Software Engineering (Handouts by – T.Sujanavan, Asst. Prof., MVSREC)

Unit 3

Analysis Modeling

Goals of Analysis Modeling

- Provides the first technical representation of a system
- Is easy to understand and maintain
- Deals with the problem of size by partitioning the system
- Uses graphics whenever possible
- Differentiates between <u>essential</u> information versus <u>implementation</u> information
- Helps in the tracking and evaluation of interfaces
- Provides tools other than narrative text to describe software logic and policy

A Set of Models

- **Flow-oriented modeling** – provides an indication of how data objects are transformed by a set of processing functions
- **Scenario-based modeling** – represents the system from the user's point of view
- **Class-based modeling** – defines objects, attributes, and relationships
- **Behavioral modeling** – depicts the states of the classes and the impact of events on these states

Requirements Analysis

Purpose

- Specifies the software's operational characteristics
- Indicates the software's interfaces with other system elements
- Establishes constraints that the software must meet
- Provides the software designer with a representation of information, function, and behavior
    - This is later translated into architectural, interface, class/data and component-level designs
- Provides the developer and customer with the means to assess quality once the software is built

Overall Objectives

- Three primary objectives
    - 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

- All elements of an analysis model are directly traceable to parts of the design model, and some parts overlap

Analysis Rules of Thumb

- The analysis 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 following
  - Information domain, function, and behavior of the system
- The model should delay the consideration of infrastructure and other non-functional models until the design phase
  - First complete the analysis of the problem domain
- The model should minimize coupling throughout the system
  - Reduce the level of interconnectedness among functions and classes
- The model should provide value to all stakeholders
- The model should be kept as simple as can be

Domain Analysis

- Definition
  - The identification, analysis, and specification of common, reusable capabilities within a specific application domain
  - Do this in terms of common objects, classes, subassemblies, and frameworks
- Sources of domain knowledge
  - Technical literature
  - Existing applications
  - Customer surveys and expert advice
  - Current/future requirements
- Outcome of domain analysis
  - Class taxonomies
  - Reuse standards
  - Functional and behavioral models
  - Domain languages

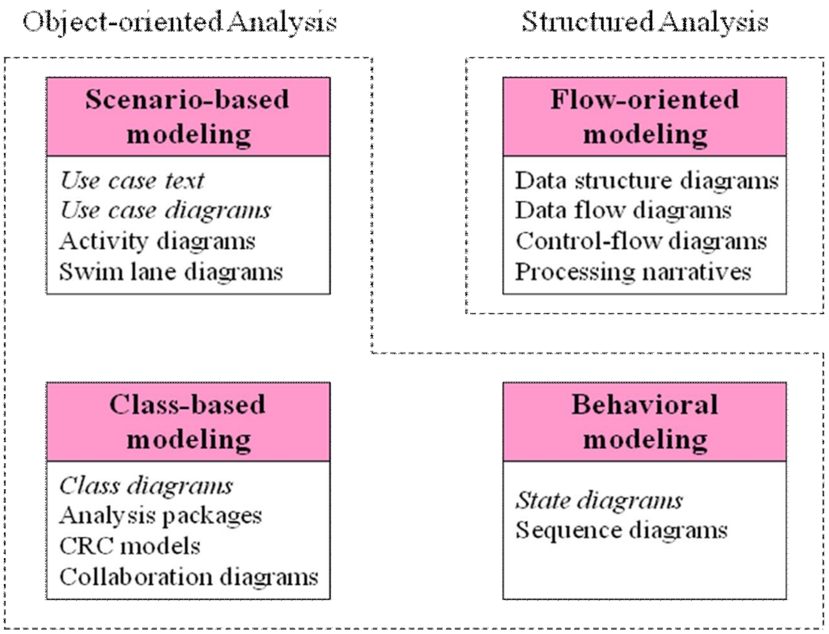
Analysis Modeling Approaches

- Structured analysis

- Considers data and the processes that transform the data as separate entities
- Data is modeled in terms of only attributes and relationships (but no operations)
- Processes are modeled to show the 1) input data, 2) the transformation that occurs on that data, and 3) the resulting output data

- Object-oriented analysis
  - Focuses on the definition of classes and the manner in which they collaborate with one another to fulfill customer requirements

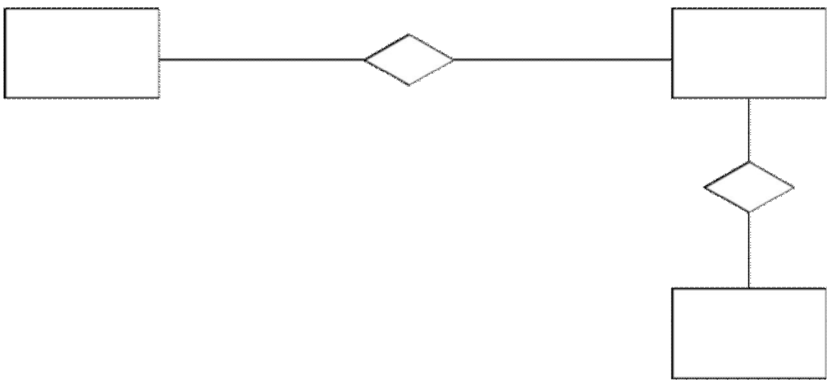
Elements of the Analysis Model



Flow-oriented Modeling

Data Modeling

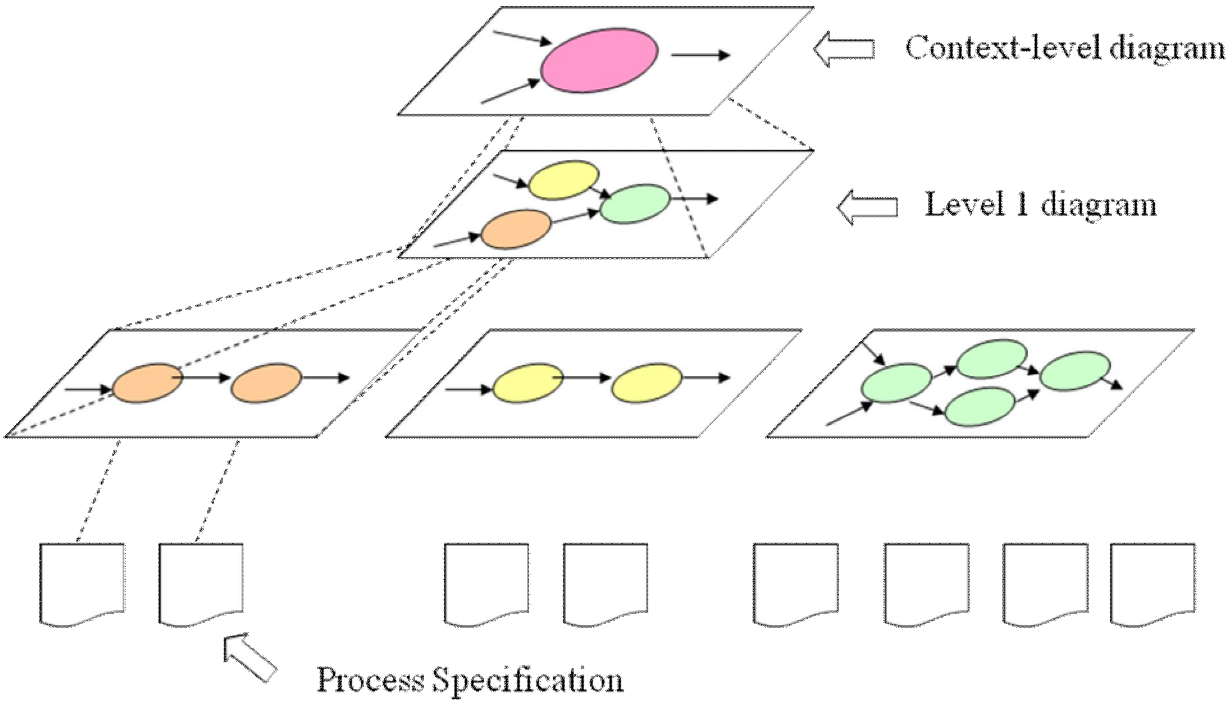
- Identify the following items
  - Data objects (Entities)
  - Data attributes
  - Relationships
  - Cardinality (number of occurrences)



Data Flow and Control Flow

- Data Flow Diagram
  - Depicts how input is transformed into output as data objects move through a system
- Process Specification
  - Describes data flow processing at the lowest level of refinement in the data flow diagrams
- Control Flow Diagram
  - Illustrates how events affect the behavior of a system through the use of state diagrams

Diagram Layering and Process Refinement



Scenario-based Modeling

Writing Use Cases

- Writing of use cases was previously described in
  Chapter 7 – Requirements Engineering
- It is effective to use the first person “I” to describe how the actor interacts with the software
- Format of the text part of a use case

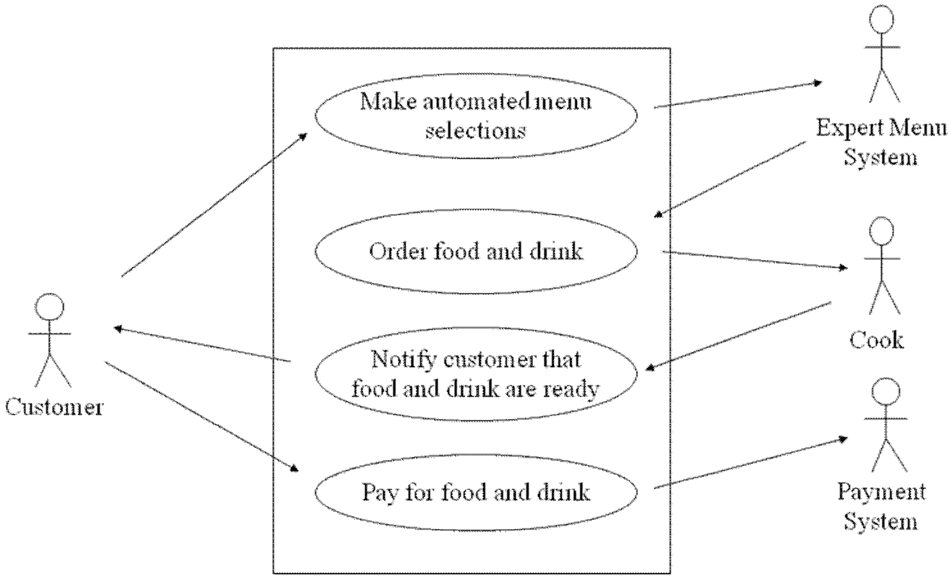
Use-case title:

Actor:

Description: I …

(See examples in Pressman textbook on pp. 188-189)

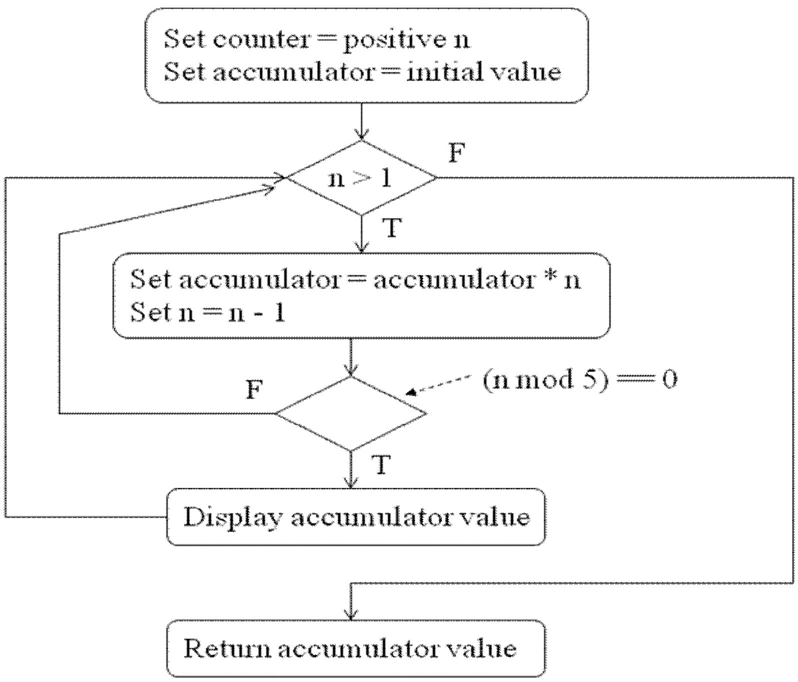
Example Use Case Diagram



Activity Diagrams

- Creation of activity diagrams was previously described in Chapter – Requirements Engineering
- Supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario
- Uses flowchart-like symbols
  - **Rounded rectangle** - represent a specific system function/action
  - **Arrow** - represents the flow of control from one function/action to another
  - **Diamond** - represents a branching decision
  - **Solid bar** – represents the fork and join of parallel activities

Example Activity Diagram



Class-based Modeling

Identifying Analysis Classes

1) Perform a grammatical parse of the problem statement or use cases
2) Classes are determined by underlining each noun or noun clause
3) A class required to implement a solution is part of the solution space
4) A class necessary only to describe a solution is part of the problem space
5) A class should NOT have an imperative procedural name (i.e., a verb)
6) List the potential class names in a table and "classify" each class according to some taxonomy and class selection characteristics
7) A potential class should satisfy nearly all (or all) of the selection characteristics to be considered a legitimate problem domain class

Potential classes	General classification	Selection Characteristics

- General classifications for a potential class
  - External entity (e.g., another system, a device, a person)
  - Thing (e.g., report, screen display)
  - Occurrence or event (e.g., movement, completion)
  - Role (e.g., manager, engineer, salesperson)
  - Organizational unit (e.g., division, group, team)
  - Place (e.g., manufacturing floor, loading dock)
  - Structure (e.g., sensor, vehicle, computer)

- Six class selection characteristics
  1) Retained information
     - Information must be remembered about the system over time
  2) Needed services
     - Set of operations that can change the attributes of a class
  3) Multiple attributes
     - Whereas, a single attribute may denote an atomic variable rather than a class
  4) Common attributes
     - A set of attributes apply to all instances of a class
  5) Common operations

        – A set of operations apply to all instances of a class

6) Essential requirements
    – Entities that produce or consume information

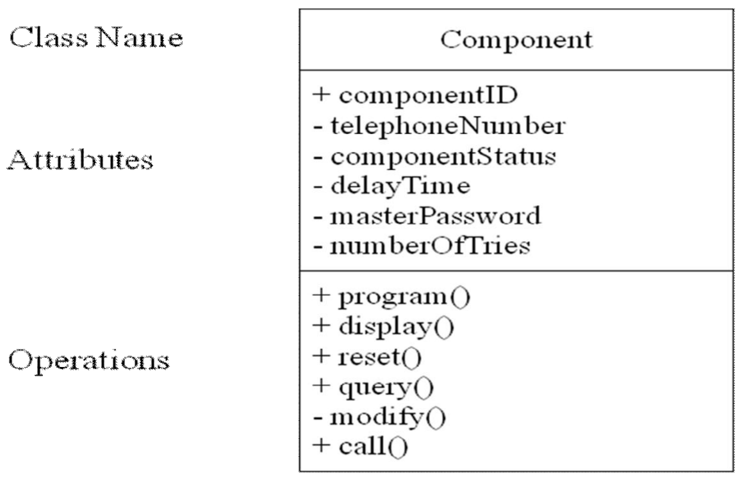
Defining Attributes of a Class

- Attributes of a class are those nouns from the grammatical parse that reasonably belong to a class
- Attributes hold the values that describe the current properties or state of a class
- An attribute may also appear initially as a potential class that is later rejected because of the class selection criteria
- In identifying attributes, the following question should be answered
    - What data items (composite and/or elementary) will fully define a specific class in the context of the problem at hand?
- Usually an item is not an attribute if more than one of them is to be associated with a class

Defining Operations of a Class

- Operations define the behavior of an object
- Four categories of operations
    - Operations that manipulate data in some way to change the state of an object (e.g., add, delete, modify)
    - Operations that perform a computation
    - Operations that inquire about the state of an object
    - Operations that monitor an object for the occurrence of a controlling event
- An operation has knowledge about the state of a class and the nature of its associations
- The action performed by an operation is based on the current values of the attributes of a class
- Using a grammatical parse again, circle the verbs; then select the verbs that relate to the problem domain classes that were previously identified

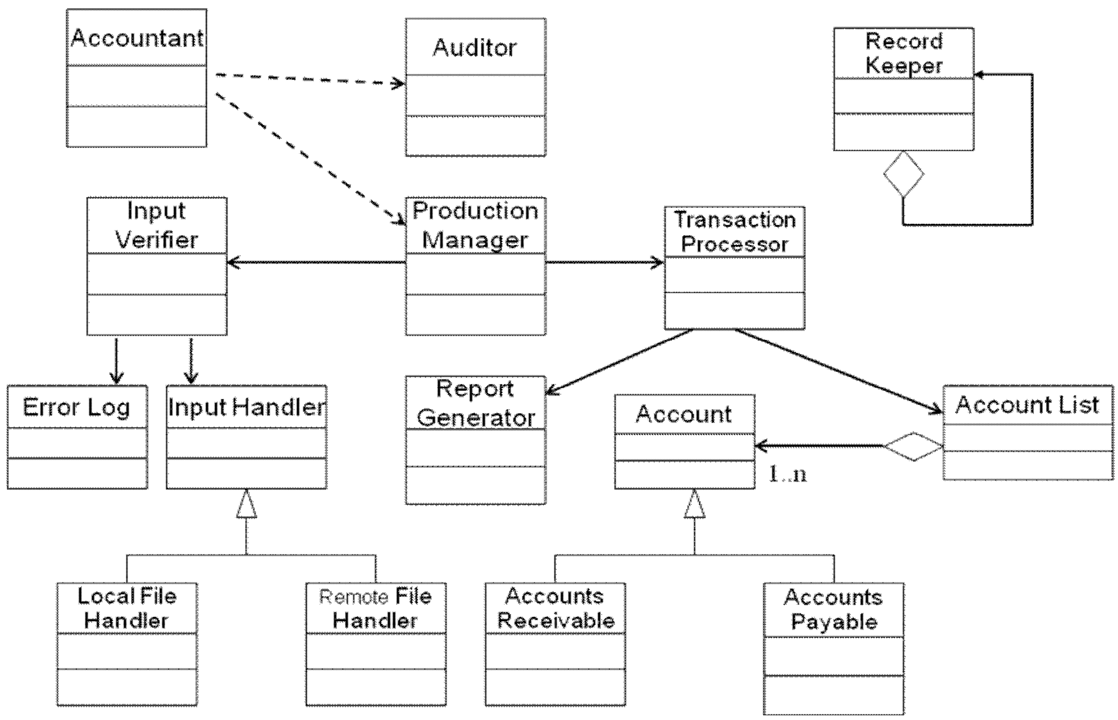
Example Class Box



Association, Generalization and Dependency (Ref: Fowler)

- Association
  - Represented by a solid line between two classes directed from the source class to the target class
  - Used for representing (i.e., pointing to) object types for attributes
  - May also be a <u>part-of</u> relationship (i.e., <u>aggregation)</u>, which is represented by a diamond-arrow
- Generalization
  - Portrays inheritance between a super class and a subclass
  - Is represented by a line with a triangle at the target end
- Dependency
  - A dependency exists between two elements if changes to the definition of one element (i.e., the source or supplier) may cause changes to the other element (i.e., the client)
  - Examples
    - One class calls a method of another class
    - One class utilizes another class as a parameter of a method

Example Class Diagram



Behavioral Modeling

Creating a Behavioral Model

1) Identify events found within the use cases and implied by the attributes in the class diagrams
2) Build a state diagram for each class, and if useful, for the whole software system

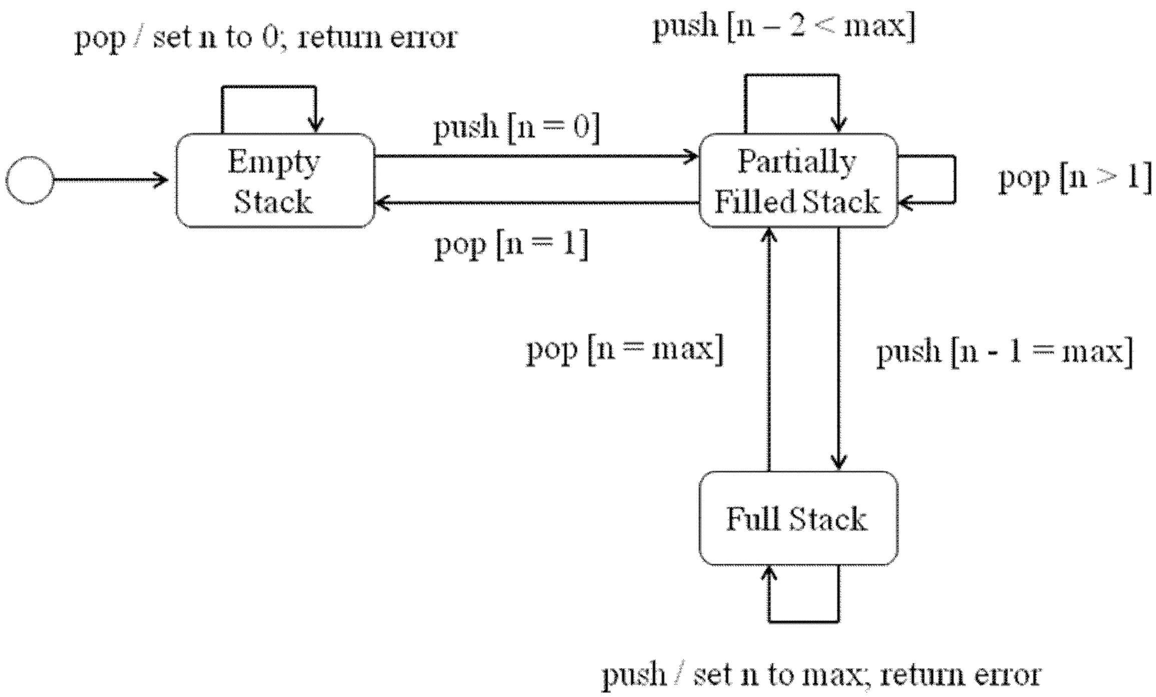
Identifying Events in Use Cases

- An event occurs whenever an actor and the system exchange information
- An event is NOT the information that is exchanged, but rather the fact that information has been exchanged
- Some events have an explicit impact on the flow of control, while others do not
  - An example is the reading of a data item from the user versus comparing the data item to some possible value

Building a State Diagram

- A state is represented by a rounded rectangle
- A transition (i.e., event) is represented by a labeled arrow leading from one state to another
  - Syntax: trigger-signature [guard]/activity
- The <u>active state</u> of an object indicates the current overall status of the object as is goes through transformation or processing
  - A state name represents one of the possible active states of an object
- The <u>passive state</u> of an object is the current value of all of an object's attributes
  - A guard in a transition may contain the checking of the passive state of an object

Example State Diagram



Design Engineering

Introduction

Five Notable Design Quotes

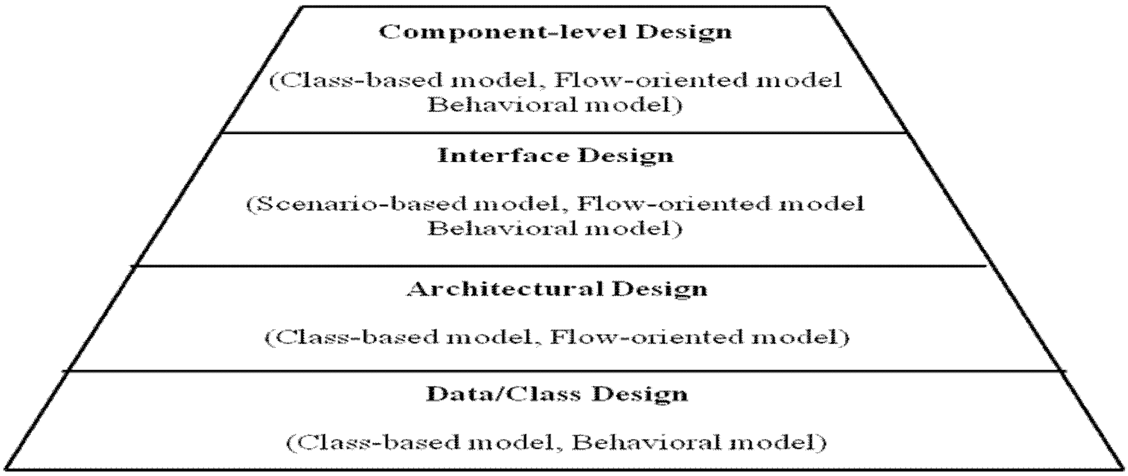
- "Questions about whether design is necessary or affordable are quite beside the point; design is inevitable. The alternative to good design is bad design, [rather than] no design at all." **Douglas Martin**
- "You can use an eraser on the drafting table or a sledge hammer on the construction site." **Frank Lloyd Wright**
- "The public is more familiar with bad design than good design. If is, in effect, conditioned to prefer bad design, because that is what it lives with; the new [design] becomes threatening, the old reassuring." **Paul Rand**
- "A common mistake that people make when trying to design something completely foolproof was to underestimate the ingenuity of complete fools." **Douglas Adams**
- "Every now and then go away, have a little relaxation, for when you come back to your work your judgment will be surer. Go some distance away because then the work appears smaller and more of it can be taken in at a glance and a lack of harmony and proportion is more readily seen." **Leonardo DaVinci**

Purpose of Design

- Design is where customer requirements, business needs, and technical considerations <u>all come together</u> in the formulation of a product or system
- The design model provides detail about the software data structures, architecture, interfaces, and components
- The design model can be assessed for quality and be improved before code is generated and tests are conducted
  - Does the design contain errors, inconsistencies, or omissions?
  - Are there better design alternatives?
  - Can the design be implemented within the constraints, schedule, and cost that have been established?
- A designer must practice <u>diversification</u> and <u>convergence</u>
  - The designer <u>selects</u> from design components, component solutions, and knowledge available through catalogs, textbooks, and experience
  - The designer then <u>chooses</u> the elements from this collection that meet the requirements defined by requirements engineering and analysis modeling
  - Convergence occurs as <u>alternatives</u> are <u>considered</u> and <u>rejected</u> until one particular configuration of components is chosen
- Software design is an <u>iterative process</u> through which requirements are translated into a blueprint for constructing the software
  - Design begins at a <u>high level</u> of abstraction that can be directly traced back to the <u>data, functional, and behavioral</u> requirements
  - As design iteration occurs, subsequent refinement leads to design representations at much <u>lower levels</u> of abstraction

From Analysis Model to Design Model

- Each element of the analysis model provides information that is necessary to create the four design models
  - The data/class design transforms analysis classes into design classes along with the data structures required to implement the software
  - The architectural design defines the relationship between major structural elements of the software; architectural styles and design patterns help achieve the requirements defined for the system
  - The interface design describes how the software communicates with systems that interoperate with it and with humans that use it
  - The component-level design transforms structural elements of the software architecture into a procedural description of software components



Task Set for Software Design

1) Examine the information domain model and design appropriate data structures for data objects and their attributes
2) Using the analysis model, select an architectural style (and design patterns) that are appropriate for the software
3) Partition the analysis model into design subsystems and allocate these subsystems within the architecture
   a) Design the subsystem interfaces
   b) Allocate analysis classes or functions to each subsystem
4) Create a set of design classes or components
   a) Translate each analysis class description into a design class
   b) Check each design class against design criteria; consider inheritance issues
   c) Define methods associated with each design class
   d) Evaluate and select design patterns for a design class or subsystem
5) Design any interface required with external systems or devices
6) Design the user interface
7) Conduct component-level design
   a) Specify all algorithms at a relatively low level of abstraction

   b) Refine the interface of each component
   c) Define component-level data structures
   d) Review each component and correct all errors uncovered
8) Develop a deployment model
   a) Show a physical layout of the system, revealing which components will be located where in the physical computing environment

Design Quality

Quality's Role

- The importance of design is quality
- Design is the place where quality is fostered
  - Provides representations of software that can be assessed for quality
  - Accurately translates a customer's requirements into a finished software product or system
  - Serves as the foundation for all software engineering activities that follow
- Without design, we risk building an unstable system that
  - Will fail when small changes are made
  - May be difficult to test
  - Cannot be assessed for quality later in the software process when time is short and most of the budget has been spent
- The quality of the design is assessed through a series of formal technical reviews or design walkthroughs

Goals of a Good Design

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

Design Quality Guidelines

1) A design should exhibit an architecture that
   a) Has been created using recognizable architectural styles or patterns
   b) Is composed of components that exhibit good design characteristics
   c) Can be implemented in an evolutionary fashion, thereby facilitating implementation and testing

2) A design should be modular; that is, the software should be logically partitioned into elements or subsystems

3) A design should contain distinct representations of data, architecture, interfaces, and components

4) A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns

5) A design should lead to components that exhibit independent functional characteristics

6) A design should lead to interfaces that reduce the complexity of connections between components and with the external environment

7) A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis

8) A design should be represented using a notation that effectively communicates its meaning

Design Concepts

- Abstraction
  - Procedural abstraction – a sequence of instructions that have a specific and limited function
  - Data abstraction – a named collection of data that describes a data object
- Architecture
  - The overall structure of the software and the ways in which the structure provides conceptual integrity for a system
  - Consists of components, connectors, and the relationship between them
- Patterns
  - A design structure that solves a particular design problem within a specific context
  - It provides a description that enables a designer to determine whether the pattern is applicable, whether the pattern can be reused, and whether the pattern can serve as a guide for developing similar patterns
- Modularity
  - Separately named and addressable components (i.e., modules) that are integrated to satisfy requirements (divide and conquer principle)
  - Makes software intellectually manageable so as to grasp the control paths, span of reference, number of variables, and overall complexity
- Information hiding
  - The designing of modules so that the algorithms and local data contained within them are inaccessible to other modules
  - This enforces access constraints to both procedural (i.e., implementation) detail and local data structures
- Functional independence
  - Modules that have a "single-minded" function and an aversion to excessive interaction with other modules

- High cohesion – a module performs only a single task
    - Low coupling – a module has the lowest amount of connection needed with other modules
- Stepwise refinement
    - Development of a program by successively refining levels of procedure detail
    - Complements abstraction, which enables a designer to specify procedure and data and yet suppress low-level details
- Refactoring
    - A reorganization technique that simplifies the design (or internal code structure) of a component without changing its function or external behavior
    - Removes redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failures
- **Design classes**
    - Refines the analysis classes by providing design detail that will enable the classes to be implemented
    - Creates a new set of design classes that implement a software infrastructure to support the business solution

Types of Design Classes

- **User interface classes** – define all abstractions necessary for human-computer interaction (usually via metaphors of real-world objects)
- **Business domain classes** – refined from analysis classes; identify attributes and services (methods) that are required to implement some element of the business domain
- **Process classes** – implement business abstractions required to fully manage the business domain classes
- **Persistent classes** – represent data stores (e.g., a database) that will persist beyond the execution of the software
- **System classes** – implement software management and control functions that enable the system to operate and communicate within its computing environment and the outside world

Characteristics of a Well-Formed Design Class

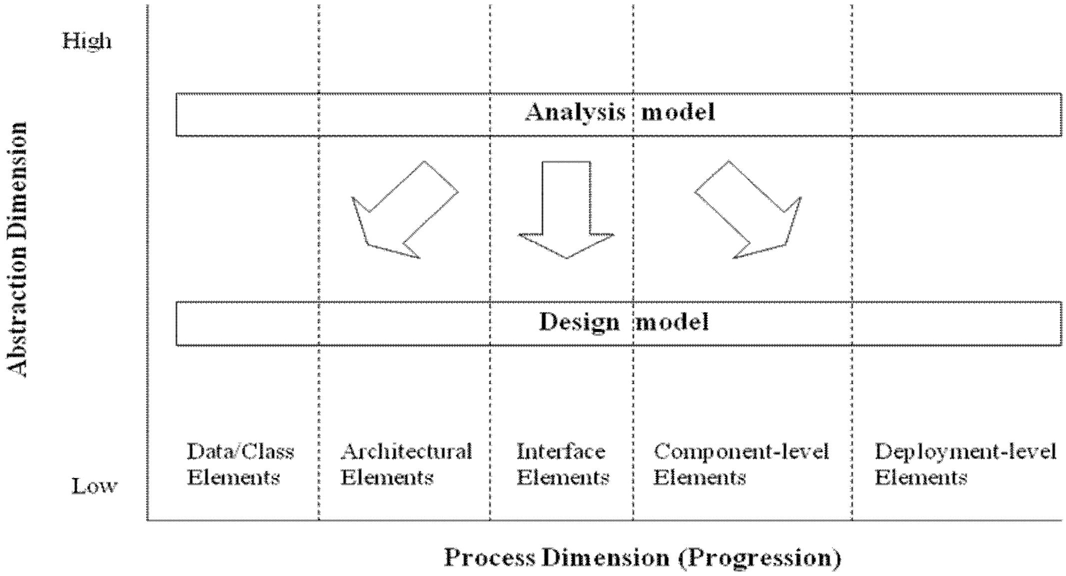
- Complete and sufficient
    - Contains the complete encapsulation of all attributes and methods that exist for the class
    - Contains only those methods that are sufficient to achieve the intent of the class
- Primitiveness
    - Each method of a class focuses on accomplishing one service for the class
- High cohesion
    - The class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities
- Low coupling

– Collaboration of the class with other classes is kept to an acceptable minimum

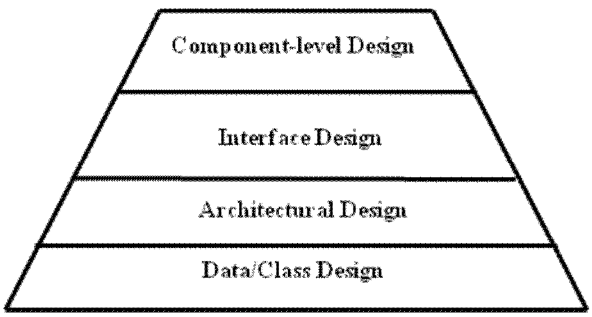
– Each class should have limited knowledge of other classes in other subsystems

- The Design Model -

Dimensions of the Design Model



- The design model can be viewed in two different dimensions
  - (Horizontally) The process dimension indicates the evolution of the parts of the design model as each design task is executed
  - (Vertically) The abstraction dimension represents the level of detail as each element of the analysis model is transformed into the design model and then iteratively refined
- Elements of the design model use many of the same UML diagrams used in the analysis model
  - The diagrams are refined and elaborated as part of the design
  - More implementation-specific detail is provided
  - Emphasis is placed on
    - Architectural structure and style
    - Interfaces between components and the outside world
    - Components that reside within the architecture
- Design model elements are not always developed in a sequential fashion
  - Preliminary architectural design sets the stage
  - It is followed by interface design and component-level design, which often occur in parallel
- The design model has the following layered elements
  - Data/class design
  - Architectural design
  - Interface design
  - Component-level design



- A fifth element that follows all of the others is deployment-level design

Design Elements

- Data/class design
  - Creates a model of data and objects that is represented at a high level of abstraction
- Architectural design
  - Depicts the overall layout of the software
- Interface design
  - Tells how information flows into and out of the system and how it is communicated among the components defined as part of the architecture
  - Includes the user interface, external interfaces, and internal interfaces
- Component-level design elements
  - Describes the internal detail of each software component by way of data structure definitions, algorithms, and interface specifications
- Deployment-level design elements
  - Indicates how software functionality and subsystems will be allocated within the physical computing environment that will support the software

Pattern-based Software Design

- Mature engineering disciplines make use of thousands of design patterns for such things as buildings, highways, electrical circuits, factories, weapon systems, vehicles, and computers
- Design patterns also serve a purpose in software engineering
- Architectural patterns (evolved from Creational, Structural & Behavioural design patterns)
  - Define the overall structure of software
  - Indicate the relationships among subsystems and software components
  - Define the rules for specifying relationships among software elements
- Design patterns
  - Address a specific element of the design such as an aggregation of components or solve some design problem, relationships among components, or the mechanisms for effecting inter-component communication
  - Consist of creational, structural, and behavioural patterns
- Coding patterns
  - Describe language-specific patterns that implement an algorithmic or data structure element of a component, a specific interface protocol, or a mechanism for communication among components