CS603 - Software Architecture and Design (Handouts)

By Wahab Ahmad

Week 1 Summery

Course Description

- Course introduces the essential concepts of software architecture and design. Both software architecture and design are important phases of software development.
- The course will provide a strong foundation to students for coming up with the skills necessary to develop architecture and realize it in design for developing applications fulfilling both functional and non-functional requirements.
- Different architectural styles and object oriented design patterns will be taught which will help students to analyze, select and implement design strategies required to realize the selected architecture. Overall course is an essential and great contribution to student's skills and their software engineering career.

Aim

- To provide the students an understanding of concepts and command to develop architecture and design for developing state of the art software applications.

Learning outcomes

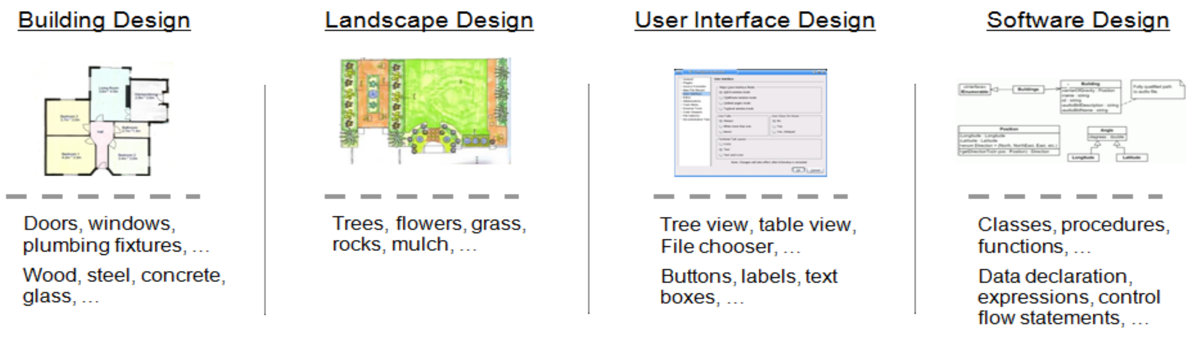
On successful completion of this course students would be able to

- Argue the importance and role of software architecture and design in software systems.
- 2. Develop Architecture and Design for software systems.
- 3. Recognize major software architectural styles and design patterns.
- 4. Apply software architecture/design strategies as per the given requirements to build state of art applications.

Introduction to the Discipline of Design

Design is a Universal Activity

- We live in a designed world.
- Design is economically important and effects our quality of life
- Any product that is an aggregate of more primitive elements, can benefit from the activity of design.

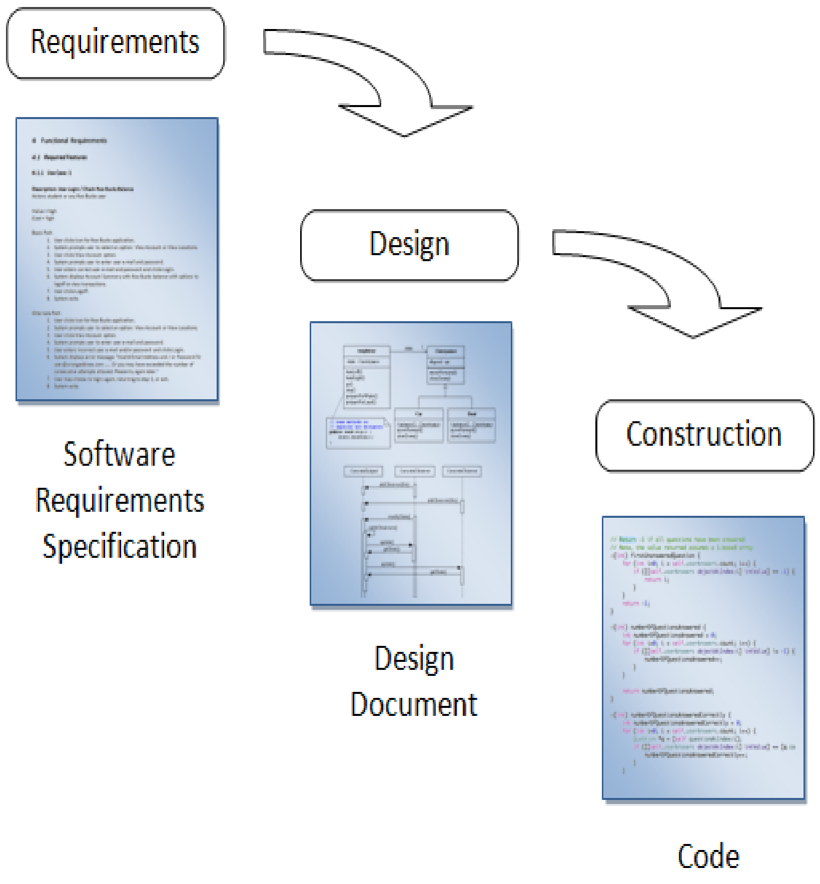

Building Design
Doors, windows, plumbing fixtures, …
Wood, steel, concrete, glass, …
Landscape Design
Trees, flowers, grass, rocks, mulch, …
User Interface Design
Tree view, table view, File chooser, …
Buttons, labels, text boxes, …
Software Design
Classes, procedures, functions, …
Data declaration, expressions, control flow statements, …

Software Products

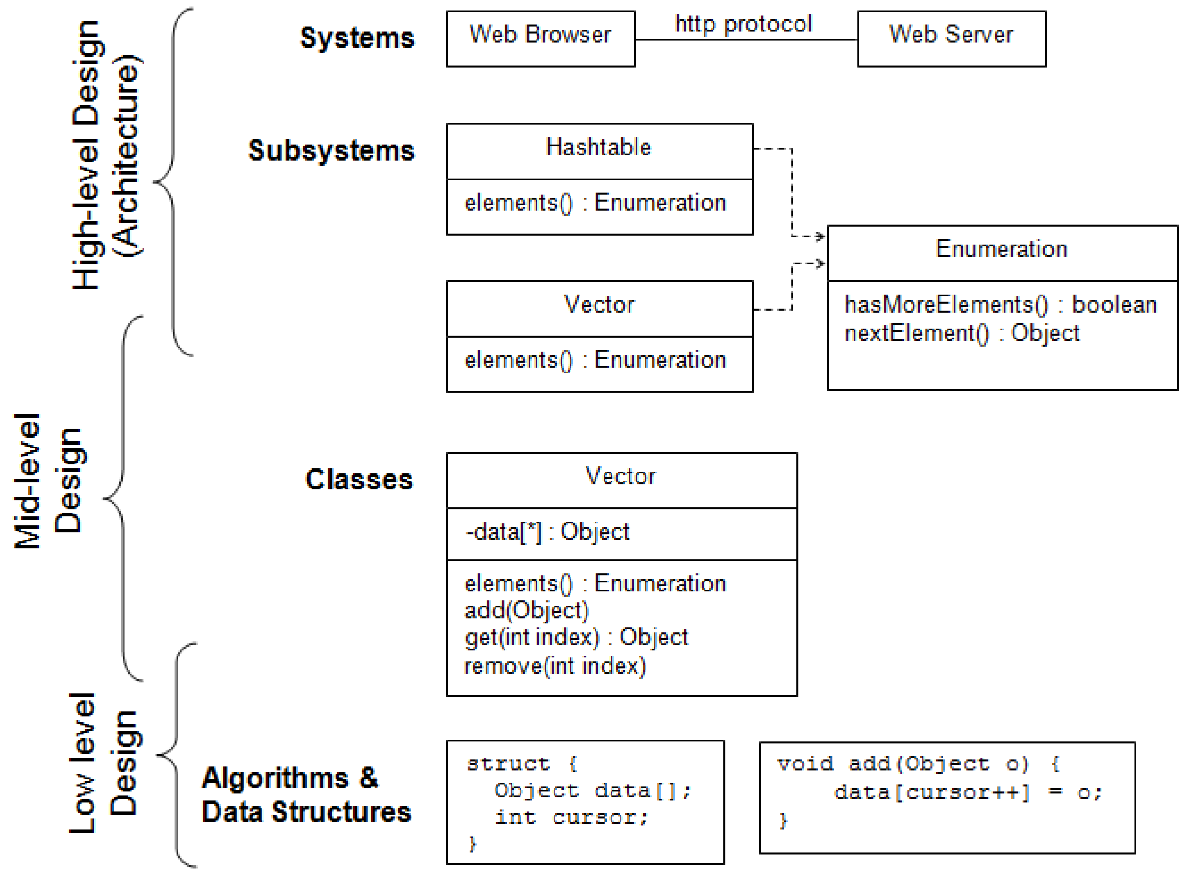
- **Software** is any executable entity, such as a program, or its parts, such as sub-programs.
- A software product is an entity comprised of one or more programs, data, and supporting materials and services that satisfies client needs and desires either as an independent artifact or as essential ingredient in some other artifact.

What Is Software Design?

- Software design is the activity of specifying the mature and composition of software products that satisfy client needs and desire, subject to constraints.
- Software designers do what designers in other disciplines do, except they do it for software products.
- Design bridges that gap between knowing what is needed (software requirements specification) to entering the code that makes it work (the construction phase).


Requirements
Software Requirements Specification
Design
Design Document
Construction
Code

Design Occurs at Different Levels



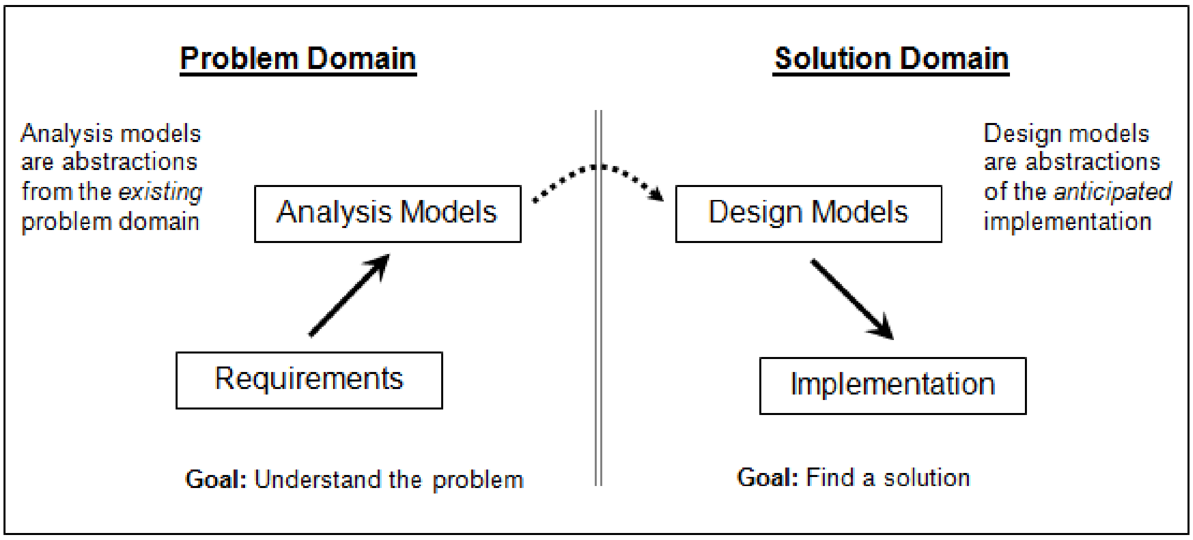
```
struct {
  Object data[];
  int cursor;
}
```

```
void add(Object o) {
    data[cursor++] = o;
}
```

Standard Levels of Design

Why Design is Hard

- Design is difficult because design is an abstraction of the solution which has yet to be created


Problem Domain
Solution Domain
Analysis models are abstractions from the existing problem domain
Design models are abstractions of the anticipated implementation
Analysis Models
Design Models
Requirements
Implementation
Goal: Understand the problem
Goal: Find a solution

Design as Problem Solving

- An especially fruitful way to think about design is as problem solving.

Advantages

- Suggests partitioning information between problem and solution
- Emphasizes that there may be more than one good solution (design)
- Suggests techniques such as changing the problem, trial and error, brainstorming, etc.

Abstraction

- Abstraction is an important problem-solving technique, especially in software design
- Abstraction is suppressing or ignoring some properties of objects, events, or situations in favor of others.

Importance of Abstraction

1.Problem simplification

- Abstracting allows us to focus on the most important aspects of a problem in (partially) solving it.

2.Structuring problem solving

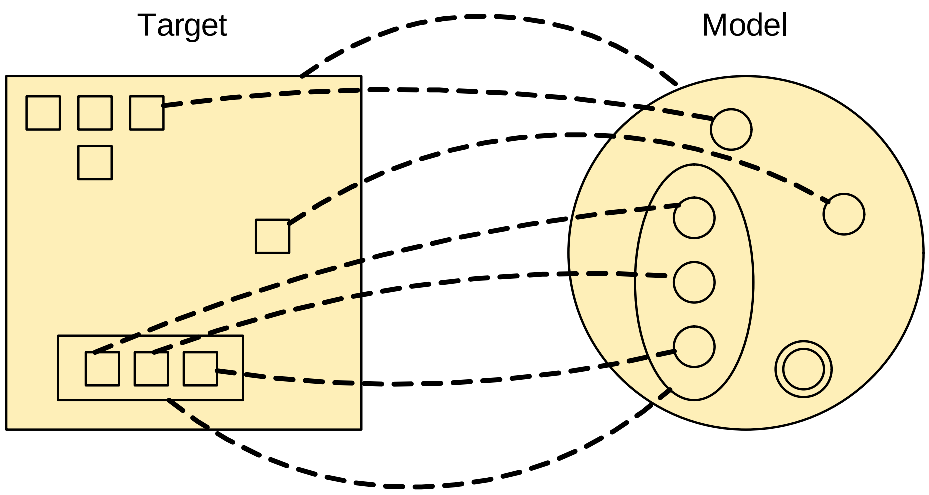
- **Top-down strategy**: Solve an abstract version of the problem, then add details (refinement)
- **Bottom-up strategy**: Solve parts of a problem and connect them for a complete solution

What is Model?

- A **model** is an entity used to represent another entity (the *target*) by establishing
- (a) a correspondence between the parts or elements of the target and the parts or elements of the model, and
- (b) a correspondence between relationships among the parts or elements of the target and relationships among the parts or elements of the model.

Modeling

- A model represents a target by having model parts corresponding to target parts, with relationships between model parts corresponding to relationships between target parts.



Modeling in Design

- Modeling is used for the following purposes:

1. Problem understanding
2. Design creation and investigation
3. Documentation

- Modeling work because models abstract details of the target.
- Models can fail if important and relevant details are left out.

Modeling in Software Design

Software design models may be divided into two broad classes: static and dynamic models

- A static model represents aspects of programs that do not change during program execution.
- A dynamic model represents what happens during program execution.

Static and Dynamic Models

- Static model examples include object and class models, component and deployment diagrams, and data structure diagrams.
- Dynamic model examples include use case descriptions, interaction diagrams, and state diagrams.

The Benefits of Good Design

- Good design reduces software complexity which makes the software easier to understand and modify. This facilitates rapid development during a project and provides the foundation for future maintenance and continued system evolution.
- It enables reuse. Good design makes it easier to reuse code.
- It improves software quality. Good design exposes defects and makes it easier to test the software.
- Complexity is the root cause of other problems such as security. A program that is difficult to understand is more likely to be vulnerable to exploits than one that is simpler.

Varieties of Design

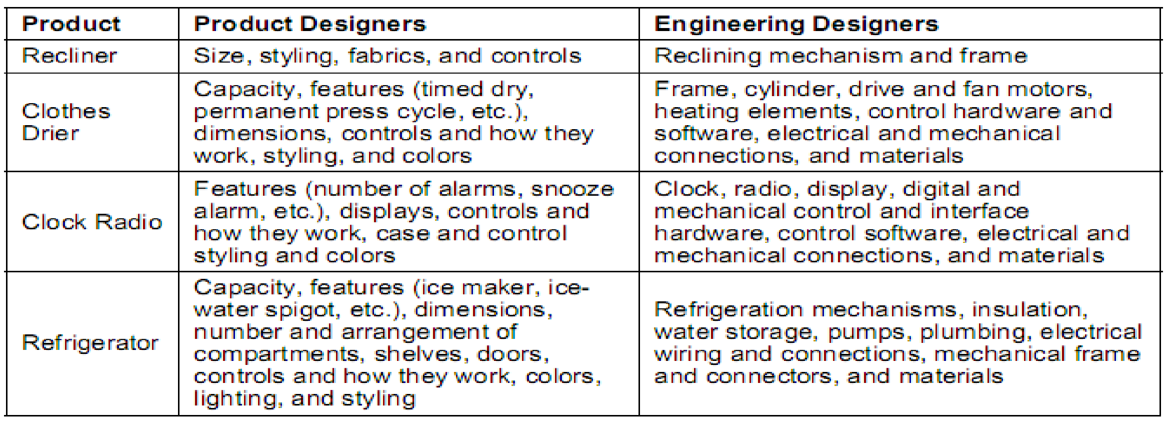
- **Product design** is a discipline that arose during the Industrial Revolution and is now an established field whose practitioners specify products.
- The major issues in product design are aesthetics, product features and capabilities, usability, manageability, manufacturability, and operability.
- **Engineering design** is the activity of specifying the technical mechanisms and workings of a product. Engineers apply mathematical and scientific principles and techniques to work out the technical details of complex products.
- Product designers and engineers often work together in design teams *to specify large and complex products.*

Product Designer vs. Engineering Designer

- Product designers are concerned with styling and aesthetics, function and usability, manufacturability and manageability.
- Industrial designers, (building) architects, interior designers, graphic designers, etc.
- Engineering designers are concerned with technical mechanisms and workings.
- Structural, mechanical, chemical, and electrical engineers
- Product designers handle the "externals" of product design while engineers take care of the "internal" technical details.

Design Teams

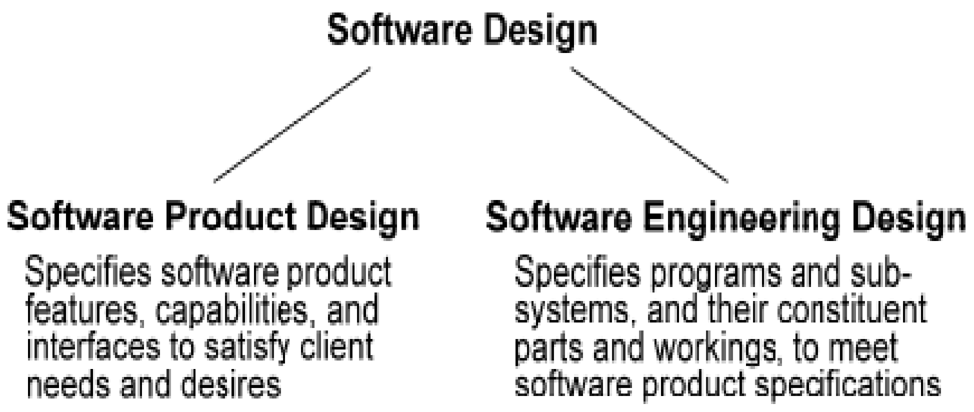
- The talents and skills of both product designers and engineers are needed to design such things. Table 1 illustrates the complementary responsibilities of product and engineering designers for several products.
- Table 1: Product and Engineering Designers' Responsibilities



Product	Product Designers	Engineering Designers
Recliner	Size, styling, fabrics, and controls	Reclining mechanism and frame
Clothes Drier	Capacity, features (timed dry, permanent press cycle, etc.), dimensions, controls and how they work, styling, and colors	Frame, cylinder, drive and fan motors, heating elements, control hardware and software, electrical and mechanical connections, and materials
Clock Radio	Features (number of alarms, snooze alarm, etc.), displays, controls and how they work, case and control styling and colors	Clock, radio, display, digital and mechanical control and interface hardware, control software, electrical and mechanical connections, and materials
Refrigerator	Capacity, features (ice maker, ice-water spigot, etc.), dimensions, number and arrangement of compartments, shelves, doors, controls and how they work, colors, lighting, and styling	Refrigeration mechanisms, insulation, water storage, pumps, plumbing, electrical wiring and connections, mechanical frame and connectors, and materials

Software design

- The field of software design can be divided into two sub-fields that each demand considerable skill and expertise: software product design and software engineering design.


Software Design
Software Product Design
Specifies software product features, capabilities, and interfaces to satisfy client needs and desires
Software Engineering Design
Specifies programs and sub-systems, and their constituent parts and workings, to meet software product specifications

Software Product Design

- Software product design is the activity of specifying software product features, capabilities, and interfaces to satisfy client needs and desires.
- Requires skills in user interface and interaction design, communications, industrial design, and marketing

Software Engineering Design

- Software engineering design is the activity of specifying programs and sub-systems, and their constituent parts and workings, to meet software product specifications.
- Requires skills in programming, algorithms, data structures, software design principles, practices, processes, techniques, architectures, and patterns

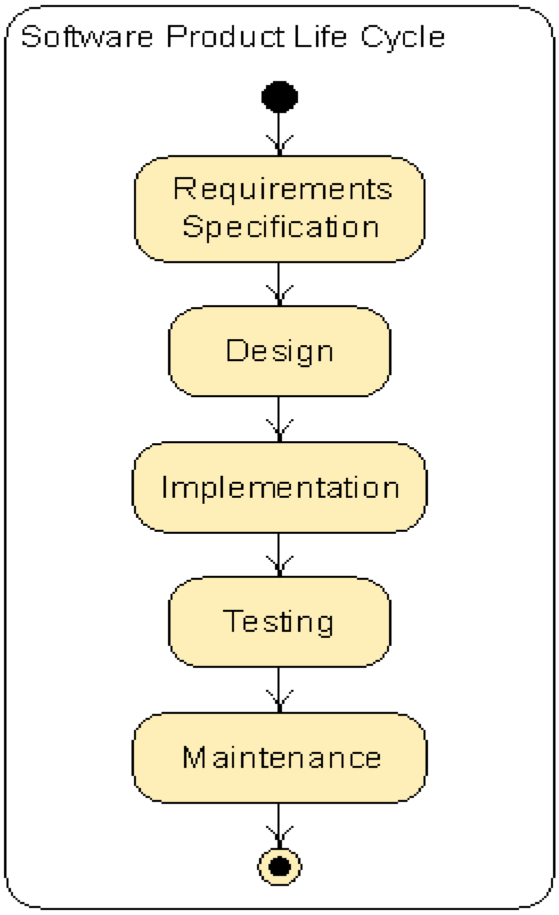
Week 2 Summery

Software Design in the Life Cycle

- The software life cycle is the sequence of activities through which a software product passes from initial conception through retirement from service**.**

Waterfall Life Cycle Model

The waterfall model captures the logical, but not the temporal, relationships between software development activities.



Requirements Specification Activity

- The goal of the requirements specification activity is to specify a product satisfying the needs and desires of clients and other interested parties.
- **specifications are recorded in a software requirements specification (SRS).**
- we assume that every SRS includes a user interface design.
- Factors that limit the range of design solutions, such as cost, time, size, user capability, and required technology, are called design **constraints. Design** constraints are usually given as part of the problem specification

Design Activity

- During the design activity, developers figure out how to build the product specified in the SRS. This includes selecting an overall program structure, specifying major parts and sub-systems and their interactions, then determining how each part or sub-system will be built.
- The result of the design activity is a **design document** recording the entire design specification. The design document solves the (engineering) design problem posed in the SRS.

Implementation Activity

- Code is written in accord with the specifications in the design document. The product of the implementation activity is a more or less finished, working program satisfying the SRS.
- Programming essentially includes some engineering design work.

Testing Activity

- Programs are run during the testing activity to find bugs.
- Testing is usually done bottom up, with small parts or program units tested alone, and then integrated collections of program units tested as separate sub-systems, and finally the entire program tested as a whole.

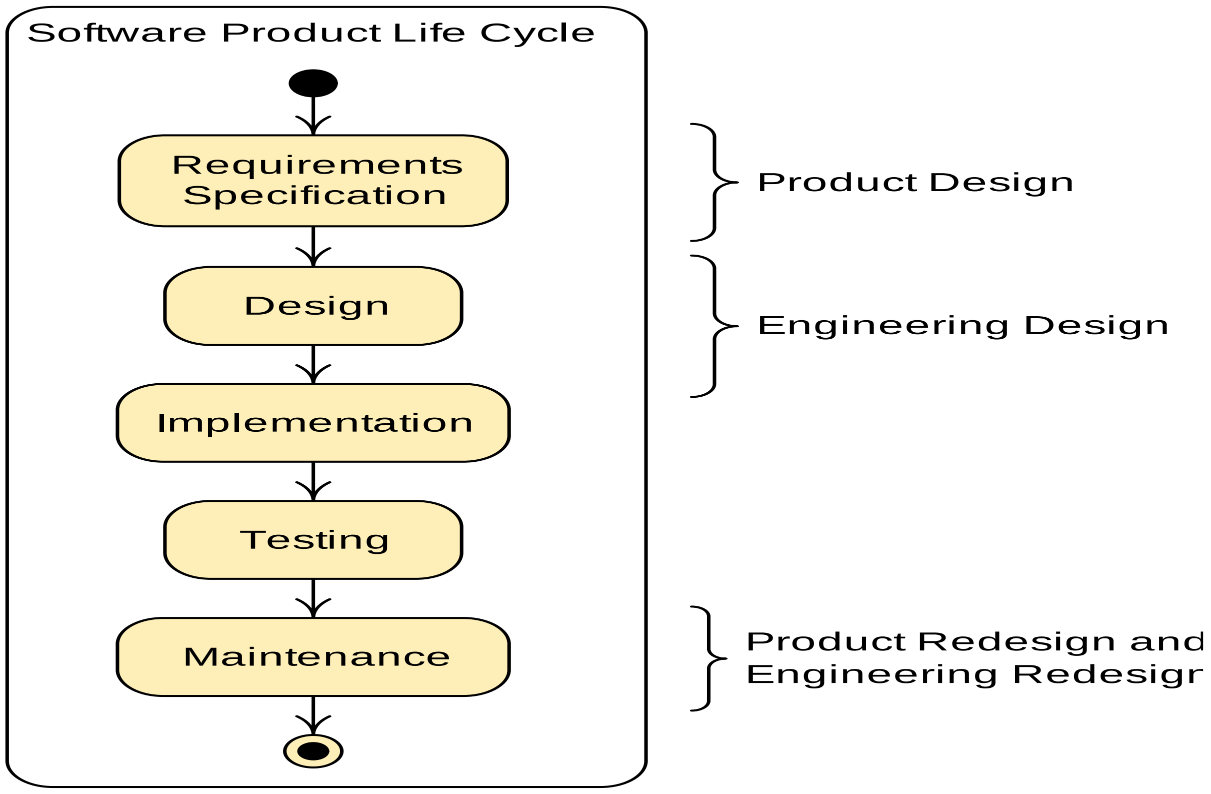
Maintenance Activity

- Maintenance activity occurs after a product has been deployed to clients
- After deployment to clients, products are corrected, ported, and enhanced during maintenance activities.

- Product design occurs during the requirements specification and maintenance activities, and engineering design occurs during the design, implementation, and maintenance activities.

Design Across the Life Cycle

Design Across the Life Cycle Figure illustrates how software design activities are spread across the life cycle.


Software Product Life Cycle
Requirements Specification
Design
Implementation
Testing
Maintenance
Product Design
Engineering Design
Product Redesign and Engineering Redesigr

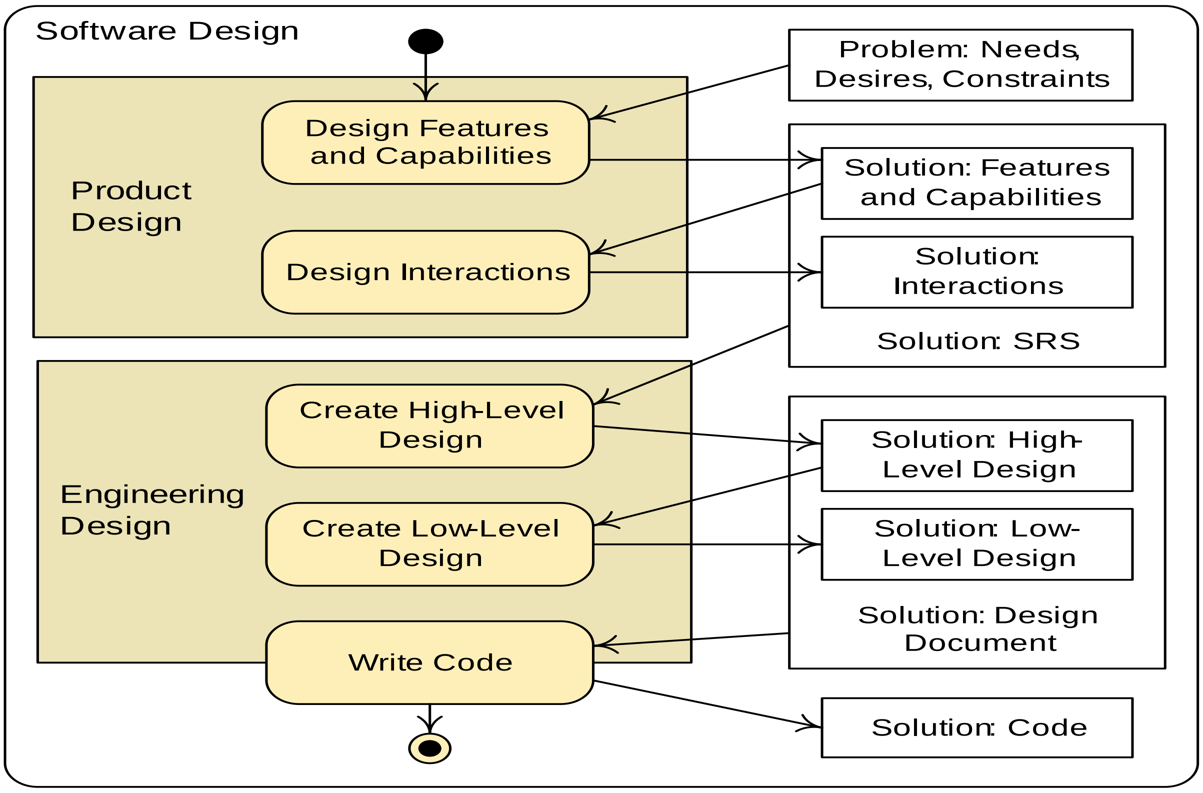
"What" Versus "How"

- Traditional way to make the distinction between requirements and design activities
- Not adequate because
  - Many "what" specifications turn out to be design decisions
  - Many "how" specifications turn out to be client or customer needs or desires
- Distinguish requirements from design based on problem solving: requirements activity formulates a problem solved in design

Design Problems and Solutions

- Problems and solutions demarcate various software design activities. Product design tackles a client problem and produces a product specification as a solution. This solution presents the problem to engineering designers, who produce a design document as their solution.

Design Problems and Solutions


Software Design
Problem: Needs, Desires, Constraints
Product Design
Design Features and Capabilities
Design Interactions
Solution: Features and Capabilities
Solution: Interactions
Solution: SRS
Engineering Design
Create High-Level Design
Create Low-Level Design
Write Code
Solution: High-Level Design
Solution: Low-Level Design
Solution: Design Document
Solution: Code

"Design" as a Verb and a Noun

- This activity is what we refer to when we use the word "design" as a verb, as in the sentence "Engineers design programs meeting requirements specifications." But we have also used "design" as a noun, as in the sentence "Engineers develop a design meeting requirements specifications."
- Obviously, the word "design" is both a verb and a noun and refers to both an activity and a thing. A design specification is the output of the design activity and should meet the goals of the design activity—it should specify a program satisfying client needs and desires, subject to constraints.

Software Engineering Design Methods

- A **software design method** is an orderly procedure for generating a precise and complete software design solution that meets clients' needs and constraints

Design Method Components

A method typically specifies the following items:

- ***Design Process*** **—**A collection of related tasks that transforms a set of inputs into a set of outputs
- ***Design Notations*** —A symbolic representational system
- ***Design Heuristics*** —Rules providing guidance, but no guarantee, for achieving some end
- Design methods also use **design principles** stating characteristics of design that make them better or worse.

History of Software Engineering Design Methods

- The first design method was **stepwise refinement**, a top-down technique for decomposing procedures into simpler procedures until programming-level operations are reached**.**
- The dominant design methods from the mid-1970s through the early 1990s were various versions of **structured design.**
- Structured design methods focus on procedural composition but include other sorts of models as well.
- Object-oriented design methods emerged in the 1990s in response to shortcomings of structured design methods.
- Object-oriented methods promote thinking about programs as collections of collaborating objects rather than in terms of procedural decomposition.

Method Neutrality

- Strongly emphasizes object-oriented notations, heuristics, and models.
- Most of the notations used in this course are UML notations, but some other important notations are included as well.
- A design task is a small job done in the design process, such as choosing classes or operations, or checking whether a model is complete. Notation and task heuristics are discussed throughout the course when notations and design tasks are introduced.

Week 3 Summery

Modeling processes with Activity diagram

Modeling

- A picture is worth 1000 words.
- A model is a representation of reality, like a model car, airplane.
- Most models have both diagrams and textual components.

What is UML?

- UML stands for "Unified Modeling Language"
- It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems.

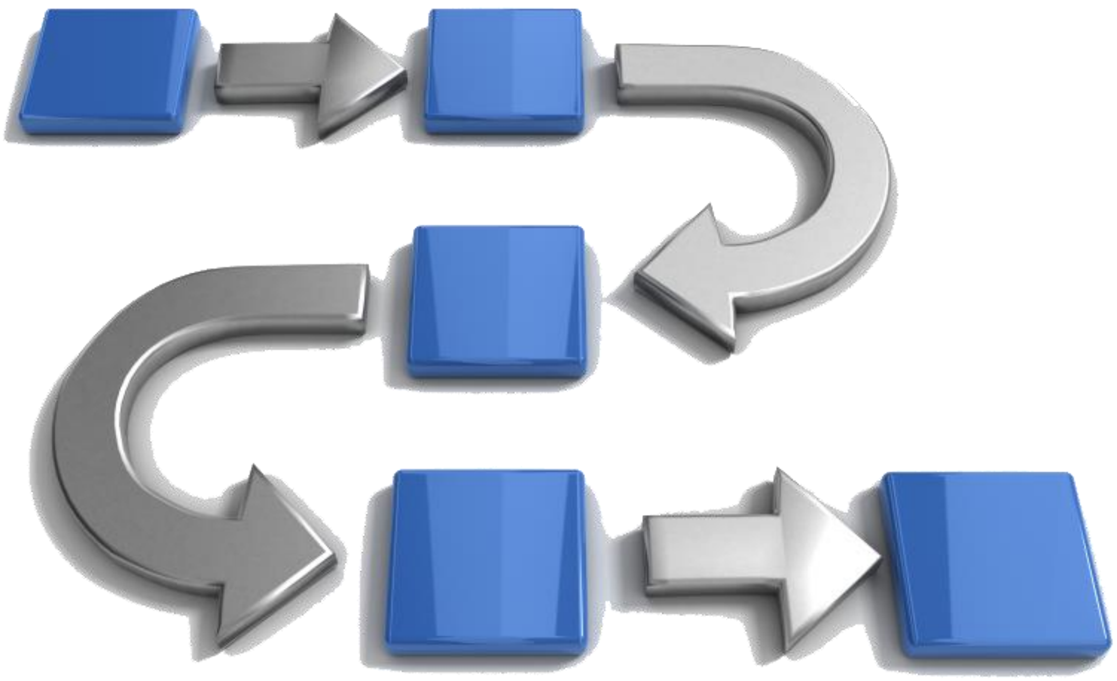
UNIFIED MODELING LANGUAGE™

What is UML?

- The UML uses mostly graphical notations to express the OO analysis and design of software projects.
- Simplifies the complex process of software design.

Process

- A process is a collection of related tasks that transforms a set of inputs into a set of outputs.



Design Process

- A design process is the core of any design endeavor, so it is essential that designers adopt an efficient and effective process.
- We need process description notations for design process.
- We will use UML Activity diagram.

Activity diagram

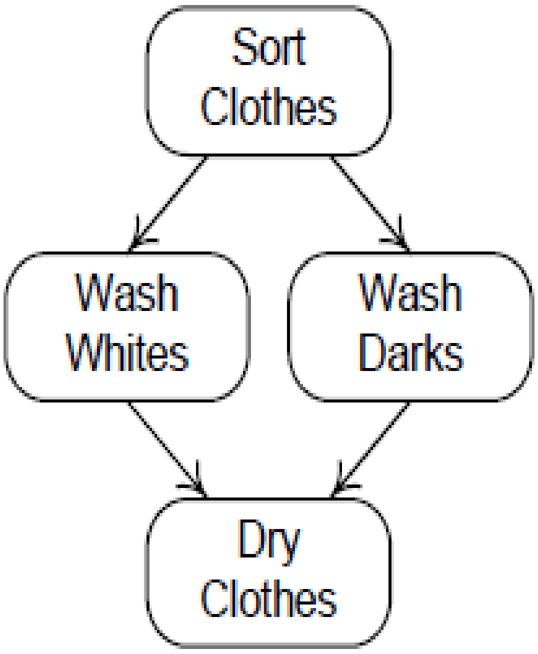
- An activity diagram shows actions and the flow of control and data between them.

Activity, action and execution

- An **activity** is a non-atomic task or procedure decomposable into actions.
    - Shipping a product
    - Wash clothes
    - …
- An **action** is a task or procedure that cannot be broken into parts (i.e. it is atomic).
    - Check products in stock
    - Check dead level
    - Package the product
    - …

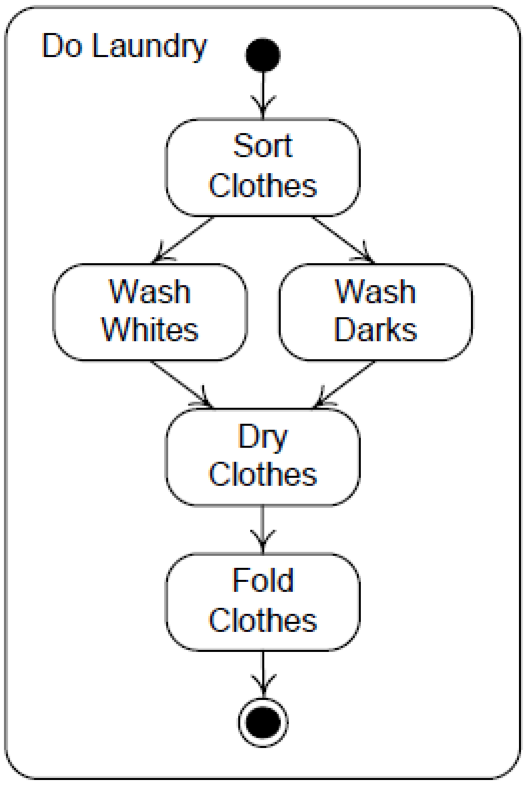
Activity, action and execution

- Activity diagrams model processes as an *activity graph*.
    - *Activity nodes* represent actions or objects
        - Rounded rectangle containing arbitrary text naming or describing some action.
    - *Activity edges* represent control or data flows.
        - Represented by solid arrows with unfilled arrow heads.



Modeling processes with Activity diagram

Activity graph elements

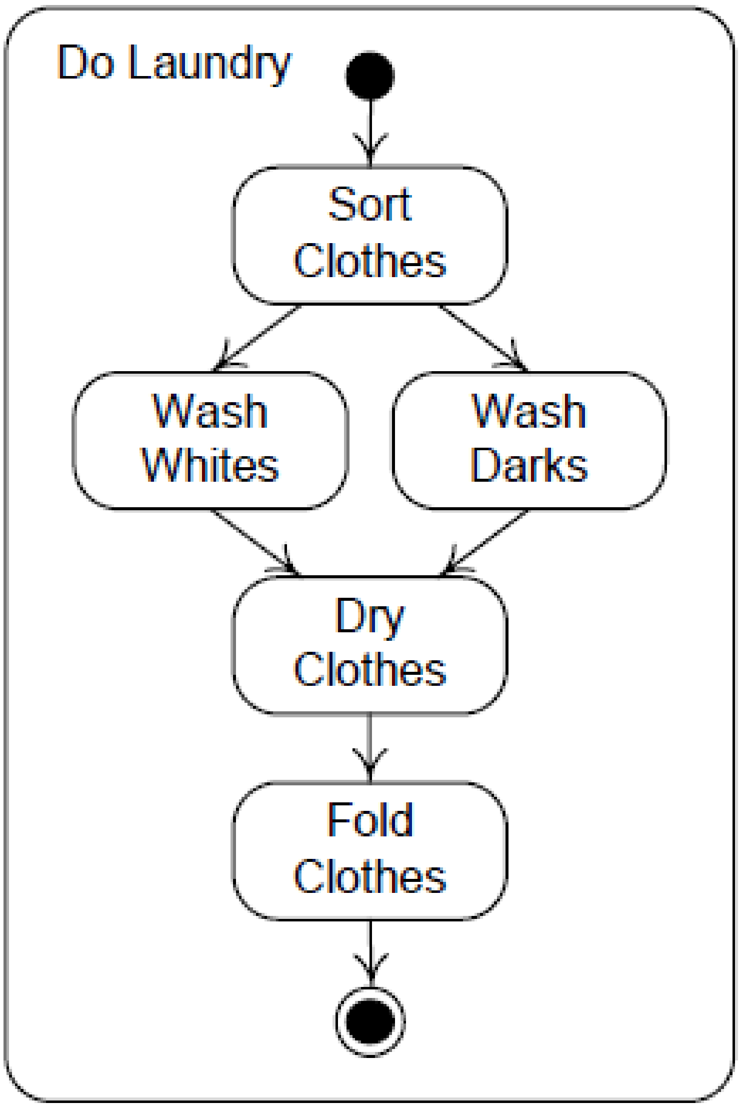


Activity diagram execution

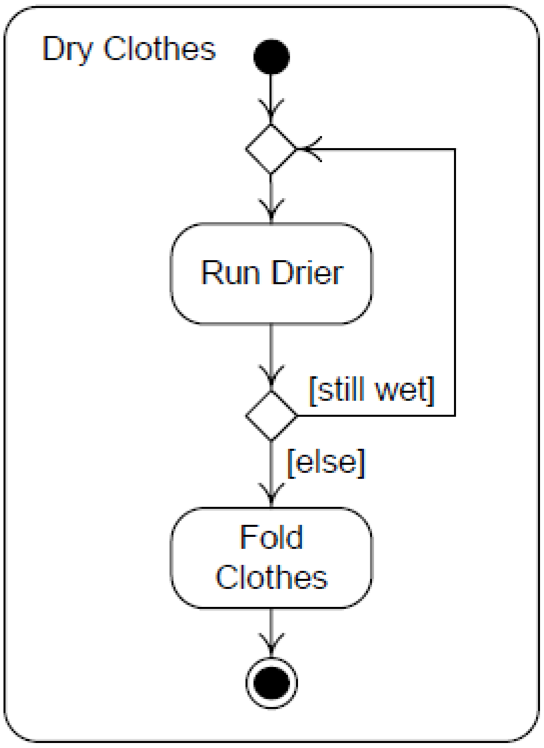
- Execution is modeled by tokens.
- When there is a token on every incoming edge of an action node, it consumes them and begins execution.
- When an action node completes execution, it produces tokens on each of its outgoing edges.
- An initial node produces a token on each outgoing edge when an activity begins.
- An activity final node consumes a token available on any incoming edge and terminates the activity.

- Execution is modeled by tokens.
- When there is a token on every incoming edge of an action node, it consumes them and begins execution.
- When an action node completes execution, it produces tokens on each of its outgoing edges.
- An initial node produces a token on each outgoing edge when an activity begins.
- An activity final node consumes a token available on any incoming edge and terminates the activity.

Activity diagram execution



Branching nodes

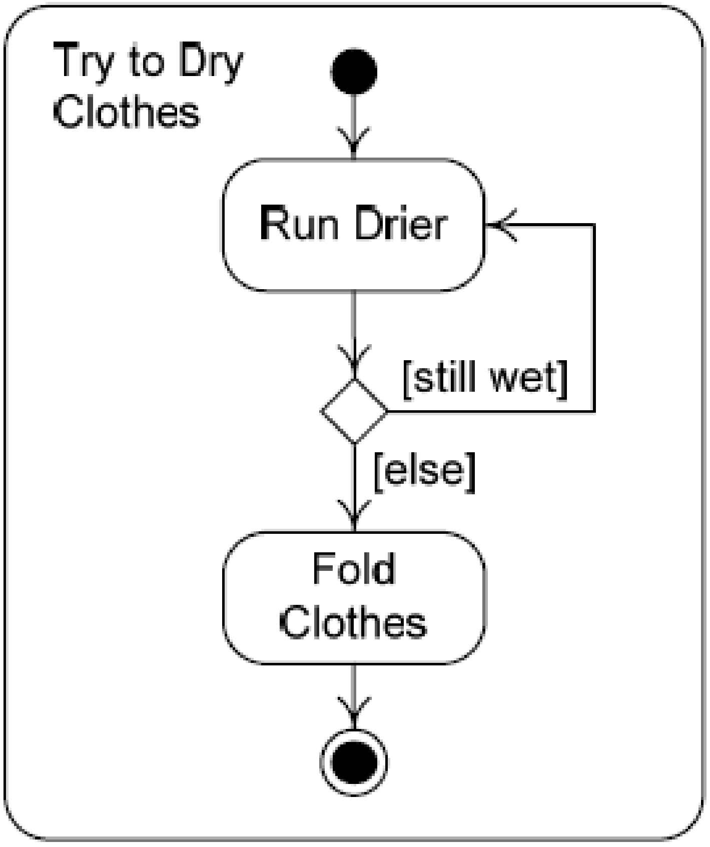


Branching execution

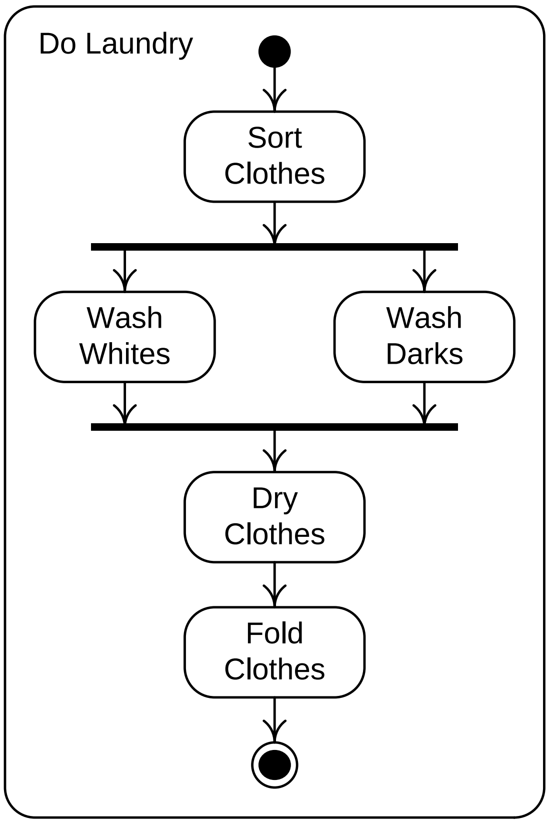
- If a token is made available on the incoming edge of a decision node, the token is made available on the outgoing edge whose guard is true.
- If a token is available on any incoming edge of a merge node, it is made available on its outgoing edge.
- Guards must be mutually exclusive.

Deadlocks

- Run Drier cannot execute: when the activity begins, there is a token on the edge from the initial node but not on the other incoming edge.



Forks and joins

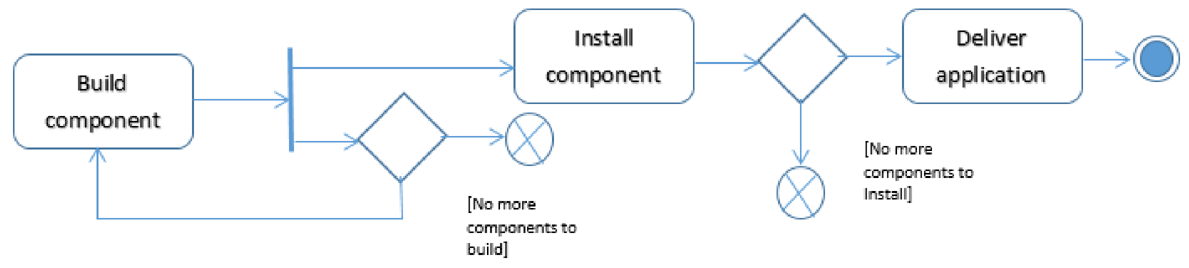


Forks and joins execution

- A token available on the incoming edge of a fork node is reproduced and made available on all its outgoing edges.
- When tokens are available on every incoming edge of a join node, a token is made available on its outgoing edge.
- Concurrency can be modeled without these nodes.

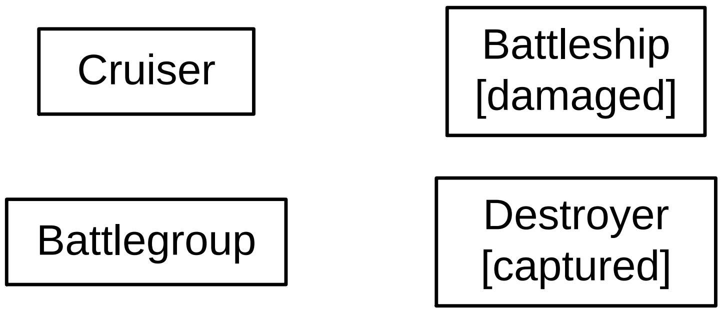
Forks and joins

- Flow final node

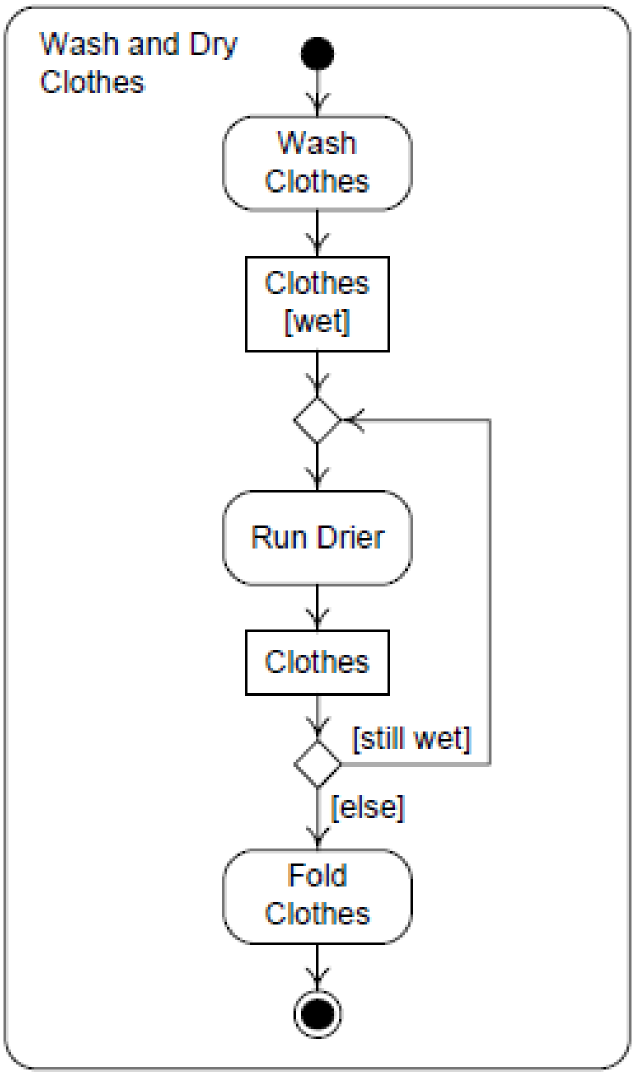

Build component
Install component
Deliver application
[No more components to build]
[No more components to Install]

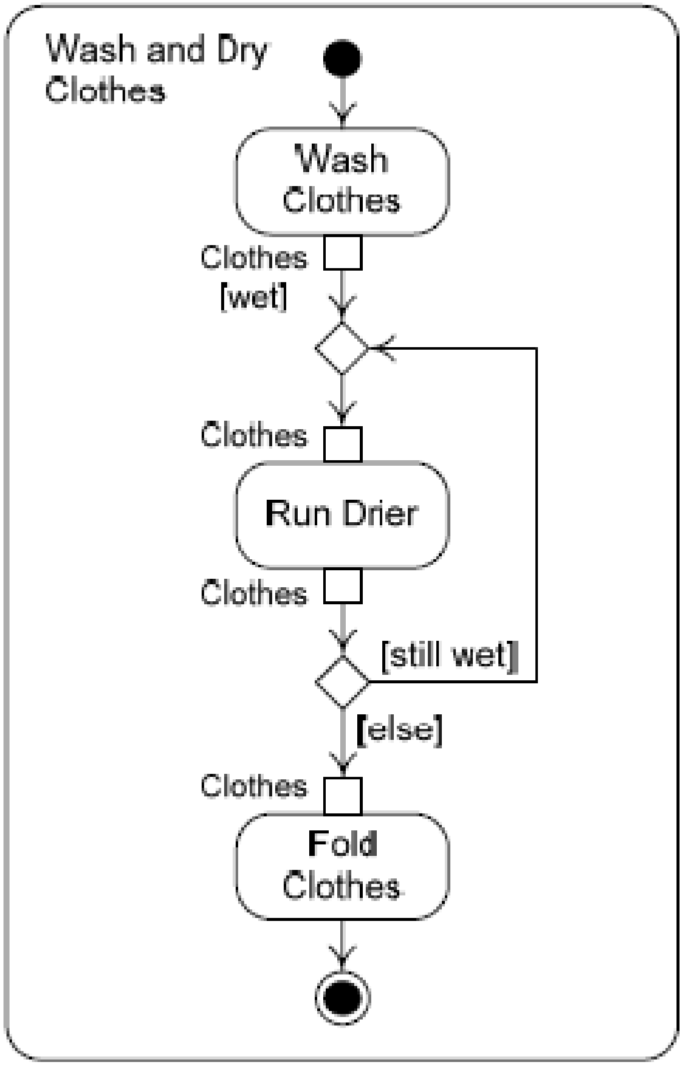
Object Nodes

- Data and objects are shown as object nodes.
- Any flow that begins or ends at an object node is a data flow.



Object Nodes





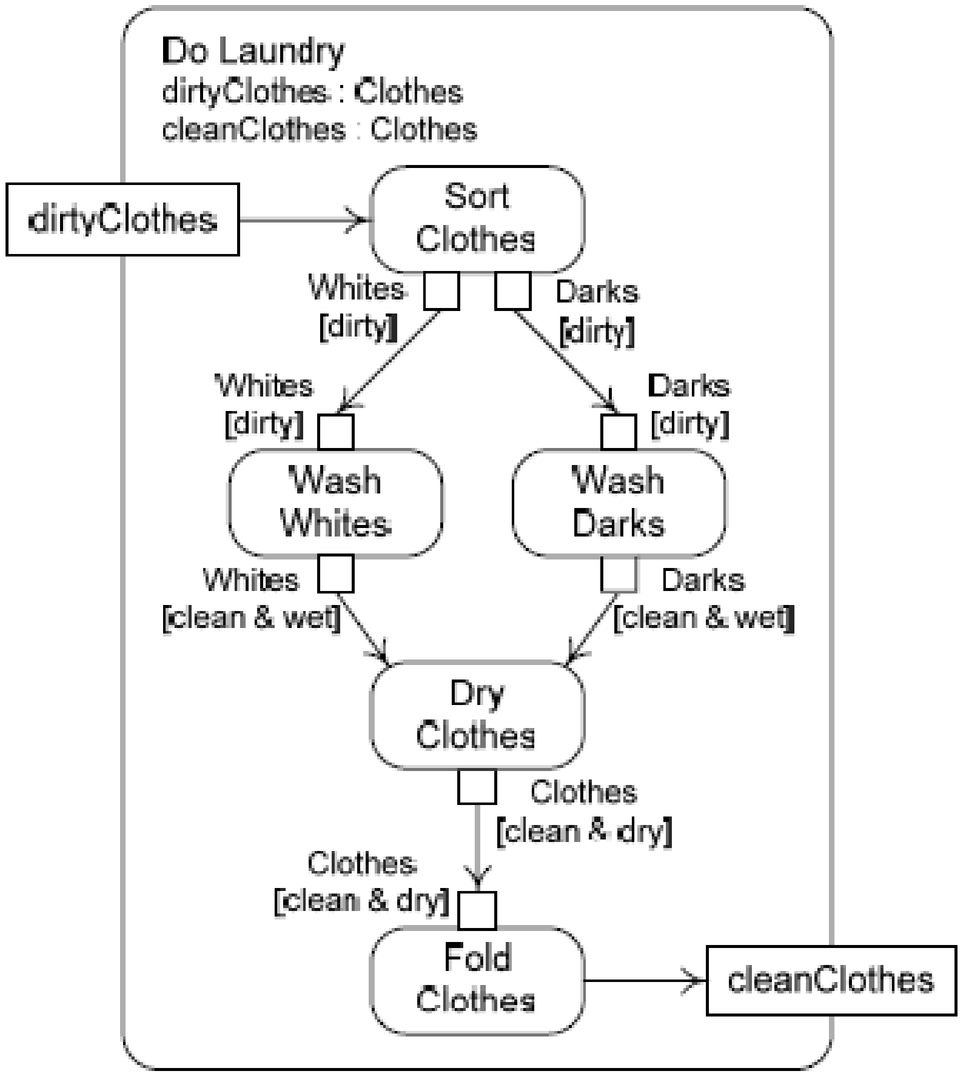
Control and Data Flows

- Control tokens do not contain data, data tokens do.
- A control flow is an activity edge, conduit for control tokens.
- A data flow is an activity edge, conduit for data tokens.
- Rules for token-based execution apply just as well to data flows as to control flows, with the addition of a mechanism for adding and removing data from tokens.

Activity parameters

- An activity parameter is an object node placed on the boundaries of an activity symbol to represent data or object inputs or outputs.
- Input activity parameters have only outgoing arrows, and output activity parameters have only incoming arrows.

Activity parameters (Example)



Activity diagram heuristics

- Model flow control and objects down the page and from left to right.
- Name activities and action nodes with verb phrases.
- Name object nodes and pins with noun phrases.
- Don’t use both control and data flows when a data flow alone can do the job.
- Make sure that all flows entering an action node can provide tokens concurrently.
- Use the [else] guard at every branch.

Summary

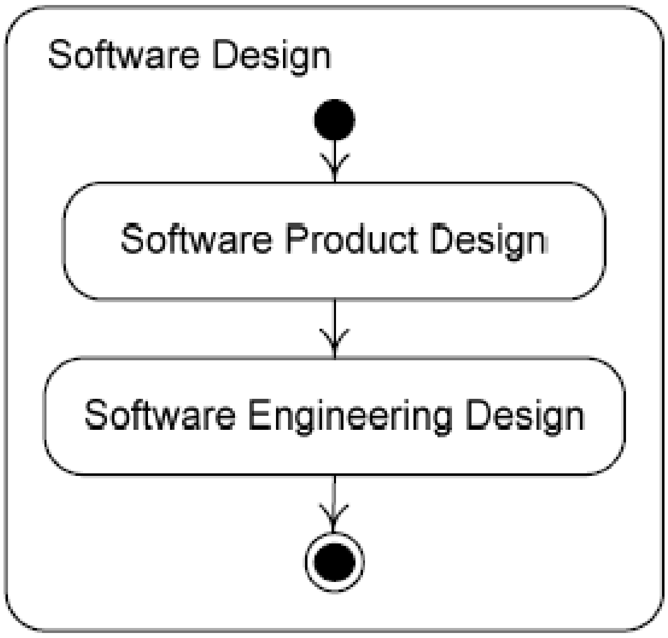
- Forks and joins
- Activity parameters
- Activity diagram heuristics

Week 4 Summery

Software design processes

Software design

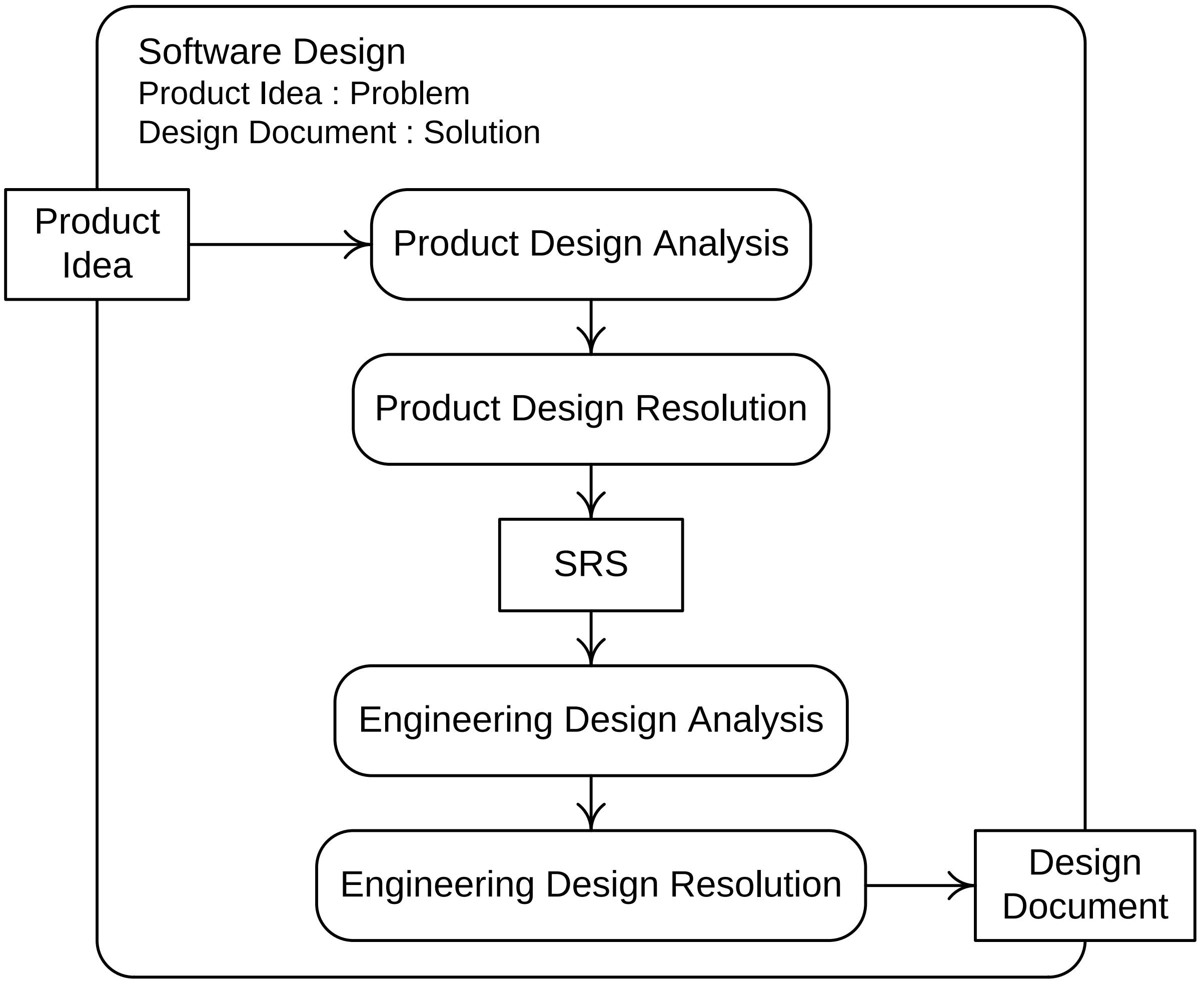
- Software design consists of two different activities.
  - Software product design
  - Software engineering design



Analysis and resolution

- The first step of “problem solving” must always be to understand the problem.
- If design is problem solving, then this activity must be the first step in design.
- *Analysis* is the activity of breaking down a design problem for the purpose of understanding it.
- Once problem is understood, next step is to solve it.
- Unfortunately, the activity of solving a design problem does not have a good, widely accepted name.
- Traditionally this activity has been called design, but this is very confusing.
- In the traditional way of speaking, design consists of the following steps:
  - *Analysis*—Understanding the problem.
  - *Design*—Solving the problem.
- In our context, we refer to the activity of solving a design problem as *resolution*.
- The terms used in our context will be:
  - **Analysis** is breaking down a design problem to understand it.
  - **Resolution** is solving a design problem

Analysis and Resolution in Software Design


Software Design
Product Idea : Problem
Design Document : Solution
Product Idea
Product Design Analysis
Product Design Resolution
SRS
Engineering Design Analysis
Engineering Design Resolution
Design Document

A generic problem solving strategy

- Understand the problem
- Generate candidate solutions
- Evaluate solutions
- Select best solution(s)
- Iterate if no solution is adequate
- Ensure the solution is complete, well-documented, and deliver it

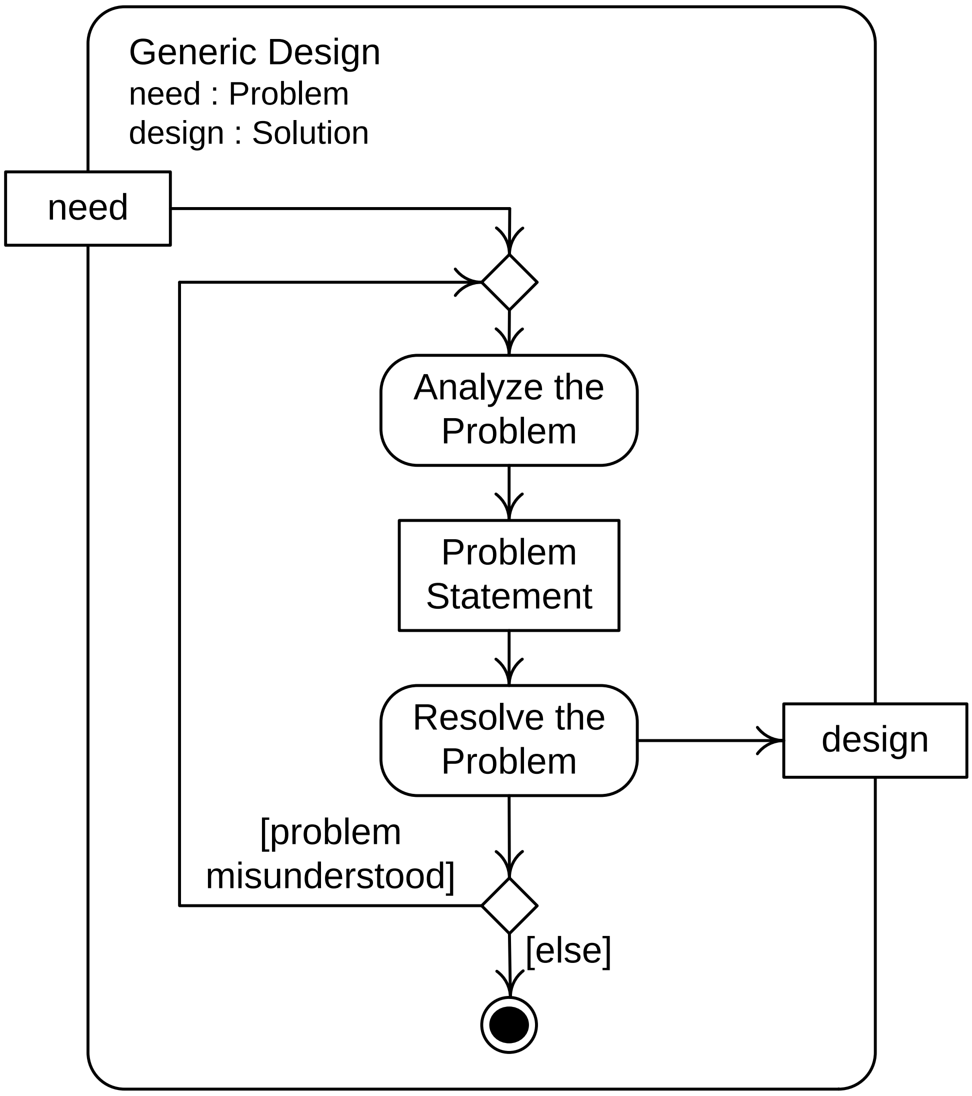
A generic problem solving strategy

- Understand the problem
- Generate candidate solutions
- Evaluate solutions
- Select best solution(s)
- Iterate if no solution is adequate
- Ensure the solution is complete, well-documented, and deliver it

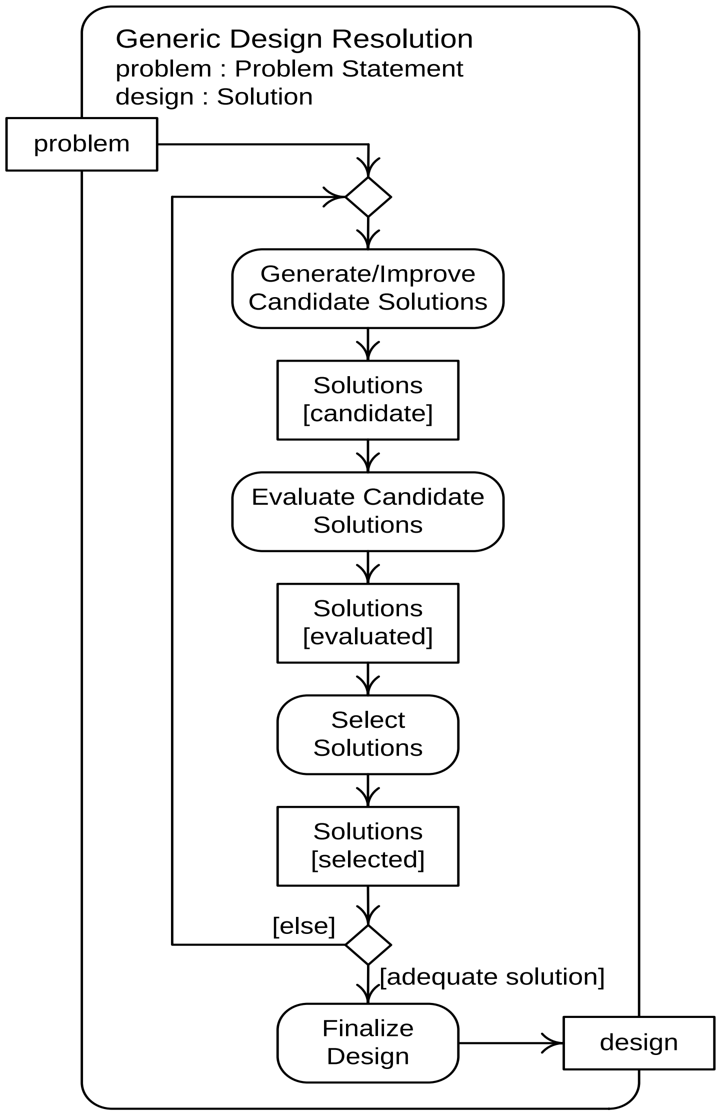
A generic design process

- Analyze the Problem
- Generate/Improve Candidate Solutions
- Evaluate Candidate Solutions
- Select Solutions
- Iterate
- Finalize the Design

Generic design process



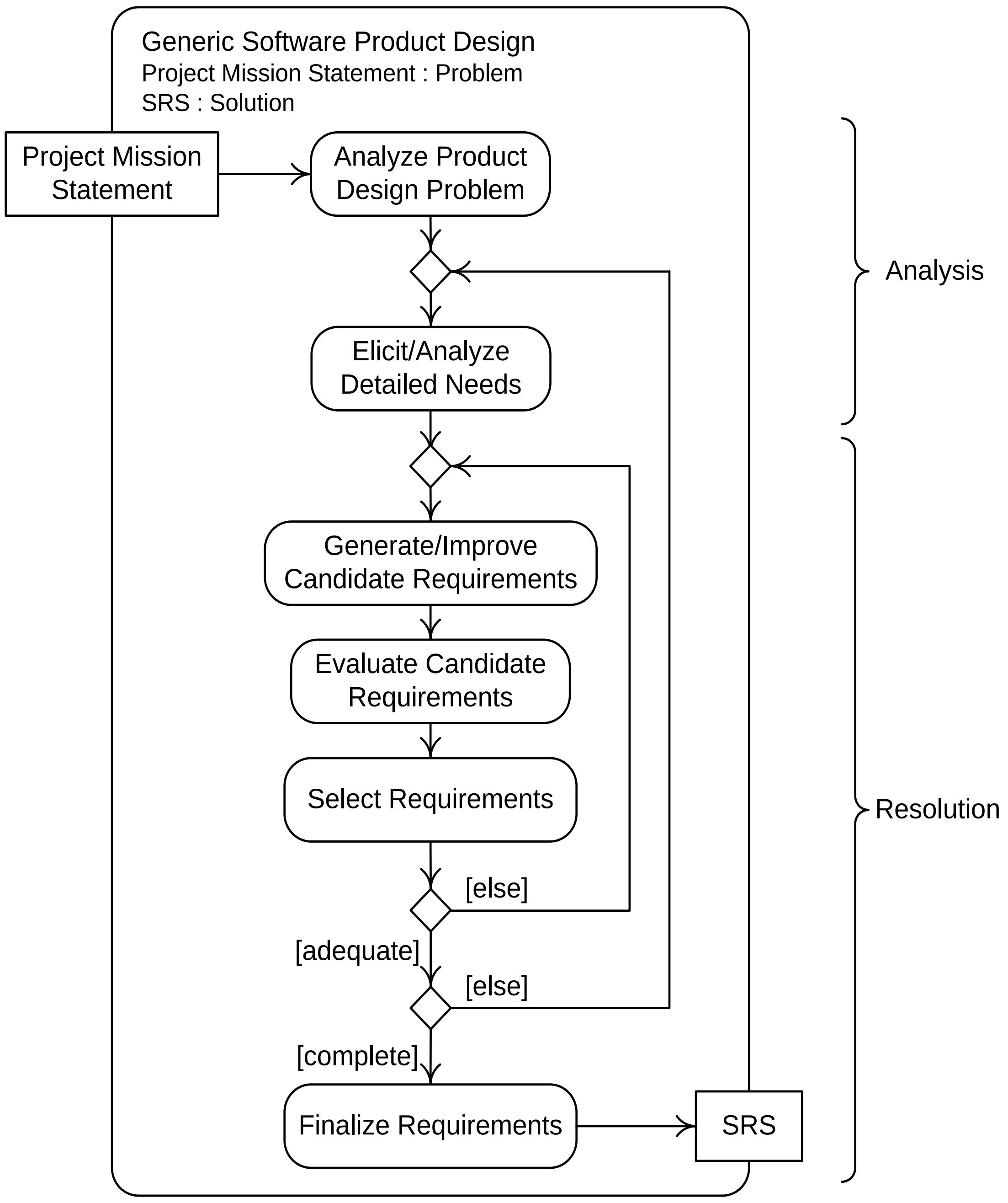
A Design Resolution Process



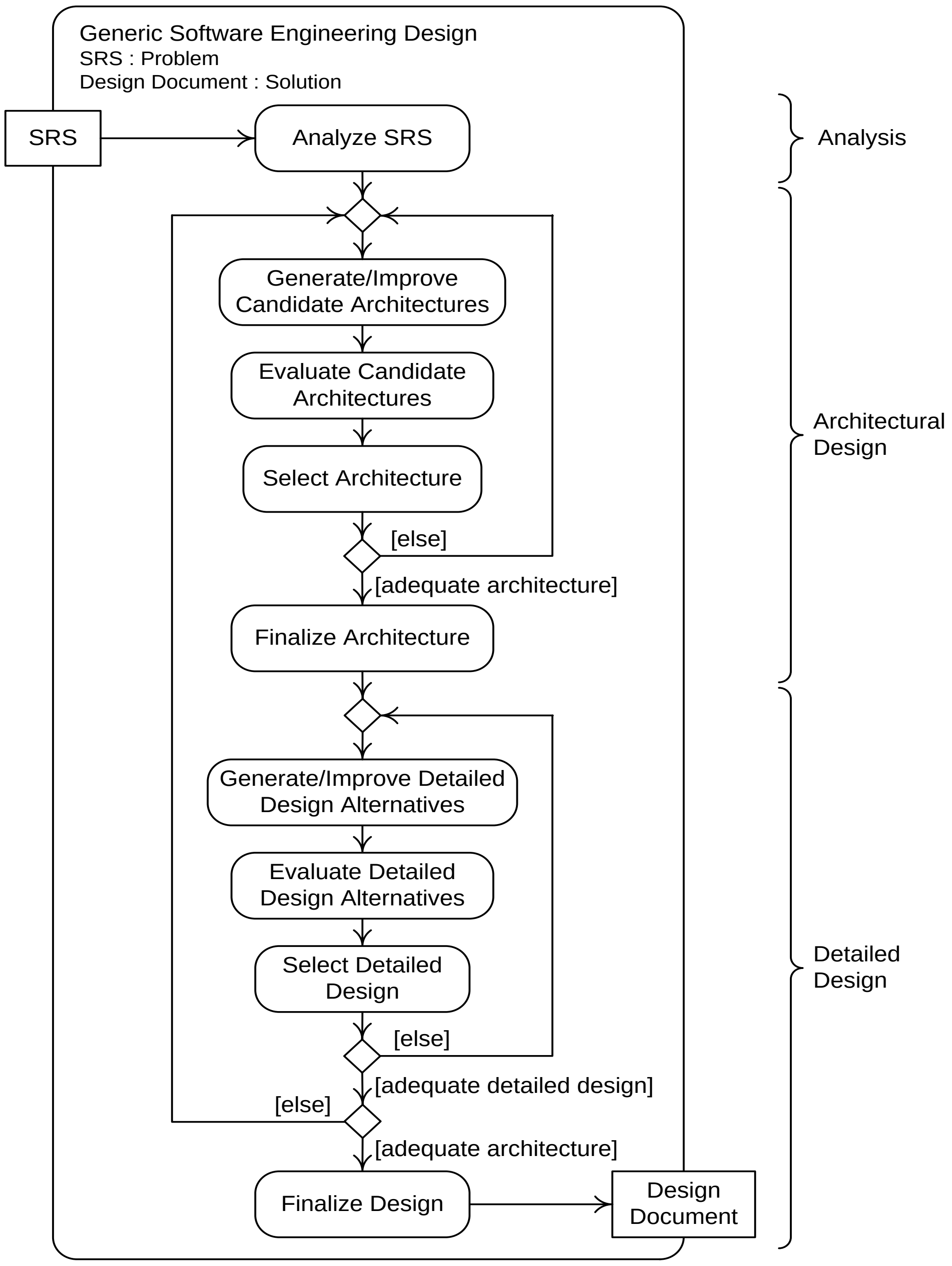
Design Process Characteristics

- Designers should generate many candidate solutions during the design process.
- The design process is highly iterative; designers must frequently reanalyze the problem and must generate and improve solutions many times.

A Generic Software Product Design Process


Generic Software Product Design
Project Mission Statement : Problem
SRS : Solution
Project Mission Statement
Analyze Product Design Problem
Elicit/Analyze Detailed Needs
Generate/Improve Candidate Requirements
Evaluate Candidate Requirements
Select Requirements
[else]
[adequate]
[else]
[complete]
Finalize Requirements
SRS
Analysis
Resolution

A Generic Software Engineering Design Process


Generic Software Engineering Design
SRS : Problem
Design Document : Solution
SRS
Analyze SRS
Generate/Improve Candidate Architectures
Evaluate Candidate Architectures
Select Architecture
[else]
[adequate architecture]
Finalize Architecture
Generate/Improve Detailed Design Alternatives
Evaluate Detailed Design Alternatives
Select Detailed Design
[else]
[adequate detailed design]
[else]
[adequate architecture]
Finalize Design
Design Document
Analysis
Architectural Design
Detailed Design

Software design management

Design require management

- Software development is complex, expensive, time consuming done by groups of people.
- If it is simply allowed to “happen,” the result is chaos.
- Chaos is avoided when software development is managed.
- Software development must be planned, organized, and controlled, and the people involved must be led.
- There are at least two sorts of business activities that must be managed.
- **Operations** are standardized activities that occur continuously or at regular intervals.
    - Hiring and performance review
    - Payroll operations
    - Shipping and receiving operations
- A **project** is a one-time effort to achieve a particular, current goal of an organization, usually subject to specific time or cost constraints.
    - Efforts to introduce new products,
    - Redesign tools and processes to save money
    - restructure an organization in response to business needs.

Project planning activities

- Software development clearly fits project management:
- **Planning**—Formulating a scheme for doing a project.
- **Organizing**—Structuring the organizational entities involved in a project and assigning them responsibilities and authority.
- **Staffing**—Filling the positions in an organizational structure and keeping them filled.

Design require management

- **Tracking**—Observing the progress of work and adjusting work and plans accordingly.
- **Leading**—Directing and helping people doing project work.

Project Planning

- The first step in working out a project plan is to determine how much work must be done and the resources needed to do it.
- **Estimation** is calculation of the approximate cost, effort, time, or resources required to achieve some end.
    - Mostly begin by estimating the size of work products such as source code, documentation, and so forth, and then deriving estimates of effort, time, cost, and other resources.
- A **schedule** specifies the start and duration of work tasks, and often the dates of milestones.
- A **milestone** is any significant event in a project.
- A **risk** is any occurrence with negative consequences.
- **Risk analysis** is an orderly process of identifying, understanding, and assessing risks.
- The final portion of the project plan is a specification of various rules governing work. Such rules fall into the following categories:
    - *Policies and Procedures*
    - *Tools and Techniques*

Project organization

- There are many ways to organize people into groups and assign them responsibilities and authority
    - Organizational structure.
- There are also many ways for people in groups to interact, make decisions, and work together
    - Team structures.

Organizational structure

- **Project organization:** Groups might be responsible for carrying projects from their inception through completion
- **Functional organization**: Groups might be responsible for just part of the project, such as design or coding or testing

Team structure

- *hierarchical team:* A team might have a leader who makes decisions, assigns work and resolves conflicts.
- *Democratic team:* A team might attempt to make decisions, assign work, and resolve conflicts though discussion, consensus, and voting.

Project staffing

- An organizational structure has groups with roles that must be filled e.g. testing group.
- Project staffing is the activity of filling the roles designated in an organizational structure and keeping them filled with appropriate individuals.
    - Hiring and orienting new employees
    - career development guidance
    - opportunities through training and education
    - Evaluating their performance

Project tracking

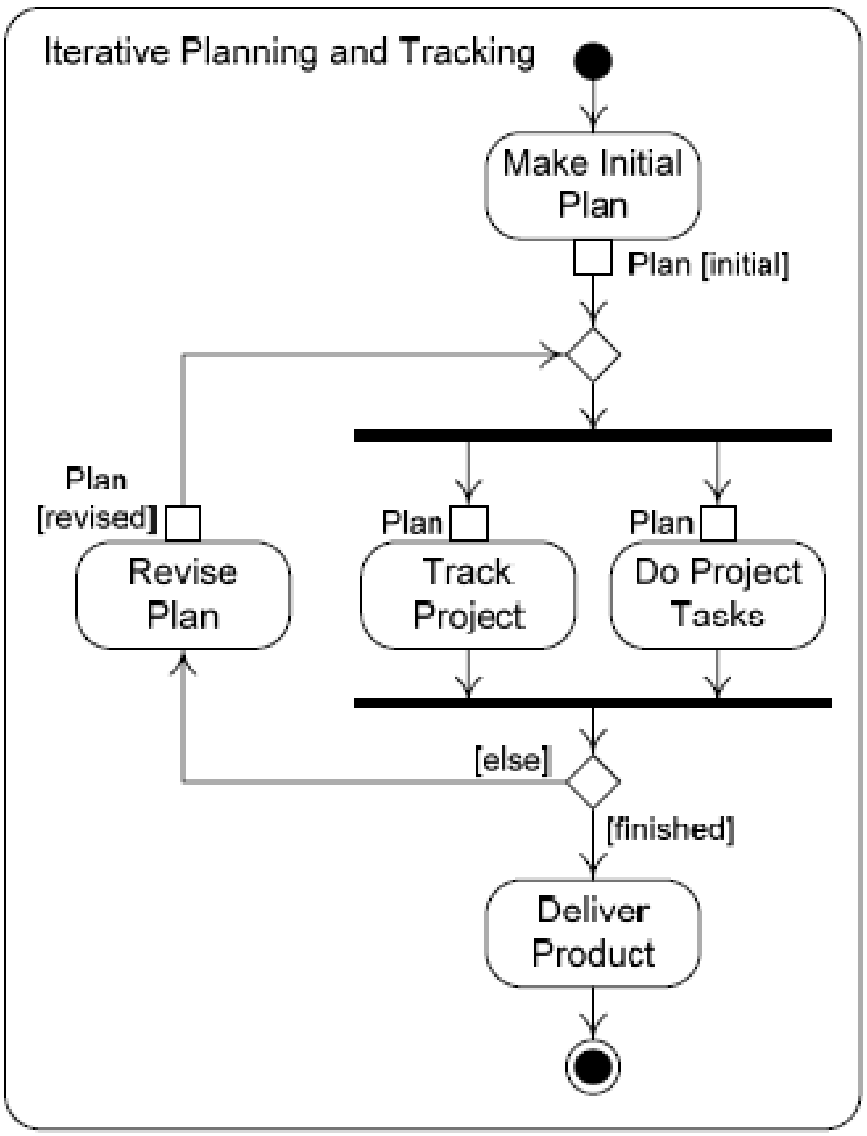
- Nothing ever goes exactly as planned, so it is essential to observe the progress of a project and adjust the work, respond to risks, and, if necessary, alter the plan.
- **Project Tracking**: Measuring and reporting the status of milestones, tasks and activities required in achieving the pre-defined project results
- Reasons for project tracking:
    - A task may simply take more or less time than expected.
    - Some of the rules governing the project may cause problems.

- The resources needed to accomplish tasks may not be as anticipated.
- Something bad may occur.
- Tracking is essential so that estimates, schedules, resource allocations, risk analyses, and rules can be revised.

Leading a project

- An adequate direction and support, a broad category of management responsibility called **leadership** is required for successful project.
- Merely directing people does not guarantee success.
  - People also need a congenial work environment, an emotionally
  - socially supportive workplace,
  - Make them feel that they are doing something important

Iterative Planning and Tracking



Summary

- Project management
- Project management activities
- Iterative planning and tracking

Week 5 Summery

Software design management

Project planning activities

- Software development clearly fits project management:
- **Planning**—Formulating a scheme for doing a project.
- **Organizing**—Structuring the organizational entities involved in a project and assigning them responsibilities and authority.
- **Staffing**—Filling the positions in an organizational structure and keeping them filled.

Design require management

- **Tracking**—Observing the progress of work and adjusting work and plans accordingly.
- **Leading**—Directing and helping people doing project work.

Design Project Decomposition

- Most aspects of project management depend on the work to be done and, in particular, on how it is decomposed.
- An obvious way to break down a design project is to divide the work according to the generic design processes discussed in the last section.

Design Project Decomposition

Work Phase		**Typical Work Products**
Product Design	Analysis: Design Problem	Statement of interested parties, product concept, project scope, markets, business goals Models (of the problem) Prototypes (exploring the problem)

	Analysis: Detailed Needs	Client surveys, questionnaires, interview transcripts, etc. Problem domain description Lists of needs, stakeholders Models (of the problem) Prototypes (exploring needs)
	Resolution: Product Specification	Requirements specifications Models (of the product) Prototypes (demonstrating the product)

Work Phase		**Typical Work Products**
Engineering Design	Analysis	Models (of the engineering problem) Prototypes (exploring the problem)
	Resolution: Architectural Design	Architectural design models Architectural design specifications Architectural prototypes
	Resolution: Detailed Design	Detailed design models Detailed design specifications Detailed design prototypes

Design Project Planning

- The initial project plan focuses on design problem analysis, with only rough plans for the remainder of the work.
- So, plan will be revised before product design resolution, engineering design analysis, and engineering design resolution.
- Initial estimates of effort, time, and resources are as precise as possible, based on the work products to be completed.
- These estimates may be based on data about work done in the past or an analogy with similar jobs with which the planners are familiar.
- The estimates are then used to block out:
    - An initial schedule
    - Allocate resources
    - Analyze risks
    - Set the rules guiding the project.
- Product analysis work is tracked against the initial plan.
- Ideally, problem analysis is complete when it is time to revise the plan, since planning the product design resolution phase requires this information.
- The plan may be altered during tracking to make this happen.
- A revised plan prepared before the product design resolution phase should have much more accurate:
    - Estimates
    - Schedule
    - Resource allocations
    - Risk analysis
    - Iterative planning and tracking continues through the engineering design with more details added each time the plan is revised.

Design Project Organization

- Design teams should be formed with responsibility of:
    - Design as a whole
    - Each major phase
    - Each sub-phase
    - Production of the various work products.
    - E.g, a large company might have a division responsible for requirements and design.

Design Project Staffing

- Organizations are staffed to fit the decomposition of design work.
- Projects need staff to:
    - Elicit and analyze needs
    - Create prototypes
    - Model systems
    - Create product designs
    - Write requirements specifications
    - Design user interaction
    - Make high-level and low-level engineering designs
    - Quality assurance.

Design Project Leadership

- Leading a design problem needs extra skills:
    - Visionary
    - Creative
    - Anticipate changes
    - Experience

Design as project driver

- Design work extends from the start of a software development project to the coding phase, and it recurs during maintenance.
- Two major products of software design, the SRS and the design document, are the blueprints for coding and testing.
- So, design is the driving activity in software development.
- By the time the software design is complete, enough information is available to make accurate and complete plans for the coding and testing phases.
- Good design work early in the life cycle is crucial for software development project success.

Context of Software Product Design

Products and markets

- Organizations create products for economic gain.
- Product development is very expensive, so an organization must be careful to create products that it can actually sell or use.
- A **market** is a set of actual or prospective customers who need or want a product, have the resources to exchange something of value for it, and are willing to do so.

Importance of market

Organizations study markets to:

- Choose which markets to sell to (**target markets**)
- Choose what products to develop
- Determine product features and characteristics
- Thus, the sorts of products that an organization decides to develop ultimately depend on the target markets to which it hopes to sell the products.

Products influence design

- A lot of what happens during product design depends on what sort of product is being designed.
- A product’s characteristics influences:
  - The decision to develop the product;
  - The resources and time devoted to product development;
  - The techniques
  - Methods, and tools used to develop the product;
  - Distribution and support of the final product.

Categorizing products

- Products fall into different categories along several dimensions.
- A **product category** is a dimension along which products may differ.
  - Target market size
  - Product line novelty
  - Technological novelty
- A **product type** is a collection of products that have the same value in a particular product category.

Target Market Size

Target market size is the number of customers a product is intended to serve.

Type	Description	Examples
Consumer	Mass consumer markets	Word processors, spreadsheets, accounting packages, computer games, operating systems
Niche Market	More than one customer but not a mass consumer market	Programs for configuration management, shipyard management, medical office records management, AquaLush
Custom	Individual customers	Systems written for one part of a company by another part, space shuttle software, weapons software

Categorizing products

- Designers of custom and niche-market products designers can identify needs and desires for a product as compared to consumer products.
- Designing consumer products is easy than designing niche-market products which is easy than designing custom products.
- Competitors are important when designing consumer and niche-market products, but this is not the case designing custom products.
- Different aspects of product design are more or less important in these different categories.
  - Consumer products place a premium on attractive user interface design.
  - Functionality is usually more important for custom and niche-market products.

Product Line Novelty

- **Product line novelty** is how “new” a product is in relation to other products in current product line.

Type	Description	Examples
New	Different from anything else in the product line	Tax preparation product in a line of accounting products, AquaLush
Derivative	Similar to one or more existing products in the product line	Database management system for individual users in a line of systems for corporate users
Maintenance Release	New release of an existing product	Third release of a spreadsheet

Product Line Novelty

- Maintenance releases pose higher constraints on designers than derivative products which pose higher constraints than designing new products.
- Designing a new product is a very big job, designing a derivative product is a smaller but still formidable task, and designing a new release may be relatively easy.

Technical Novelty

- Technical novelty means “how much new technology” is incorporated in a product, w.r.t. target market at a particular time.

Type	Description	Examples
Visionary Technology	New technology must be developed for the product	Mobile computing (2000), Wearable automatic lecture note-taker (2004)
Leading-Edge Technology	Proven technology not yet in widespread use	Peer-to-peer file-sharing products (2002), AquaLush (2006)
Established Technology	Widely used, standard technology	Products with graphical user interfaces (2000)

Technical Novelty

- Designing products with visionary or leading-edge technologies is most difficult kind of software design.
    - Hard to figure out what clients want.
    - Whether products with new technology will attract customers
- Products with visionary technology may never be built if efforts to develop the new technology fail.
- Even if they are a technological success, they may still fail in the marketplace if customers don’t like the new technology.
- Leading-edge and established technology products are more likely to succeed.

Week 6 Summery

Project Mission Statement

- A **project mission statement** is a document that defines a development project's goals and limits.
- The project mission statement plays two important roles:

1. **Launches a development project**
2. **States the software design problem**

- The project mission statement is the main input to the product design process.

Project Mission Statement Template

1. Introduction
2. Product Vision and Project Scope
3. Target Markets
4. Stakeholders
5. Assumptions and Constraints
6. Business Requirements

Introduction

- The introduction contains background information to provide context.
- Information about the major business opportunity that the new product will take advantage of and the product operating environment.

Product Vision and Project Scope

- A **product vision statement** is a general description of the product's purpose and form.
- The **project scope** is the work to be done on a project.
    - Often only part of the product vision.
    - May list what will *not* to be done as well as what will be done.

Target Market

- Upper management chooses the target market segments for a new product or release during product planning.
- Target markets are those market segments to which the organization intends to sell the new product. Market segments determine users, features, competitors, and so forth**.**

Stakeholders

- A **stakeholder** is anyone affected by a product or involved in or influencing its development.
    - Product users and purchasers
    - Developers and their managers
    - Marketing, sales, distribution, and product support personnel
    - Regulators, inspectors, and lawyers
- Developers must know the target market and stakeholders to build a product satisfying stakeholders' needs.

Assumptions and Constraints

- An **assumption** is something that developers take for granted.
    - Feature of the problem
    - Examples: target deployment environments, levels of user support
- A **constraint** is any factor that limits developers.
    - Restriction on the solution
    - Examples: cost and time limits, conformance to regulations

Business Requirements

A **business requirement** is a statement of a client or development organization goal that a product must meet.

- Time, cost, quality, or business results
- Should be stated so that it is clear whether it is satisfied (quantitative goals)
- Broad goals related to business, not detailed product specifications

Needs Elicitation

Needs Vs Requirements

- Stakeholder needs and desires define the product design *problem*.
- Requirements specify the product design *solution*.
- Needs and requirements statements are similar, but the heart of product design is moving from needs to requirements.
  - Conflicting needs and desires
  - Tradeoffs (needs and constraints)
  - Ways of satisfying needs and desires

Needs Elicitation Challenges

- Stakeholders often cannot explain their work, or articulate their needs and desires.
- Needs and desires can only be understood in a larger context that
- includes understanding the problem domain.
- Stakeholders make mistakes, leave things out, and are misleading.
- Stakeholders often don't understand the capabilities and limitations of technology.
- Designers are faced with a flood of information, often contradictory, incomplete, and confusing.

How to tackle Elicitation Challenges

- Designers must obtain information from stakeholders in a systematic fashion using several elicitation techniques and must document and analyze the results to ensure that needs and desires are understood correctly and completely.
- The main way to organize requirements elicitation is to work from the top down through levels of abstraction. Organization within each level of abstraction is achieved by focusing on particular product aspects, which depend on the product itself.

Elicitation Heuristics

- Learn about the problem domain first.

If designers don’t understand the problem domain, they need to elicit, document, and analyze information

about it *before* eliciting needs.

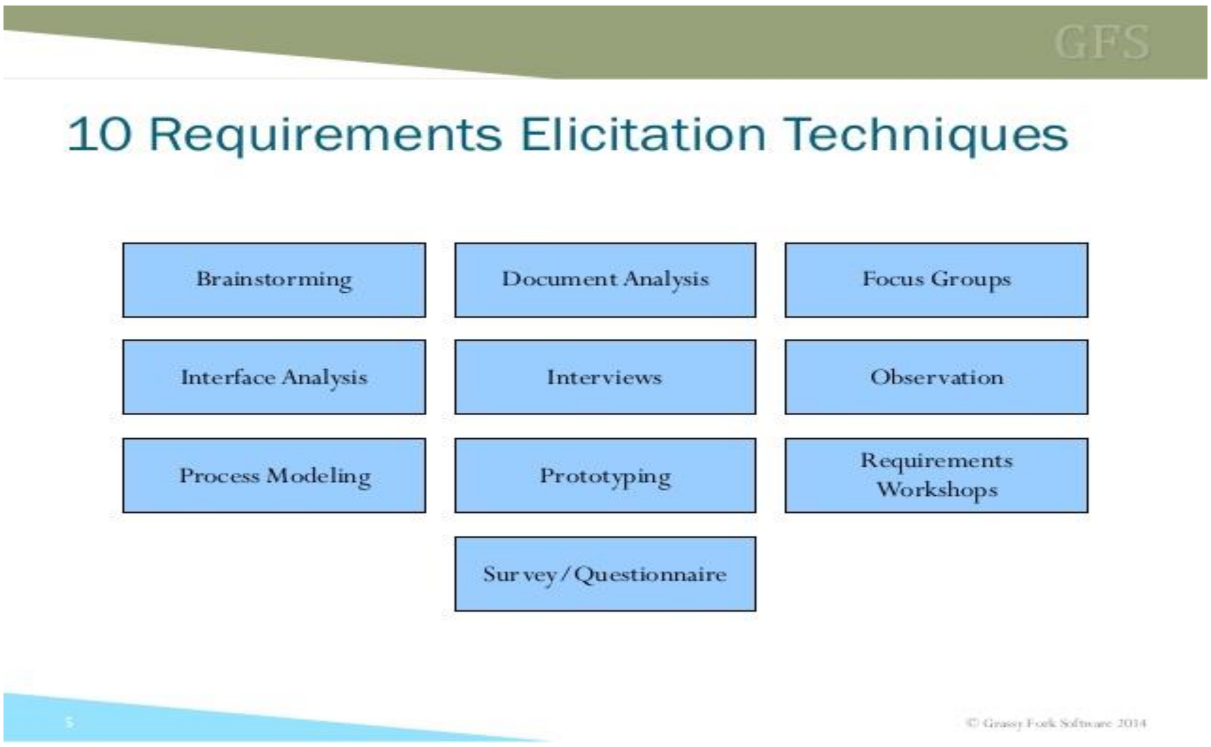
- Determine stakeholder goals as the context of stakeholder needs and desires

What a stakeholder needs and wants is a consequence of his or her goals. For example, a user may need a product to record sample data. Why would the user need this? Because the user’s goal is to monitor a manufacturing process by sampling and analyzing its output

- Study user tasks.

For example, suppose users currently collect and measure samples by hand, record the data in a log book, use a calculator to compute statistics, enter the results on a paper graph, and study the graph to see if the process is running properly.

Elicitation Techniques



GFS

10 Requirements Elicitation Techniques

Brainstorming
Document Analysis
Focus Groups
Interface Analysis
Interviews
Observation
Process Modeling
Prototyping
Requirements Workshops
Survey/Questionnaire

© Grassy Fork Software 2014

Elicitation Techniques

- **Interviews-** question and answer session during which one or more designers ask questions of one or more stakeholders or problem-domain experts
- Most important technique for recording responses
- **Observation-** Many products automate or support work done by people, so designers need to understand how people do their work to design such products
- Especially useful for eliciting derivative product and maintenance release needs because it can reveal many opportunities for product improvement
- **Focus Groups —**Is an informal discussion among six to nine people led by a facilitator who keeps the group on topic. Focus groups consist of stakeholders or stakeholder representatives who discuss some aspect of the product.
- Main technique of obtaining needs for consumer products, especially new products and those with visionary or leading-edge technologies.
- **Prototype** — A working model of part or all of a final product. Prototypes provide a useful basis for conversations with stakeholders about features, capabilities, and user interface issues such as interaction protocols.
- Especially useful for products with visionary technology because they help people understand what a product with the new technology will be like.
- **Questionnaires –** It is efficient technique to elicit information from many people.
- Close ended questions

Easier to analyze and range of possible responses is well-understood

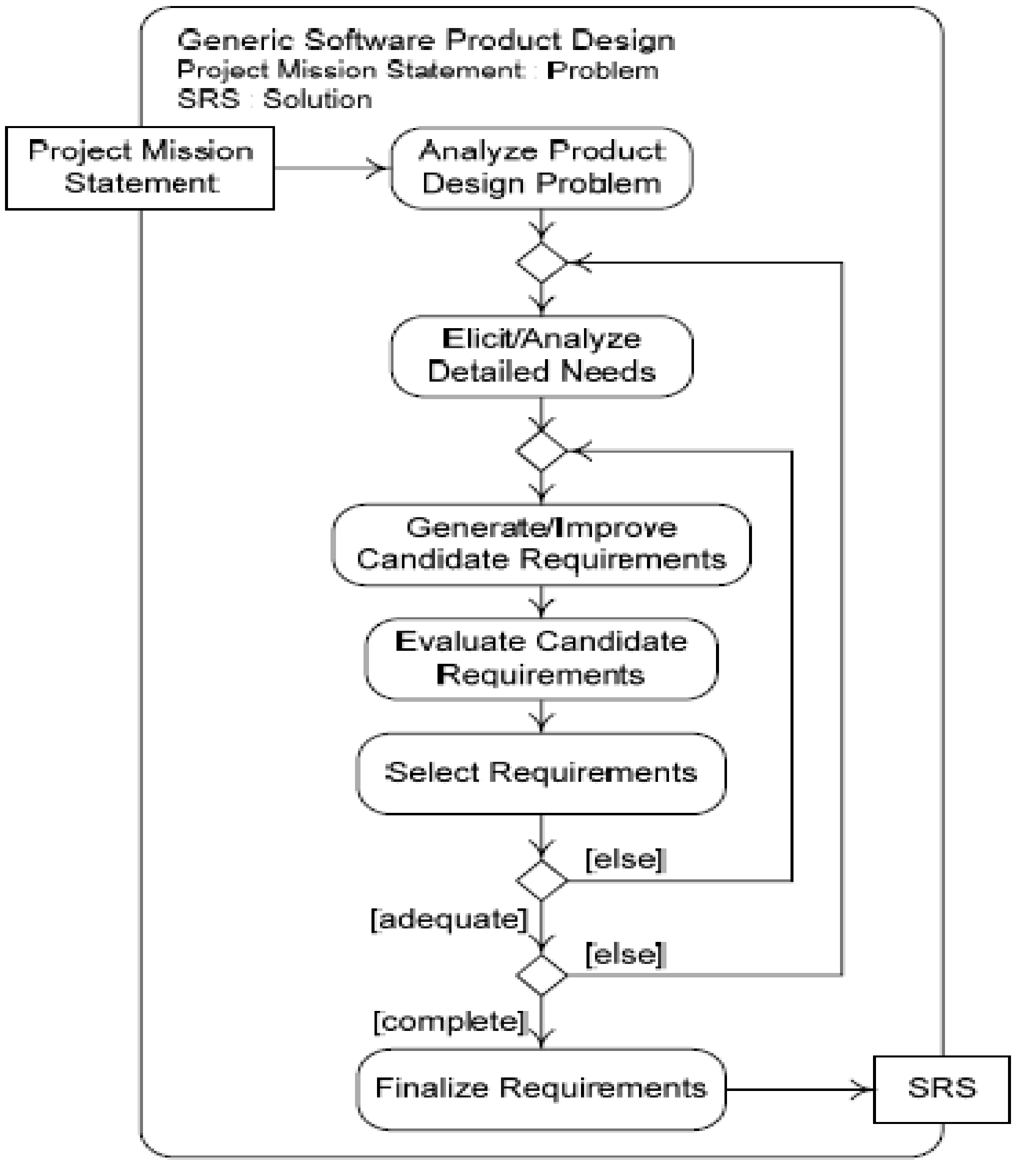
- Open ended questions

It includes detailed responses and relatively harder to understand

Week 7 Summery

Product Design Process Overview

Introduction


Generic Software Product Design
Project Mission Statement : Problem
SRS : Solution
Project Mission Statement
Analyze Product Design Problem
Elicit/Analyze Detailed Needs
Generate/Improve Candidate Requirements
Evaluate Candidate Requirements
Select Requirements
[else]
[adequate]
[else]
[complete]
Finalize Requirements
SRS

Steps of Software Design Process

There are six steps of software design process:

- Understanding of Design problem
- Elicit/Analyze Detailed Needs
- Generating/Improve Candidate Requirements
- Evaluate Candidate Requirements

- Select Requirements
- Finalize Requirements

1. **Understanding of Design Problem**

   The nature of this task depends on whether there is an adequate project mission statement

   A good project mission statement defines the product design problem, so the designers need only study the mission statement and research any parts of it they do not understand.

2. **Elicitation of Detailed Needs**

   Second step in design process is comprised of eliciting and analyzing detailed needs

Designers needs to learn much more about stakeholder needs and desires, especially those of users and purchasers that will meet its business requirements.

3,4,5-Improvement, Evaluation and Selection of Requirements

- Third step proceeds by generating and refining requirements, therefore fulfilling the needs determined during analysis
- Once alternative requirements are generated and stated, they are evaluated in fourth step
- In fifth step, requirements are selected on basis of evaluation

6. Requirements Finalization

- The last step of the software product design process is to finalize the SRS

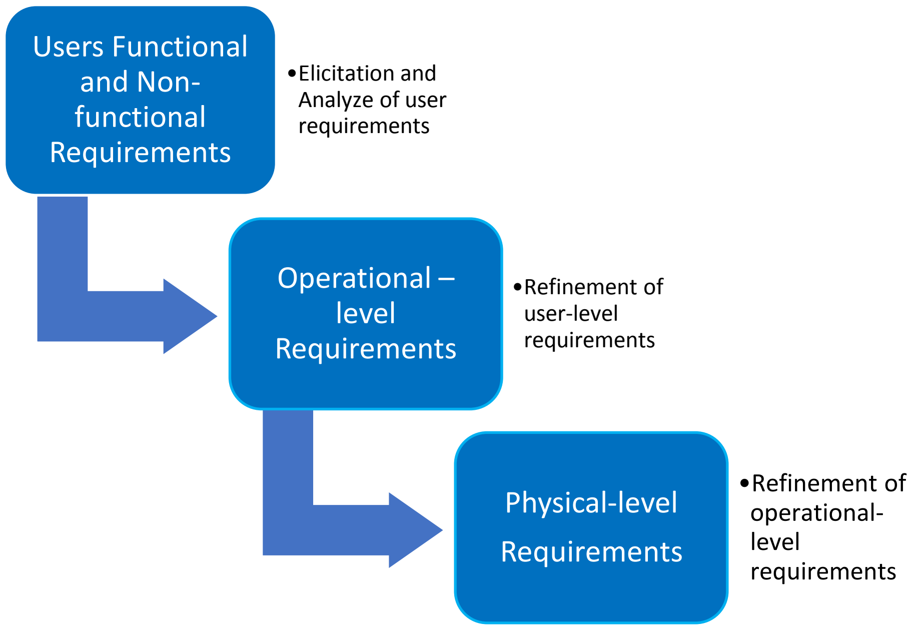
So, we start with Project Mission statement that act as input to Design process and the outcome of this process is SRS (Software Requirement Specification).

Outer iteration in Fig 1 reflects refinement activity of product details specification.

Product Design Process: A Top-Down Process

- Product design resolution sets technical requirements at a high level of abstraction and then refine them until all product details are specified.
- During this process, user-level needs are elicited and analyzed first, and user-level functional, data , and non-functional requirements are generated , refined, and evaluated until they are adequate
- The user-level requirements provide an abstract solution to the design problem. They are then refined to produce operational–level requirements
- Operational-level requirements are refined to produce physical-level requirements

Refinement Process


Users Functional and Non-functional Requirements
•Elicitation and Analyze of user requirements
Operational – level Requirements
•Refinement of user-level requirements
Physical-level Requirements
•Refinement of operational-level requirements

Refinement is complete when these 3 level requirements are specified

Product Design Process: A User Centered Approach

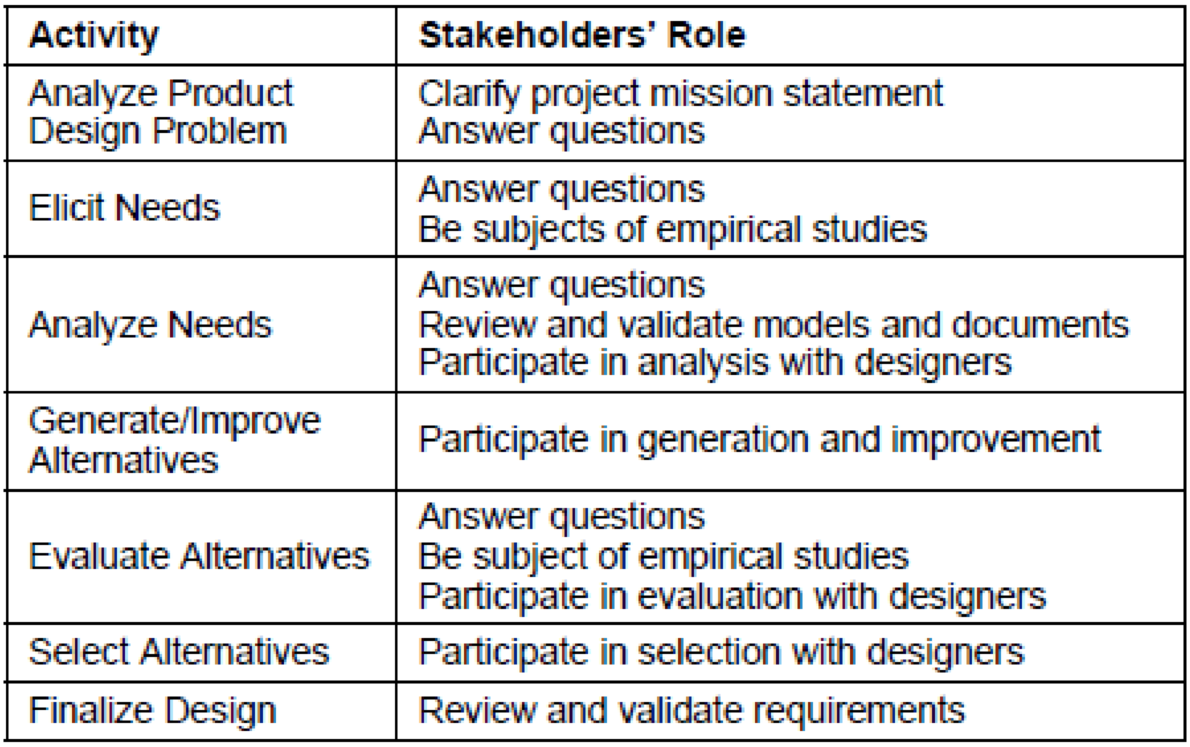
User-centered design comprises the following three principles:

- *Stakeholder Focus* – Determine the needs and desires of all stakeholders (especially users), and involve them in evaluating the design and perhaps even in generating the design
- *Empirical Evaluation* – Gather stakeholder needs and desires and assess design quality by collecting data rather than by relying on guesses.
- *Iteration* – Improve designs repeatedly until they are adequate.

Terminologies

- **Requirement Elicitation –** Collecting stakeholder needs and desires is called *requirements elicitation* , or *needs identification* or *needs elicitation*
- **Requirements Analysis –** Understanding stakeholder needs is called *needs analysis* or *requirements analysis*
- **Requirements Validation –** Confirming with stakeholder that a product design satisfies their needs and desires is called *requirements validation* or just *validation.*

Role of Stakeholders



Activity	Stakeholders' Role
Analyze Product Design Problem	Clarify project mission statement Answer questions
Elicit Needs	Answer questions Be subjects of empirical studies
Analyze Needs	Answer questions Review and validate models and documents Participate in analysis with designers
Generate/Improve Alternatives	Participate in generation and improvement
Evaluate Alternatives	Answer questions Be subject of empirical studies Participate in evaluation with designers
Select Alternatives	Participate in selection with designers
Finalize Design	Review and validate requirements

Needs Documentation and Analysis

Formulating & Organizing Documentation

- The raw data collected from interviews, observation, focus groups, workshops, competitive studies and so forth needs to be sorted, stated clearly, and organized.
- First step is to divide the data into two categories:
- ❑ Data about the problem domain
- ❑ Data about Stakeholders' goals, needs, and desires

Documenting the Problem Domain

- Data about the problem domain can be further categorized and grouped to form an organized set of notes.
- ❑ **Problem Domain Glossary** is a useful tool in understanding the domain . Most problem domains have their own terminology that designers must learn
- ❑ **Organization Chart** can be made to display data about the stakeholders' organization

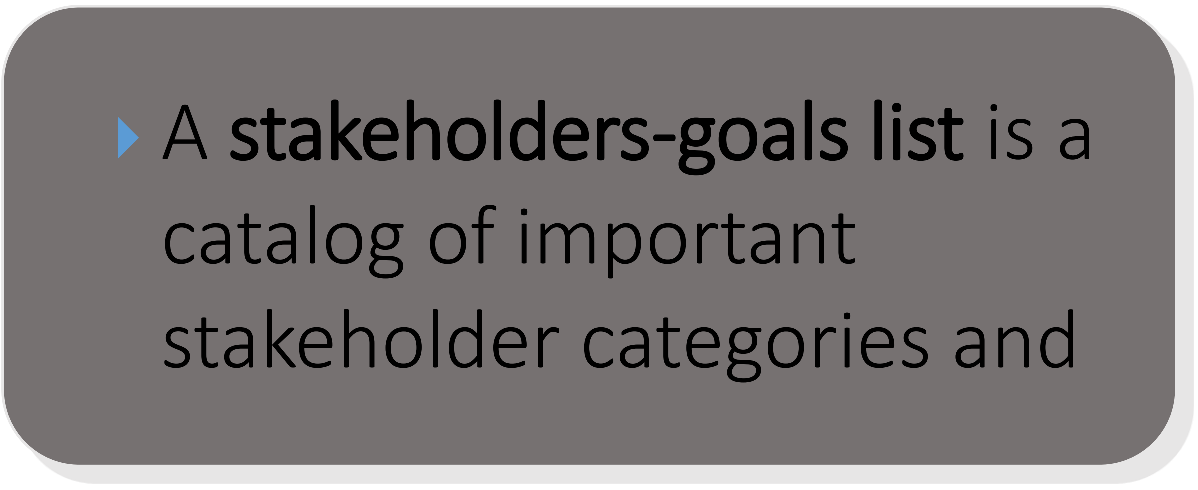
Cont.. UML Activity Diagrams

- ❑ **UML Activity Diagrams** are useful tools for organizing and documenting problem domain information about business processes or user processes.

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. Data about processes obtained from interviews, observation, focus groups, or document studies can be represented in a activity diagram much better than in text.

Documenting Goals, Needs and Desires

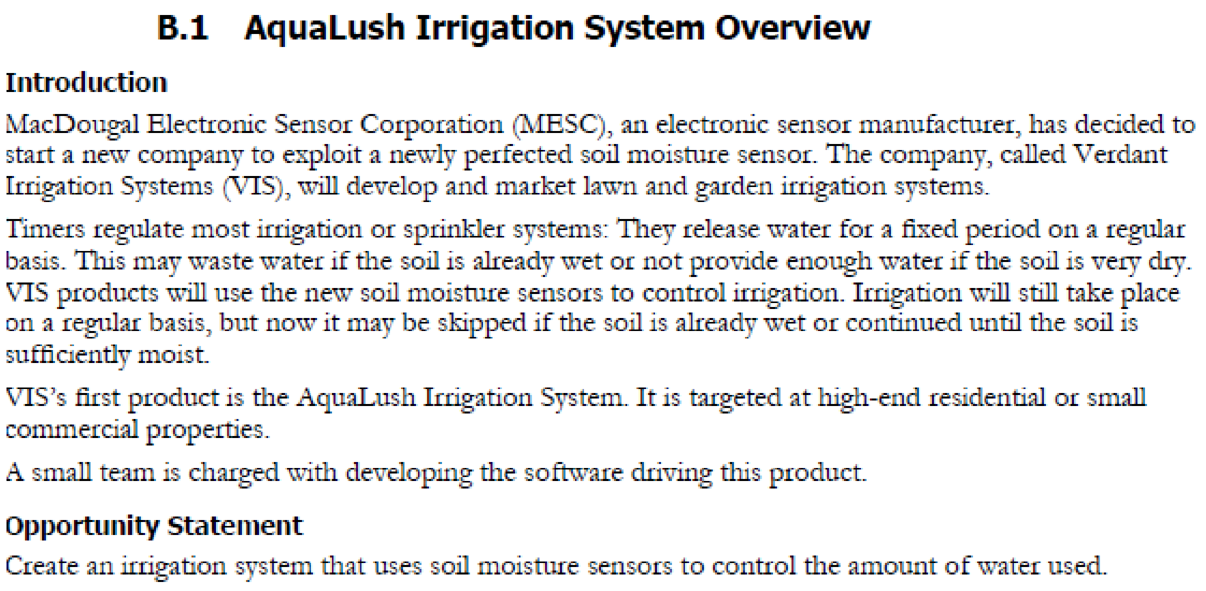
- Raw data about stakeholders' goals , needs and desires can be organized into two lists:
- ❑ A stakeholders-goal list
- ❑ Needs list



- A **stakeholders-goals list** is a catalog of important stakeholder categories and

Stakeholders Groups are based on roles, not individuals.

AquaLush Case Study



B.1 AquaLush Irrigation System Overview

Introduction

MacDougal Electronic Sensor Corporation (MESC), an electronic sensor manufacturer, has decided to start a new company to exploit a newly perfected soil moisture sensor. The company, called Verdant Irrigation Systems (VIS), will develop and market lawn and garden irrigation systems.

Timers regulate most irrigation or sprinkler systems: They release water for a fixed period on a regular basis. This may waste water if the soil is already wet or not provide enough water if the soil is very dry. VIS products will use the new soil moisture sensors to control irrigation. Irrigation will still take place on a regular basis, but now it may be skipped if the soil is already wet or continued until the soil is sufficiently moist.

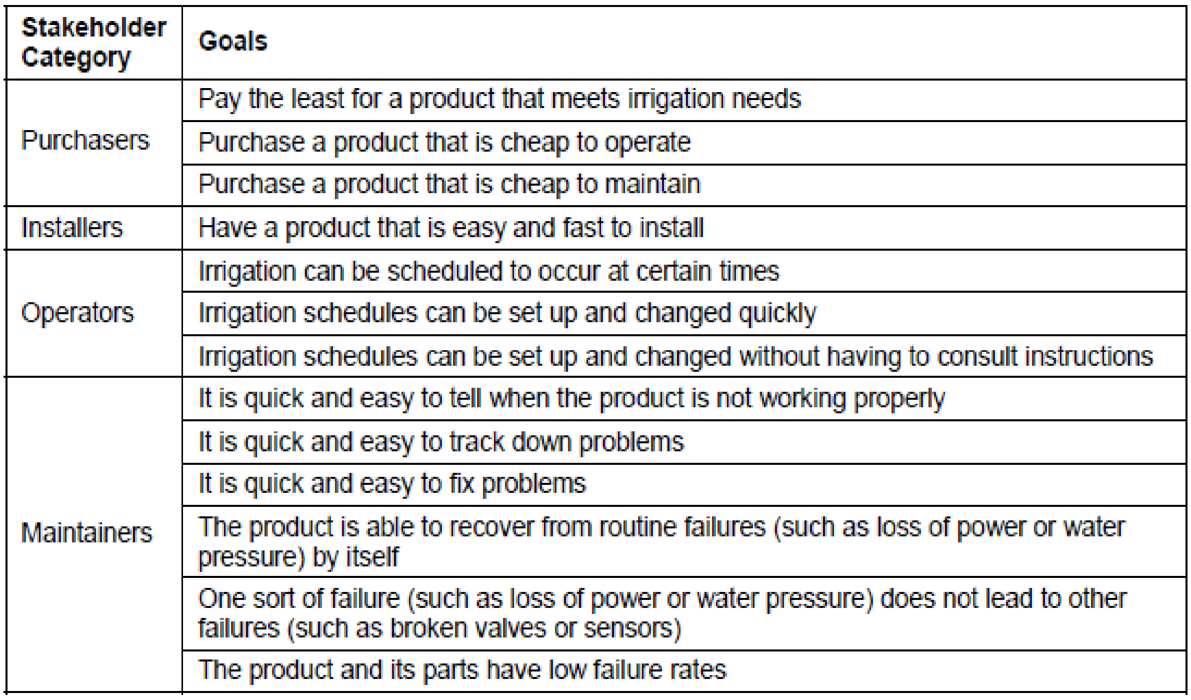
VIS's first product is the AquaLush Irrigation System. It is targeted at high-end residential or small commercial properties.

A small team is charged with developing the software driving this product.

Opportunity Statement

Create an irrigation system that uses soil moisture sensors to control the amount of water used.

Stakeholders' Roles : An Example



Stakeholder Category	Goals
Purchasers	Pay the least for a product that meets irrigation needs
	Purchase a product that is cheap to operate
	Purchase a product that is cheap to maintain
Installers	Have a product that is easy and fast to install
Operators	Irrigation can be scheduled to occur at certain times
	Irrigation schedules can be set up and changed quickly
	Irrigation schedules can be set up and changed without having to consult instructions
Maintainers	It is quick and easy to tell when the product is not working properly
	It is quick and easy to track down problems
	It is quick and easy to fix problems
	The product is able to recover from routine failures (such as loss of power or water pressure) by itself
	One sort of failure (such as loss of power or water pressure) does not lead to other failures (such as broken valves or sensors)
	The product and its parts have low failure rates

Table : Stakeholders' Goals List

- A **need statement** documents a single product feature, function, or property needed or desired by one or more stakeholders.

A need statement should

Name the stakeholder category or categories

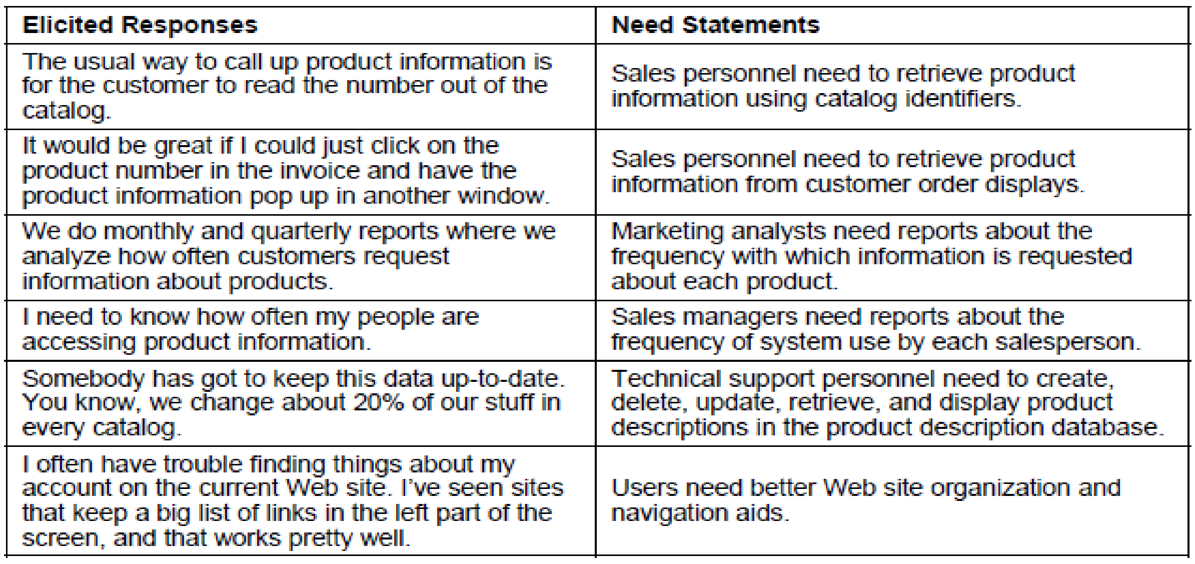
State one specific need

Be a positive declarative sentence

Often requires interpretation of raw data

- **Example:**

Here is a list of “Elicited Needs” and “Needs Statements”



Elicited Responses	Need Statements
The usual way to call up product information is for the customer to read the number out of the catalog.	Sales personnel need to retrieve product information using catalog identifiers.
It would be great if I could just click on the product number in the invoice and have the product information pop up in another window.	Sales personnel need to retrieve product information from customer order displays.
We do monthly and quarterly reports where we analyze how often customers request information about products.	Marketing analysts need reports about the frequency with which information is requested about each product.
I need to know how often my people are accessing product information.	Sales managers need reports about the frequency of system use by each salesperson.
Somebody has got to keep this data up-to-date. You know, we change about 20% of our stuff in every catalog.	Technical support personnel need to create, delete, update, retrieve, and display product descriptions in the product description database.
I often have trouble finding things about my account on the current Web site. I’ve seen sites that keep a big list of links in the left part of the screen, and that works pretty well.	Users need better Web site organization and navigation aids.

Table : Elicited Needs and Needs Statements

Problem Modeling

- Many kinds of models can represent the problem and help designers understand it.
- Models document the problem, can be reviewed with stakeholders for consistency.
- Many modeling notations and techniques are useful for analysis like
  - Various UML diagrams
  - Use case descriptions, user interface diagrams, dialog maps

Checking Needs Documentation

- **Correctness**—A statement is **correct** if it is contingent and accords with the facts.
- **Scope**—A goal or need is within the project scope if it can be satisfied using the planned features of the product created by the project.
- **Terminological Consistency -** Terminological consistency is using words with the same meaning and not using synonyms.

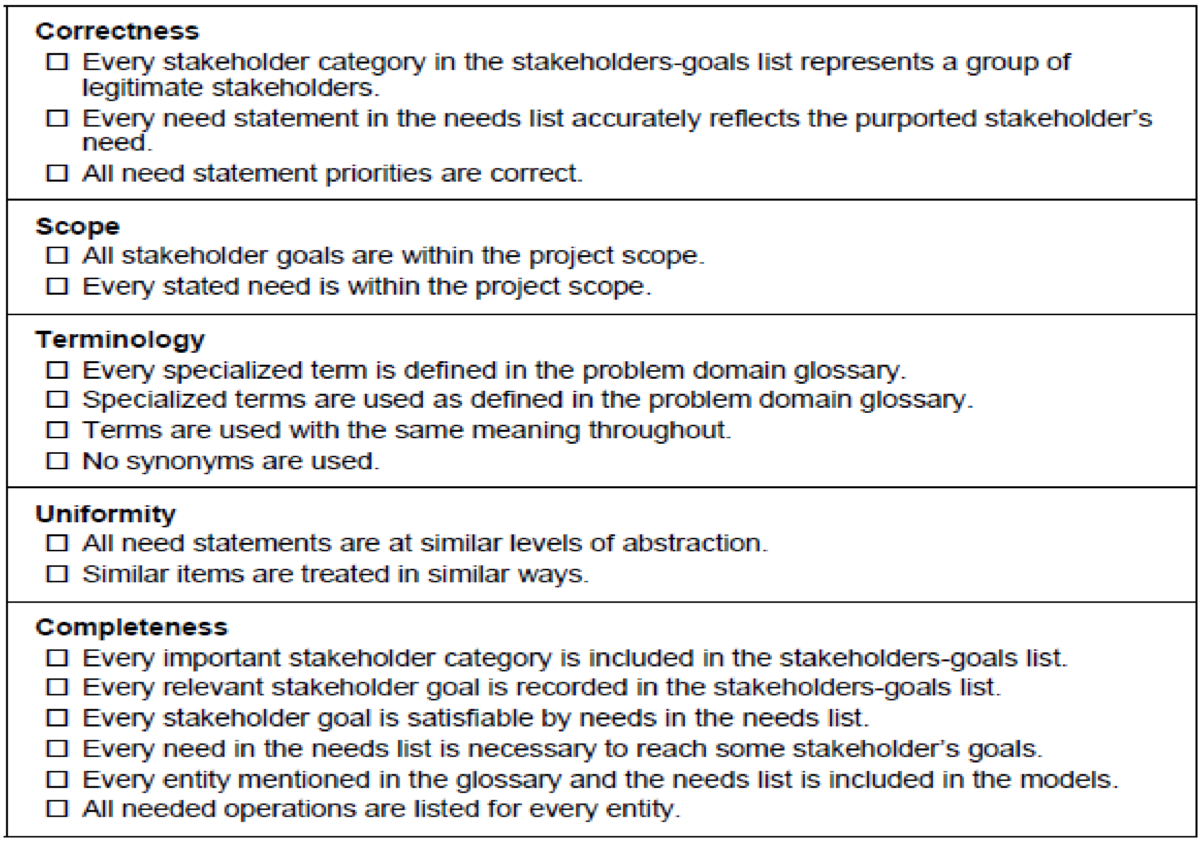
Continued

- **Uniformity**—A description has uniformity when it treats similar items in similar ways.
- **Completeness**—Documentation is complete when it contains all relevant material.

Review Activities

- Developers should use checklists
- Stakeholders should review documents

A Needs Documentation Checklist



Correctness
- ☐ Every stakeholder category in the stakeholders-goals list represents a group of legitimate stakeholders.
- ☐ Every need statement in the needs list accurately reflects the purported stakeholder's need.
- ☐ All need statement priorities are correct.

Scope
- ☐ All stakeholder goals are within the project scope.
- ☐ Every stated need is within the project scope.

Terminology
- ☐ Every specialized term is defined in the problem domain glossary.
- ☐ Specialized terms are used as defined in the problem domain glossary.
- ☐ Terms are used with the same meaning throughout.
- ☐ No synonyms are used.

Uniformity
- ☐ All need statements are at similar levels of abstraction.
- ☐ Similar items are treated in similar ways.

Completeness
- ☐ Every important stakeholder category is included in the stakeholders-goals list.
- ☐ Every relevant stakeholder goal is recorded in the stakeholders-goals list.
- ☐ Every stakeholder goal is satisfiable by needs in the needs list.
- ☐ Every need in the needs list is necessary to reach some stakeholder's goals.
- ☐ Every entity mentioned in the glossary and the needs list is included in the models.
- ☐ All needed operations are listed for every entity.

Week 8 Summery

Software Requirement Specification (SRS)

Software Requirement

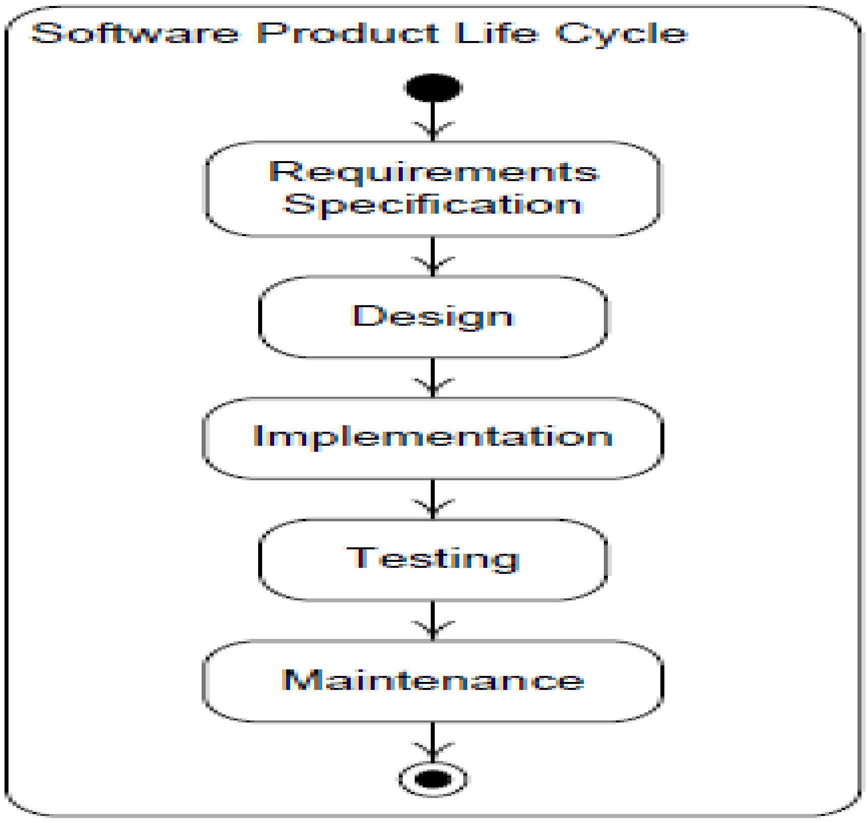
- A **software (product) requirement** is a statement that a software product must have a certain feature, function, capability, or property.
- Requirements are captured in **specifications**, which are simply statements that must be true of a product.

Software Requirement Specification

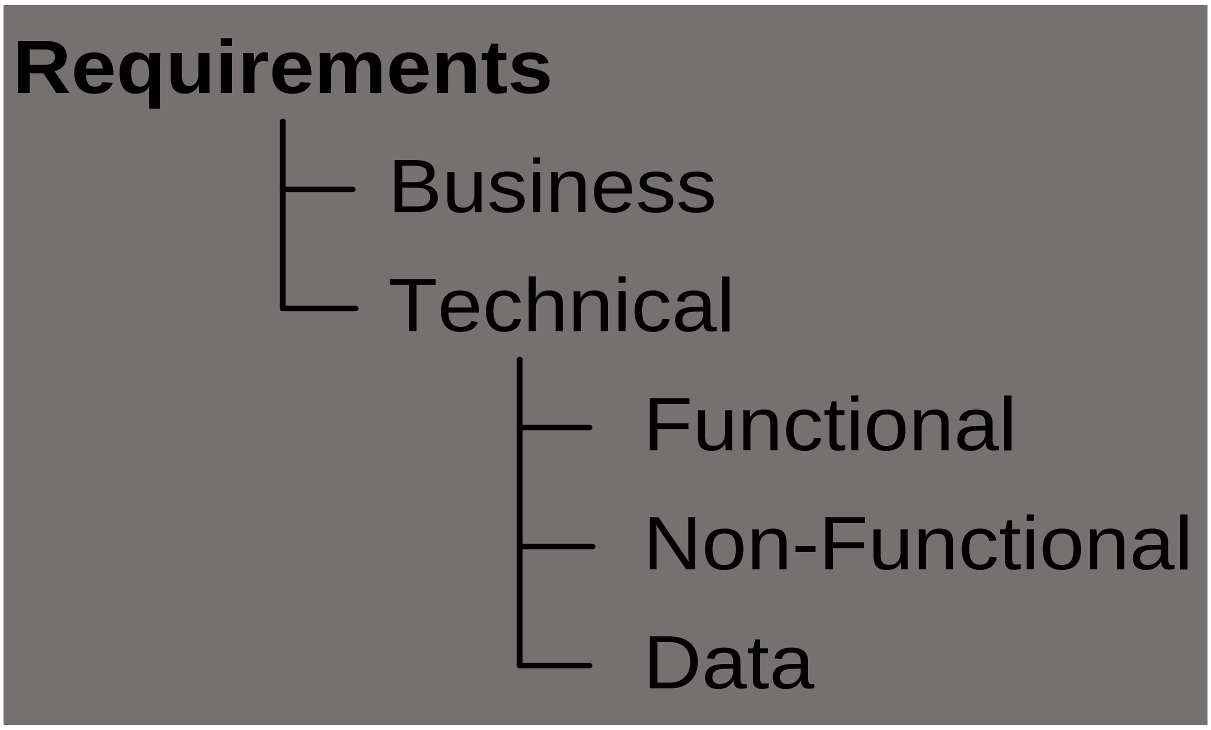
A **software requirements specification (SRS)** is a document cataloging all the requirements for a software product.

- This activity is aimed at finding out from the product's intended clients, and other interested parties, what they need and want from a software product. These needs and desires are translated into specifications of the functions, capabilities, appearance, behavior, and other characteristics of the software product.
- These specifications constitute the software product requirements, and they are recorded in a **software requirements specification (SRS)** document.

SRS in Software Product Life Cycle



Types of Requirements



Business Requirements

- Business requirements relate to a business' objectives, vision and goals and are often captured by business analysts who analyze business activities and processes.
- **Example:**

If a company's need is to track its field employees by means of an employee tracking system, the business requirements for the project might be described as:

"Implement a web and mobile based employee tracking system that tracks field employees and increases efficiency by means of monitoring employee activity, absenteeism and productivity. "

Functional Requirements

- A **functional requirement** is a statement of how a software product must map program inputs to program outputs.
- Functional requirements are specifications of the product's externally observable behavior, so they are often called **behavioral requirements**.

Examples:

- Display the name, total size, available space and format of a flash drive connected to the USB port.
- Upon request from managers, the system must produce daily, weekly, monthly, quarterly, or yearly sales reports in HTML format.

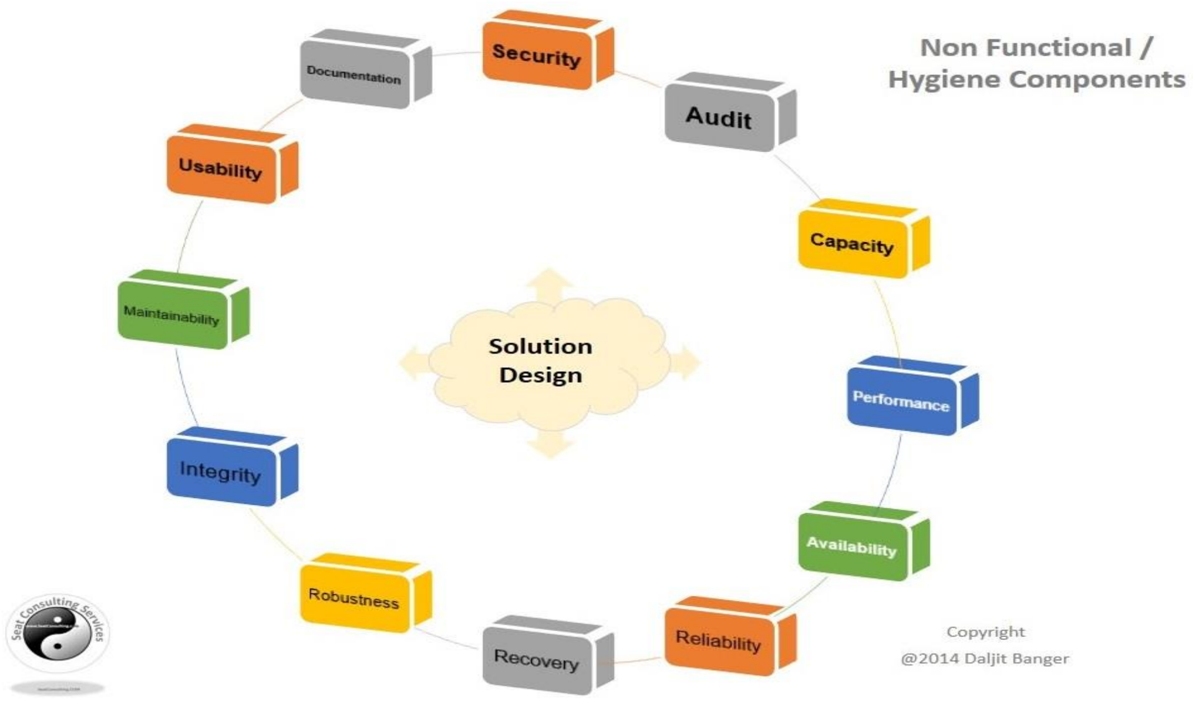
Non-Functional Requirements

- A **non-functional requirement** is a statement that a software product must have certain properties.
- Non-functional requirements are also called **non-behavioral requirements**.

The following statements are examples of non-functional requirements:

- The payroll system must process the payroll for all XYZ Corp employees in six hours or less.
- The system must run without failure for at least 24 hours after being restarted, under normal conditions of use.

Non-Functional Attributes



Data Requirements

- A **data requirement** is a statement that certain data must be input to, output from, or stored by a product.
- Data requirements describe the format, structure, type, and allowable values of data entering, leaving, or stored by the product.
- The Computer Assignment System must store customer names in fields recording first, last, and middle names.
- The system must display all times in time fields with the format *hh:mm:ss*,

where *hh* is a two-digit military time hour field, *mm* is a two-digit military time minutes field, and *ss* is a two-digit military time seconds field.

Levels of Abstraction

- A **user-level requirement** is statement about how a product must support stakeholders in achieving their goals or tasks.
- An **operational-level requirement** is a statement about inputs, outputs, operations, characteristics, etc. that a product must provide, without reference to physical realization.
- A **physical-level requirement** is a statement about the physical form of a product, its physical interfaces, or its data formats.

SRS Template

- There is no universal SRS template. There are many templates available in books and on the internet. Most employ requirements types and levels of abstraction to help organize the

material.

- Templates must be adapted for the product at hand.

IEEE Template

1. Product Description

1.1 Product Vision

1.2 Business Requirements

1.3 Users and Other Stakeholders

1.4 Project Scope

1.5 Assumptions

1.6 Constraints

2. Functional Requirements

3. Data Requirements

4. Non-Functional Requirements

5. Interface Requirements

5.1 User Interfaces

5.2 Hardware Interfaces

5.3 Software Interfaces

IEEE SRS Template

1. Product Description

1.1 Product Vision

- A **product vision statement** is a general description of a product's.

purpose and form.

1.2 Business Requirements

- **Business requirements** are statements of client or development organization goals that a product must meet

1.3 Users and Other Stakeholders

- **A stakeholder** is anyone affected by a product or involved in or

influencing its development. Developers must know who the stakeholders are so that they are all consulted (or at least considered) in designing, building, deploying, and supporting the product.

1.4 Project Scope

The **project scope** is the work to be done in the project.

IEEE Template

1.5 Assumptions

- An **assumption** is something that the developers may take for granted. It is important to make assumptions explicit so that all stakeholders are aware of them and can call them into question.

1.6 Constraints

- A **constraint** is any factor that limits developers. Of course developers must be aware of all constraints

2. Functional Requirements

- **Functional requirements** are specifications of the product's externally

observable behavior.

IEEE Format

3. Data Requirements

- **Data requirements** describe the format, structure, type, and allowable values of data entering, leaving, or stored by the product.

4. Non-Functional Requirements

- A non-functional requirement is a statement that a software product must have certain properties.

5. Interface Requirements

5.1 User Interfaces

5.2 Hardware Interfaces

5.3 Software Interfaces

SRS Description

- In this template, the design problem is documented in the “Product Description” section, which contains most of the information from the project mission statement. If a project mission statement exists, it should be referenced rather than reproduced...
- The sections named “Functional Requirements,” “Data Requirements,” and “Non-Functional Requirements” contain specifications mainly at the user and operational levels of abstraction.

SRS

- The SRS is often referred to as the “parent” document because all subsequent project management documents, such as design specifications, statements of work, software architecture specifications, testing and validation plans, and documentation plans, are related to it.
- It’s important to note that an SRS contains functional and nonfunctional requirements only; it doesn’t offer design suggestions, possible solutions to technology or business issues, or any other information other than what the development team understands the customer’s system requirements to be.
- **Example:**

https://www.scribd.com/doc/11934168/SRS-of-ATM

Week 9 Summery

Software Design and Architecture (Prototyping)

Prototypes

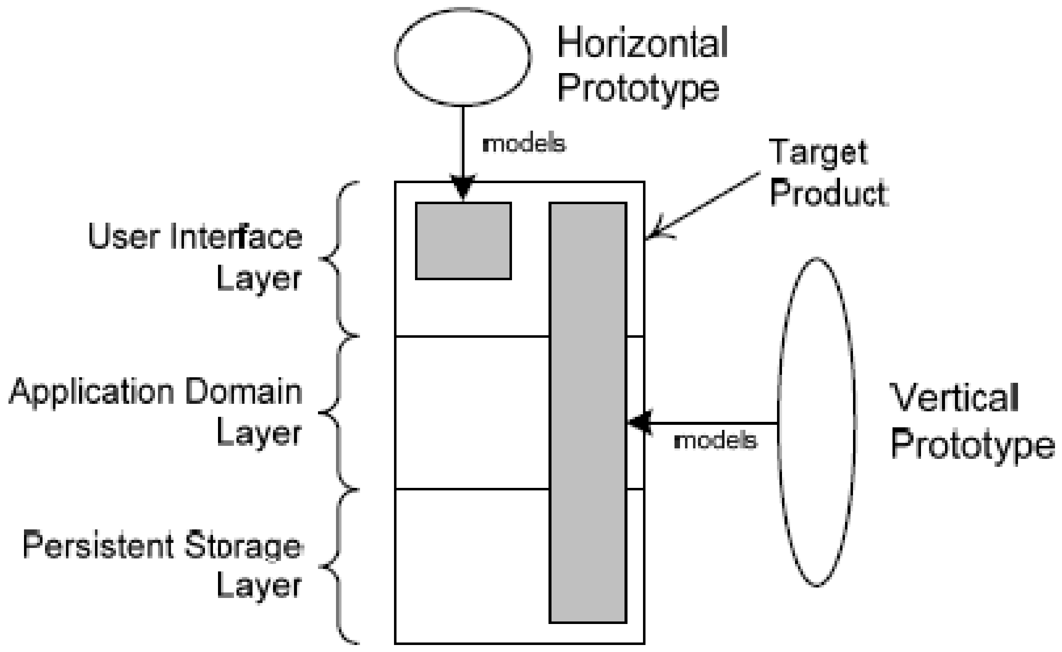
A prototype is a special kind of model.

- Represent a target (the product)
- Must work in some way

A **prototype** is a working model of part or all of a final product.

Horizontal & Vertical Prototypes

- A **horizontal prototype** realizes part or all of a product's user interface.
  - One program layer
  - Mock-ups
- A **vertical prototype** does processing apart from that required to present a user interface.
  - Cuts across program layers
  - Proof of concept prototype



Throwaway and Evolutionary Prototypes

- A **throwaway prototype** is developed as a design aid and then discarded.
    - Exploratory prototype
    - Quick to build
    - Risky to use in the final product
- An **evolutionary prototype** is a prototype that becomes (part of) the final product.
    - Iterative development
    - More expensive to build

Low- and High-Fidelity Prototypes

- **Fidelity** is how closely a prototype represents the final product it models.
- Low-fidelity prototypes
    - Paper
    - “Executed” by walking through interactions
    - Very quick and easy
- High-fidelity prototypes
    - Usually electronic

    - Take longer to build (good tools help)

Prototype Uses

- User involvement
- Enhanced communication
- Eliminates ambiguities
- Improves accuracy
- Early identification of problem
- Developers understanding of problem

Prototyping Risks

- Using a throwaway prototype as the basis for development
    - Avoid making high-fidelity throwaway prototypes
    - Make it very clear to stakeholders that the prototype only *appears* to work
- Fixation on appearance rather than function
    - Use low-fidelity prototypes for needs elicitation
- Final product does not look like prototype
    - Ensure that high-fidelity prototypes are accurate representations

Summary

- A variety of models are used for several tasks in product design.
- A **prototype** is a working model of (part of) a final product.
- Prototypes can be throwaway or evolutionary, horizontal or vertical, and have varying degrees of fidelity.
- Prototypes are useful for needs elicitation, for alternative generation, evaluation, and selection, and for design finalization.
- Risks attendant on the use of prototypes can usually be mitigated.

Software Architecture and design
(Introduction to UML)

Modeling

- A picture is worth 1000 words.
- A model is a representation of reality, like a model car, airplane.
- Most models have both diagrams and textual components.

Why Modeling?

- Visualization.
- Communicate with customer.
- Reduction of complexity.

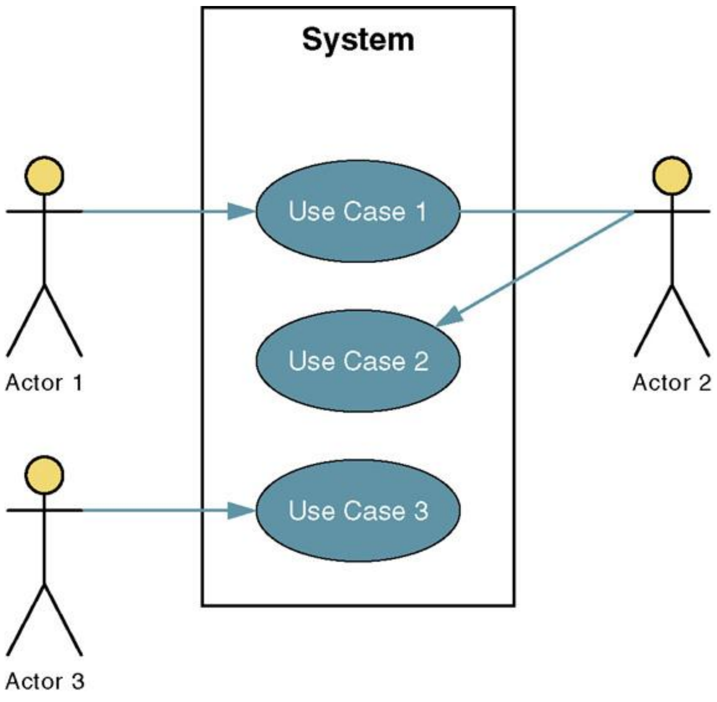
UML

- Unified Modeling Language
- Has become Industry standard
- Uses graphical notations to express OO analysis and design of software
- Simplifies the complex process of software design
- Using graphical notations is better than natural language
- Helps acquire the overall view of system
- Not dependent on any language

Use case modeling (Part 1)

- Use case diagrams describe what tasks the system performs.
  - E.g. Order placement, a ticket reservation, assignment submission etc.
- Who uses the system
  - A customer, a librarian, a student etc.
- Which user interacts with which usecase.

Sample Use case model

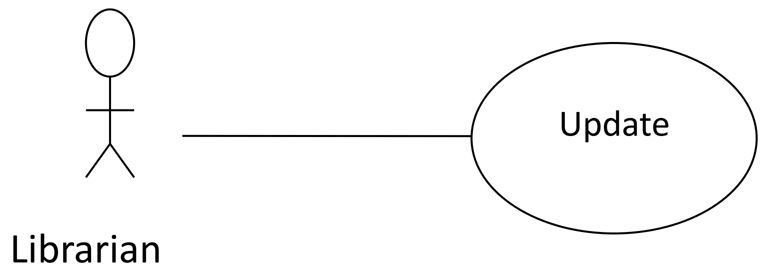


Components of use case

- Use case: subset of the overall system functionality.
- Actor: Anyone or anything that needs to interact with the system to exchange information.
- Association: which actor interacts with which use case.

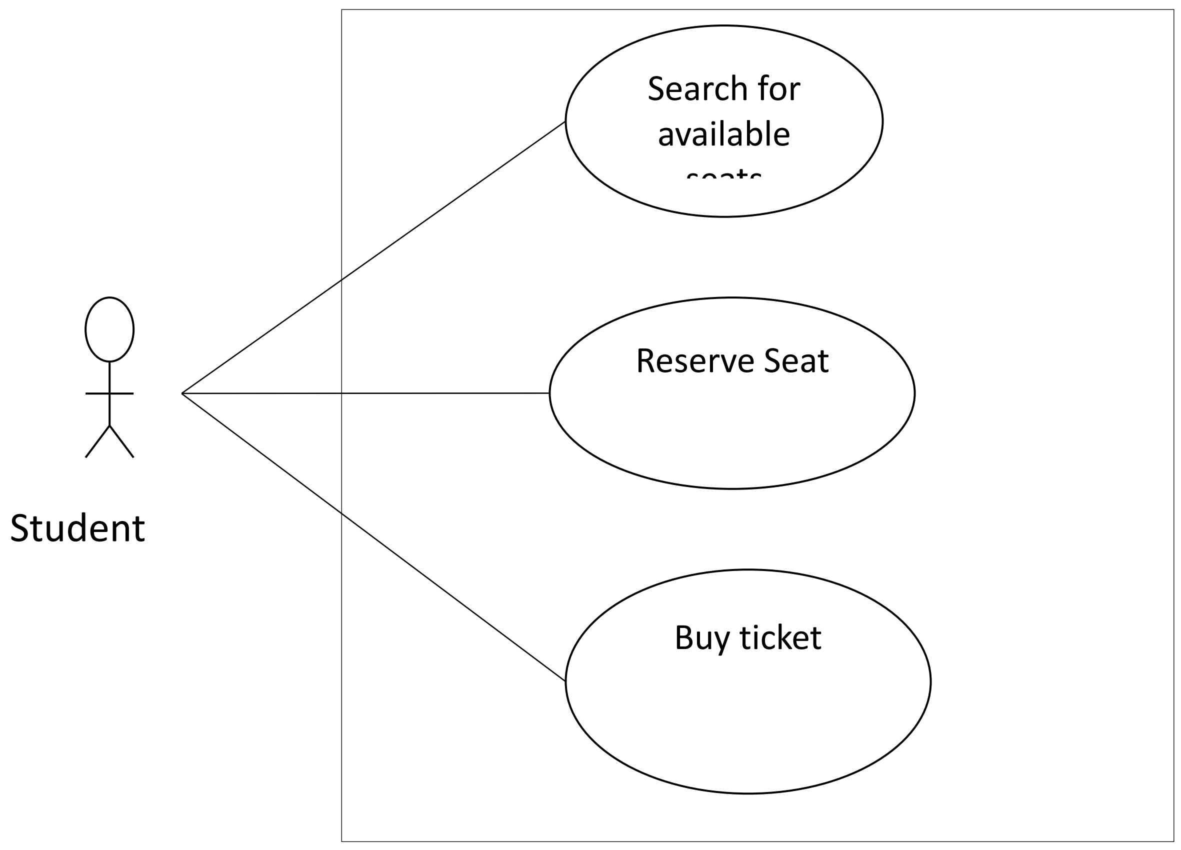
Sample use case diagram

- A Librarian updates a book catalogue



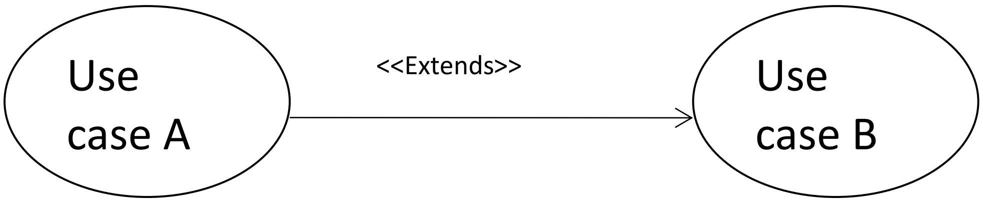
Sample use case diagram

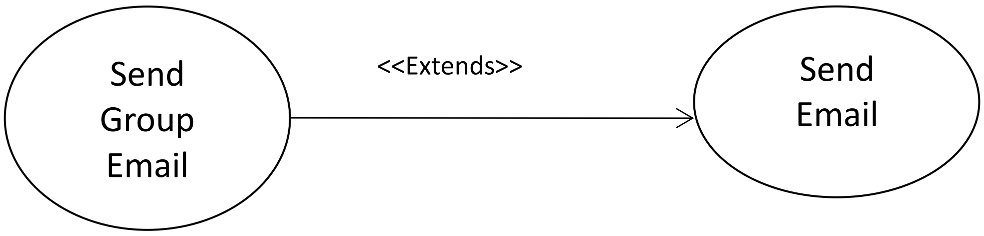
- A passenger bus ticket



Reuse (dependency) in use case

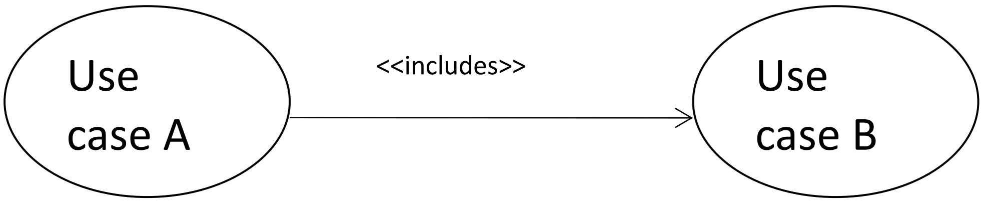
- Extends: An extend dependency, formerly called an Extends relationship is a generalization relationship where an extending use case continues the behavior of a base use case

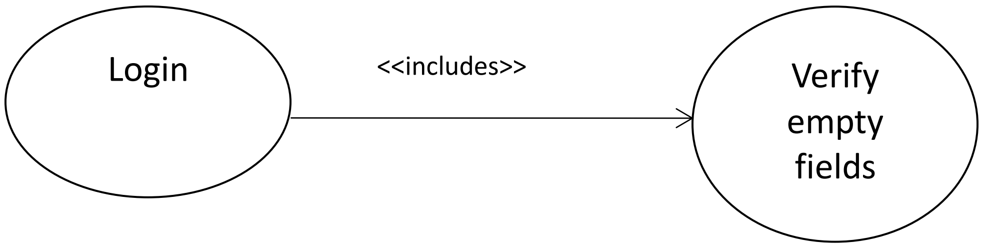




Reuse (dependency) in use case

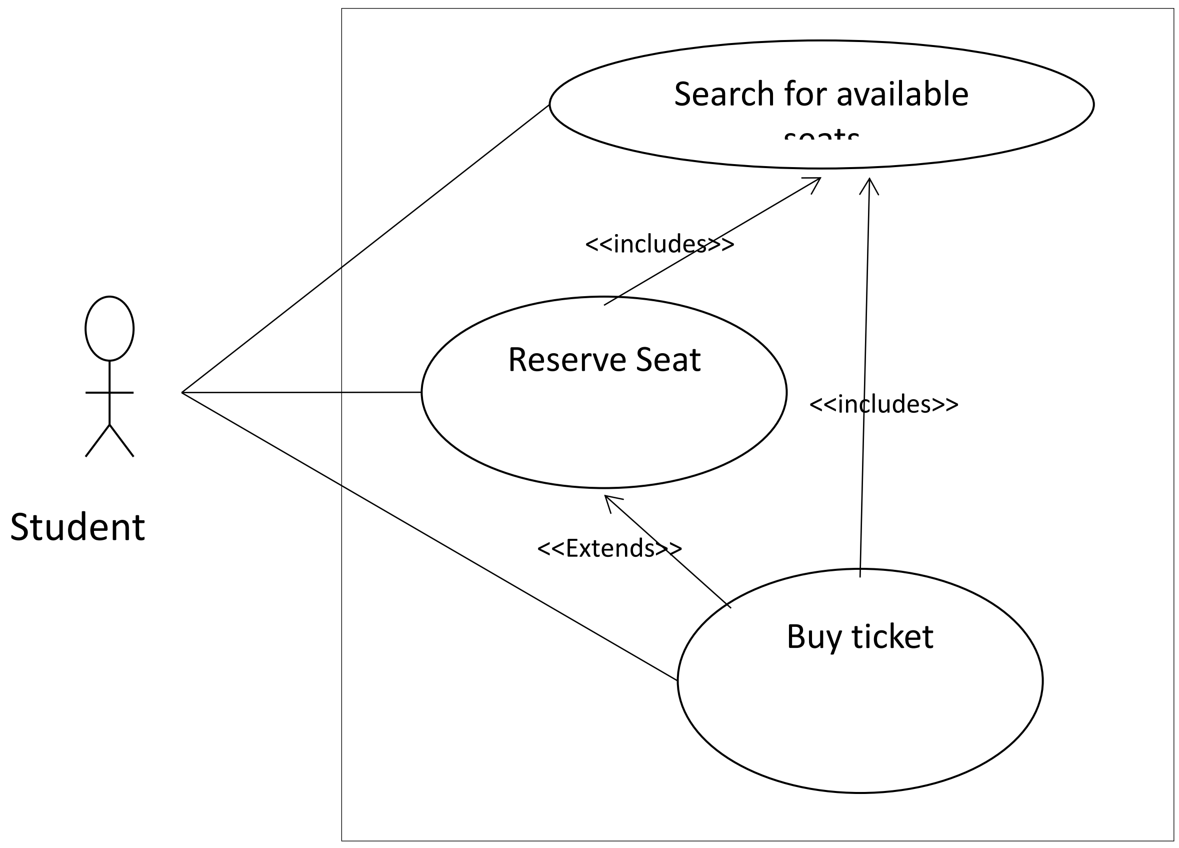
- Includes: An include dependency, is a generalization relationship denoting the inclusion of the behavior described by another use case.





Sample use case diagram

- A Librarian updates a book catalogue



Includes vs Extends

- Includes
  - You have a piece of behavior that is similar across many use cases
  - Break this out as a separate use-case and let the other ones "include" it.
  - <<Includes>> keyword is used.
- Extends
  - A use-case is similar to another one but does a little bit more
  - Put the normal behavior in one use-case and the exceptional behavior somewhere else.
  - <<Extends>> keyword is used

Week 10 Summery

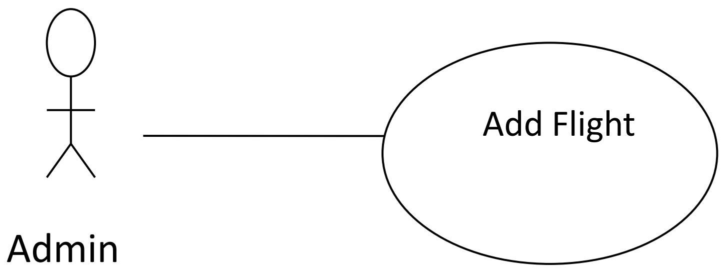
Use case modeling
(Part 2)

- Step-1: Identify business actors.
- Step-2: Identify business use cases.
- Step-3: Construct use-case model diagram.
- Step-4: Documents business requirements use-case narratives.

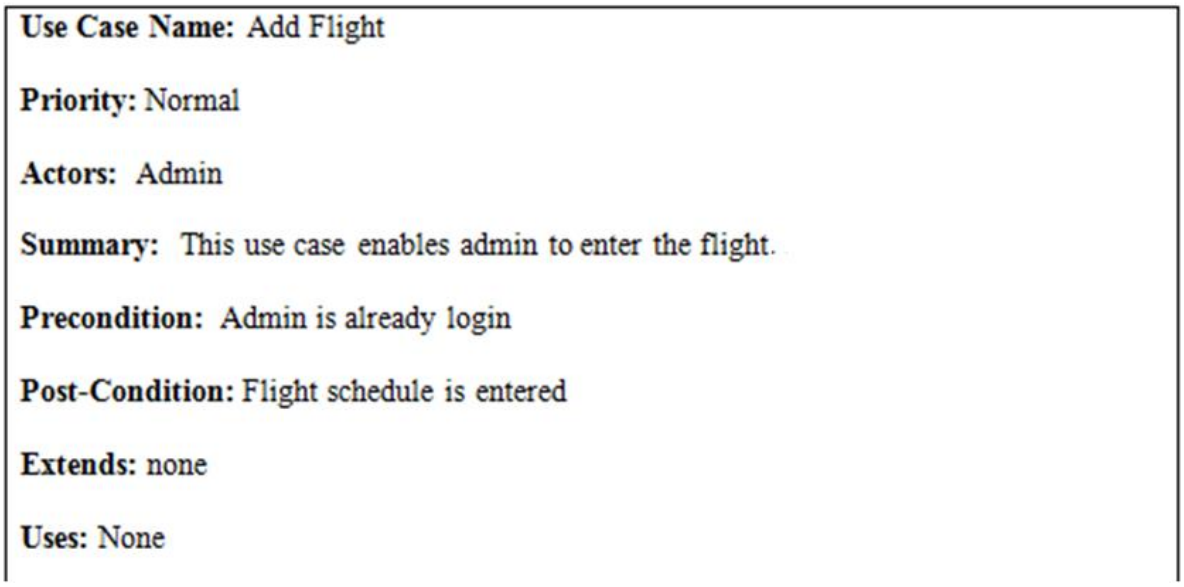
- **Step-1**: Identify business actors.
  - Who or what provides inputs to the system?
  - Who or what receives outputs from the system?
  - Are interfaces required to other systems?
  - Who will maintain information in the system?
- Actors should be named with a noun or noun phrase

- **Step-2**: Identify business use cases.
  - What are the main tasks of the actor?
  - What information does the actor need from the system?
  - What information does the actor provide to the system?
- Use cases should be named with a verb phrase specifying the goal of the actor (e.g. PlaceOrder)

- **Step-3:** Construct use-case model diagram.



- **Step-4:** Documents business requirements use-case narratives.



Use Case Name: Add Flight

Priority: Normal

Actors: Admin

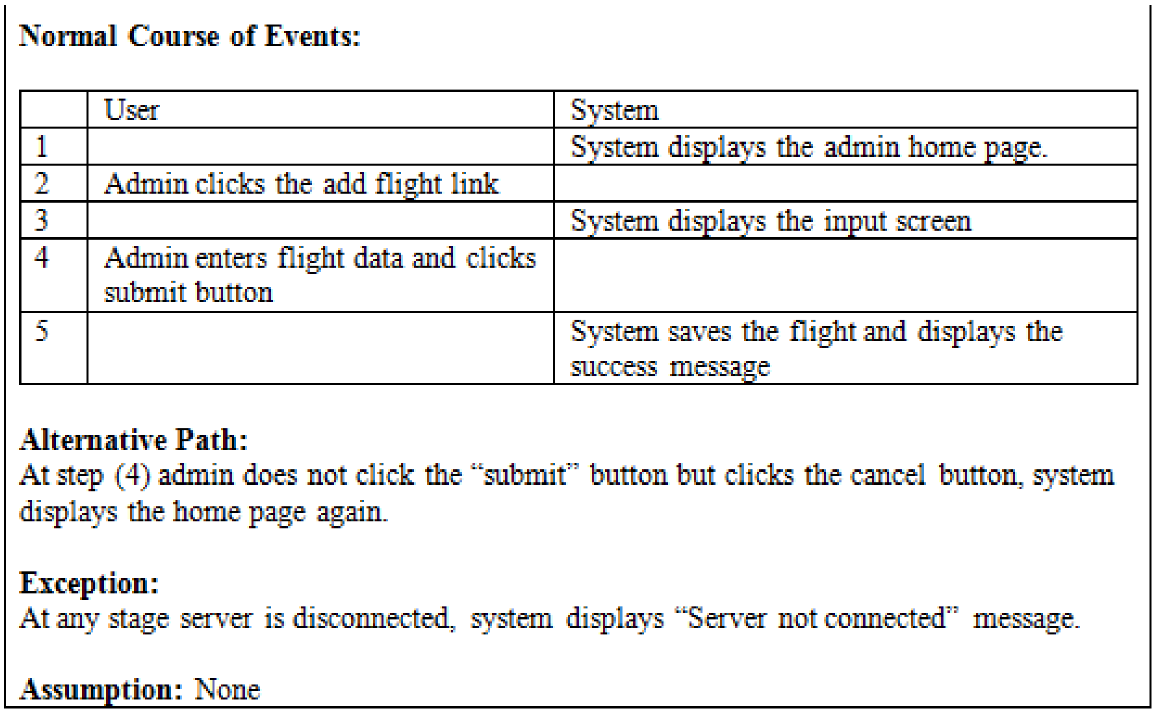
Summary: This use case enables admin to enter the flight.

Precondition: Admin is already login

Post-Condition: Flight schedule is entered

Extends: none

Uses: None



Normal Course of Events:

	User	System
1		System displays the admin home page.
2	Admin clicks the add flight link	
3		System displays the input screen
4	Admin enters flight data and clicks submit button	
5		System saves the flight and displays the success message

Alternative Path:
At step (4) admin does not click the "submit" button but clicks the cancel button, system displays the home page again.

Exception:
At any stage server is disconnected, system displays "Server not connected" message.

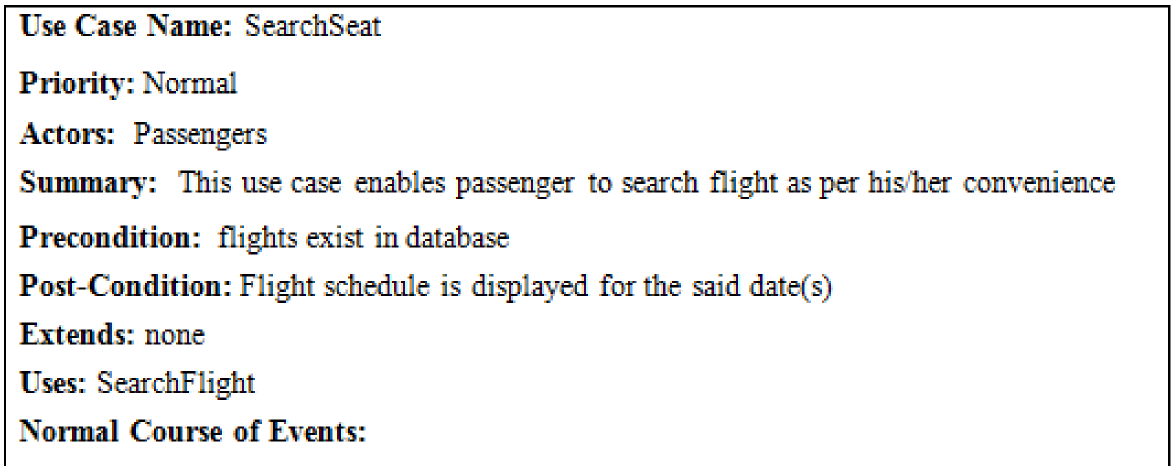
Assumption: None

Air Ticket Reservation System

- Reservations on local system
- Passenger goes to client terminal in local office
- Searches flights/seats.
- Takes print of available seats.
- Booking staff confirms seat.
- Client terminal also displays flash news/updates.
- Admin can Add/Edit/Cancel flight schedule (Email is sent to passengers)
- Admin can cancel ticket.
- Admin can Add/Edit/Cancel Reservation

Air Ticket Reservation System

- **Actors**
    - Passenger
    - Admin
    - ?
- **Use cases**
    - ViewNewsFlash
    - PrintSchedule
    - SearchSeat
    - AddFlight
    - ReserveSeat
    - EditReservation
    - CancelReservation
- **Use cases**
    - SendEmail
    - AddFlight
    - EditFlight
    - CancelFlight
    - AddUser
    - EditUser
    - DeleteUser



Use Case Name: SearchSeat

Priority: Normal

Actors: Passengers

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

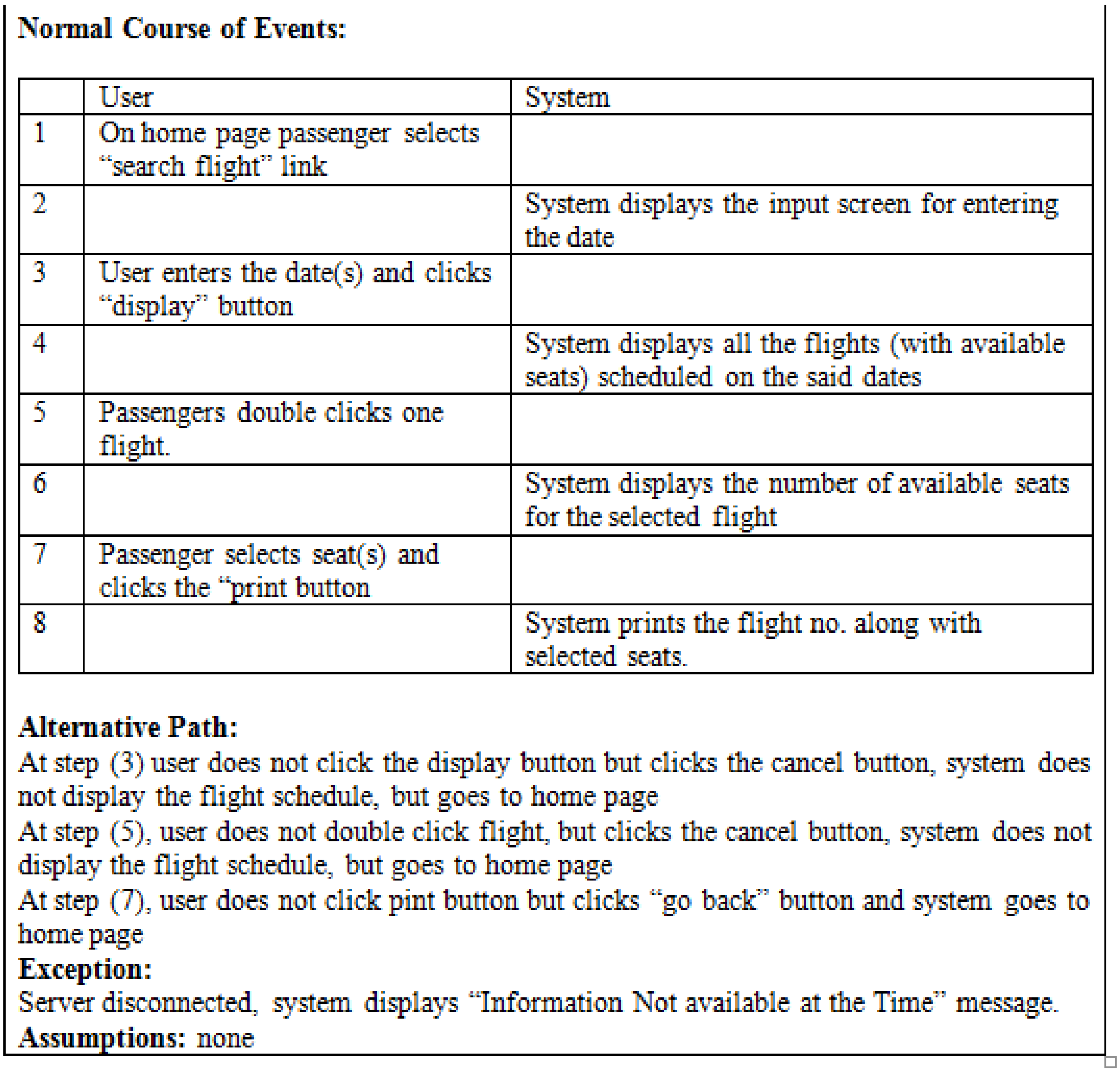
Precondition: flights exist in database

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

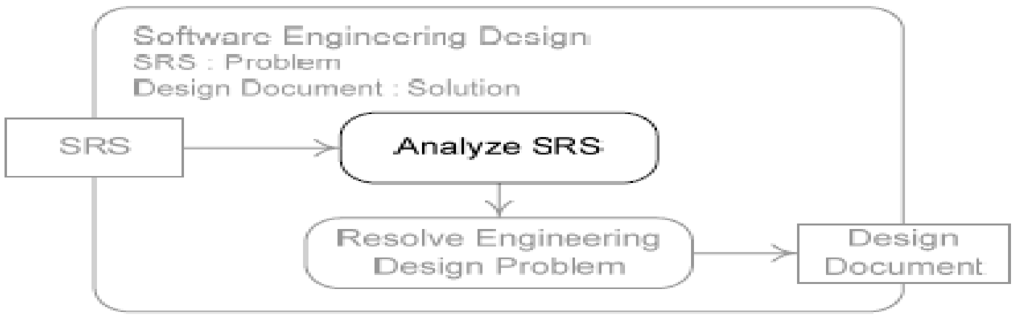
Week 11 Summery

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


Software Engineering Design
SRS : Problem
Design Document : Solution
SRS
Analyze SRS
Resolve Engineering Design Problem
Design Document

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

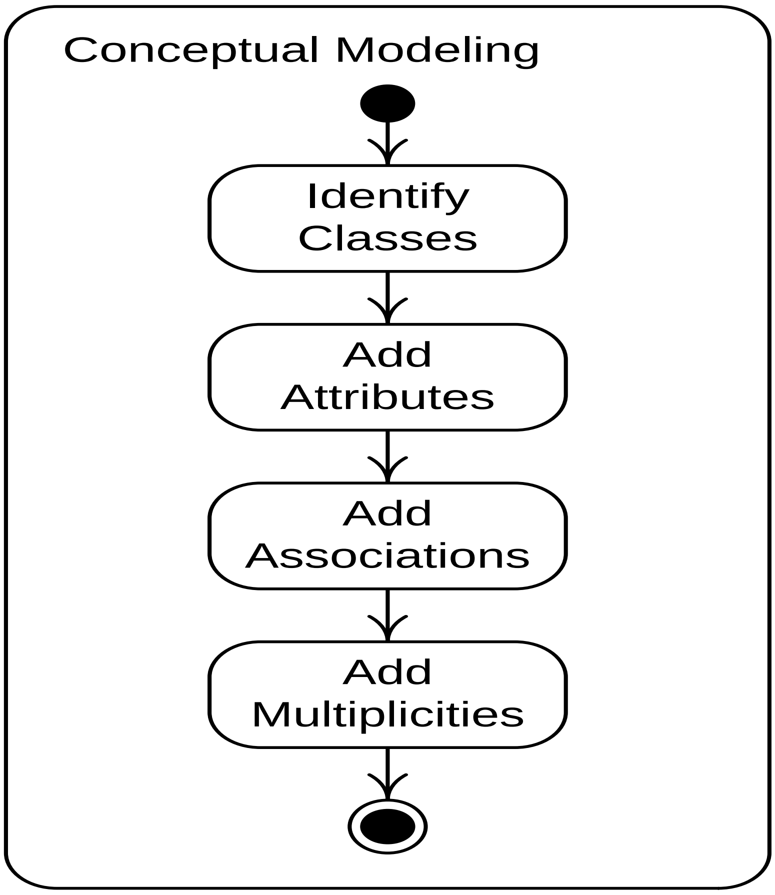
Conceptual models

- 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

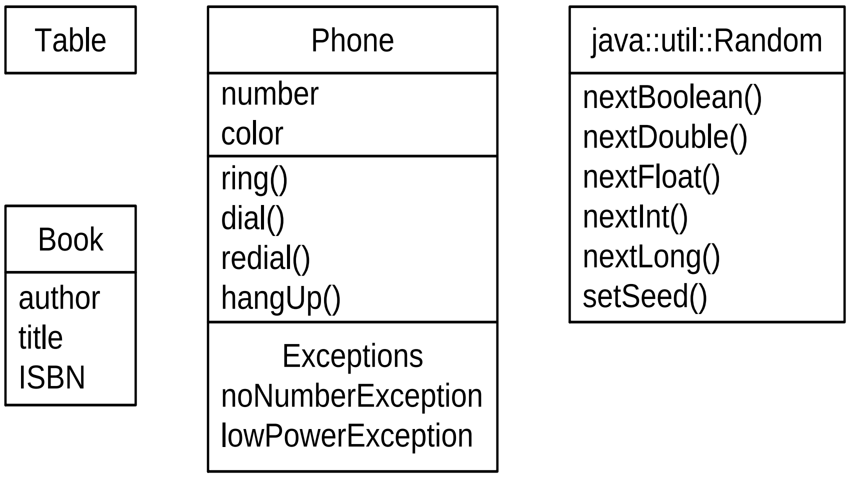
Classes and Objects

- 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*

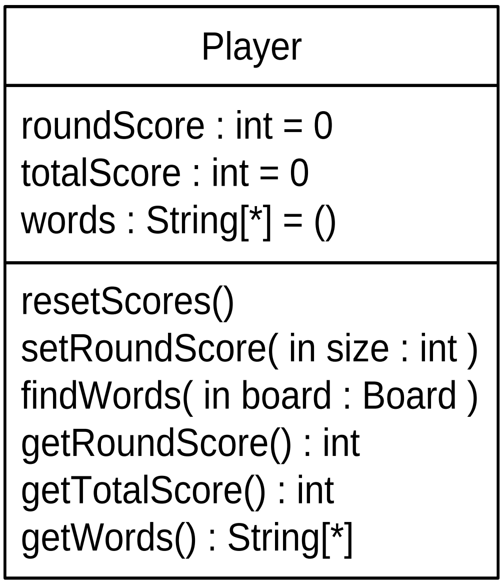
- *name*—simple name, cannot be suppressed
- *type*—any string, may be suppressed with the :
- *multiplicity*—number of values stored in attribute
    - list of ranges of the form *n*..*k*, such that *n* <=*k*
    - *k* may be *
    - *n*..*n* is the same as *n*
    - 0..* is the same as *
    - 1 by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

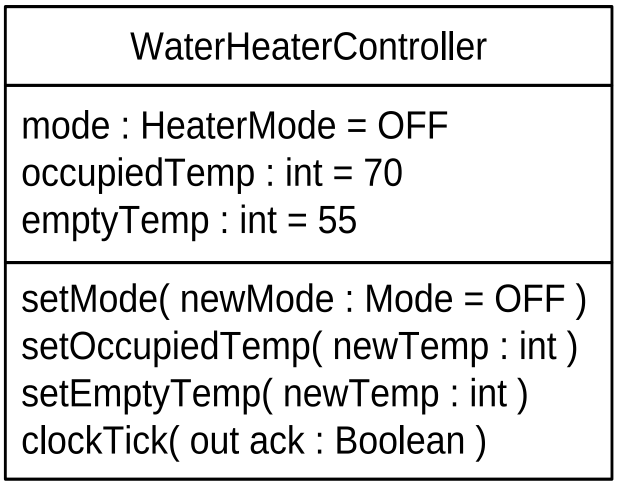
Operation Specification Format

- *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.

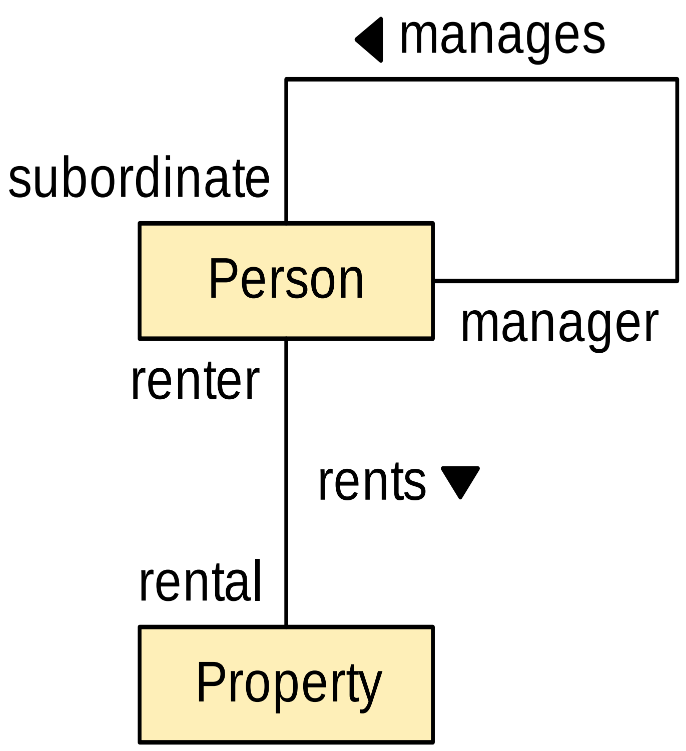
Attribute and Operation Examples




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

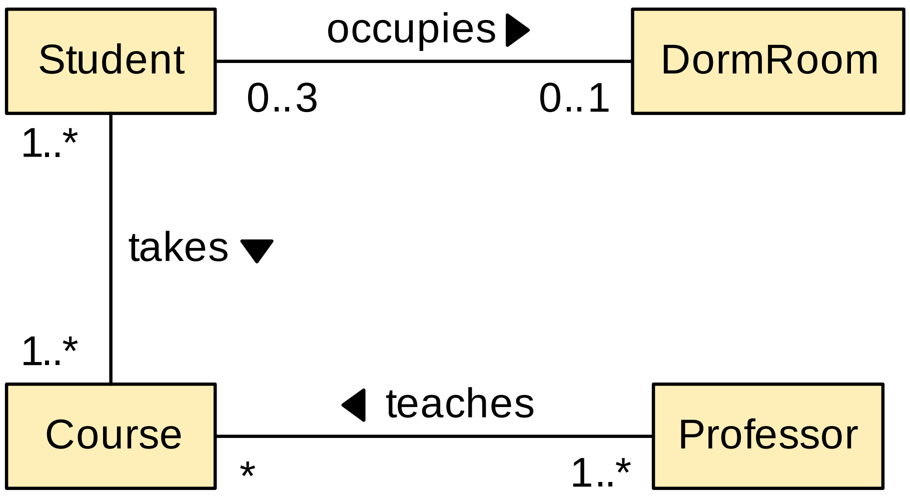
Association Lines

- 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)

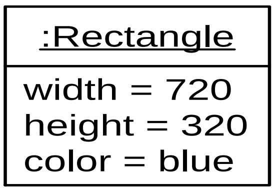
Object Name Format

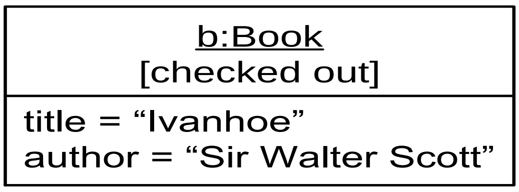
- *object-name : class-name*
  [*stateList*]
- *object-name*—simple name
- *class-name*—a name (simple or composite)
- *stateList*—list of strings; if suppressed, the square brackets are omitted
- The *object-name* and *class-name* may both be suppressed, but not simultaneously

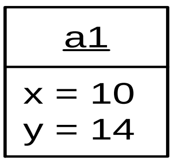
Object Attribute Format

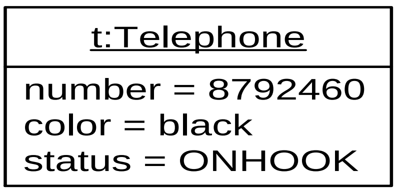
- *attribute-name = value*
- *attribute-name*—simple name
- *value*—any string
- Any attribute and its current value may be suppressed together

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

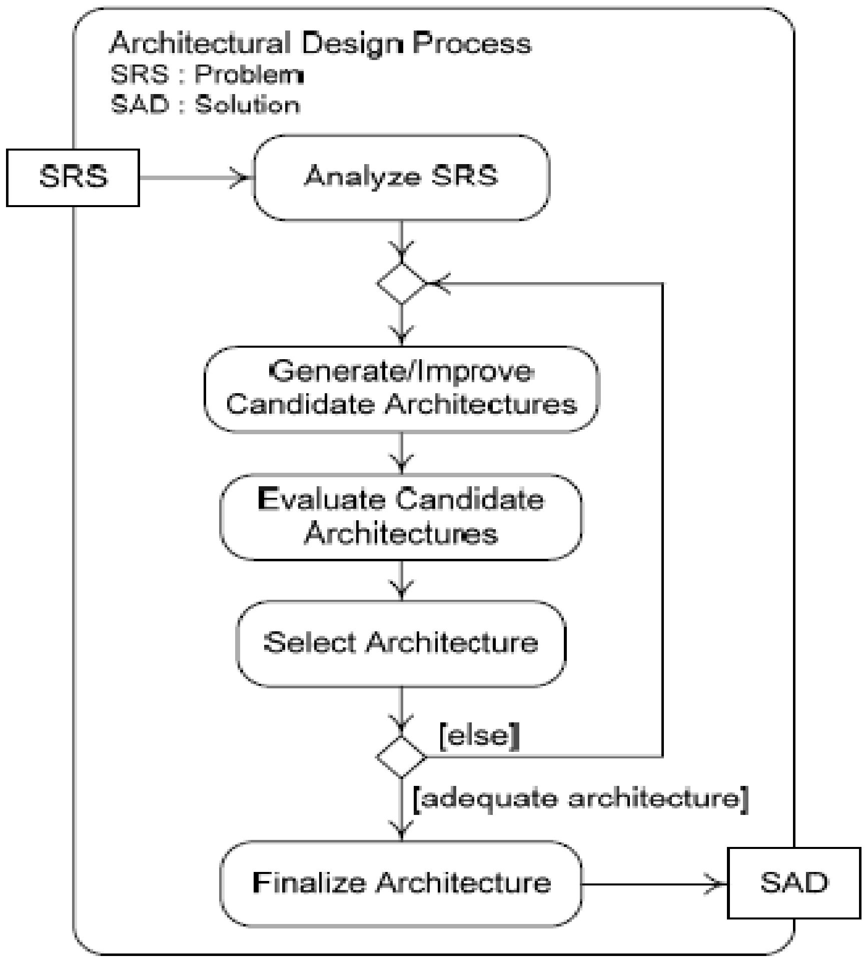
- Engineering design resolution has 2 phases:”architectural design” and “detailed design”.
- Architectural design is a problem-solving activity whose input is the product description in an SRS and whose output is the abstract specification of a program realizing the desired product. Architectural design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



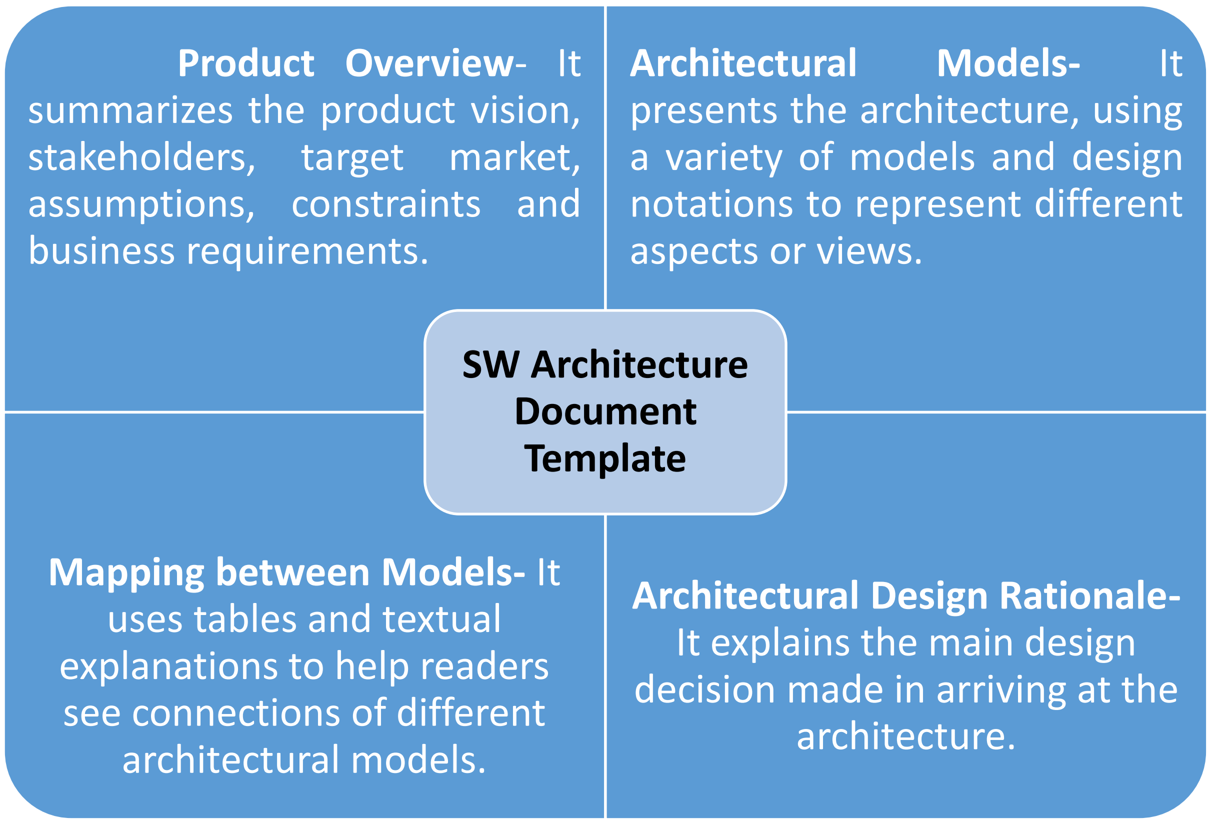
A **software architecture document(SAD)** is simply a document that specifies the architecture of a software system,

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.



SW Architecture Document Template

Product Overview- It summarizes the product vision, stakeholders, target market, assumptions, constraints and business requirements.

Architectural Models- It presents the architecture, using a variety of models and design notations to represent different aspects or views.

Mapping between Models- It uses tables and textual explanations to help readers see connections of different architectural models.

Architectural Design Rationale- It explains the main design decision made in arriving at the architecture.

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.

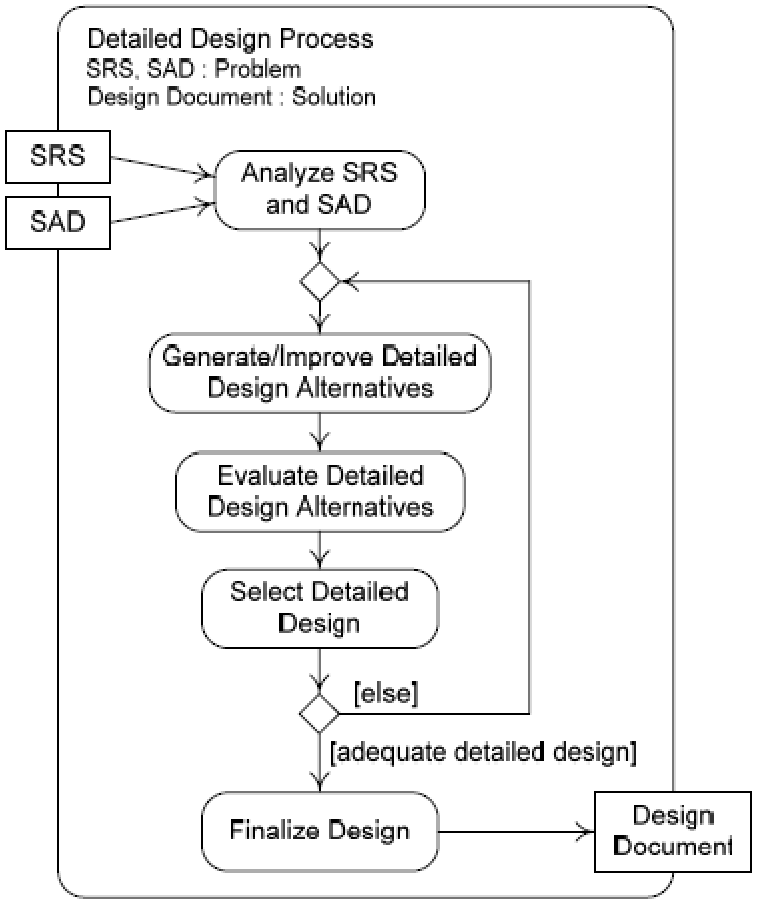

Detailed Design Process
SRS, SAD : Problem
Design Document : Solution
SRS
SAD
Analyze SRS and SAD
Generate/Improve Detailed Design Alternatives
Evaluate Detailed Design Alternatives
Select Detailed Design
[else]
[adequate detailed design]
Finalize Design
Design Document

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

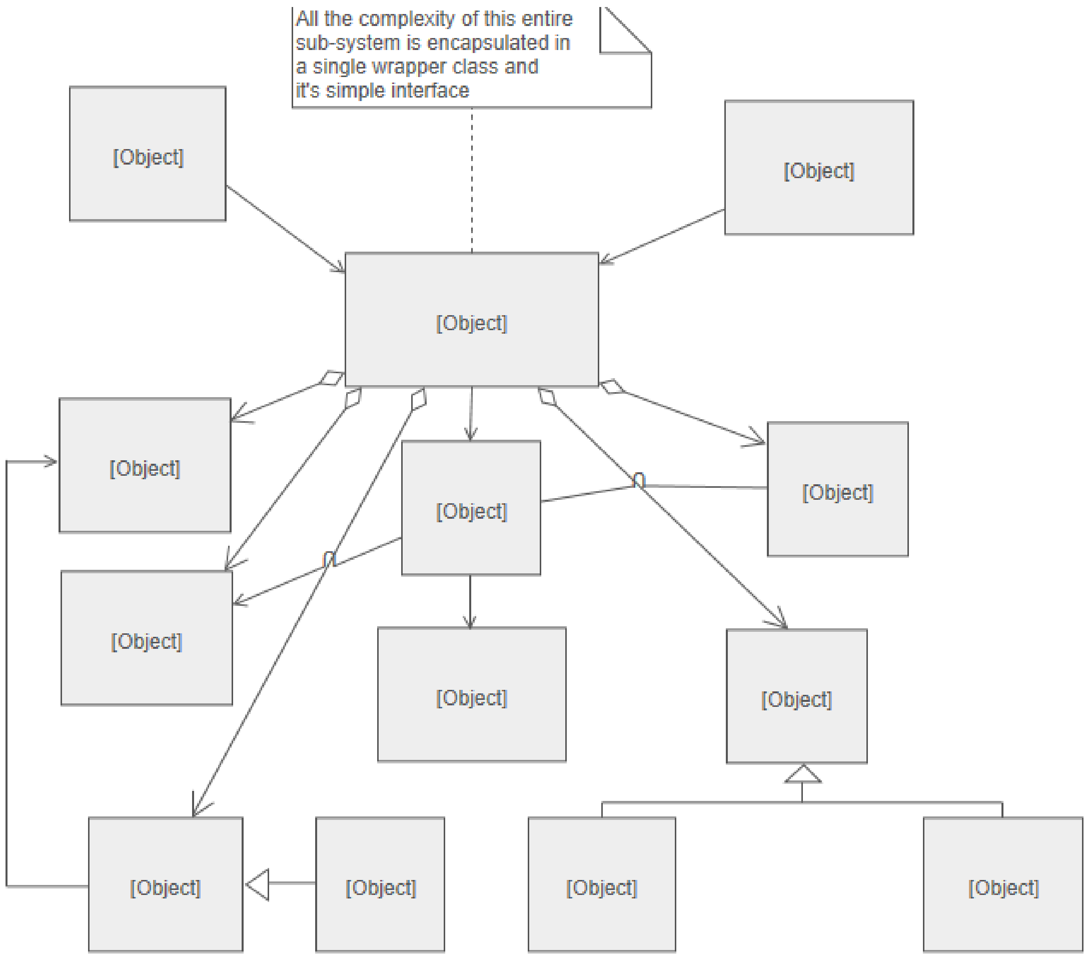
- Structural Design Patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.
- These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns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


All the complexity of this entire sub-system is encapsulated in a single wrapper class and it's simple interface
[Object]

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

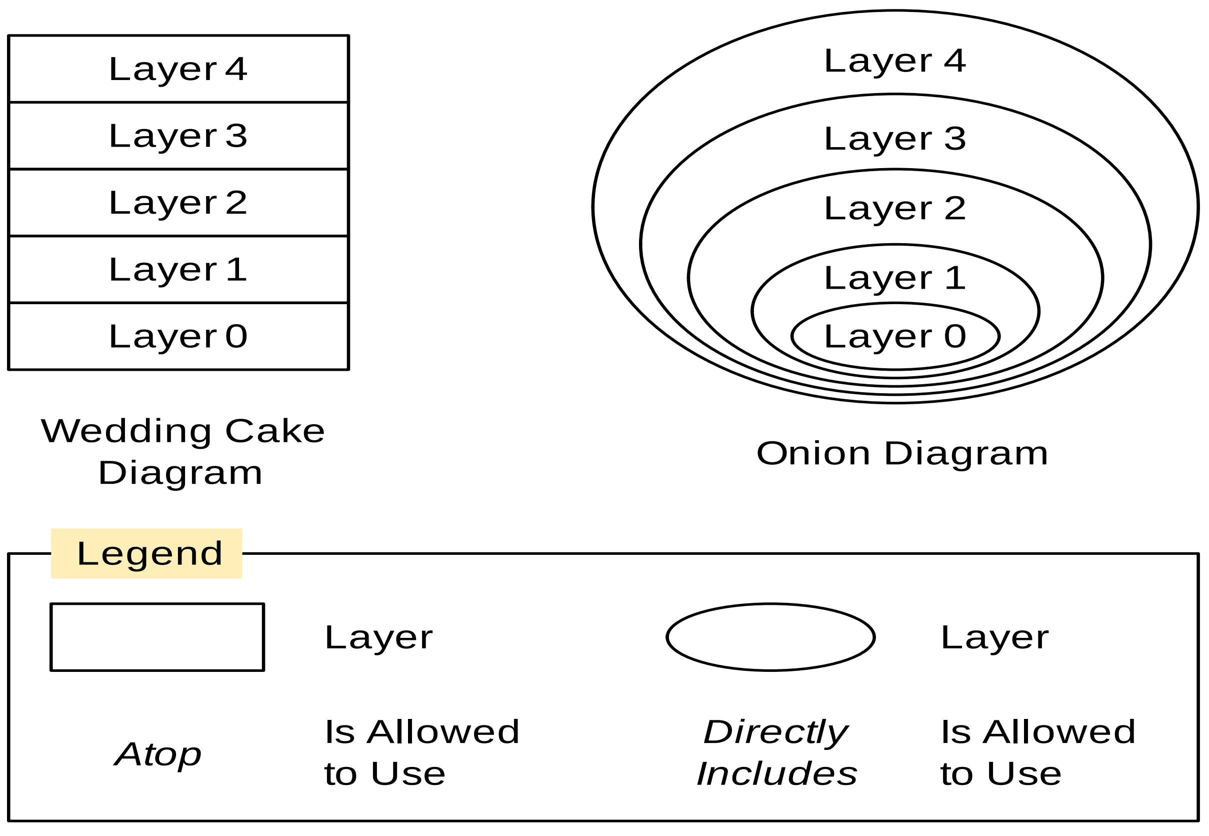
- Module A **uses** module B if a correct version of B must be present for A to execute correctly.
- Module A **calls** or **invokes** module B if A triggers execution of B.
- Note that
    - A module may use but not invoke another

- 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



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

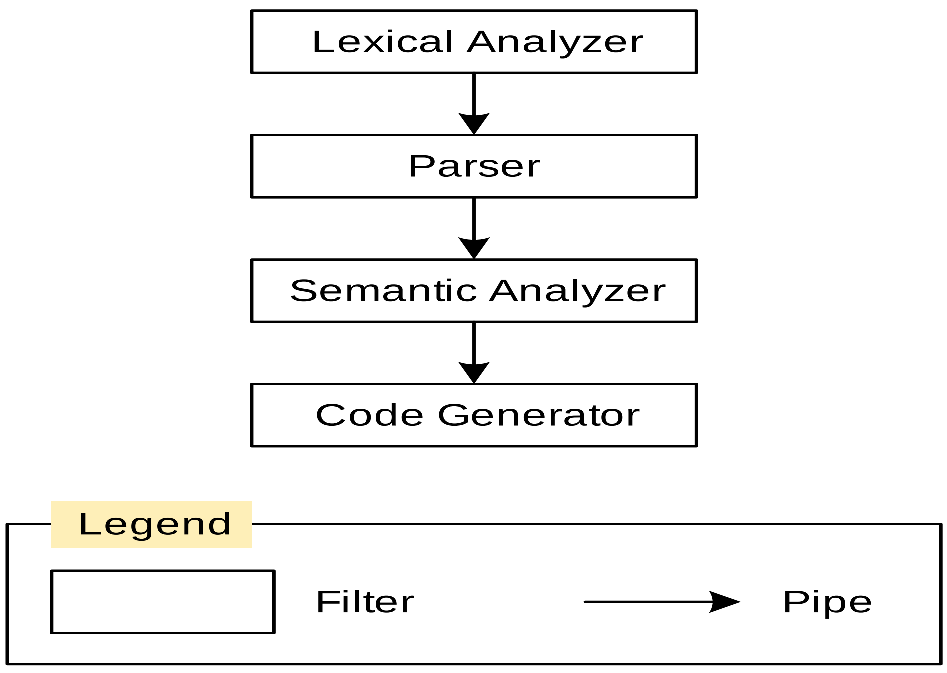
Architectural Styles

- 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.

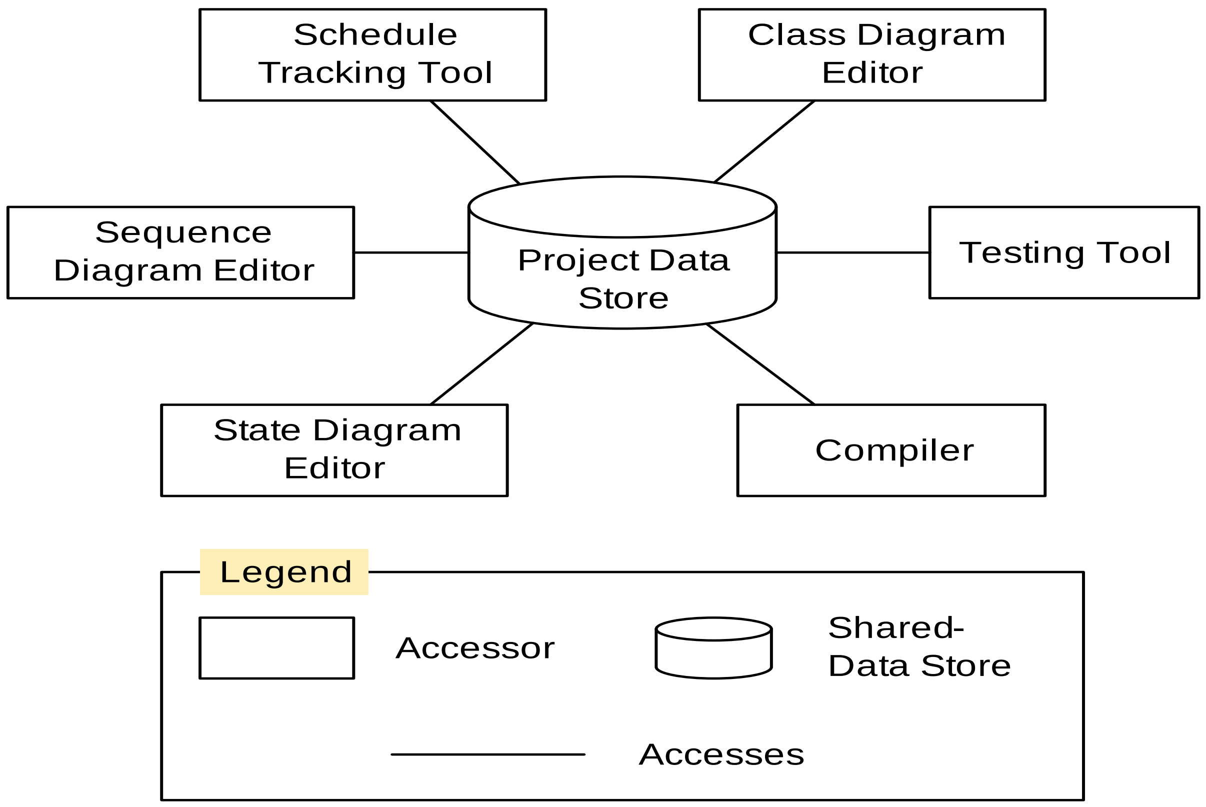
Pipe-and-Filter Disadvantages

- 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.

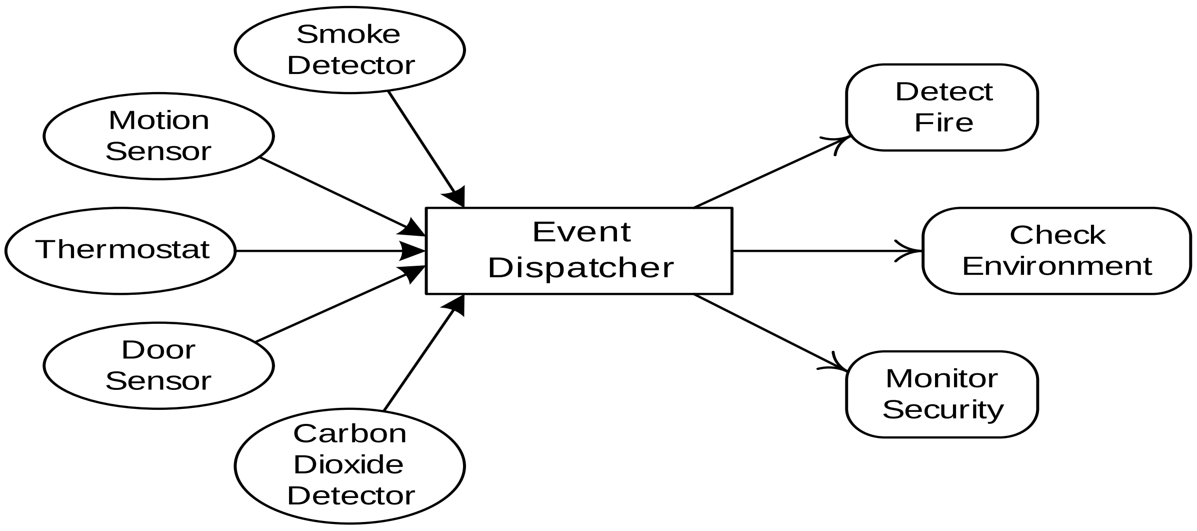
Shared-Data Style Disadvantages

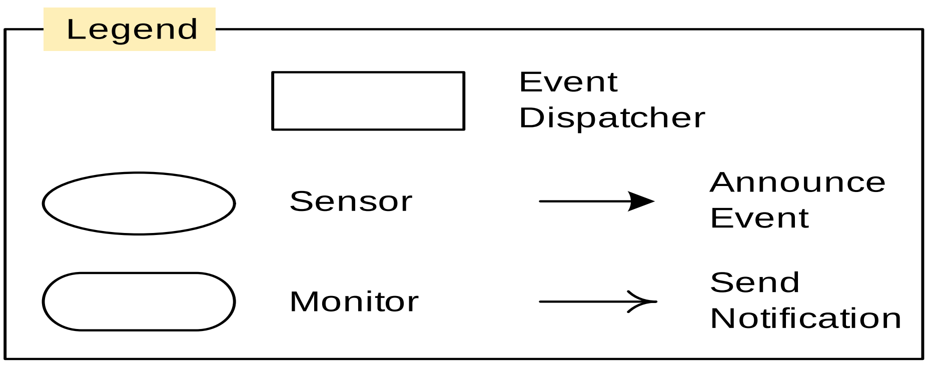
- Forcing all data through the shared-data store may degrade performance.
- If the shared-data store fails, the entire program is crippled.

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.

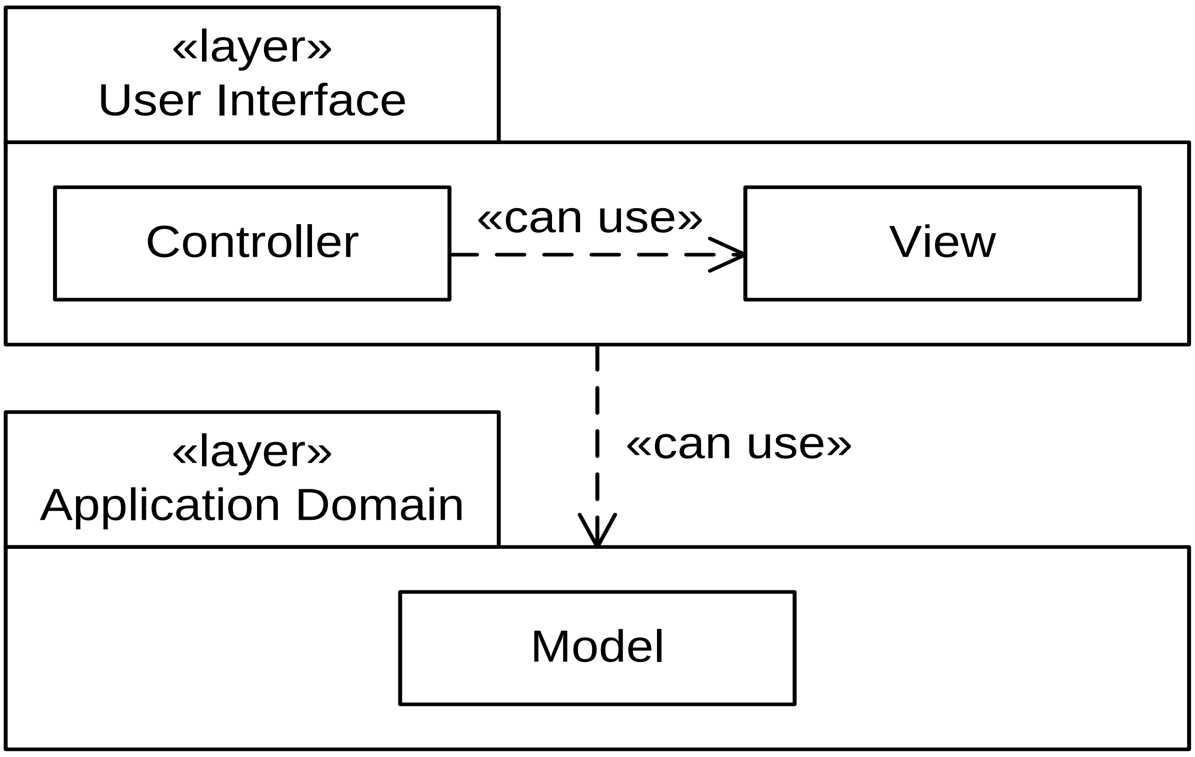
Event-Driven Style Disadvantages

- 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.

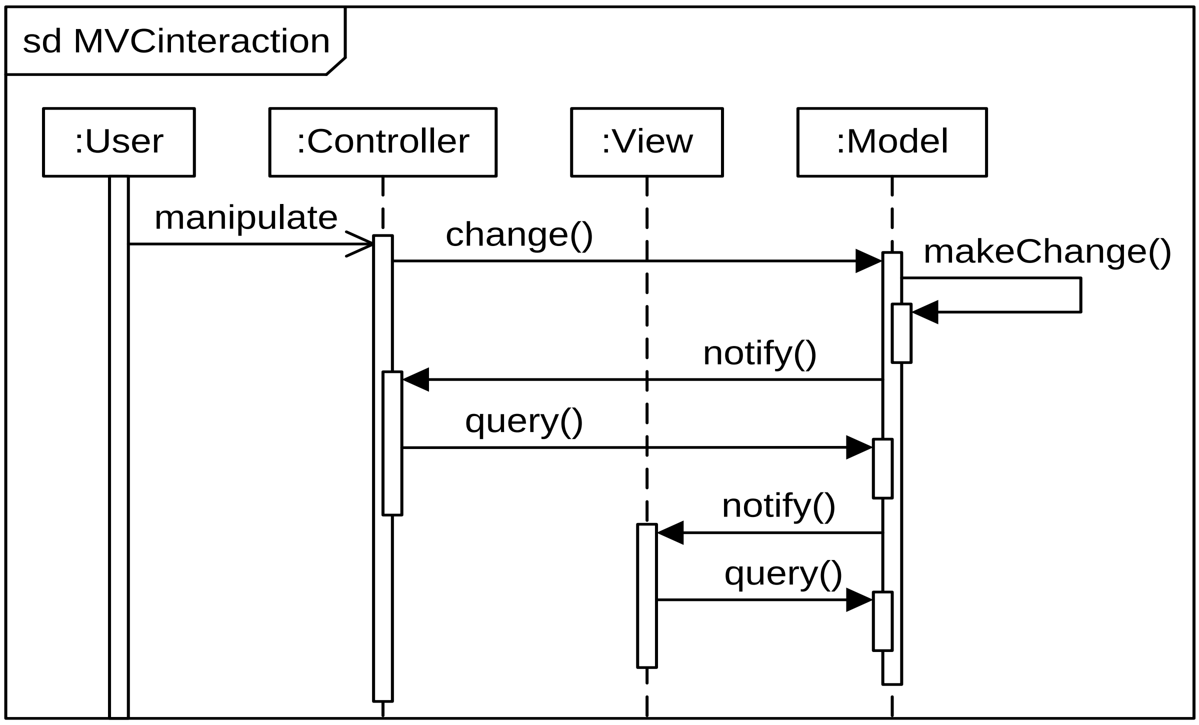
Model-View-Controller (MVC) Style

- 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

Virtual University

By Wahab Ahmad