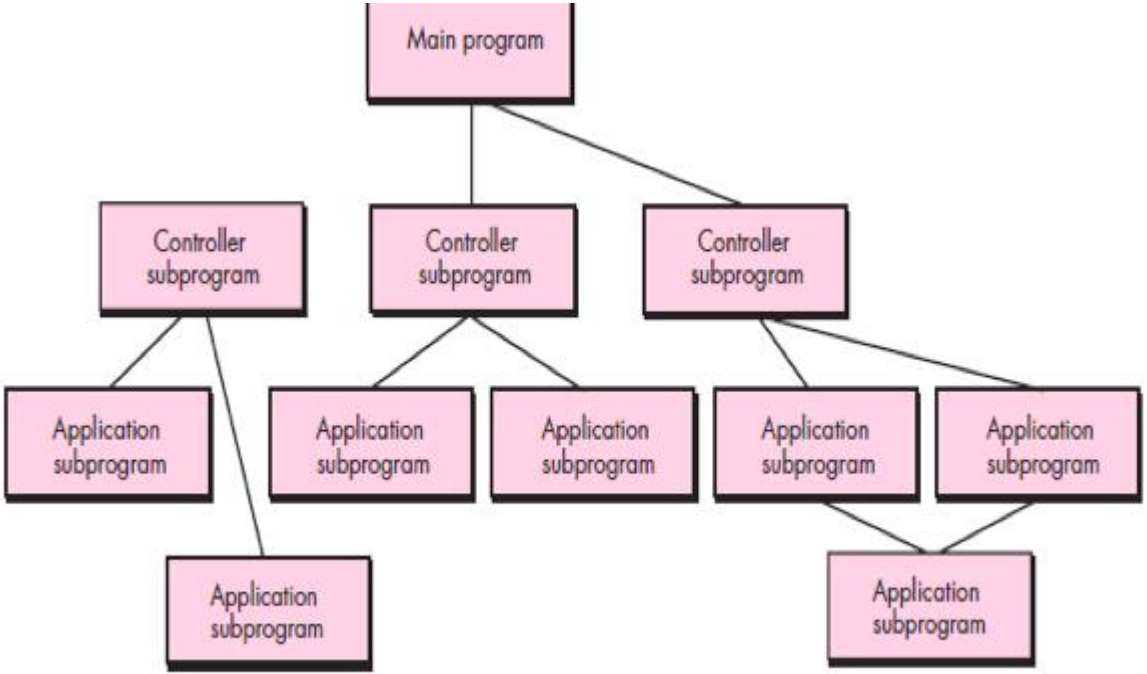
Sub styles of call and return architecture:

- **Main program/subprogram architectures: main program invokes a number of program components** that in turn may invoke still other components.

- **Remote procedure call architectures:** The **components of a main program/subprogram** architecture are **distributed across multiple computers on a network.**

Architectural Styles

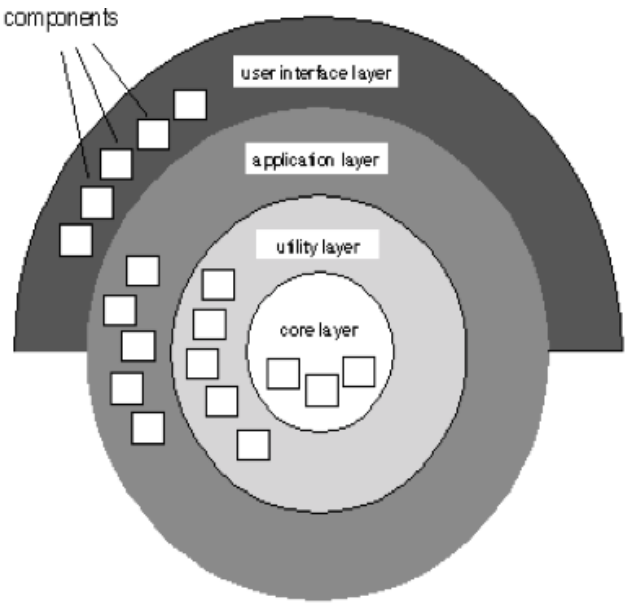
3. Call and return architecture:



Architectural Styles

4. Layered architecture:

Layered architecture focuses on the **grouping of related functionality within an application** into distinct layers that are stacked vertically on top of each other.



Issued related to Architectural Patterns

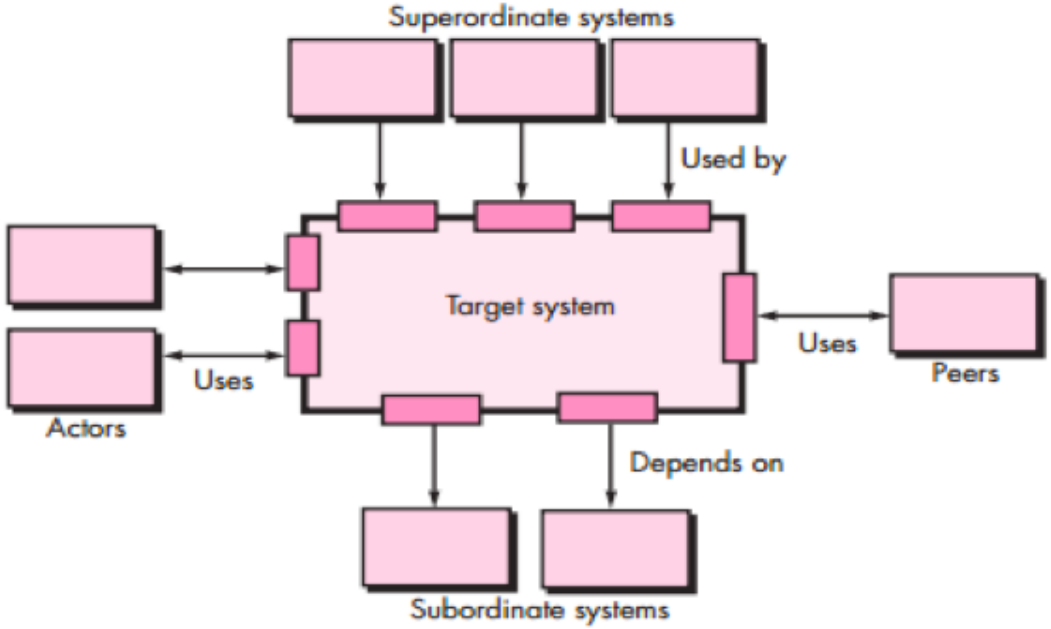
- **Concurrency**—applications must **handle multiple tasks** in a manner that simulates parallelism.
  - Example 1: **Operating System Process Management pattern** that provides built-in OS features that allow components to execute concurrently.
  - Example 2: **Task Scheduler pattern** invokes the next concurrent object
- **Distribution -** The distribution problem **addresses the manner in which systems or components** within systems **communicate** with one another in a **distributed environment.**
  - Example 1: **Broker pattern:** broker acts as a middle man to constitute communication.

Issued related to Architectural Patterns

- **Persistence - Data persists if it survives past the execution** of the process that created it.
- **Persistent data are stored** in a database or file and **modified later**.
  - Example 1: **Database management system pattern** which stores and retrieves data.

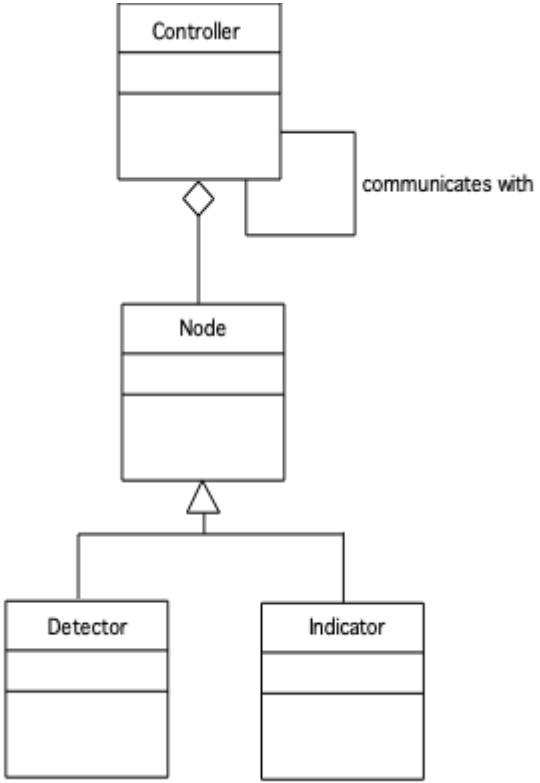
Architectural Design

- The software must be placed into context
  - The **design** should **define** the **external entities** (other systems, devices, people) that the **software interacts** with and the **nature of the interaction**



Architectural Design

- A set of architectural archetypes should be identified
  - An **archetype** is an abstraction (similar to a class) that **represents one element of system behavior**.
  - - An archetype is a class or **pattern that represents a core abstraction that is critical to the design** of an architecture for the target system.



Architectural Design

- The **designer specifies the structure** of the system by **defining and refining software components** that implement each archetype.

Architectural Design

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.

Different 3 types of cohesion

- **Functional -** Based on **operations** , this level of cohesion occurs when a component **performs a targeted computation** and then returns a result.
- **Layer -** Exhibited by packages, components, and classes, this type of cohesion occurs when a **higher layer accesses the services of a lower layer**, but **lower layers do not access higher layers.**
- **Communicational. All operations that access the same data are defined** within one class. Focus is only on data in question.

Architectural Design

Coupling

a qualitative measure of the **degree to which classes are connected to one another**.

Different 9 types of coupling

1. **Content coupling** **Occurs when one component secretly modifies data** that is internal to **another component** (friend function /class)
2. **Common coupling** Occurs when a number of components **all make use of a global variable**. Common coupling can **lead to error propagation.**
3. **Control coupling – Occurs when one operation controls flow of another operation** and control signals are sent. Changes done to an operation are to be monitored for errors. (send and acknowledge)

Architectural Design

Coupling

a qualitative measure of the **degree to which classes are connected to one another**.

Different 9 types of coupling

4. **Stamp coupling –** Occurs when systems **operation are in nested state** , modifying the system becomes complex. (semaphores)
   Similar to common coupling but here global data is accessible to selected routines.
5. **Data coupling - Occurs when operations pass long strings of data arguments.**

   Band width of communication increases complexity increases maintenance becomes difficult.

Architectural Design

Coupling

a qualitative measure of the **degree to which classes are connected to one another**.

Different 9 types of coupling

6. Routine call coupling - Occurs when one operation invokes another

7. Type use coupling. Occurs when component **A** uses a data type defined in component **B.**

If the type definition changes, every component that uses the definition must also change.

Architectural Design

Coupling

a qualitative measure of the **degree to which classes are connected to one another**.

Different 9 types of coupling

8. **Inclusion or import coupling.** Occurs when component **A** imports or includes a package or the content of component **B**.

9. **External coupling -** Occurs when **a component communicates or collaborates with infrastructure components** (e.g., operating system functions, database capability, telecommunication functions).

This type of coupling is necessary but it should also be limited.

Module 3

3.1 Design Quality, Classification of Design activities. Design concepts: Modularity & Layering. Introduction to Pattern-based software design.

3.2 Software Architecture, Data design, Object -Oriented versus Function - oriented Design, Design of software object, methods, Cohesion & Coupling between objects.

3.3 User Interface design: Rules, User Interface Analysis Steps in Interface design, Design Evaluation

3.4 Software Reuse, Component-based software design

User Interface Design Rules

1. Place user in control

- A system or user interface **should be designed keeping user in mind.**
- Interface designed should be **easy to use and build.**

User Interface Design

Design principles that allow the user to maintain control:

- **Define interaction modes in a way that does not force a user into unnecessary or undesired actions.**
  Example: Spell Checking mode
- **Provide for flexible interaction.** Because different **users have different interaction preferences**, choices should be provided.
  Example: software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen etc.

- **Allow user interaction to be interruptible and undoable.**
- **Hide technical internals from the casual user.**
- **Design for direct interaction with objects that appear on the screen.**

User Interface Design Rules

2. Reduce the users' memory load:

- The **more a user has to remember**, the **more error-prone the interaction** with the system will be.
- well-designed user interface does not burdon the user's memory.
- design an interface that to reduce the users' memory load:
  - **Reduce demand on short-term memory –** complex tasks demand on short term memory is high - interface designed to reduce dependency.
  - **Establish meaningful defaults -** "reset" option should be available for enabling original default values.
  - Defines short cuts that are more intuitive (easy to remember)
  - Ctrl- s, Ctrl-p , alt-p

User Interface Design Rules

2. Reduce the users' memory load:

- **The visual layout of the interface should be based on a real-world metaphor.**
- **Disclose information in a progressive fashion -**

  The interface should be organized hierarchically.

  An example, word-processing applications

User Interface Design Rules

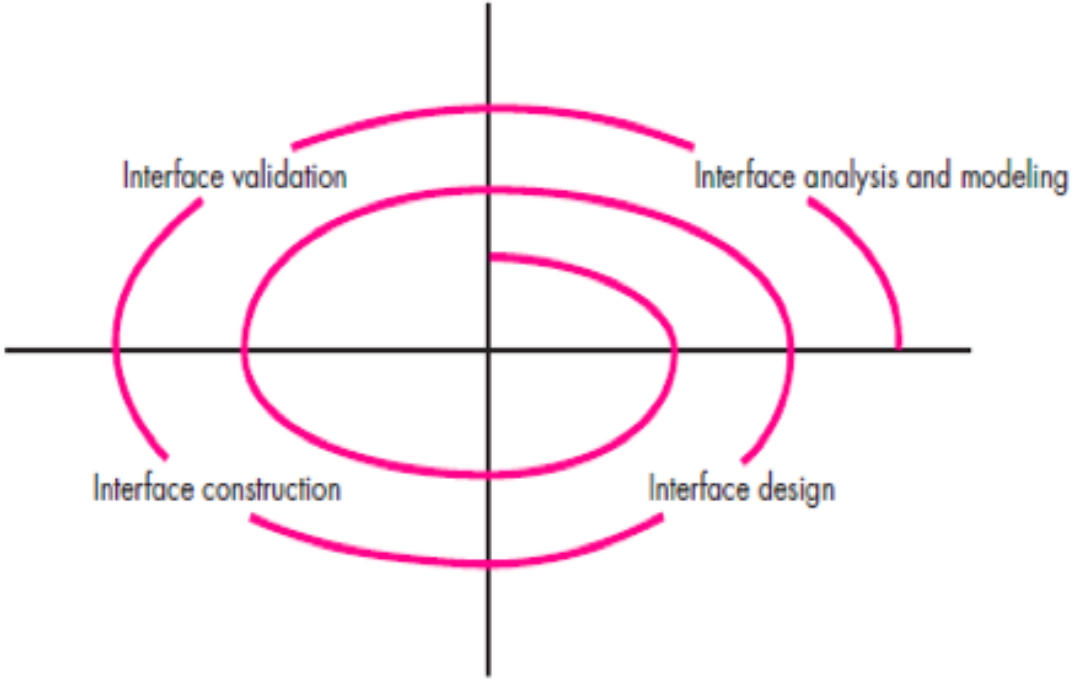
3. Make the interface consistent:

"Things that look different should act different. Things that look the same should act the same."

- **Allow the user to put the current task into a meaningful context.**
  Complex layers to be implemented with clear indication to the user about the context of the work at hand.
- **Maintain consistency across a family of applications.**
  A set of applications (or products) should all implement the same design rules - consistency is maintained for all interaction.
- **If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.**

User Interface Design Process

Iterative, Incremental



User Interface Design Process

(1) Interface analysis **–** focuses on the **profile of the users** who will interact with the system. Different **user categories** are **defined**.

- Once **general requirements** have been **defined**, a more **detailed *task analysis*** is conducted.

(2) *Interface design* **–** is to **define a set of interface objects and actions** that enable a user to perform all defined tasks to **meet goals** defined.

(3) Interface construction - normally begins with the **creation of a prototype** that enables usage scenarios to be evaluated.

(4) Interface validation - focuses on

a) the ability of the interface to implement **every user task correctly**, **accommodate all task va**riations, and to **achieve all general user requi**rements;

b) the degree to which the **interface is easy to use** and **easy to learn**

c) the **users' acceptance** of the **interface** as a useful tool in their work

Interface Analysis

Understand the problem before you attempt to design a solution.

- The people (**end users**) who will interact with the system through the interface.
- The **tasks that end users must perfo**rm to do their work,
- The **content that is presented** as part of the interface
- The **environment** in which these **tasks will be conducted**

Interface Analysis Steps

1. **User Analysis**

- **User interviews**
- **Sales Input**: Sales people meet with users on a regular basis and can gather information.
- **Marketing input.**
- **Support input -** Support staff talks with users on a daily basis.

Interface Analysis Steps

2. Task Analysis & Modelling

- Gather information about the **tasks and subtasks** performed by the user, their **workflow and hierarchy**
  - **Use cases.**
  - **Task elaboration** - stepwise elaboration (also called functional decomposition or stepwise refinement) for software to accomplish some desired function.
  - **Object elaboration:** Examine use case and other information obtained from the user.
  - **Workflow analysis -** When a number of different users, makes use of a user interface workflow analysis becomes important.
  - E.g: Pharmaceutical Company
  - **Hierarchical representation -** Once workflow has been established, a task hierarchy can be defined for each user type.

Interface Analysis Steps

3. Analysis of Display Content - the presentation of a variety of different types of content.

•For modern applications, display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or video files).

4. Analysis of the Work Environment

Interface Design Steps

- Using **information developed during interface analysis**, define interface objects and actions (operations).
- **Define events (user actions) that will cause the state of the user interface to change**. Model this behaviour.
- Depict each **interface state as it will actually look to the end-user.**
- **Indicate how the user interprets the state of** the system from information provided through the **interface.**

Module 3

3.1 Design Quality, Classification of Design activities. Design concepts: Modularity & Layering. Introduction to Pattern-based software design.

3.2 Software Architecture, Data design, Object -Oriented versus Function - oriented Design, Design of software object, methods, Cohesion & Coupling between objects.

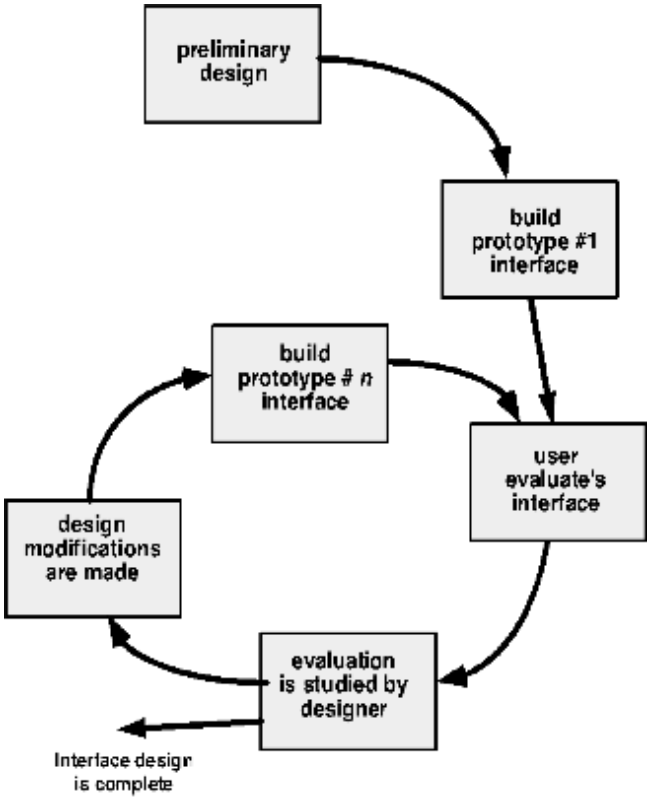
3.3 User Interface design: Rules, User Interface Analysis Steps in Interface design, Design Evaluation

3.4 Software Reuse, Component-based software design

Design issues

- Response time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility – for physically challenged users
- Internationalization

Design Evaluation Cycle



Design with Reuse

- Building software with reusable components
- Systems are designed by composing existing components that have been used in other systems
- to achieve better software, more quickly and at lower cost, adopt a design process that is based on systematic reuse

Reuse based Software Engineering

Application system reuse

- The whole of an application system may be reused either by incorporating it without change into other systems or by configuring the application for different customers.
- Application families that have a common architecture, but which are tailored for specific customers, may be developed.

Component reuse

- Components of an application from sub-systems to single objects may be reused.

Object and Function reuse

- Software components that implement a single function, such as a mathematical function, or an object class may be reused.

Benefits of Reuse

- Increased reliability
  - Components exercised in working systems
- Reduced process risk
  - Less uncertainty in development costs
- Effective use of specialists
  - Reuse components instead of people
- Standards compliance
  - Embed standards in reusable components
- Accelerated development
  - Avoid original development and hence speed-up production

Components Based Software Engineering (CBSE)

- **Approach to software development that relies on reuse**

- It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific
- **Components are more abstract than object classes** and can be considered to be stand-alone service providers
- Components can range in size from simple functions to entire application systems

Key goals of CBSE

- Save time and money when building large and complex systems
- Enhance the software quality
- Detect defects within the systems
- Improved efficiency

Component Interfaces

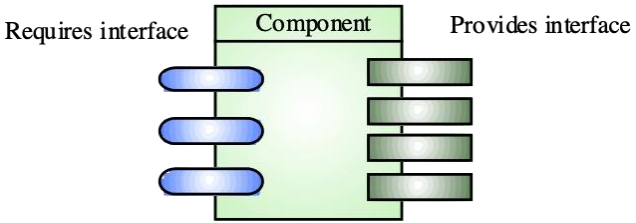
These interfaces reflect the services that the component provides and the services that the component requires to operate correctly.

Provides interface

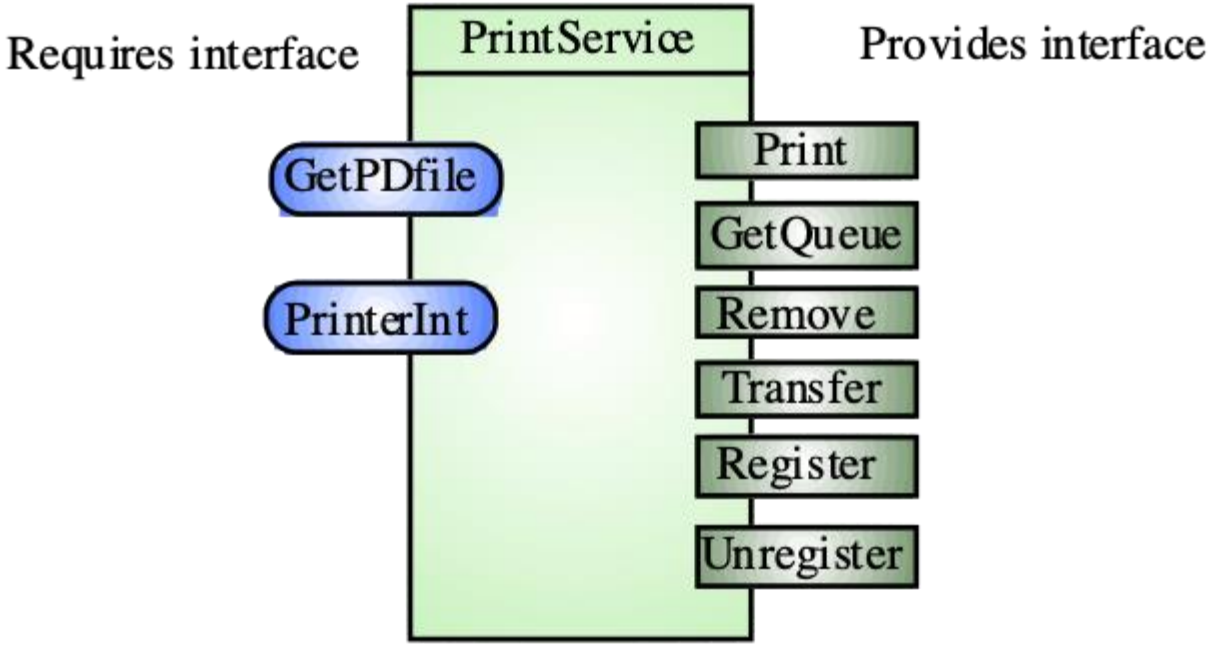
Defines the services that are provided by the component to other components

Requires interface

Defines the services that specifies what services must be made available for the component to execute as specified



Example: Printing service Component



CBSE Process

- CBSE processes are software processes that support component-based software engineering.
- They take into account the possibilities of reuse and the different process activities involved in developing and using reusable components.

1. **Development for reuse**
   - This process is concerned with **developing components or services that will be reused** in other applications.
   - It usually involves generalizing existing components
1. **Development with reuse**
   - This is the process of **developing new applications using existing components and services**.

CBSE for reuse

CBSE for reuse is the process of developing reusable components and making them available for reuse through a component management system.

CBSE with reuse