Software Engineering

UNIT - II:

CRITICAL SYSTEMS, SOFTWARE PROCESSES: Critical Systems: A simple safety- critical system; System dependability; Availability and reliability. Software Processes: Models, Process iteration, Process activities; The Rational Unified Process; Computer-Aided Software Engineering.

REQUIREMENTS: Software Requirements: Functional and Non-functional requirements; User requirements; System requirements; Interface specification; The software requirements document. Requirements Engineering Processes: Feasibility studies; Requirements elicitation and analysis; Requirements validation; Requirements management.

1. A Simple Safety-Critical System:

- **Definition:**
    - A safety-critical system is one in which the failure of the system could result in harm to people, damage to the environment, or financial loss.
- **Characteristics:**
    - Emphasis on reliability, fault tolerance, and fail-safe mechanisms.
    - Examples: Airplane control systems, medical devices.

2. System Dependability:

- **Dependability Attributes:**
    - **Reliability:** The ability of a system to perform its functions without failure.
    - **Availability:** The proportion of time the system is operational.
    - **Safety:** The system's ability to operate without causing harm.
    - **Security:** Protection against unauthorized access and harm.

3. Availability and Reliability:

- **Availability:**
    - The system should be available for use when required.
    - Calculated as the ratio of operational time to total time.
- **Reliability:**
    - The probability that the system will perform without failure over a specified time.

II. Software Processes:

1. Models:

- **Definition:**
    - A software process model is a representation of the process followed by a development team.

- **Waterfall Model:**
  - Linear, sequential process.
  - Progresses through phases: requirements, design, implementation, testing, deployment, maintenance.
- **Iterative and Incremental Models:**
  - Development is done in increments, with each increment building upon the previous one.
  - Allows for flexibility and feedback.

2. Process Iteration:

- **Iterative Development:**
  - Repeating development cycles.
  - Each iteration adds new features or improvements.
- **Benefits:**
  - Early feedback, flexibility in accommodating changes.

3. Process Activities:

- **Specification:**
  - Defining the system requirements.
- **Design and Implementation:**
  - Creating a blueprint and translating it into code.
- **Validation:**
  - Ensuring the system meets specified requirements.
- **Evolution:**
  - Making modifications to the system based on changing requirements or improvements.

4. The Rational Unified Process (RUP):

- **Iterative Process:**
  - Divided into phases: Inception, Elaboration, Construction, Transition.
  - Each phase has specific goals and activities.
- **Key Characteristics:**
  - Emphasizes iterative development and continuous feedback.
  - Adaptable to various project sizes and types.

5. Computer-Aided Software Engineering (CASE):

- **Definition:**

    - The use of software tools to assist in the software development process.
- **Benefits:**
    - Increased productivity, improved quality, and better management of the development process.
- **Examples:**
    - Code editors, debuggers, version control systems.

III. Challenges in Critical Systems and Software Processes:

- **Critical Systems Challenges:**
    - Balancing safety, reliability, and functionality.
    - Rigorous testing and verification.
- **Software Processes Challenges:**
    - Adapting to changing requirements.
    - Managing complexity in large projects.

IV. Best Practices:

- **Documentation:**
    - Thorough documentation is crucial for understanding, maintaining, and evolving critical systems.
- **Testing and Validation:**
    - Rigorous testing and validation processes are essential for critical systems.
- **Continuous Improvement:**
    - Adopting a mindset of continuous improvement in software processes to enhance efficiency and quality.

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?

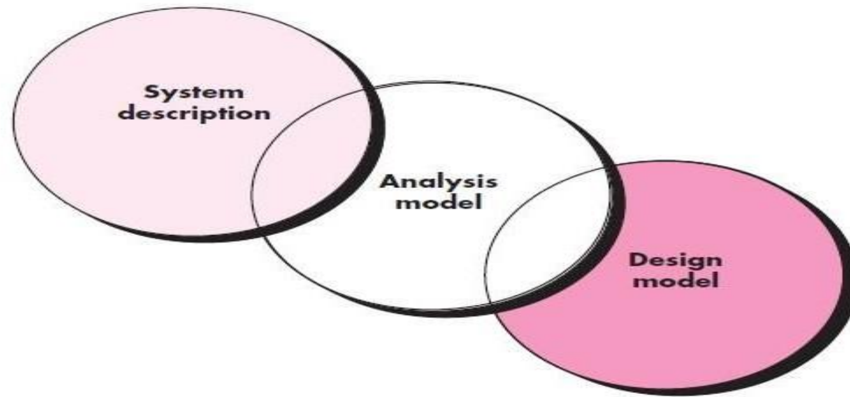


Fig : The requirements model as a bridge between the system description and the design model

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.

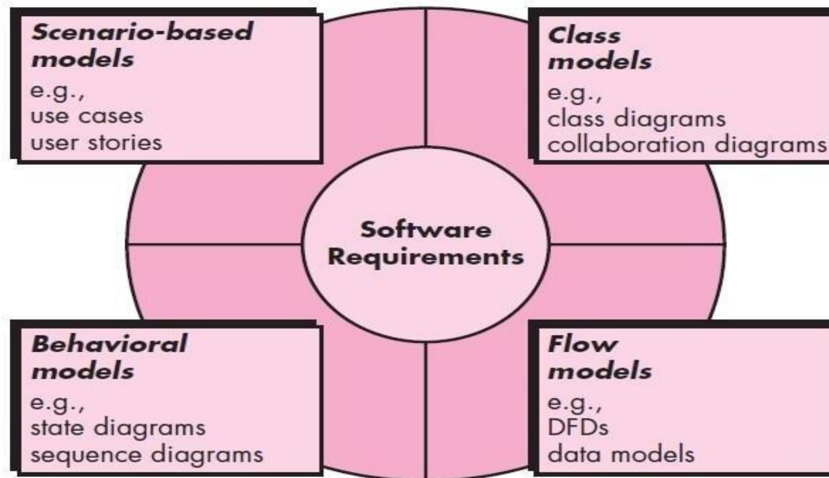


Fig : Elements of the analysis model

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 providevalue 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.

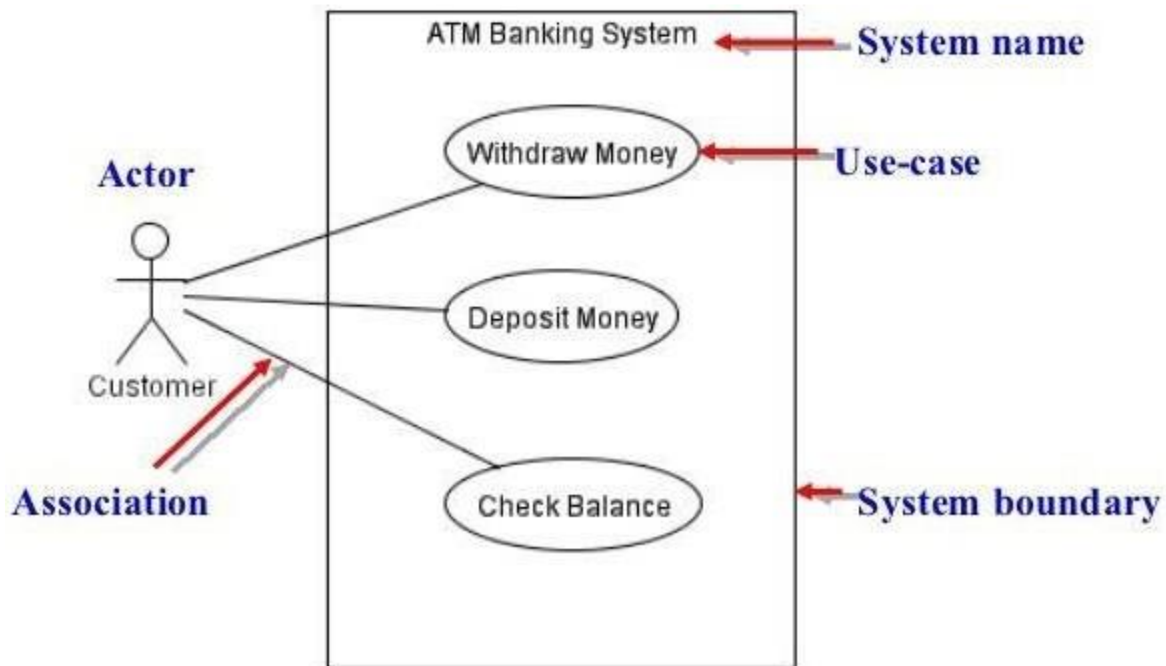


Fig : Simple Use Case Diagram

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.

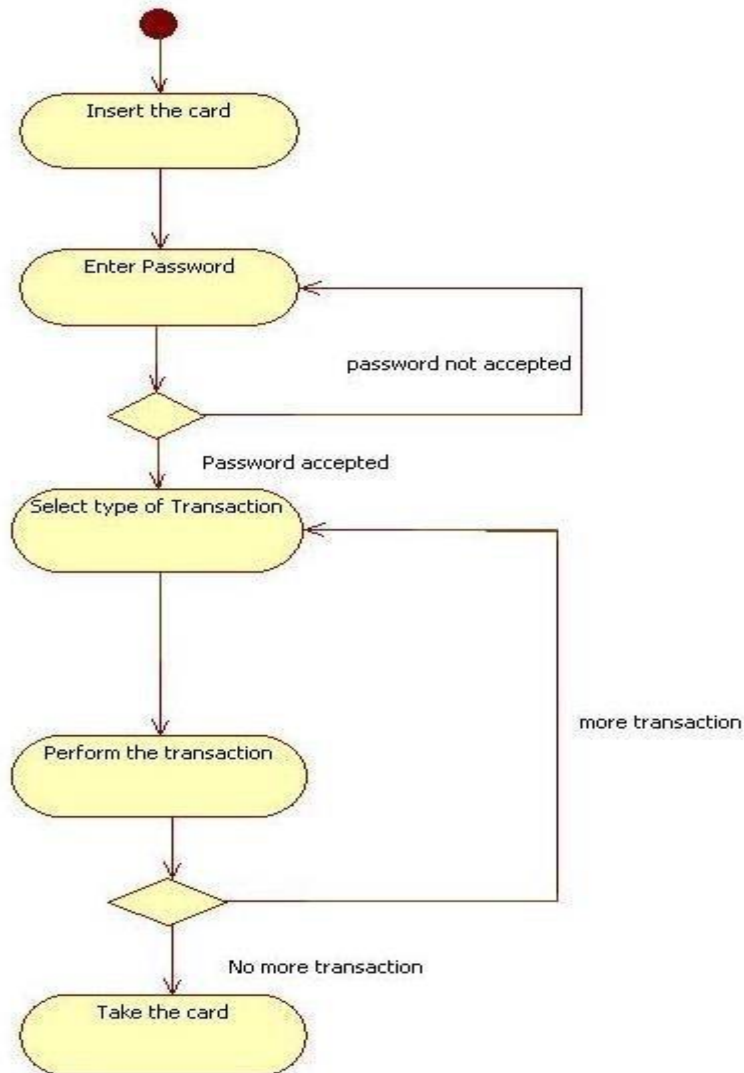


Fig : Activity Diagram for ATM

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*

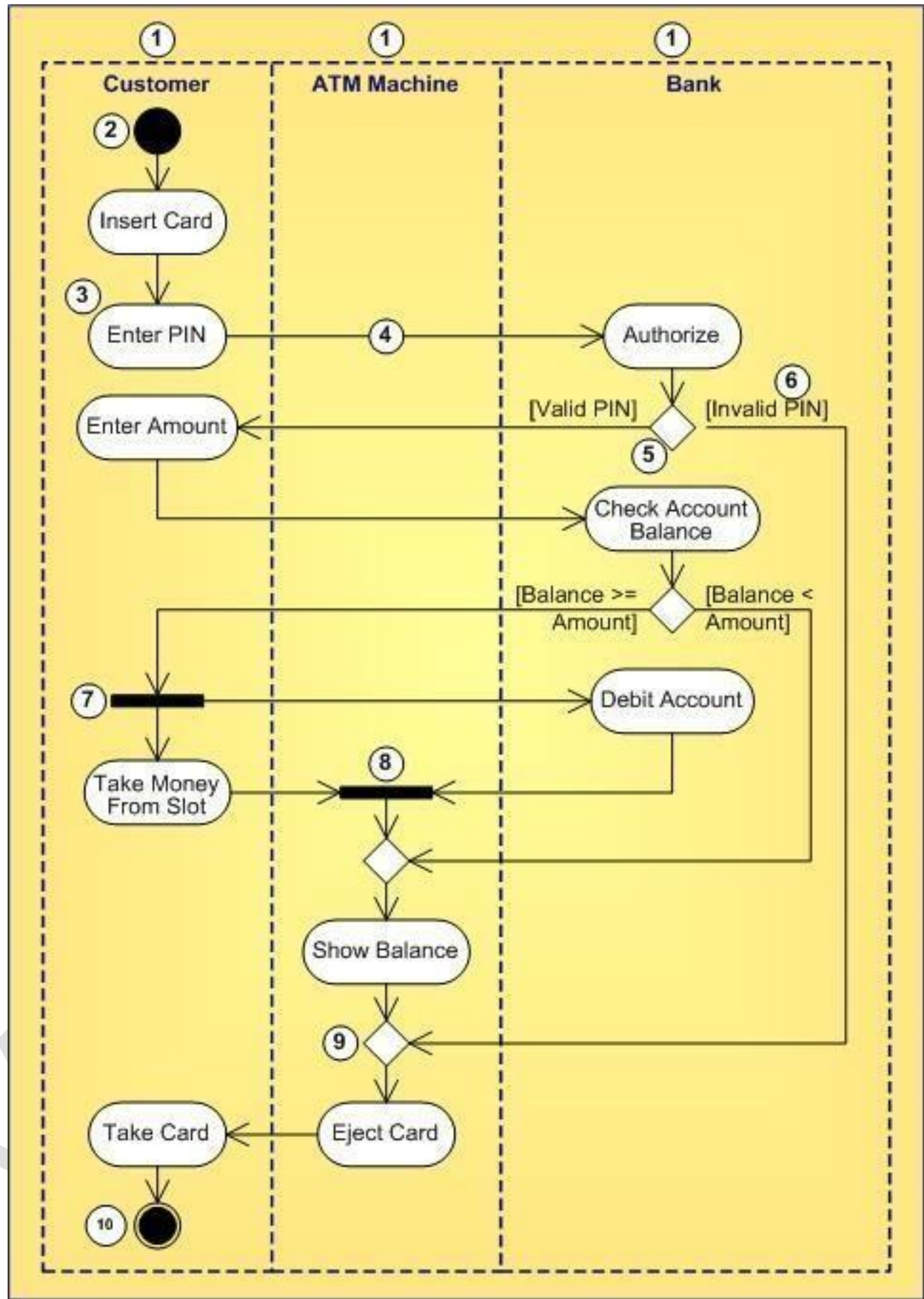


Fig : ***swimlane diagram for ATM***

DATA MODELING CONCEPTS

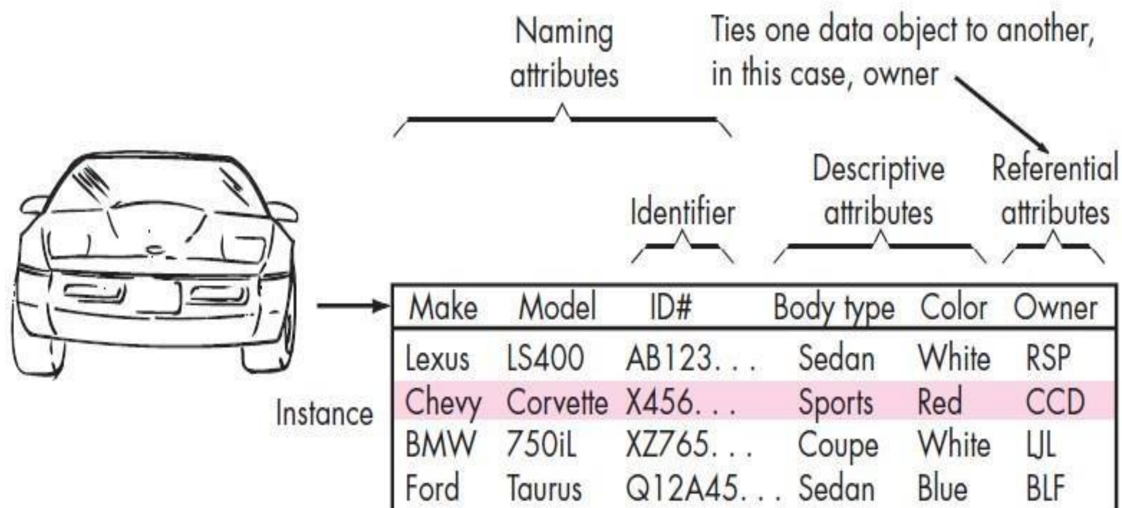
Data modeling is the process of documenting a complex software system design as an easily understood diagram, using text and symbols to represent the way data needs to flow. The diagram can be used as a blueprint for the construction of new software or for re-engineering a legacy application. The most 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.



Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

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

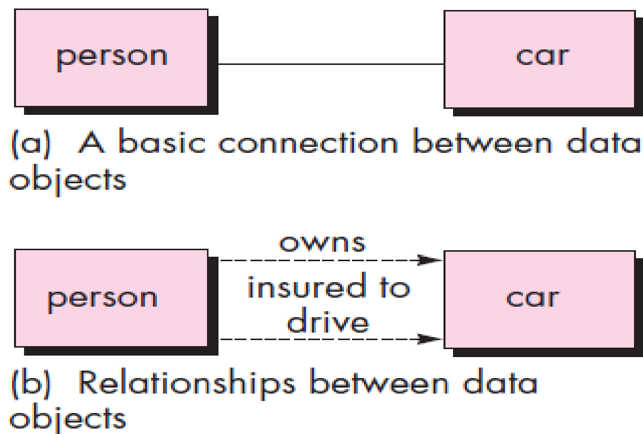


Fig : Relationships between data objects

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 theobjects, 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.

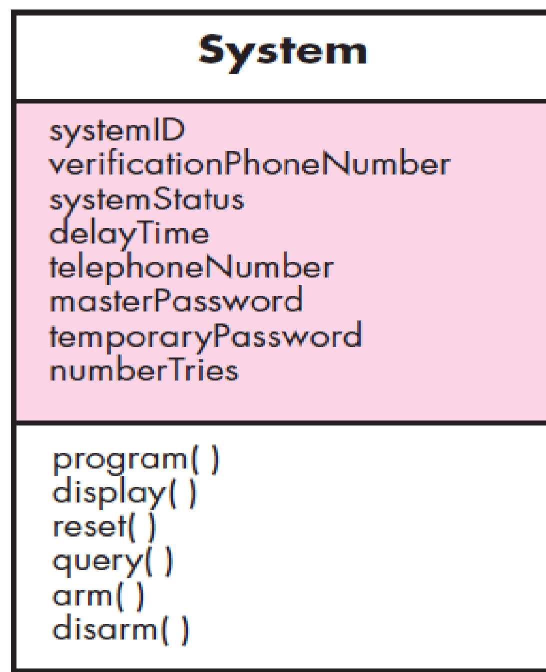


Fig : Class diagram for the system class

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.

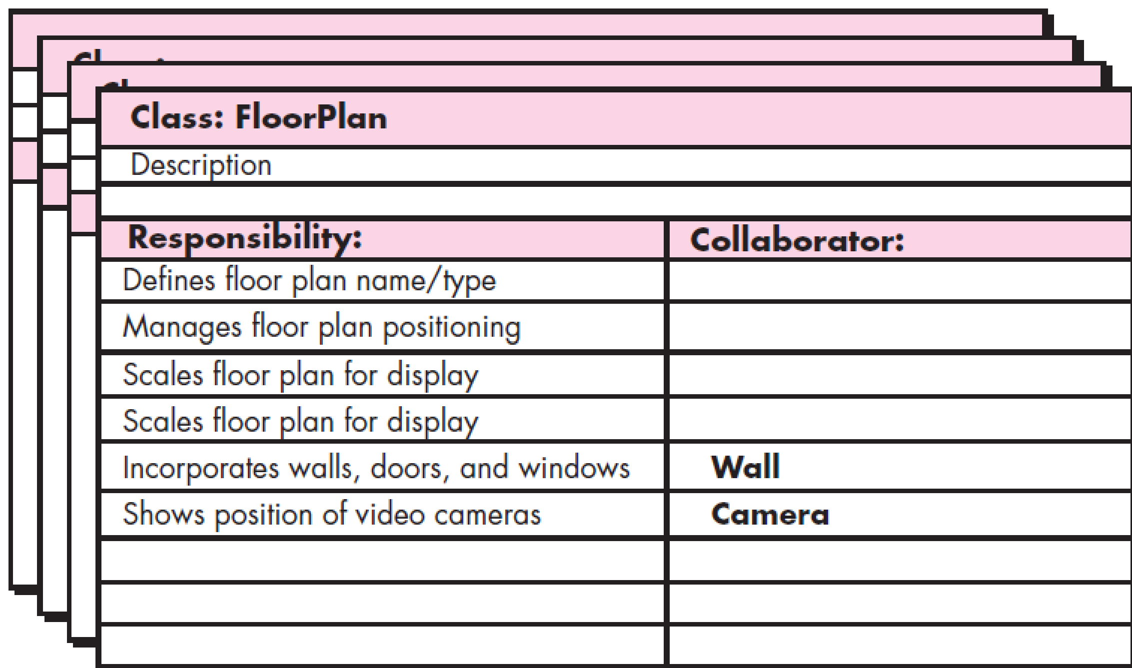


Fig : A CRC model index card

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>>**).

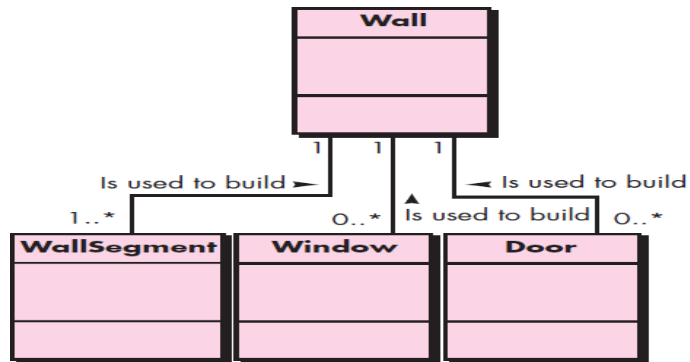


Fig : Multiplicity

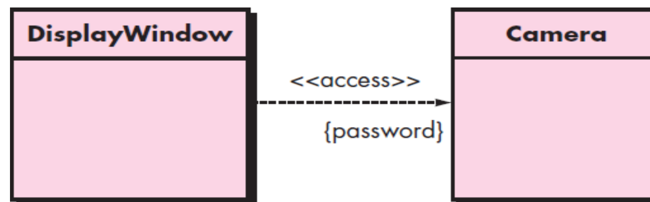
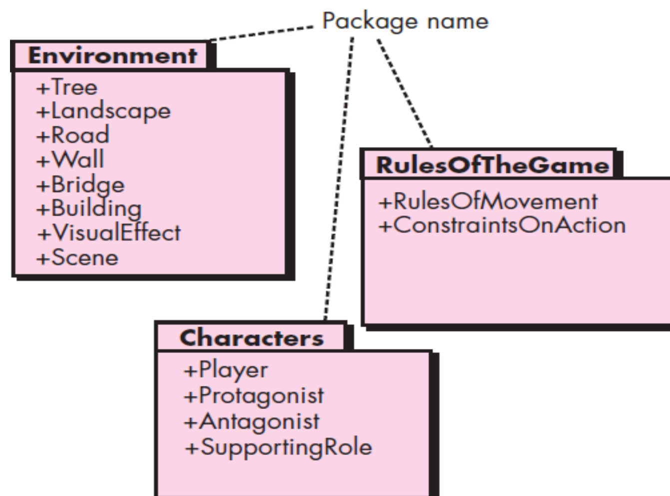


Fig : Dependencies

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