Requirements engineering:

Requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity. Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system. It encompasses seven distinct tasks: ***inception, elicitation, elaboration, negotiation, specification, validation,*** and ***management***.

Inception. In general, most projects begin when a business need is identified. At project inception, you establish a basic understanding of the problem, the people who want a solution,

a) Identifying Stakeholders: The usual stakeholders are: business operations managers, product managers, marketing people, internal and external customers, end users consultants, product engineers, software engineers, and support and maintenance engineers.

b) Recognizing Multiple Viewpoints: Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. As information from multiple viewpoints is collected. You should categorize all stakeholder information in a way that will allow decision makers to choose an internally consistent set of requirements for the system.

Working toward Collaboration: If five stakeholders are involved in a software project, you may have five different opinions about the proper set of requirements. The job of a requirements engineer is to identify areas of commonality and areas of conflict or inconsistency. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion" may make the final decision about which requirements make the cut.

b)Elicitation. Ask the customer, what the objectives for the system or product are, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. A number of problems that are encountered as elicitation occur.

- **Problems of scope**. The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
- **Problems of understanding**. The customers/users are not completely sure of what is needed, specify requirements that conflict with the needs of customers/users, or specify requirements that are ambiguous or untestable.
- **Problems of volatility**. The requirements change over time.

c)Elaboration. The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information.

Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system.

d)Negotiation. Different customers or users propose conflicting requirements, arguing that their version is "essential for our special needs." It is a process of negotiation. Customers, users, and other stakeholders are asked to *rank requirements* and then discuss conflicts in priority.

e)Specification. A specification can be a written document(SRS) that is presented in a consistent and therefore more understandable manner

f) Validation: Is each requirement consistent with the overall objective for the system. Have all requirements been specified at the proper level. Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system.

g)**Requirements Management**: During requirements management, the project team performs a set of activities to identify, control, and track requirements and changes to the requirements at any time as the project proceeds .

Requirements modeling:

The intent of the analysis model is to understand more about what they really require.

Analysis Rules of Thumb:

- The analysis model should focus on requirements that are visible within the problem.
- Each element of the analysis model should add to an overall understanding of software requirements
- The model should delay the consideration of infrastructure and other nonfunctional models until the design phase
- The model should provide value to all stakeholders.
- The model should be kept as simple as can be

Elements of the Analysis Model:

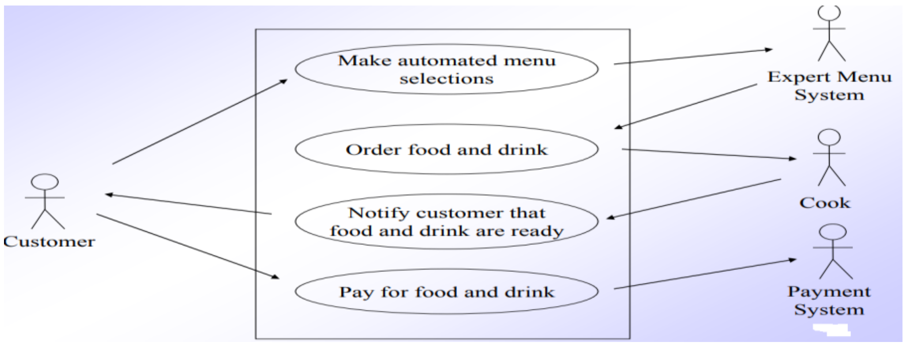
Flow-oriented modeling – provides an indication of how data objects are transformed by a set of processing functions
Scenario-based modeling – represents the system from the user's point of view
Class-based modeling – defines objects, attributes, and relationships
Behavioral modeling – depicts the states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

Scenario-based modeling:

Requirements modeling with UML begins with the creation of scenarios in the form of *use cases, activity diagrams, and swimlane diagrams.*

Creating a Preliminary Use Case: A use case describes a specific usage scenario from the point of view of a defined actor.

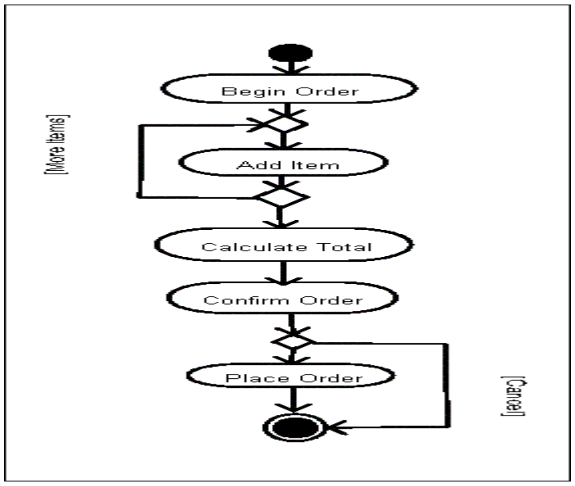
Use Case Diagram:



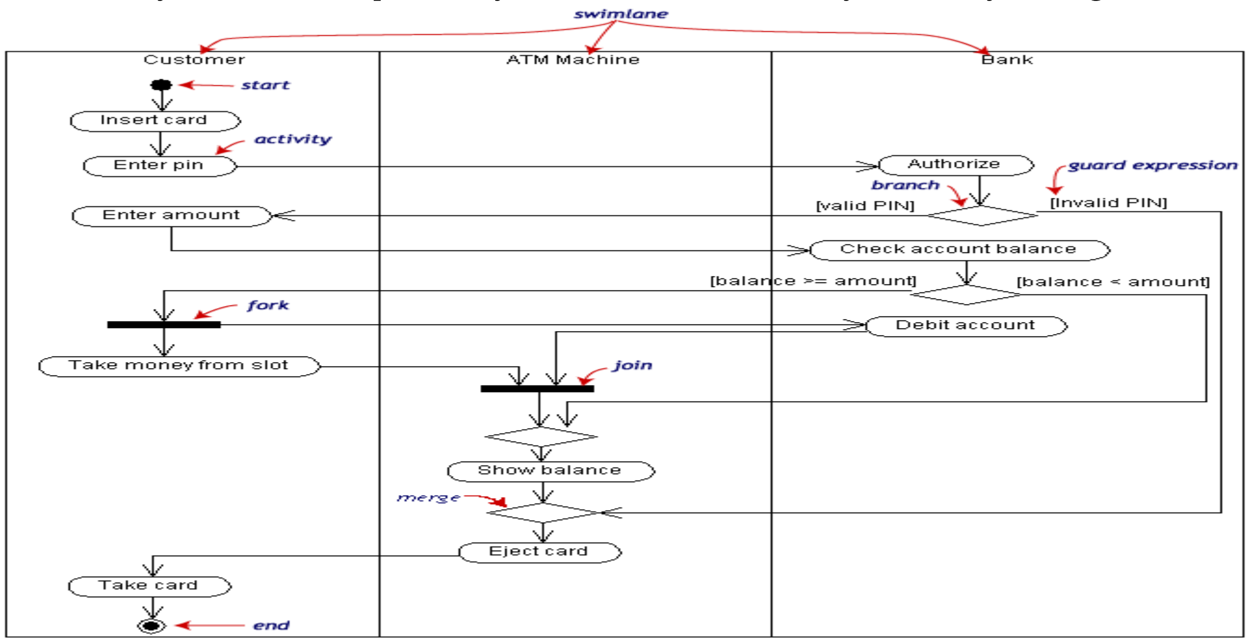
Developing an Activity Diagram: Supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario

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



Swimlane Diagrams: The UML swimlane diagram is a useful variation of the activity diagram and allows you to represent the flow of activities described by the use case and at the same time indicate which actor or analysis class has responsibility for the action described by an activity rectangle.



Data modeling concepts

A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is relationships.

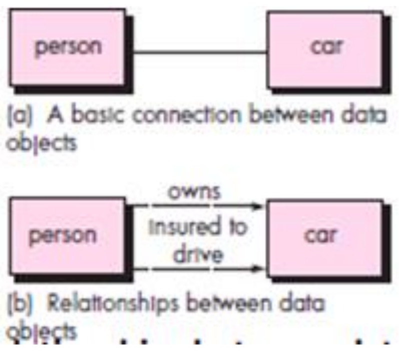
Data Objects:

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) or 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 Attributes: Data attributes define the properties of a data object.
Relationships: Data objects are connected to one another in different ways.

For example,
• A person owns a car.
• A person is insured to drive a car.



(a) A basic connection between data objects

(b) Relationships between data objects

Class-based modeling

The elements of a class- based model include classes and objects, attributes, operations, class responsibility, collaborator (CRC) models, collaboration diagrams, and packages.

Identifying Analysis Classes
Classes are determined by underlining each *noun* or *noun phrase* and entering it into a simple table. If the class (noun) is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a solution, it is part of the problem space.

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 about it must be remembered 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

Specifying Attributes: 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
Example :StudentName(Noun),ID(Noun),Branch(Noun)

Defining Operations: Operations define the behavior of an object. Although many different types of operations exist, they can generally be divided into four broad categories: (1) operations that manipulate data in some way. (2) Operations that perform a computation, (3) operations that inquire about the state of an object, and (4) operations that monitor an object for the occurrence of a controlling event.

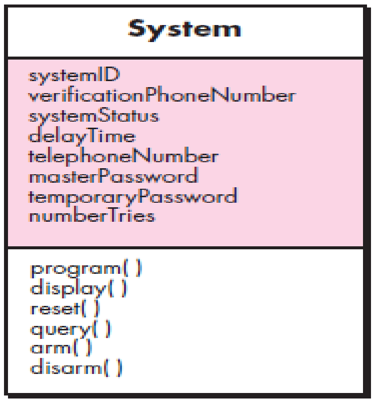
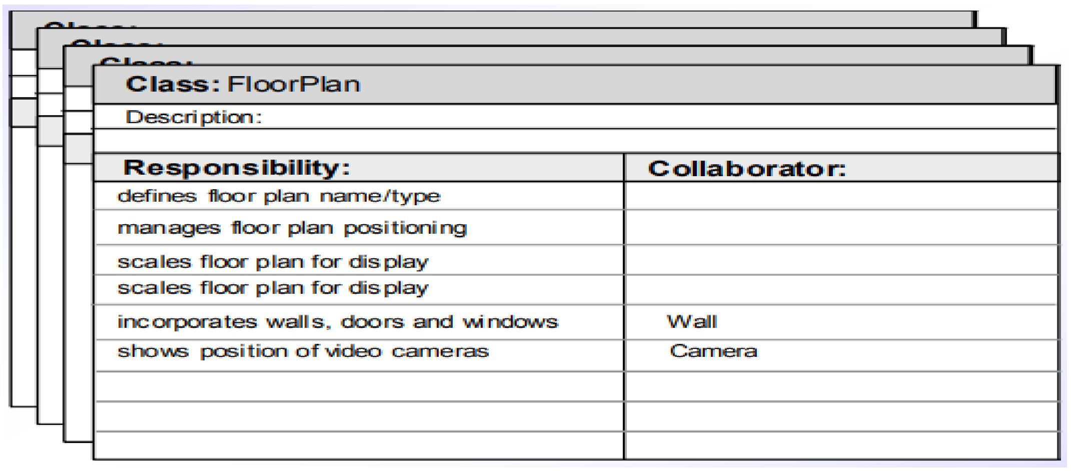


Fig 6.9: Class diagram for the System class

Class-Responsibility-Collaborator (CRC) Modeling: Class-responsibility-collaborator (CRC) modeling provides a simple means for *identifying* and *organizing* the classes that are relevant to system or product requirements.



Class: FloorPlan
Description:

Responsibility:	Collaborator:
defines floor plan name/type	
manages floor plan positioning	
scales floor plan for display	
scales floor plan for display	
incorporates walls, doors and windows	Wall
shows position of video cameras	Camera

Class Types:

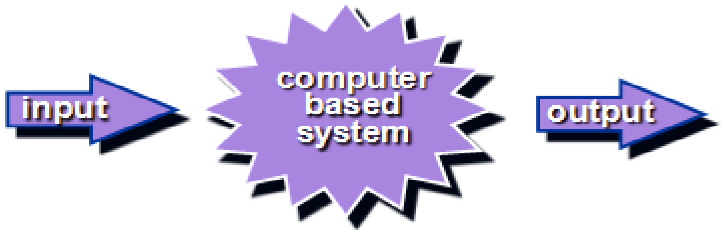
Entity classes: Represent things that are to be stored in a database and persist throughout the duration of the application.
Boundary classes are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
Controller classes manage a "unit of work" from start to finish.

Collaborations.
To help in the identification of collaborators, you can examine three different generic relationships between classes
(1) the is-part-of relationship,
(2) the has-knowledge-of relationship, and
(3) the depends-upon relationship.

Analysis Packages: An important part of analysis modeling is *categorization*. That is, various elements of the analysis model are categorized in a manner that *packages* them as a grouping—called an *analysis package*

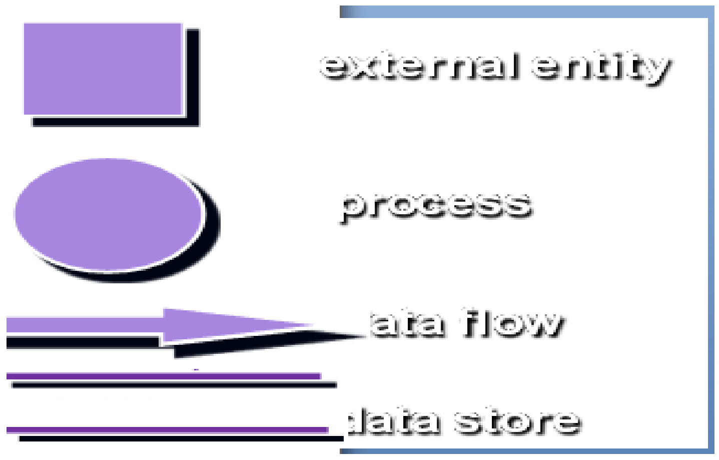
Flow-oriented modeling

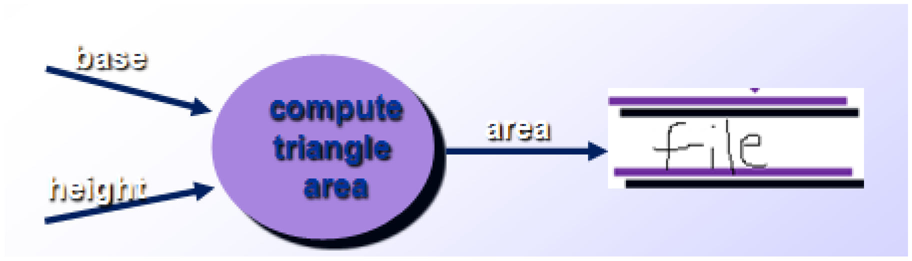
- Represents how data objects are transformed as they move through the system.
- The DFD takes an input-process-output insight into system requirements and flow.



- Data objects are represented by labeled arrows and transformations are represented by circles (called *bubbles*).

Flow Modeling Notations:

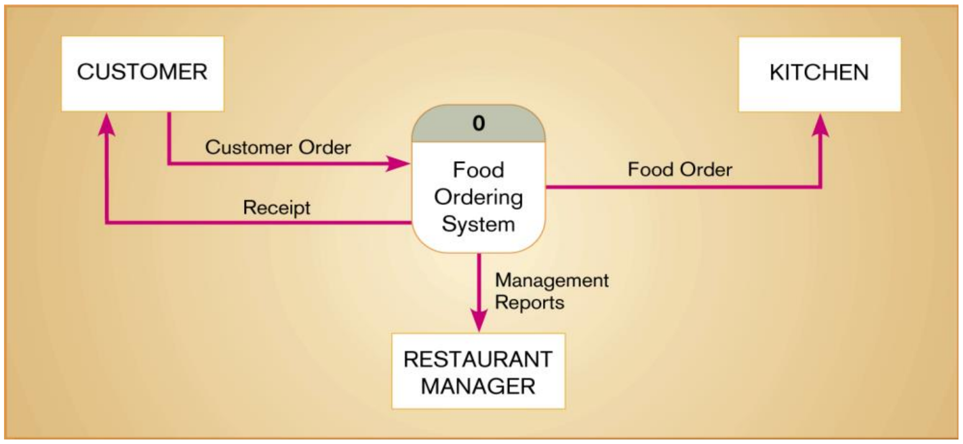




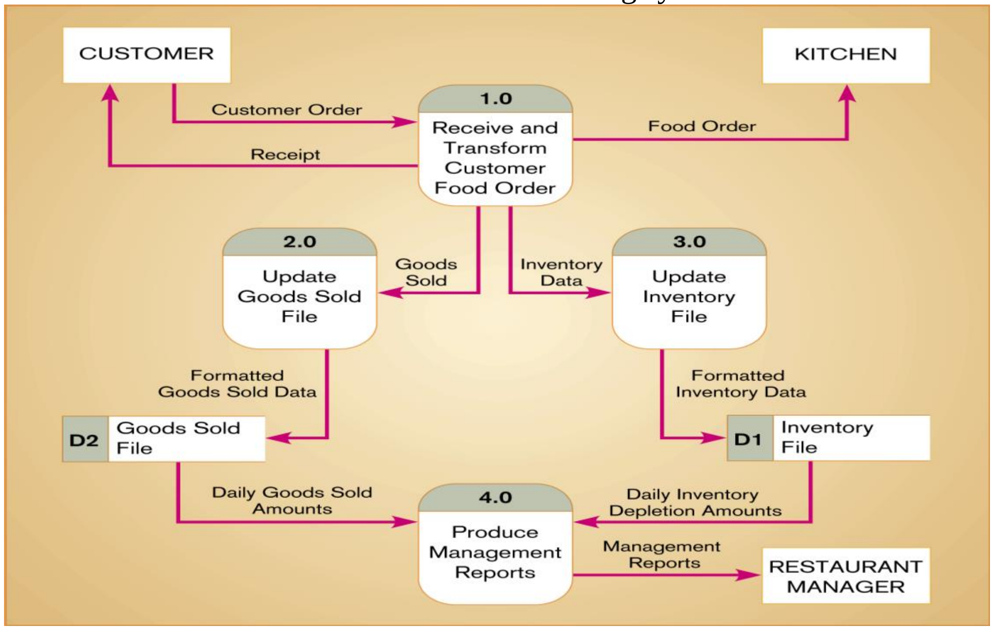
Creating a Data Flow Model: A data flow diagram (DFD) that represents a system's major processes, data flows and data stores at a high level of detail

- use a grammatical parse to determine "operations"
- Determine external entities (producers and consumers of data)
- Create a level 0 DFD

Context diagram of Food ordering system



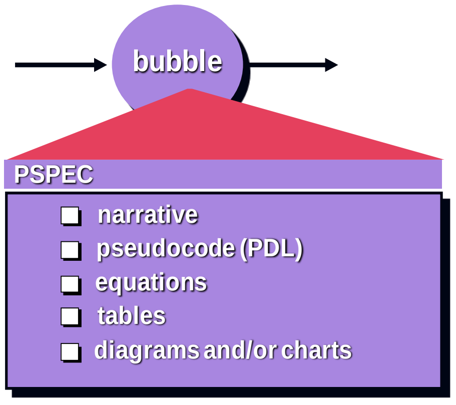
Level-1 DFD of food ordering system:



Creating a Control Flow Model: The diagram represents "events" and the processes that manage these events. An event or control item is implemented as a Boolean value (e.g., true or false, on or off, 1 or 0)

The Process Specification

The *Process Specification* (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. It is a “mini” specification for each transform at the lowest refined of a DFD.



Creating a behavioral model

We can represent the behavior of the system as a function of specific events and time.

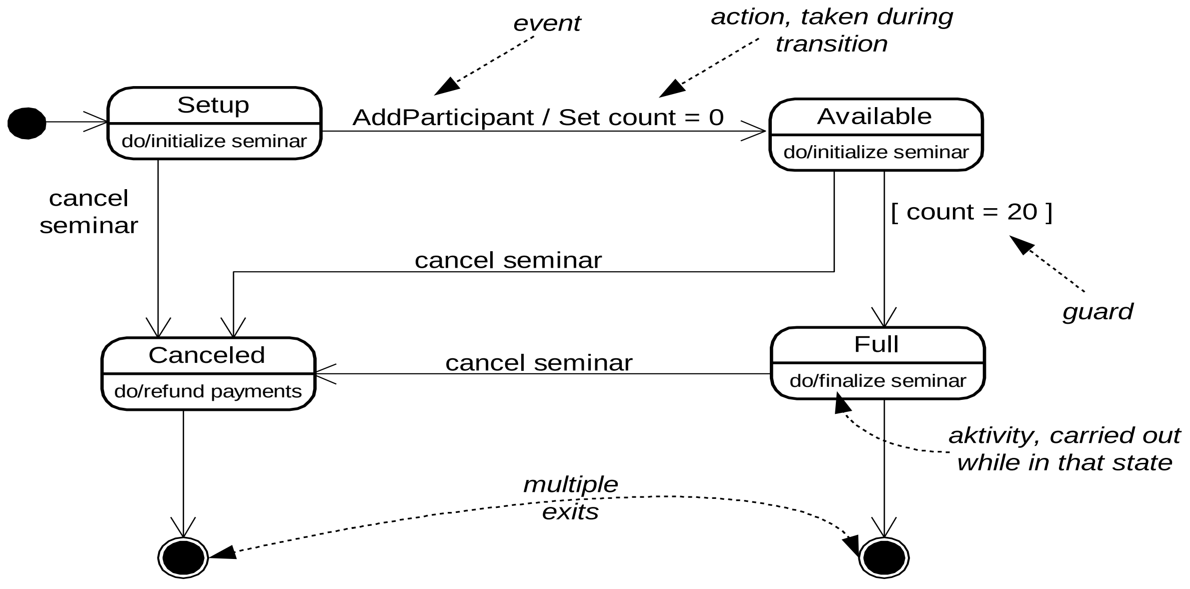
1) Identify events found within the use cases and implied by the attributes in the class diagrams
2) Build a state diagram for each class, and if useful, for the whole software system

Identifying Events with the Use Case: 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

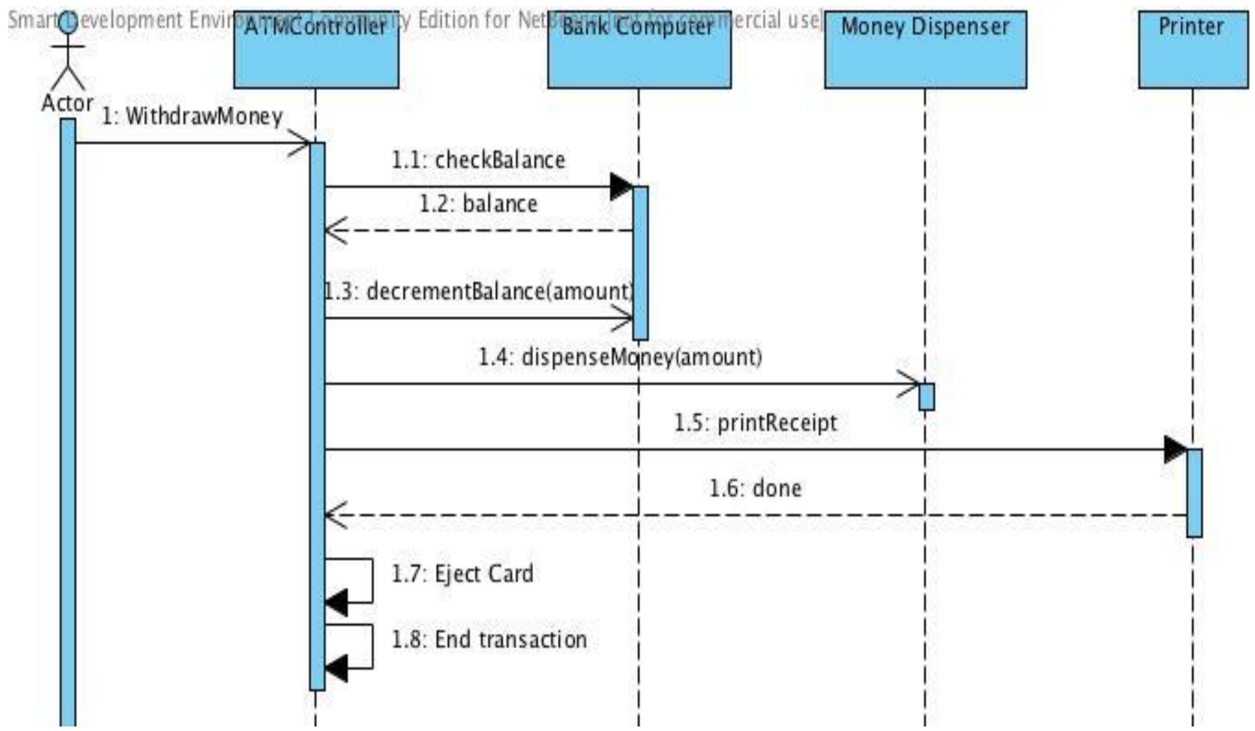
An example is the reading of a data item from the user versus comparing the data item to some possible value

State diagram Representations:

- 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
- 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(event) in a transition may contain the checking of the passive state of an object

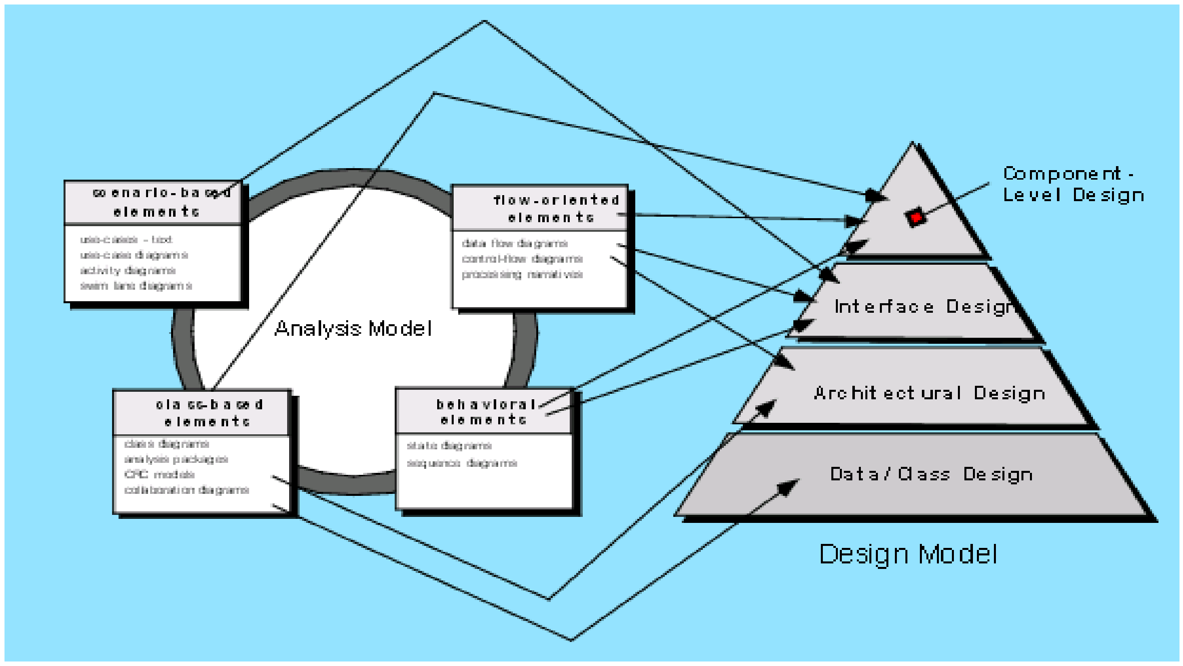


Sequence diagrams. The second type of behavioral representation, called a sequence diagram in UML, indicates how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time.



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 *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 humans who use it.

The *component-level design* transforms structural elements of the software architecture into a software components (modules).

Design Process and Quality:

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

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:

- 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.
- 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.

Quality Attributes:

- *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.

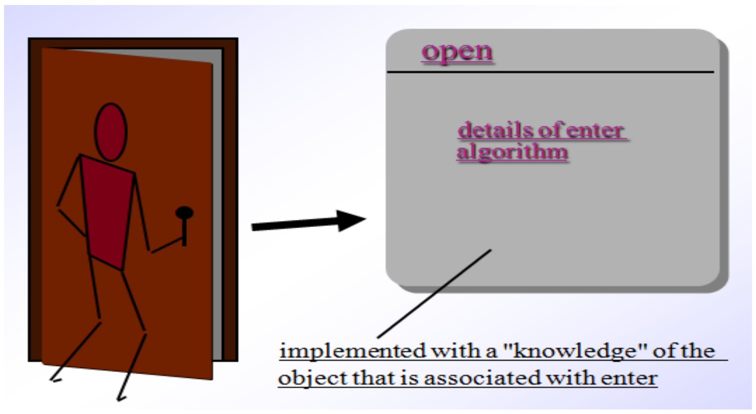
Design Concepts:

- abstraction—data, procedure, control
- architecture—the overall structure of the software
- patterns—"conveys the essence" of a proven design solution
- modularity—compartmentalization of data and function
- hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- refinement—elaboration of detail for all abstractions
- Refactoring—a reorganization technique that simplifies the design

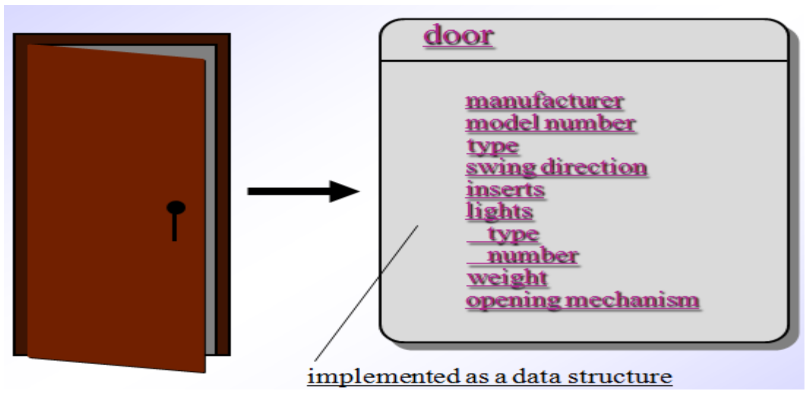
Abstraction

- Allows designers to focus on solving a problem without being concerned about irrelevant lower level details.
- 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 procedural orientation is taken.

Procedural Abstraction


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

Data abstraction:



Architecture

Architecture is the overall structure of organization of program components (modules).

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.

Patterns

A design pattern "conveys the essence of a proven design solution to a complex problem"

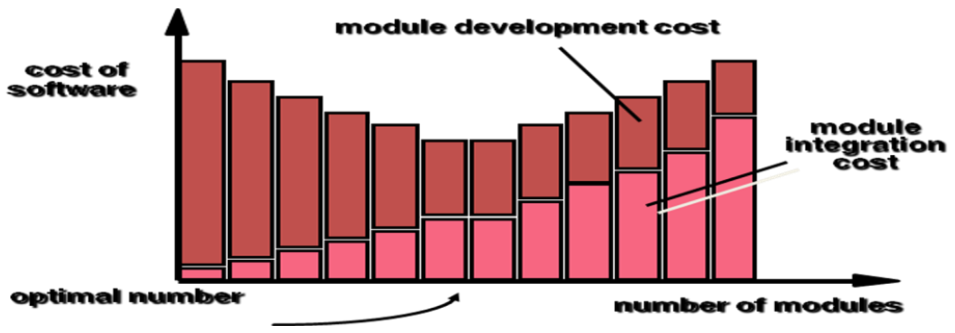
Design Pattern Template

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

Modularity

Software is divided into separately named and addressable components, sometimes called modules that are integrated to satisfy problem requirements. 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

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.

Functional Independence

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

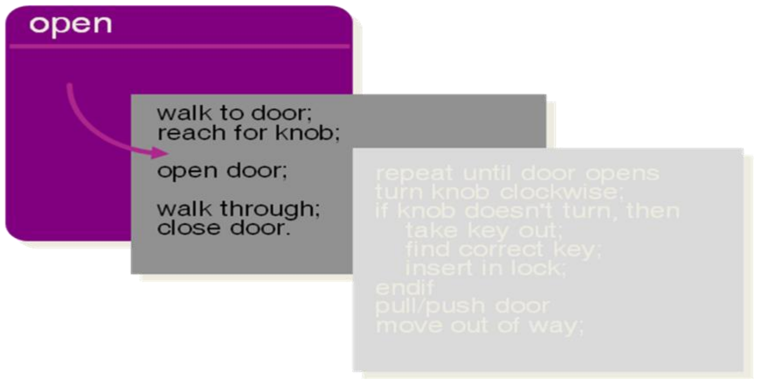
Cohesion is the degree to which module performs one and only one function.

Coupling is the degree to which module is connected to other modules in the system.

Refinement

It is the elaboration of detail for all abstractions, providing more and more detail as each successive refinement (elaboration) occurs.

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



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.

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.

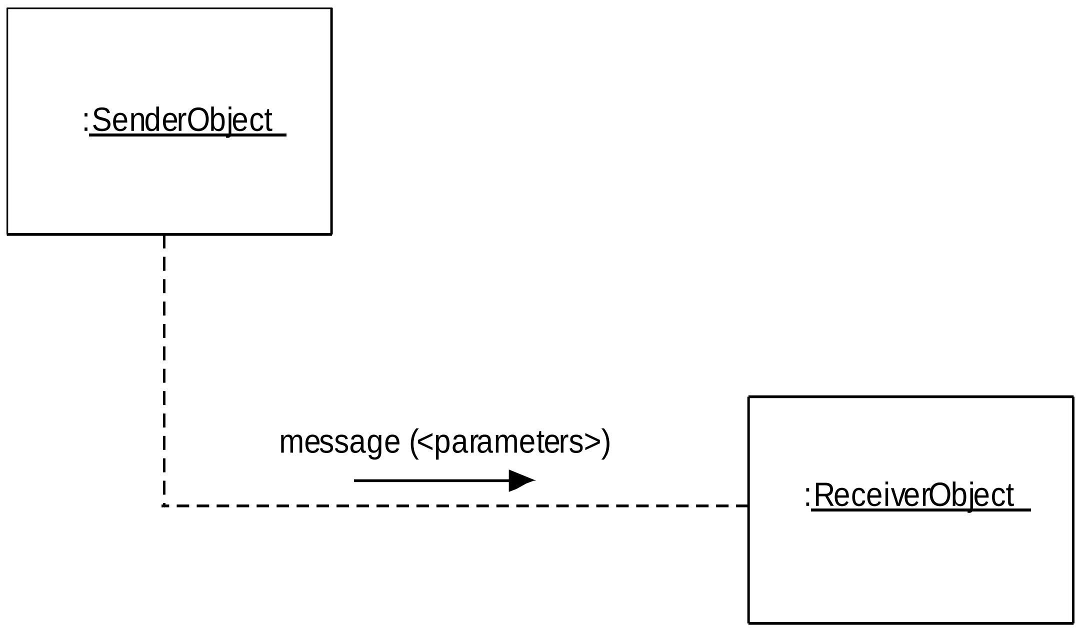
Five different classes' types are shown below:

1. *User Interface classes*: Necessary for Human computer Interaction.
2. *Business domain classes*: The classes identify the attributes and services that are required to implement functionality of the system.
3. *Process classes:* implement lower-level 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.

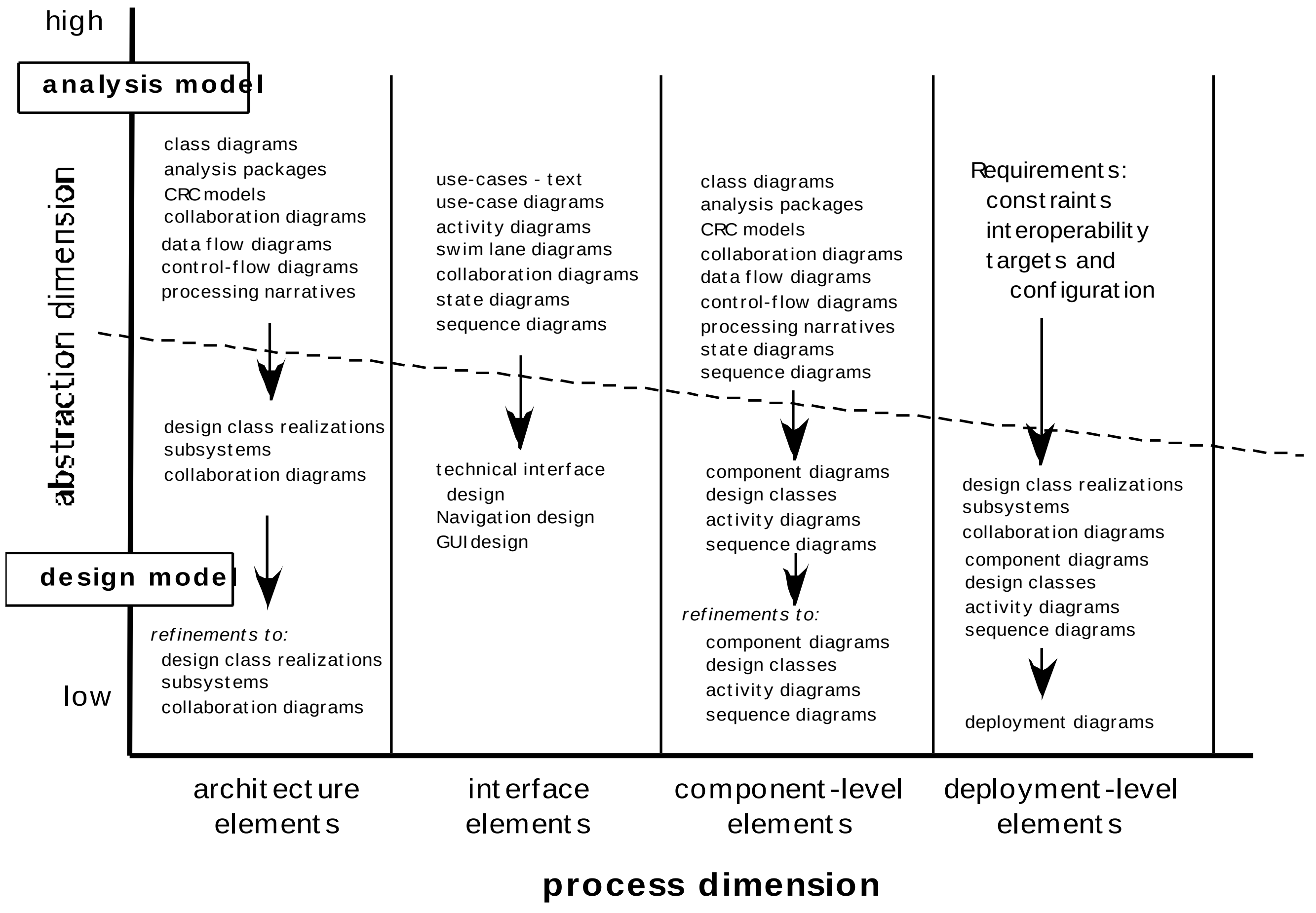
Inheritance

- 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



The Design Model



Design Model Elements:

- Data elements
  - Data model --> data structures
  - Data model --> database architecture
- Architectural elements “similar to the floor plan of a house”
  - 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.

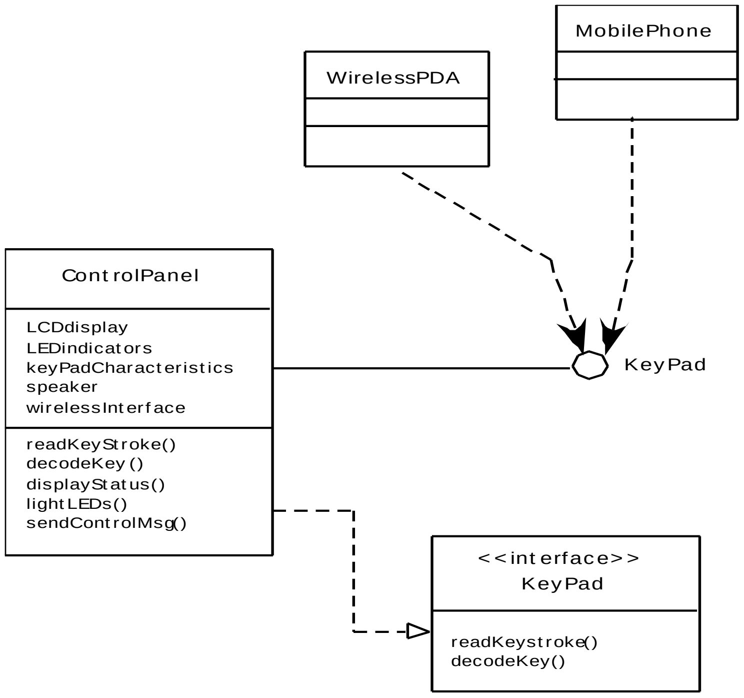


Figure 9.6 UML interface representation for **ControlPanel**

- 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.