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 within the context of SE, Design Process and Design quality, Design concepts, The Design Model, Pattern-based Software Design.

BUILDING THE ANALYSIS MODEL

REQUIREMENTS ANALYSIS: Requirements analysis results in specification of software's operational characteristics, indicates software's interface with other system elements and establishes constraints that software must meet.

- ✓ It allows the Software Engineers (sometimes called analysts/modelers) to elaborate on basic requirements established during earlier Requirements Engineering tasks and build models that depict user scenarios, functional activities, problem classes and their relationships, behavior and flow of data.
- ✓ It provides the designer with a representation of function, information and behavior that can be translated to architectural, interface and component- level designs.
- ✓ Finally, analysis model and requirements specification provide developer and the customer with the means to assess quality once software is built.
- ❖ Analyst should model what is known and use that model as the basis for design of software increment.

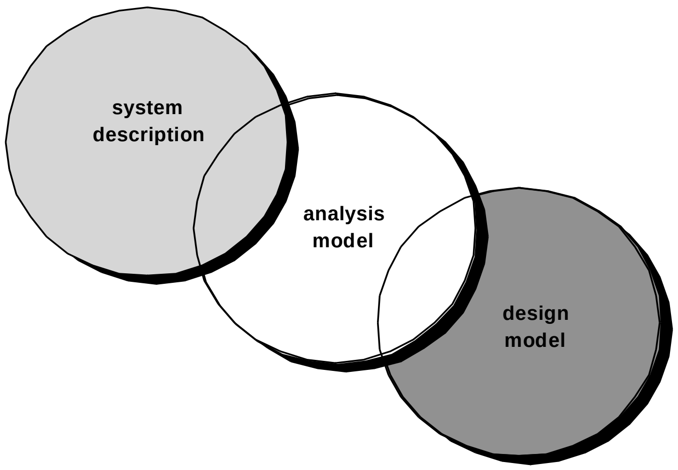


Figure: Analysis Model as a Bridge between System Description and Design Model

Overall Objectives and Philosophy:

The analysis model must achieve 3 primary objectives:

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

All elements of analysis model are directly traceable to parts of the design model. A clear division of design and analysis tasks between these two important modeling activities is not always possible.

Analysis Rules Of Thumb:

1. "The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high". (Don't show details that explain how system works.)
2. "Each element of analysis model should add to an overall understanding of software requirements and provide insight into the information domain, function and behavior of system".
3. "Delay consideration of infrastructure and other non- functional models until design". (For ex. A DB may be required, but classes, functions required to access it should be considered only after problem domain analysis is completed.)
4. "Minimize coupling throughout the system". It is important to represent relationships between classes and functions. Efforts should be made to reduce their "interconnectedness".
5. "Be certain that the analysis model provides value to all stakeholders". Each constituency has its own use for model.
6. "Keep the model as simple as it can be". Don't add additional diagrams when they provide no new information.

DOMAIN ANALYSIS: "Domain analysis is the identification, analysis and specification of common requirements from a specific application domain, for reuse on multiple projects within that application domain."

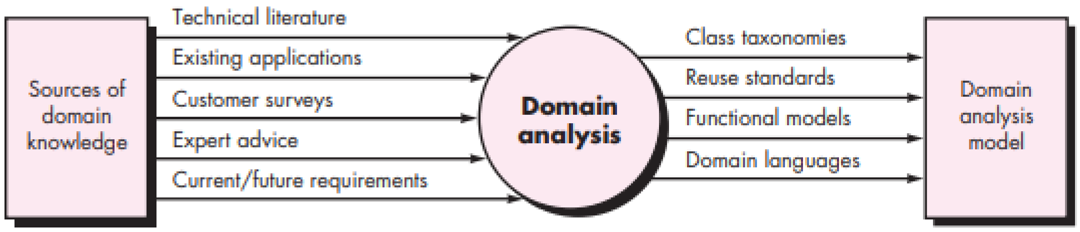


Figure: Input and output for Domain Analysis

- ❖ The "Specific Application Domain" can range from banking, multimedia video games to software embedded within medical devices.

Goal of domain analysis is to find or create those classes/analysis classes and/or common functions and features that are broadly applicable, so they may be reused. Role of domain analyst is to discover and define reusable analysis patterns, analysis classes and related information that may be used by many people working on similar but not necessarily same applications.

DATA MODELING CONCEPTS: Analysis modeling begins with data modeling. Its concepts are:

1. **Data Objects:** "A data object is a representation of any composite information that must be understood by software". A data object can be an external entity (Ex: Anything that produces or consumes information), a thing (Ex: Report or display), an occurrence (Ex: Telephone call) or an event (Ex: an alarm), a role, an organizational unit, a place or a structure.

- ✓ A data object encapsulates data (attributes) only. There is no reference within a data object to operations that act on the data.
- ✓ Data object description incorporates data object and all of its attributes.

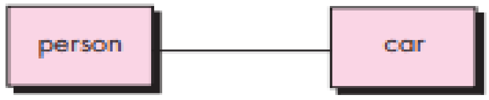
2. **Data Attributes:** "These define properties of a data object and take on one of three different characteristics". They may be used to:
   - Name an instance of data object.
   - Describe the instance of data object.
   - Make reference to another instance in another table.

One or more attributes must be defined as an identifier; it becomes a "key" when we want to find an instance of the data object. Values for identifiers are generally unique.

3. **Relationships:** Data objects are connected to one another in different ways. We can define a set of object/relationship parts that define relevant relationships.

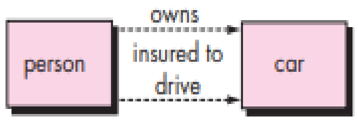
<u>Types of Relationships:</u>

*1) **Association Relationship:***



(a) A basic connection between data objects

*2) **Dependency Relationship:***



(b) Relationships between data objects

4. **Cardinality and Modality:** Cardinality refers to the process of mapping between various objects. For example, if object X relates to object Y, cardinality specifies how many occurrences of object X are related to how many occurrences of object Y. Cardinality also defines "the maximum number of objects that can participate in a relationship".

Modality provides an indication of whether or not a particular data object must participate in the relationship. Modality is "1" if an occurrence of relationship is mandatory or it is "0".

ANALYSIS MODELING APPROACHES:

1. **Structured Analysis:** Considers data and processes that transform data as separate entities. Data objects are modeled in a manner that shows how they transform data as data objects flow through the system.
2. **Object-Oriented Analysis:** It focuses on definition of classes and the manner in which they collaborate with one another to effect customer requirements.UML and unified process are predominantly object oriented.

The intent of Object-Oriented Analysis is to define all classes and relationships and behavior associated with them that are relevant to the problem to be solved. To accomplish this, a number of tasks must occur:

1. Basic user requirements must be communicated between user & Software Engineer.
2. Classes must be identified (i.e., attributes and methods are defined).
3. A class hierarchy is defined.

4. Object-to-object relationships should be represented.
5. Object behavior must be modeled.
6. Repeat tasks 1 to 5 until model is complete.

- ❖ Object-Oriented Analysis builds a class-oriented model that relies on understanding of Object- Oriented concepts.

Analysis modeling leads to derivation of each of the modeling elements as shown below:

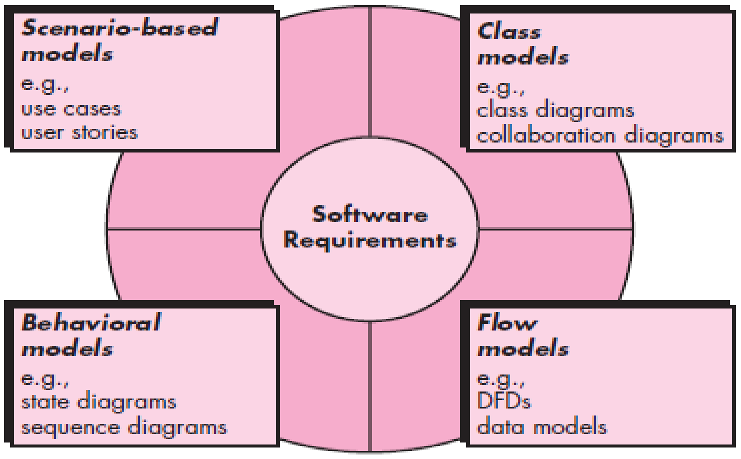

Scenario-based models
e.g., use cases user stories
Class models
e.g., class diagrams collaboration diagrams
Software Requirements
Behavioral models
e.g., state diagrams sequence diagrams
Flow models
e.g., DFDs data models

Figure: Elements of Analysis Model

- ❖ The specific content of each element (diagrams/models used to construct element) differs from project to project.

SCENARIO-BASED MODELING: Analysis modeling with UML begins with creation of scenarios in the form of use-cases, activity diagrams and swim lane diagrams.

1) WRITING USE CASES: “A Use Case is defined as a set of sequence of actions performed by an actor to obtain a specific output”.

"An actor may be a person that uses a system or product, or a system itself, anything that performs an action in system".

- ❖ A use-case captures the interactions that occur between producers and consumers of information and system itself.
- ❖ The concept of a use-case is relatively easy to understand- describe a specific usage scenario in straight forward language from point of view of defined actor.

What to write about? The first two requirement engineering tasks-inception and elicitation-provide us the information we need to begin writing use cases.

- ✓ To begin developing a set of use-cases, activities performed by a specific actor are listed.
- ✓ As conversations with stakeholder progress, the Requirement Engineering team develops use-cases for each of activities noted.
- ✓ A variation of a formal use-case presents the interaction as an ordered sequence of user actions. Each action is represented as a declarative sentence.

It is important to note that sequential presentation does not consider any alternative interactions. Such use cases are referred to as "***primary scenarios***".
A description of alternative interactions is essential for complete understanding of the function, by asking the following questions: [Answers result in secondary use cases].

1. Can the actor take some other action at that point?
2. Is it possible that actor will encounter some error at this point? If so, what might it be?
3. Is it possible that actor will encounter some other behavior? If so, what might it be?

2) USE – CASE DIAGRAMS: "A Use Case is defined as a set of sequence of actions performed by an actor to obtain a specific output".

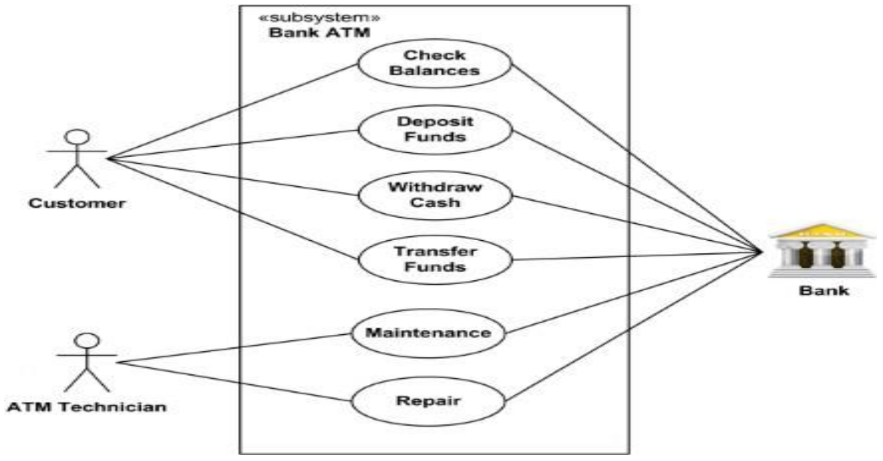


Figure: Use-Case Diagram for ATM

3) ACTIVITY DIAGRAMS: "The UML activity diagram supplements (helps) the use-case by providing a graphical representation of flow of interaction within a specific scenario".

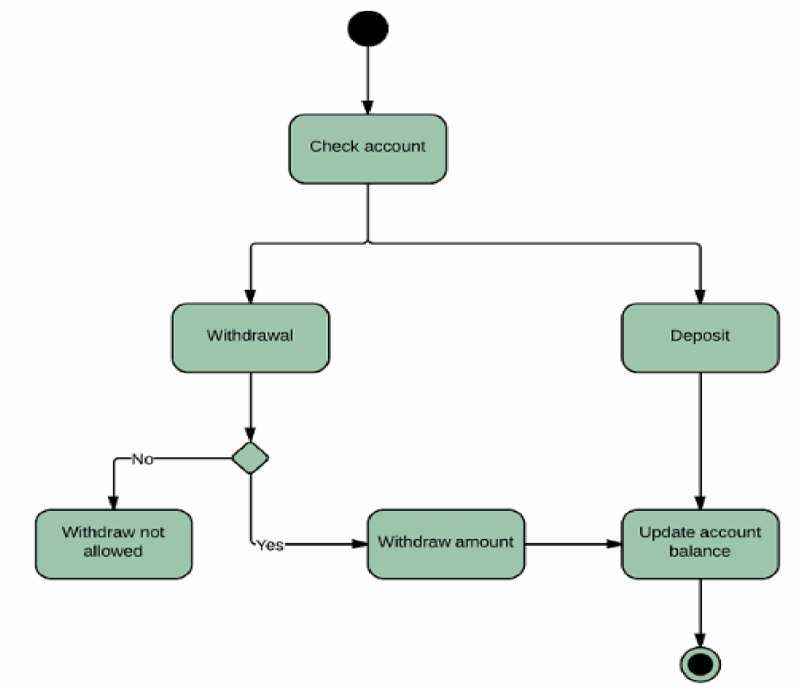


Figure: Activity Diagram for Bank

Similar to flow chart, an activity diagram uses *Rounded Rectangles* to imply a specific system function, *Arrows* to represent flow through the system, *Diamonds* to depict a branching decision and *Solid Horizontal Lines* to indicate parallel activities that occur.

4) **SWIMLANE DIAGRAMS:** "It is a useful variation of activity diagram and allows the analyst to represent flow of activities described by use-case and indicate which actor or analysis class has responsibility for action described by an activity rectangle."

Activities are represented as parallel segments that divide diagram vertically like lanes of swimming pool.

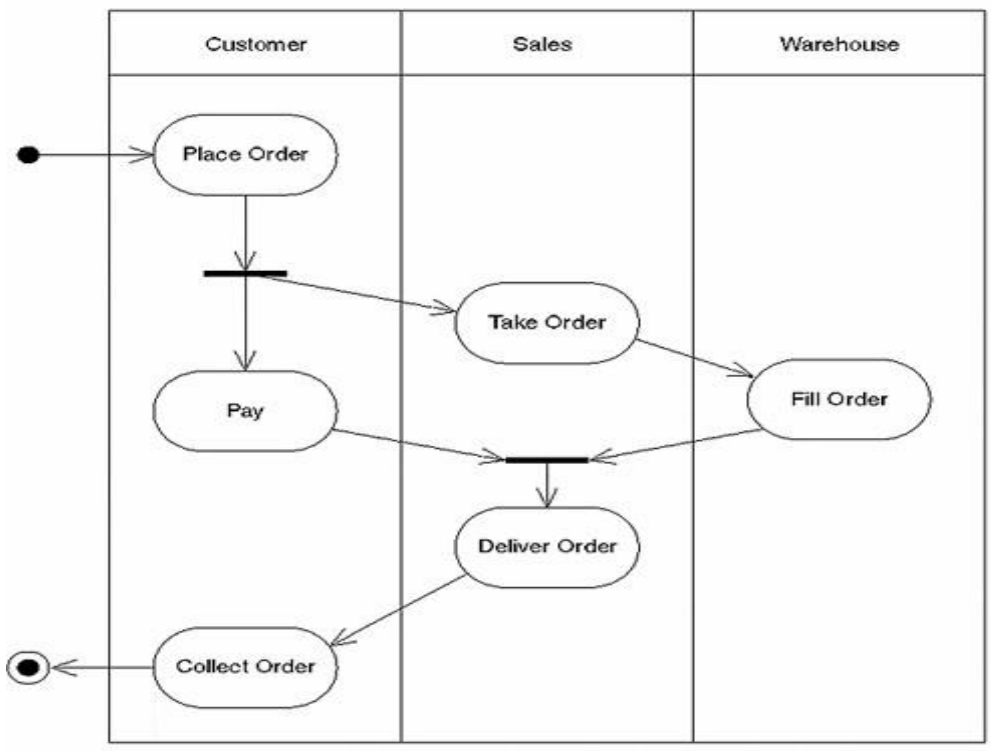
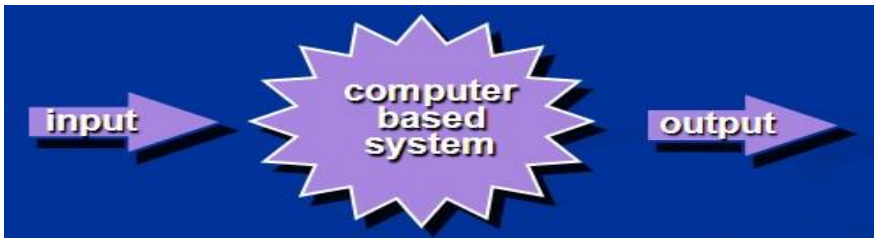


Figure: Swimlane Diagram

FLOW - ORIENTED MODELING: Data Flow Diagrams (DFD) can be used to complement UML diagrams and provide additional insight into system requirements and data flow. DFD takes an ***input-process-output*** view of a system, i.e., data objects flow into the software, transformed by processing elements and resultant data objects flow out of software.



- ❖ Data objects are represented by labeled arrows, i.e., Data Flow.
- ❖ External entities are represented by square. (Producer/Consumer of data).
- ❖ Processes/Transformations are represented by bubbles/circles.
- ❖ Data stores represented by____________ (or) cylinders.

1. **CREATING A DATA FLOW MODEL:** Data Flow Diagram enables the software engineer to develop models of information and functional domains at the same time.

Guidelines for DFDs:

1) Level 0 DFD should depict software /system as a single bubble.
2) Primary input and output should be carefully noted.
3) Refinement should begin by isolating candidate processes, data objects and data stores to be represented at next level.
4) All arrows & bubbles should be labeled with meaningful names.
5) Information flow continuity must be maintained between levels.
6) One bubble at a time should be refined.

There is a natural tendency to overcomplicate DFD, when analyst attempts to show too much detail early.

❖ DFD is represented in a hierarchical fashion i.e., the first data flow model (sometimes called a **Level 0 DFD or Context Level DFD**) represents system as a whole. Subsequent DFD's refines context diagram, providing increasing detail with each level. Level 0 DFD will contain only one bubble (Main process).

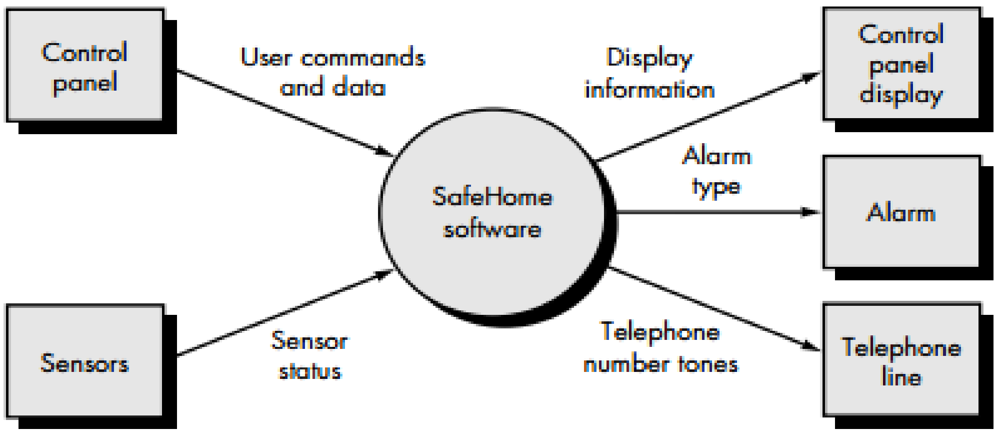


Figure: Context-Level DFD/DFD-0 Diagram

The level-0 DFD is then expanded into a level-1 DFD. This is done by performing a "grammatical parse" on the narrative that describes context level bubble.

A ***processing narrative*** is similar to use-case in style but different in purpose. It provides an overall description of the function to be developed, not from one actor point of view.

By performing a grammatical parse on processing narrative for a bubble at any DFD level, we can generate much useful information about how to proceed with refinement to next level.

The processes represented at a level-1 DFD can be further refined into lower levels, the continuity of information flow will be maintained between all these levels. Refinement of DFDs continues until each bubble performs a simple function, so that it will be easily implemented as a program component. This concept called as "cohesion" can be used to assess simplicity of a given function.

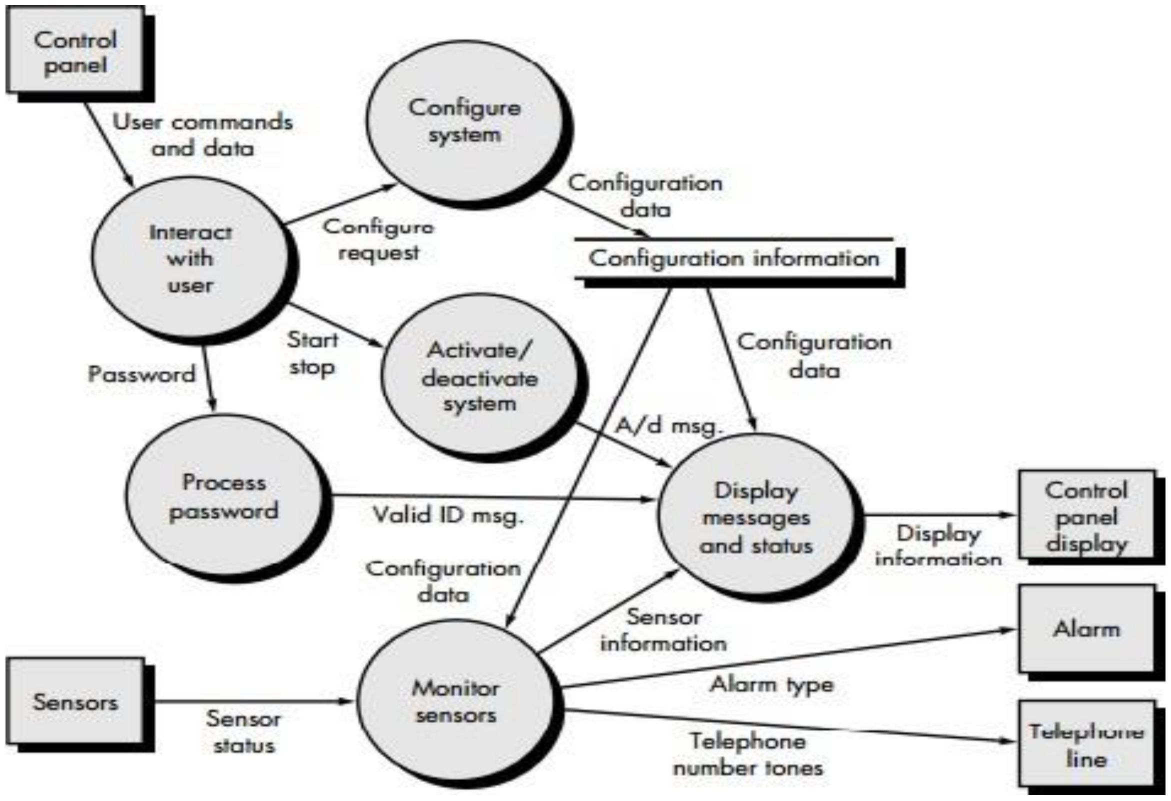


Fig.: Level -1 DFD Diagram

2. **CREATING A CONTROL FLOW MODEL:** A large class of applications is driven by events rather than data, produce control information and process information with concern for time and performance. Such applications require use of control flow modeling in addition to data flow modeling. Event or control item is implemented as a Boolean value (T/F, on/off, yes/no, 1/0) or a discrete list of conditions.

Guidelines to select possible events:

1) List all sensors that are "read" by the software.
2) List all interrupt conditions.
3) List all data conditions.
4) List all "switches" actuated by an operator.
5) Describe behavior of a system by identifying its states, identify how each state is reached, and define state transitions.
6) Focus on possible omissions-common error in specifying control.

<u>Control Specification</u>: Control specification (CSPEC) represents behavior of system in 2 different ways.

1. It contains a state diagram that is sequential specification of behavior.
2. It can also contain a program activation table- a combinational specification of behavior.

❖ By reviewing the state diagram, software engineer can determine system behavior, and more importantly, can ascertain whether there are "holes"' in the specified behavior.

❖ CSPEC describes the behavior of the system, but it gives us no information about inner working of processes, that are activated as a result of this behavior, which is the disadvantage of CSPEC.

<u>Process Specification</u>: The process specification (PSPEC) is used to describe all flow model processes that appear at final level of refinement. The content of PSPEC includes narrative text, a Program Design Language (PDL) description of process algorithm, mathematical equations, tables, diagrams or charts.

- By providing PSPEC to accompany each bubble in flow model, software engineer creates a "mini-spec" that can serve as guide for design of software component to implement process.

CLASS-BASED MODELING:

<u>IDENTIFYING ANALYSIS CLASSES</u>: To identify classes by examining the problem statement or by performing a "grammatical parse" on processing narratives developed for the system to be built, underline each noun or noun clause and enter it into a simple table. Synonyms should be noted. Analysis classes manifest in one of following ways:

1) **External Entities:** External entities produce or consume information to be used by a computer-based system.
2) **Things:** Things are part of information domain for the problem. (Ex: Reports, Designs, Signals)
3) **Occurrences/Events:** These occur within context of system operation. (Ex: Completion of a robot movement.)
4) **Roles:** Roles are played by people who interact with system. (Ex: Manager)
5) **Organizational Units:** These are the units relevant to an application. (Ex: Team, Division)
6) **Structures:** Structures define a class of objects. (Ex: Sensor)
7) **Places:** Places establish context of the problem.(Ex: The Manufacturing Floor)

Coad and Yourdon suggest six selection characteristics that should be used to include each potential class in analysis model.

1) **Retained Information:** The potential class will be useful during analysis only if information about it must be remembered so that system can function.
2) **Needed Services:** Class must have set of identifiable operations that can change value of its attributes in some way.
3) **Multiple Attributes:** A class with multiple attributes is more useful during design, than a single attribute.
4) **Common Attributes:** A set of attributes can be defined for potential class, and these attributes apply to all instances of class.
5) **Common Operations:** A set of operations can be defined for potential class, and these operations apply to all instances of class.
6) **Essential Requirements:** External entities that produce or consume information essential to operation of any solution for system will always be defined as classes in requirements model.

<u>Specifying Attributes</u>: Attributes define the class, i.e., what is meant by the class in the context of problem space. To develop a meaningful set of attributes for an analysis class, a software engineer can again study a use-case and select those “things” that reasonably “belong” to the class.

- ✓ What data items fully define class in context of problem at hand? Should be answered.

<u>Defining Operations</u>: Operations define behavior of an object. Operations can be classified as:

1) Operations that manipulate data in some way. (Ex: Adding, Deleting, Selecting)
2) Operations that perform computation.
3) Operations that inquire about state of an object.
4) Operations that monitor an object for occurrence of a controlling object.

- ❖ An operation must have "knowledge" of the nature of the class attributes and associations.
- ❖ The analyst can again study a use-case and select those operations that reasonably belong to the class.

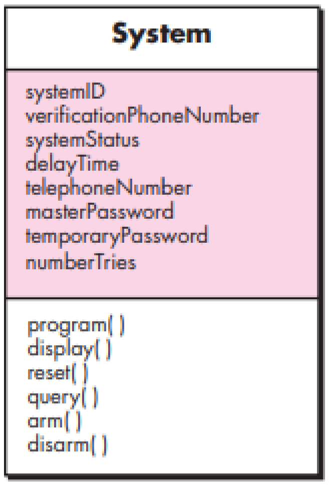


Fig.: Class Diagram

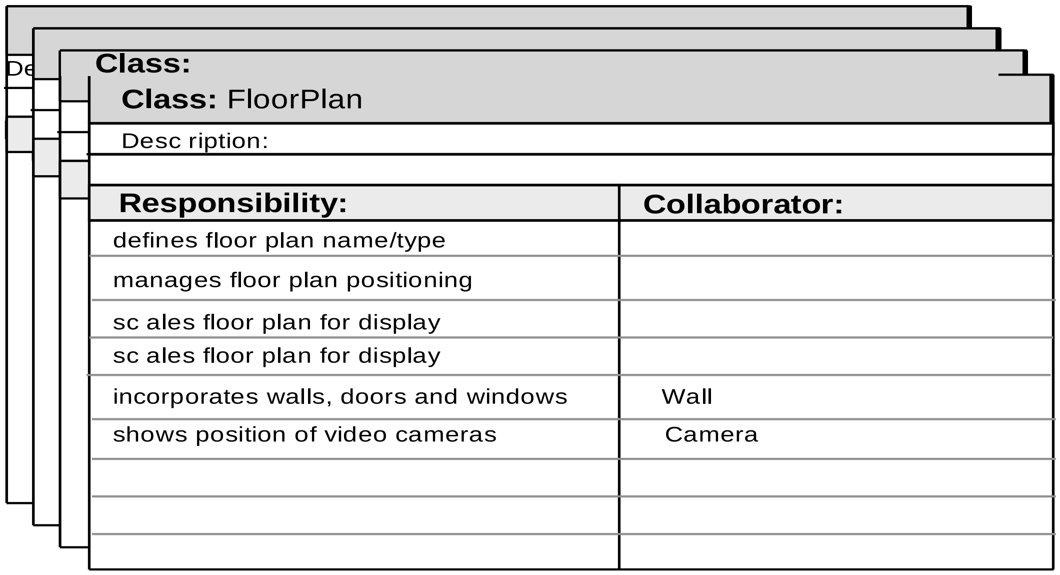
<u>CLASS-RESPONSIBILITY-COLLABORATOR (CRC) MODELING</u>: "CRC Modeling provides a means for identifying and organizing the classes that are relevant to system or product requirements". The intent of CRC model is to develop an organized representation of classes, using actual or virtual index cards.

- ✓ Responsibilities are the attributes and operations that relevant for the class.
- ✓ Collaborators are those classes that are required to provide a class with information needed to complete a responsibility.

Types of Classes in CRC:

1. **Entity Class:** It Contains information important to users. It is also called model or business classes, are executed directly from statement of the problem. These classes represent things that are to be stored in a database and persist throughout duration of the application.
2. **Boundary Class:** Used to create interface that the user sees and interacts with as the software is used. These classes are designed with responsibility of managing the way entity objects are represented to users.
3. **Controller Class:** Manage a "unit of work" from start to finish. These can be designed to manage,
    i. Creation or update of entity objects;
    ii. Complex communication between sets of objects;
    iii. Instantiation of boundary objects, as they obtain information from entity objects.
    iv. Validation of data communicated between objects or between user &application.

- ✓ Controller classes are not considered until design has begun.


De
Class:
Class: FloorPlan
Desc ription:
Responsibility:
Collaborator:
defines floor plan name/type
manages floor plan positioning
sc ales floor plan for display
sc ales floor plan for display
incorporates walls, doors and windows
Wall
shows position of video cameras
Camera
Fig: CRC Modeling Cards

Responsibilities:

<u>Guidelines for Allocating Responsibilities to Classes</u>:

1. <u>System Intelligence should be distributed across classes to best address the needs of the problem</u>: Intelligence is what the system knows and what it can do."Dumb" classes (those have few responsibilities) can be modeled to act as servants to few "smart" classes (those have many responsibilities). Flow of control will be straight forward.

   Disadvantages:
   - ✓ Concentrates all intelligence within a few classes, making changes more difficult.
   - ✓ Tends to require more classes, so more development effort.

❖ If system intelligence is more evenly distributed across classes in an application, maintainability of software is enhanced, impact of side effects due to change are reduced.

2. <u>Each Responsibility should be stated as generally as possible</u>: It implies that general responsibilities (both attributes and operations) should reside high in class hierarchy (parent class)
3. <u>Information and behavior related to it should reside within the same class</u>: Data and processes that manipulate data should be packaged as class, i.e., Encapsulation.
4. <u>Information about one thing should be localized with a single class, not distributed across multiple classes</u>: A single class should take on the responsibility for storing and manipulating specific type of information. If information is distributed, software becomes more difficult to test and maintain.
5. <u>Responsibilities should be shared among related class when appropriate</u>: A variety of related objects must exhibit the same behavior at the same time.

Collaborators: Classes fulfill their responsibilities in one of two ways:

i. A class can use its own operations to manipulate its own attributes, there by fulfilling a particular responsibility.

ii. A class can collaborate with their classes.

Collaborations identify relationships between classes. When a set of classes collaborate to achieve some requirement, they can be organized into a sub-system. If a class cannot fulfill a responsibility, then it needs to interact with another class, hence collaboration. To identify collaborations, analysts can examine three different generic relationships between classes:

- The depends-upon relationship (dependency relationship).
- The has-knowledge-of relationship.
- The is-part-of relationship(Aggregation)

Index Card: Index card, contains a list of responsibilities, and corresponding collaborations that enable responsibilities to be fulfilled.

❖ When a complete CRC model has been developed, representatives from customer & software organizations can review.

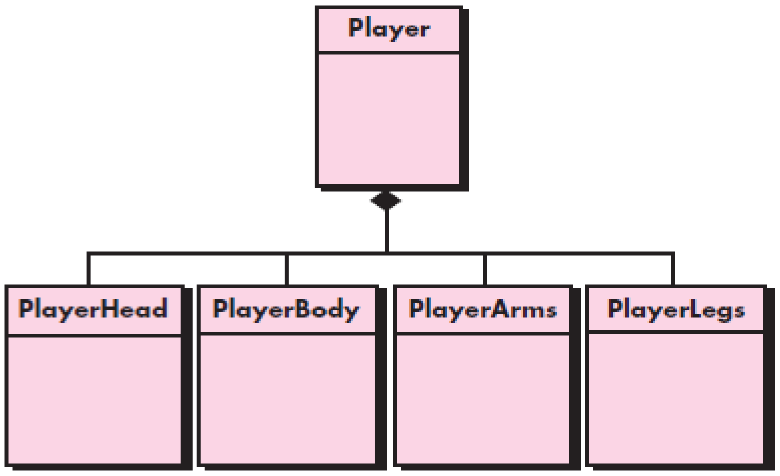


Fig: Composite Aggregate Class

The model uses following approach:

1. All participants are given a subset of CRC model index cards. Cards that collaborate should be separated.
2. All use -case scenarios and diagrams should be organized into categories.
3. Review leader reads the use-case deliberately. As the review leader comes to a named class, he/she passes a taken to the person holding corresponding class index card.
4. When token is passed, holder of class card is asked to describe responsibilities noted on the card. Group determines whether one of the responsibilities satisfies use-case required.
5. If responsibilities and collaboration noted on index cards cannot accommodate use-case, modifications are made to the cards. This may include definition of new classes or specification of new or revised responsibilities or collaborations on existing cards.

This modus operandi continues until the use-case is finished. When all use-cases have been reviewed, analysis, modeling continues.

Associations and Dependencies:

❖ **Association Relationship:** In many instances, two analysis classes are related to one another in some fashion, much like two data objects may be related to one another. In UML, these relationships are called "associations".

***Multiplicity*:** In an association relationship, multiplicity is used represent the cardinality (mapping) between two or more analysis classes. Ex: 0..*, 0..1, 1.. * etc.

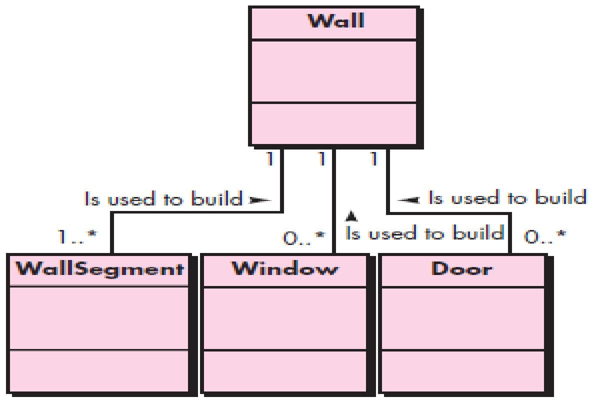


Fig: Association with Multiplicity

- **Dependency Relationship:** In many cases, a client-server relationship exists between two analysis classes, where client class depends on server class in some way. This establishes a dependency relationship. In UML, dependencies are defined by a Stereotype, which is an "extensibility mechanism" that allows a software engineer to define special modeling element.

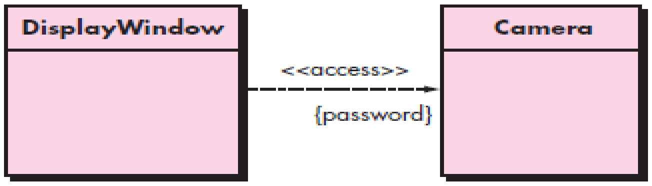


Fig: Dependency Relationship

ANALYSIS PACKAGES: Various elements of the analysis model (Ex: Use-cases, Analysis classes) are categorized in a manner that packages them as a grouping-called as an "Analysis Packages".

- ✓ The plus (+) sign preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages.
- ✓ Minus (-) sign indicates that an element is hidden from all other packages.
- ✓ A # sign indicates that an element is accessible only to classes contained within a given package.

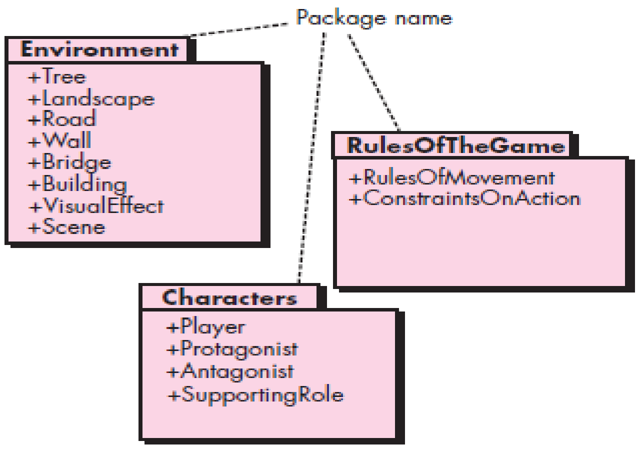


Fig: Analysis Packages

CREATING A BEHAVIORAL MODEL:

The behavioral model indicates how software will respond to external events. To create the model, analyst must perform the following steps:

1. Evaluate all use-cases to fully understand the sequence of interaction within the system.
2. Identify events that drive interaction sequence and understand how these events relate to specific classes.
3. Create a sequence for each use-case.
4. Build a state diagram for the system.
5. Review behavioral model to verify accuracy and consistency.

Identifying events with the use-case: The use-case represents a sequence of activities that involves actor and the system. In general, an event occurs whenever the system and an actor exchange information.

A use-case is examined for points of information exchange. An actor should be identified for each event. Information that is exchanged should be noted and any constraints should be listed. Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events or recognizing events.

STATE REPRESENTATIONS: In contest of behavioral modeling, the different characterizations of states must be considered:

1. The state of each class as the system performs its function.
2. The state of the system as observed from the outside as the system performs its function.

State of a class takes both passive and active characteristics. A passive state is simply current status of an object's attribute. Active state is the current status of an object as it undergoes a continuity transformation or processing.

❖ An event (sometimes called a trigger) must occur to force an object to make a transition from one state to other.

State Diagrams for Analysis Classes: One component of a behavioral modeling in a UML state diagram that represents active states for each class and the events that cause changes between these active states.

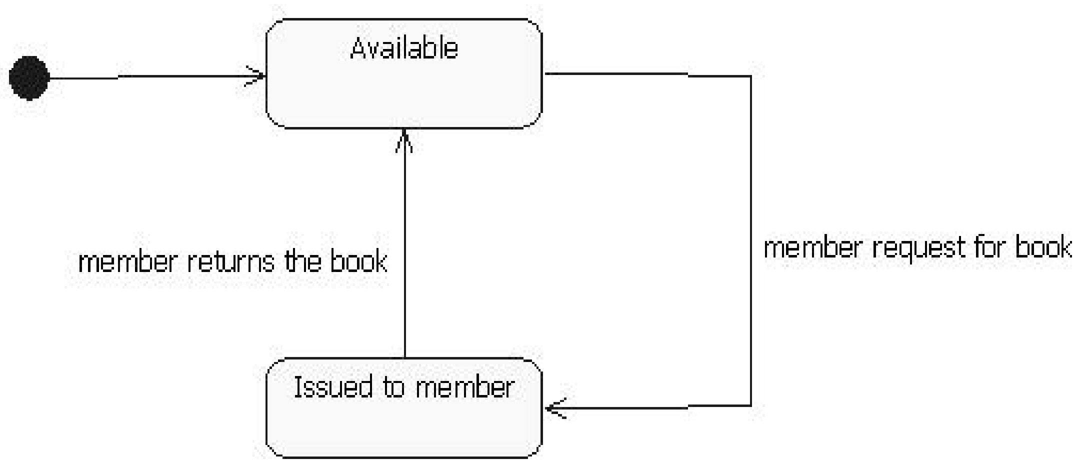


Fig: State Diagram

Each arrow shown in figure represents a transition from one active state of a class to another. The labels shown for each arrow represents the event that triggers the transition. Guard is a Boolean condition (1/0) that must be satisfied, for a transition to occur.

Along with specifying the event, analyst can also specify a guard and an action. An action occurs concurrently with state transition or as a consequence of it.

Sequence Diagrams for Analysis Classes: The sequence diagram indicates how events cause transitions from identified examining a use-case, modeler create sequence diagram.

- ❖ It is a shorthand version of use-case, represents key classes and events that cause behavior to flow from class to class.
- ❖ Once a complete sequence diagram has been developed, all of the events can be collaborate into a set of input and output events, useful to create effective design.

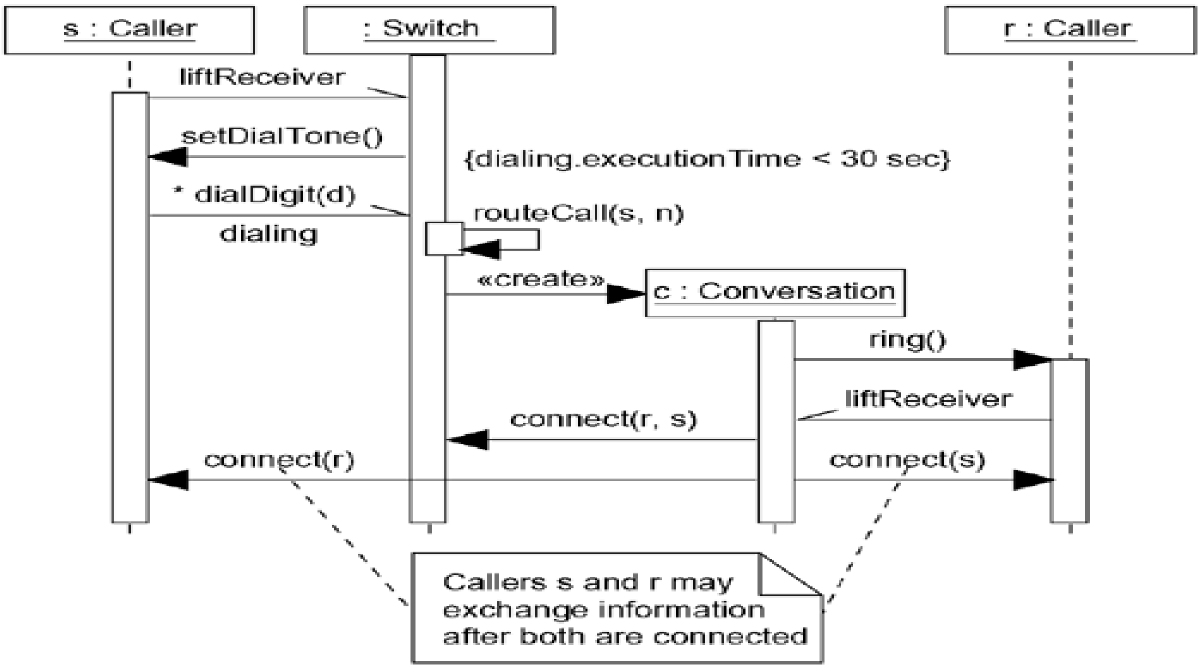


Fig: Sequence Diagram

DESIGN ENGINEERING

Design is a core engineering activity. Design creates a model of the software. Design engineering encompasses the set of principles, concepts and practices that lead to the development of a high quality system or product.

The goal of design engineering is to produce a model or representation that exhibits firmness, commodity and delight. Design engineering for computer software changes continually as new methods, better analysis and broader understanding evolve.

❖ A product should be ***designed*** in a ***flexible*** manner to develop quality software.

DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING:

Software design is the last software engineering action within modeling activity and sets the stage for development. The analysis model, manifested by scenario-based, class- based, flow-oriented and behavior elements, feed the design task.

Design model produces a data/class design, an architectural design, an interface design, a component design and a deployment design.

✓ The data/class design transforms the analysis class models into design class realizations and data structures require implementing the software. Part of class design may occur as each software component is designed.

✓ The architectural design defines the relationship between major structural elements of software, the architectural styles and design patterns and the constraints that affect the way in which architectural can be derived from system specifications, the analysis model and interaction of subsystems defined within analysis model.

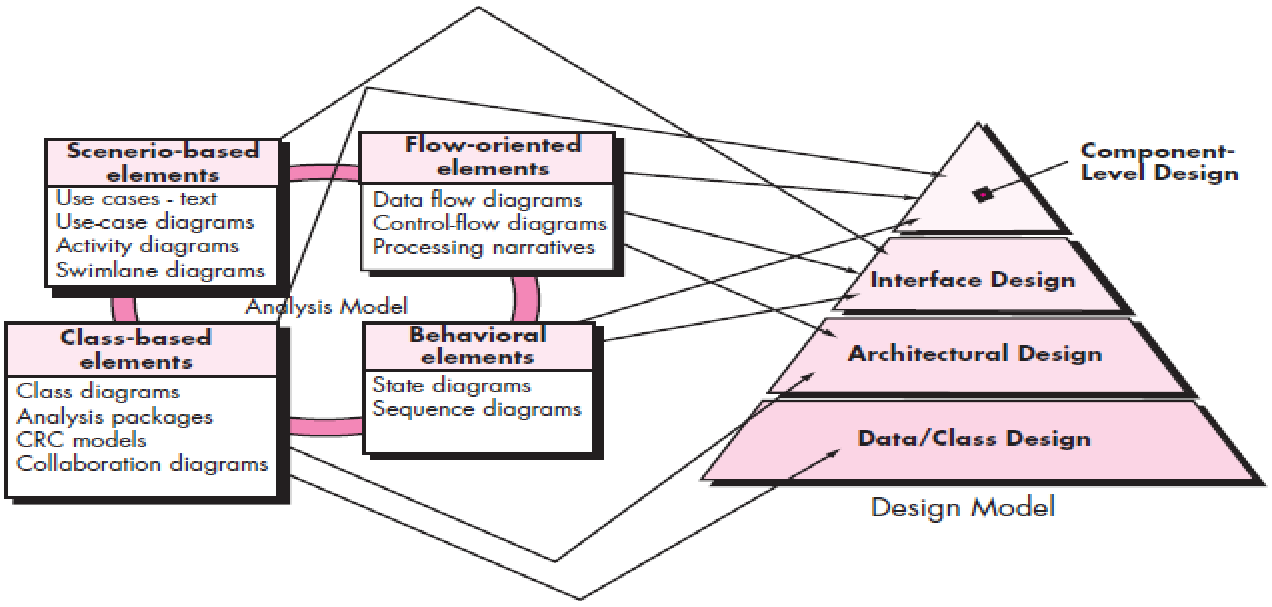


Fig: Translating the Requirements model to Design model

- ✓ The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. Usage scenarios and behavioral models provide much of information required for the interface design.
- ✓ The component-level design transforms structural element of the software architecture into a procedural description of software components. Information from class-based models, flow models, behavioral models serve as basis for component design.

DESIGN PROCESS AND DESIGN QUALITY:

Importance of software design can be stated with one word QUALITY. Software design serves as foundation for all software engineering and software support activities that follow.

Software design is an interactive process through which requirements are translated into a "blueprint" for constructing the software. Initially, design is represented at a high level of abstraction. As iteration occurs, subsequent refinement leads to design representations at lower levels of abstraction.

The following characteristics serve as a guide for evaluation of good design:

1. The design must implement all explicit requirements contained in analysis model, and accommodate all implicit requirements desired by customer.
2. Design must be a readable, understandable guide for those who generate code, who test and support the software.
3. Design should provide a complete picture of the software, addressing data, functional and behavioral domains.

- ✓ Each of these characteristics is goal of the design process.

Quality Guidelines: Guidelines for quality design are:

1. A design should exhibit an architecture that
   a) has been created using recognizable architectural styles/patterns.
   b) composed of components that exhibit good design characteristics.
   c) can be implemented in an evolutionary fashion.
2. A design should be modular; software should be logically partitioned into elements or sub-systems.
3. It should contain distinct representations of data, architecture, interfaces and components.
4. It should lead to data structures that are appropriate for the classes to be implemented.
5. It should lead to components that exhibit independent functional characteristics.
6. It should lead to interfaces that reduce complexity of connections between components and external environment.
7. It should be represented using a repeatable (iterative) method.
8. It should be represented using a notation that effectively communicates its meaning.

- ❖ Design engineering encourages good design through the application of fundamental design principles, systematic methodology and through review.

Quality Attributes: Hewlett-Packard (HP) developed a set software quality attributes, given by the acronym ***FURPS***:

1. ***Functionality:*** It is assessed by evaluating feature set and capabilities of the program, generality of functions and security of overall system.
2. ***Usability:*** It is assessed by considering human factors, overall aesthetics, consistency and documentation.
3. ***Reliability:*** It is evaluated by measuring frequency & severity of failure, ability to recover, accuracy of output results, Mean- Time-To-Failure (MTTF) and predictability of the program.
4. ***Performance:*** It is measured by processing speed, response time, resource consumption, throughout and efficiency.
5. ***Supportability:*** It combines the ability to extend the program (extensibility), adaptability, serviceability, which represent maintainability of the project.

These quality attributes must be considered as soon as design commences, but not after the design is complete and construction has begun.

DESIGN CONCEPTS

Fundamental software design concepts provide necessary framework for "getting it right".

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

- ✓ A *data abstraction* is a named collection of data that describes a data object.
- ✓ A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. Name of procedural abstraction implies these functions, but specific details are suppressed.

2. Architecture: It is the structure or organization of program components (modules), their interaction, and structure of data that are used by components. "Components can be generalized to represent major system elements & their interactions.

A set of architectural patterns enable a software engineer to reuse design level concepts. One goal of software design is to derive an architectural rendering of a system, which serves as a framework to conduct detailed design activities.

Architectural design can be represented using one or more of a number of different models:

1. <u>Structural models:</u> Represent architecture as collection of program components.
2. <u>Framework models:</u> Increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.
3. <u>Dynamic models:</u> Address behavioral change aspects of program architecture.
4. <u>Process models:</u> Focus on design of business or technical process that the system must accommodate.
5. <u>Functional models:</u> Can be used to represent functional hierarchy of a system.

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

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

i. Whether pattern is applicable to the current work.
ii. Whether pattern can be reused (hence, saving design time)
iii. Whether pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

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

Modularity is the single attribute of software that allows a program to the intellectually manageable (by breaking big process into modules). Modularity leads to a "divide and conquer" strategy, it's easier to solve a complex problem when you break it into manageable pieces, hence effort required to develop becomes negligibly small.

We modularize a design, so that development can be more easily planned, software increments can be defined and delivered, changes can be more easily accommodated, testing and debugging can be conducted more efficiently and long- term maintenance can be conducted without serious side effects.

5. **Information Hiding:** It suggests that "modules should be specified and designed so that information (algorithms, data) contained within a module is inaccessible to other modules that have no need for such information."

Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module. As most data and procedure are hidden from other parts of the software, errors during modification are less likely to propagate to other locations within the software.

6. **Functional Independence:** It is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
   Functional independence is a key to good design and design is the key to software quality. Independence is assessed using two qualitative criteria:
   - ❖ ***Cohesion:*** Cohesion is an indication of relative functional strength of a module. A cohesive module performs a single task, requiring little interaction with other components.
   - ❖ ***Coupling:*** Coupling is an indication of interconnection among modules in software architecture. Coupling depends on the interface complexity between modules.

7. **Refinement:** Stepwise refinement is a top-down design strategy, which is actually a process of elaboration. A program is developed by successively refining levels of procedural detail.

It defines/begins with a statement of function that is defined at a high level of abstraction. Refinement helps the designer to reveal low-level details as design progresses, thus in creating a complete design model.

8. **Refactoring:** "Refactoring is the process of changing a software system in such a way that it does not alter external behavior of the code (design) yet improves its internal structure".

When software is refactored, the existing design is examined for redundancy, unused design elements, poorly constructed data structures, unnecessary algorithms etc for better design.

9. **Design Classes:** As the design model evolves, the software team must define a set of design classes that:
    - ❖ Refine analysis classes by providing design detail that will enable the classes to be implemented.
    - ❖ Create a new set of design classes that implement a software infrastructure to support the business solution.

Design classes provide more technical detail as a guide for implementation. Five different types of design classes, each representing a different layer of design architecture are suggested. They are:

1. <u>*User Interface Classes:*</u> These define all abstractions that are necessary for Human Computer Interaction (HCI). HCI occurs within context of a metaphor (Ex: Order form, a checkbook) and design classes for interface may be visual representations of elements of metaphor.
2. <u>*Business Domain Classes:*</u> These are often refinements of the analysis classes defined earlier. The classes identify attributes and services (operations) that are required to implement some element of the business domain.
3. <u>*Process Classes:*</u> These implement lower-level business abstractions required to fully manage business domain classes.
4. <u>*Persistent Classes:*</u> These represent data stores (DBs) that will persist beyond execution of the software.
5. <u>*System Classes:*</u> These implement software management and control functions that enable system to operate and communicate. These are also known as supporting classes.

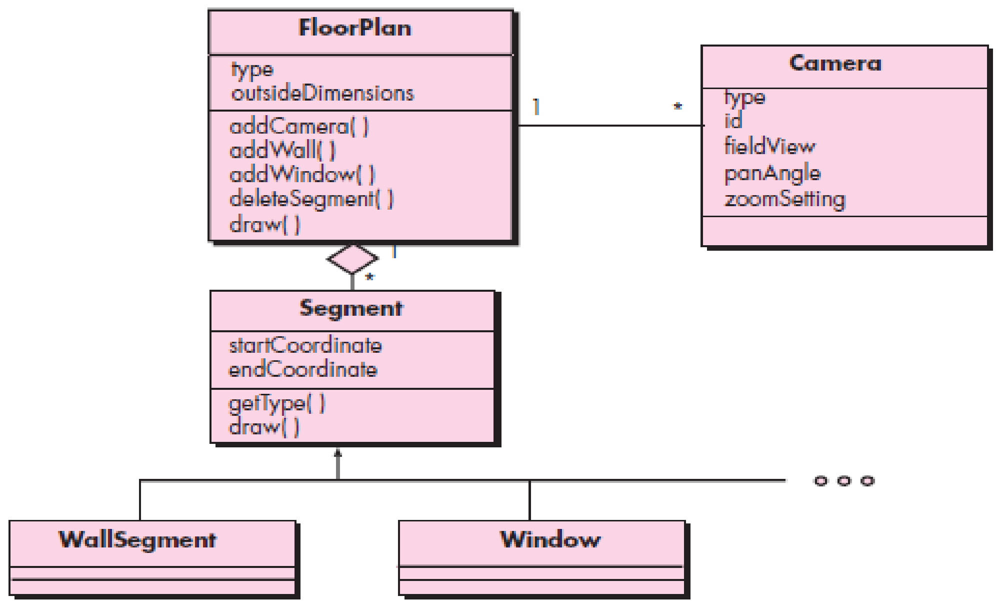
As design model evolves, software team must develop a complete set of attributes and operations for each design class.

Four characteristics of a well-formed design class:

1. ***Complete and Sufficient*****:** A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class.(No more and No less).
2. ***Primitiveness*****:** Methods associated with a design class should be focused on accomplishing one service the class. Once the service has been implemented, with a method, the class should not provide another way to accomplish same thing.
3. ***High Cohesion*****:** A cohesion design class has a small, focused set of responsibilities and single- mindedly applies attributes and methods to implement those responsibilities.
4. ***Low Coupling*****:** Collaboration between design classes should be kept to an acceptable minimum. If a design model is highly coupled, system is difficult to implement, test & maintain. So, design classes in a subsystem should have only limited knowledge of classes in other subsystems. It is also called as "*Law of Demeter*", suggests that a method should only send messages to methods in neighboring classes.

FIGURE 8.3

Design class for FloorPlan and composite aggregation for the class (see sidebar discussion)



THE DESIGN MODEL

The design model can be viewed in two different dimensions:

- ✓ The process dimension indicates evolution of design model as design tasks are executed as part of the software process.
- ✓ The abstraction dimension represents level of detail as each element of analysis model is transformed into a design equivalent and then refined iteratively.

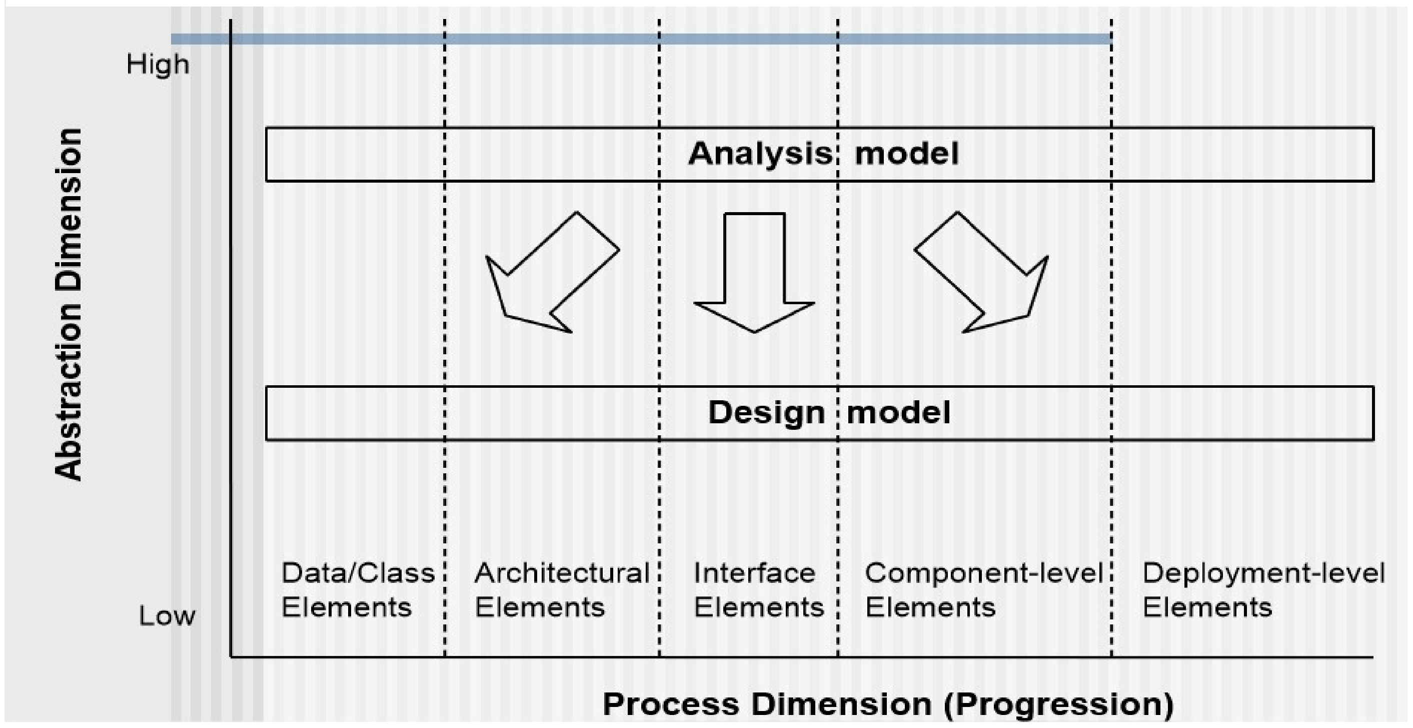


Fig: Dimensions of the Design Model

The elements of design model use many of same UML diagrams that were used in analysis model..The difference is that these diagrams are refined and elaborated as part of design, more implementation- specific detail is provided and emphasis is on architectural structure and style, components & interfaces.

Elements of design model:

1. **Data Design Elements:** Data design also sometimes referred as "Data Architecting". Data design creates a model of data and/or information that is represented at a high level of abstraction.

In many software applications, architecture of data will have a profound influence on architecture of software that must process it. Structure of data always plays important role in software design.

- At program component level, design of data structures and associated algorithms required to manipulate them is essential to the creation of high-quality applications.
- At application level, translation of data model into a DB is important to achieve business objectives.
- At business level, collection of information stored in DBs and reorganized into a "data warehouse" enables data mining or knowledge discovery.

2. **Architectural Design Elements:** These give us an overall view of the software. It is derived from 3 sources:

i. Information about application domain for software to be built.

ii. Specific analysis model elements such as DFDs or analysis classes, their relationships and collaborations for the problem.

iii. Availability of architectural patterns and styles.

3. **Interface Design Elements:** These tell how information flows into and out of the system and how it is communicated among components designed as part of the architecture. There are 3 important elements of interface design:

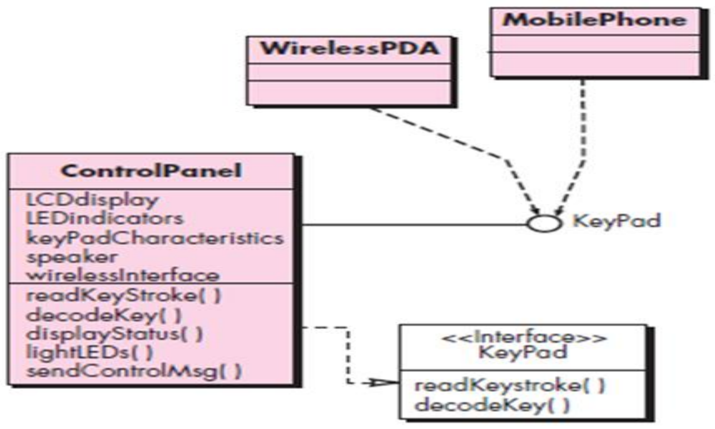
i. **User Interface (UI):** Design of a UI incorporates aesthetic elements (Ex: color, layout, graphics), ergonomic elements (information layout and placement, navigation), and technical elements (UI patterns, reusable components). In general, UI is a unique subsystem within overall application architecture.

ii. **External interfaces to other systems, devices, networks, other producers/consumers of information:** The design of external interfaces requires definitive information about the entity to which information is sent or received. In every case, this information should be collected during Requirement Engineering and verified. This design should incorporate error checking and appropriate security features.

iii. **Internal interfaces between various design components:** It is closely aligned with component level design. Design realizations of analysis classes represent all operations and messaging schemes required to enable communication and collaboration between operations in various classes.

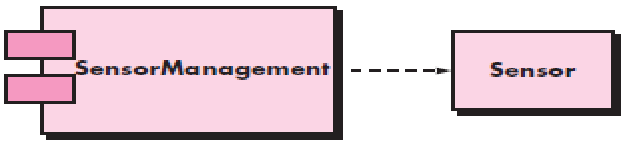
In some cases, an interface is modeled in same way as a class."An interface is a set of operations that describes some part of the behavior of a class and provides access to those operations."

Interface representation for Control-Panel



4. **Component-Level Design Elements:** Component level for software fully describes the internal detail of each software component. To accomplish this, component-level design defines detail for all processing that occurs within a component and an interface that allows access to all component operations.

A UML component diagram



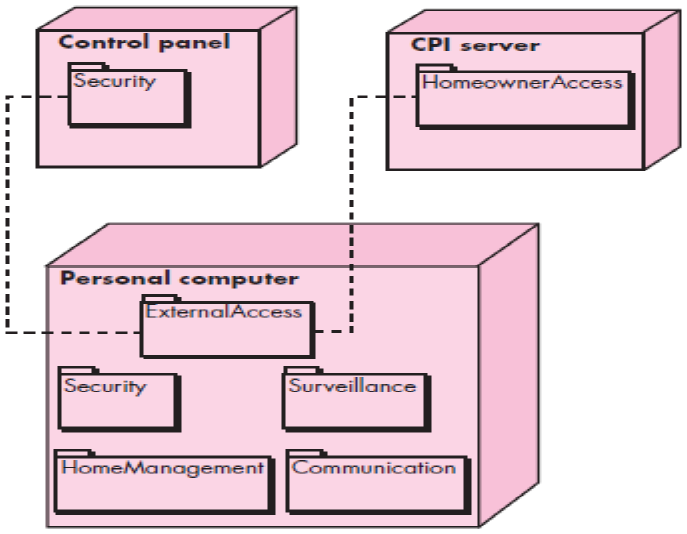
Design details of a component can be modeled at many different levels of abstraction. An activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudo code or some diagrammatic form.

5. **Deployment-Level Design Elements:** These indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

Deployment diagram shows the computing environment but does not explicitly indicate configuration details. Each instance of deployment is identified.

During design, a UML deployment diagram is first developed, and then refined. In a deployment diagram, each subsystem would be elaborated to indicate components that it implements.

A UML deployment diagram



PATTERN-BASED SOFTWARE DESIGN

Throughout the design process, a software engineer should look for every opportunity to reuse existing design patterns (when they meet needs of the design) rather than creating new ones.

Describing a Design Pattern: Mature engineering disciplines make use of thousands of design patterns, for things such as buildings, highways, electrical circuits, factories, weapons, computers etc. A description of design pattern may also consider a set of design forces.

Design forces describe non-functional requirements (Ex: Portability) associated the software for which the pattern is to be applied. These also define the constraints that may restrict the manner in which design is to be implemented. Design forces describe the environment and constraints that must exist to make design pattern applicable.

- ✓ Pattern characteristics (classes, responsibilities & collaborations) indicate the attributes of the design that may be adjusted to enable the pattern to accommodate a variety of problems.

The names of design patterns should be chosen with care and should have a meaningful name.

Using Patterns in Design: Design patterns can be used throughout software design. The problem description is examined at various levels of abstraction to determine if it is amenable to one or more following types of patterns:

1. **Architectural Patterns:** These patterns,

- ✓ Define overall structure of the software,
- ✓ Indicate relationships among subsystems & software components,
- ✓ Define rules for specifying relationships among the elements (class, components, packages, subsystems) of the architecture.

2. **Design Patterns:** These patterns address a specific element of the design such as an aggregation of components to solve some design problem, relationships among components, or mechanisms for effecting component-to component communication.
3. **Coding Patterns:** These are also called idioms; these language- specific patterns generally implement an algorithmic element of a component, a specific interface protocol, or a mechanism for communication among components.

Each of these pattern types differs in the level of abstraction and degree to which it provides direct guidance for construction activity of software process.

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."

Plug points enable designer to integrate problem specific classes or functionality within the skeleton. In O-O context, framework is collection of co-operating classes.

- ✓ To be most effective, frameworks are applied with no changes, additional design elements may be added, but only via plug points that allow designer to flesh out the framework selection.