UNIT-III

Building the Analysis Model: Requirements Analysis Modeling approaches, Data modeling concepts, Object oriented analysis , Scenario based modeling, Flow oriented modeling, Class-based modeling, Creating a Behavioral Modeling.

Design Engineering: Design with in the context of SE, Design Process and Design quality, Design concepts, The Design Model, Pattern-based Software Design.

Building the Analysis Model

- Requirements analysis
- Flow-oriented modeling
- Scenario-based modeling
- Class-based modeling
- Behavioral 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 essential information versus implementation 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

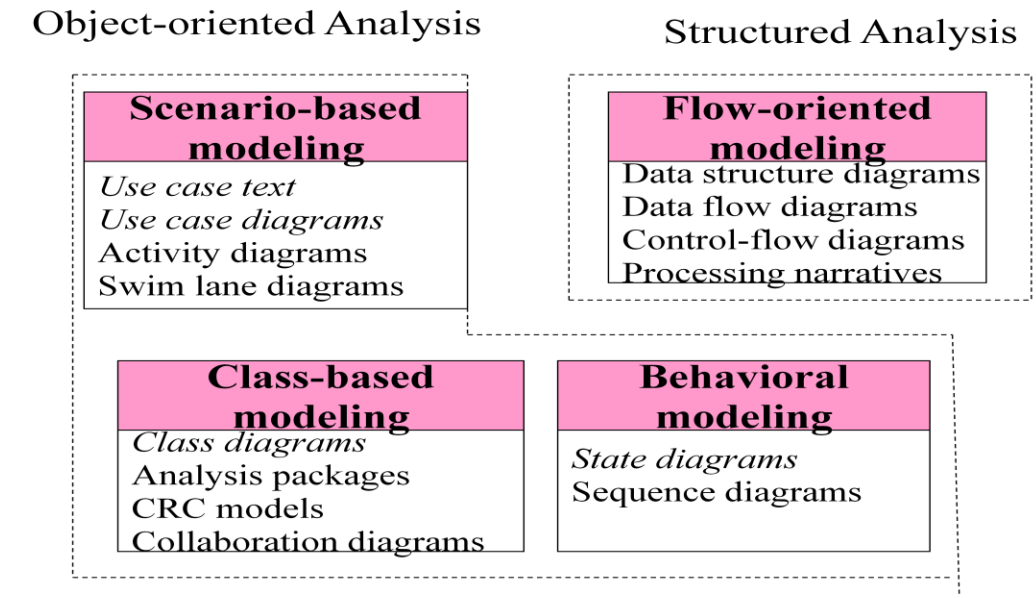
Domain Analysis

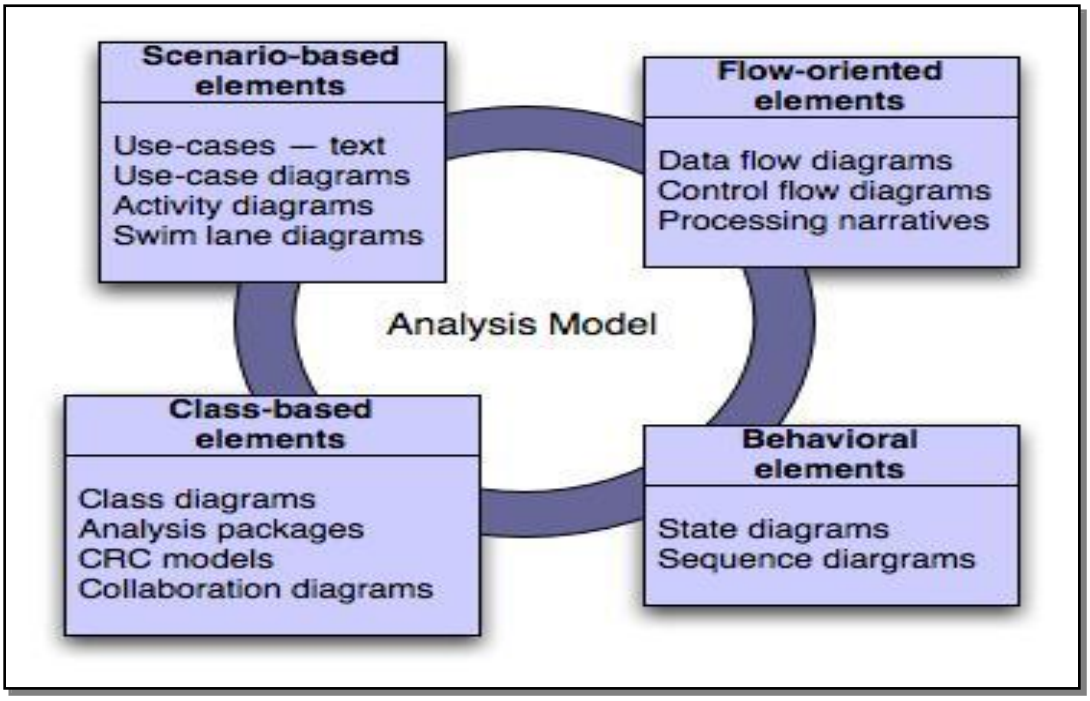
- Definition
  - The identification, analysis, and specification of common, reusable capabilities within a specific application domain
  - Do this in terms of common objects, classes, subassemblies, and frameworks
- Sources of domain knowledge
  - Technical literature
  - Existing applications
  - Customer 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

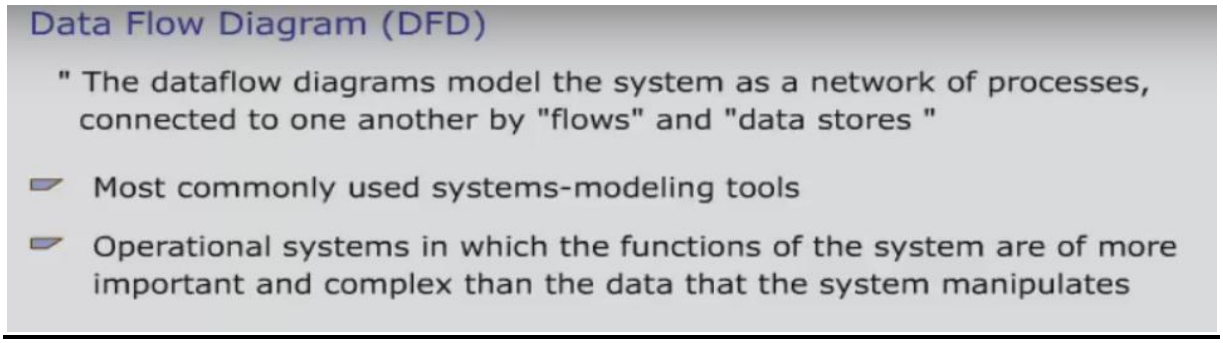


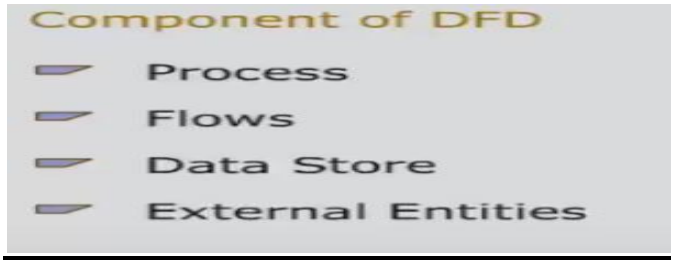


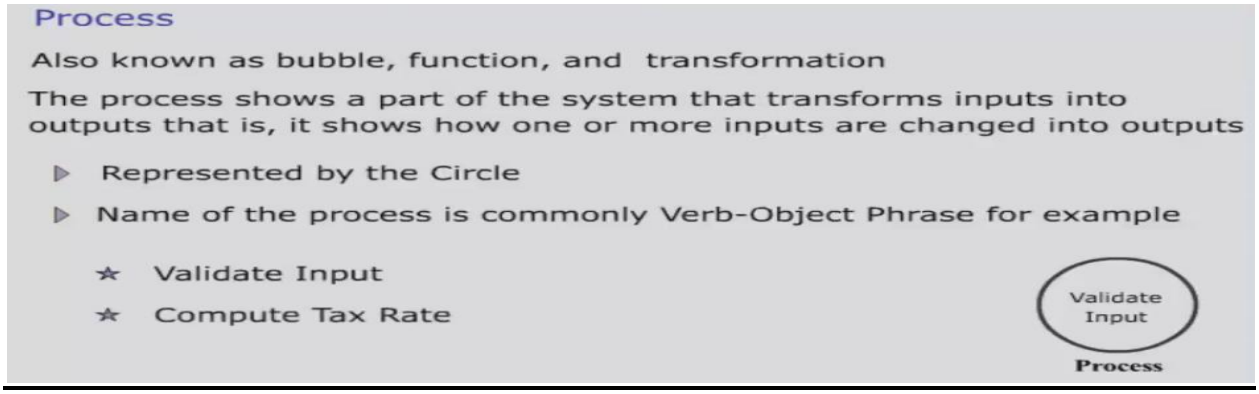
Flow-oriented Modeling

- Data 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

Data Flow Diagram





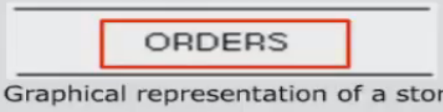


The Flow

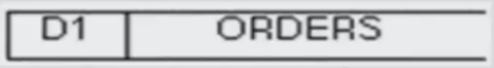
- Flow is represented by Arrow
- Flow represents data in motion
- Flow are labeled with nouns
- Flow show the direction of the data movement

The Store

- Store is used to model data at rest
- Store is represented by two parallel lines are
- Store are named
- Name of the store is plural of the packets associated with store



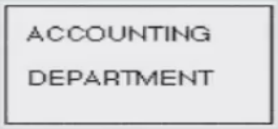
Graphical representation of a store



An alternative notation for a store

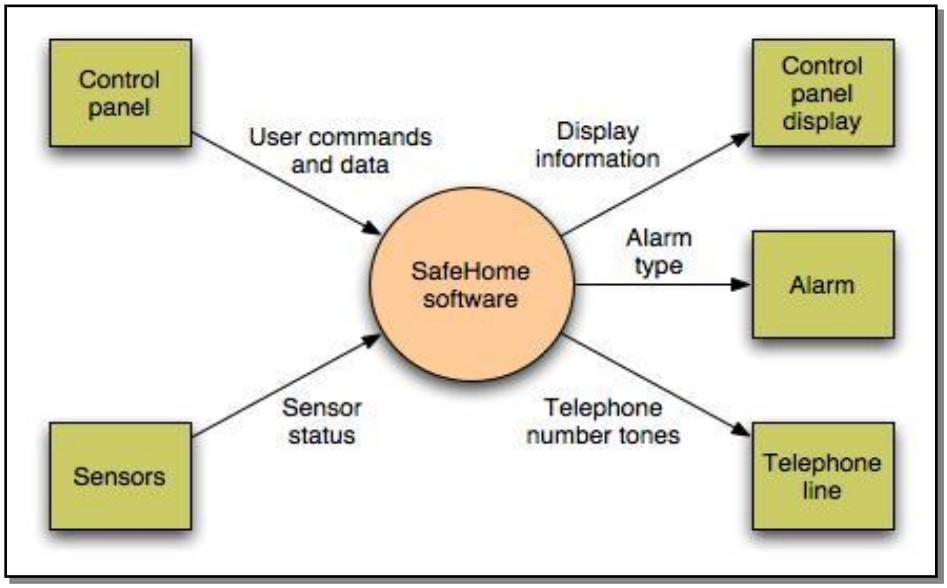
External Entity

- Represented by rectangular box
- External entities model entities which communicate with the system



Graphical representation of External Entity

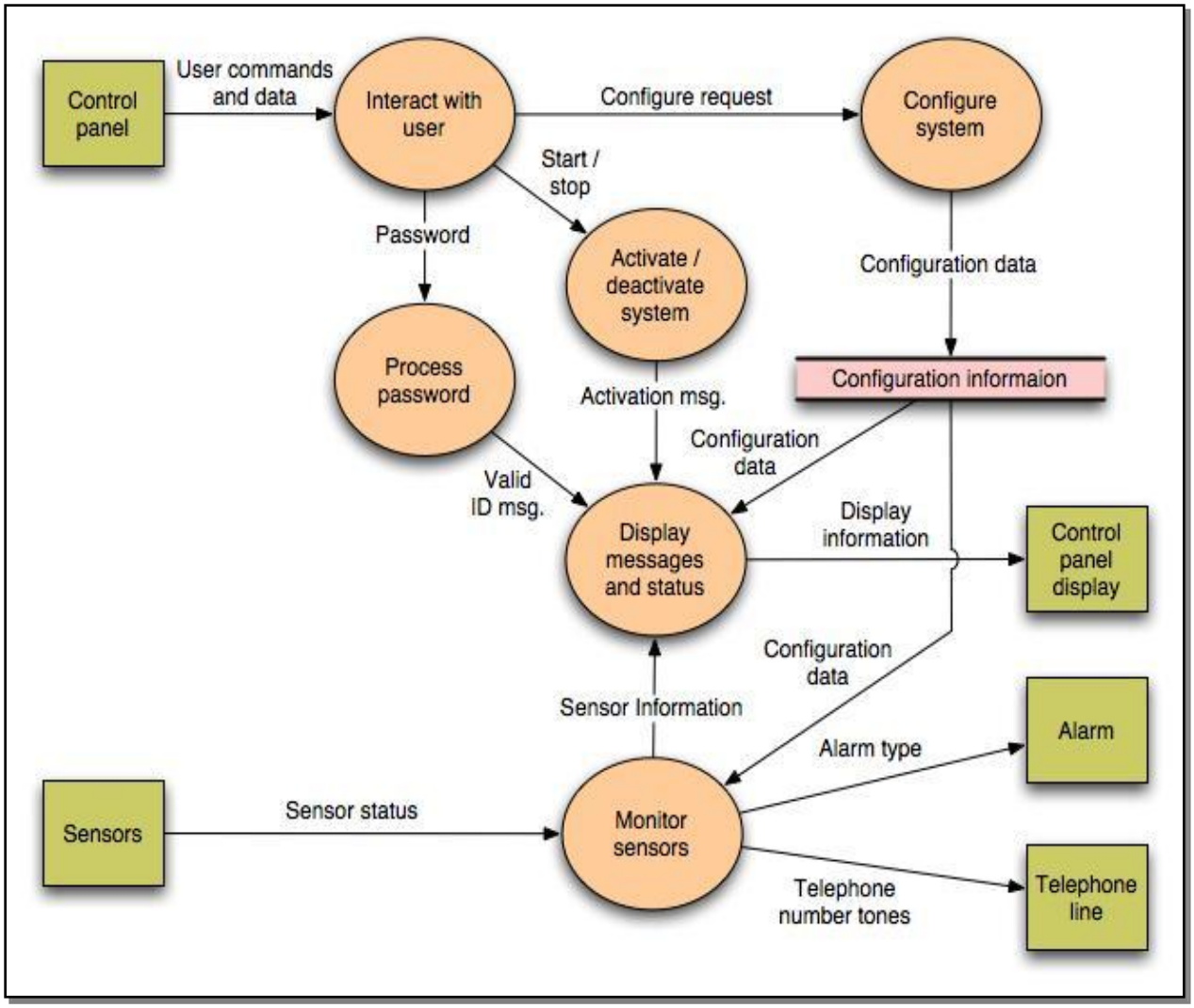
Context-level DFD for *SafeHome* security function

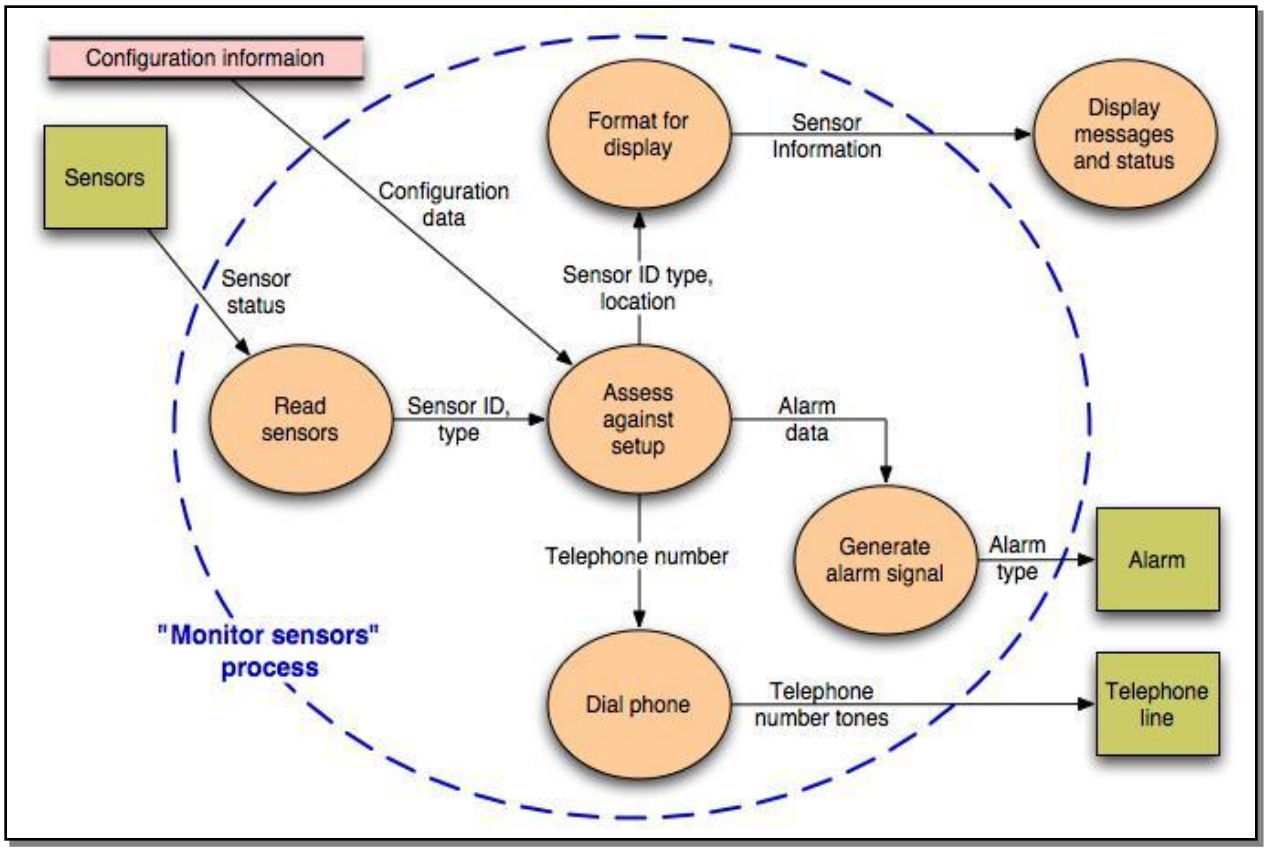


Grammatical Parse

- The ***SafeHome* security function** enables the **homeowner** to configure the **security system** when it is installed, monitors all **sensors** connected to the security system, and interacts with the homeowner through the **Internet**, a **PC**, or a **control panel**.
- During **installation**, the *SafeHome* PC is used to program and configure the system. Each sensor is assigned a **number** and **type**, a **master password** is programmed for arming and disarming the system, and **telephone number(s)** are input for dialing when a **sensor event** occurs.
- When a sensor event is recognized, the software invokes an audible **alarm** attached to the system. After a **delay time** that is specified by the homeowner during system configuration activities, the software *dials* a telephone number of a **monitoring service**, provides information about the **location**, reporting the nature of the event that has been detected. The telephone number will be redialed every 20 seconds until a **telephone connection** is obtained.
- The homeowner receives **security information** via a control panel, the PC, or a browser, collectively called an **interface**. The interface displays prompting **messages** and system **status information** on the control panel, the PC, or the browser window. Homeowner interaction takes the following form…

Level 2 DFD that refines the monitor sensors process





Control Flow Diagram

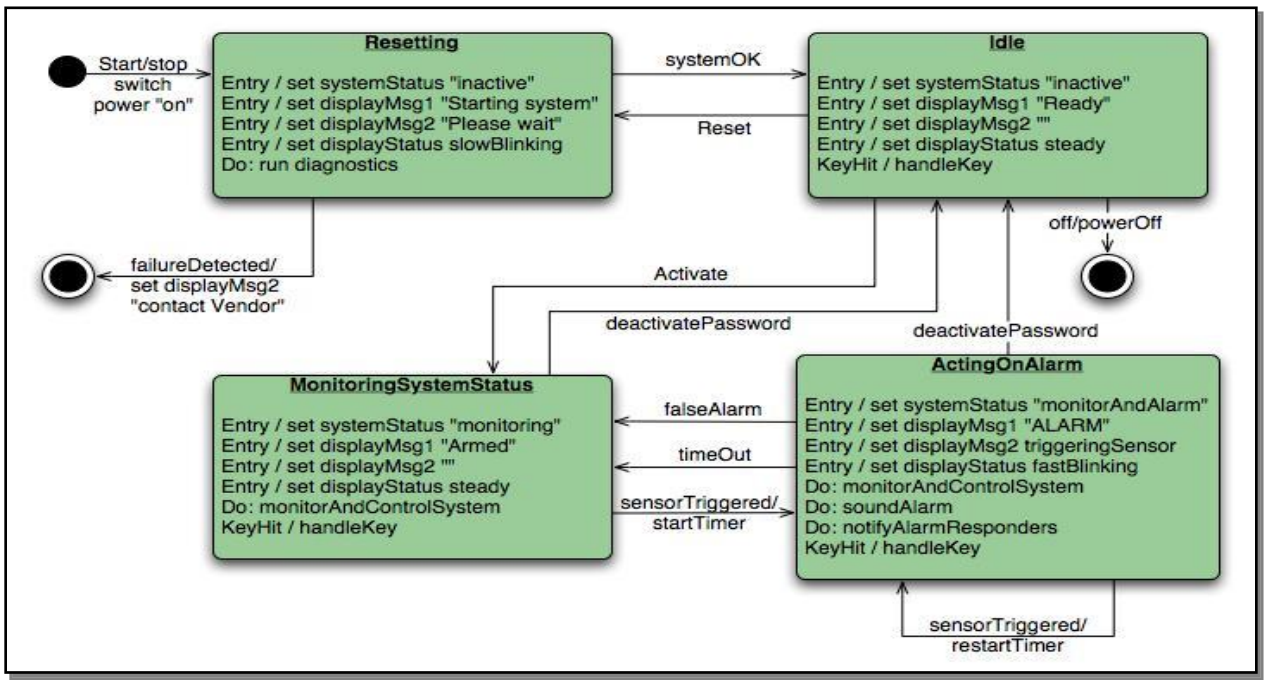
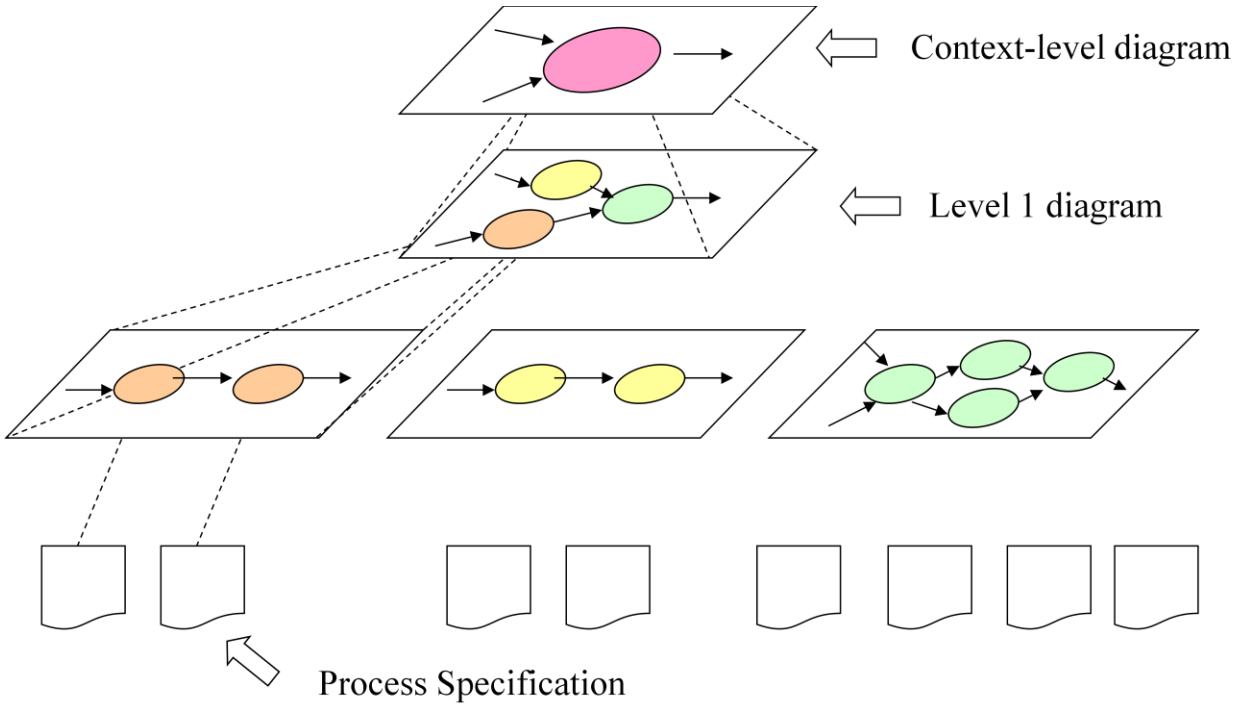


Diagram Layering and Process Refinement



Scenario-based Modeling

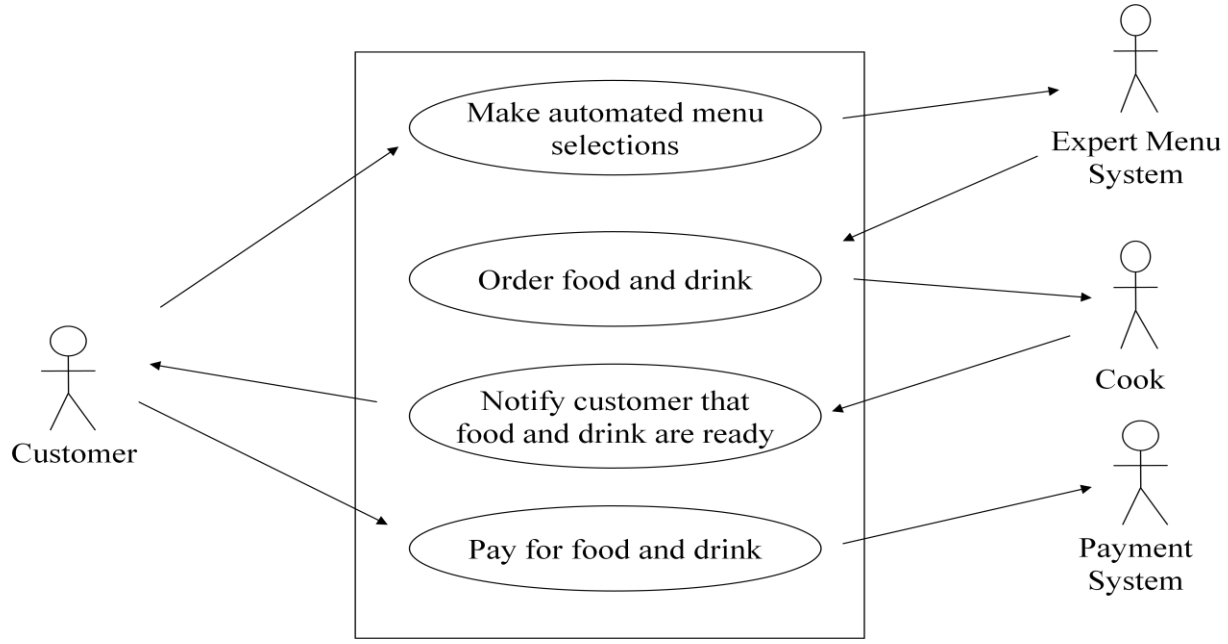
- *Use case text*
- *Use case diagrams*
- Activity diagrams
- Swim lane diagrams

Writing Use Cases

Use-case title: Actor: Description: I …

Use Case Diagram

Example Use Case Diagram:



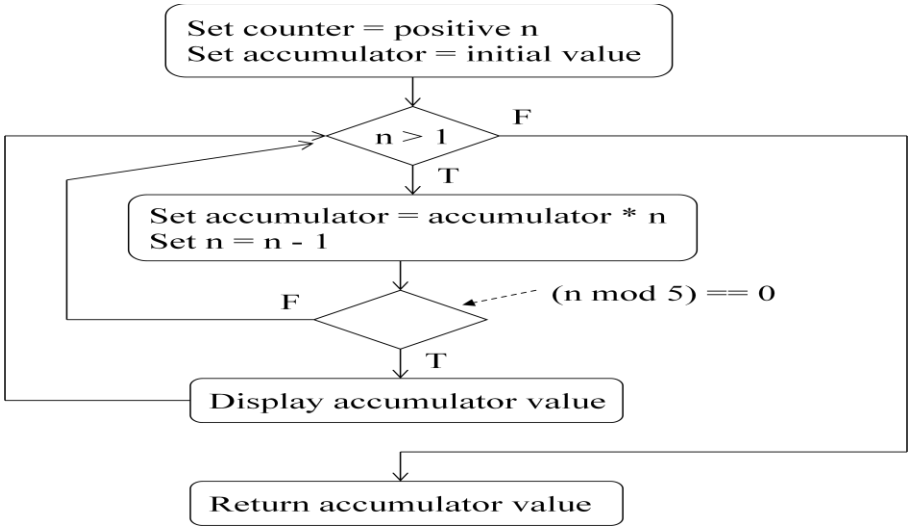
Alternative Actions

- Can the actor take some other action at this point?
- Is it possible that the actor will encounter some error condition at this point?
- Is it possible that the actor will encounter behavior invoked by some event outside the actor's control?

Activity Diagrams

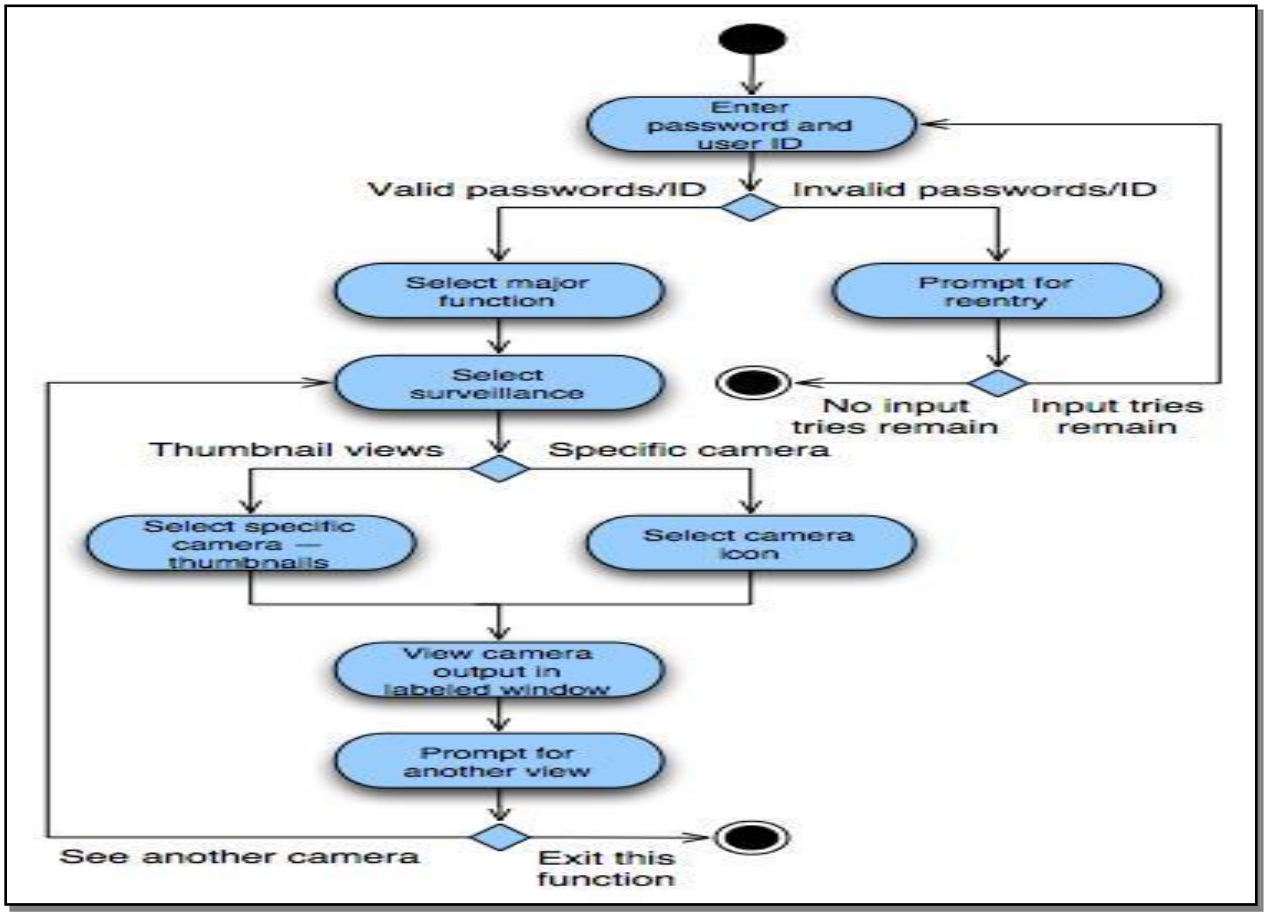
- The use case by providing a graphical representation of the flow of interaction within a specific scenario
- Uses flowchart-like symbols
  - **Rounded rectangle** - represent a specific system function/action
  - **Arrow** - represents the flow of control from one function/action to another
  - **Diamond** - represents a branching decision
  - **Solid bar –** represents the fork and join of parallel activities

Example Activity Diagram

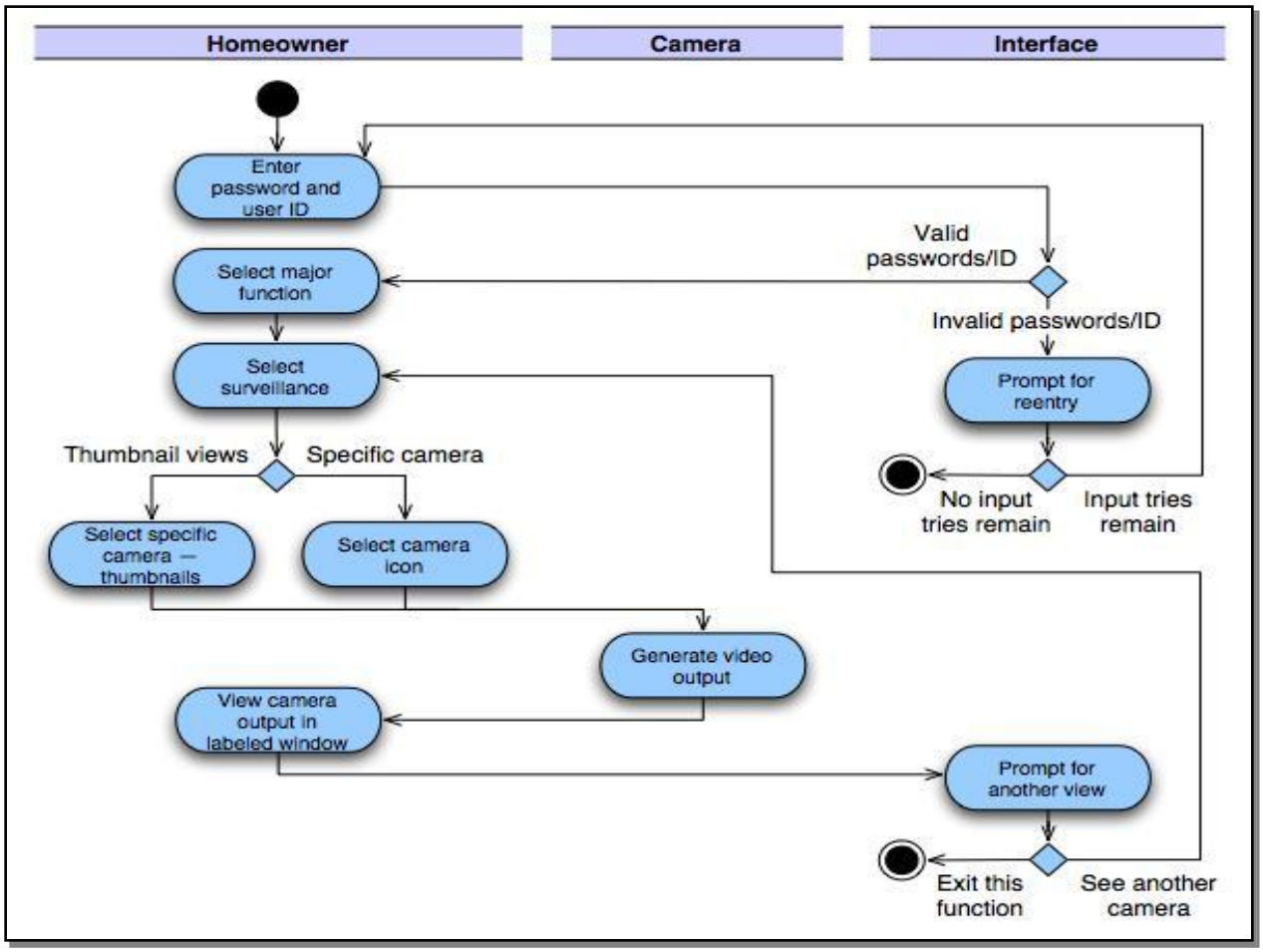


Activity diagram

For Access camera surveillance—display camera views function



Swimlane 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)

Potential class	Classification	Accept / Reject
homeowner	role; external entity	reject: 1, 2 fail
sensor	external entity	accept
control panel	external entity	accept
installation	occurrence	reject
(security) system	thing	accept
number, type	not objects, attributes	reject: 3 fails
master password	thing	reject: 3 fails
telephone number	thing	reject: 3 fails
sensor event	occurrence	accept
audible alarm	external entity	accept: 1 fails
monitoring service	organizational unit; ee	reject: 1, 2 fail

- 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

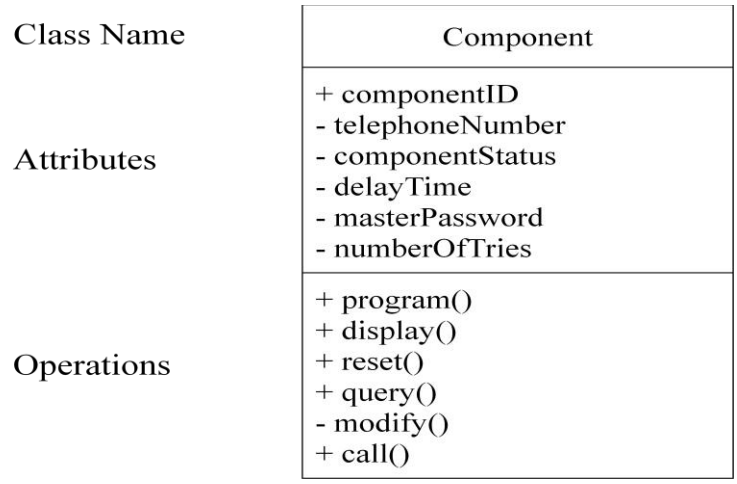
- Attributes of a class are those nouns from the grammatical parse that reasonably belong to a class
- Attributes hold the values that describe the current properties or state of a class
- An attribute may also appear initially as a potential class that is later rejected because of the class selection 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

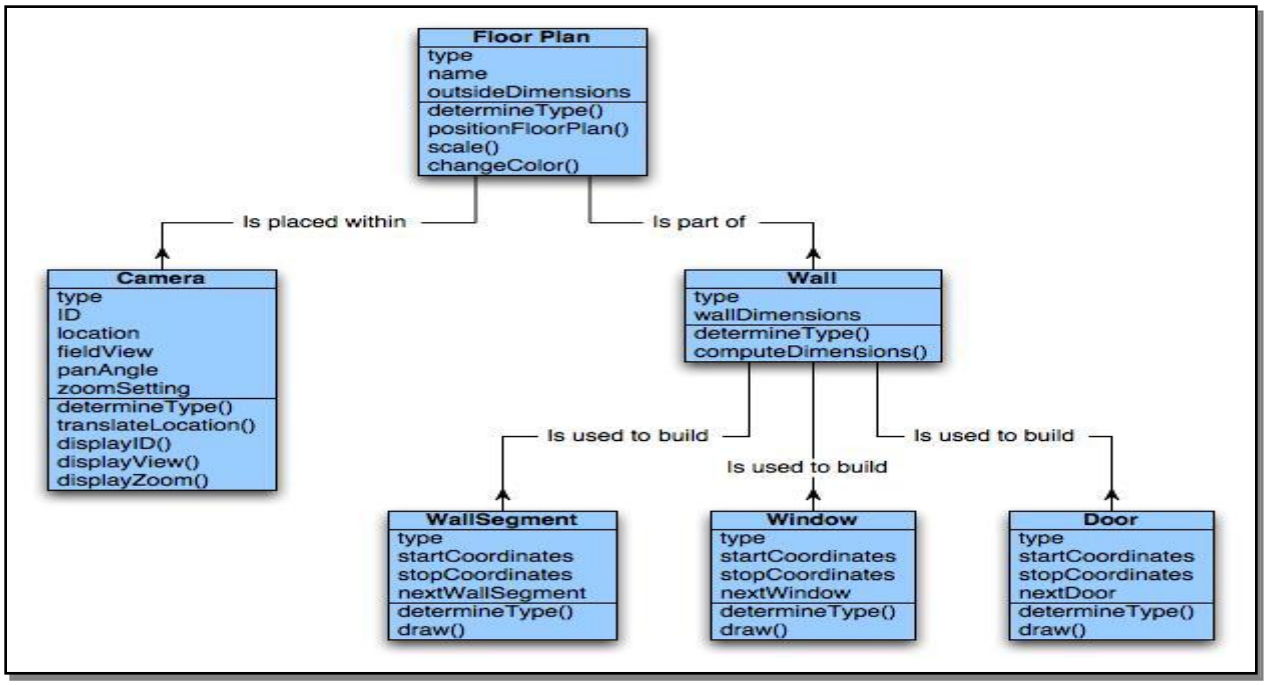
Example Class Box



Association, Generalization and Dependency

- **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 part-of relationship (i.e., aggregation), 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



CRC Modeling: A CRC model index card for FloorPlan class

Class: FloorPlan	
Description	
Responsibility	**Collaborator**
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Incorporates walls, doors, windows	Wall
Shows position of video cameras	Camera

Class Responsibilities

- Distribute system intelligence across classes.
- State each responsibility as generally as possible.
- Put information and the behavior related to it in the same class.
- Localize information about one thing rather than distributing it across multiple classes.
- Share responsibilities among related classes, when appropriate.

Class Collaborations

- Relationships between classes:
  - is-part-of — used when classes are part of an aggregate class.
  - has-knowledge-of — used when one class must acquire information from another class.
  - depends-on — used in all other cases.

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

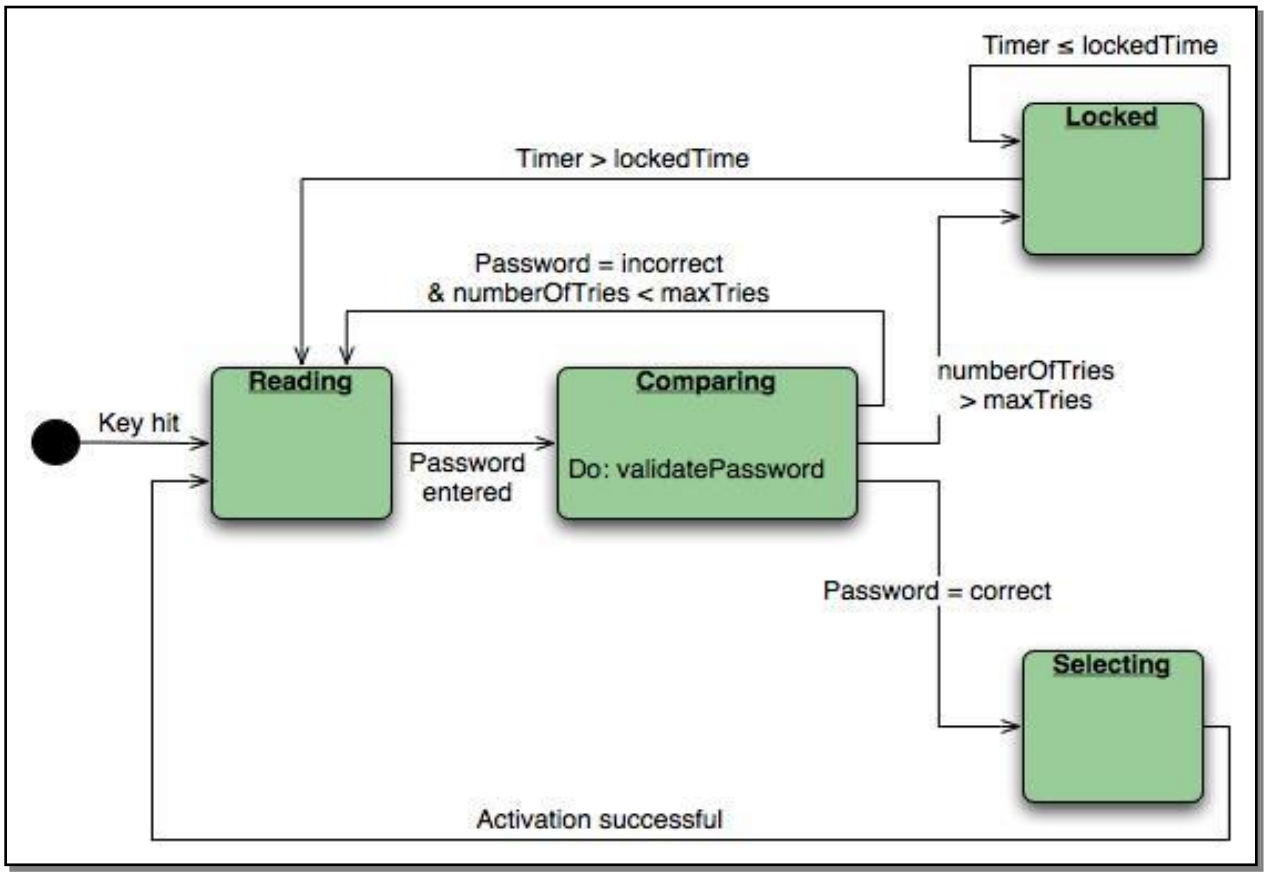
- An event occurs whenever an actor and the system exchange information
- An event is NOT the information that is exchanged, but rather the fact that information has been exchanged
- Some events have an explicit impact on the flow of control, while others do not
  - An example is the reading of a data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

Building a State Diagram

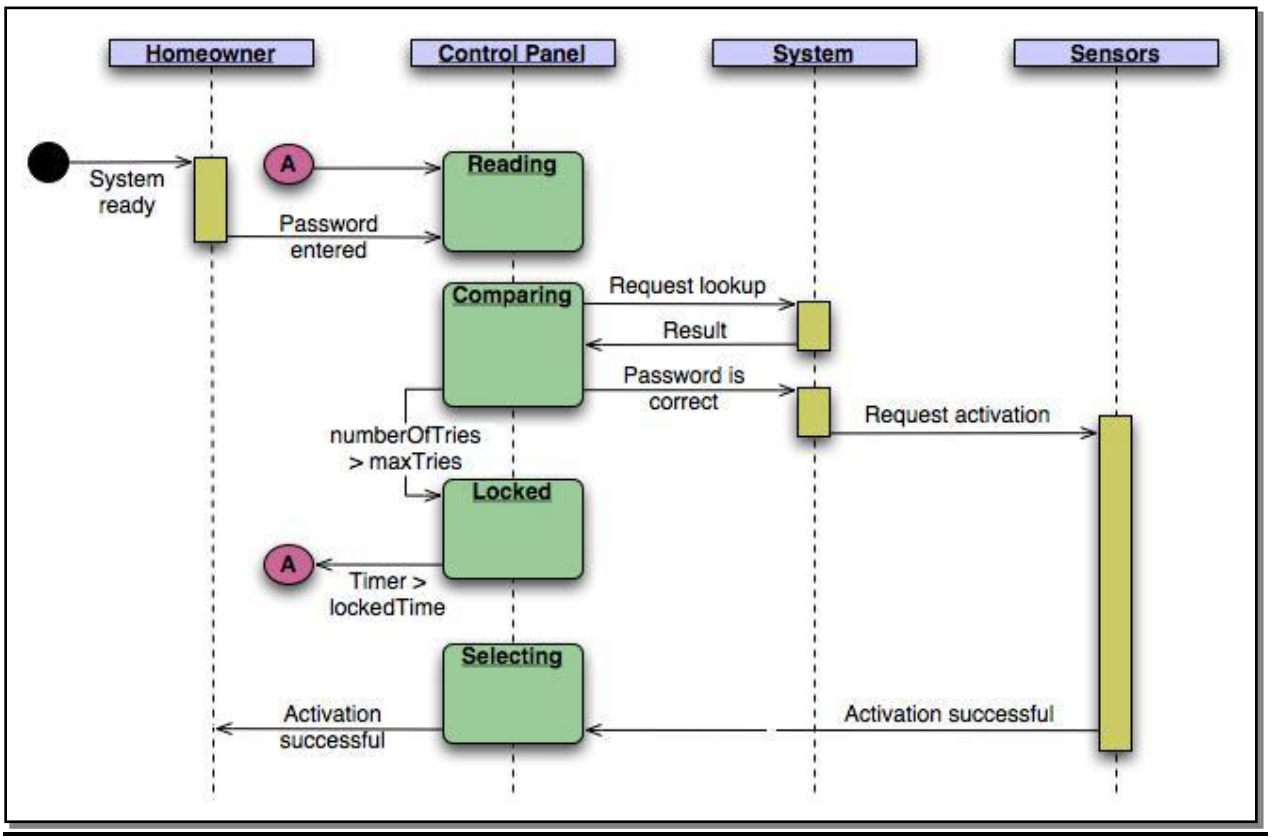
- A state is represented by a rounded rectangle
- A transition (i.e., event) is represented by a labeled arrow leading from one state to another
  - Syntax: trigger-signature [guard]/activity

- The active state 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 passive state 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



Sequence Diagram



Design Engineering

The design model consists of the data design, architectural design, interface design, and component-level design.

The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight.

To accomplish this, a designer must practice diversification and then

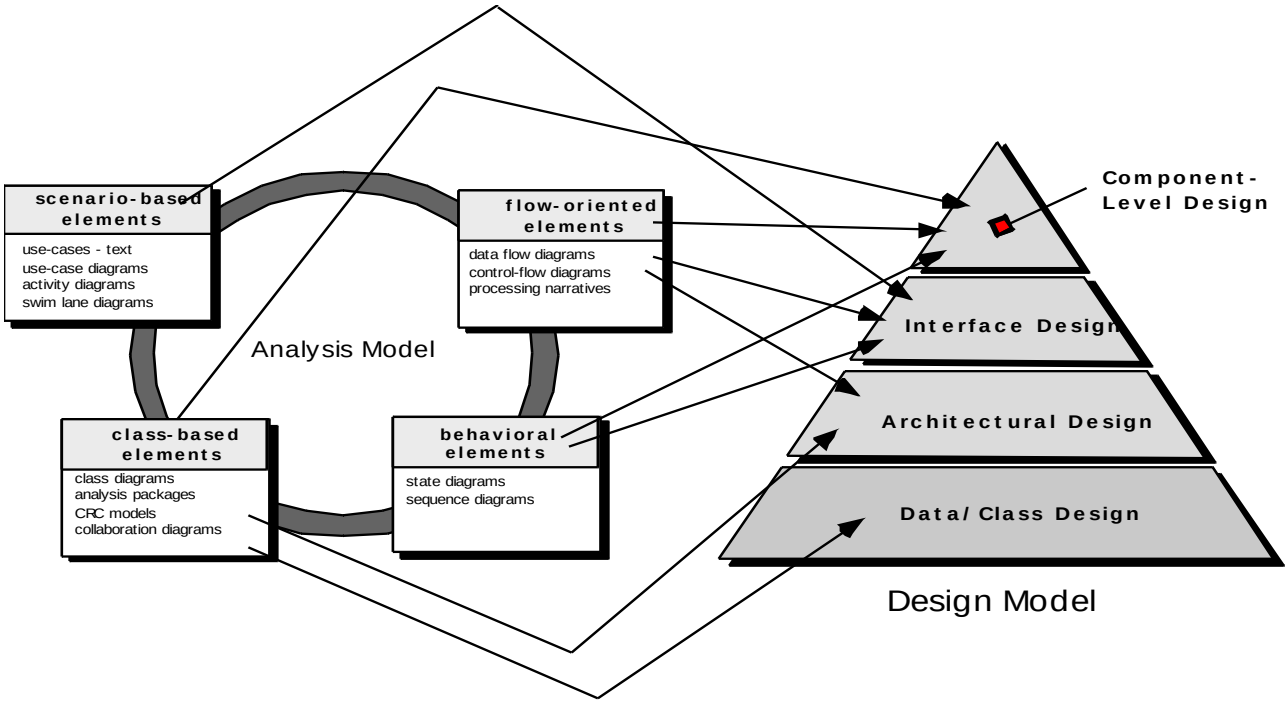
Design within the Context of Software Engineering

Software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

The flow of information during software design is illustrated in Figure below. The analysis model, manifested by scenario-based, class-based, flow-oriented and behavioral elements, feed the design task.

The *architectural design* defines the relationship between more structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which the architectural design can be implemented.

The *architectural design* can be derived from the System Specs, the analysis model, and interaction of subsystems defined within the analysis model.



The *interface design* describes how the software communicates with systems that interpolate with it, and with humans who use it. An interface implies a flow of information (data, and or control) and a specific type of behavior.

The *component-level design* transforms structural elements of the software architecture into a procedural description of software components.

The importance of software design can be stated with a single word – *quality*. Design is the place where quality is fostered in software engineering. Design provides us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer's requirements into a finished software product or system.

Design Process and Design Quality

Software design is an iterative process through which requirements are translated into a "**blueprint**" for constructing the software.

Initially, the **blueprint** depicts a holistic view of software, i.e. the design is represented at a high-level of abstraction.

Throughout the design process, the quality of the evolving design is assessed with a series of formal technique reviews or design walkthroughs.

Three characteristics serve as a guide for the evaluation of a good design:

- The 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.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Quality Guidelines

In order to evaluate the quality of a design representation, we must establish technical criteria for good design.

1. 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

      a. For smaller systems, design can sometimes be developed linearly.
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.

Quality Attributes

Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS. The FURPS quality attributes represent a target for all software design:

- ✓ *Functionality*: is assessed by evaluating the features set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- ✓ *Usability*: is assessed by considering human factors, overall aesthetics, consistency, and documentation.
- ✓ *Reliability*: is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure, the ability to recover from failure, and the predictability of the program.
- ✓ *Performance*: is measured by processing speed, response time, resource consumption, throughput, and efficiency.
- ✓ *Supportability*: combines the ability to extend the program extensibility, adaptability, serviceability ➔ maintainability. In addition, testability, compatibility, configurability, etc.

Design Concepts

The significant design concepts are abstraction, refinement, modularity, architecture, patterns, refactoring, functional independence, information hiding, and OO design concepts.

Abstraction

At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided.

As we move through different levels of abstraction, we work to create procedural and data abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. An example of a procedural abstraction would be the word *open* for a door.

A *data abstraction* is a named collection of data that describes a data object. In the context of the procedural abstraction *open,* we can define a data abstraction called

door. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g. **door type, swing direction, weight)**.

Architecture

Software architecture alludes to the "overall structure of the software and the ways in which the structure provides conceptual integrity for a system."

In its simplest from, architecture is the structure of organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.

Te goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which detailed design activities are constructed.

A set of architectural patterns enable a software engineer to reuse design-level concepts.

The architectural design can be represented using one or more of a number of different models.

Structural models represent architecture as an organized collection of program components.

Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.

Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

Process models focus on the design of business or technical process that the system must accommodate.

Functional models can be used to represent the functional hierarchy of a system.

Architectural design will be discussed in Chapter 10.

Patterns

A design pattern "conveys the essence of a proven design solution to a recurring problem within a certain context amidst computing concerns."

A design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine:

1. whether the pattern is applicable to the current work,
2. whether the pattern can be reused, and
3. whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Modularity

Software architecture and design patterns embody *modularity*; that is, software is divided into separately named and addressable components, sometimes called modules that are integrated to satisfy problem requirements.

Monolithic software (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.

It is the compartmentalization of data and function. It is easier to solve a complex problem when you break it into manageable pieces. “Divide-and-conquer”

Don’t over-modularize. The simplicity of each small module will be overshadowed by the complexity of integration “Cost”.

Information Hiding

It is about controlled interfaces. Modules should be specified and design so that information (algorithm and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining by a set of independent modules that communicate with one another only that information necessary to achieve software function.

The use of Information Hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to other location within the software.

Functional Independence

The concept of *functional Independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding.

Design software so that each module addresses a specific sub-function of requirements and has a simple interface when viewed from other parts of the program structure.

Functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: **cohesion** and **coupling.**

Cohesion is an indication of the relative functional strength of a module.

Coupling is an indication of the relative interdependence among modules.

A cohesive module should do just one thing.

Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world “lowest possible”.

Cohesion is the “single-mindedness’ of a component

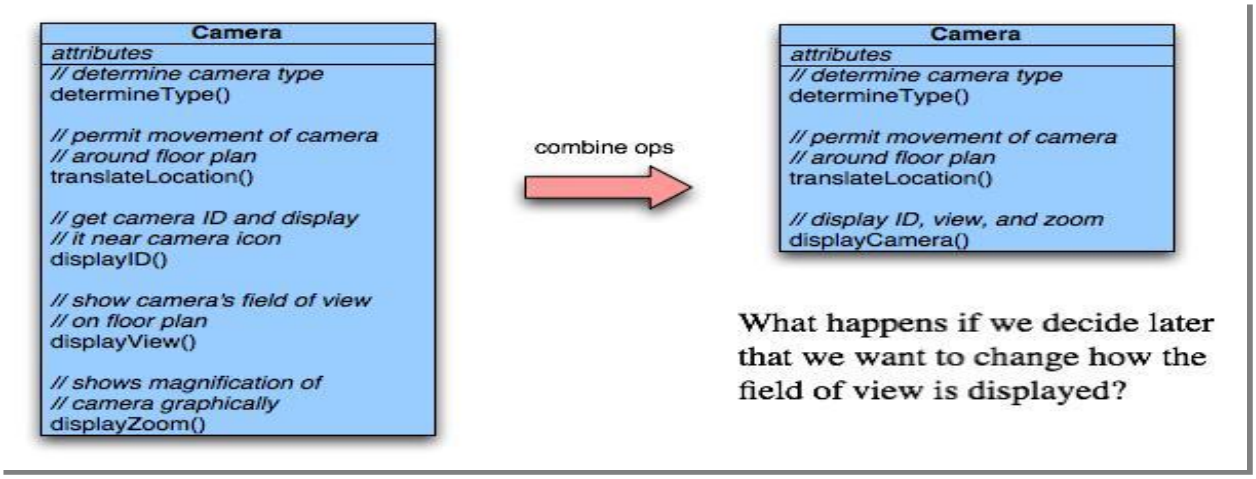
It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself. The objective is to keep cohesion as high as possible The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)

Kinds of cohesion

The different classes of cohesion that a module may possess are:

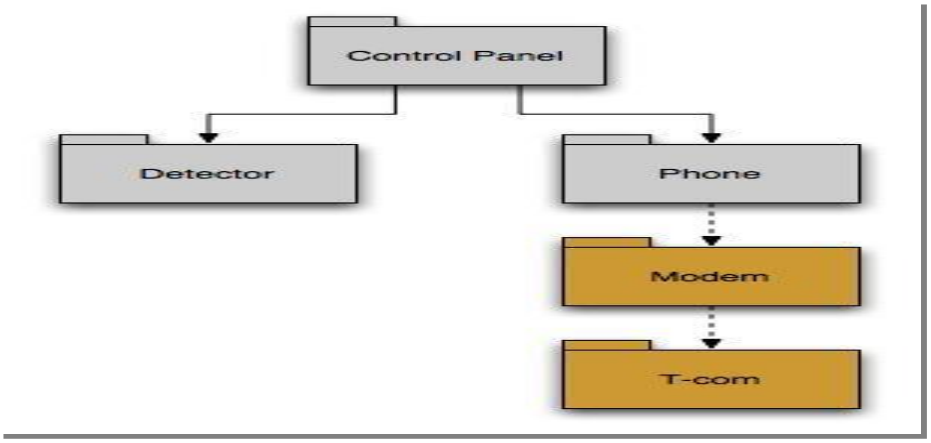
Functional

Typically applies to operations. Occurs when a module performs one and only one computation and then returns a result.



Layer

A higher layer component accesses the services of a lower layer component. Applies to packages, components, and classes. Occurs when a higher layer can access a lower layer, but lower layers do not access higher layers.



Communicational

All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.

Example: A StudentRecord class that adds, removes, updates, and accesses various fields of a student record for client components. ,

Sequential

Components or operations are grouped in a manner that allows the first to provide input to the next and so on in order to implement a sequence of operations. A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next.

For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.

Procedural

Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no data passed between them. A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective.

E.g. the algorithm for decoding a message.

Temporal

Operations are grouped to perform a specific behavior or establish a certain state such as program start-up or when an error is detected.

E.g. The set of functions responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion.

Utility

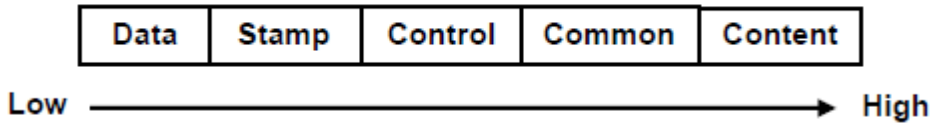
Components, classes, or operations are grouped within the same category because of similar general functions but are otherwise unrelated to each other

Coupling

As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases. As complexity rises, the difficulty of implementing, testing, and maintaining software also increases. Coupling is a qualitative measure of the degree to which operations and classes are connected to one another

The objective is to keep coupling as low as possible.

Five types of coupling can occur between any two modules



Data coupling: Two modules are data coupled, if they communicate through a parameter. An example is an elementary data item passed as a parameter between two modules,

E.g. an integer, a float, a character, etc.

This data item should be problem related and not used for the control purpose.

Stamp coupling: Two modules are stamp coupled, if they communicate using a composite data item.

E.g: A record in PASCAL or a structure in C.

Control coupling: Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another.

An example of control coupling is a flag set in one module and tested in another module.

Common coupling: Two modules are common coupled, if they share data through some global data items. Content coupling: Content coupling exists between two modules, if they share code.

E.g. a branch from one module into another module.

Refinement

It is the elaboration of detail for all abstractions. It is a top down strategy.

A program is developed by successfully refining levels of procedural detail.

A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

We begin with a statement of function or data that is defined at a high level of abstraction.

The statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the data.

Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction enables a designer to specify procedure and data and yet suppress low-level details.

Refinement helps the designer to reveal low-level details as design progresses.

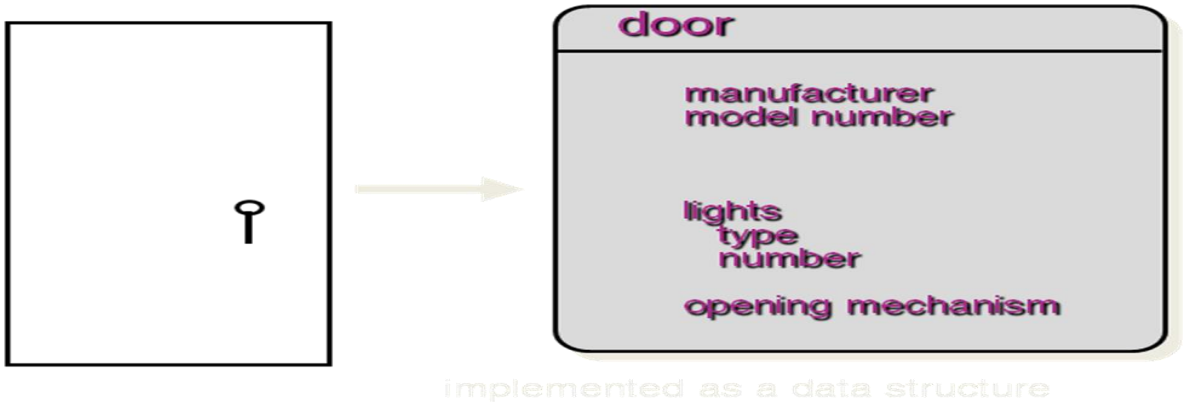
Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement “elaboration” occurs.

Refactoring

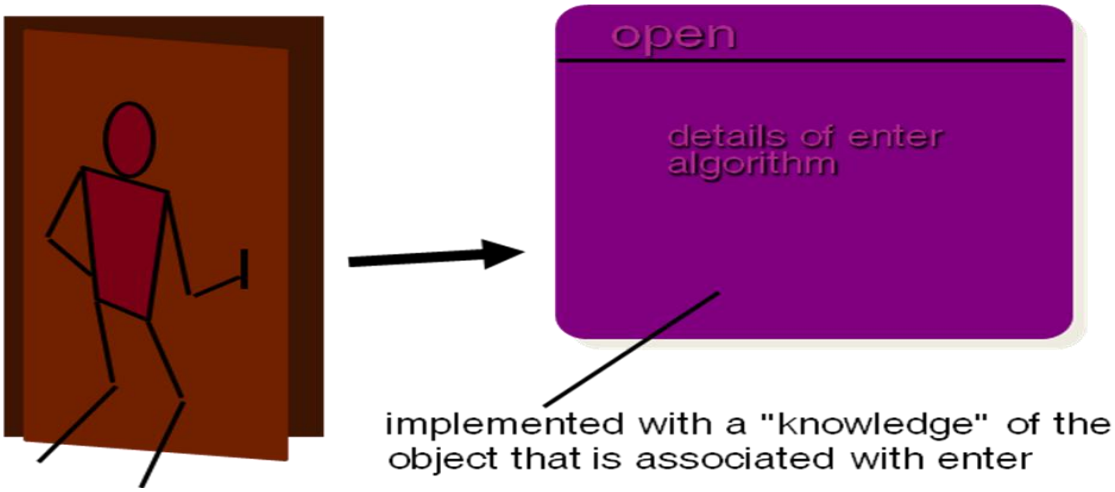
It is a reorganization technique that simplifies the design of a component without changing its function or behavior. When software is re-factored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary

algorithms, poorly constructed data structures, or any other design failures that can be corrected to yield a better design.

Data Abstraction



Procedural Abstraction



"The overall structure of the software and the ways in which that structure provides conceptual integrity for a system."

- Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods

- Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
- Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

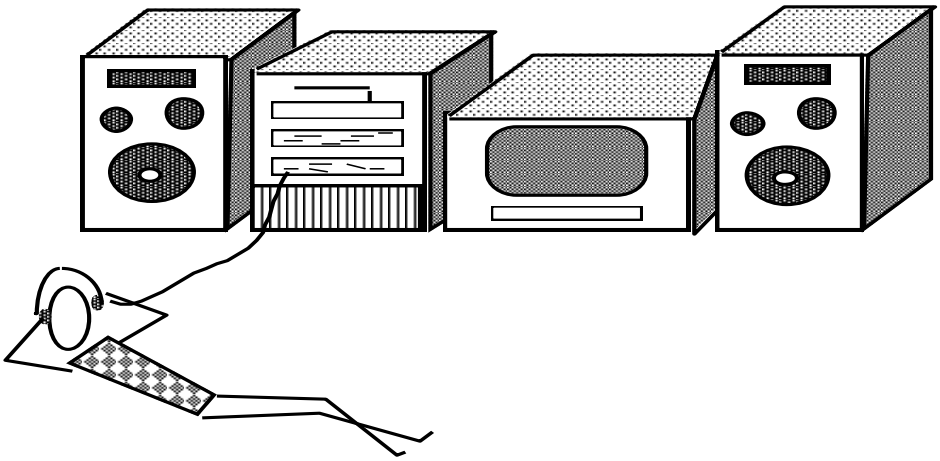
Patterns

Design Pattern Template

Pattern name—describes the essence of the pattern in a short but expressive name
Intent—describes the pattern and what it does
Also-known-as—lists any synonyms for the pattern
Motivation—provides an example of the problem
Applicability—notes specific design situations in which the pattern is applicable
Structure—describes the classes that are required to implement the pattern
Participants—describes the responsibilities of the classes that are required to implement the pattern
Collaborations—describes how the participants collaborate to carry out their responsibilities
Consequences—describes the "design forces" that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented
Related patterns—cross-references related design patterns

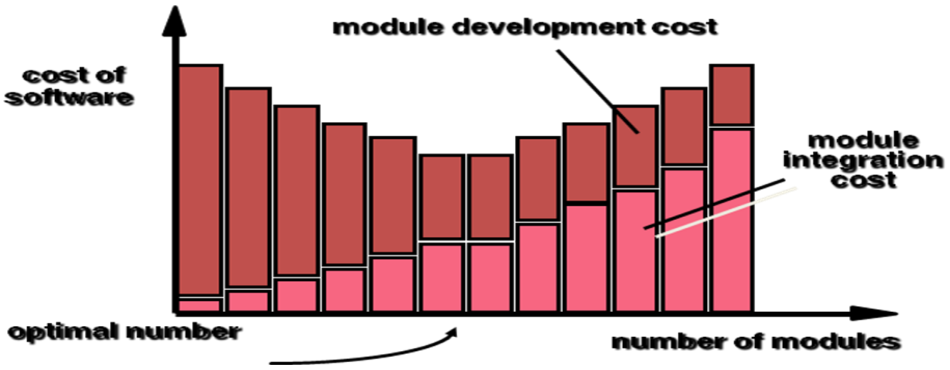
Modular Design

easier to build, easier to change, easier to fix …

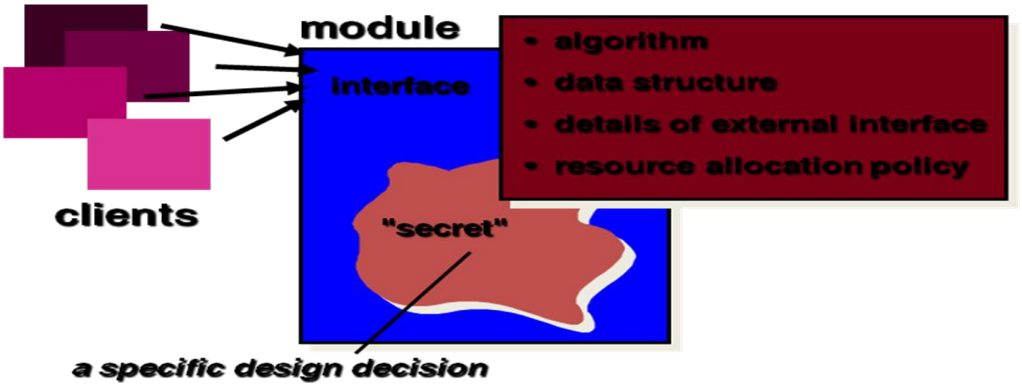


Modularity: Trade-offs

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



Information Hiding



Why Information Hiding?

- Reduces the likelihood of “side effects”
- Limits the global impact of local design decisions
- Emphasizes communication through controlled interfaces
- Discourages the use of global data
- Leads to encapsulation—an attribute of high quality design
- Results in higher quality software

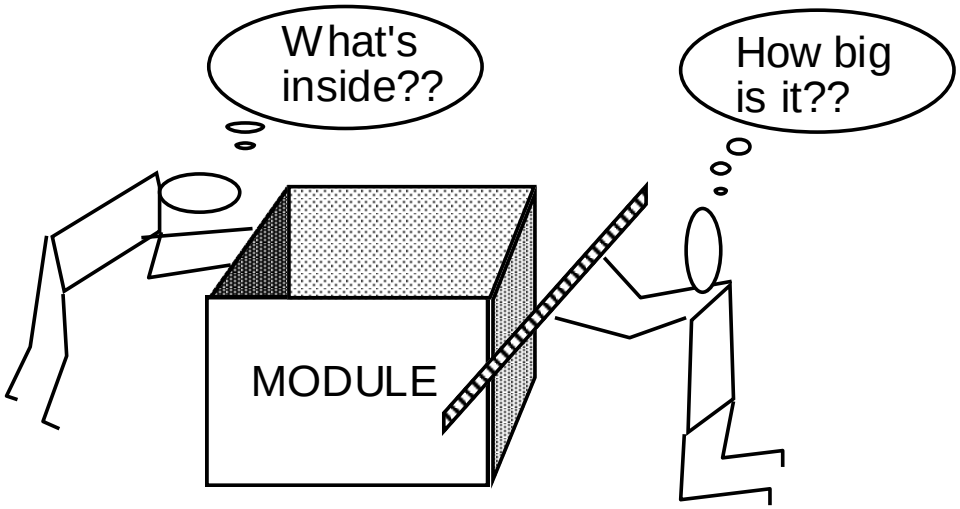
Stepwise Refinement

Functional Independence

COHESION - the degree to which a module performs one and only one function.

COUPLING - the degree to which a module is "connected" to other modules in the system.

Sizing Modules: Two Views



Refactoring

- Fowler [FOW99] defines refactoring in the following manner:
  - "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."
- When software is re-factored, the existing design is examined for
  - redundancy
  - unused design elements
  - inefficient or unnecessary algorithms
  - poorly constructed or inappropriate data structures,
  - or any other design failure that can be corrected to yield a better design.

OO Design Concepts

- Entity classes
  - Boundary classes
  - Controller classes
- Inheritance—all responsibilities of a super-class 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

As the design model evolves, the software team must define a set of *design classes* that refines the analysis classes and creates a new set of design classes.

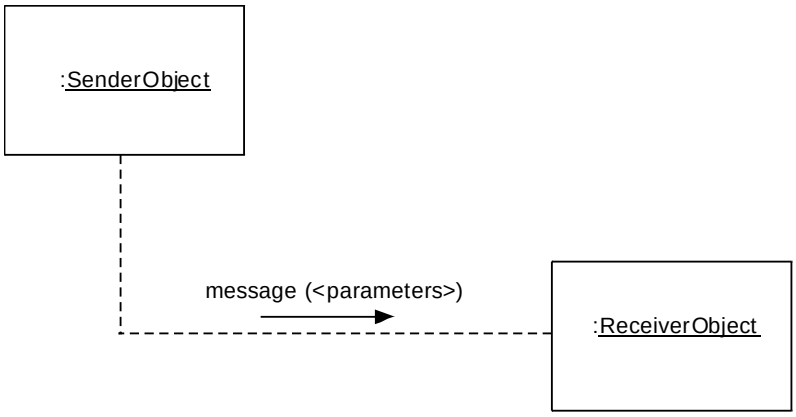
Five different classes' types are shown below:

1. *User Interface classes*: define all abstractions that are necessary for HCI.
2. *Business domain classes*: are often refinements of the analysis classes defined earlier. The classes identify the attributes and services that are required to implement some element of the business domain.
3. *Process classes:* implement lower-level business abstractions required to fully manage the business domain classes.
4. *Persistent classes:* represent data stores that will persist beyond the execution of the software.
5. *System classes*: implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

Inheritance (Example)

- Design options:
  - The class can be designed and built from scratch. That is, inheritance is not used.
  - The class hierarchy can be searched to determine if a class higher in the hierarchy (a super-class) contains most of the required attributes and operations. The new class inherits from the super-class and additions may then be added, as required.
  - The class hierarchy can be restructured so that the required attributes and operations can be inherited by the new class.
  - Characteristics of an existing class can be overridden and different versions of attributes or operations are implemented for the new class.

Messages



Polymorphism

Conventional approach …

case of graphtype:

if graphtype = linegraph then DrawLineGraph (data);

if graphtype = piechart then DrawPieChart (data);

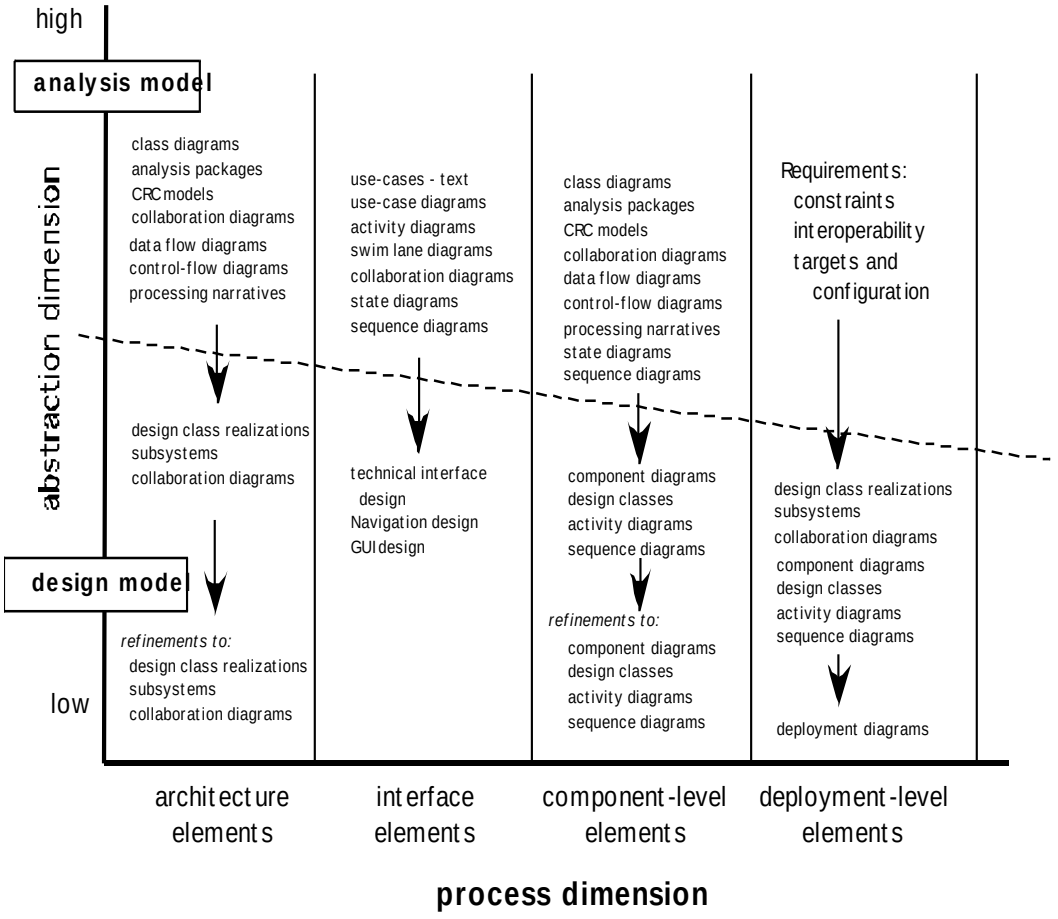
if graphtype = histogram then DrawHisto (data);

if graphtype = kiviat then DrawKiviat (data);

end case;

All of the graphs become subclasses of a general class called graph. Using a concept called overloading , each subclass defines an operation called *draw*. An object can send a *draw* message to any one of the objects instantiated from any one of the subclasses. The object receiving the message will invoke its own *draw* operation to create the appropriate graph.

The Design Model



- Data elements
- Data model --> data structures
- Data model --> database architecture
- Architectural elements "similar to the floor plan of a house"
  - "**You can use an eraser on the drafting table or a sledge hammer on the construction site**." Frank Lloyd Wright
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Patterns and "styles"
- Interface elements "The way in which utilities connections come into the house and are distributed among the rooms"
  - the user interface (UI)

- external interfaces to other systems, devices, networks or other producers or consumers of information
- internal interfaces between various design components.

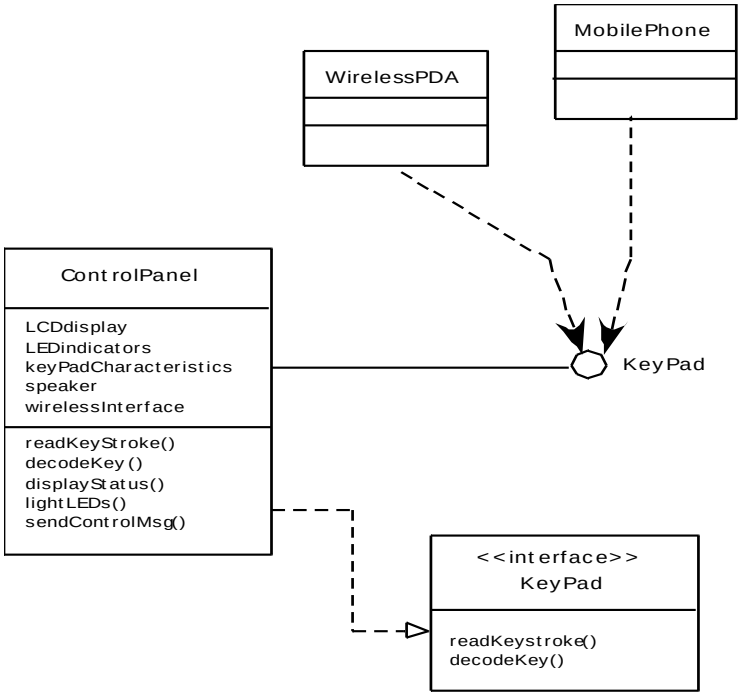


Figure 9.6 UML interface representation for **ControlPanel**

- Component elements

  It is equivalent to a set of detailed drawings and specs for each room in a house. The component-level design for software fully describes the internal detail of each software component.

- Deployment elements

  Indicates how software functionally and subsystem terms will be allocated within the physical computing environment that will support the software.

Pattern-Based Software Design

Describing a Design Pattern

- The best designers in any field have an uncanny ability to see patterns that characterize a problem and corresponding patterns that can be combined to create a solution
- A description of a design pattern may also consider a set of design forces.
  - *Design forces* describe non-functional requirements (e.g., ease of maintainability, portability) associated the software for which the pattern is to be applied.
  - Forces define the constraints that may restrict the manner in which the design is to be implemented.
  - Design forces describe the environment and conditions that must exist to make the design pattern applicable.
- The pattern characteristics (classes, responsibilities, and collaborations) indicate the attributes of the design that may be adjusted to enable the pattern to accommodate a variety of problems.
- These attributes represent characteristics of the design that can be searched (via database) so that an appropriate pattern can be found.
- Finally, guidance associated with the use of a design pattern provides an indication of the ramification of design decisions.
- The name of design patterns should be chosen with care.
- One of the key technical problems in software reuse is the inability to find existing patterns when hundreds or thousands of candidate patterns exist.
- The search of the “right” pattern is aided immeasurably by a meaningful pattern name.

Using Patterns in Design

Design patterns can be used throughout software design.

Once the analysis model has been developed, the designer can examine a detailed representation of the problem to be solved and the constraints that are imposed by the problem.

The problem description is examined at various levels of abstraction to determine if it is amenable to one or more of the following design patterns:

Architectural patterns: These patterns define the overall structure of the software, indicate the relationships among subsystems and software components, and define the rules for specifying relationships among the elements (classes, packages, components, subsystems) of the architecture.

Design patterns: These patterns address a specific element of the design such as an aggregation of components to solve some design problems, relationships among components, or the mechanisms for effecting component-to-component communication.

Idioms: Sometimes called ***coding patterns***, these language-specific patterns generally implement an algorithmic element of a component, a specific interface protocol, or a mechanism for communication among components.

Frameworks

- A framework is not an architectural pattern, but rather a skeleton with a collection of “plug points” (also called *hooks* and *slots*) that enable it to be adapted to a specific problem domain.
- Gamma et al note that:
  - Design patterns are more abstract than frameworks.
  - Design patterns are smaller architectural elements than frameworks
  - Design patterns are less specialized than frameworks