Use Case Name: SearchSeat

Priority: Normal

Actors: Passengers

Summary: This use case enables passenger to search flight as per his/her convenience

Precondition: flights exist in database

Post-Condition: Flight schedule is displayed for the said date(s)

Extends: none

Uses: SearchFlight

Normal Course of Events:

Normal Course of Events:

	User	System
1	On home page passenger selects "search flight" link	
2		System displays the input screen for entering the date
3	User enters the date(s) and clicks "display" button	
4		System displays all the flights (with available seats) scheduled on the said dates
5	Passengers double clicks one flight.	
6		System displays the number of available seats for the selected flight
7	Passenger selects seat(s) and clicks the "print button	
8		System prints the flight no. along with selected seats.

Alternative Path:
At step (3) user does not click the display button but clicks the cancel button, system does not display the flight schedule, but goes to home page
At step (5), user does not double click flight, but clicks the cancel button, system does not display the flight schedule, but goes to home page
At step (7), user does not click pint button but clicks "go back" button and system goes to home page
Exception:
Server disconnected, system displays "Information Not available at the Time" message.
Assumptions: none

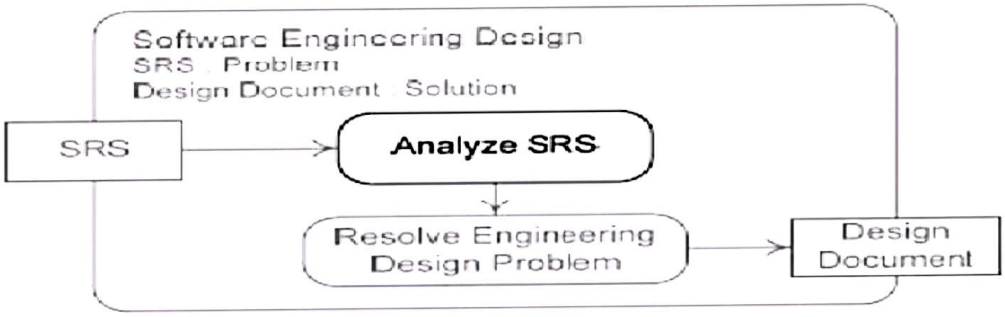
Week 11 Summery

Introduction to Engineering Design Analysis

Engineering Design Analysis

- Engineering design analysis activities consist mainly of studying the SRS and product design models and producing new models of the problem.
- It is also considering the inconsistencies and incompleteness in the SRS often come to light. If so, engineering designers must ask product designers for clarification or elaboration.
- This may lead product designers to redo part of the design, which may in turn lead to discussion and consultation with stakeholders.
- Analyzing the SRS and product design models can improve both their quality and the quality of the product design itself, besides laying the foundation for engineering design resolution.

Analysis Goals, Inputs, and Activities



1. Understand an engineering design problem using
    1. SRS
    2. Product design models
2. Achieve understanding by
    1. Studying the SRS and design models
    2. Making analysis models

Analysis Models

- An analysis model is any representation of a design problem.
- Both static and dynamic models
- Object-oriented and other kinds of models

Class and Object Models

- A **class (object) model** is a representation of classes (objects) in a problem or a software solution.
- Class (object) diagrams are graphical forms of class (object) models.

Types of Class Models

- *Analysis or conceptual models*—Important entities or concepts in the problem, their attributes, important relationships
- *Design class models*—Classes in a software system, attributes, operations, associations, but no implementation details
- *Implementation class models*—Classes in a software system with implementation details
- Analysis models represent the *problem*; design and implementation models represent the *solution*.

Classes and Objects

- An **object** is an entity that holds data and exhibits behavior.
- A **class** is an abstraction of a set of objects with common operations and attributes.
- An **attribute** is a data item held by an object or class.
- An **operation** is an object or class behavior.
- An **association** is a connection between classes representing a relation on the sets of instances of the connected classes.

Summary

- Engineering design begins with analysis of the SRS and product design models.
- Analysis modeling helps designers understand the design problem.
- Class models include analysis (conceptual), design, and implementation class models.

Conceptual Modeling

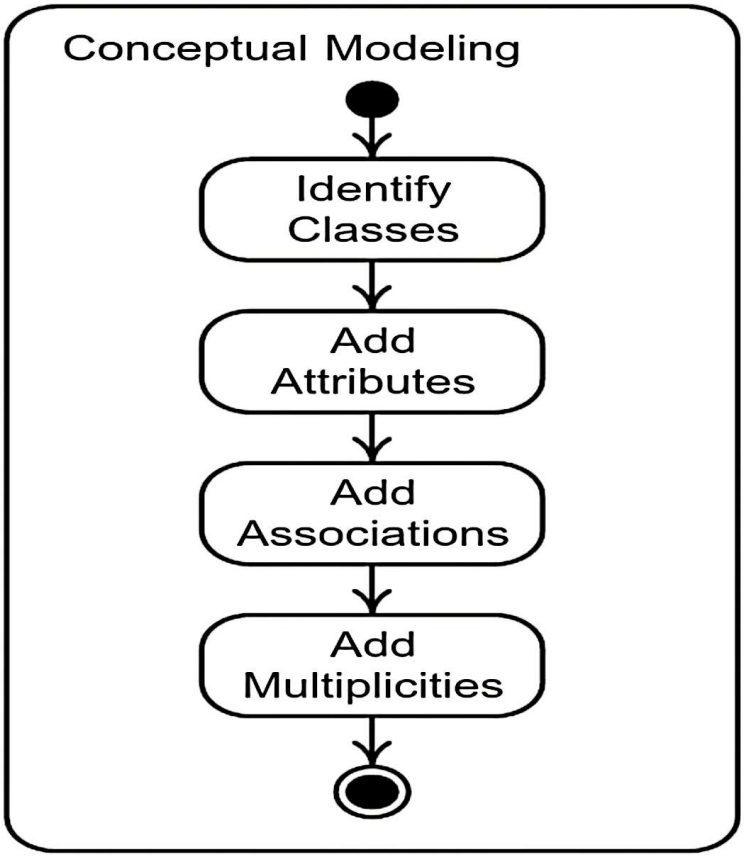
Conceptual models

- A conceptual model is a static model of the important entities in a problem, their responsibilities or attributes, the important relationships among them and perhaps their behaviors.
- Conceptual models are about real-world entities in the problem domain and not about software.

Uses of Conceptual Models

- In Product design
  - Understanding the problem domain
  - Setting data requirements
  - Validating requirements
- In Engineering design
  - Understanding a product design
  - Providing a basis for engineering design modeling

Conceptual Modeling Process



Identifying Classes-Brainstorming

- Study the product design (SRS, use case models, other models)
- Look for nouns and noun phrases for
  - Physical entities
  - Individuals, roles, groups, organizations
  - Real things managed, tracked, recorded, or represented in the product
  - People, devices, or systems that interact with the product (actors)

Identifying Classes-Rationalizing

- Remove noun phrases designating properties (they may be attributes)
- Remove noun phrases designating behaviors (they may be operations)
- Combine different names for the same thing

- Remove entities that do not directly interact with the product
- Clarify vague nouns or noun phrases
- Remove irrelevant or implementation entities

Adding Attributes 1

- Study the SRS and product design models looking for adjectives and other modifiers.
- Use names from the problem domain.
- Include only those types, multiplicities, and initial values specified in the problem.

Adding Attributes 2

- Don't add object identifiers unless they are important in the problem
- Don't add implementation attributes
- Add operations sparingly

Adding Associations Brainstorming

- Study the SRS and product design models looking for verbs and prepositions describing relationships between model entities
- Look for relationships such as
  - Physical or organizational proximity;
  - Control, coordination, or influence;
  - Creation, destruction, or modification;
  - Communication; and
  - Ownership or containment.

Adding Associations—Rationalizing 1

- Limit the number of associations to at most one between any pair of classes
- Combine different names for the same association
- Break associations among three or more classes into binary associations

Adding Associations—Rationalizing 2

- Make association names descriptive and precise.
- Add rolenames where they are needed

Adding Multiplicities

- Take pairs of associated entities in turn.
  - Make one class the target, the other the source.
  - Determine how many instances of the target class can be related to a single instance of the source class.
  - Reverse the target and source and determine the other multiplicity.
- Consult the product design.
- Add only multiplicities important in the problem.

Summary

- A conceptual model represents the important entities in a design problem along with their properties and relationships.
- Conceptual models represent the design *problem*, not the software *solution*.
- Conceptual models are useful throughout product design and in engineering design analysis.
- There is a process for conceptual modeling.
- Process steps can be done by analyzing the text of product design artifacts.
- Several heuristics guide designers in conceptual modeling.

Week 12 Summery

UML Class Diagram

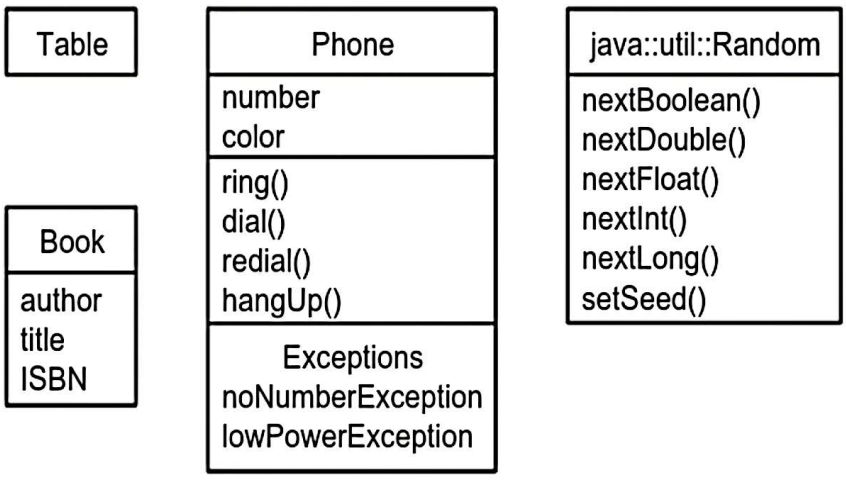
Classes and Objects

- An **object** is an entity that holds data and exhibits behavior.
- A **class** is an abstraction of a set of objects with common operations and attributes.
- An **attribute** is a data item held by an object or class.
- An **operation** is an object or class behavior.
- An **association** is a connection between classes representing a relation on the sets of instances of the connected classes.

UML Names

- A **name** in UML is character string that identifies a model element.
  - Simple name: sequence of letters, digits, or punctuation characters
  - Composite name: sequence of simple names separated by the double colon (::)
- **Examples**
  - Java::util::Vector
  - veryLongNameWithoutPunctuationCharacters
  - short_name

UML Class Symbol



- Compartments
    1. Class name
    2. Attributes
    3. Operations
    4. Other compartments
- Compartment order
- Suppressing compartments
- Class name compartment must contain a name (simple or composite)

Attribute Specification Format

name : *type* [*multiplicity*] = *initial-value*

- *name*—simple name, cannot be suppressed
- *type*—any string, may be suppressed with the :
- *multiplicity*—number of values stored in attribute
    - list of ranges of the form *n*..*k*, such that *n* <=*k*
    - *k* may be *
    - *n*..*n* is the same as *n*
    - 0..* is the same as *
    - 1 by default
    - if suppressed, square brackets are omitted
- *initial-value*—any string, may be suppressed along with the =

Operation Specification Format

- *name*(*parameter-list*) : *return-type-list*
- *name*—simple name, cannot be suppressed
- *parameter-list*
    - *direction param-name* : *param-type* = *default-value*
    - *direction*—in, out, inout, return; in when suppressed
    - *param-name*—simple name; cannot be suppressed

- *param-type*—any string; cannot be suppressed
- *default-value*—any string; if suppressed, so is =

- *return-type-list*—any comma-separated list of strings; if omitted (with :) indicates no return value
- The *parameter-list* and *return-type-list* may be suppressed together.

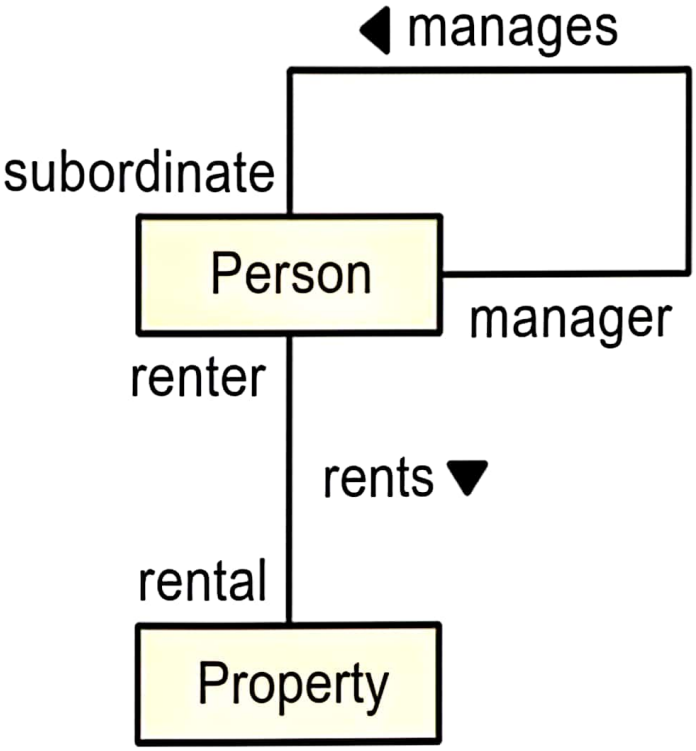
Attribute and Operation Examples

Player
roundScore : int = 0 totalScore : int = 0 words : String[*] = ()
resetScores() setRoundScore(in size : int) findWords(in board : Board) getRoundScore() : int getTotalScore() : int getWords() : String[*]

WaterHeaterController
mode : HeaterMode = OFF occupiedTemp : int = 70 emptyTemp : int = 55
setMode(newMode : Mode = OFF) setOccupiedTemp(newTemp : int) setEmptyTemp(newTemp : int) clockTick(out ack : Boolean)

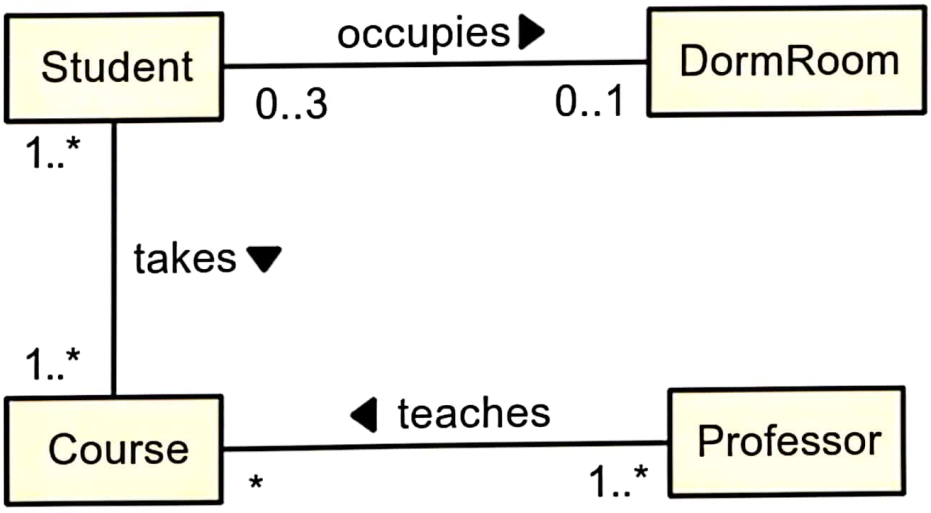
Association Lines

- Labeled or unlabeled lines
- Readable in two directions
- Direction arrows
- Rolenames



Association Multiplicity

- The multiplicity at the target class end of an association is the number of instances of the target class that can be associated with a single instance of the source class.



Class Diagram Rules

- Class symbols must have a name compartment.
- Compartments must be in order.
- Attributes and operations must be listed one per line.
- Attribute and operation specifications must be syntactically correct.

Class Diagram Heuristics 1

- Name classes, attributes, and roles with noun phrases.
- Name operations and associations with verb phrases.
- Capitalize class names only.
- Center class and compartment names but left-justify other compartment contents.

Class Diagram Heuristics 2

- Stick to binary associations.
- Prefer association names to rolenames.
- Place association names, rolenames and multiplicities on opposite sides of the line

Class Diagram Uses

- Central static modeling tool in object-oriented design
    - Conceptual models
    - Design class diagrams
    - Implementation class diagrams
- Can be used throughout both the product and engineering design processes
- UML class diagrams can be used for all types of class models, and throughout the design process.

UML Object Diagram

Object Diagrams

- Object diagrams are used much less often than class diagrams
- Object symbols have only two compartments:
  - Object name
  - Attributes (may be suppressed)

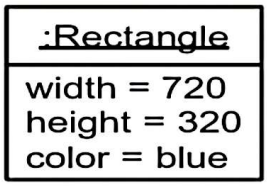
Object Name Format

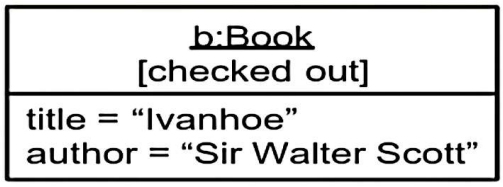
- *object-name : class-name*
  [*stateList*]
- *object-name*—simple name
- *class-name*—a name (simple or composite)
- *stateList*—list of strings; if suppressed, the square brackets are omitted
- The *object-name* and *class-name* may both be suppressed, but not simultaneously

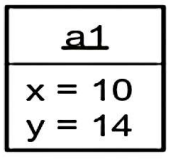
Object Attribute Format

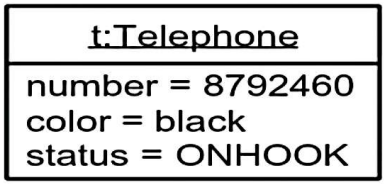
- *attribute-name = value*
- *attribute-name*—simple name
- *value*—any string
- Any attribute and its current value may be suppressed together

Examples of Object Symbols









Object Links

- Show that particular objects participate in a relation between sets of objects
- Instances of associations
- Shown using a *link line*
  - Solid line (no arrowheads)
  - Underlined association name
- Link lines *never* have multiplicities

Object Diagram Uses

- Show the state of one or more objects at a moment during execution
- Dynamic models as opposed to class diagrams, which are static models
- UML object diagrams represent the state of objects during execution.

Week 13 Summery

Introduction to Architectural Design

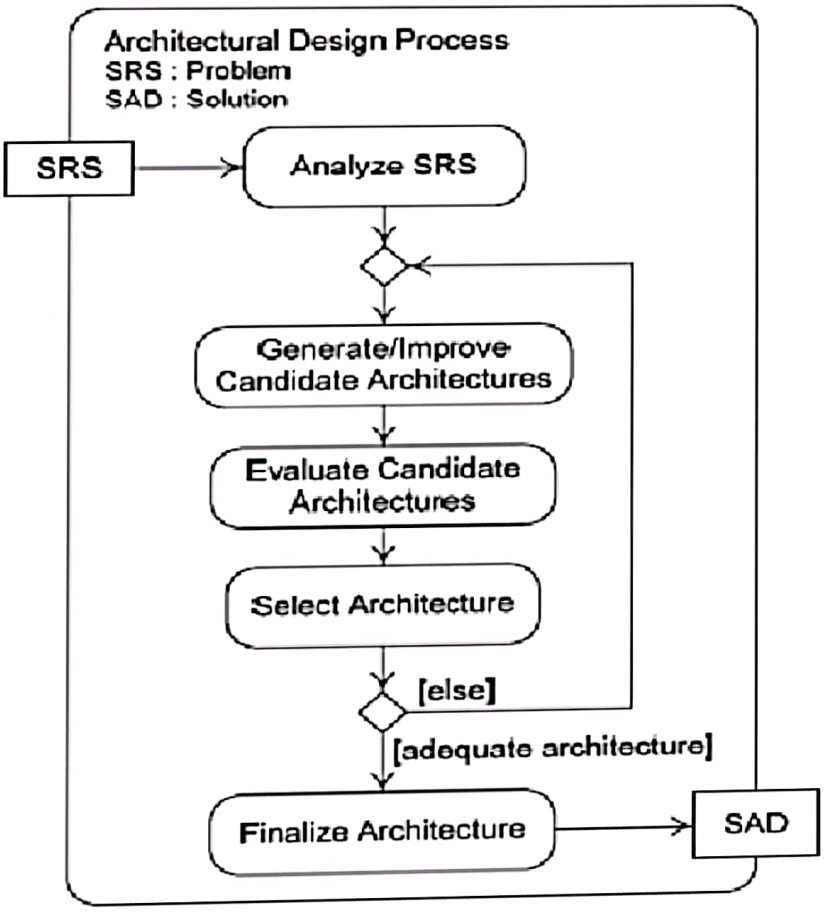
- Engineering design resolution has 2 phases:"architectural design" and "detailed design".
- Architectural design is a problem-solving activity whose input is the product description in an SRS and whose output is the abstract specification of a program realizing the desired product. Architectural design sits between software product design and detailed design.

Some architectural design occurs during product design for following reasons:

- Product designers must judge the feasibility of their designs, which may be difficult without some initial engineering design work.
- Stakeholders must be convinced that their needs will be met, which may be difficult without demonstrating how the engineers plan to build the product.
- Tradeoffs of feasible product, schedule and budget will not be clear without architecture.
- In a very small program consisting of only a handful of classes or modules interacting in simple ways, the software architecture is hardly distinguishable from the detailed design, so it is appropriate for the architecture to be simple and quite abstract. However, a very large and complicated system with hundred or thousand of classes, distributed over many machines , interacting with many peripheral devices , and with demanding non-functional requirements, demands a detailed high level specification that is carefully worked out and analyzed.

Architectural Design Process

- The architectural design process is a straight forward application of the generic design process to the problem of architectural design.



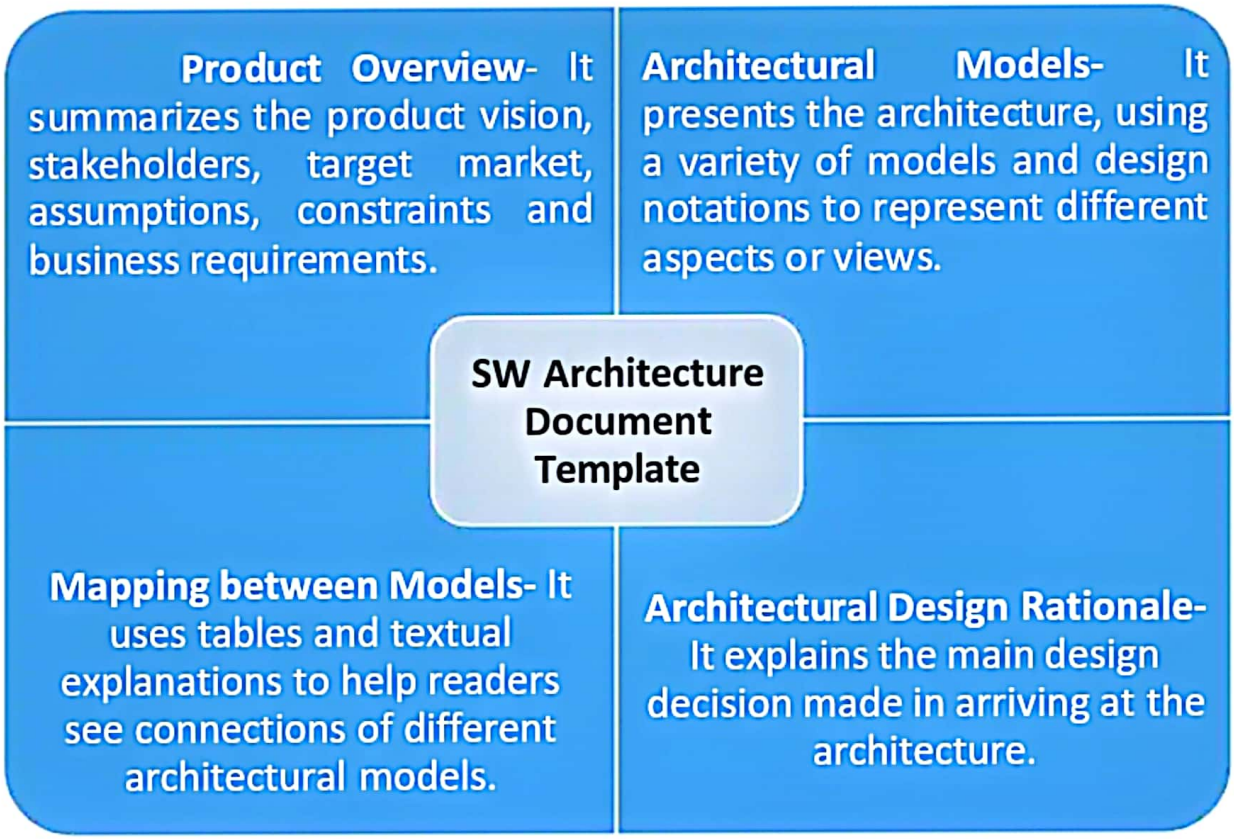
A **software architecture document(SAD)** is simply a document that specifies the architecture of a software system,

Software Architecture Document Template

1. Product Overview
2. Architectural Models
3. Mapping between Models
4. Architecture Design Rationale

Figure 9.2 – Software Architecture Document Template

The above template is appropriate for documenting the software architectures of small-to-medium sized systems.



Quality Attributes

- SW architecture is crucial not only for satisfying a product's functional requirements but also for satisfying its non-functional requirements. These are also called quality attributes.

A **quality attributes** is a characteristics or property of a software product independent of its function that is important in satisfying stakeholder needs and desires.

Types of Quality Attributes

- Quality attributes fall into two categories: **development attributes** and **operational attributes.**
- Development attributes include properties important to development organization stake holders, such as maintainability and reusability.
- Operational quality attributes include properties like performance, availability, reliability and security.

- **Example:** Consider a program responsible for matching fingerprints read from scanners against a database to allow people into and out of a secure facility.

Besides its functional requirements, this program has some obvious non-functional requirements. For example, it must respond quickly, it must be available the entire time people are entering or leaving the facility, it must match finger prints fairly reliably, and it must resist attackers.

Introduction to Detailed Design

Detailed Design

- **Detailed design** is the activity of specifying the internal elements of all major program parts; their structure, relationships, and processing; and often their algorithms and data structures.
- Every program has a software architecture, but its level of abstraction is highly variable, depending on program size. Very large programs have many large sub-systems described during architectural design at high level of abstraction. Small programs have a few small components described in some detail during architectural design.

- During detailed design, designers specify class responsibilities, class attributes, class operations, object interactions, object states, state changes, processes, and process synchronization.
- Programmers choose control structures, program entity names, primitive types, parameter passing mechanisms, and programming idioms. But either party can make a variety of low-level design involving packaging visibility, algorithms, and data structure.

The Scope of Detailed Design

- Detailed design fills in the design specifications left open after architectural design. The main goal of detailed design is to specify the program in sufficient detail so that programmers can implement it.

- At the highest levels of abstraction, detailed design may be like architectural design in specifying the main parts of major sub-systems, including their states and transitions, collaborations, responsibilities, interfaces, properties, and relationships with other components. At the lowest levels of abstraction, detailed design may specify implementation details down to pseudo code and data formats. Detailed design representations include static models, such as class diagrams, operation specifications, and data structure diagrams, as well as dynamic models, such as interaction diagrams, state diagrams, and pseudo code.

Stages of Detailed Design

Because of its size, complexity, and range of abstraction and representation, it is helpful to further divide detailed design into two stages:

- ❑ Mid-level design
- ❑ Low-level design

Mid-Level Design

> **Mid-level design** is the activity of specifying software at the level of medium-sized components, such as compilation units or classes, and their properties, relationships, and interactions.

Mid-level design must specify both static and dynamic aspects of components. Specifications include the DeSCRIPTR aspects i.e,

Decomposition—Mid-level components are the parts comprising architectural components. These parts must be identified.

States and State Transitions—Some components have important states and state-based behavior. Designers must specify the states and state transitions of such components.

Collaborations—Designers must specify the dynamic flow of control and data among mid-level components enabling them to solve problems collaboratively. Component interactions include object message passing behavior, calling relationships between operations, data flow, and processes and process synchronization.

Responsibilities—Designers must specify mid-level component obligations to carry out tasks or maintain data.

- **Interfaces**—Designers must describe the communication mechanisms mid-level components use when interacting. This includes interface syntax, semantics, and pragmatics.
- **Properties**—Designers must state important mid-level component properties, usually having to do with quality attributes, such as security, reliability, and modifiability.
- **Relationships**—Some component relationships are established during architectural design, but many are not. The most important of these established during mid-level design are inheritance relationships, interface realization and dependency relationships, and visibility and accessibility associations.

Low level Design

> **Low-level design** is the activity of filling in small details at the lowest levels of abstraction.

Low-level design involves issues beyond DeSCRIPTR specifications involving coding and programming

Languages:

- **Packaging**—Decisions must be made and documented about how to bundle code into various compilation units, libraries, packages, and so forth.
- **Algorithms**—Certain algorithms may be chosen depending on consideration of time and space efficiency, ease of implementation and maintenance, and programming language support. Sometimes designers may specify the processing steps of individual operations.
- **Implementation**—Designers must resolve implementation issues, notably

the visibility and accessibility of various entities, and how to realize associations.

- **Data Structures and Types**—Designers may decide how to store data to meet product requirements, which involves selecting data structures and

choosing variable data types.

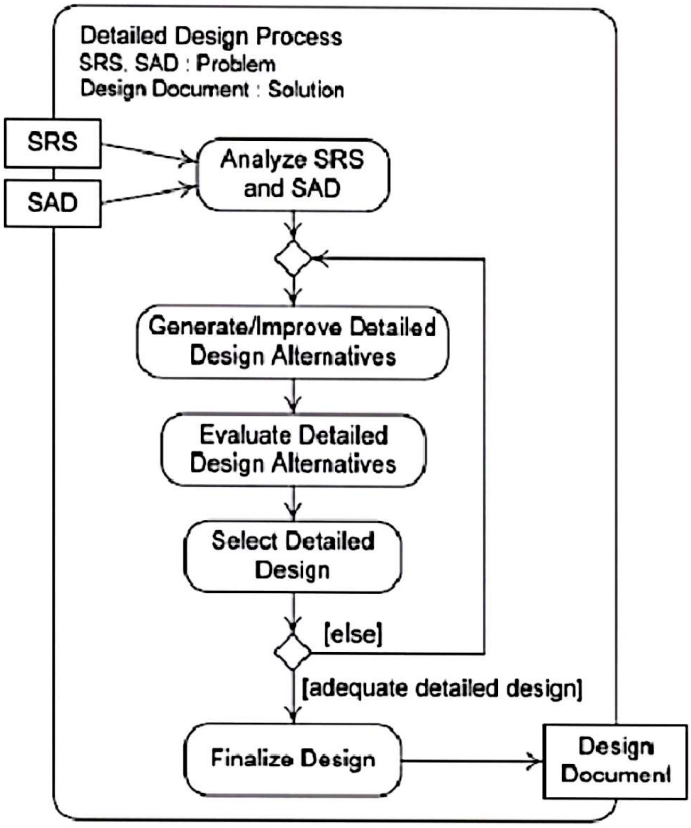


Fig 13.2.1: Detailed Design Process

A **design document** is a complete engineering design specification. It has two parts : a software architecture document(SAD) and a **detailed design document (DDD).**

Week 14 Summery

Design Patterns

- Design patterns are recurring solutions to object-oriented design problems in a particular context.
- Patterns describe recurring data-oriented problems and the data modeling solutions that can be used to solve them.

Patterns

- Documenting patterns is one way that you can reuse and possibly share the information that you have learned about how it is best to solve a specific program design problem.
- Design patterns are experience captured in a well-structured and consistent format; they provide blueprints that guide designers to solve specific problems by specifying important design characteristics, such as the classes that need to be created, their level of granularity, their relationships, and how all these classes and relationships work together to solve a problem. They provide this information in a generic sense, so that they can be reused many times over, in different software systems, without ever doing it the same way twice.

Benefits of Patterns

- When used effectively, they can help improve efficiency in the detailed design effort by providing high-quality reusable solutions that can be applied in many practical applications.
- A design pattern saves you from "reinventing the wheel," or worse, inventing a "new wheel" that is slightly out of round, too small for its intended use, and too narrow for the ground it will roll over. Design patterns, if used effectively, will invariably make you a better software designer
- It solves a problem: Patterns capture solutions, not just abstract principles or strategies. • It is a proven concept: Patterns capture solutions with a track record, not theories or speculation.
- There are many benefits from studying and applying design patterns. First, they can help designers and programmers become more efficient. It is now common to find built-in

- support for design patterns in today's popular language frameworks, such as Java and the
- .NET framework. Therefore, knowing about design patterns can help programmers come
- up to speed quicker in these environments and enable them to quickly apply them to particular
- problems.

GOF Design Pattern Format

The basic template includes ten things as described below

- **Name**

· Works as idiom

Name has to be meaningful

- **Problem**
- A statement of the problem which describes its intent
- The goals and objectives it wants to reach within the given context
- **Context**
- Preconditions under which the problem and its solutions seem to occur
- Result or consequence
- State or configuration after the pattern has been applied
- **Forces**

· Relevant forces and constraints and their interactions and conflicts.

· motivational scenario for the pattern.

GOF Design Pattern

- **Solution**

· Static and dynamic relationships describing how to realize the pattern.

· Instructions on how to construct the work products.

· Pictures, diagrams, prose which highlight the pattern's structure, participants, and collaborations.

- **Examples**
  - One or more sample applications to illustrate
  - a specific context
  - how the pattern is applied
- **Resulting context**

· the state or configuration after the pattern has been applied

· consequences (good and bad) of applying the pattern

- **Rationale**

· justification of the steps or rules in the pattern

  - how and why it resolves the forces to achieve the desired goals, principles, and philosophies

· how does the pattern actually work

- **Related patterns**
  - the static and dynamic relationships between this pattern and other patterns
- **Known uses**
  - to demonstrate that this is a proven solution to a recurring problem

Classification of Design Patterns

Design Patterns

When first studying design patterns, it is important to understand what each pattern

does and how it does it.

In the influential work presented by Gamma, Helm, Johnson, and Vissides (1995) design patterns are classified based on purpose and scope. The purpose of a design pattern identifies the functional essence of the pattern; therefore, it serves as fundamental differentiation criteria between design patterns.

GOF(Gang of Four) Classification

Three different purposes are identified by the Gang of Four (GoF):

- **Creational**
- **Structural**
- **Behavioral**

Creational Pattern

Creational design patterns are the ones that attempt to efficiently manage the creation process of objects in a software system.

The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation. Creational patterns are overall known for abstracting the instantiation process of one or more objects.

Creational Patterns Types

- Abstract Factory
  Creates an instance of several families of classes
- Builder
  Separates object construction from its representation
- Factory Method
  Creates an instance of several derived classes
- Object Pool
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- Prototype
  A fully initialized instance to be copied or cloned
- Singleton
  A class of which only a single instance can exist

Abstract Pattern : Creational

- The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes
- The "factory" object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them.
- https://sourcemaking.com/design_patterns/abstract_factory

Structural Design Patterns

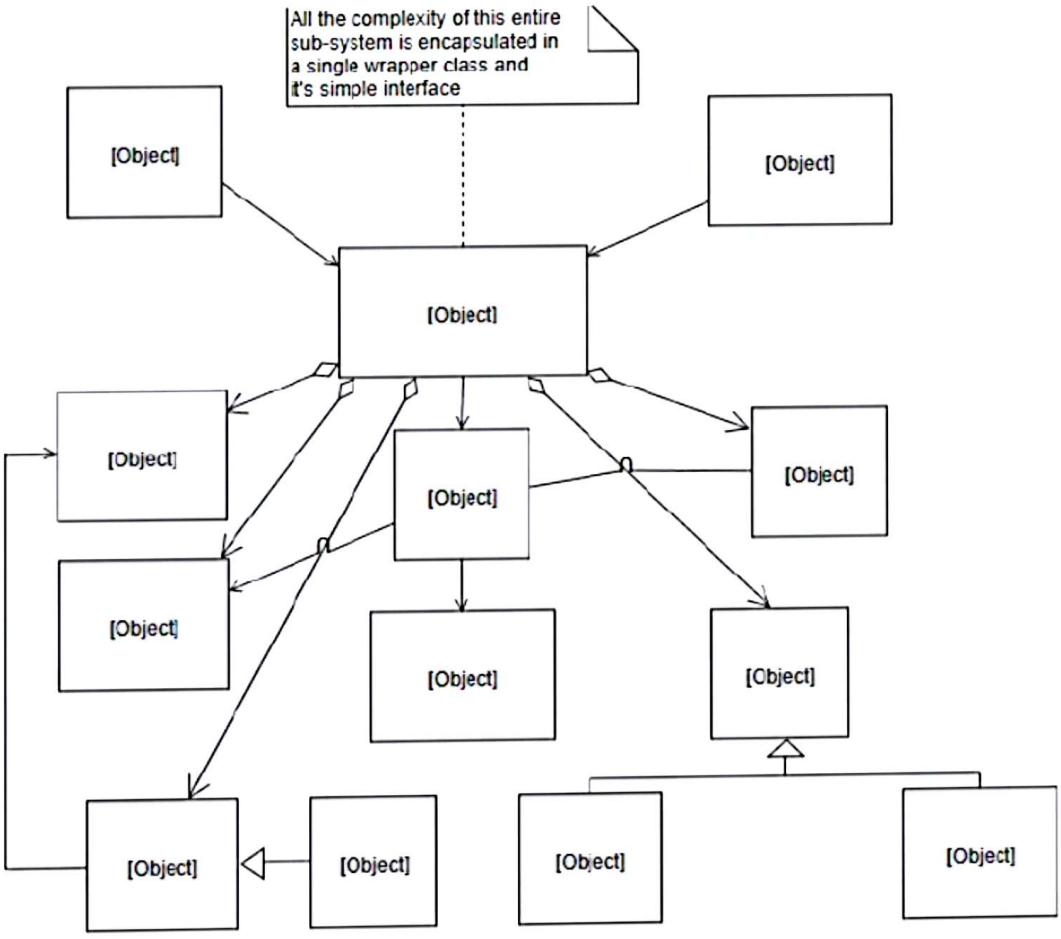
- Structural Design Patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.
- These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

Structural Pattern Types

- Adapter
  Match interfaces of different classes
- Bridge
  Separates an object's interface from its implementation
- Composite
  A tree structure of simple and composite objects
- Decorator
  Add responsibilities to objects dynamically
- Facade
  A single class that represents an entire subsystem
- Flyweight
  A fine-grained instance used for efficient sharing
- Private Class Data
  Restricts accessor/mutator access
- Proxy
  An object representing another object

Façade Pattern: Structural

- The Facade defines a unified, higher level interface to a subsystem that makes it easier to use. Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department.
- Facade discusses encapsulating a complex subsystem within a single interface object. This reduces the learning curve necessary to successfully control the subsystem
- Wrap a complicated subsystem with a simpler interface.
- https://sourcemaking.com/design_patterns/facade



Behavioral Design Patterns

- These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.
- They identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

Behavioral Design Pattern Types

- Chain of responsibility
  A way of passing a request between a chain of objects
- Command
  Encapsulate a command request as an object
- Interpreter
  A way to include language elements in a program
- Iterator
  Sequentially access the elements of a collection
- Mediator
  Defines simplified communication between classes
- Memento
  Capture and restore an object's internal state
- Null Object
  Designed to act as a default value of an object
- Observer
  A way of notifying change to a number of classes
- State
  Alter an object's behavior when its state changes
- Strategy
  Encapsulates an algorithm inside a class
- Template method
  Defer the exact steps of an algorithm to a subclass
- Visitor
  Defines a new operation to a class without change

Observer Pattern: Behavioral

- The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically.

Example:

Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price which is broadcast to all of the bidders in the form of a new bid.

- https://sourcemaking.com/design_patterns/observer

Week 15 Summery

Architectural Styles:
Layered Architectures

Software Architecture

- A software architecture is the structure of a program comprised by its major constituents, their responsibilities and properties, and the relationships and interactions between them.

Architectural Styles

- An **architectural style** is a paradigm of program or system constituent types and their interactions.
- To present several important architectural styles, including
- Layered style
- Pipe-and-Filter style
- Shared-Data style
- Event-Driven style
- Model-View-Controller style
- Hybrid architectures

Layered Style Architectures

- The program is partitioned into an array of layers or groups.
- Layers use the services of the layer or layers below and provide services to the layer or layers above.
- The Layered style is among the most widely used of all architectural styles.

Uses and Invokes

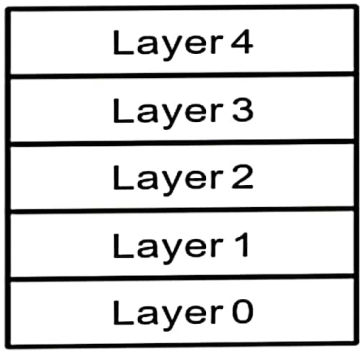
- Module A **uses** module B if a correct version of B must be present for A to execute correctly.
- Module A **calls** or **invokes** module B if A triggers execution of B.
- Note that
    - A module may use but not invoke another

- A module may invoke but not use another
- A module may both use and invoke another
- A module may neither use nor invoke another

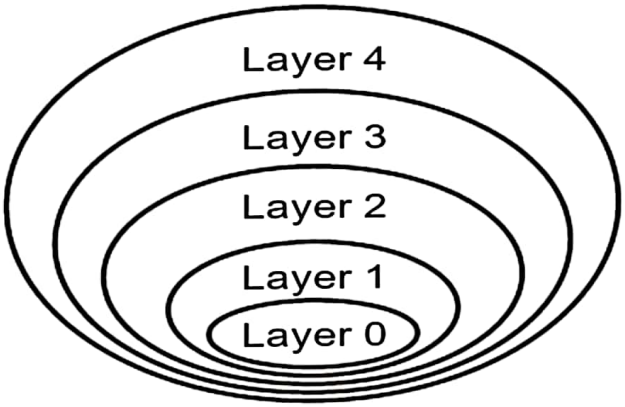
Layer Constraints

- *Static structure*—The software is partitioned into layers that each provide a cohesive set of services with a well-defined interface.
- *Dynamic structure*—Each layer is allowed to use only the layer directly below it (**Strict Layered** style) or the all the layers below it (**Relaxed Layered** style).

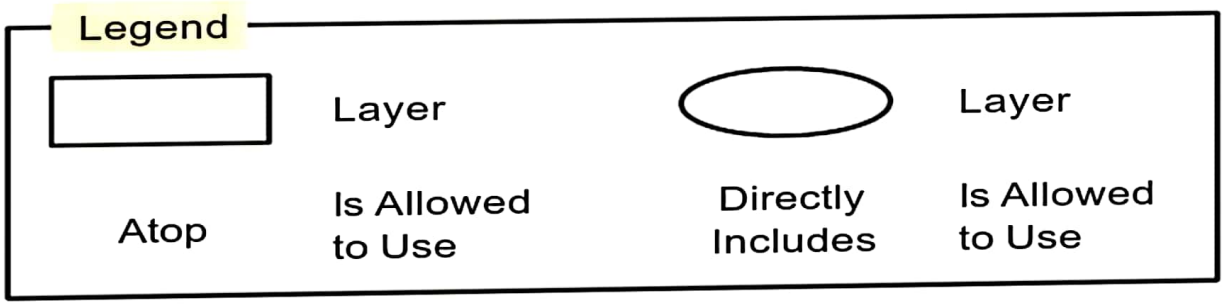
Representing Layers



Wedding Cake Diagram



Onion Diagram



Forming Layers

- Levels of abstraction
  - Example: Network communication layers
- Virtual machines
  - Examples: Operating systems, interpreters
- Information hiding, decoupling, etc
  - Examples: User interface layers, virtual device layers

Layered Style Advantages

- Layers are highly cohesive and promote information hiding.
- Layers are not strongly coupled to layers above them, reducing overall coupling.
- Layers help decompose programs, reducing complexity.
- Layers are easy to alter or fix by replacing entire layers, and easy to enhance by adding functionality to a layer.
- Layers are usually easy to reuse.

Layered Style Disadvantages

- Passing everything through many layers can complicate systems and damage performance.
- Debugging through multiple layers can be difficult.
- Layer constraints may have to be violated to achieve unforeseen functionality.

Other Architectural Styles

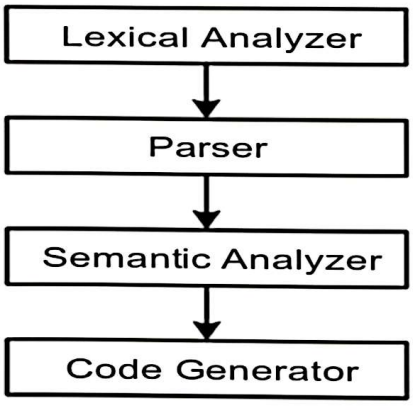
Architectural Styles

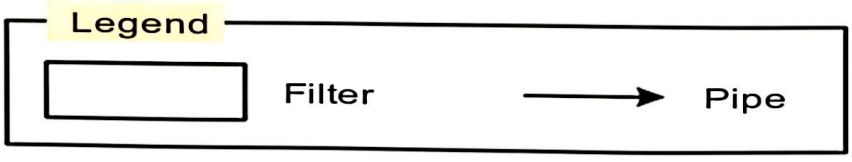
- An **architectural style** is a paradigm of program or system constituent types and their interactions.
- To present several important architectural styles, including
- Layered style
- Pipe-and-Filter style
- Shared-Data style
- Event-Driven style
- Model-View-Controller style
- Hybrid architectures

Pipe-and-Filter Style

- A **filter** is a program component that transforms an input stream to an output stream.
- A **pipe** is conduit for a stream.
- The **Pipe-and-Filter** style is a dynamic model in which program components are filters connected by pipes.

Pipe-and-Filter Example





Pipe-and-Filter Characteristics

- Pipes are isolated and usually only communicate through data streams, so they are easy to write, test, reuse, and replace.
- Filters may execute concurrently.
  - Requires pipes to synchronize filters
- Pipe-and-filter topologies should be acyclic graphs.
  - Avoids timing and deadlock issues
- A simple linear arrangement is a *pipeline*.

Pipe-and-Filter Advantages

- Filters can be modified and replaced easily.
- Filters can be rearranged with little effort, making it easy to develop similar programs.
- Filters are highly reusable.
- Concurrency is supported and is relatively easy to implement.

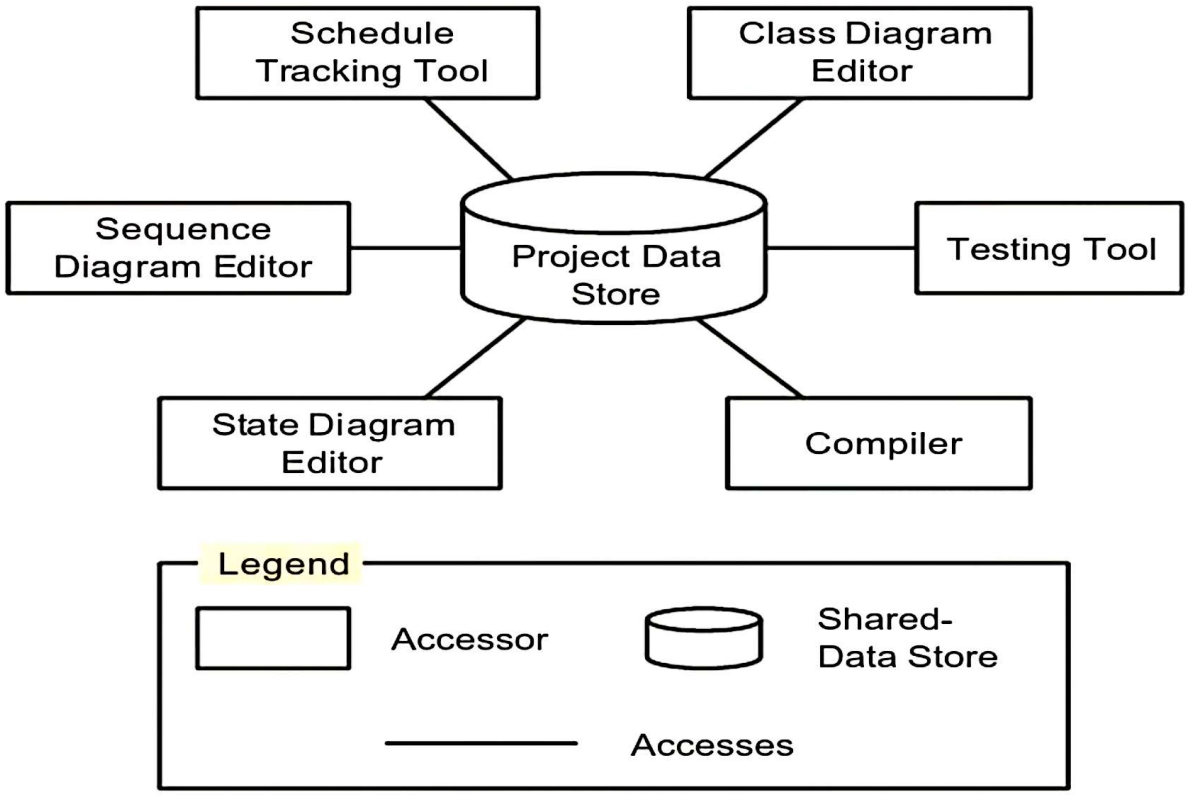
Pipe-and-Filter Disadvantages

- Filters communicate only through pipes, which makes it difficult to coordinate them.
- Filters usually work on simple data streams, which may result in wasted data conversion effort.
- Error handling is difficult.

Shared-Data Style

- One or more *shared-data stores* are used by one or more *shared-data accessors* that communicate solely through the shared-data stores.
- Two variants:
  - **Blackboard style**—The shared-data stores activate the accessors when the stores change.
  - **Repository style**—The shared-data stores are passive and manipulated by the accessors.
- This is a dynamic model only.

Shared-Data Style Example



Shared-Data Style Advantages

- Shared-data accessors communicate only through the shared-data store, so they are easy to change, replace, remove, or add to.
- Accessor independence increases robustness and fault tolerance.
- Placing all data in the shared-data store makes it easier to secure and control.

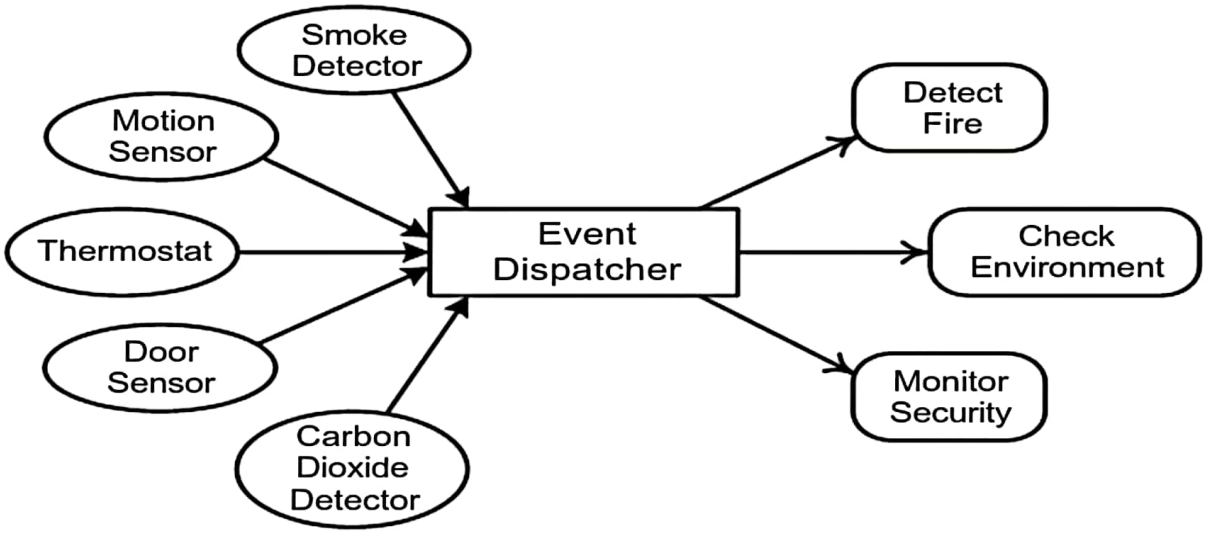
Shared-Data Style Disadvantages

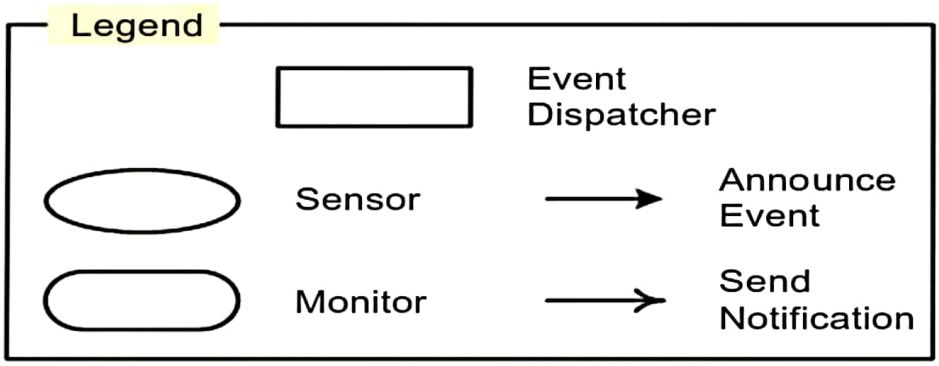
- Forcing all data through the shared-data store may degrade performance.
- If the shared-data store fails, the entire program is crippled.

Event-Driven Style

- Also called the **Implicit Invocation** style
- An **event** is any noteworthy occurrence.
- An *event dispatcher* mediates between components that announce and are notified of events.
- This is a dynamic model only

Event-Driven Style Example





Stylistic Variations

- Events may be notifications or they may carry data.
- Events may have constraints honored by the dispatcher, or the dispatcher may manipulate events.
- Events may be dispatched synchronously or asynchronously.
- Event registration may be constrained in various ways.

Event-Driven Style Advantages

- It is easy to add or remove components.
- Components are decoupled, so they are highly reusable, changeable, and replaceable.
- Systems built with this style are robust and fault tolerant.

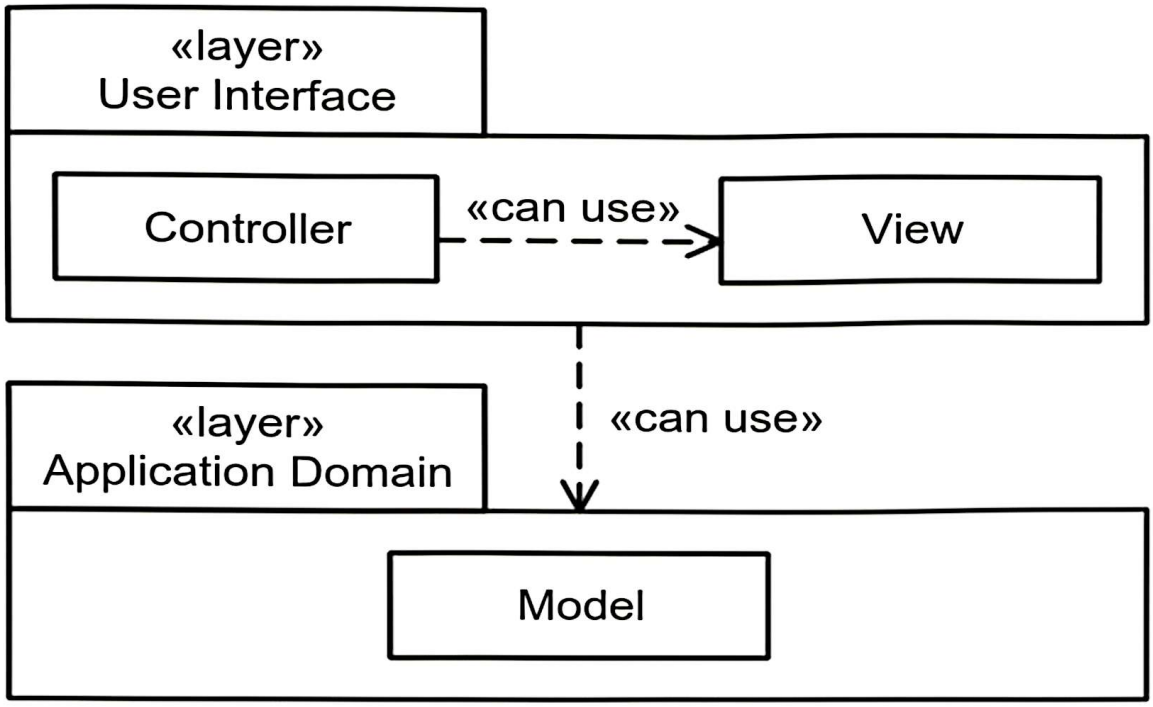
Event-Driven Style Disadvantages

- Component interaction may be awkward when mediated by the event dispatcher.
- There are no guarantees about event sequencing or timing, which may make it difficult to write correct programs.
- Event traffic tends to be highly variable, which may make it difficult to achieve performance goals.

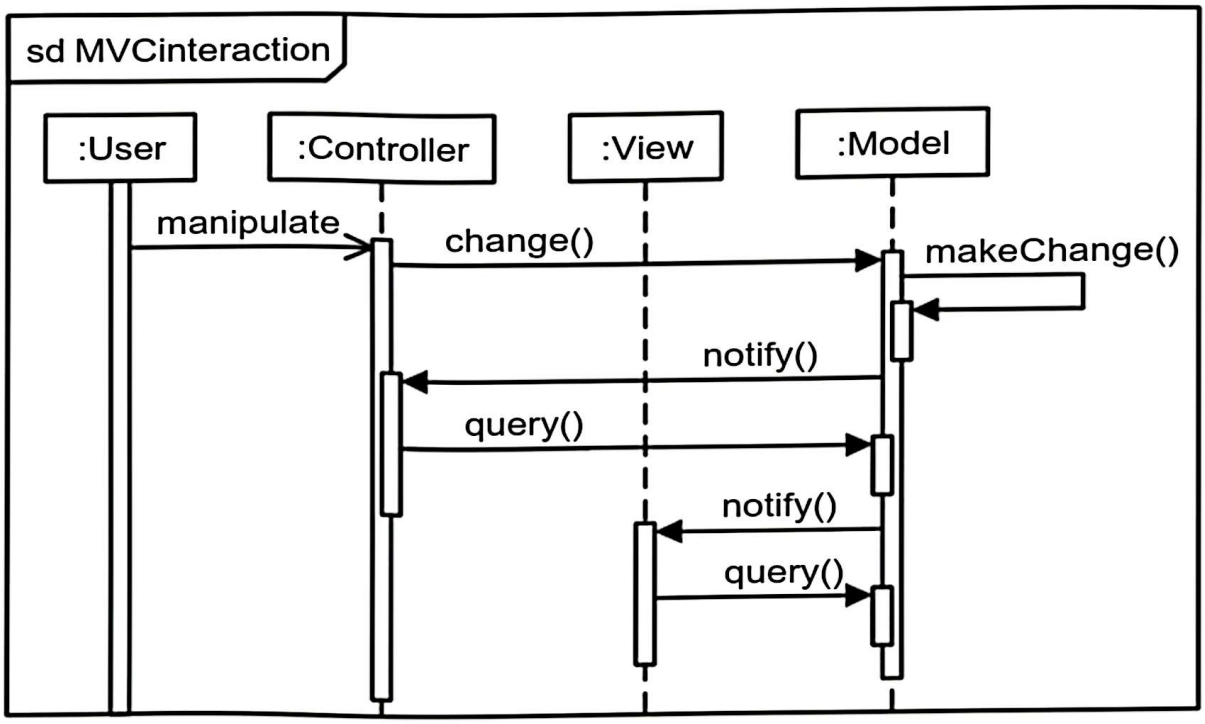
Model-View-Controller (MVC) Style

- This style models how to set up the relationships between user interface and problem-domain components.
- *Model*—A problem-domain component with data and operations for achieving program goals independent of the user interface
- *View*—A data display component
- *Controller*—A component that receives and acts on user input

MVC Static Structure



MVC Behavior



MVC Advantages

- Views and controllers can be added, removed, or changed without disturbing the model.
- Views can be added or changed during execution.
- User interface components can be changed, even at runtime.

MVC Disadvantages

- Views and controller are often hard to separate.
- Frequent updates may slow data display and degrade user interface performance.
- The MVC style makes user interface components highly dependent on model components.

Hybrid Architectures

Most systems of any size include several architectural styles, often at different levels of abstraction.

- An overall a system may have a Layered style, but the one layer may use the Event-Driven style, and another the Shared-Data style.
- An overall system may have a Pipe-and-Filter style, but the individual filters may have Layered styles.

THE END

By Wahab Ahmad