

UNIT- III

Requirements Analysis, Scenario-Based Modeling, UML Models That Supplement the Use Case, Data Modeling Concepts, Class-Based Modeling, Requirements Modeling Strategies, Flow-Oriented Modeling, creating a Behavioral Model, Patterns for Requirements Modelling, Requirements Modeling for WebApps

REQUIREMENTS MODELING: SCENARIOS, INFORMATION, AND ANALYSIS CLASSES

REQUIREMENTS ANALYSIS

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering.

The requirements modeling action results in one or more of the following types of models:

- ***Scenario-based models*** of requirements from the point of view of various system “actors”
- ***Data models*** that depict the information domain for the problem
- ***Class-oriented models*** that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements
- ***Flow-oriented models*** that represent the functional elements of the system and how they transform data as it moves through the system
- ***Behavioral models*** that depict how the software behaves as a consequence of external “events”

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model provides the developer and the customer with the means to assess quality once software is built.

Throughout requirements modeling, primary focus is on ***what, not how***. What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined, and what constraints apply?

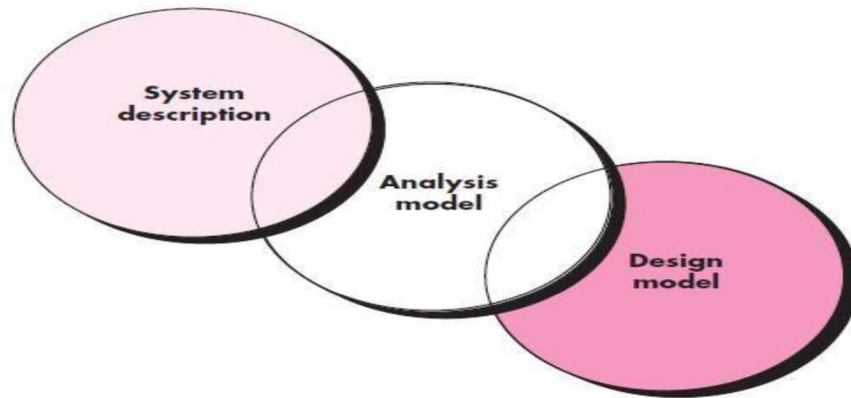


Fig : The requirements model as a bridge between the system description and the design model

The requirements model must achieve three primary objectives:

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

The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design that describes the software's application architecture, user interface, and component-level structure.

Analysis Rules of Thumb

Arlow and Neustadt suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

- *The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.*
- *Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.*
- *Delay consideration of infrastructure and other nonfunctional models until design.*

That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.

- ***Minimize coupling throughout the system.*** It is important to represent relationships between classes and functions. However, if the level of “interconnectedness” is extremely high, effort should be made to reduce it.
- ***Be certain that the requirements model provides value to all stakeholders.*** Each constituency has its own use for the model
- ***Keep the model as simple as it can be.*** Don’t create additional diagrams when they add no new information. Don’t use complex notational forms, when a simple list will do.

Domain Analysis

Domain analysis doesn’t look at a specific application, but rather at the domain in which the application resides.

The “specific application domain” can range from avionics to banking, from multimedia video games to software embedded within medical devices. The goal of domain analysis is straightforward: to identify common problem solving elements that are applicable to all applications within the domain, to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.

Requirements Modeling Approaches

One view of requirements modeling, called ***structured analysis***, considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their *attributes and relationships*.

A second approach to analysis modeling, called ***object-oriented analysis***, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process are predominantly object oriented.

Each element of the requirements model is represented in following figure presents the problem from a different point of view.

Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships between the objects, and the collaborations that occur between the classes that are defined.

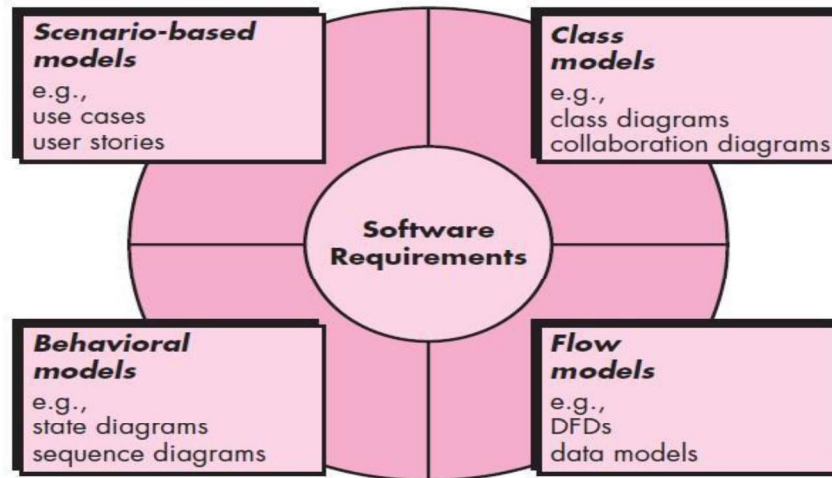


Fig : Elements of the analysis model

Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally,

Flow-oriented elements represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

SCENARIO-BASED MODELING

Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

Creating a Preliminary Use Case

Alistair Cockburn characterizes a use case as a “contract for behavior”, the “contract” defines the way in which an actor uses a computer-based system to accomplish some goal. In essence, a use case captures the interactions that occur between producers and consumers of information and the system itself.

A use case describes a specific usage scenario in straightforward language from the point of view of a defined actor. These are the questions that must be answered if use cases are to provide value as a requirements modeling tool. (1) what to write about, (2) how much to write about it, (3) how detailed to make your description, and (4) how to organize the description?

To begin developing a set of use cases, list the functions or activities performed by a specific actor.

Refining a Preliminary Use Case

Each step in the primary scenario is evaluated by asking the following questions:

- *Can the actor take some other action at this point?*
- *Is it possible that the actor will encounter some error condition at this point? If so, what might it be?*
- *Is it possible that the actor will encounter some other behavior at this point (e.g., behavior that is invoked by some event outside the actor's control)? If so, what might it be?*

Cockburn recommends using a “brainstorming” session to derive a reasonably complete set of exceptions for each use case. In addition to the **three** generic questions suggested earlier in this section, the following issues should also be explored:

- *Are there cases in which some “validation function” occurs during this use case?* This implies that validation function is invoked and a potential error condition might occur.
- *Are there cases in which a supporting function (or actor) will fail to respond appropriately?* For example, a user action awaits a response but the function that is to respond times out.
- *Can poor system performance result in unexpected or improper user actions?* For example, a Web-based interface responds too slowly, resulting in a user making multiple selects on a processing button. These selects queue inappropriately and ultimately generate an error condition.

Writing a Formal Use Case

The typical outline for formal use cases can be in following manner

- The **goal in context** identifies the overall scope of the use case.
 - The **precondition** describes what is known to be true before the use case is initiated.
 - The **trigger** identifies the event or condition that “gets the use case started”
 - The **scenario** lists the specific actions that are required by the actor and the appropriate system responses.
 - **Exceptions** identify the situations uncovered as the preliminary use case is refined
- Additional headings may or may not be included and are reasonably self-explanatory.

Every modeling notation has limitations, and the use case is no exception. A use case focuses on functional and behavioral requirements and is generally inappropriate for nonfunctional requirements

However, scenario-based modeling is appropriate for a significant majority of all situations that you will encounter as a software engineer.

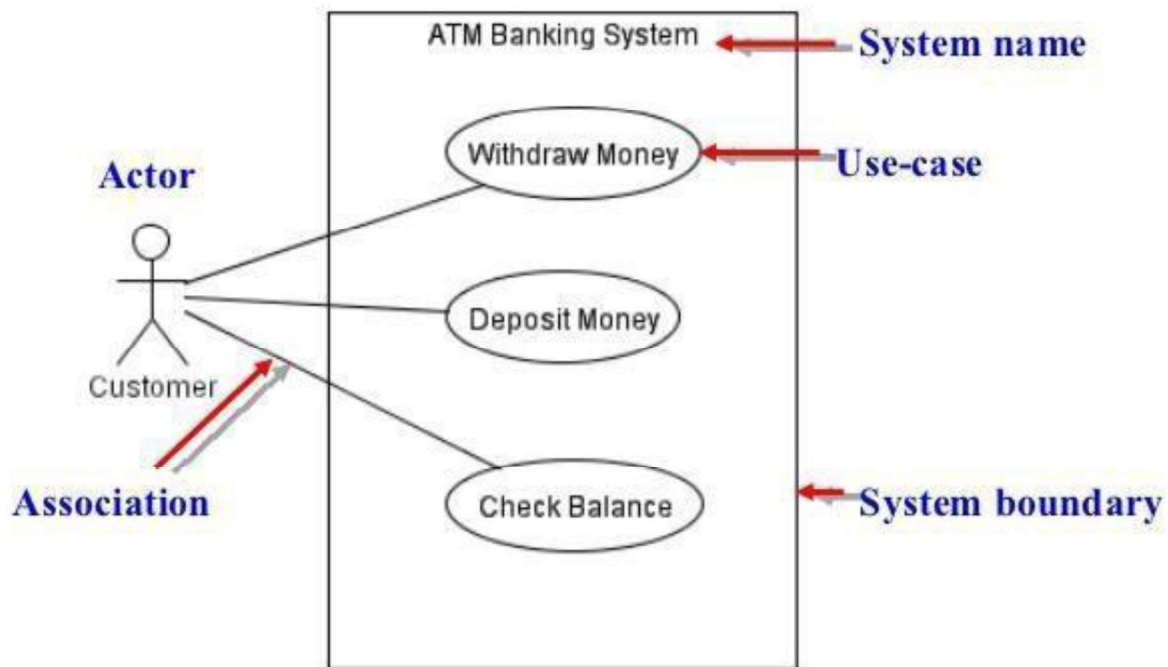


Fig : Simple Use Case Diagram

UML MODELS THAT SUPPLEMENT THE USE CASE

Developing an Activity Diagram

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart, an activity diagram uses rounded rectangles to imply a specific system function, arrows to represent flow through the system, decision diamonds to depict a branching decision (each arrow emanating from the diamond is labeled), and solid horizontal lines to indicate that parallel activities are occurring. i.e A UML activity diagram represents the actions and decisions that occur as some function is performed.

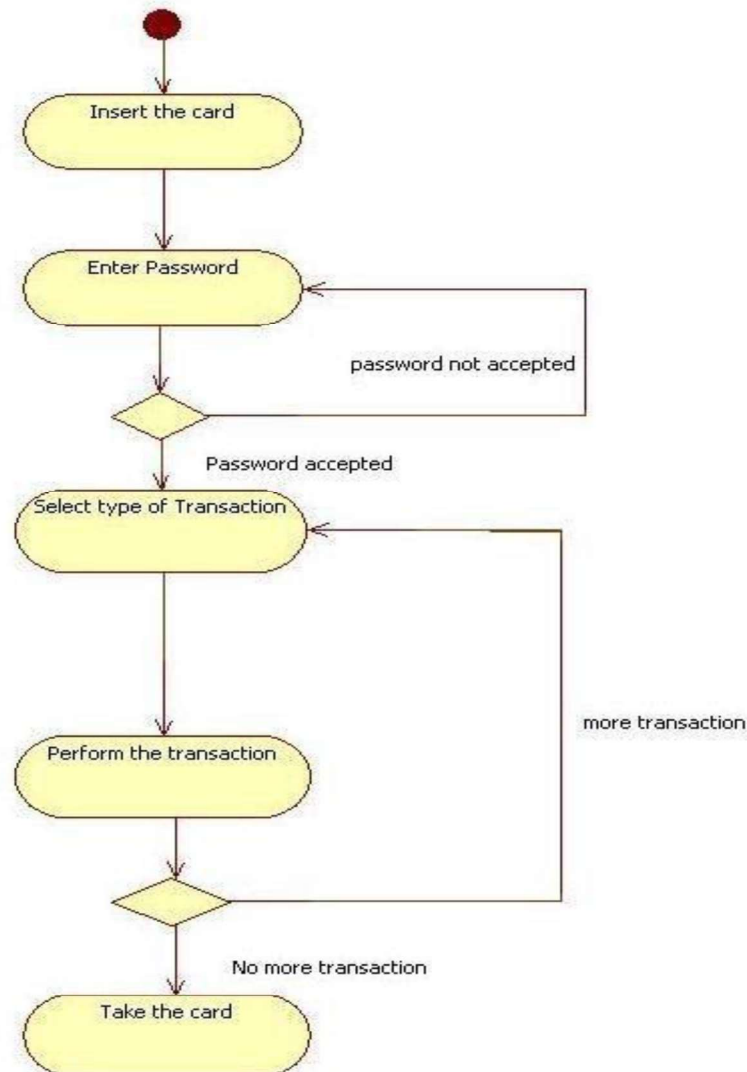


Fig : Activity Diagram for ATM

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. Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

The following figure represents *swimlane diagram for ATM*

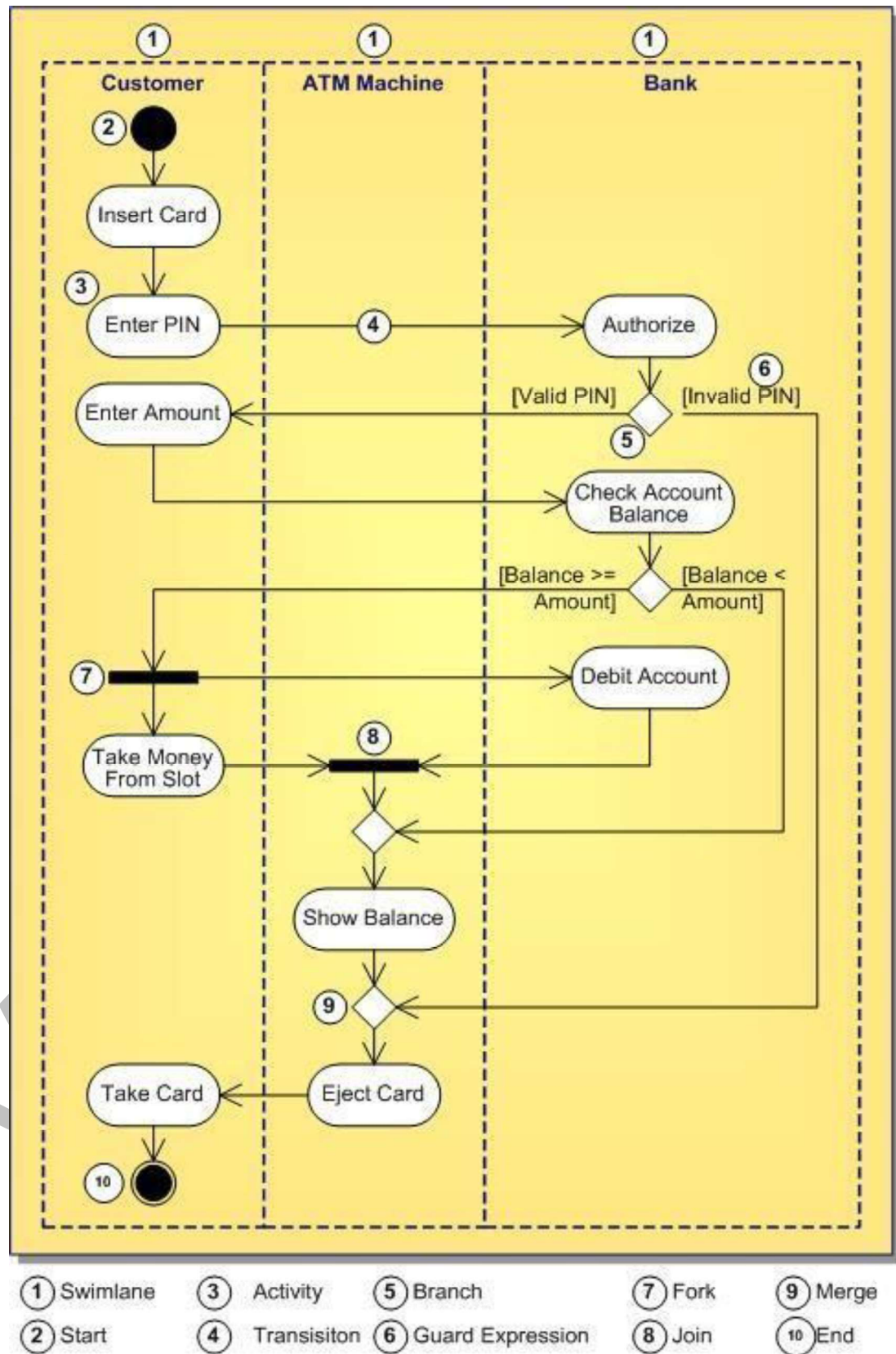


Fig : swimlane diagram for ATM

DATA MODELING CONCEPTS

Data modeling is the process of documenting a complex software system design as an easily understood diagram, using text and symbols to represent the way data needs to flow. The diagram can be used as a blueprint for the construction of new software or for re-engineering a legacy application. The most widely used data Model by the Software engineers is **Entity-Relationship Diagram (ERD)**, it addresses the issues and represents all data objects that are entered, stored, transformed, and produced within an application.

Data Objects

A *data object* is a representation of composite information that must be understood by software. 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).

For example, a **person** or a **car** can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. 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. Therefore, the data object can be represented as a table as shown in following table. The headings in the table reflect attributes of the object.

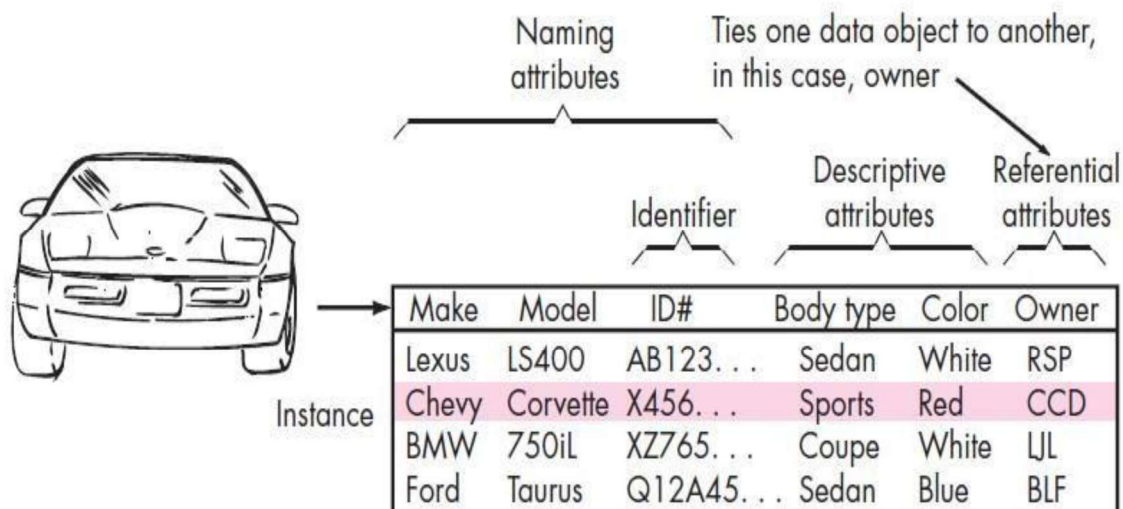


Fig : Tabular representation of data objects

Data Attributes

Data attributes define the properties of a data object and take on one of **three** different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table.

Relationships

Data objects are connected to one another in different ways. Consider the two data objects, **person** and **car**. These objects can be represented using the following simple notation and relationships are 1) A person *owns* a car, 2) A person *is insured to drive* a car

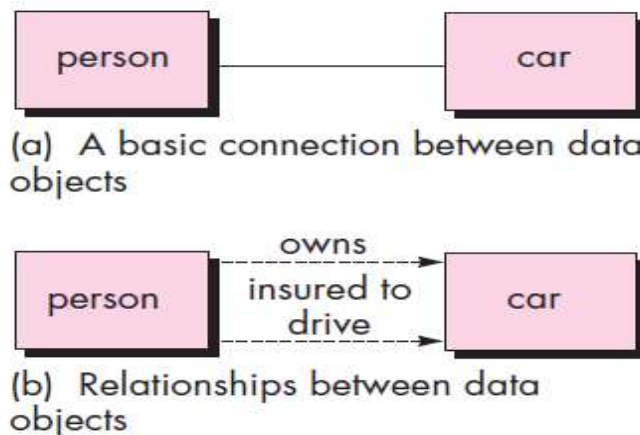


Fig : Relationships between data objects

CLASS-BASED MODELING

Class-based modeling represents the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships between the objects, and the collaborations that occur between the classes that are defined. 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

We can begin to identify classes by examining the usage scenarios developed as part of the requirements model and performing a “**grammatical parse**” on the use cases developed for the system to be built.

Analysis classes manifest themselves in one of the following ways:

- **External entities** (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- **Things** (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
- **Occurrences or events** (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- **Roles** (e.g., manager, engineer, salesperson) played by people who interact with the system.
- **Organizational units** (e.g., division, group, team) that are relevant to an application.
- **Places** (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- **Structures** (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

Coad and Yourdon suggest **six** selection characteristics that should be used as you consider each potential class for inclusion in the **analysis model**:

1. **Retained information.** The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
2. **Needed services.** The potential class must have a set of identifiable operations that can change the value of its attributes in some way.
3. **Multiple attributes.** During requirement analysis, the focus should be on “major” information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
4. **Common attributes.** A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.
5. **Common operations.** A set of operations can be defined for the potential class and these operations apply to all instances of the class.
6. **Essential requirements.** External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

.2 Specifying Attributes

Attributes describe a class that has been selected for inclusion in the requirements model. In essence, it is the attributes that define the class—that clarify what is meant by the class in the context of the problem space.

To develop a meaningful set of attributes for an analysis class, you should study each use case and select those “things” that reasonably “belong” to the class.

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 (e.g., adding, deleting, reformatting, selecting), (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.

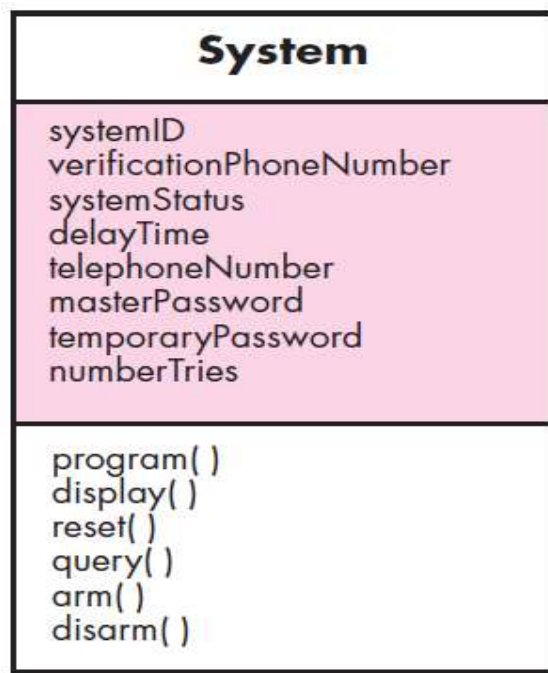


Fig : Class diagram for the system class

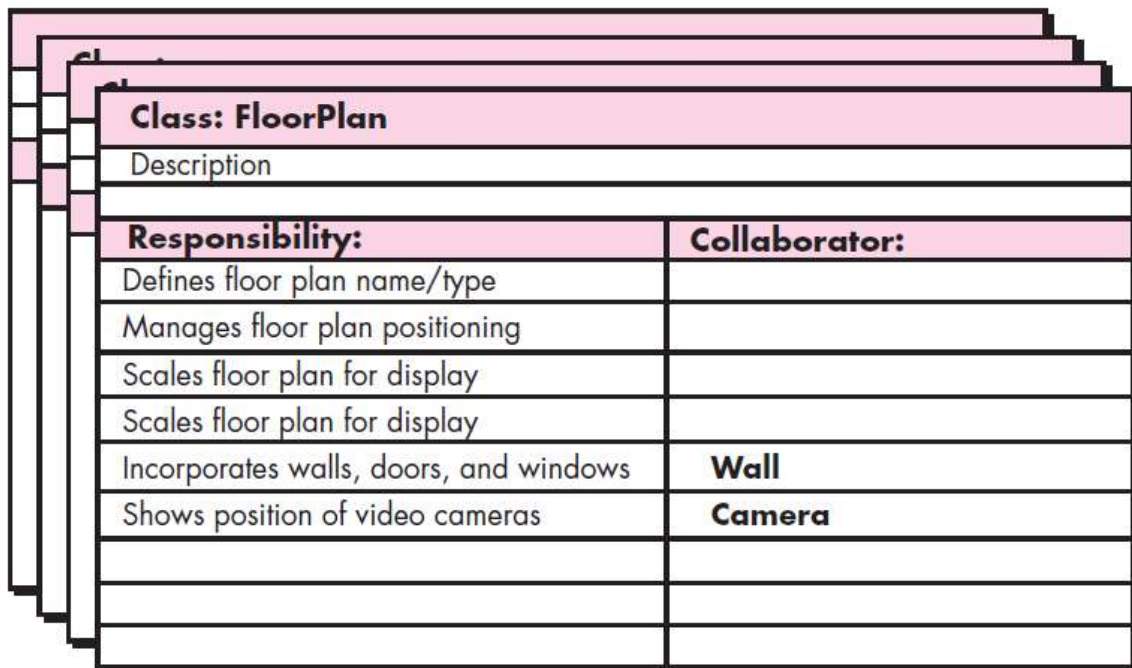
Class-Responsibility-Collaborator (CRC) Modeling

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

Ambler describes CRC modeling in the following way :

A CRC model is really a collection of standard **index cards** that represent classes. The cards are divided into **three** sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the **left** and the collaborators on the **right**.

The CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. **Responsibilities** are the attributes and operations that are relevant for the class. i.e., a responsibility is “anything the class knows or does” **Collaborators** are those classes that are required to provide a class with the information needed to complete a responsibility. In general, a *collaboration* implies either a request for information or a request for some action. A simple CRC index card is illustrated in following figure.



| 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 |
| | |
| | |
| | |

Fig : A CRC model index card

Classes : The taxonomy of class types can be extended by considering the following categories:

- **Entity classes**, also called **model or business** classes, are extracted directly from the statement of the problem. These classes typically 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 that the user sees and interacts with as the software is used. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
- **Controller classes** manage a “unit of work” from start to finish. That is, controller classes can be designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, (4) validation of data communicated between objects or between the user and the application. In general, controller classes are not considered until the design activity has begun.

Responsibilities : Wirfs-Brock and her colleagues suggest five guidelines for allocating responsibilities to classes:

1. **System intelligence should be distributed across classes to best address the needs of the problem.** Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do.
2. **Each responsibility should be stated as generally as possible.** This guideline implies that general responsibilities should reside high in the class hierarchy
3. **Information and the behavior related to it should reside within the same class.** This achieves the object-oriented principle called *encapsulation*. Data and the processes that manipulate the data should be packaged as a cohesive unit.
4. **Information about one thing should be localized with a single class, not distributed across multiple classes.** A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.
5. **Responsibilities should be shared among related classes, when appropriate.** There are many cases in which a variety of related objects must all exhibit the same behavior at the same time.

Collaborations. Classes fulfill their responsibilities in one of **two** ways:

1. A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
2. A class can collaborate with other classes.

When a complete CRC model has been developed, stakeholders can review the model using the following approach :

1. All participants in the review (of the CRC model) are given a subset of the CRC model index cards. Cards that collaborate should be separated (i.e., no reviewer should have two cards that collaborate).
2. All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
3. The review leader reads the use case deliberately. As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card.
4. When the token is passed, the holder of the card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use case, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

Associations and Dependencies

An **association** defines a relationship between classes. An association may be further defined by indicating **multiplicity**. **Multiplicity** defines how many of one class are related to how many of another class.

A client-server relationship exists between two analysis classes. In such cases, a client class depends on the server class in some way and a **dependency relationship** is established. Dependencies are defined by a **stereotype**. A **stereotype** is an “**extensibility mechanism**” within UML that allows you to define a special modeling element whose semantics are custom defined. In UML. Stereotypes are represented in double angle brackets (e.g., <<**stereotype**>>).

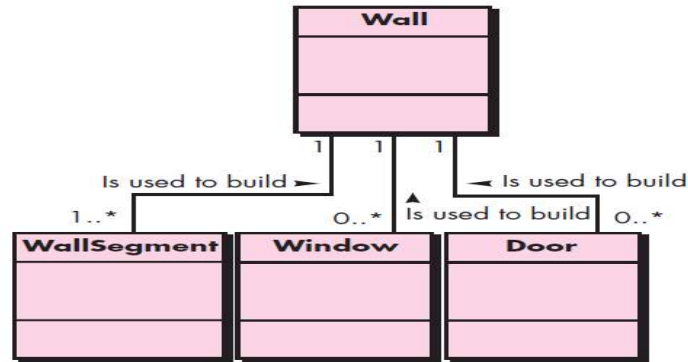


Fig : Multiplicity

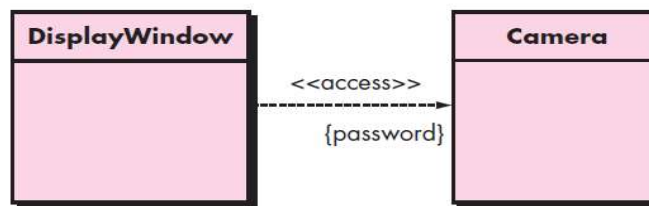


Fig : Dependencies

Analysis Packages

An important part of analysis modeling is categorization. That is, various elements of the analysis model (e.g., use cases, analysis classes) are categorized in a manner that packages them as a grouping—called an *analysis package*—that is given a representative name.

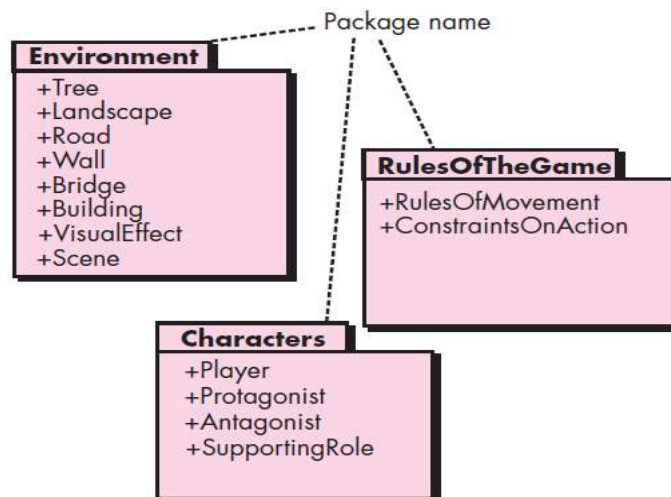


Fig : Packages

Requirements Modeling (Flow, Behavior, Patterns and WEBAPPS)

REQUIREMENTS MODELING STRATEGIES

One view of requirements modeling, called *structured analysis*,. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

A second approach to analysis modeled, called *object-oriented analysis*, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements.

FLOW-ORIENTED MODELING

Flow-oriented modeling is perceived as an outdated technique by some software engineers, it continues to be one of the most widely used requirements analysis notations in use today. The *data flow diagram (DFD)* is the representation of Flow-oriented modeling. **The purpose of data flow diagrams is to provide a semantic bridge between users and systems developers.”**

The DFD takes an input-process-output view of a system. That is, data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software. Data objects are represented by labeled **arrows**, and transformations are represented by **circles (also called bubbles)**. The DFD is presented in a hierarchical fashion. That is, the first data flow model (sometimes called a level **0 DFD** or *context diagram*) represents the system as a whole. Subsequent data flow diagrams refine the context diagram, providing increasing detail with each subsequent level.

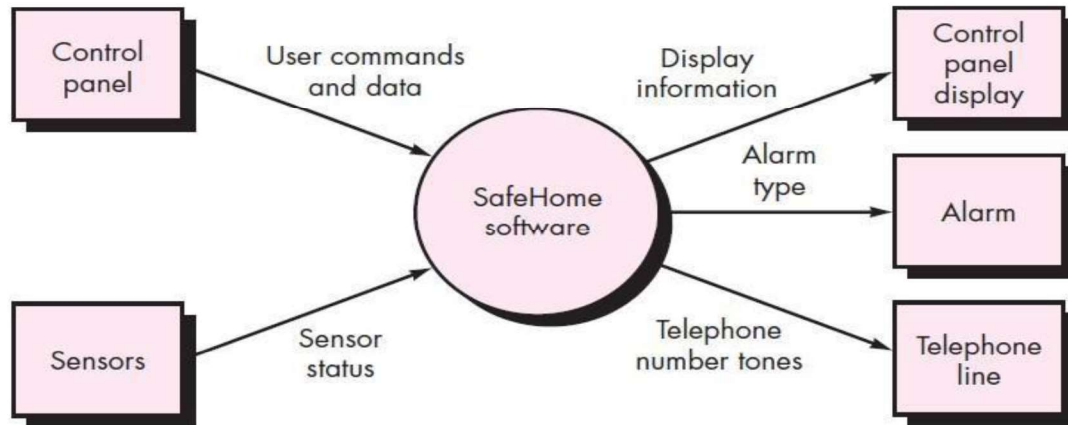


Fig : Context-level DFD for the Safe Home security function

Creating a Data Flow Model

The data flow diagram enables you to develop models of the information domain and functional domain. As the DFD is refined into greater levels of detail, you perform an implicit functional decomposition of the system. At the same time, the DFD refinement results in a corresponding refinement of data as it moves through the processes that embody the application.

A few simple guidelines can aid immeasurably during the derivation of a data flow diagram:

- (1) The level 0 data flow diagram should depict the 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 the next level;
- (4) All arrows and bubbles should be labeled with meaningful names;
- (5) *Information flow continuity* must be maintained from level to level,2 and
- (6) One bubble at a time should be refined. There is a natural tendency to overcomplicate the data flow diagram.

A **level 0** DFD for the security function is shown in above figure. The primary **external entities (boxes)** produce information for use by the system and consume information generated by the system. The labeled arrows represent data objects or data object hierarchies.

The **level 0** DFD must now be expanded into a **level 1** data flow model. you should apply a “grammatical parse” to the use case narrative that describes the context-level bubble. That is, isolate all nouns (and noun phrases) and verbs (and verb phrases). *The grammatical parse is not*

foolproof, but it can provide you with an excellent jump start, if you're struggling to define data objects and the transforms that operate on them.

The processes represented at **DFD level 1** can be further refined into **lower levels**. The refinement of DFDs continues until each bubble performs a simple function. That is, until the process represented by the bubble performs a function that would be easily implemented as a program component. a concept, **Cohesion** can be used to assess the processing focus of a given function. i.e refine DFDs until each bubble is “**single-minded.**”

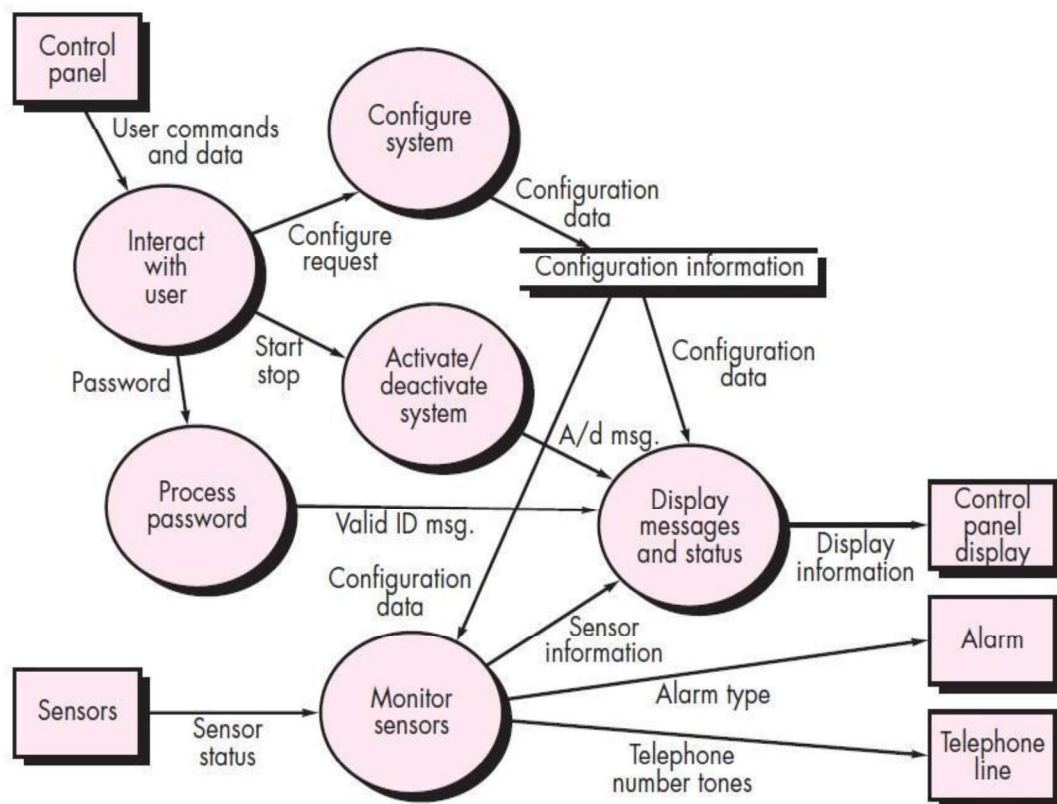


Fig: Level 1 DFD for SafeHome security function

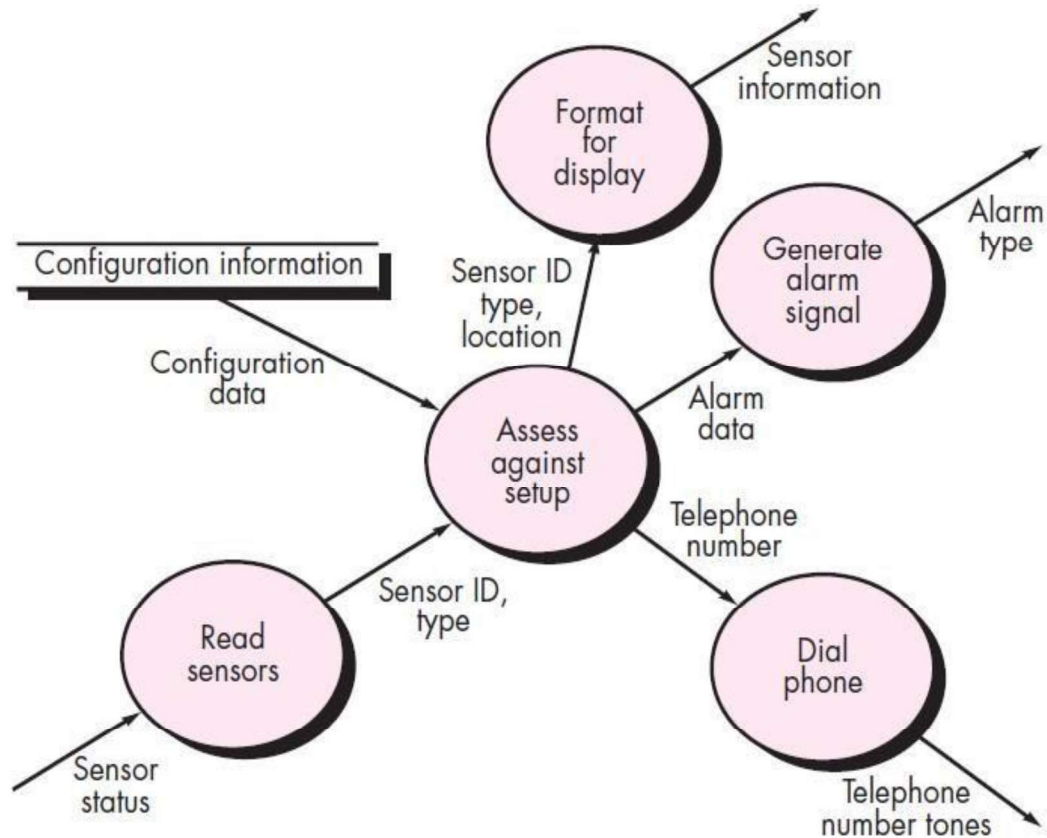


Fig : Level 2 DFD that refines the monitor sensors process

2.14.2. Creating a Control Flow Model

The data model and the data flow diagram are all that is necessary to obtain meaningful insight into software requirements. The following guidelines are suggested for creating a Control Flow Model

- List all sensors that are “read” by the software.
- List all interrupt conditions.
- List all “switches” that are actuated by an operator.
- List all data conditions.
- Recalling the noun/verb parse that was applied to the processing narrative, review all “control items” as possible control specification inputs/outputs.
- Describe the behavior of a system by identifying its states, identify how each state is reached, and define the transitions between states.
- Focus on possible omissions—a very common error in specifying control;

The Control Specification

A *control specification* (CSPEC) represents the behavior of the system in **two** different ways. The CSPEC contains a state diagram that is a sequential specification of behavior. It can also contain a program activation table—a combinatorial specification of behavior. The following figure depicts a preliminary state diagram for the level 1 control flow model for *SafeHome*. The diagram indicates how the system responds to events as it traverses the four states defined at this level. By reviewing the state diagram, we can determine the behavior of the system and, more important, ascertain whether there are “holes” in the specified behavior.

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

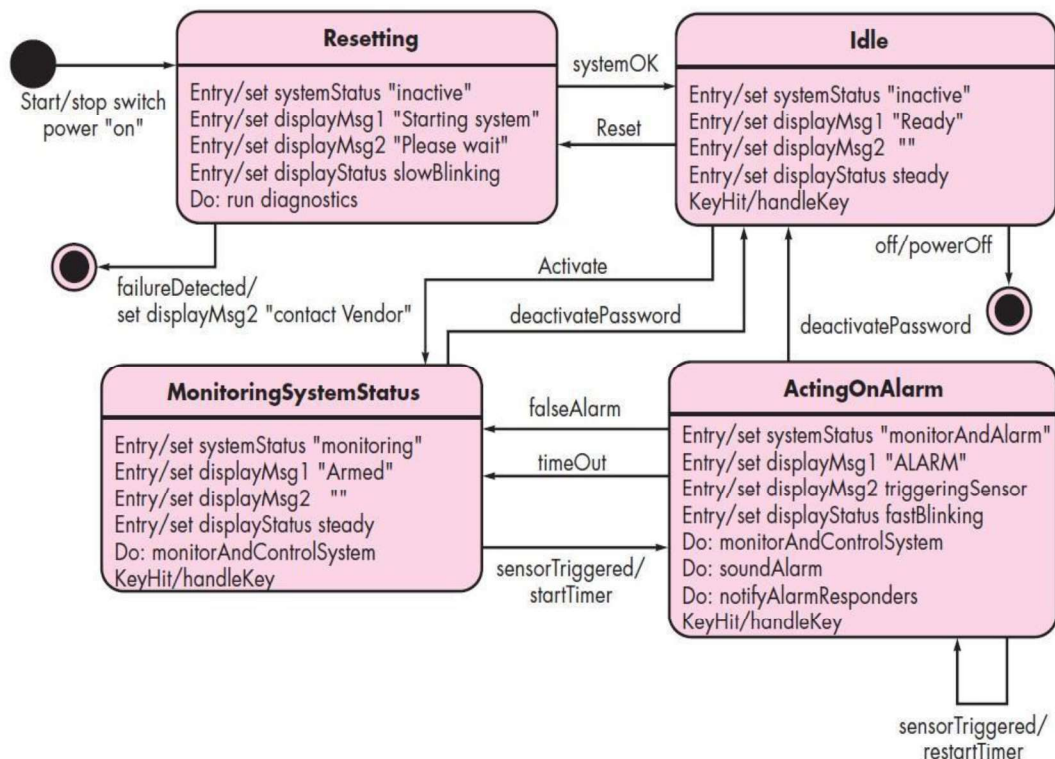


Fig : State diagram for SafeHome security function

The Process Specification

The *process specification* (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. The content of the process specification can include narrative text, a program design language (PDL) description of the process algorithm,

mathematical equations, tables, or UML activity diagrams. By providing a PSPEC to accompany each bubble in the flow model, you can create a “mini-spec” that serves as a guide for design of the software component that will implement the bubble.

CREATING A BEHAVIORAL MODEL

The *behavioral model* indicates how software will respond to external events or stimuli. To create the model, you should perform the following steps:

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

Identifying Events with the Use Case

The use case represents a sequence of activities that involves actors 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. To illustrate, we reconsider the use case for a portion of the *SafeHome* security function. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

The underlined portions of the use case scenario indicate events. An actor should be identified for each event; the information that is exchanged should be noted, and any conditions or constraints should be listed. Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events .

State Representations

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its Function **Two** different behavioral representations are discussed in the paragraphs that follow. The **first** indicates how

an individual class changes state based on external events and the **second** shows the behavior of the software as a function of time.

State diagrams for analysis classes. One component of a behavioral model is a UML state diagram that represents active states for each class and the events (triggers) that cause changes between these active states. The following figure illustrates a state diagram for the **ControlPanel** object in the *SafeHome* security function. Each arrow shown in figure represents a transition from one active state of an object to another. The labels shown for each arrow represent the event that triggers the transition

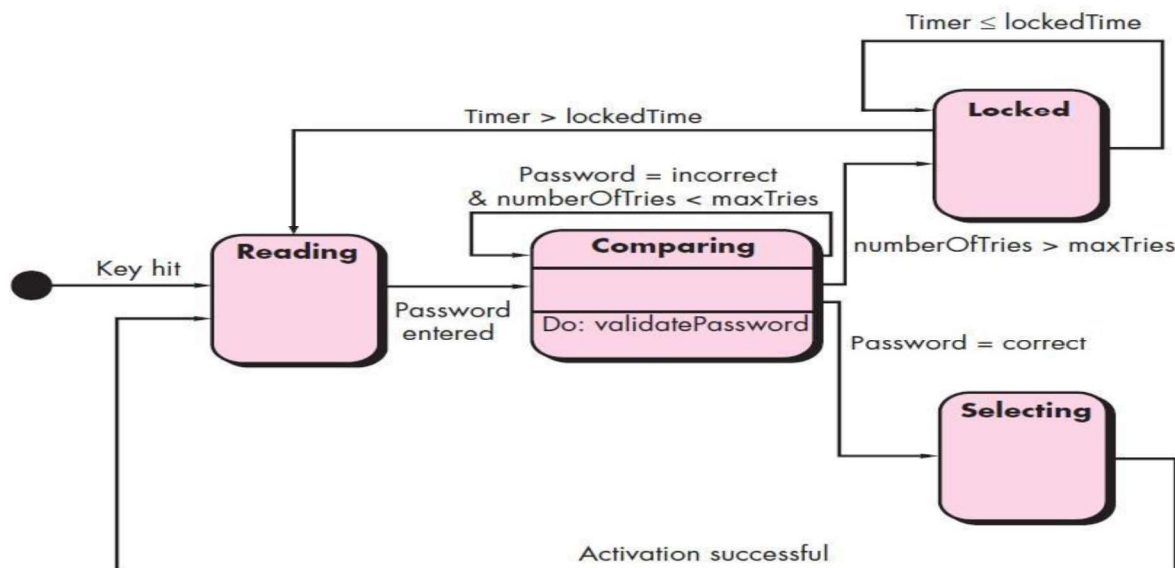


Fig : State diagram for the Control Panel class

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. In essence, the sequence diagram is a shorthand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.

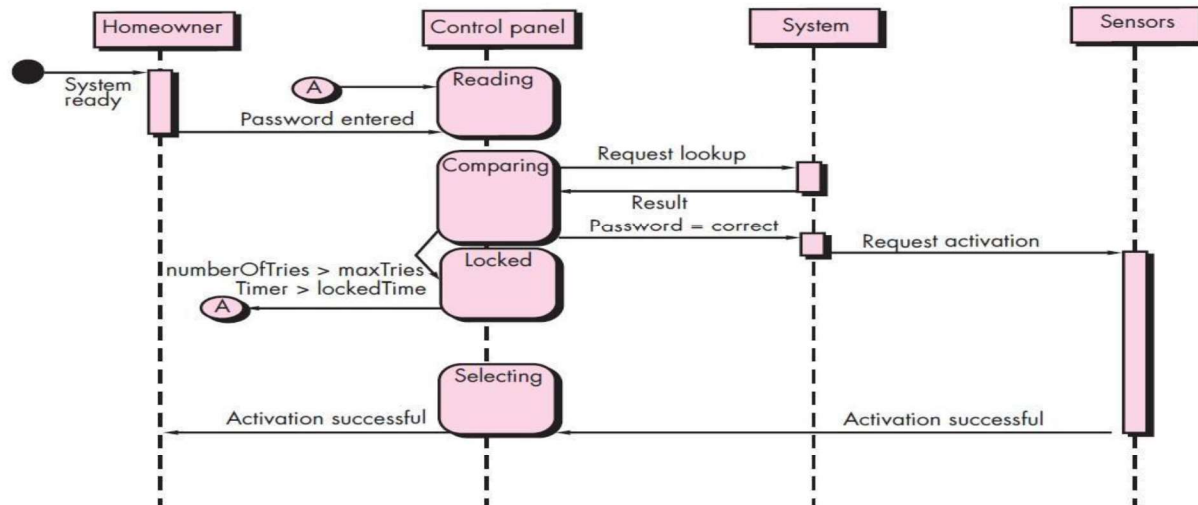


Fig : Sequence diagram (partial) for the *SafeHome* security function

PATTERNS FOR REQUIREMENTS MODELING

Software patterns are a mechanism for capturing domain knowledge in a way that allows it to be reapplied when a new problem is encountered. In some cases, the domain knowledge is applied to a new problem within the same application domain. The domain knowledge captured by a pattern can be applied by analogy to a completely different application domain.

The pattern can be reused when performing requirements modeling for an application within a domain. Analysis patterns are stored in a repository so that members of the software team can use search facilities to find and reuse them. Once an appropriate pattern is selected, it is integrated into the requirements model by reference to the pattern name.

Discovering Analysis Patterns

The requirements model is comprised of a wide variety of elements: **scenario-based (use cases), data-oriented (the data model), class-based, flow-oriented, and behavioral**. Each of these elements examines the problem from a different perspective, and each provides an opportunity to discover patterns that may occur throughout an application domain, or by analogy, across different application domains.

The most basic element in the description of a requirements model is the **use case**. Use cases may serve as the basis for discovering one or more analysis patterns.

A *semantic analysis pattern* (SAP) “is a pattern that describes a small set of coherent use cases that together describe a basic generic application”

REQUIREMENTS MODELING FOR WEBAPPS

Requirements analysis does take time, but solving the wrong problem takes even more time.

How Much Analysis Is Enough?

The degree to which requirements modeling for WebApps is emphasized depends on the following factors:

- Size and complexity of WebApp increment.
- Number of stakeholders
- Size of the WebApp team.
- Degree to which members of the WebApp team have worked together
- Degree to which the organization's success is directly dependent on the success of the design of a specific part of the WebApp.

It only demands an analysis of those requirements that affect only that part of the WebApp.

Requirements Modeling Input

The requirements model provides a detailed indication of the true structure of the problem and provides insight into the shape of the solution. Requirements analysis refines this understanding by providing additional interpretation. As the problem structure is delineated as part of the requirements model.

Requirements Modeling Output

Requirements analysis provides a disciplined mechanism for representing and evaluating WebApp content and function, the modes of interaction that users will encounter, and the environment and infrastructure in which the WebApp resides. Each of these characteristics can be represented as a set of models that allow the WebApp requirements to be analyzed in a structured manner. While the specific models depend largely upon the nature of the WebApp, there are **five** main classes of models:

- **Content model**—identifies the full spectrum of content to be provided by the WebApp. Content includes text, graphics and images, video, and audio data.
- **Interaction model**—describes the manner in which users interact with the WebApp.

- **Functional model**—defines the operations that will be applied to WebApp content and describes other processing functions that are independent of content but necessary to the end user.
- **Navigation model**—defines the overall navigation strategy for the WebApp.
- **Configuration model**—describes the environment and infrastructure in which the WebApp resides.

4. Content. Model for WebApps

The content model contains structural elements that provide an important view of content requirements for a WebApp. These structural elements encompass content objects and all analysis classes, user-visible entities that are created or manipulated as a user interacts with the Content can be developed prior to the implementation of the WebApp, while the WebApp is being built, or long after the WebApp is operational.

A *content object* might be a textual description of a product, an article describing a news event, an action photograph taken at a sporting event, a user's response on a discussion forum, an animated representation of a corporate logo, a short video of a speech, or an audio overlay for a collection of presentation slides. The content objects might be stored as separate files, embedded directly into Web pages, or obtained dynamically from a database. Content objects can be determined directly from use cases by examining the scenario description for direct and indirect references to content. The content model must be capable of describing the content object **Component.**

Interaction Model for WebApps

Interaction model that can be composed of one or more of the following elements: (1) use cases, (2) sequence diagrams, (3) state diagrams,¹⁶ and/or (4) user interface prototypes.

Functional Model for WebApps

The *functional model* addresses two processing elements of the WebApp, each representing a different level of procedural abstraction: (1) user-observable functionality that is delivered by the WebApp to end users, and (2) the operations contained within analysis classes that implement behaviors associated with the class.

User-observable functionality encompasses any processing functions that are initiated directly by the user.

Configuration Models for WebApps

The configuration model is nothing more than a list of server-side and client-side attributes. However, for more complex WebApps, a variety of configuration complexities may have an impact on analysis and design. The UML deployment diagram can be used in situations in which complex configuration architectures must be considered.

Navigation Modeling

Navigation modeling considers how each user category will navigate from one WebApp element (e.g., content object) to another. The mechanics of navigation are defined as part of design. At this stage, you should focus on overall navigation requirements. The following questions should be considered:

- Should certain elements be easier to reach than others? What is the priority for presentation?
- Should certain elements be emphasized to force users to navigate in their direction?
- How should navigation errors be handled?
- Should navigation to related groups of elements be given priority over navigation to a specific element?
- Should navigation be accomplished via links, via search-based access, or by some other means?
- Should certain elements be presented to users based on the context of previous navigation actions?
- Should a navigation log be maintained for users?
- Should a full navigation map or menu be available at every point in a user's interaction?
- Should navigation design be driven by the most commonly expected user behaviors or by the perceived importance of the defined WebApp elements?
- Can a user "store" his previous navigation through the WebApp to expedite future usage?
- For which user category should optimal navigation be designed?
- How should links external to the WebApp be handled? Overlaying the existing browser window? As a new browser window? As a separate frame?

These and many other questions should be asked and answered as part of navigation analysis.