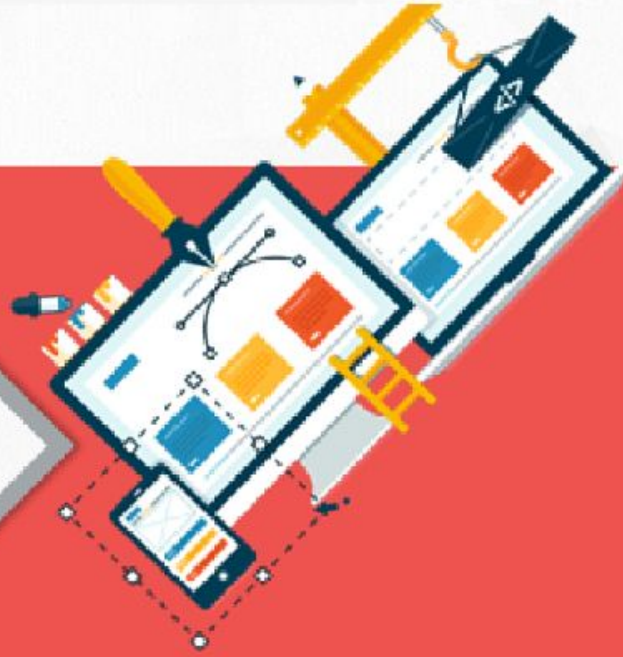
Software Engineering

Unit-2
Software Design



Software Engineering: A Practitioner's Approach, 8/e
by Roger S. Pressman

Software Engineering 10/e
By Ian Sommerville

Syllabus : Software Engineering

Module #01: Introduction To Software Engineering

Software Process Structure – Process Models & Activities – Agile Development - Requirements Engineering.

Module #02: Software Modeling

Design Concepts – Architectural Design – Component Level Design – User Interface Design.

Module #03: Quality Management

Review Techniques – Software Quality Assurance – Software Testing Strategies – Software Configuration Management – Product Metrics.

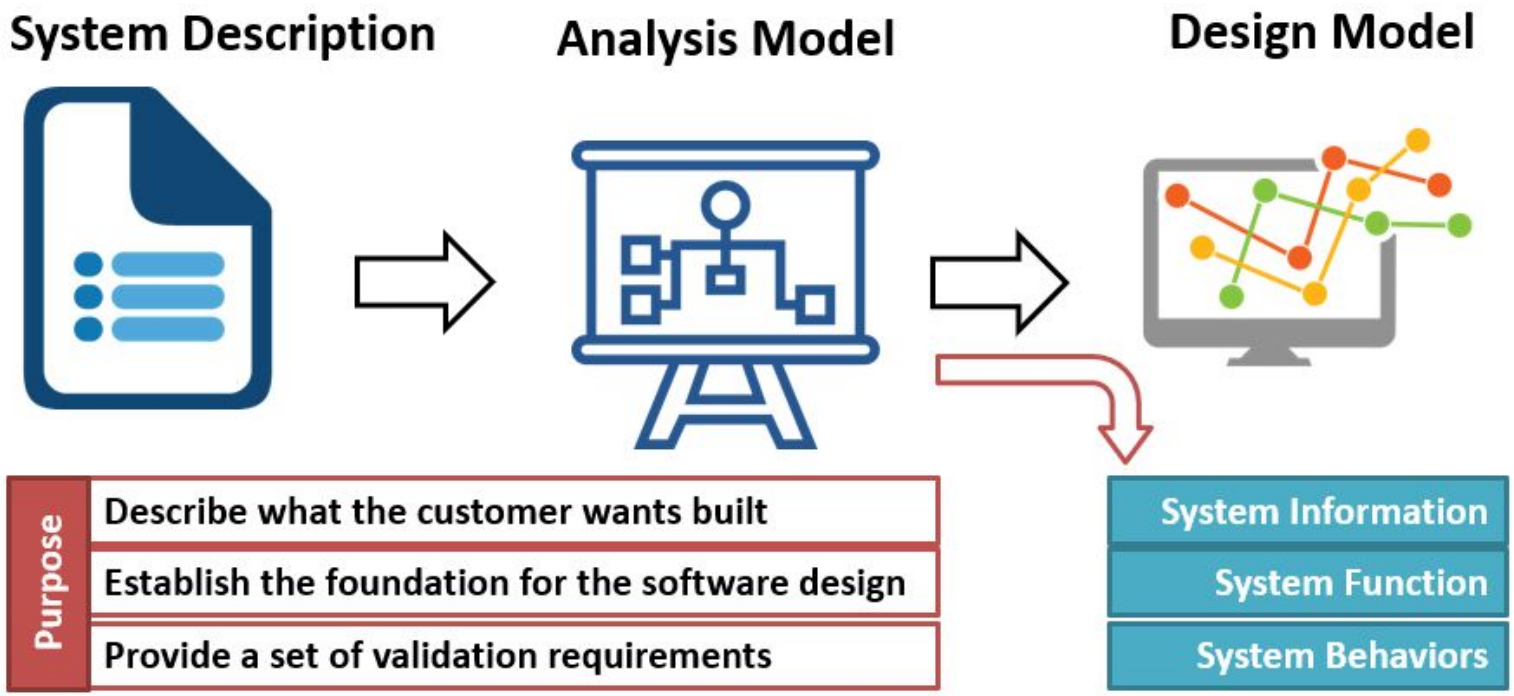
Module #04: Managing Software Projects

Project Management Concepts – Process & Project Metrics – Estimation For Software Projects – Project Scheduling – Risk Management.

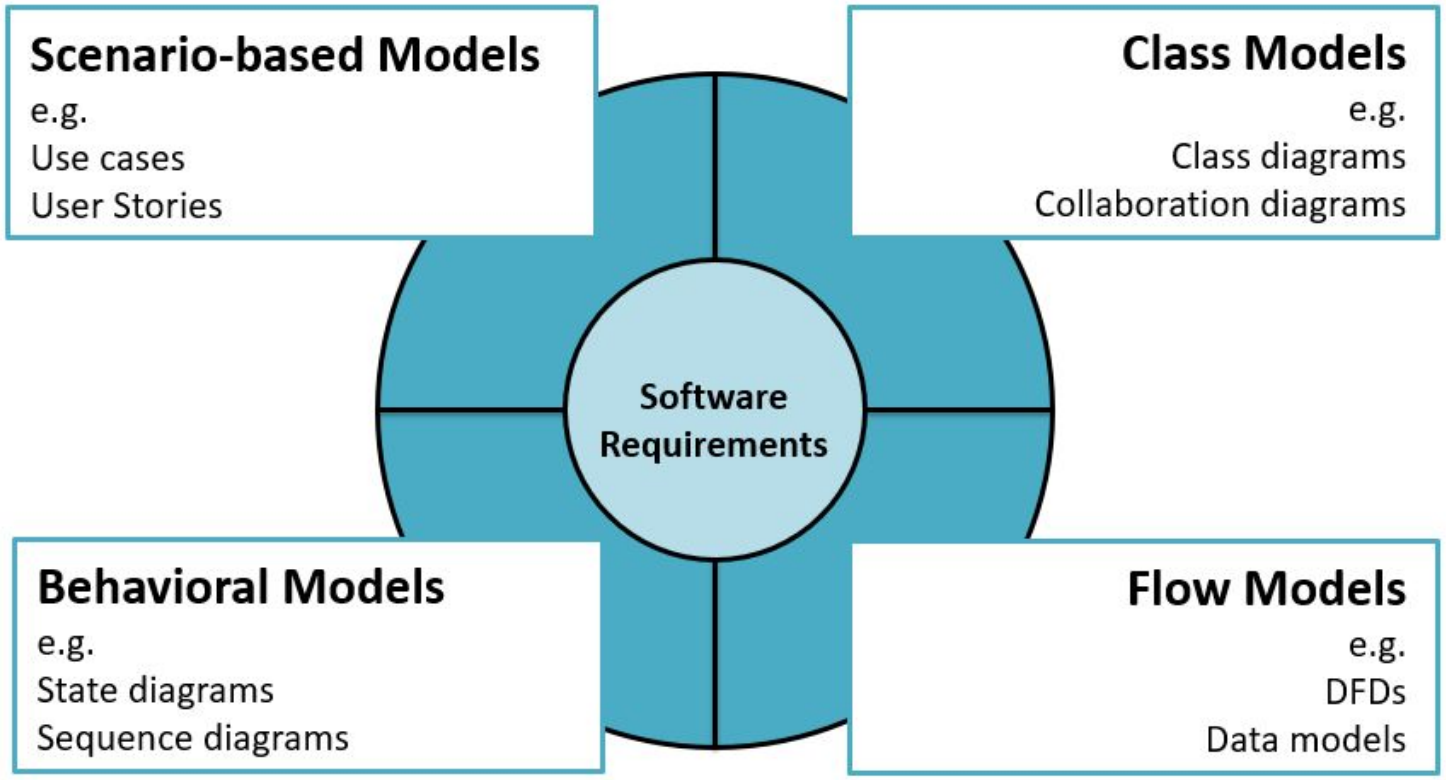
Module #05: Reliability & Security

Reliability Engineering – Reliability & Availability – Reliability Testing. Security Requirements & Design.

The requirement analysis model



Elements of the Requirements Model

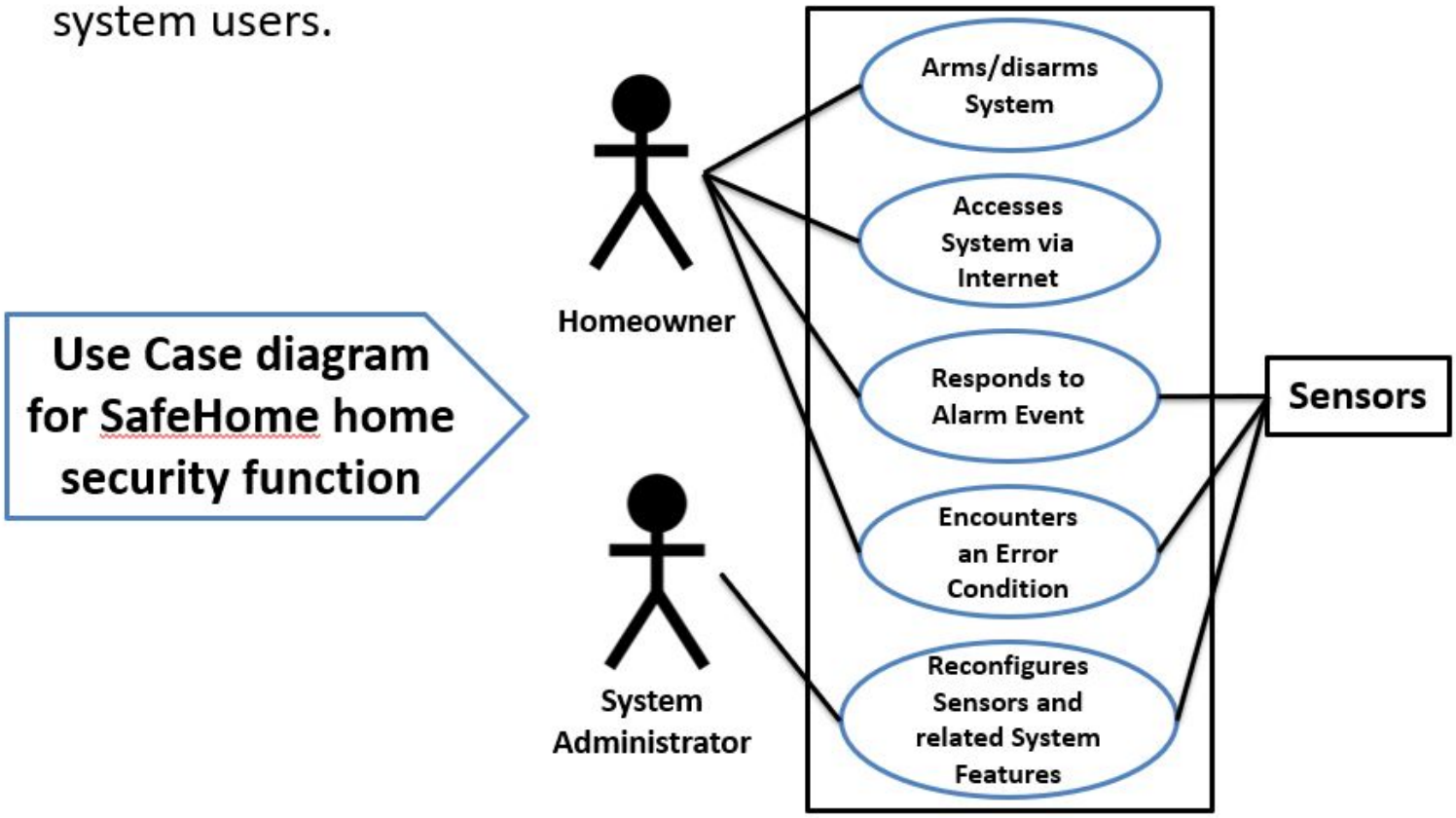


Use-Cases

- A collection of user scenarios that describe the thread of usage of a system
- Each scenario is described from the point-of-view of an "actor"
  - Actor: a person or device that interacts with the software
- Each scenario answers the following questions:
  - Who is the primary actor, the secondary actor (s)?
  - What are the actor's goals?
  - What preconditions should exist before the story begins?
  - What main tasks or functions are performed by the actor?
  - What extensions might be considered as the story is described?

Use-Case Diagram

- It is referred as the diagram used to describe a set of actions (use cases) that some system should perform in collaboration with system users.

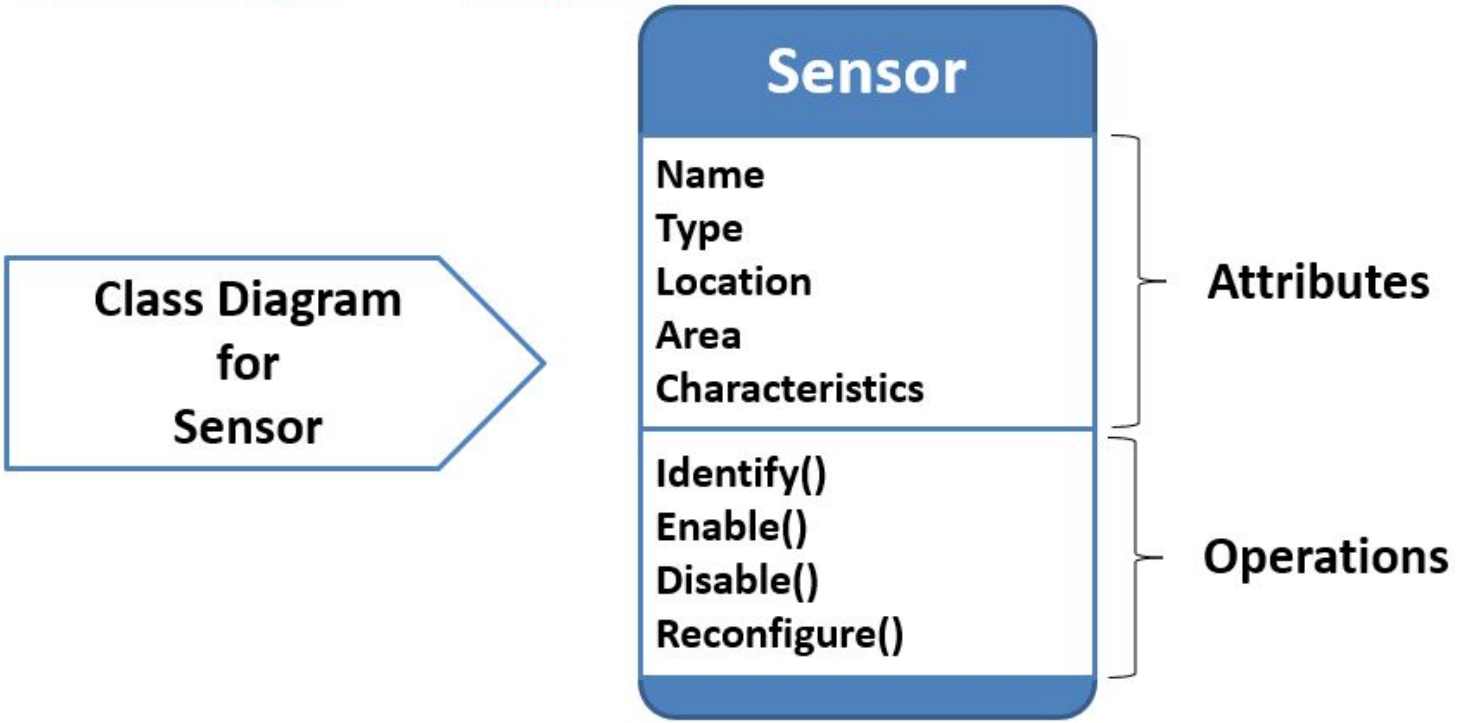


Use-Case

1	**Use Case Title**	**Login**
2	**Abbreviated Title**	Login
3	**Use Case Id**	1
4	**Actors**	Librarian , Members, Asst. Librarian
5	**Description:** To interact with the system, LMS will validate its registration with this system. It also defines the actions a user can perform in LMS.	
5.1	**Pre Conditions:** User must have proper client installed on user terminal	
5.2	**Task Sequence** 1. System show Login Screen 2. User Fill in required information. Enter user name and password 3. System acknowledge entry	
5.3	**Post Conditions:** System transfer control to user main screen to proceed further actions	
5.4	**Exception:** If no user found then system display Invalid user name password error message and transfer control to Task Sequence no.1	
6	**Modification history:** Date 08-01-2018	
7	**Author:** Pradyumansinh Jadeja Project ID LMS	

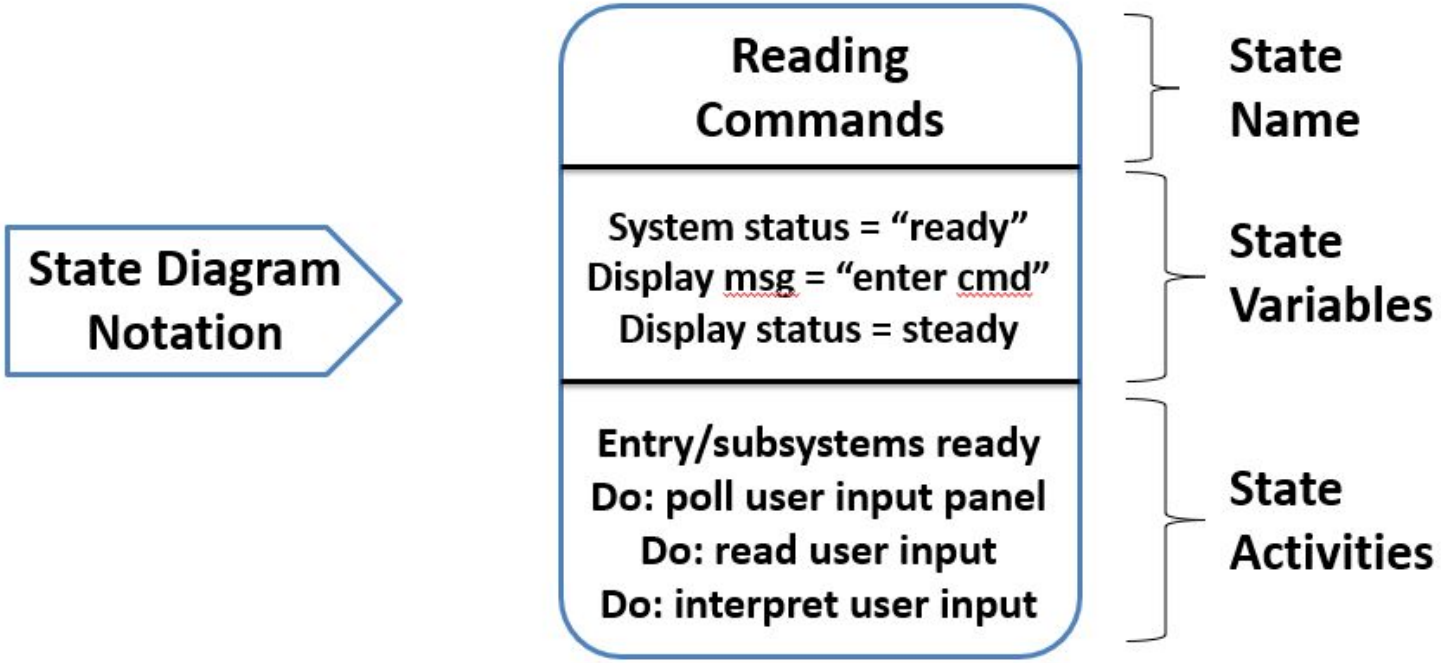
Class Diagram

- It **describes** the structure of a system by showing the **system's classes**, their **attributes**, **operations** (or methods), and the **relationships** among **objects**.



State Diagram

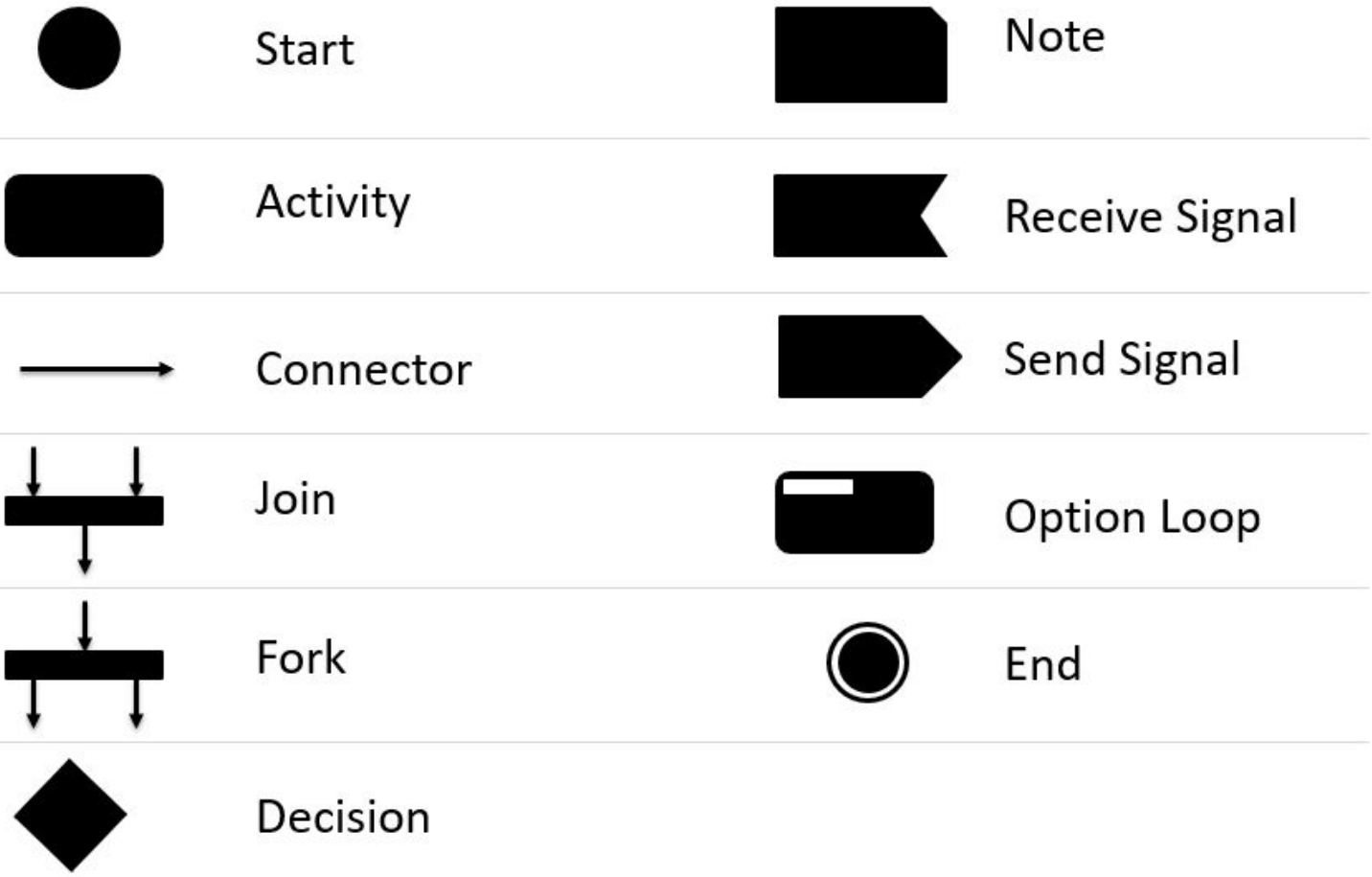
- It is used to **describe** the **behaviour** of **systems**.
- It requires that the system described is composed of a finite number of states.



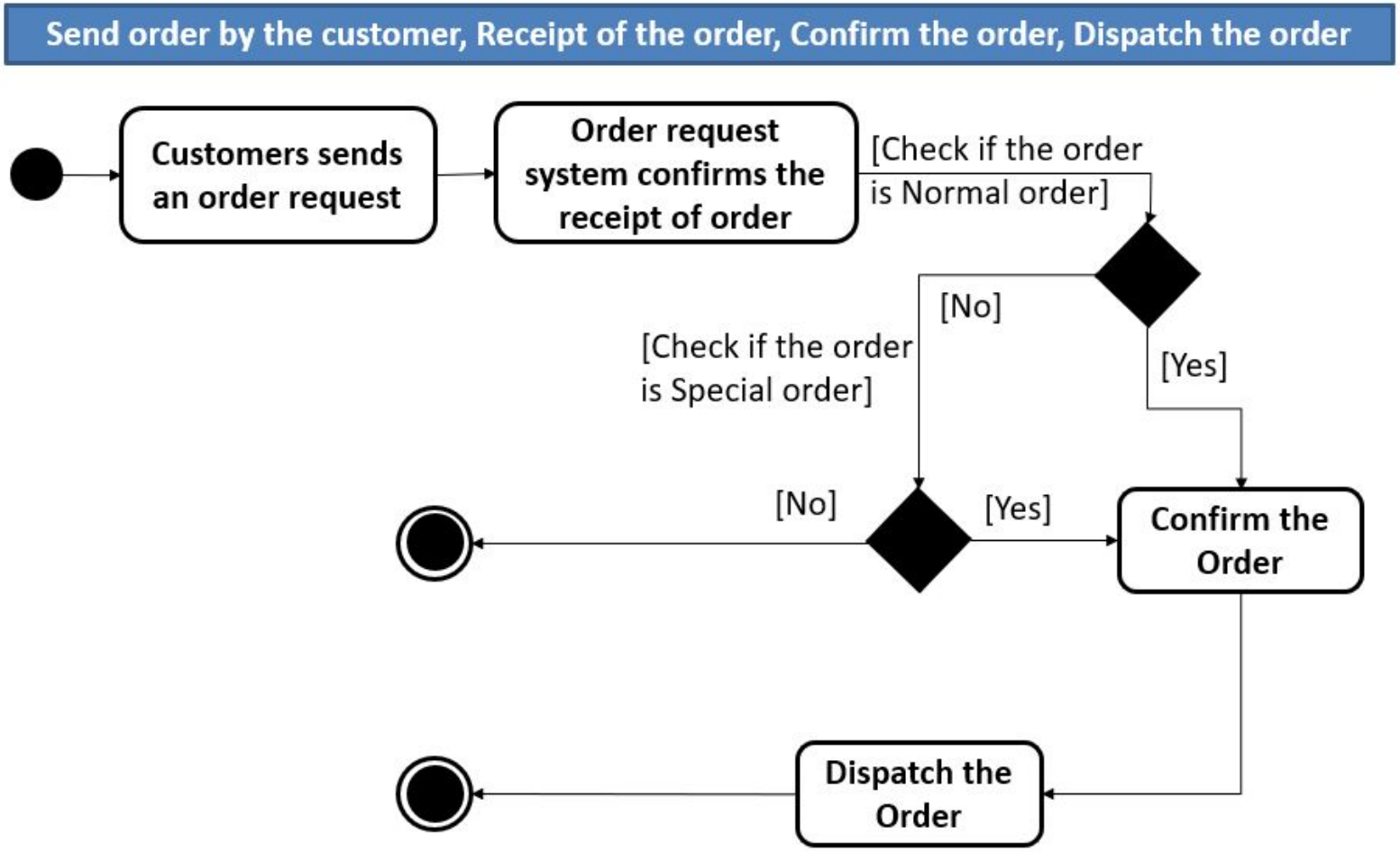
Activity & Swimlane Diagram

- **Activity diagram** is basically a **flowchart** to **represent** the **flow** from one **activity** to another **activity**
- The activity can be described as an operation of the system.
- A **swimlane diagram** is a type of activity diagram. Like activity diagram, it diagrams a process from start to finish, but it also **divides** these **steps** into **categories** to help **distinguish** which departments or employees are **responsible** for each set **of actions**
- A swim lane diagram is also useful in helping **clarify responsibilities** and help departments work together in a world where departments often don't understand what the other departments do

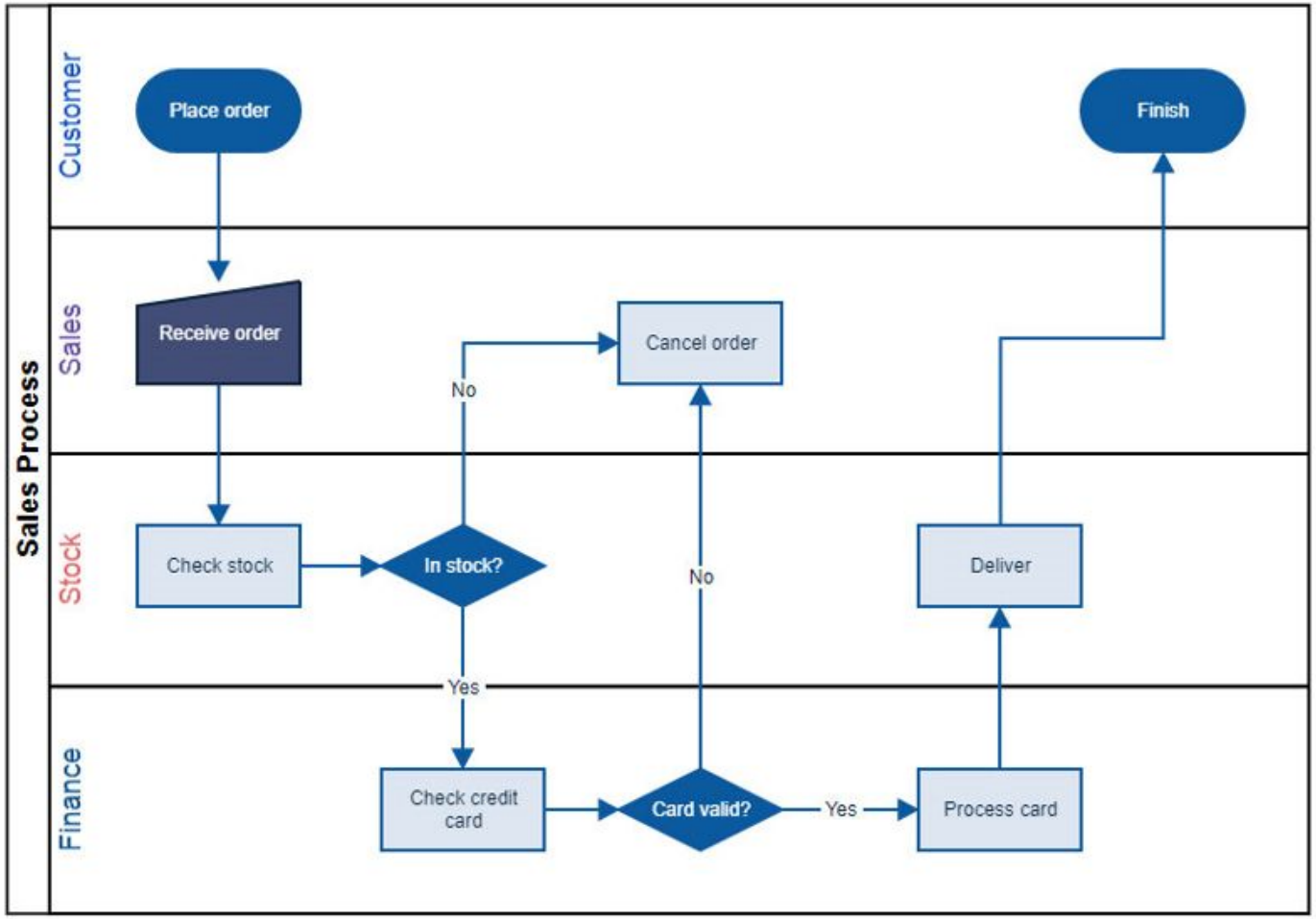
Activity Diagram Symbols



Activity diagram of order processing

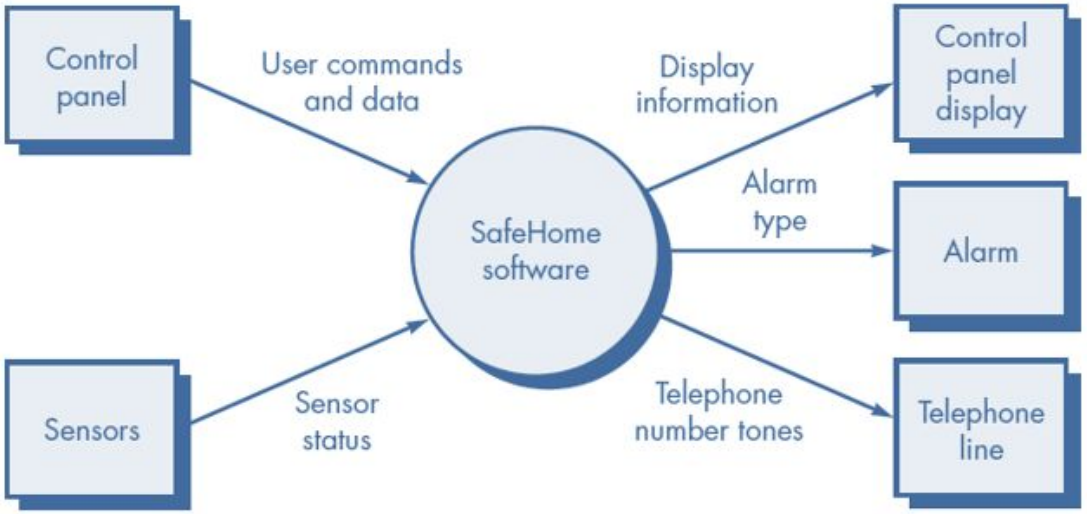


Swimlane diagram of order processing



Data Flow Diagram (DFD)

- It is a graphical representation of the "flow" of data through an information system, modelling its process aspects
- It is often **used** as a preliminary step to create an overview of the system, which can later be elaborated



Context-level DFD for the SafeHome security function

Software Requirements Specification

Software Requirements Specification

- Software Requirement Specification (SRS) is a document that completely describes what the proposed software should do without describing how software will do it.
- It contains:
  - a complete information description
  - a detailed functional description
  - a representation of system behaviour
  - an indication of performance requirements and design constraints
  - appropriate validation criteria
  - other information suitable to requirements
- SRS is also helping the clients to understand their own needs.

Standard Template for writing SRS

- **Front Page**

  Software Requirements Specification

  for

  <Project>

  Version <no.>

  Prepared by <author>

  <organization>

  <date created>
- **Table of Contents**
- **Revision History**

Standard Template for writing SRS (Cont…)

1. Introduction

- 1.1 Purpose
- 1.2 Document Conventions
- 1.3 Intended Audience and Reading Suggestions
- 1.4 Project Scope
- 1.5 References

2. Overall Description

- 2.1 Product Perspective
- 2.2 Product Features
- 2.3 User Classes and Characteristics
- 2.4 Operating Environment
- 2.5 Design and Implementation Constraints

Standard Template for writing SRS (Cont…)

2.6 User Documentation

2.7 Assumptions and Dependencies

3. System Features

3.1 System Feature 1

3.2 System Feature 2 (and so on)

4. External Interface Requirements

4.1 User Interfaces

4.2 Hardware Interfaces

4.3 Software Interfaces

4.4 Communications Interfaces

Standard Template for writing SRS (Cont…)

5. Other Nonfunctional Requirements

- 5.1 Performance Requirements
- 5.2 Safety Requirements
- 5.3 Security Requirements
- 5.4 Software Quality Attributes

6. Other Requirements

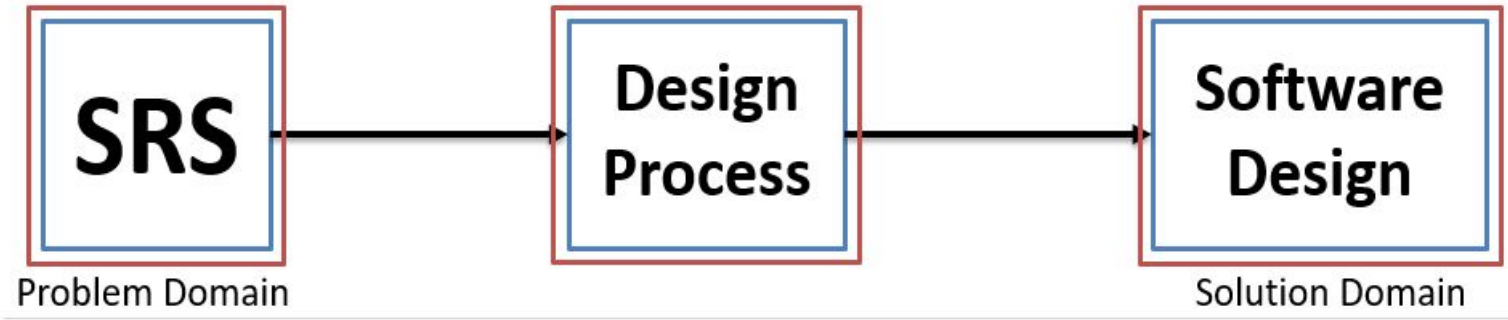
Appendix A: Glossary

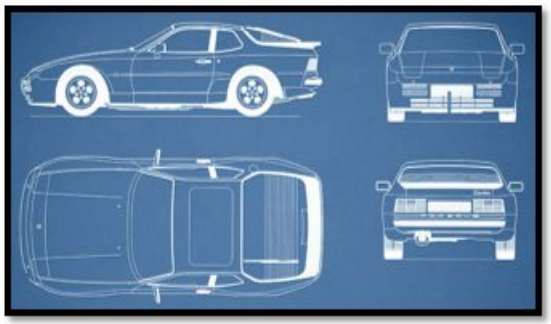
Appendix B: Analysis Models

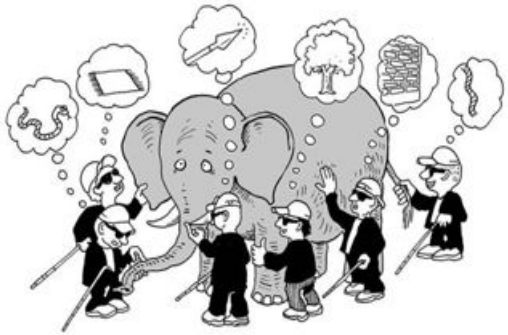
Appendix C: Issues List

What is Design?

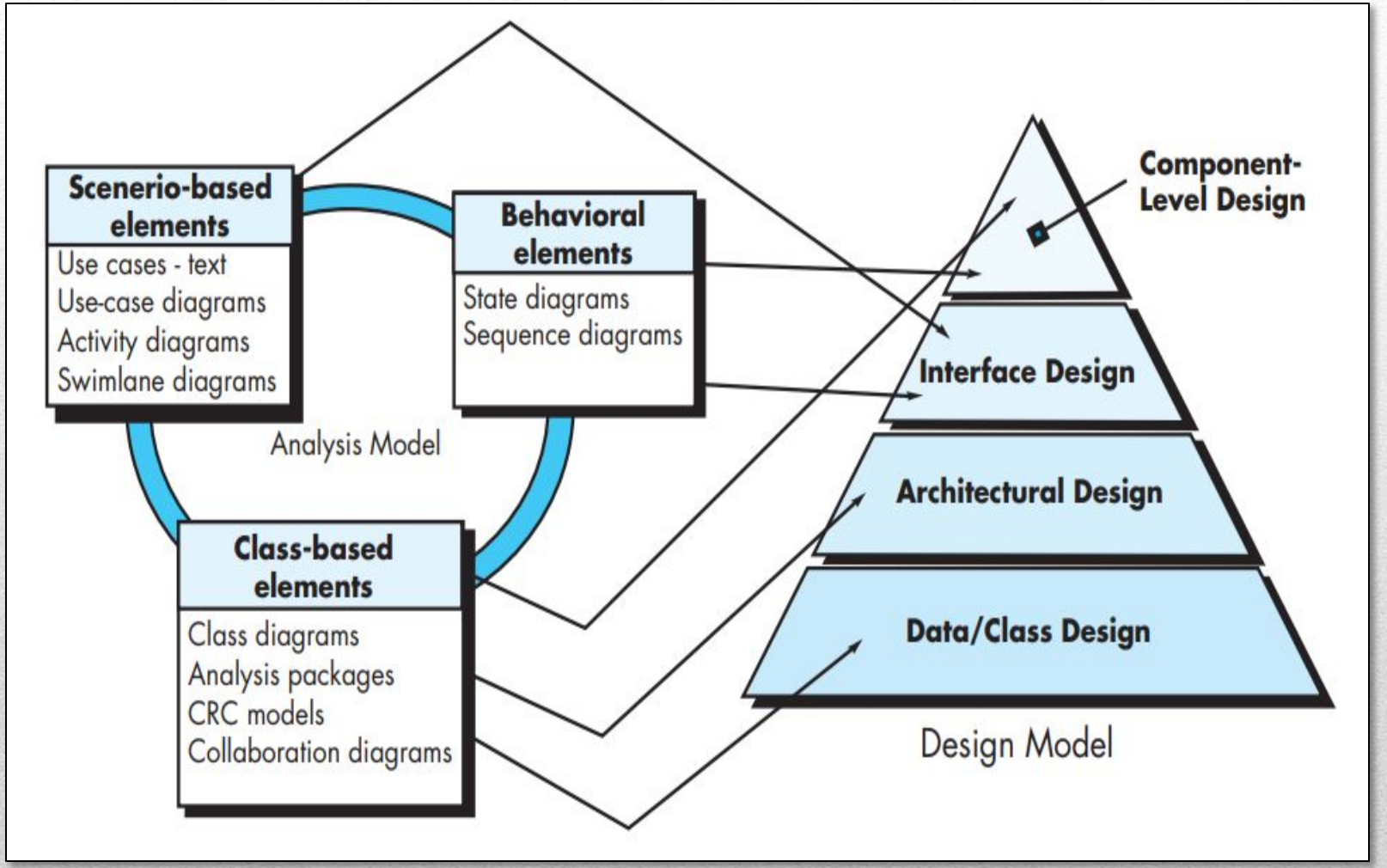
- A **meaningful representation** of something to be built
- It's a **process** by which **requirements** are **translated** into **blueprint** for constructing a software
- **Blueprint** gives us the **holistic view** (entire view) of a **software**







Design Model in Software Engineering



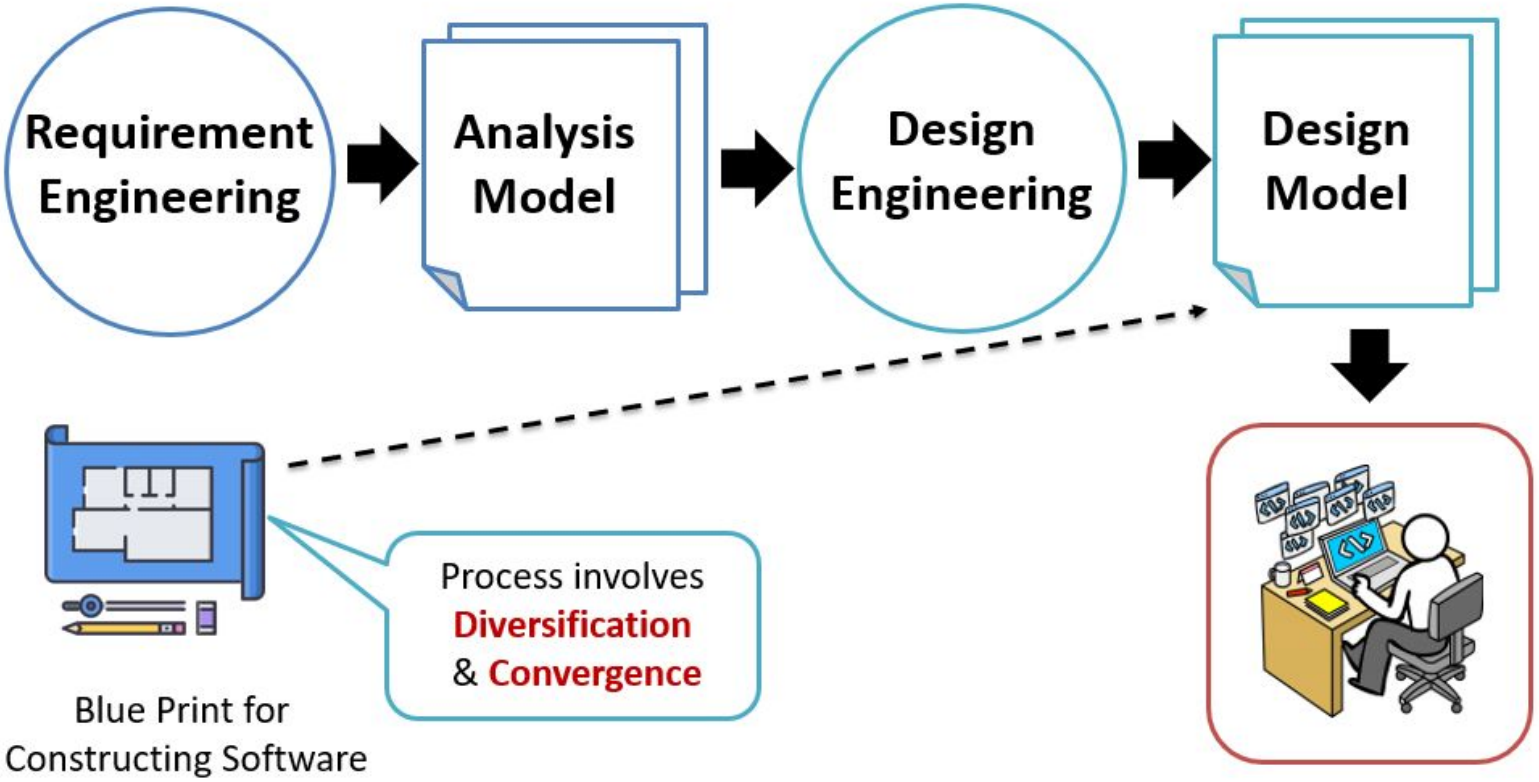
The Process of Design

- During high-level Design, Attempting to answer the following Questions.
  - What exactly is design?
  - What general approaches do designers use?
  - What types of design are there?

Definition: in the context of software, *design* is a problem-solving process whose objective is to find and describe a way to implement the system's functional requirements, while respecting the constraints imposed by the quality, platform and process requirements (including the budget and deadlines), and while adhering to general principles of good quality.

- A Designer is faced with a series of *Design Issues*, which are Sub-problems of the Overall Design Problem. Each issue normally has Several Alternative Solutions, also known as Design *Options*. The designer makes a *Design Decision* to **Resolve each issue** – this Process involves choosing what He or She Considers to be the Best option from Among the Alternatives.

Software Design Process?



The most creative part of the development process

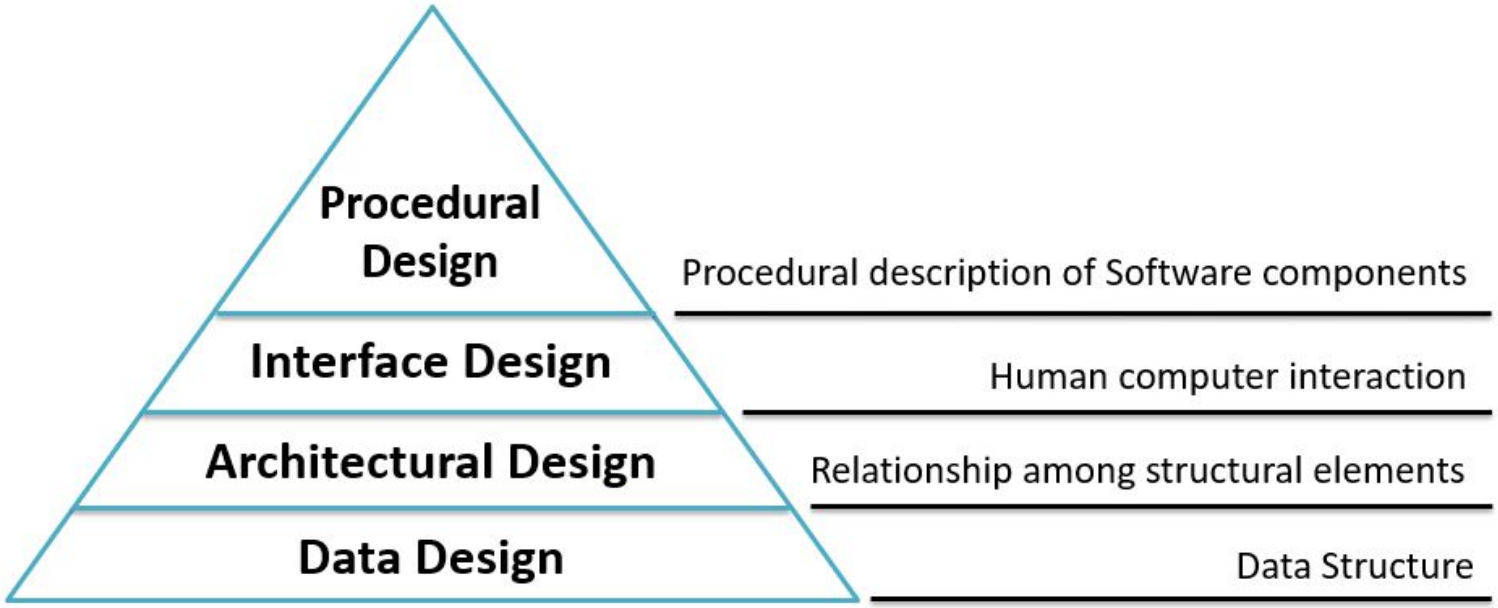
Design Concepts

The beginning of **wisdom** for a **software engineer** is to **recognize** the **difference** between **getting program to work** and **getting it right**

- » Abstraction
- » Architecture
- » Pattern
- » Separation of Concern
- » Modularity
- » Information Hiding
- » Functional Independence
- » Refinement
- » Aspects
- » Refactoring

Design Models

- It is **creative** activity
- Most **critical activity** during system development
- Has great **impact** on **testing** and **maintenance**
- Design document forms **reference for later phases**



Design Models Cont.

Data Design

It **transforms** **class models** into **design class** realization and **prepares** **data structure** (**data design**) required to implement the software.

Architectural Design

It **defines** the **relationship** **between** major **structural** **elements** of the software

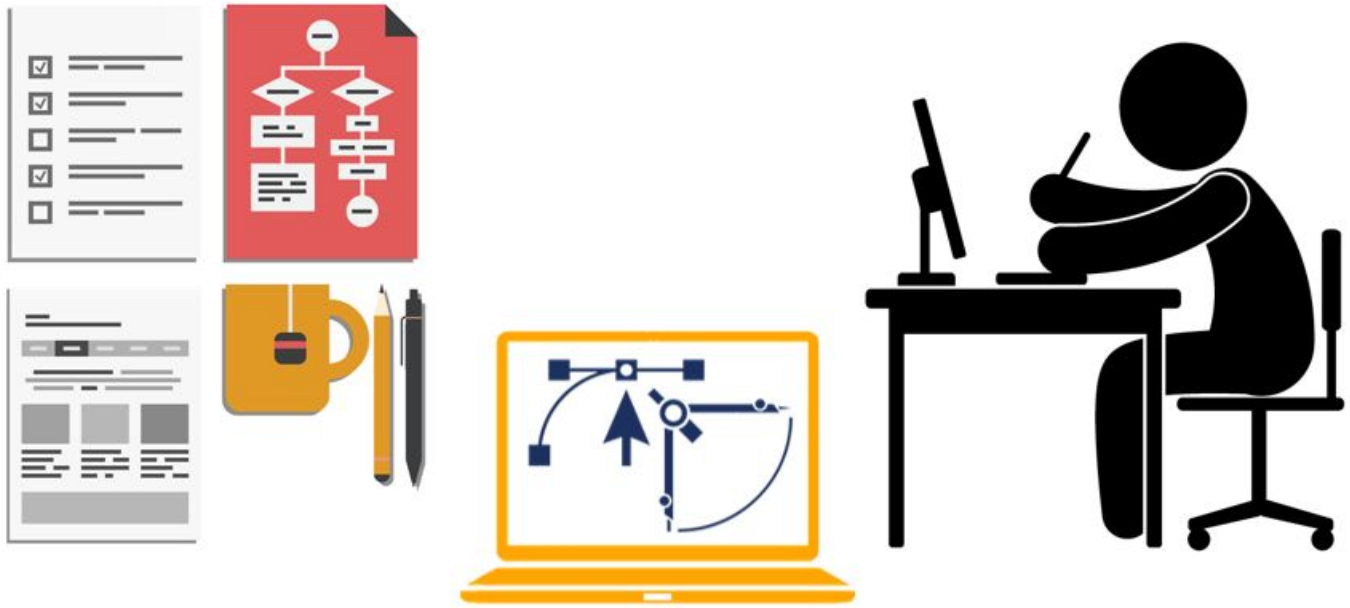
Interface Design

It defines **how** **software** **communicates** with **systems** & with **humans**. An interface implies flow of information & behavior.

Procedural Design

It **transforms** **structural** **elements** of software into **procedural** **description** of software components

Architectural Design



Software Architecture & Design

- **Large systems** are **decomposed** into **subsystems**
- **Sub-systems** provide **related services**
- Initial design process includes
  - Identifying sub-systems
  - Establishing a framework for sub-system control and communication

Why to document the Architecture?

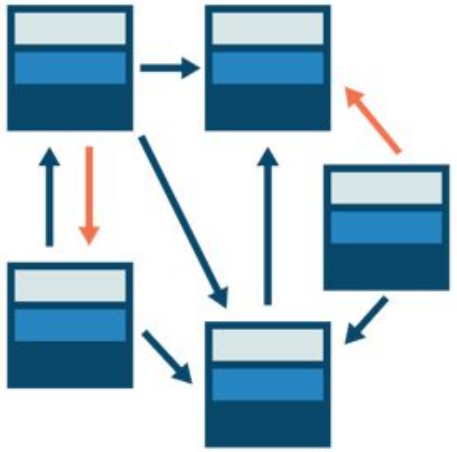
- **Stakeholder Communication:** High-level presentation of system
- **System Analysis:** Big effect on performance, reliability, maintainability and other –ilities **(Usability, Maintainability, Scalability, Reliability, Extensibility, Security, Portability)**
- **Large-scale Reuse:** Similar requirements similar architecture

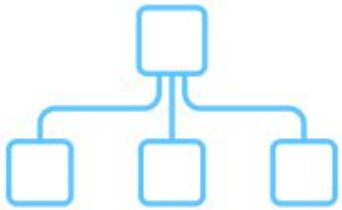
Software Architecture & Design

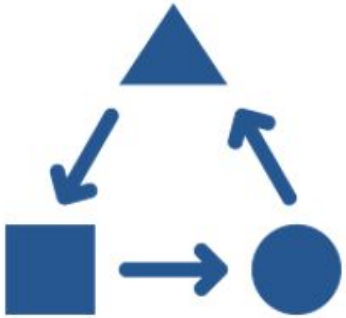
- Architectural design represents the **structure of data and program components**
- It considers,
  - **Architectural style** that the system will take,
  - **Structure** and **properties** of the **components** that constitute the system, and
  - **Interrelationships** that occur among all architectural components of a system.
- Representations of software architecture are an **enabler for communication between all parties** (stakeholders).
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together"

Architectural Styles

- Data-centered architecture style
- Data-flow architectures
- Call and return architecture
- Object-oriented architecture
- Layered architecture



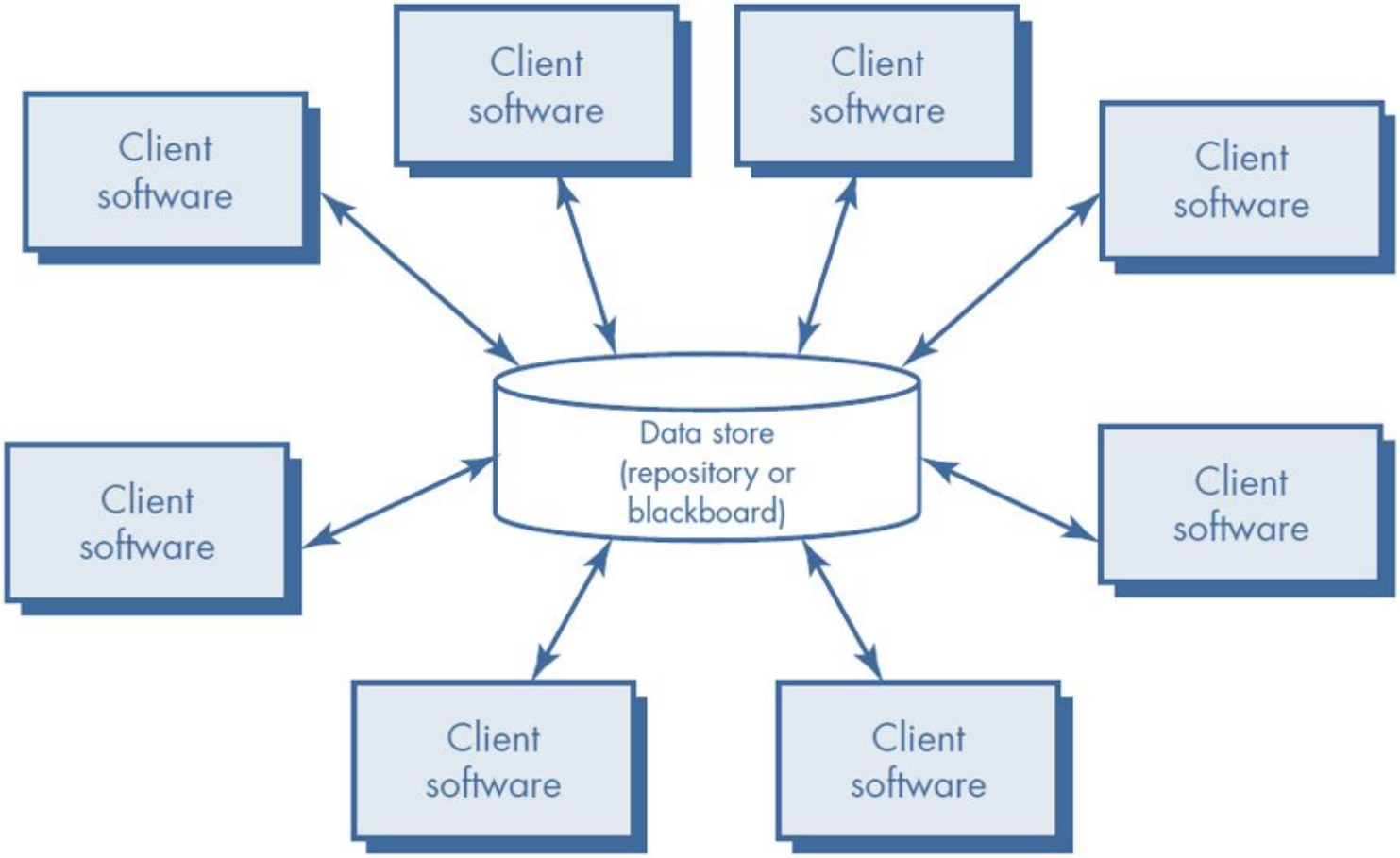




Architectural Styles cont..

- Each style describes a system category that encompasses,
  - A **set of components** (Ex., a database, computational modules) that **perform a function** required by a system.
  - A set of **connectors** that **enable "communication, coordination and cooperation"** among components.
  - **Constraints** that define **how components can be integrated** to form the system.
  - **Semantic models** that enable a designer **to understand the overall properties** of a system.

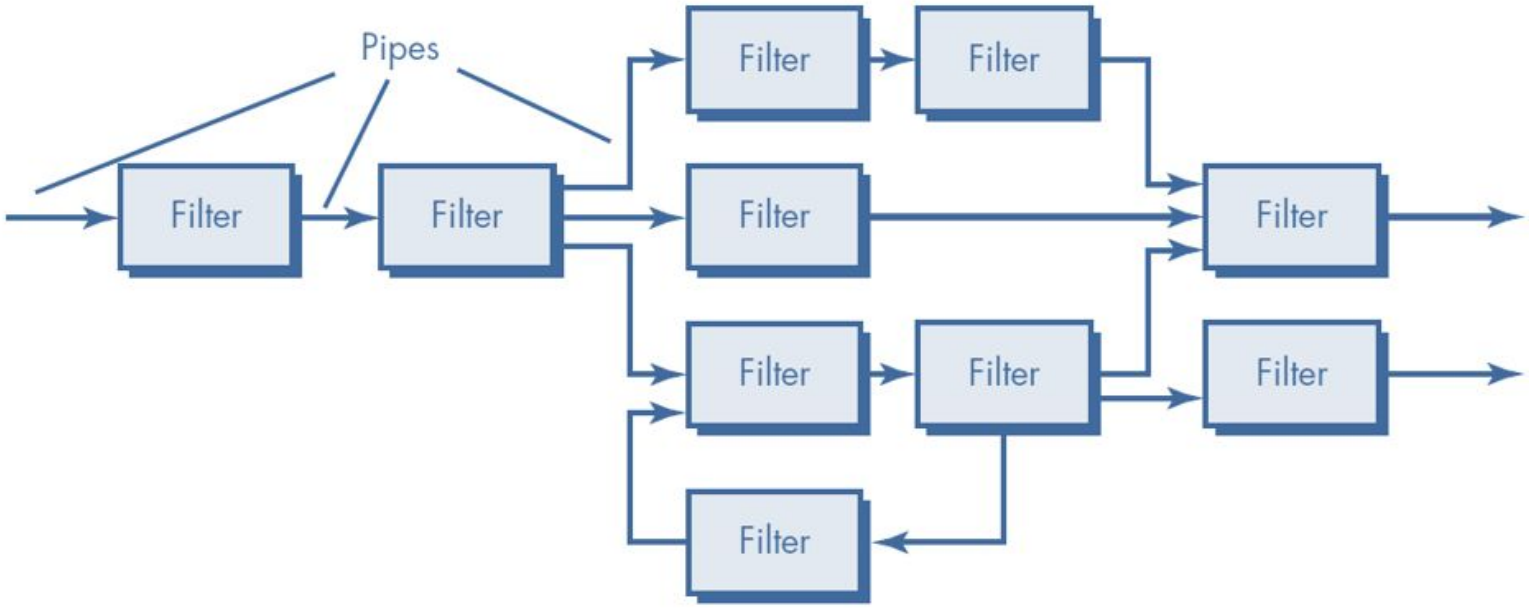
Data-centered architecture style



Data-centered architecture style

- **A data store** (Ex., a file or database) **resides at the center** of this architecture and is **accessed frequently** by other components.
- Client **software accesses a central repository**.
- In **some cases** the data **repository is passive**.
  - That is, client software accesses the data independent of any changes to the data or the actions of other client software.

Data-flow architectures



Pipes and filters

Data-flow architectures cont..

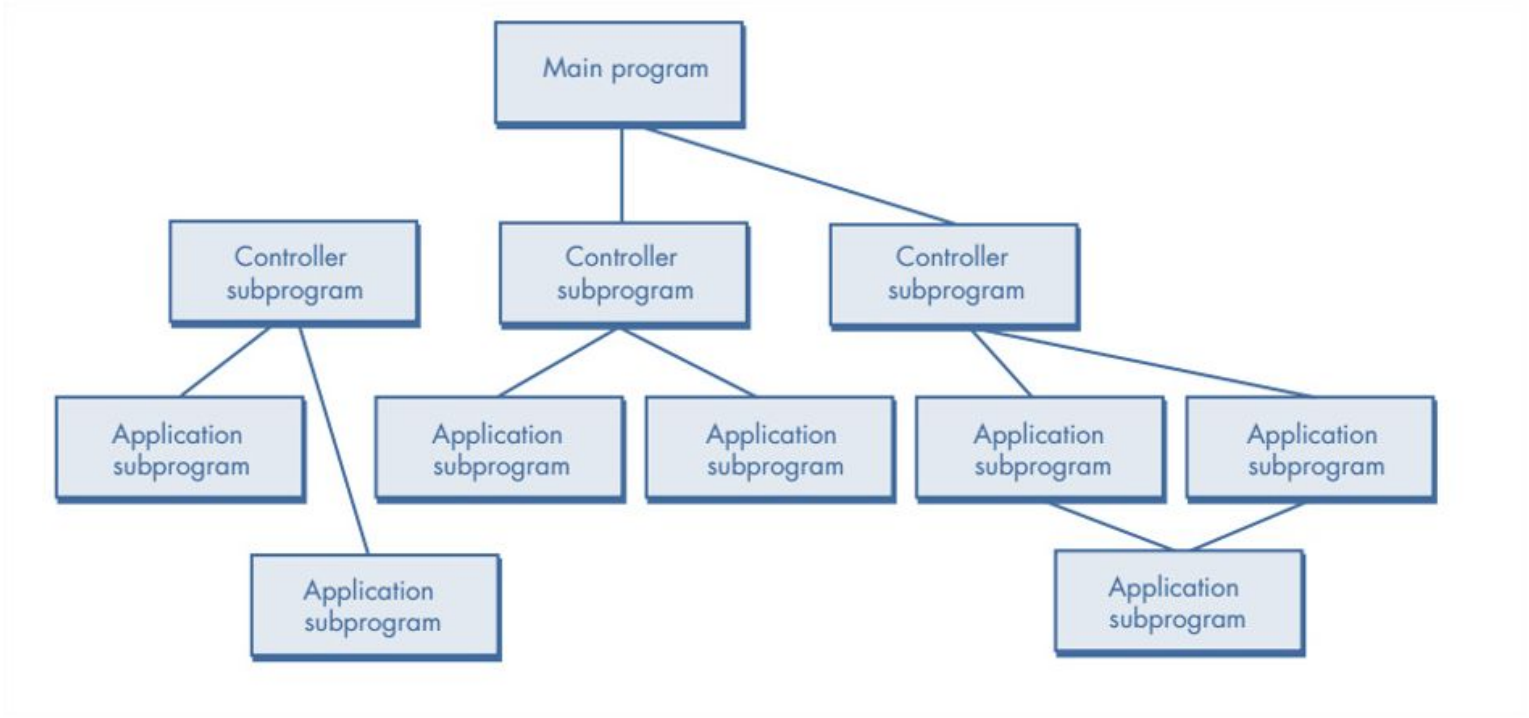
- This architecture is applied **when input data are to be transformed**.
- A set of components (called **filters**) **connected by pipes** that **transmit data** from one component to the next.
- Each filter works independently of those components upstream and downstream, is designed to
  - expect data input of a certain form, and
  - produces data output (to the next filter) of a specified form.

Call and return architecture

- 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 sub styles exist within this category as below.

1. **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.
2. **Remote procedure call architectures**
   - The components of a main **program/subprogram** architecture are **distributed across multiple computers** on a network.

Call and return architecture

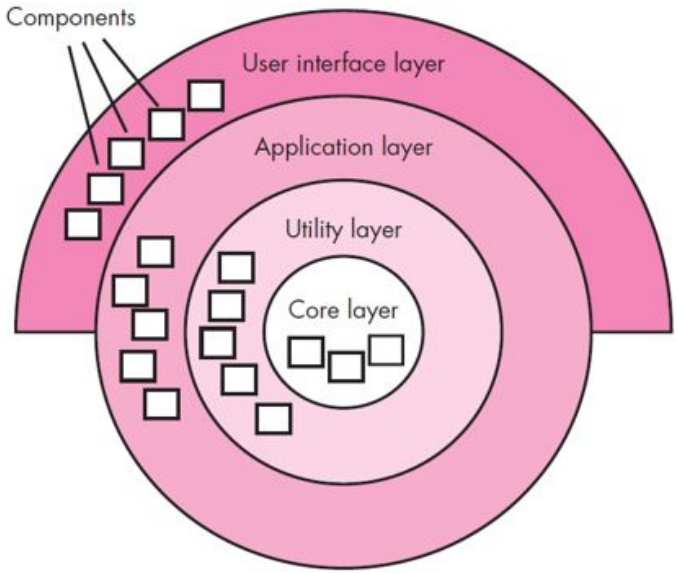


Object-oriented architecture

- The **components** of a system **encapsulate data and the operations** that must be applied to manipulate the data.
- **Communication** and **coordination** between components is accomplished **via message passing**.

Layered architecture

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



User Interface Design

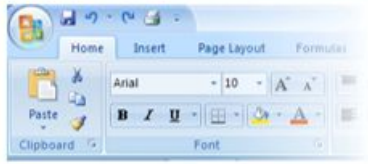
Golden Rules of User Interface Design

Place the User in Control	Reduce the User's Memory Load
	

Make the Interface Consistent

Microsoft Word

Microsoft Excel



Microsoft Powerpoint

Place the User in Control

- During a requirements-gathering session for a major new information system, a key user was asked about.
- Following are the **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**.
  - Provide for **flexible interaction**.
  - Allow user **interaction** to be interruptible and **undoable**.
  - Streamline interaction as skill levels advance and allow the interaction to be customized.
  - **Hide technical internals** from the casual user.
  - Design for **direct interaction with objects** that appear on the screen.

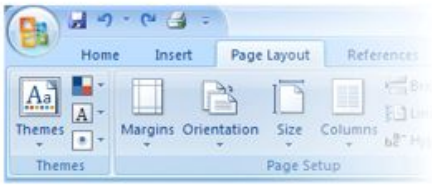
Reduce the User's Memory Load

- The **more a user has to remember, the more error-prone** the interaction with the system will be.
- Following are the design principles that enable an interface to reduce the user's memory load:
  - **Reduce demand** on **short-term memory**.
  - Establish **meaningful defaults**.
  - **Define shortcuts** that are intuitive.
  - The **visual layout** of the interface should be **based on** a **real-world metaphor**.
  - Disclose **information** in a **progressive fashion**.

Make the Interface Consistent

- The interface should **present** and acquire **information** in a **consistent fashion**.
- Following are the design principles that help make the interface consistent:
  - Maintain **consistency across a family of applications**.
  - If past interactive models have created user expectations, **do not make changes unless there is a compelling** (convincing) **reason** to do so.

Microsoft Word



Microsoft Excel

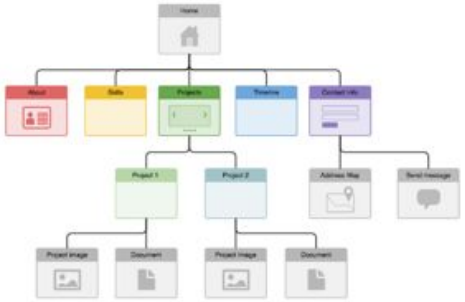
Microsoft Powerpoint

Web Application Design

Web Application Design

- Design for WebApp encompasses technical and nontechnical activities that include:
  - **Establishing the look and feel** of the WebApp
  - Defining the **overall architectural structure**
  - **Developing** the **content** and **functionality** that reside within the architecture
  - **Planning** the **navigation** that occurs within the WebApp



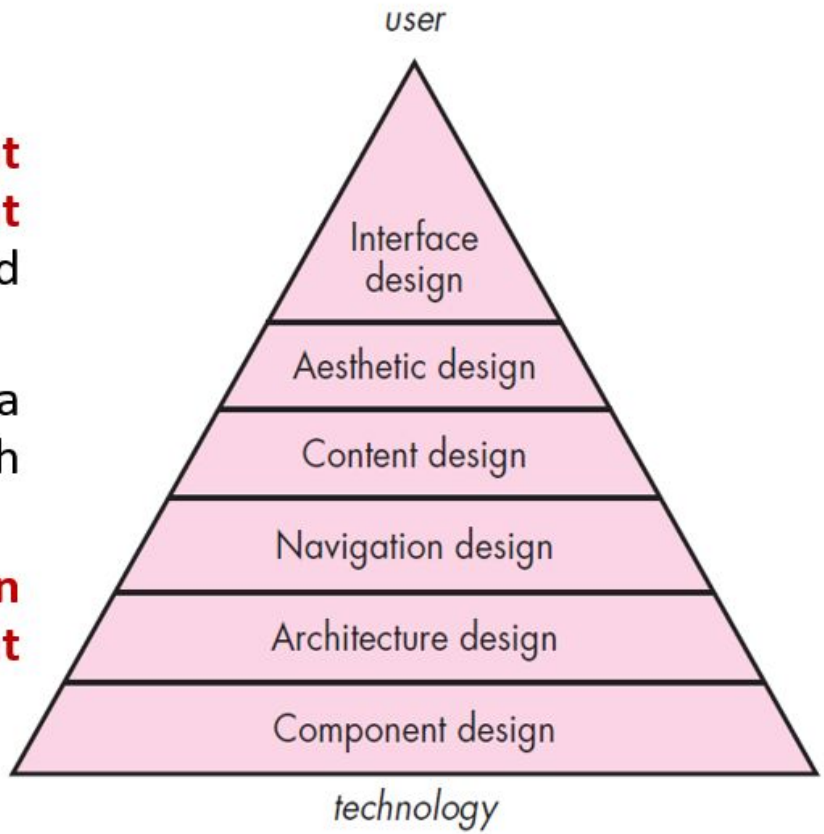



Web App Interface Design

The objectives of a WebApp interface are to:

- Establish a **consistent window** into the **content** and **functionality** provided by the interface.
- **Guide the user** through a series of interactions with the WebApp.
- **Organize** the **navigation options** and **content** available to the user.

Design pyramid for WebApps



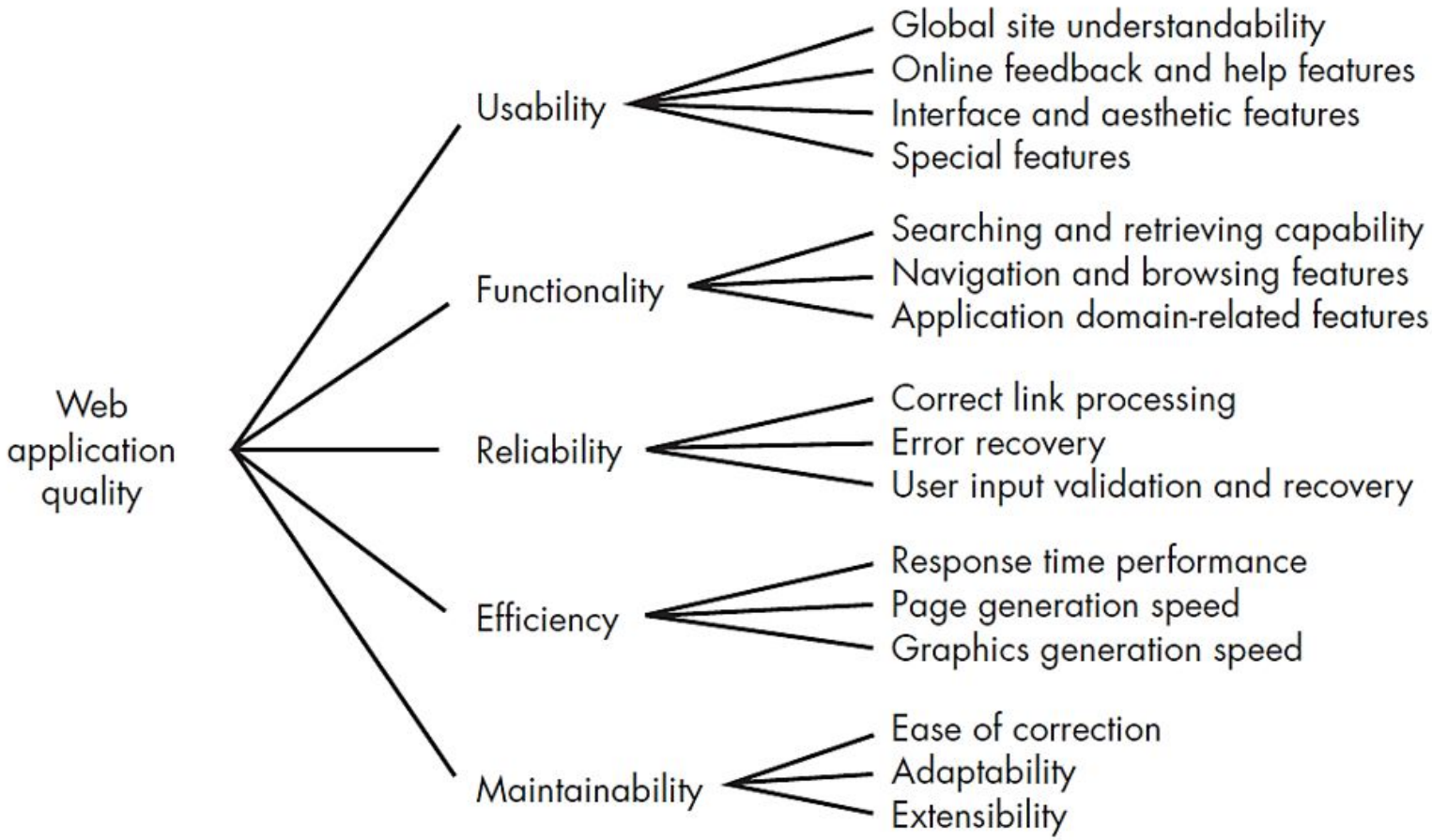
Design pyramid for WebApps Cont.

- **Interface Design**
  - One of the challenges of interface design for WebApps is the **nature of the user's entry point**.
- **Aesthetic Design**
  - Also called **graphic design**, is an **artistic endeavor** (offer) that **complements the technical aspects** of WebApp design.
- **Content Design**
  - **Generate content and design** the representation for content to be used within a WebApp.
- **Architecture Design**
  - It is tied to the goals established for a WebApp,
  - the content to be presented, the users who will visit, and the navigation that has been established.

Design pyramid for WebApps Cont.

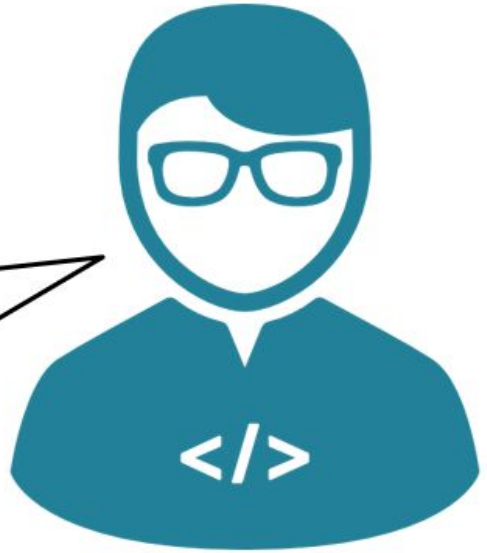
- **Navigation Design**
  - **Define navigation pathways** that enable users to access WebApp content and functions.
- **Component-Level Design**
  - Modern WebApps deliver increasingly sophisticated processing functions that,
    - **Perform localized processing** to generate content and navigation 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.
    - Establish **data interfaces with external corporate** systems.

WebApp Design Quality Requirement



Component Level Design

Effective programmers should not waste their **time** in **debugging**, they should **not introduce bug** to start with



Component-Level Design or Procedural Design

- **Component** is a modular, deployable and replaceable part of a system that **encapsulates** implementation and exposes a set of interfaces.
- Component-level design **occurs after data, architectural and interface designs** have been established.
- It **defines** the **data structures, algorithms, interface characteristics,** and **communication mechanisms** allocated to each component.
- The intent is to **translate** the **design model** into **operational software**.
- But the abstraction level of the existing design model is relatively high and the abstraction level of the operational program is low.

Function Oriented Approach

- The following are the features of a typical function-oriented design approach:
  1. A system is **viewed** as something that **performs a set of functions**.
     - Starting at this high level view of the system, **each function** is successively **refined into more detailed functions**.
     - For example, consider a **function create-new-library member**
       - which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge.
       - This function may consist of the following **sub-functions**:
         - » **assign-membership-number, create-member-record and print-bill**
     - Each of these sub-functions may be split into more detailed sub-functions and so on.

Function Oriented Approach Cont..

2. The system **state is centralized** and **shared among different functions**.
   - For Ex., data such as **member-records** is available for reference and updating to several functions such as:
     - create-new-member
     - delete-member
     - update-member-record

Object Oriented Approach

- In the object-oriented design approach, the **system is viewed** as **collection of objects** (i.e., entities).
- The **state is decentralized** among the objects and **each object manages its own state** information.
- For example, in a Library Automation Software,
  - each library member may be a separate object with its own data and functions to operate on these data.
  - In fact, the functions defined for one object cannot refer or change data of other objects.
- **Objects** have their **own internal data** which define their state.

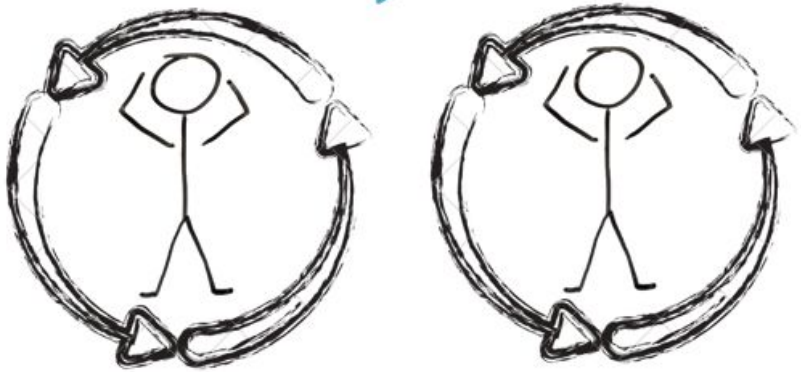
Cohesion & Coupling

A **good software design** implies **clean decomposition** of the **problem into modules**, and the **neat arrangement** of these **modules** in a **hierarchy**.

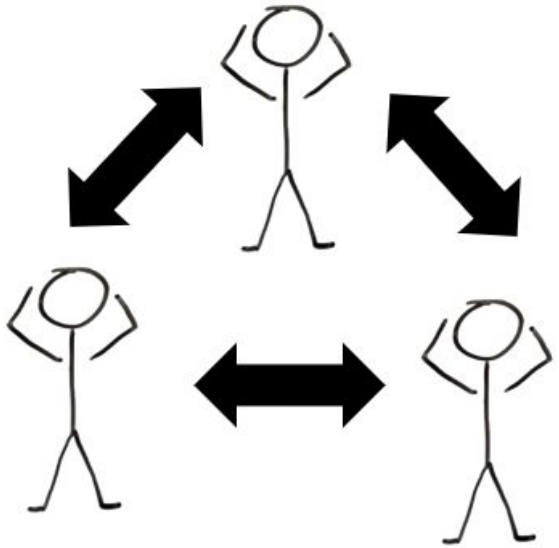
The primary **characteristics** of **neat module decomposition** are **high cohesion** and **low coupling**.

Cohesion & Coupling

A **cohesive module** performs a single task, requiring little interaction with other components.



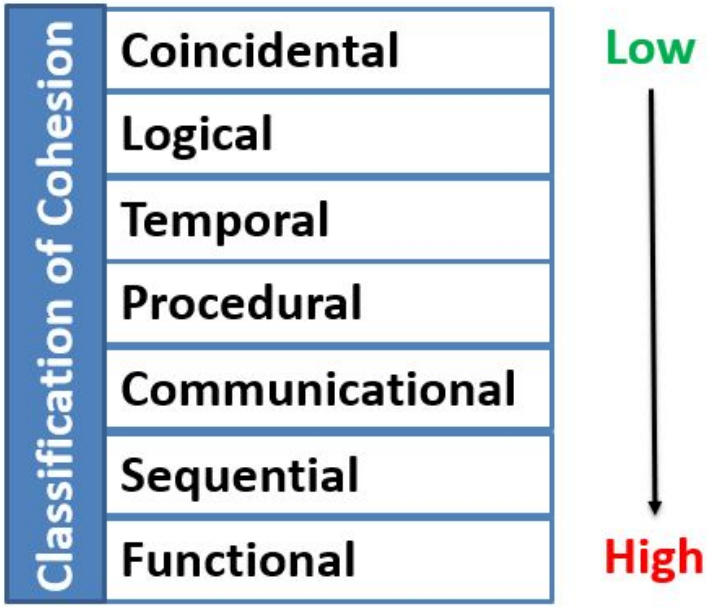
A **Coupling** is an indication of the relative interdependence among modules.



Cohesion

- 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.
- Stated simply, a **cohesive module** should (ideally) **do just one thing**.
- A module having **high cohesion** and **low coupling** is said to be **functionally independent** of other modules.
- By the term functional independence, we mean that a **cohesive module performs a single task or function**.

Classification of Cohesion



Classification of Cohesion cont.

Coincidental cohesion

- A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely.
- In this case, the module contains a random collection of functions.
- It is likely that the functions have been put in the module out of pure coincidence without any thought or design.
- For Ex., in a transaction processing system (TPS), the get-input, print-error, and summarize-members functions are grouped into one module.

Classification of Cohesion cont.

Logical cohesion

- A module is said to be logically cohesive, if all elements of the module perform similar operations.
- For Ex., error handling, data input, data output, etc.
- An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.

Temporal cohesion

- When a module contains functions that are related by the fact that all the functions must be executed in the same time span.
- For Ex., the set of functions responsible for initialization, start-up, shutdown of some process, etc.

Classification of Cohesion cont.

Procedural cohesion

- If the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective
- For Ex., the algorithm for decoding a message.

Communicational cohesion

- If **all functions** of the module **refer** to the **same data structure**
- For Ex., the set of functions defined on an array or a stack.

Classification of Cohesion cont.

Sequential cohesion

- If the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next.
- For Ex., In a Transaction Processing System, the get-input, validate-input, sort-input functions are grouped into one module.

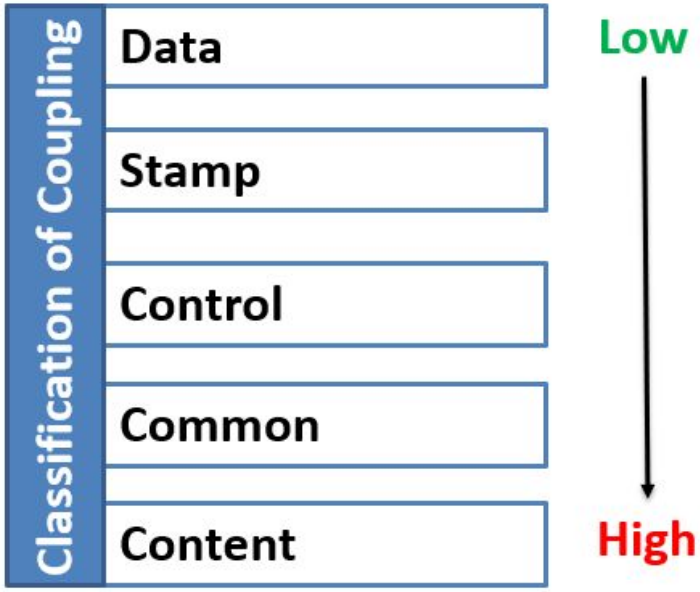
Functional cohesion

- If different elements of a module cooperate to achieve a single function.
- For Ex., A module containing all the functions required to manage employees' pay-roll exhibits functional cohesion.

Coupling

- **Coupling** between two modules is a **measure of the degree** of **interdependence** or interaction **between** the **two modules**.
- A module having high cohesion and low coupling is said to be functionally independent of other modules.
- If **two modules interchange large amounts of data**, then they are **highly interdependent**.
- The degree of coupling between two modules depends on their interface complexity.
- The interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.

Classification of Coupling



Classification of Coupling Cont.

Data coupling

- Two modules are data coupled, if **they communicate through a parameter**.
- An example is an elementary (primal) data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc.

Stamp coupling

- This is a special case (or extension) of data coupling
- Two modules (``A'' and ``B'') exhibit stamp coupling if **one passes** directly to the other a **composite data item** - such as a record (or structure), array, or (pointer to) a list or tree.
- This occurs when **ClassB** is **declared as a type** for an argument of an **operation of ClassA**

Classification of Coupling Cont.

Control coupling

- If **data** from **one** module **is used to direct the order of instructions** execution **in another**.
- An example of control coupling is a flag set in one module and tested in another module.

Common coupling

- Two modules are common coupled, if they **share data** through some **global data items**.
- Common coupling can **leads to uncontrolled error propagation** and **unforeseen side effects** when changes are made.

Classification of Coupling Cont.

Content coupling

- Content coupling occurs when **one component secretly modifies data** that is **internal to another component.**
- This violets information hiding – a basic design concept
- Content coupling exists between two modules, if they share code.

Summary

- Design Concepts
- Design principles
- Architectural Design
  - Architectural Styles
    1. Data-centered architecture style
    2. Data-flow architectures
    3. Call and return architecture
    4. Object-oriented architecture
    5. Layered architecture
- Component-Level Design
  1. Function Oriented Approach
  2. Object Oriented Approach
- Cohesion and Coupling
- User Interface Design
  - Design Rules for User Interface
- Web Application Design