Module 2: Software Requirements and Design (9 hrs.) – Comprehension level

Requirements Engineering: Eliciting requirements, Functional and non- Functional requirements, SRS, Requirements modelling: Developing Use Cases, Developing Activity diagram and Swim lane diagram, Design : Design concepts, Architectural design, Component based design, User interface design.

USER INTERFACE DESIGN & COMPONENT LEVEL DESIGN

Department of Computer Science and Engineering
School of Engineering, Presidency University

Contents – User Interface Design

Contents – Component Level Design

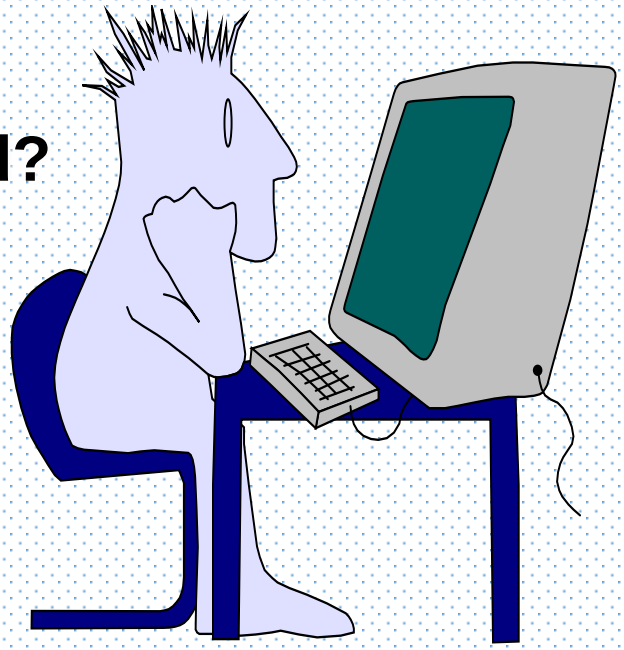
1. Component
   - Object-oriented view
   - Traditional view
   - Process related view
2. Designing class based components
   - Basic design Principles
   - Component level design guidelines
   - Cohesion
   - Coupling
3. Conducting Component level design
4. Component level design for web Apps
5. Designing Traditional components

1. Interface Design

Easy to learn?

Easy to use?

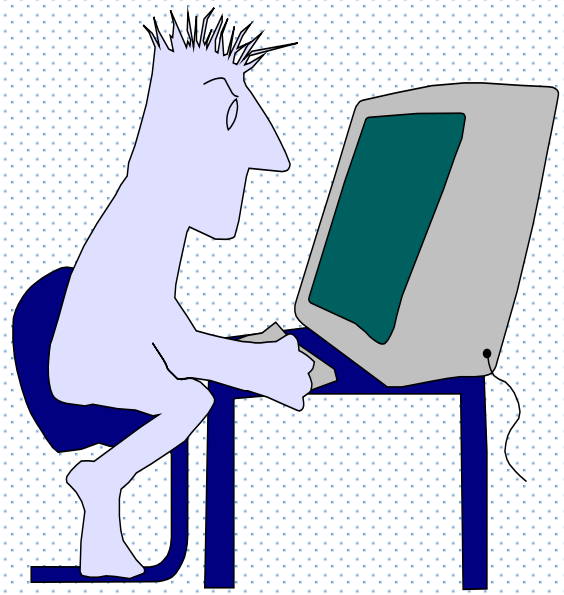
Easy to understand?



Interface Design

Typical Interface Design Errors

lack of consistency
too much memorization
no guidance / help
no context sensitivity
poor response
unfriendly



2. Golden Rules for Interface Design

2.1 Place the User in Control

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions. (Ex: Spell – Check)
- Provide for flexible interaction. (Ex: Keyboard, Mouse, Voice)
- Allow user interaction to be interruptible and undoable.
- Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user.
- Design for direct interaction with objects that appear on the screen. (Ex: Stretch the screen)

2.2 Reduce the User's Memory Load

- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- The visual layout of the interface should be based on a real world metaphor.
- Disclose information in a progressive fashion.

- Ex: Swiggy Application.

2.3 Make the Interface Consistent

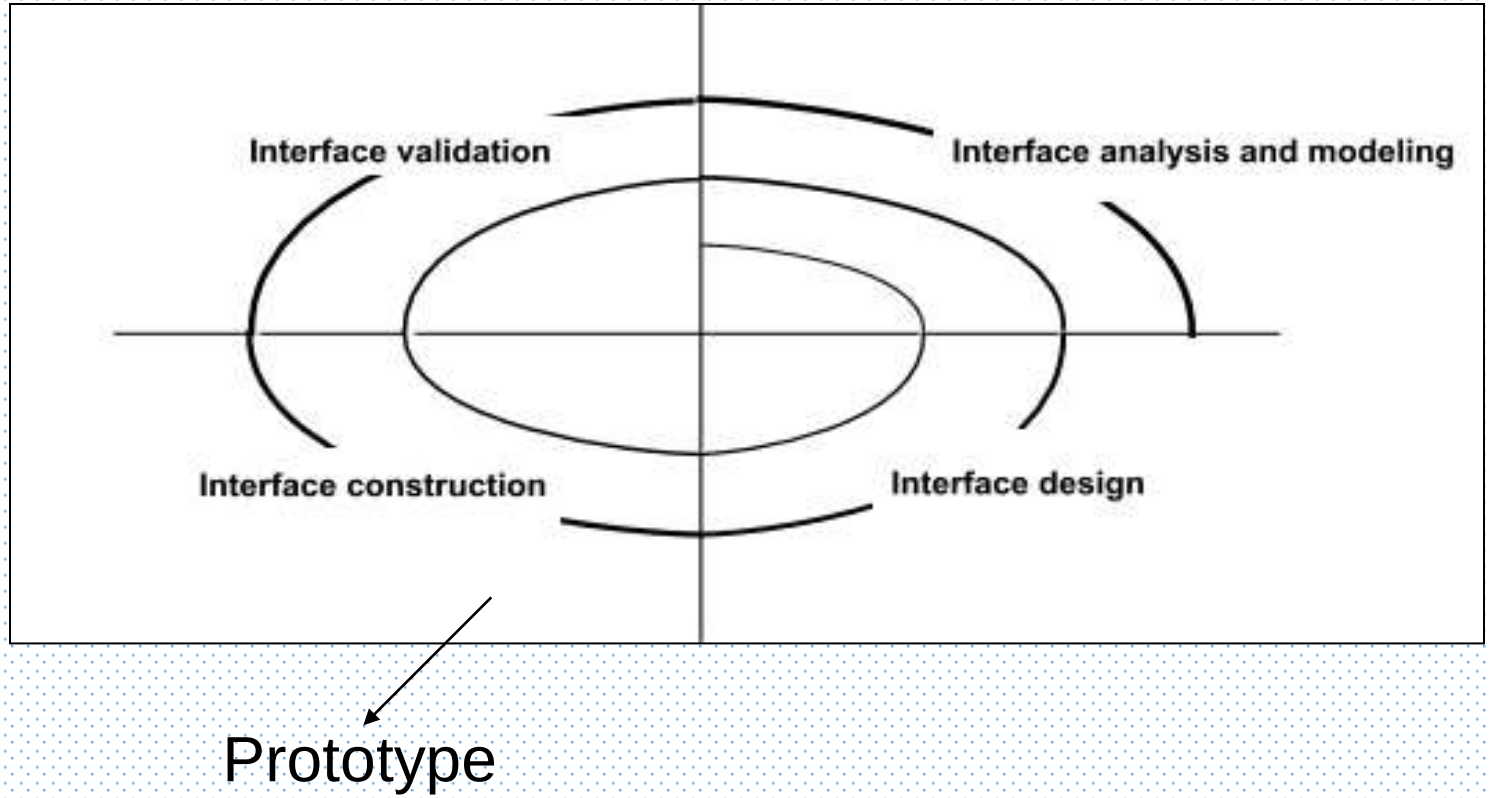
- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

3. User Interface Design Models

- ***User model*** — a profile of all end users of the system created by Software Designer
- ***Design model*** — a design realization of the user model created by Software Designer
- ***Mental model (system perception)*** — the end user's mental image of what the interface is

  - To cater to Novice, Intermediate and high-knowledge users
- ***Implementation model*** — the interface "look and feel" coupled with supporting information that describe interface syntax and semantics

User Interface Design Process



4. WebApp Interface Design

- ***Where am I?*** The interface should:
  - provide an indication of the WebApp that has been accessed
  - inform the user of her location in the content hierarchy.
- ***What can I do now?*** The interface should always help the user understand his current options
  - what functions are available?
  - what links are live?
  - what content is relevant?
- ***Where have I been, where am I going?*** The interface must facilitate navigation.
  - Provide a "map" (implemented in a way that is easy to understand) of where the user has been and what paths may be taken to move elsewhere within the WebApp.

Effective WebApp Interfaces

- Bruce Tognozzi [TOG01] suggests…
  - *Effective interfaces are visually apparent and forgiving,* instilling in their users a sense of control. Users quickly see the breadth of their options, grasp how to achieve their goals, and do their work.
  - *Effective interfaces do not concern the user with the inner workings of the system*. Work is carefully and continuously saved, with full option for the user to undo any activity at any time.
  - *Effective applications and services perform a maximum of work,* while requiring a minimum of information from users.

5. WebApp Interface Design Principles I

- *Anticipation* - A WebApp should be designed so that it anticipates the use's next move.
- *Communication* - The interface should communicate the status of any activity initiated by the user
- *Consistency* - The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout)
- *Controlled autonomy* - The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- *Efficiency* - The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the Web engineer who designs and builds it or the client-server environment that executes it.

WebApp Interface Design Principles II

- *Focus* - The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.
- *Fitt's Law* - "The time to acquire a target is a function of the distance to and size of the target."
- *Human interface objects* - A vast library of reusable human interface objects has been developed for WebApps.
- *Latency reduction* - The WebApp should use multi-tasking in a way that lets the user proceed with work as if the operation has been completed.
- *Learnability* - A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.

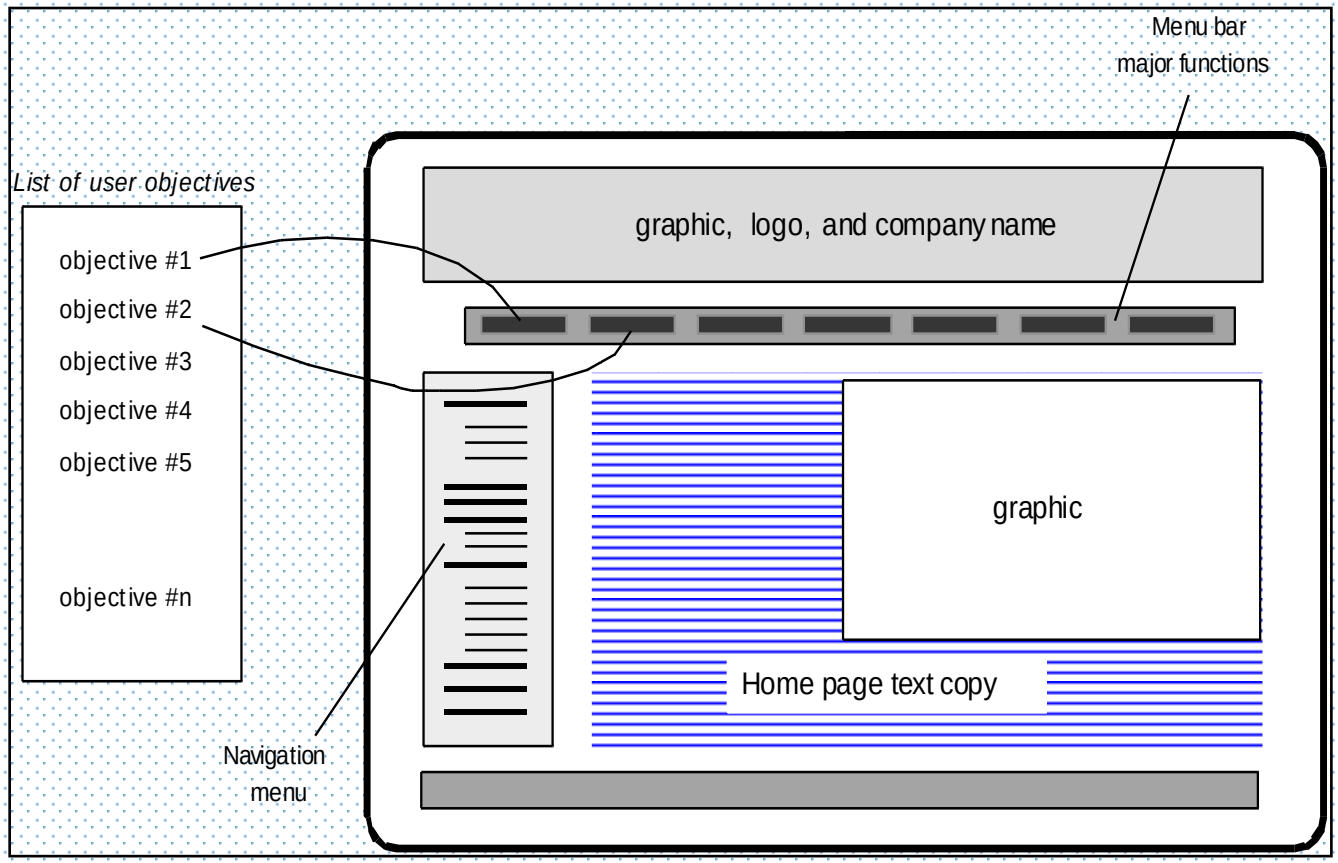
WebApp Interface Design Principles III

- *Maintain work product integrity* - A work product (e.g., a form completed by the user, a user specified list) must be automatically saved so that it will not be lost if an error occurs.
- *Readability* - All information presented through the interface should be readable by young and old.
- *Track state* - When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.
- *Visible navigation* - A well-designed WebApp interface provides "the illusion that users are in the same place, with the work brought to them."

6. WebApp Interface Design Workflow - I

- Review information contained in the analysis model and refine as required.
- Develop a rough sketch of the WebApp interface layout.
- Map user objectives into specific interface actions.
- Define a set of user tasks that are associated with each action.
- Storyboard screen images for each interface action.
- Refine interface layout and storyboards using input from aesthetic design.

Mapping User Objectives



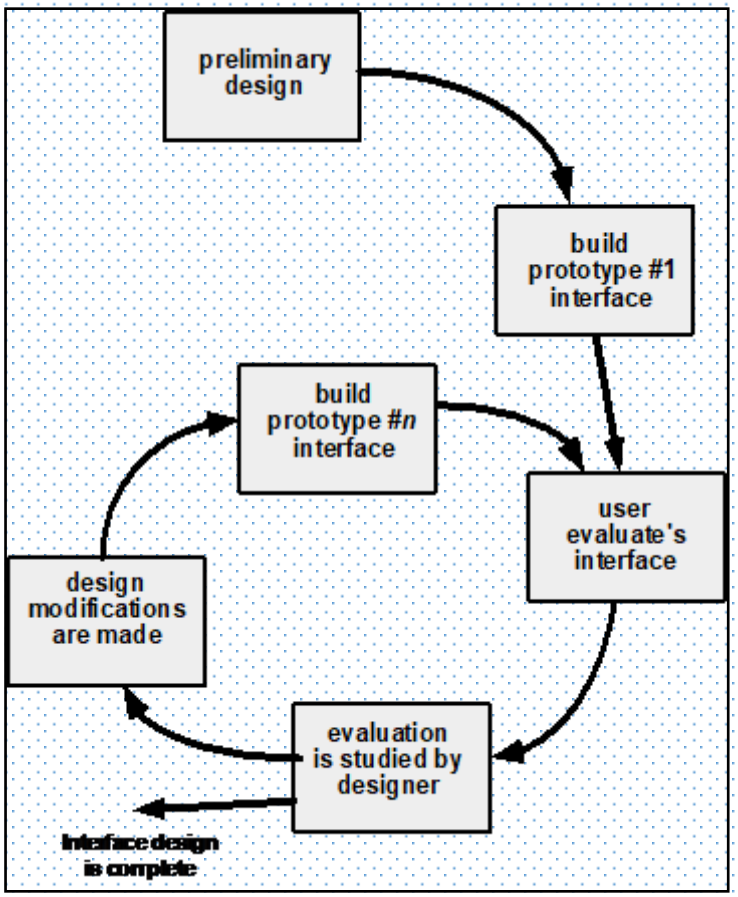
WebApp Interface Design Workflow - II

- Identify user interface objects that are required to implement the interface.
- Develop a procedural representation of the user's interaction with the interface. (Activity diagram depicting the flow of activities that occur when as the user interacts with the WebApp)
- Develop a behavioral representation of the interface. (UML State Diagrams)
- Describe the interface layout for each state.
- Refine and review the interface design model.

7. Aesthetic Design

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

8. Design Evaluation Cycle



Component Level Design

1. Component

- A *component* is a modular building block for computer software. More formally, the *OMG Unified Modeling Language Specification* [OMG03a] defines a component as ". . . a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."

1.1 Object - Oriented view

- Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation.
- As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined.

1.2 Traditional view

Traditional component, also called a module, resides within the software architecture and serves one of three important roles:

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

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

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

1.3 Traditional view

- As the software architecture is developed, you choose components or design patterns from the catalog and use them to populate the architecture.
- Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration.

2. Designing class based components

2.1 Basic Design Principles

- The open – Closed Principle(OCP)
- The Liskov substitution Principle(LSP)
- Dependency Inversion Principle(DIP)
- The Interface Segregation Principle(ISP)
- The Release Reuse Equivalency Principle(CCP)
- The Common Closure Principle (CCP)
- The Common Reuse Principle(CRP)

2.2 Component Level Design guidelines

- *Components*
- *Interfaces*
- *Dependencies and Inheritance*

2.3 Cohesion

- Cohesion implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.

2.4 Coupling

- Coupling is a qualitative measure of the degree to which classes are connected to one another.
- As classes (and components) become more interdependent, coupling increases.
- An important objective in component-level design is to keep coupling as low as is possible.

Some Coupling categories

- ***Content Coupling***
- ***Common Coupling***
- ***Control Coupling***

3. Conducting Component level Design

Steps for designing

1. Identify all design classes that correspond to the problem domain.
2. Identify all design classes that correspond to the infrastructure domain.
3. Elaborate all design classes that are not acquired as reusable components.
4. Describe persistent data sources (databases and files) and identify the classes required to manage them.
5. Develop and elaborate behavioral representations for a class or component.
6. Elaborate deployment diagrams to provide additional implementation detail.
7. Refactor every component-level design representation and always consider alternatives.

4. Component-level design for web Apps

4.1 Content Design at the Component Level

- Content design at the component level focuses on content objects and the manner in which they may be packaged for presentation to a Web App end user.
- A number of potential content components can be defined for the video surveillance capability:

(1) the content objects that represent the space layout (the floor plan) with

additional icons representing the location of sensors and video cameras,

(2) the collection of thumbnail video captures (each a separate data object), and

(3) the streaming video window for a specific camera.

4.2 Functional Design at the Component Level

Modern Web applications deliver increasingly sophisticated processing functions that

(1) perform localized processing to generate content and navigation capability in a

dynamic fashion,

(2) provide computation or data processing capability that is appropriate for the Web

App's business domain,

(3) provide sophisticated database query and access, or

(4) establish data interfaces with external corporate systems

5. Design Traditional Components

- The constructs are sequence, condition, and repetition. Sequence implements processing steps that are essential in the specification of any algorithm.
- ***Condition*** provides the facility for selected processing based on some ***logical*** occurrence, and ***repetition*** allows for looping.
- A limited number of logical constructs also contributes to a human understanding process that psychologists call ***chunking.***
- ***Various Notation***
  - Graphical design notation
  - Tabular design notation
  - Program design Language

MODULE II

UNDERSTANDING REQUIREMENTS (CHAPTER 4)

Department of Computer Science and Engineering
School of Engineering, Presidency University

Module 2: Software Requirements and Design (9 hrs.) – Comprehension level

Requirements Engineering: Eliciting requirements, Functional and non- Functional requirements, SRS, **Requirements modelling**: Developing Use Cases, Developing Activity diagram and Swim lane diagram, **Design** : Design concepts, Architectural design, Component based design, User interface design.

Contents

1. Requirements Engineering - I

1. Inception - ask a set of questions that establish …
   - basic understanding of the problem
   - the people who want a solution
   - the nature of the solution that is desired, and
   - the effectiveness of preliminary communication and collaboration between the customer and the developer
2. Elicitation - elicit requirements from all stakeholders
3. Elaboration - create an analysis model that identifies data, function and behavioral requirements
4. Negotiation - agree on a deliverable system that is realistic for developers and customers

1. Requirements Engineering - I

5. Specification - can be any one (or more) of the following
   - A written document
   - A set of models
   - A formal mathematical specification
   - A collection of user scenarios (use-cases)
   - A prototype

6. Validation - a review mechanism that looks for
   - errors in content or interpretation
   - areas where clarification may be required
   - missing information
   - inconsistencies (a major problem when large products or systems are engineered)
   - conflicting or unrealistic (unachievable) requirements.

7. Requirements management

Functional and Non functional Requirements

Functional Requirements

- These are the requirements that the end user specifically demands as basic facilities that the system should offer.
- All these functionalities need to be necessarily incorporated into the system as a part of the contract.

Non Functional Requirements

- These are basically the quality constraints that the system must satisfy according to the project contract.
- The priority or extent to which these factors are implemented varies from one project to other. They are also called non-behavioral requirements.

FUNCTIONAL REQUIREMENTS	NON FUNCTIONAL REQUIREMENTS
1. A functional requirement defines a system or its component.	A non-functional requirement defines the quality attribute of a software system.
2. It specifies “What should the software system do?”	It places constraints on “How should the software system fulfill the functional requirements?”
3. Functional requirement is specified by User.	Non-functional requirement is specified by technical peoples e.g. Architect, Technical leaders and software developers.
4. It is mandatory.	It is not mandatory.
5. It is captured in use case.	It is captured as a quality attribute.
6. Defined at a component level.	Applied to a system as a whole.
7. Helps you verify the functionality of the software.	Helps you to verify the performance of the software.
8. Functional Testing like System, Integration, End to End, API testing, etc are done.	Non-Functional Testing like Performance, Stress, Usability, Security testing, etc are done.
9. Usually easy to define.	Usually more difficult to define.

Software Requirements Specification

- A ***software requirements specification*** **(SRS)** is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence.
- It is important to note that a formal SRS is not always written

Introduction

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

2. Overall Description

- 2.1 Product Perspective
- 2.2 Product Features
- 2.3 User Classes and Characteristics
- 2.4 Operating Environment

- 2.5 Design and Implementation Constraints
- 2.6 User Documentation
- 2.7 Assumptions and Dependencies

3. System Features

- 3.1 System Feature 1
- 3.2 System Feature 2 (and so on)

4. External Interface Requirements

- 4.1 User Interfaces
- 4.2 Hardware Interfaces
- 4.3 Software Interfaces
- 4.4 Communications Interfaces

5. Other Nonfunctional Requirements

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

6. Other Requirements

2. Establishing the ground work

- Identify stakeholders
  - *"who else do you think I should talk to?"*
- Recognize multiple points of view
- Work toward collaboration
- The first questions
  - *Who is behind the request for this work?*
  - *Who will use the solution?*
  - *What will be the economic benefit of a successful solution*
  - *Is there another source for the solution that you need?*

3. Eliciting Requirements

3.1 Collaborative Requirements Gathering

- meetings are conducted and attended by both software engineers and customers
- rules for preparation and participation are established
- an agenda is suggested
- a "facilitator" (can be a customer, a developer, or an outsider) controls the meeting
- a "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room or virtual forum) is used
- the goal is
  - *to identify the problem*
  - *propose elements of the solution*
  - *negotiate different approaches, and*
  - *specify a preliminary set of solution requirements*

3.2 Quality Function Deployment

Normal requirements: *Requirements which are stated during the meeting with the customer*

Example:) normal requirements might be requested types of graphical displays, specific system functions

Expected requirements: *Requirements are implicit to the product or system that are not explicitly stated by the customer.*

Exciting requirements: *features go beyond the customer's expectations and prove to be very satisfying when present.*

Example:) *software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product.*

3.3 Usage Scenarios

3.4 Elicitation Work Products

4. Developing use cases

Use Case Methodology (UCM) (for Requirements Elicitation)

Key Terms in UCM

- **SUD** – System under Discussion

- **Actor**

  Actors are basically users of the SUD who are **external entities** (people or other systems) who interact with the SUD to achieve a desired goal. Actors are represented as stick figures.

  Types of Actors:

  - **Primary Actor**

  The Actor(s) using the system to achieve a goal.

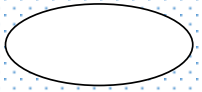
  - **Secondary Actor**

  Actors that the system needs assistance from to achieve the primary actors goal.

- **Use Case**

  A use case is a collection of possible sequences of interactions between the SUD and its Actors, relating to a particular goal. The collection of Use Cases should define all system behavior relevant to the actors to assure them that their goals will be carried out properly.

  A use case is drawn as a horizontal ellipse.

Key Terms in UCM

Use Cases:

- Hold Functional Requirements in an easy to read, easy to track text format.
- Represents the goal of an interaction between an actor and the system. The goal represents a meaningful and measurable objective for the actor.
- Records a set of paths (scenarios) that traverse an actor from a trigger event (start of the use case) to the goal (success scenarios).
- Records a set of scenarios that traverse an actor from a trigger event toward a goal but fall short of the goal (failure scenarios).
- Are multi-level: one use case can use/extend the functionality of another.
- Use Case Names Begin With a Strong Verb
- Use Cases are named using the domain terminologies

Use Cases Do Not…

- Specify user interface design. They specify the intent, not the action detail

  Mock up screens/ Prototype may be included depicting the functionality
- Specify implementation detail (unless it is of particular importance to the actor to be assured that the goal is properly met)

Key Terms in UCM

- **Use Case Relationships (Associations)**

  Associations between actors and use cases are indicated in use case diagrams by solid lines. An association exists whenever an actor is involved with an interaction described by a use case.

  There are several types of relationships that may appear on a use case diagram:

- An association between an actor and a use case
- An association between two use cases
- A generalization between two actors
- A generalization between two use cases

Key Terms in UCM

- **Includes Relationship**

  "X *includes* Y" indicates that the task "X" **has a** subtask "Y"; that is, in the process of completing task "X", task "Y" will be completed at least once.

- **Extends Relationship**

  "X *extends* Y" indicates that "X" **is a** task of the same type as "Y", but "X" is a special, more specific case of doing "Y". That is, doing X is a lot like doing Y, but X has a few extra processes to it that go above and beyond the things that must be done in order to complete Y.

Key Terms in UCM

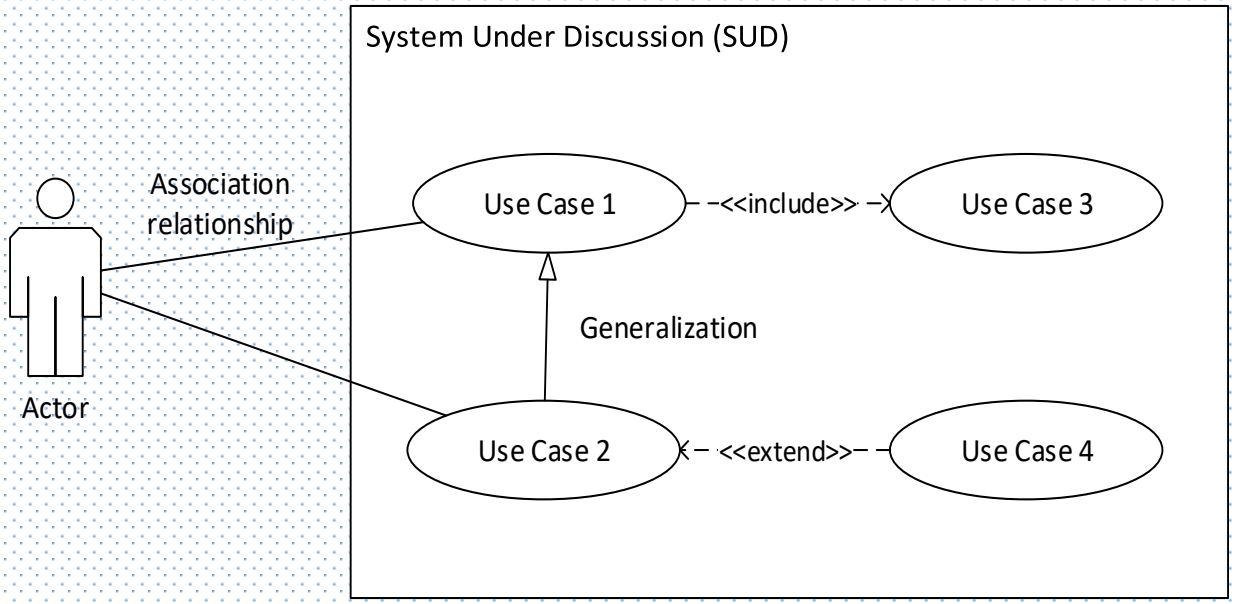
- **Use Case Diagram**

  A use case diagram shows the relationships among actors and use cases within a SUD.

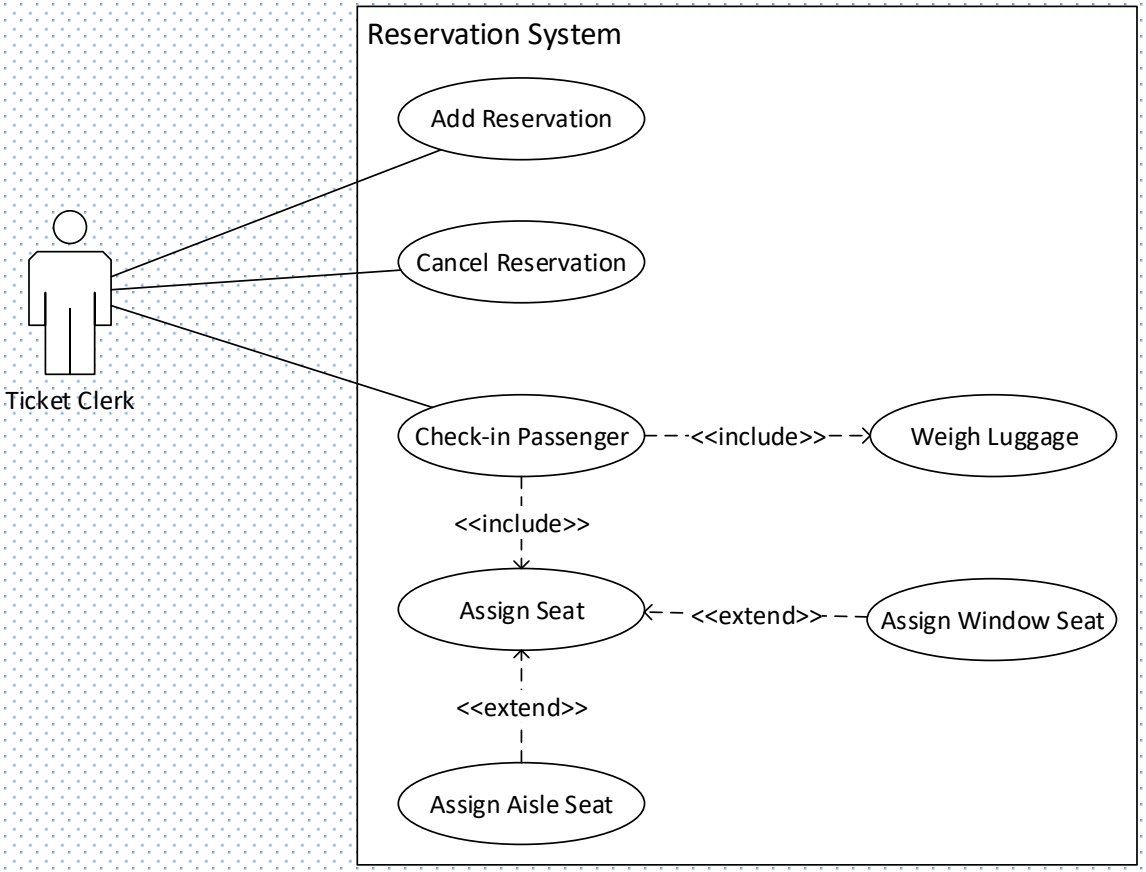
- **Use Case Model**

  A use case model is comprised of one or more use case diagrams and any supporting documentation such as use case specifications and actor definitions. Within most use case models the use case specifications tend to be the primary artifact with use case diagrams filling a supporting role as the "glue" that keeps the requirements model together.

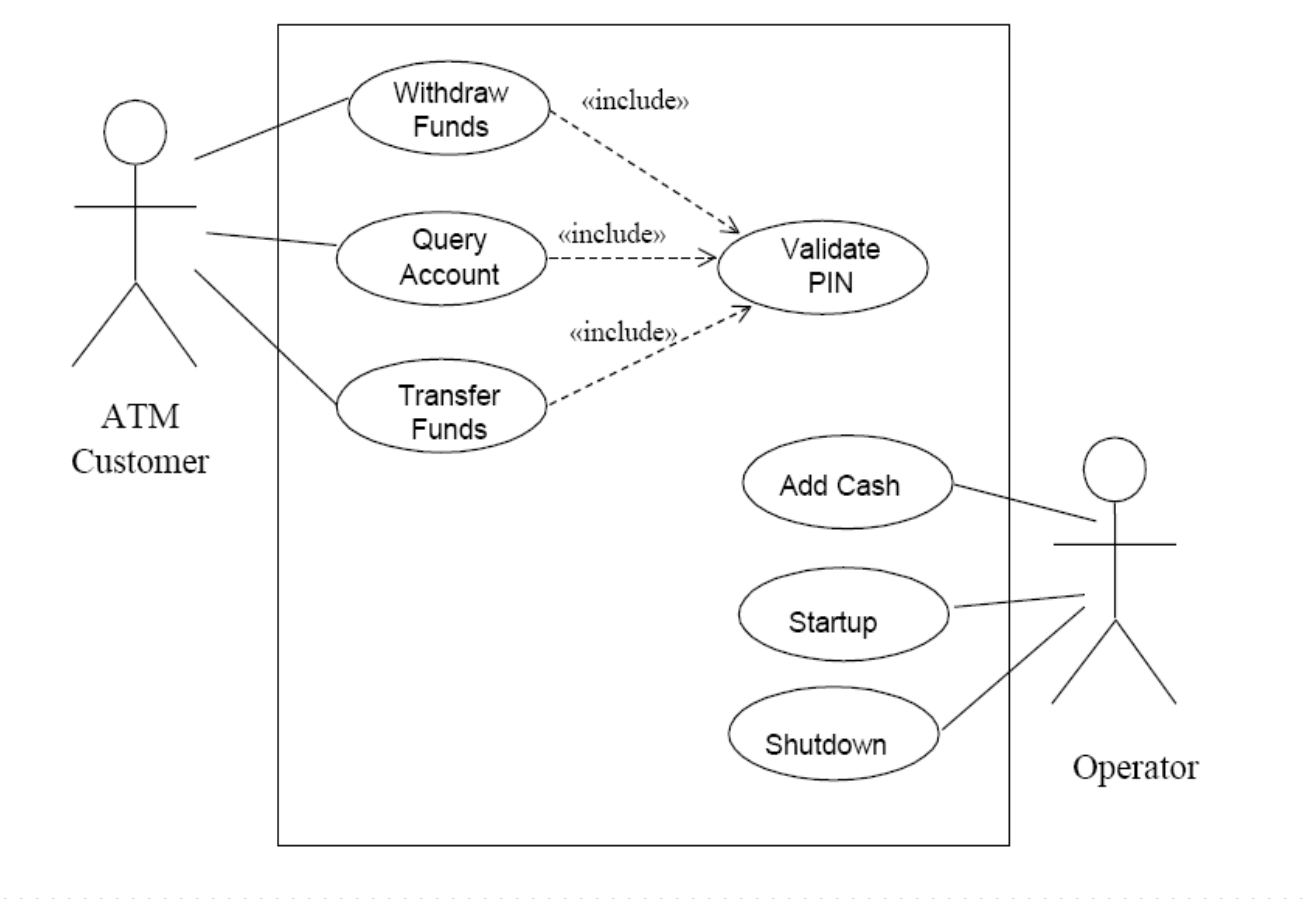
Use Case Diagram



Use Case Diagram
Airline Reservation System

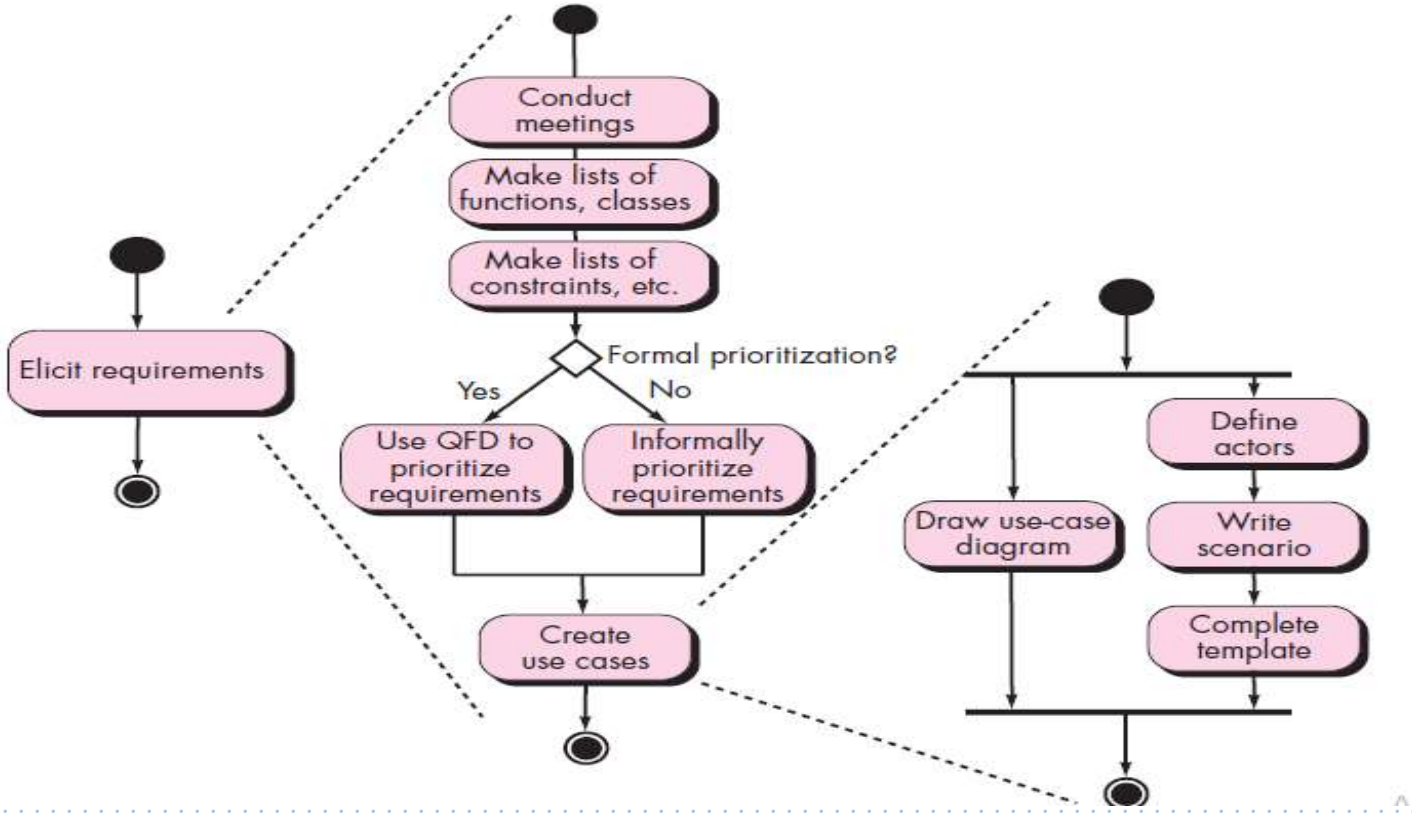


Use Case Diagram
ATM Transaction



5. Building the Requirements Model

Activity Diagram for Eliciting Requirements



Elements

- *Scenario based elements*
- *Class based elements*
- *Behavioral elements*
- *Flow – oriented elements*

Analysis Patterns

- suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

6. Negotiating Requirements

- Identify the key stakeholders
  - These are the people who will be involved in the negotiation
- Determine each of the stakeholders "win conditions"
  - Win conditions are not always obvious
- Negotiate
  - Work toward a set of requirements that lead to "win-win"

7. Validating Requirements - I

- Is each requirement consistent with the overall objective for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?

Validating Requirements - I

- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function and behavior of the system to be built.
- Has the requirements model been “partitioned” in a way that exposes progressively more detailed information about the system.
- Have requirements patterns been used to simplify the requirements model. Have all patterns been properly validated? Are all patterns consistent with customer requirements?

CHAPTER-5

REQUIREMENT ANALYSIS
DATA MODELLING
CLASS MODELLING

Department of Computer Science and Engineering
School of Engineering, Presidency University

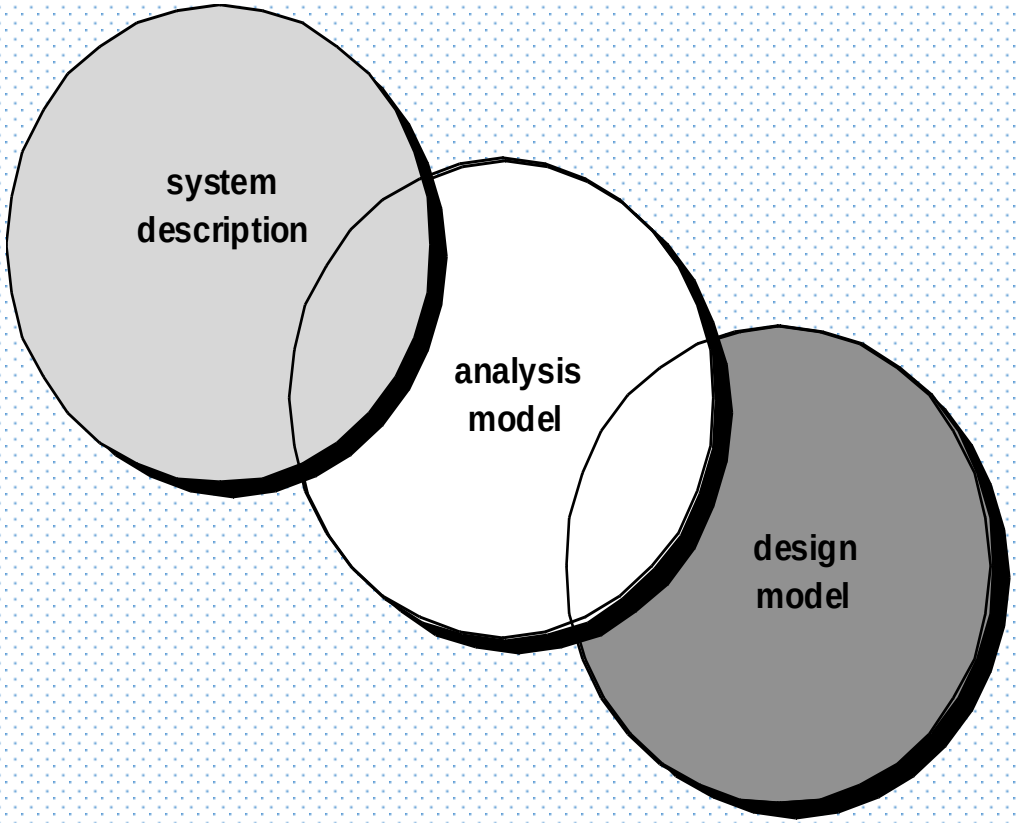
Contents

Contents

1. What is Requirement Analysis?

- Requirements analysis
  - specifies software's operational characteristics
  - indicates software's interface with other system elements
  - establishes constraints that software must meet
- Requirements analysis allows the software engineer (called an *analyst* or *modeler* in this role) to:
  - elaborate on basic requirements established during earlier requirement engineering tasks
  - build models that depict user scenarios, functional activities, problem classes and their relationships, system and class behavior, and the flow of data as it is transformed.

Requirement Analysis - A Bridge



1.1 Objectives and philosophy

- To describe what the customer requires
- To establish a basis for the creation of a software design
- To define a set of requirements that can be validated once the software is built.

1.2 Rules of Thumb

- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.
- Each element of the analysis model should add to an overall understanding of software requirements and provide insight into the information domain, function and behavior of the system.
- Delay consideration of infrastructure and other non-functional models until design.
- Minimize coupling throughout the system.
- Be certain that the analysis model provides value to all stakeholders.
- Keep the model as simple as it can be.

1.3 Domain Analysis

- Analysis is an attempt to build a model that describes the application domain
- Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain
- Object-oriented domain analysis is the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks

Domain Analysis

1. Define the domain to be investigated.
2. Collect a representative sample of applications in the domain.
3. Analyze each application in the sample.
4. Develop an analysis model for the objects.

I/O for Domain Analysis

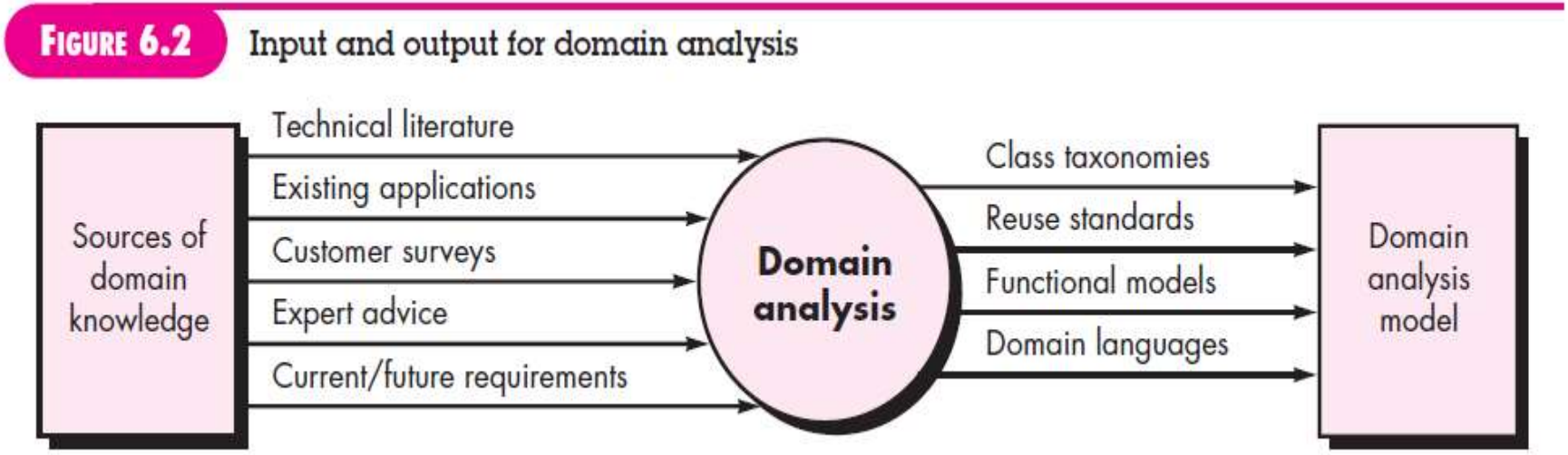
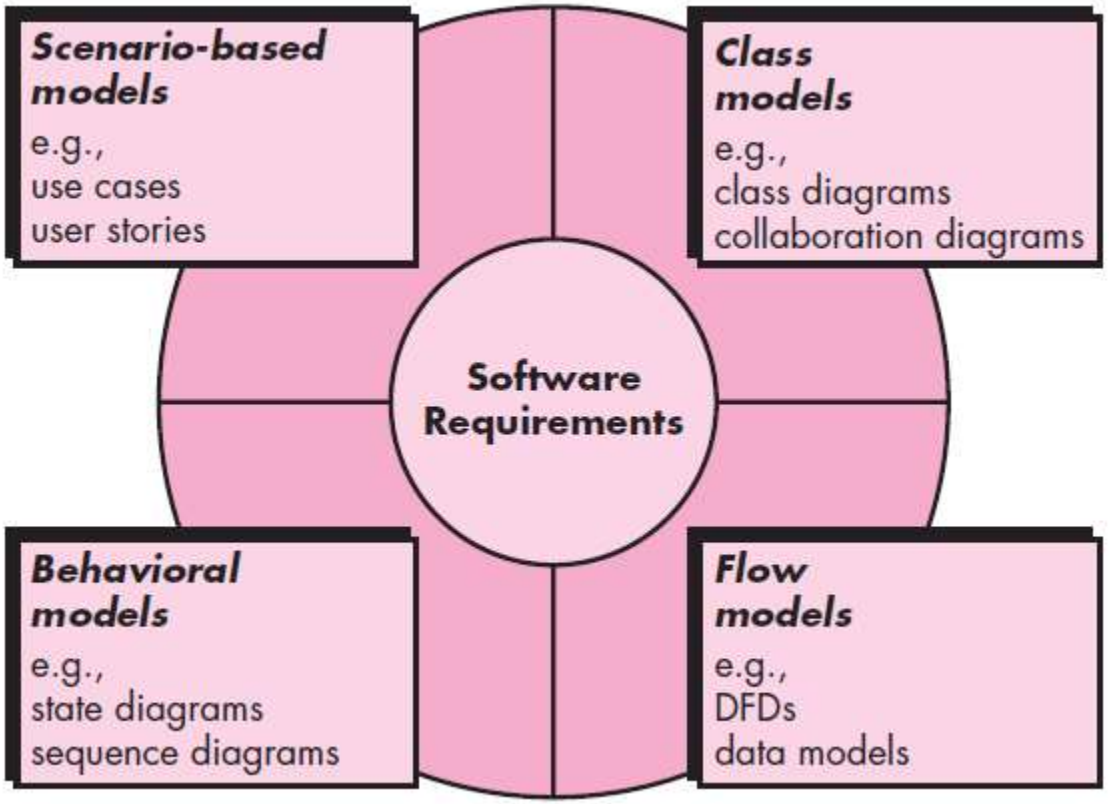


FIGURE 6.2 Input and output for domain analysis

Elements of Requirement Analysis



2. Scenario-Based Models

- “[Use-cases] a“[Use-cases] are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use-cases).” - Ivar Jacobson
- Represented via use cases

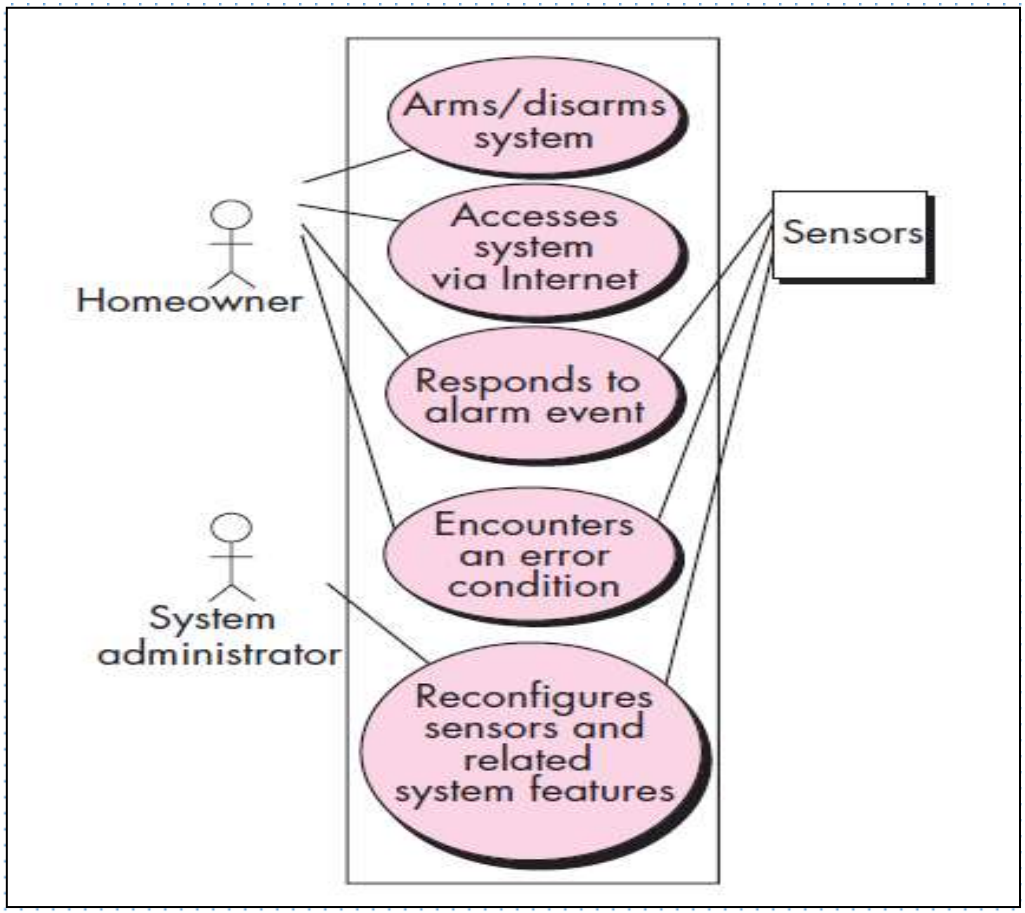
2.1 Creating Preliminary use case

- What to write about? – Inception and elicitation will provide the information for begin with use cases.

2.2 Refining Preliminary use case

2.3 Writing a formal use case

Scenario-Based Models



Use Case Diagram of Safe Home System

3. UML Models that supplement the Use Case

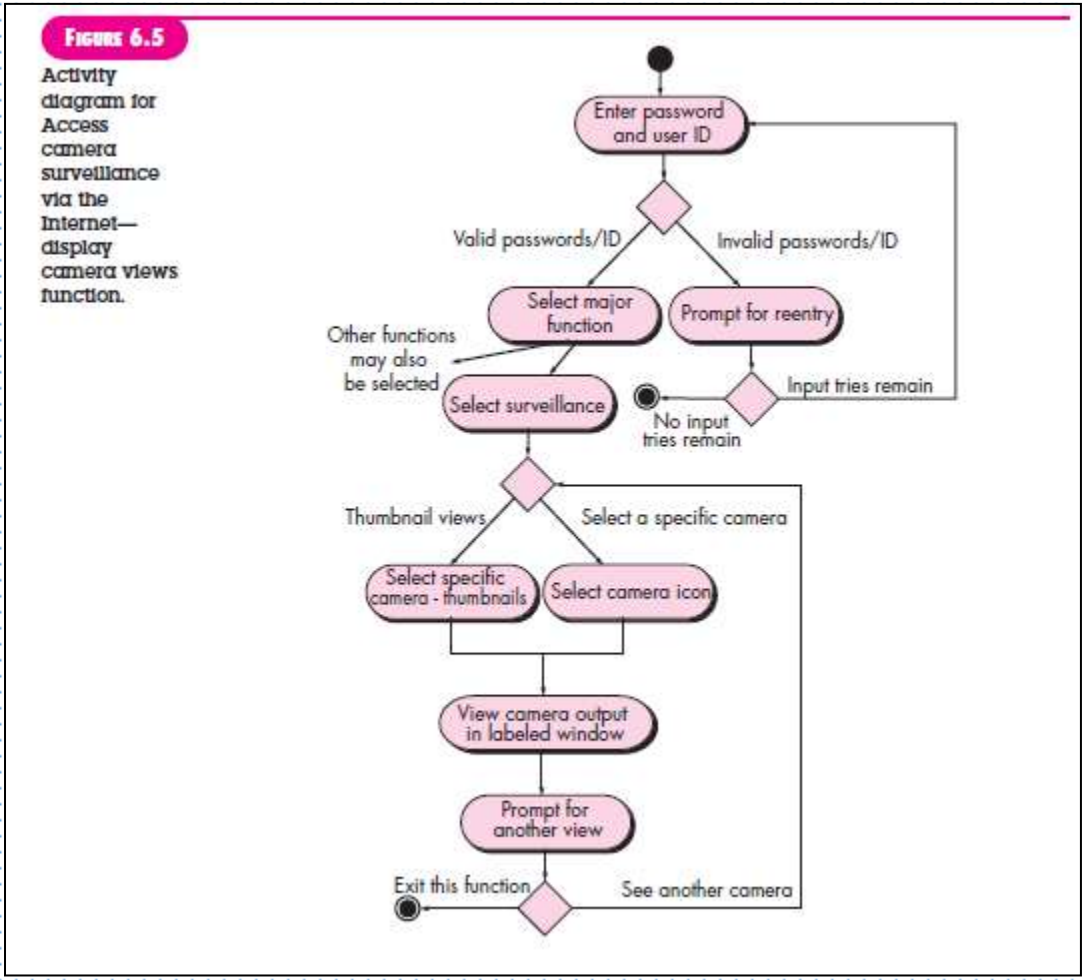
3.1. Activity Diagram

Supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario.

3.2. Swimlane Diagram

Allows the modeler to represent the flow of activities described by the use-case and at the same time indicate which actor (if there are multiple actors involved in a specific use-case) has responsibility for the action described for an activity.

Activity Diagram



Swimlane Diagram

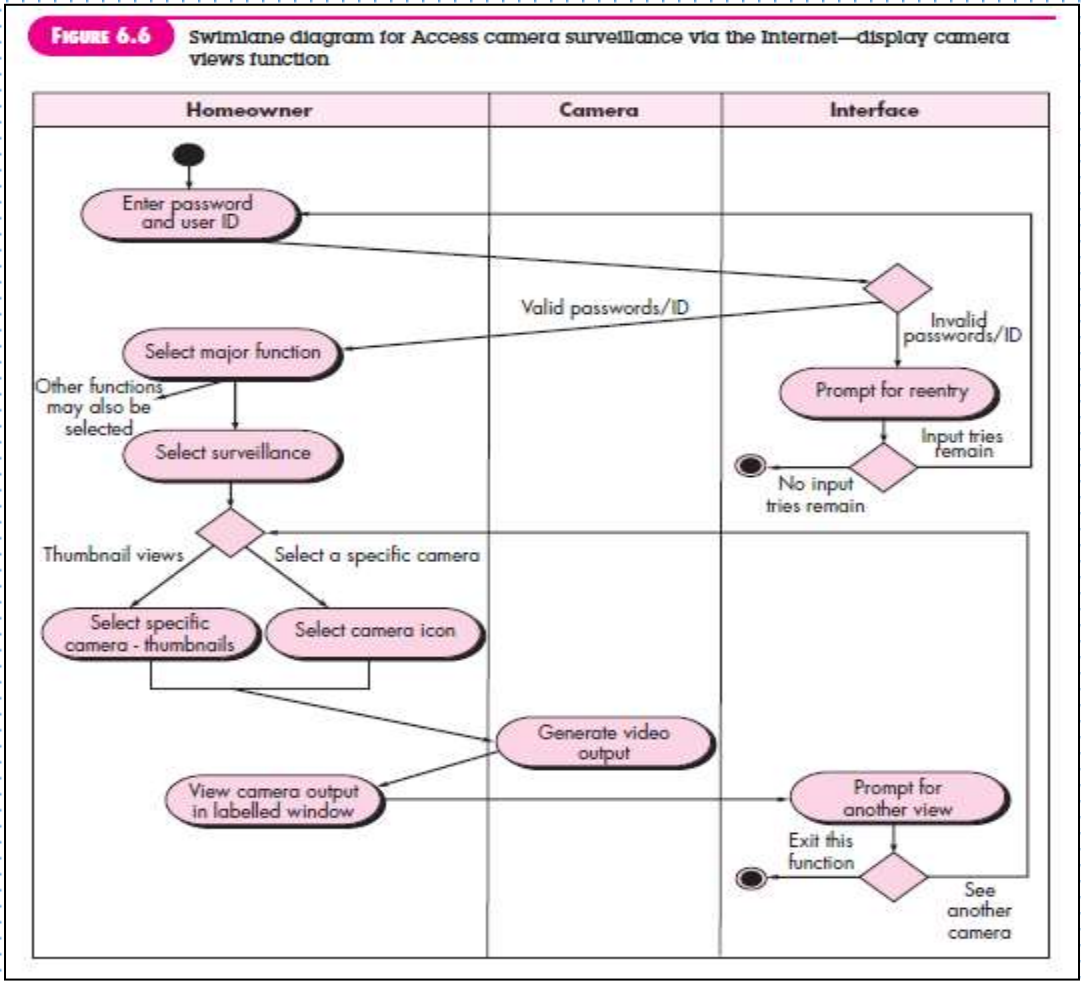


FIGURE 6.6 Swimlane diagram for Access camera surveillance via the Internet—display camera views function

4. Data Modeling

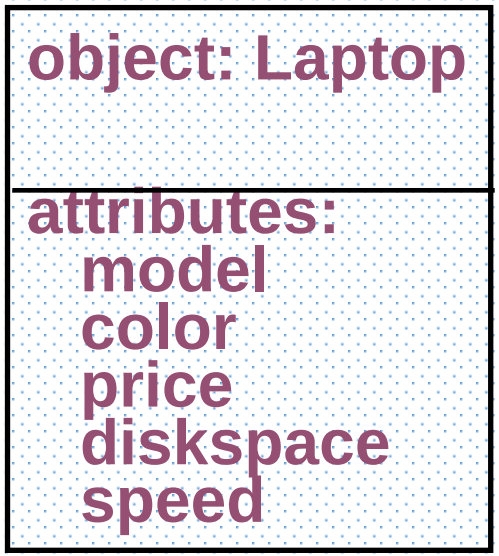
- Examines data objects independently of processing
- Focuses attention on the data domain
- Creates a model at the customer's level of abstraction
- Indicates how data objects relate to one another

What is a Data Object

- Can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call), an event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file).
- The description of the data object incorporates the data object and all of its attributes.
- A data object encapsulates data only - there is no reference within a data object to operations that act on the data.

Data Object and Attributes

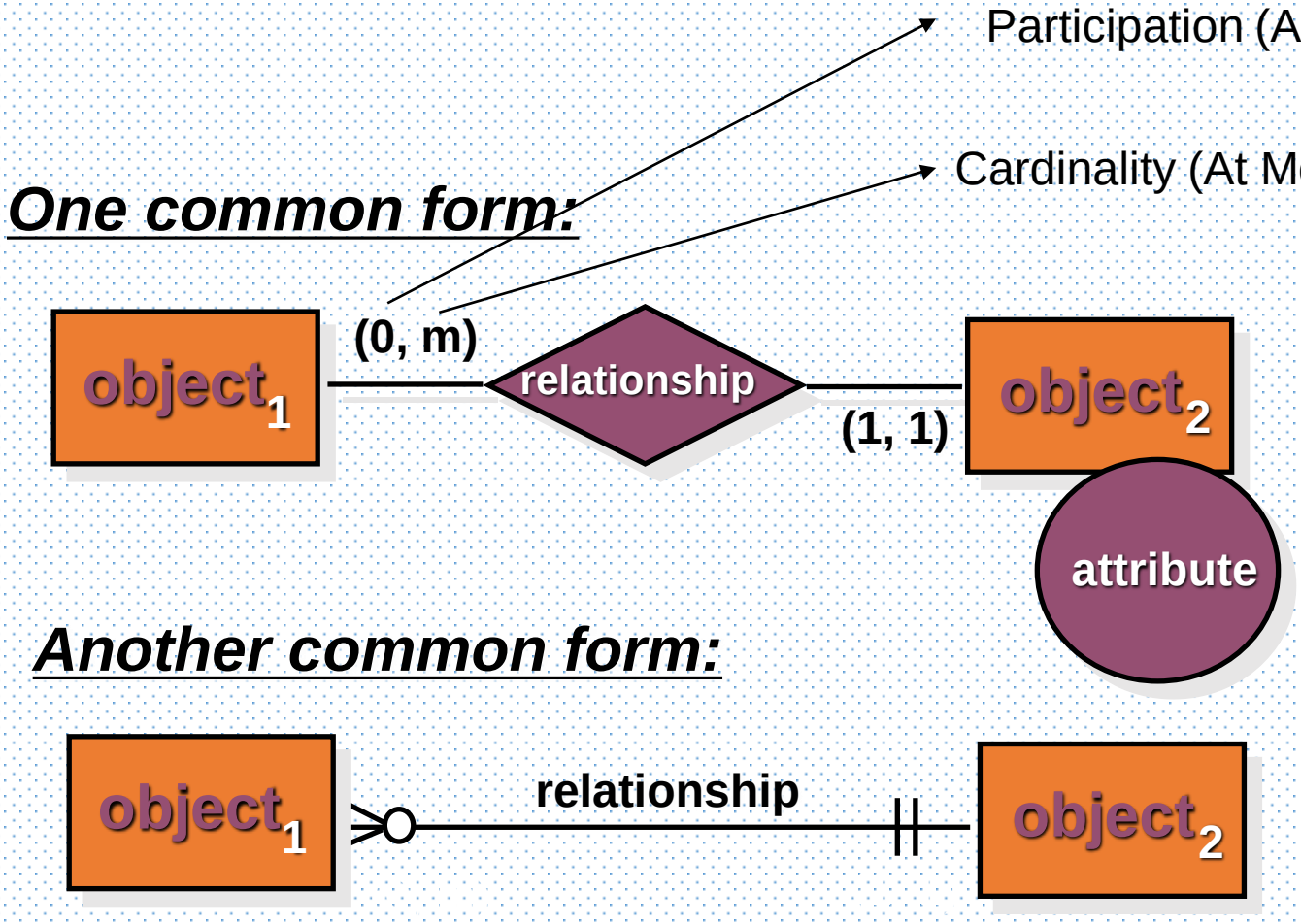
- A data object contains a set of attributes that act as an aspect, quality, characteristic, or descriptor of the object



Data Relationships

- Data objects are connected to one another in different ways.
  - A connection is established between **person** and **car** because the two objects are related.
    - A person *owns* a car
    - person *is insured to drive* a car
- The relationships *owns* and define the relevant connections between **person** and **car.**
- Several instances of a relationship can exist
- Objects can be related in many different ways

ERD Notation
(Entity Relationship Diagram)

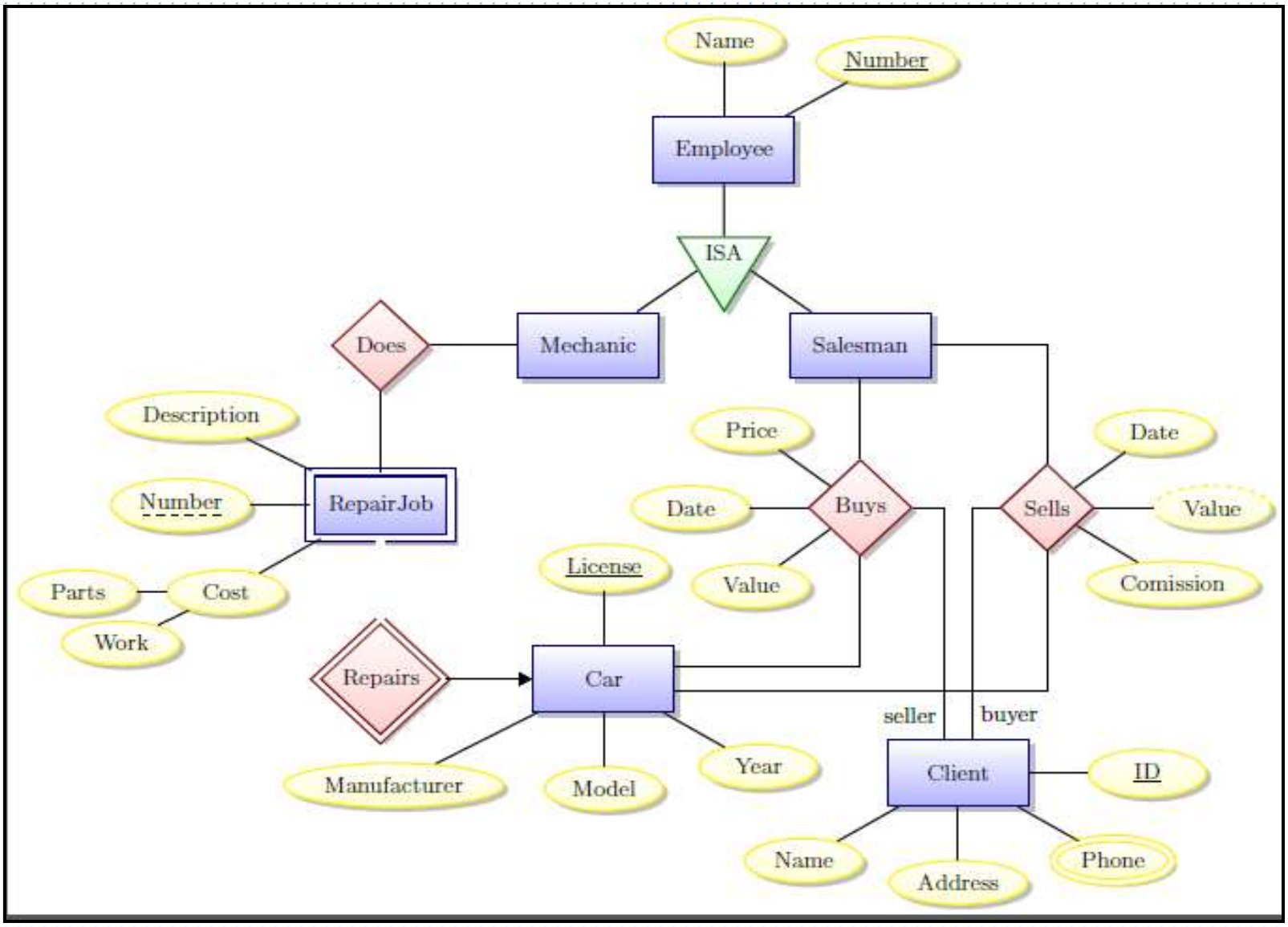


Multiplicity = Participation + Cardinality

Building an ERD

- *Level 1* - model all data objects (entities) and their “connections” to one another
- *Level 2* - model all entities and relationships
- *Level 3* - model all entities, relationships, and the attributes that provide further depth

ERD – Example


Name
Number
Employee
ISA
Does
Mechanic
Salesman
Description
Number
RepairJob
Parts
Cost
Work
Price
Date
Buys
Value
Sells
Value
Comission
License
Repairs
Car
Manufacturer
Model
Year
seller
buyer
Client
ID
Name
Address
Phone

5. Class-Based Modeling

Class-based Modeling includes:

1. Identifying classes (often from use cases as a starting point)
2. Identifying associations between classes
3. Identifying general attributes and responsibilities of classes
4. Modeling interactions between classes
5. Modeling how individual objects change state -- helps identify operations
6. Checking the model against requirements, making adjustments, iterating through the process more than once

Identifying Analysis Classes

There are two ways to identify classes:

1. Natural Language Analysis Method (based on parts of speech)
2. CRC cards

In Natural Language Analysis method, we map parts of speech to object model components:

- Nouns usually map to classes, objects, or attributes
- Verbs usually map to operations or associations

Natural Language Analysis method

Part of speech	model component	Examples
Proper noun	Instance (object)	Alice, Ace of Hearts
Common noun	Class (or attribute)	Field Officer, PlayingCard, value
Doing verb	Operation	Creates, submits, shuffles
Being verb	Inheritance	Is a kind of, is one of either
Having verb	Aggregation/Composition	Has, consists of, includes
Modal verb	Constraint	Must be
Adjective	Helps identify an attribute	a *yellow* ball (i.e. color)

Natural Language Analysis method

Categories of Classes:

- **Entity class** -- these represent persistent information tracked by a system. This is the closest parallel to "real world" objects.
- **Boundary class** -- these represent interactions between user and system. (For instance, a button, a form, a display)
- **Control class** -- usually set up to manage a given usage of the system. Often represent the control of some activity performed by a system

Entity Class

Also called as “Model” or “Business” classes

- Terms that are domain-specific in use cases
- Recurring nouns
- Real-world entities and activities tracked by system
- They correspond to “Database Table”

Examples of entity classes are:

- Employee
- Student
- Field Officer

Boundary Class

- Identify general user interface controls that initiate a use case
- Identify forms or windows for entering data into a system
- Identify messages used by system to respond to a user

Examples of boundary classes are:

- Student Attendance Report
- Employee Details Capture Form

Finding Controller Class

- Creation or update of entity objects
- Instantiation of boundary objects as they obtain information from entity objects
- Complex communication between sets of objects
- Validation of data communicated between objects or between the user and the application.

Examples of controller classes are:

- Product Controller in a Product automation system
- Home Controller in a Home automation system

Natural Language Analysis method Applicability

- **Natural Language Analysis Method** is used in:

J2EE: Front-Controller

- JSP-Servlets programming
- Swing applications
- Spring Framework

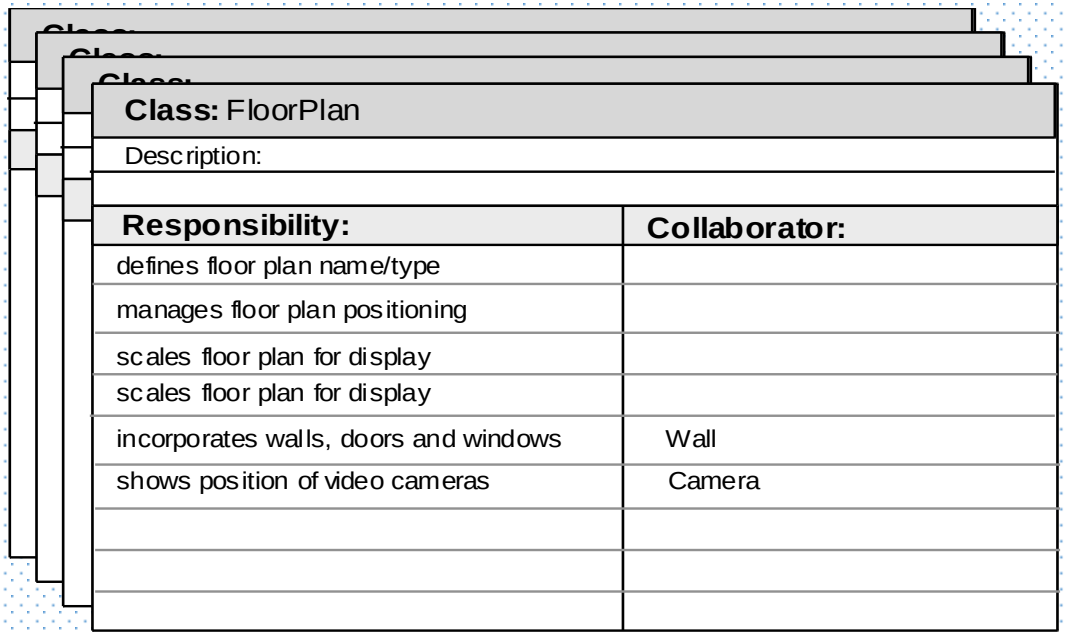
C# .NET: Page-Controller

- MVC (Model-View-Controller) applications
- Windows Forms applications
- Web Applications

CRC Cards

CRC (Class-Responsibility-Collaborator) modeling provides a mean for identifying and organizing the classes that are relevant to the system.

CRC Layout:



CRC Cards - Class

Class:

- Represents a type of object being modeled
- One card per class
- Can represent an entity class, boundary class or controller class

CRC Cards - Responsibility

- System intelligence should be distributed across classes to best address the needs of the problem
- Each responsibility should be stated as generally as possible
- Information and the behavior related to it should reside within the same class
- Information about one thing should be localized with a single class, not distributed across multiple classes.
- Responsibilities should be shared among related classes, when appropriate.

CRC Cards - Collaborations

- Classes fulfill their responsibilities in one of two ways:
  - A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
  - a class can collaborate with other classes.
- Collaborations identify relationships between classes
- Collaborations are identified by determining whether a class can fulfill each responsibility itself
- Three different generic relationships between classes
  - the *is-part-of* relationship
  - the *has-knowledge-of* relationship
  - the *depends-upon* relationship

CRC Card Session

The general process:

- Start with a scenario (usually representing a normal course through a use case)
- Identify initial classes/objects and make cards for them (this is can often be done by picking out the nouns)
- Going through a scenario helps identify responsibilities of a chosen object
- Identify collaborations between objects that have been created
- Sometimes, we'll identify a collaboration with a new object type that doesn't have a card yet -- this helps discover new classes
- When new classes are created, walk through scenarios again to discover any new responsibilities and collaborators (it's an iterative process)
- More use cases/scenarios will yield more classes, responsibilities, and collaborators

CRC Card Session

Finding responsibilities

- Look for verbs in the scenario descriptions. These often tell us what an object *does*
- Also ask what the class *knows*. This tells us what an object needs to store. Sometimes a primary responsibility of a class is management of certain unique information

Finding collaborators

- If a class has a responsibility that required it to get, or modify, information it doesn't have on its own, it will need to collaborate with another class
- Most often, one class specifically initiates the collaboration
- Usually, the collaboration is a request for information or a request to do something
- The *initiator's* card should list the helper class as a collaborator
- In this case, the initiator class *depends on* the collaborator class to accomplish its tasks

CHAPTER 6 & 7
DESIGN CONCEPTS & ARCHITECTURAL DESIGN

Department of Computer Science and Engineering
School of Engineering, Presidency University

Module 2: Software Requirements and Design (9 hrs.) – Comprehension level

Requirements Engineering: Eliciting requirements, Functional and non- Functional requirements, SRS, Requirements modelling: Developing Use Cases, Developing Activity diagram and Swim lane diagram, Design : Design concepts, Architectural design, Component based design, User interface design.

Chapter 6 – Design Concepts

1. What is Design?
2. Analysis to Design transition
3. Quality Guidelines
4. Design Principles
5. Fundamental Design Concepts
    1. Abstraction
    2. Software Architecture
    3. Separation of Concerns
    4. Modularity
    5. Functional Independence – Cohesion and Coupling
    6. OO Design Concepts
    7. Design Model and Elements

Chapter 7 – Architectural Design

1. What is Software Architecture
2. Importance of Software Architecture
3. Architectural Styles
    1. *Data-centered Architectures*
    2. *Data Flow Architecture*
    3. *Call and Return Architecture*
    4. *Layered Architecture*
4. Architectural Patterns
    1. *Concurrency*
    2. *Persistence*
    3. *Distribution*
5. Architecture Partitioning

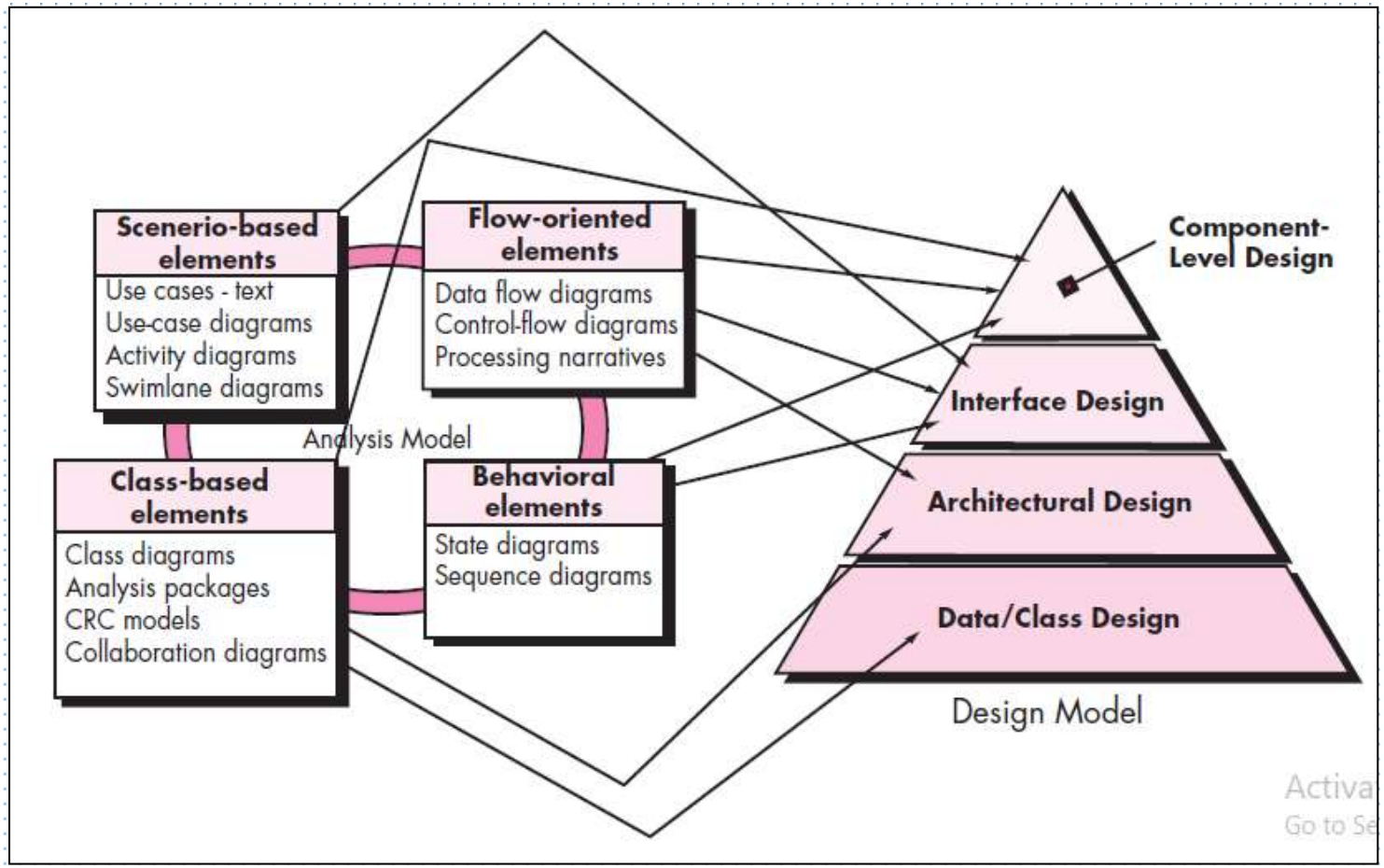
1. What is Design

Software design is the process by which an agent creates a **specification** of a **software artifact**, intended to **accomplish goals**, using a set of **basic components** and **subject to constraints.**

Good software design should exhibit:

- ***Firmness***: A program should not have any bugs that inhibit its function.
- ***Commodity***: A program should be suitable for the purposes for which it was intended.
- ***Delight***: The experience of using the program should be pleasurable one.

2. Analysis to Design Model Transition



Design and Quality

- *Design must implement all of the explicit requirements* contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- *Design must be a readable, understandable guide* for those who generate code and for those who test and subsequently support the software.
- *Design should provide a complete picture of the software*, addressing the data, functional, and behavioral domains from an implementation perspective.

3. Quality Guidelines

- *A design should exhibit an architecture* that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics and (3) can be implemented in an evolutionary fashion
  - For smaller systems, design can sometimes be developed linearly.
- *A design should be modular*; that is, the software should be logically partitioned into elements or subsystems
- *A design should contain distinct representations* of data, architecture, interfaces, and components.
- *A design should lead to data structures that are appropriate* for the classes to be implemented and are drawn from recognizable data patterns.

Quality Guidelines

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

4. Design Principles

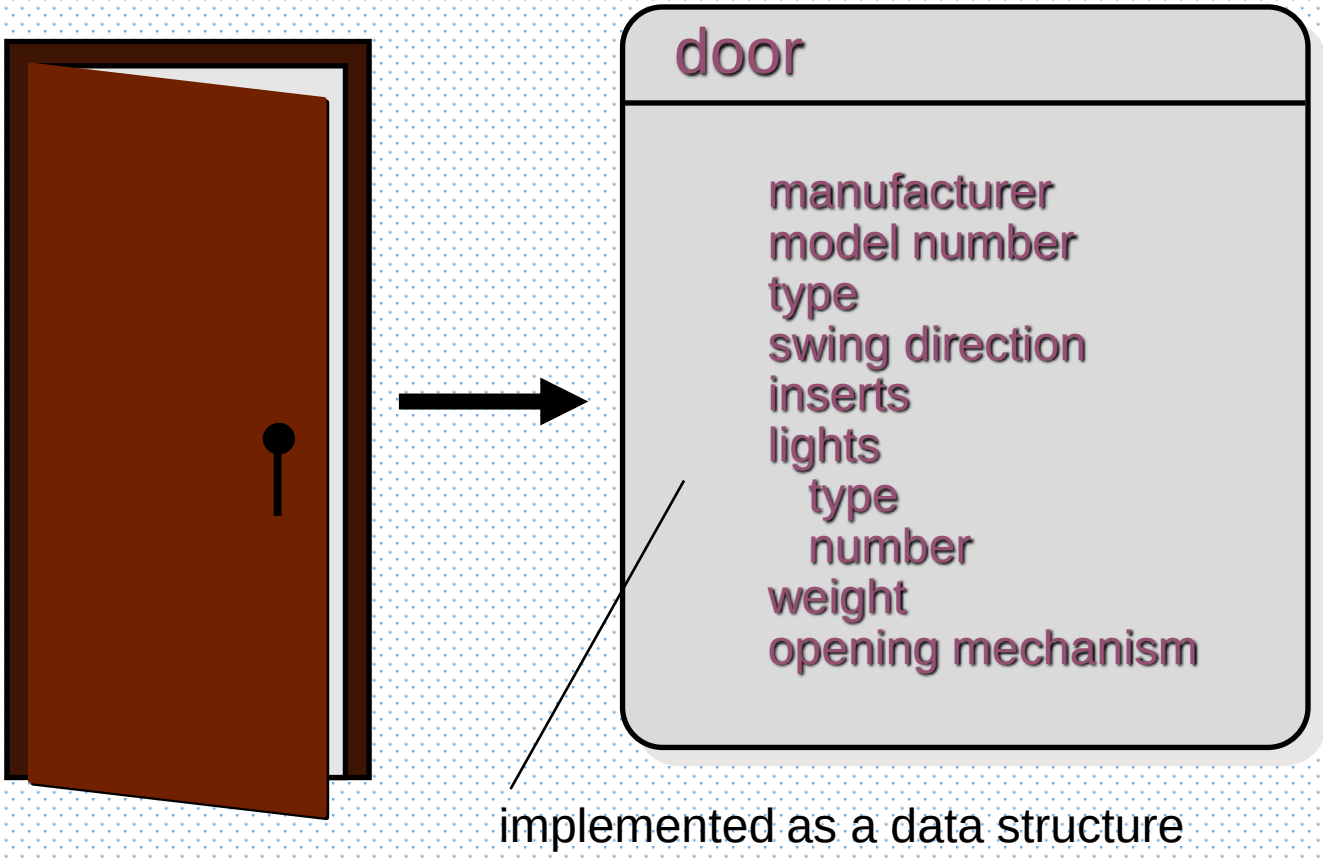
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.
- Design is not coding, coding is not design.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

From Davis [DAV95]

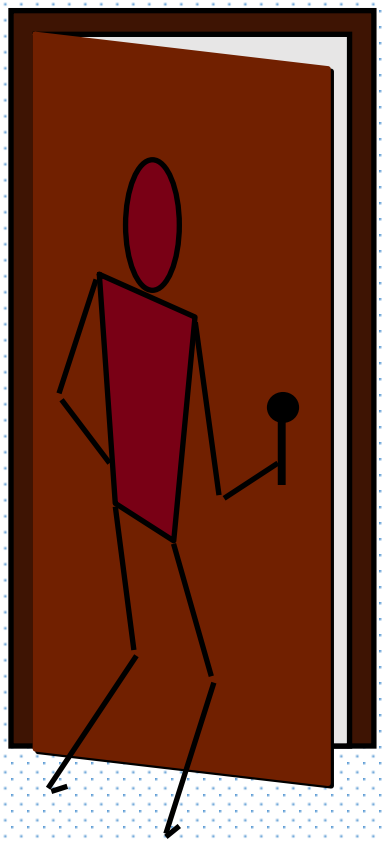
5. Fundamental Concepts

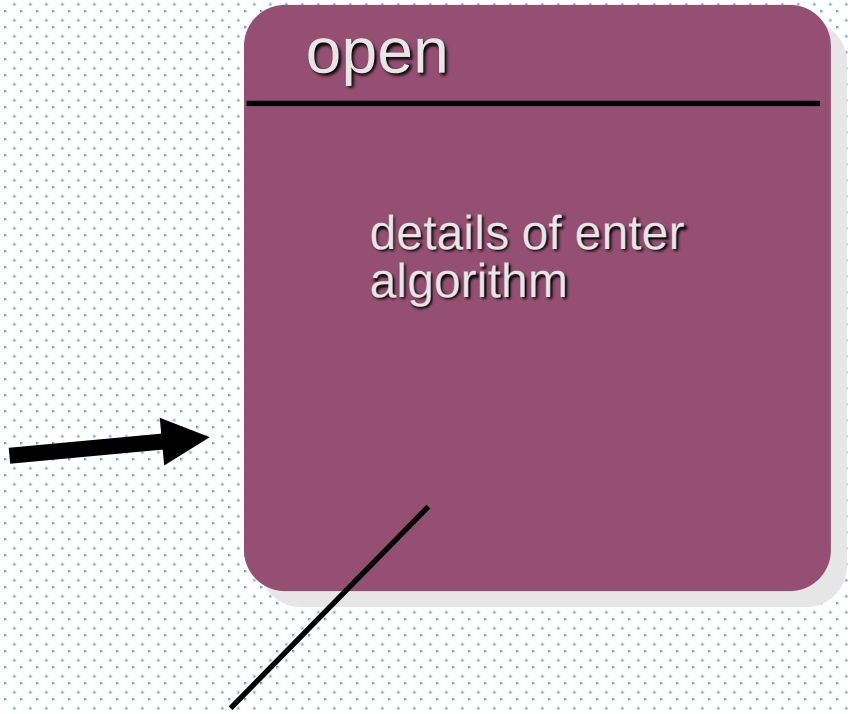
- *Abstraction* - data, procedure, control
- *Architecture* - the overall structure of the software
- *Separation of concerns* - any complex problem can be more easily handled if it is subdivided into pieces
- *Modularity* - compartmentalization of data and function
- *Functional independence* - single-minded function and low coupling
- *Design Classes* - provide design detail that will enable analysis classes to be implemented

5.1 Data Abstraction



Procedural Abstraction





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

5.2 Software Architecture

Software architecture refers to the *high level structures of a software system* and the *discipline of creating such structures and systems*.

Each structure comprises:

- Software elements
- Relations among the elements
- Properties of both elements and relations

5.3 Separation of Concerns

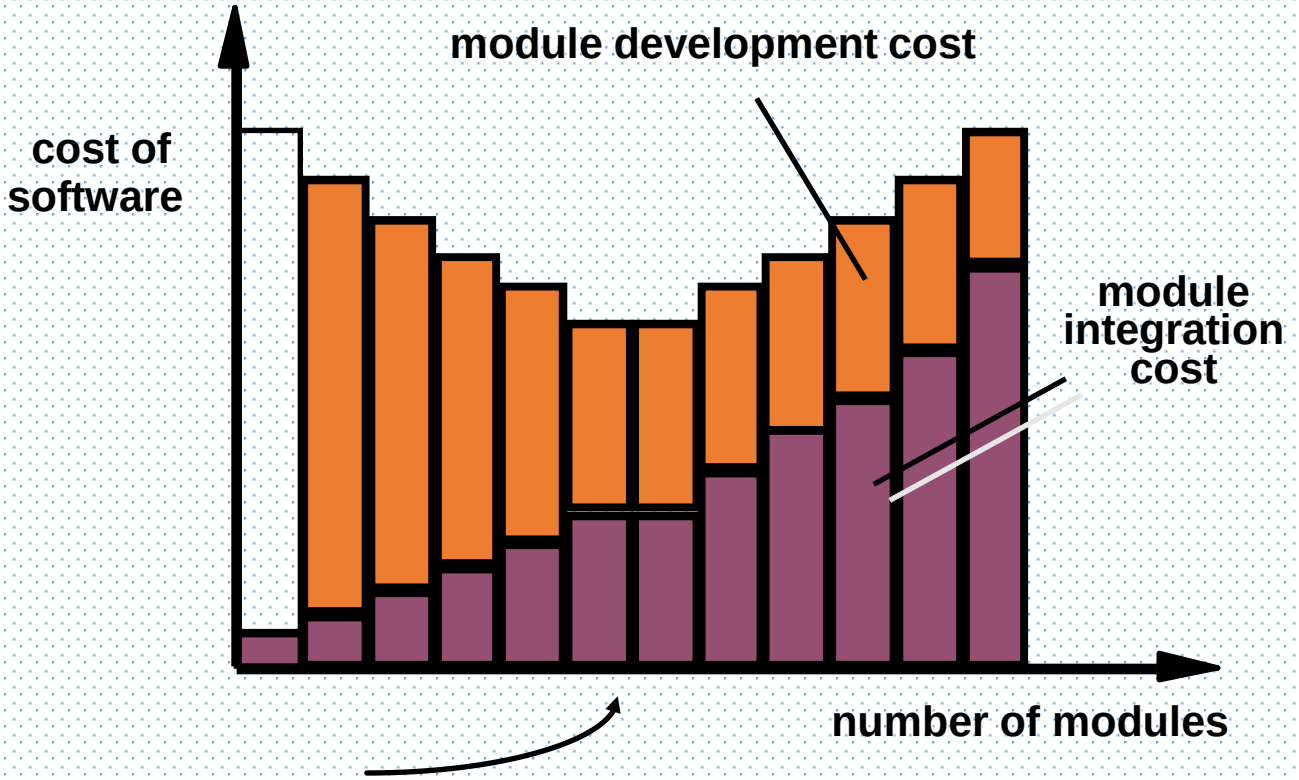
- Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more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

5.4 Modularity

- "Modularity is the single attribute of software that allows a program to be intellectually manageable".
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
  - *The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.*
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

Modularity Trade-Offs

What is the "right" number of modules for a specific software design?



5.5 Functional Independence

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- *Cohesion* is an indication of the relative functional strength of a module.
  - A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.
- *Coupling* is an indication of the relative interdependence among modules.
  - Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

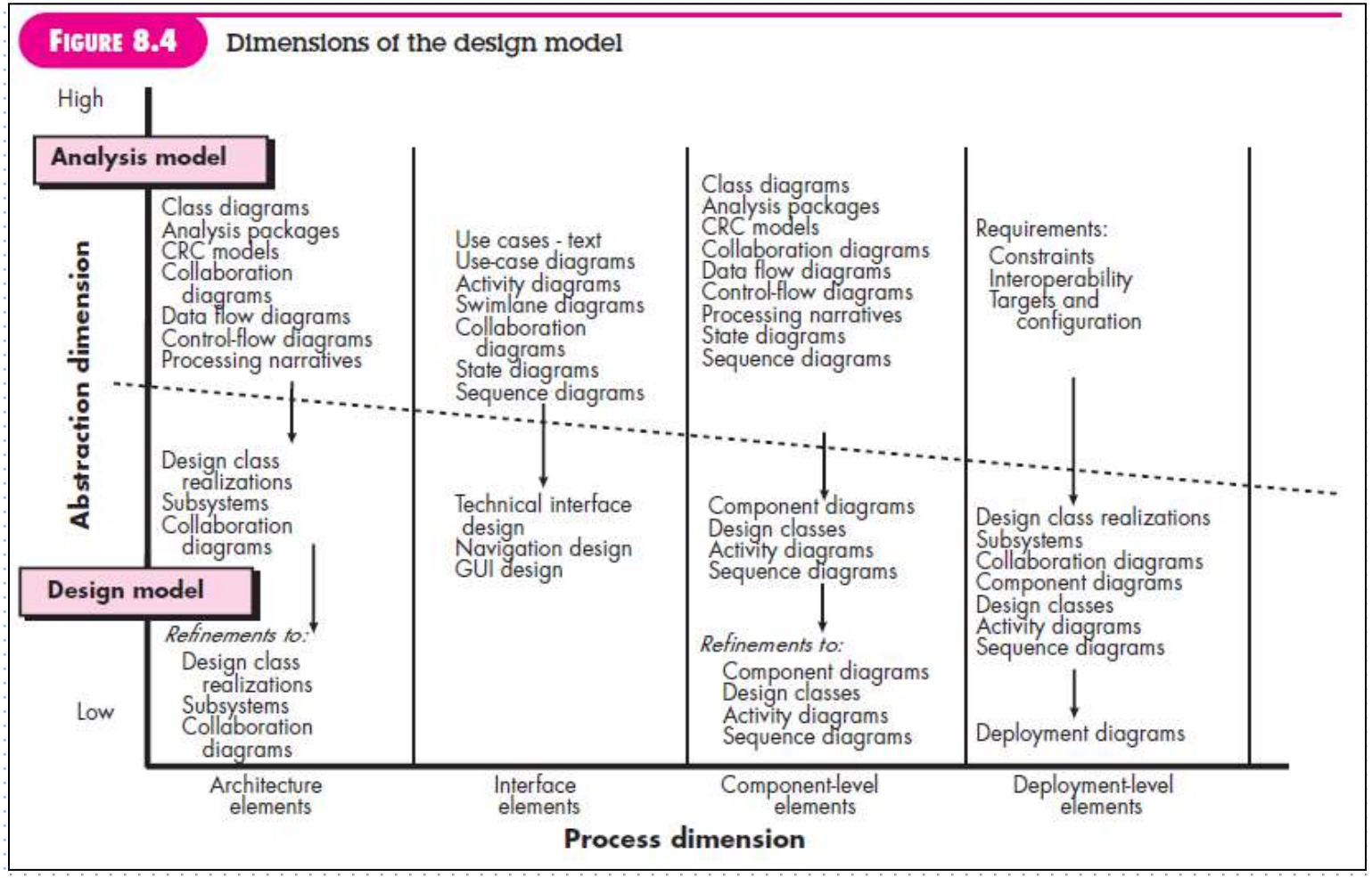
5.6 OO Design Concepts

- Design classes
  - Entity classes
  - Boundary classes
  - Controller classes
- Inheritance - all responsibilities of a superclass is immediately inherited by all subclasses
- Messages - stimulate some behavior to occur in the receiving object
- Polymorphism - a characteristic that greatly reduces the effort required to extend the design

Design Classes

- Analysis domain classes are refined during design to become entity classes
- *Boundary classes* are developed during design to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
  - Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
- *Controller classes* are designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

5.7 The Design Model



Design Model Elements

- Data elements
  - Data model --> data structures
  - Data model --> database architecture
- Architectural elements
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Patterns and “styles”
- Interface elements
  - the user interface (UI)
  - external interfaces to other systems, devices, networks or other producers or consumers of information
  - internal interfaces between various design components.
- Component elements
- Deployment elements

Architectural Design

1.1 Software Architecture

- The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

1.2 Importance of Software Architecture

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

1.3 Architectural Styles

- **An architectural style is a transformation that is imposed on the design of an entire system.**

Each style describes a system category that encompasses:

(1) Set of components (e.g., a database, computational modules) that perform a function required by a system

(2) Set of connectors that enable "communication, coordination and cooperation" among components

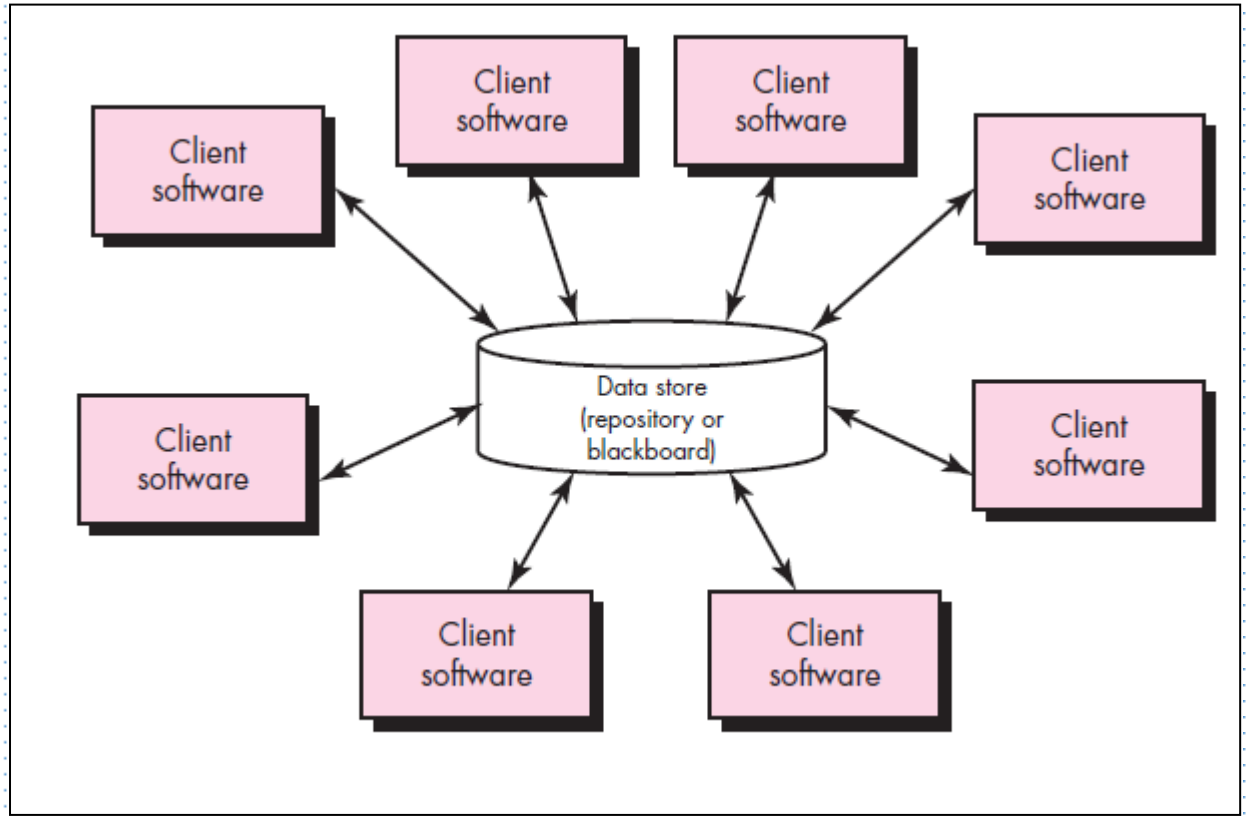
(3) Constraints that define how components can be integrated to form the system, and

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

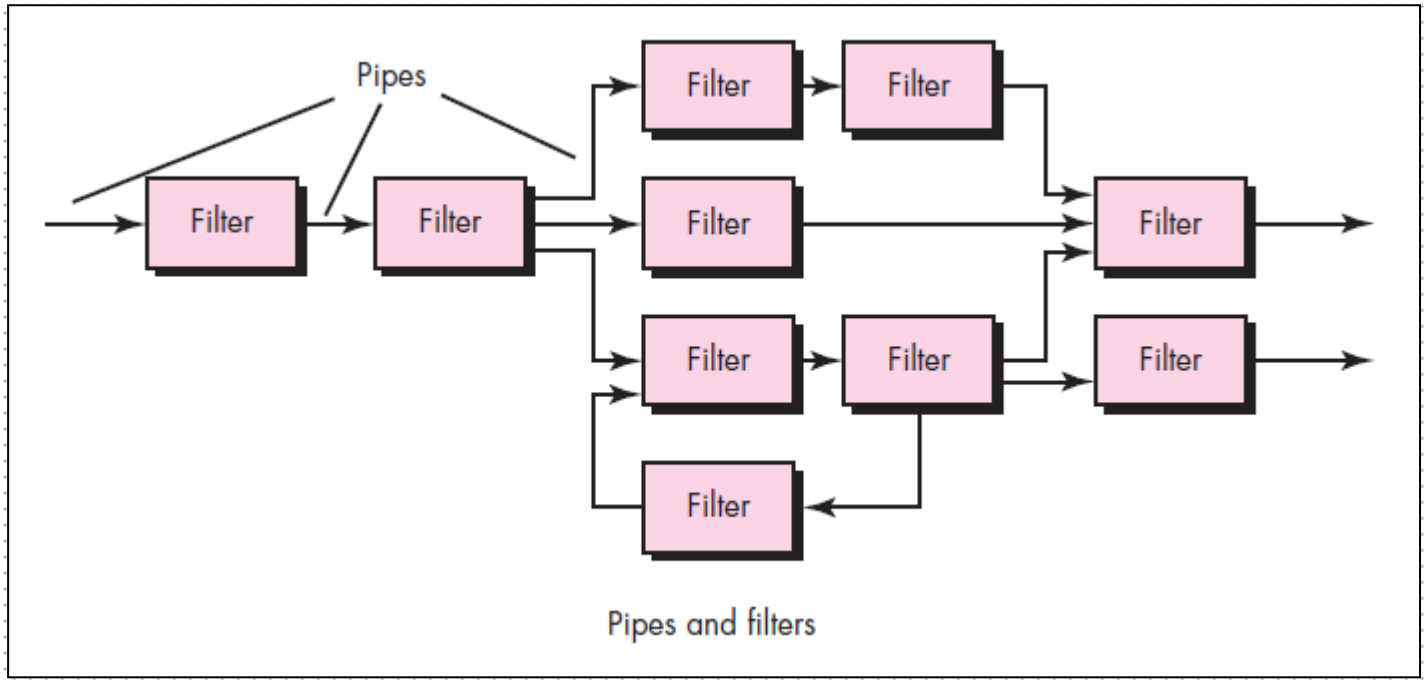
Types of Architectural Styles

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

Data-Centered Architectures

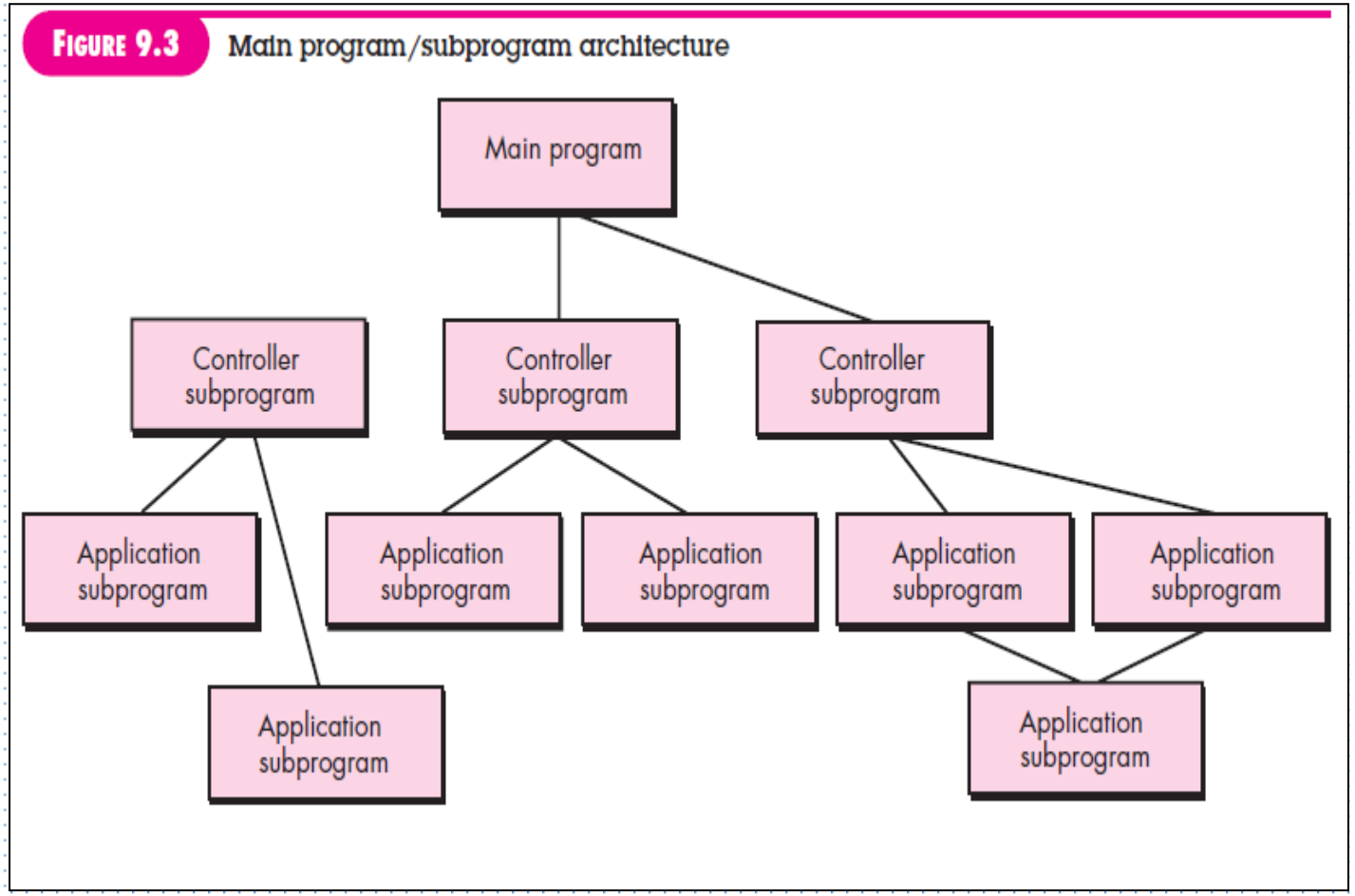


Data Flow Architectures

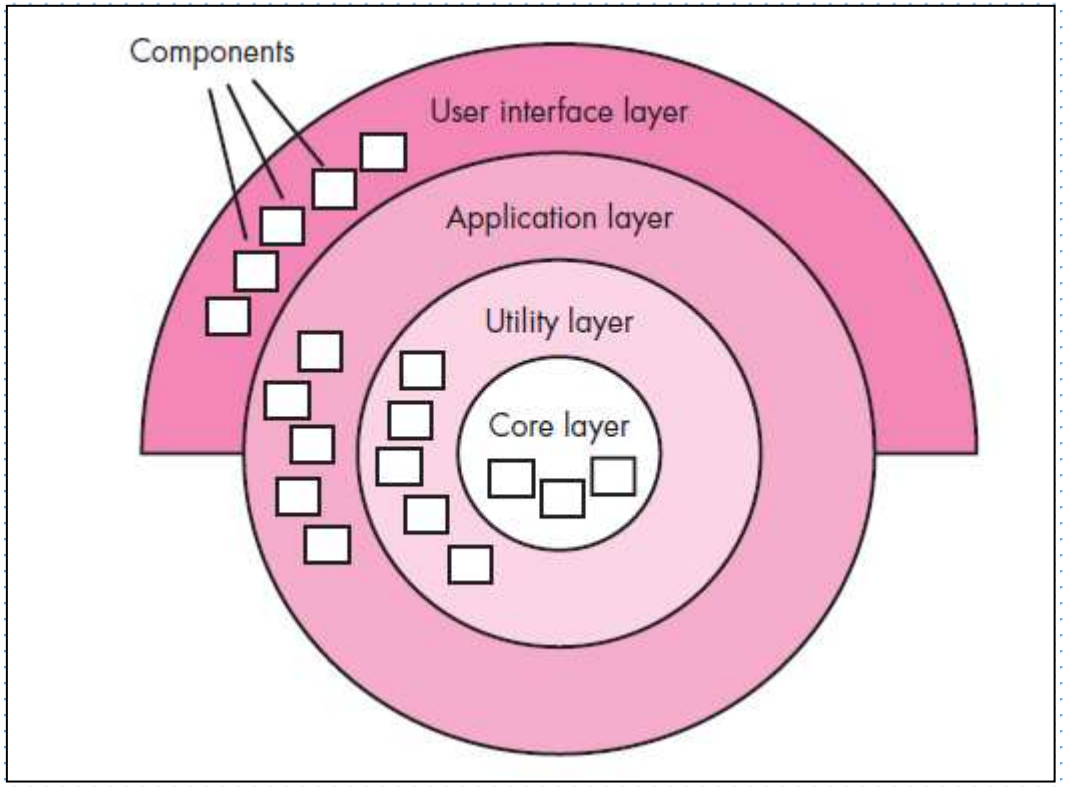


Pipes and filters

Call and Return Architectures



Layered Architectures



4. Architectural Patterns

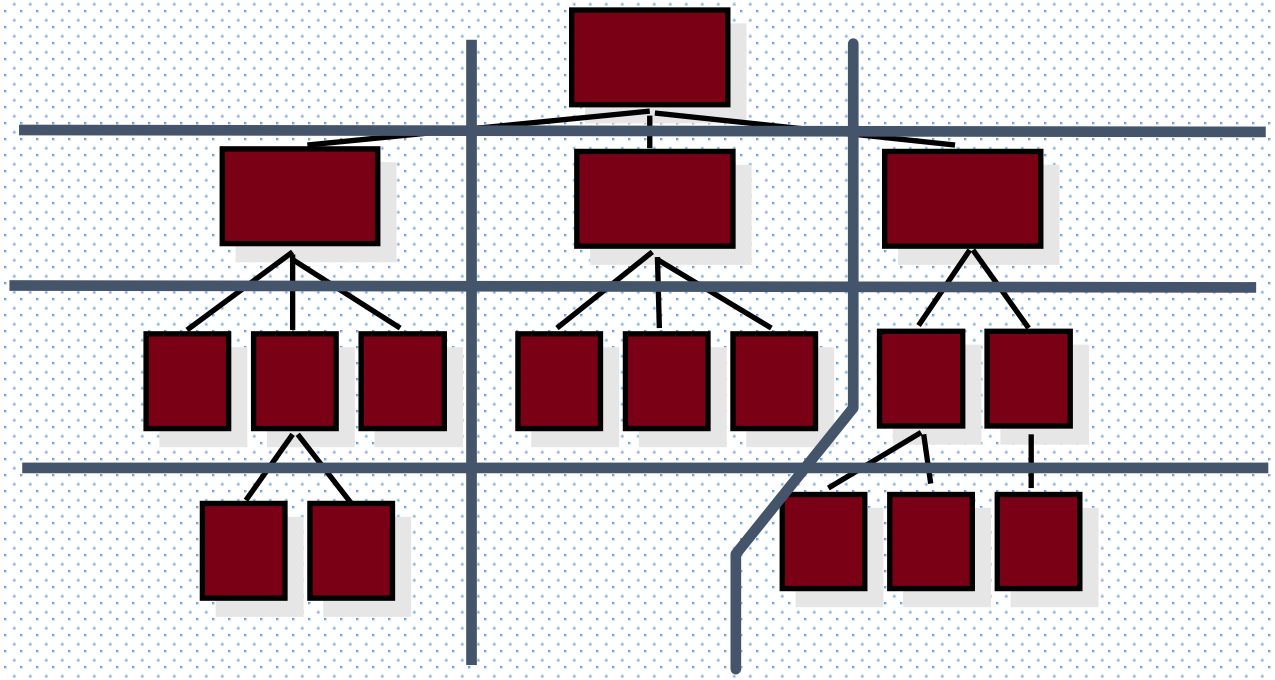
- Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints.
- The pattern proposes an architectural solution that can serve as the basis for architectural design.

Architectural Patterns

- Concurrency - applications must handle multiple tasks in a manner that simulates parallelism
  - *operating system process management* pattern
  - *task scheduler* pattern
- Persistence - Data persists if it survives past the execution of the process that created it. Two patterns are common:
  - a *database management system* pattern that applies the storage and retrieval capability of a DBMS to the application architecture
  - an *application level persistence* pattern that builds persistence features into the application architecture
- Distribution - the manner in which systems or components within systems communicate with one another in a distributed environment
  - A *broker* acts as a 'middle-man' between the client component and a server component.

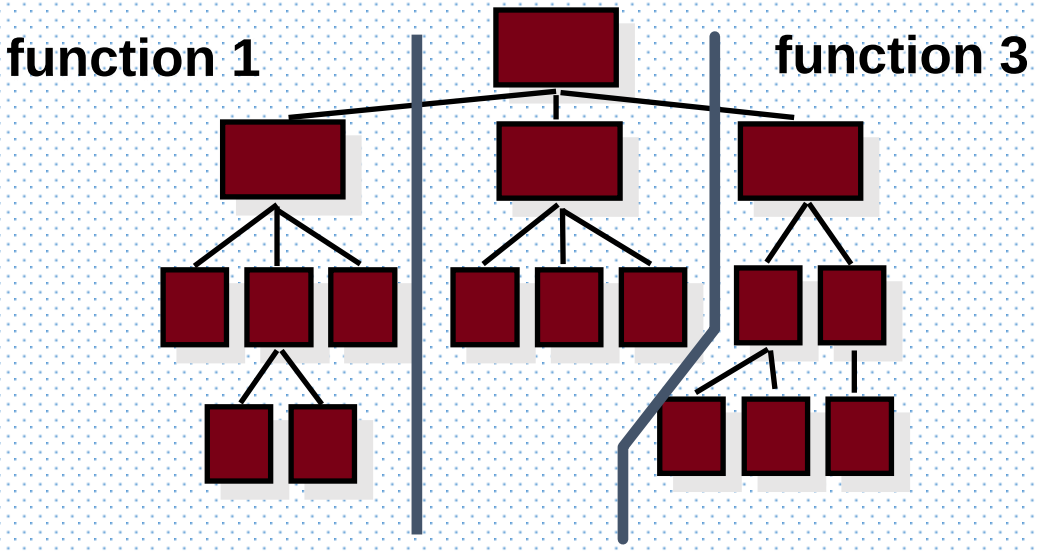
5. Partitioning the Architecture

Partitioning can be done “Horizontally” or “Vertically”



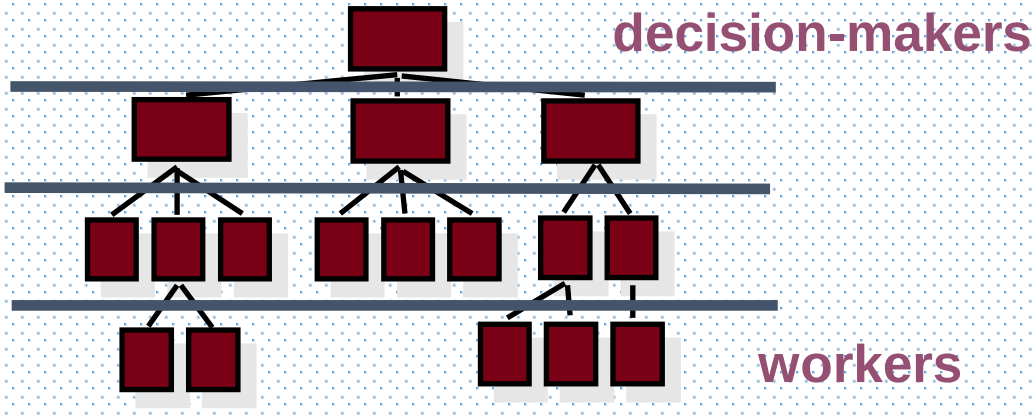
Horizontal Partitioning

- Define separate branches of the module hierarchy for each major function
- Use control modules to coordinate communication between functions



Vertical Partitioning

- Design so that decision making and work are stratified
- Decision making modules should reside at the top of the architecture



Why Data Partitioning

- Results in software that is easier to test
- Leads to software that is easier to maintain
- Results in propagation of fewer side effects
- Results in software that is easier to extend