

Unit 2

Requirement Engineering

Requirement Engineering

- The broad **spectrum** of tasks and techniques that lead to an understanding of requirements is called *requirements engineering*.
- From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity.
- It must be adapted to the needs of the process, the project, the product, and the people doing the work.

Requirement Engineering

- Requirements engineering builds a bridge to design and construction. But where does the bridge originate?
- One could argue that it begins at the feet of the project stakeholders (e.g., managers, customers, end users), where
 - business need is defined
 - user scenarios are described
 - functions and features are delineated, and
 - project constraints are identified.
- Others might suggest that it begins with a broader system definition.



Requirement Engineering

- Requirements engineering encompasses seven distinct tasks:
 1. inception,
 2. elicitation,
 3. elaboration,
 4. negotiation,
 5. specification,
 6. validation,
 7. and management.

Requirement Engineering

1. Inception:-

- How does a software project get started?
- In general, most projects begin when a business need is identified or a potential new market or service is discovered.
- Stakeholders from the business community (e.g., business managers, marketing people, product managers):
 - define a business case for the idea
 - try to identify the breadth and depth of the market
 - do a rough feasibility analysis, and
 - identify a working description of the project's scope.

Requirement Engineering

1. Inception:-

- At project inception
 - you establish a basic understanding of the problem,
 - the people who want a solution,
 - the nature of the solution that is desired
 - the effectiveness of preliminary communication
 - and collaboration between the other stakeholders and the software team

Requirement Engineering

2. Elicitation:-

- This phase focuses on gathering the requirements from the stakeholders.
- The right people must be involved in this phase, no space for mistake.
- It certainly seems simple enough—ask the customer, the users, and Others:
 - what the objectives for the system or product are
 - how the system or product fits into the needs of the business, and finally,
 - how the system or product is to be used on a day-to-day basis. But it isn't simple—it's very hard.

Requirement Engineering

2. Elicitation:-

- The following problems can occur in elicitation phase

1. Problem of Scope:-

- The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.

2. Problem of Understanding:-

- Not having clear understanding between the developer and customer. Sometimes customer might not know what they want or the developer might misunderstand the requirement.
- Customer have a poor understanding of the capabilities and limitations of their computing environment, -don't have a full understanding of the problem domain.

Requirement Engineering

2. Elicitation:-

- The following problems can occur in elicitation phase

3. Problems of volatility:-

- Requirements changing over time can cause difficulty in developing project. It can lead to loss and wastage of resources and time.

Requirement Engineering

3. Elaboration:-

- The information obtained from the customer during inception and elicitation is expanded and refined during elaboration.
- This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information.
- Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system

Requirement Engineering

4. Negotiation:-

- Negotiation is between the developer and the customer about limited resources, delivery time, project cost and overall estimation of project
- Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority.
- requirements are eliminated, combined, and/or modified

Requirement Engineering

5. Specification:-

- In this task the requirement engineer constructs a final work product
- The work product is in the form of software requirement specification document.
- A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.
- It collects all user, system, functional and Non-functional requirements
- ER, DFD, Data dictionary model used in this phase
- SRS document is submitted to the customer in the language he/she will understand

Requirement Engineering

6. Validation:-

- The work products or SRS produced as a consequence of requirements engineering are assessed for quality during a validation step.
- Requirements validation examines the specification to ensure that
 - all software requirements have been stated unambiguously
 - inconsistencies, omissions, and errors have been detected and corrected; and
 - the work products conform to the standards established for the process, the project, and the product.
- It also check any missing information or want to add any additional information

Requirement Engineering

6. Validation:-

- The work products or SRS produced as a consequence of requirements engineering are assessed for quality during a validation step.
- Requirements validation examines the specification to ensure that
 - all software requirements have been stated unambiguously
 - inconsistencies, omissions, and errors have been detected and corrected; and
 - the work products conform to the standards established for the process, the project, and the product.
- It also check any missing information or want to add any additional information

Requirement Engineering

6. Validation:-

- The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for:
 - errors in content or interpretation
 - areas where clarification may be required,
 - missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements

Requirement Engineering

7. Requirement Management:-

- Requirements management is a set of activities that help the project team to identify, control, and track requirements and changes to requirements at any time as the project proceeds.
- It tracks on new additional requirement implementation, which will does not affect on overall system.

Establishing the groundwork

- Customer(s) or end users may be located in a different city or country, - may have only a vague idea of what is required,
 - may have conflicting opinions about the system to be built,
 - may have limited technical knowledge, and
 - may have limited time to interact with the requirements engineer.
- None of these things are desirable, but all are fairly common, and you are often forced to work within the constraints imposed by this situation

Establishing the groundwork

- the steps required to establish the groundwork for an understanding of software requirements—to get the project started.

1. Identifying Stakeholders:-

- Sommerville and Sawyer define a stakeholder as “anyone who benefits in a direct or indirect way from the system which is being developed.”
- We have already identified the usual suspects: business operations managers, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers, support and maintenance engineers, and others.
- Each stakeholder has a different view of the system, achieves different benefits when the system is successfully developed,

Establishing the groundwork

2. Recognizing Multiple Viewpoints:-

- Because many different stakeholders exist, the requirements of the system will be explored from many different points of view.
 - E.g. 1] Business managers are interested in a feature set that can be built within budget.
 - 2] End users may want features that are familiar to them and that are easy to learn and use.
- As information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another.
- You should categorize all stakeholder information in a way that will allow decision makers to choose an internally consistent set of requirements for the system.

Establishing the groundwork

3. Working toward collaboration:-

- If five stakeholders are involved in a software project, you may have five (or more) different opinions about the proper set of requirements.
- Customers must work together with software development team to create a successful system
- The job of a requirements engineer is to identify areas of commonality (i.e., requirements on which all stakeholders agree) and areas of conflict or inconsistency.
- a strong "project champion"(e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

Establishing the groundwork

4. Asking the first question (For identifying Stakeholders)

- first set of questions focuses on the customer and other stakeholders, the overall project goals and benefits.
- For example, you might ask:
 - Who is behind the request for this work?
 - Who will use the solution?
 - What will be the economic benefit of a successful solution?
 - Is there another source for the solution that you need?

Establishing the groundwork

4. Asking the first question (Understanding problems and solution)

- The next set of questions enables you to gain a better understanding of the problem and allows the customer to voice his or her perceptions about a solution:
 - How would you characterize “good” output that would be generated by a successful solution?
 - What problem(s) will this solution address?
 - Can you show me (or describe) the business environment in which the solution will be used?

Establishing the groundwork

4. Asking the first question (effectiveness of the communication activity)
 - The final set of questions focuses on the effectiveness of the communication activity-
 - Are you the right person to answer these questions? Are your answers “official”?
 - Are my questions relevant to the problem that you have?
 - Am I asking too many questions?

Establishing the groundwork

5. Non-functional requirements

- A non functional requirement(NFR) can be described as a quality attribute, performance attribute, security attribute or a general constraint on a system.
- Quality function deployment technique(QFD) attempts to translate unspoken customer need into system requirements.
- QFD assist software team in identifying nonfunctional requirements.
- A list of NFR's (e.g. requirement that address usability, testability, security or maintainability) is developed.

Establishing the groundwork

6. Traceability

- It is a software engineering term that refers to documented links between software engineering work products.
- Traceability matrix allows a requirements engineer to represent the relationship between requirements and other software engineering workproducts.
- Rows—requirement names
- Columns-name of software engineering work products.

Eliciting Requirements

- Requirements elicitation (also called *requirements gathering*) combines elements of **problem solving, elaboration, negotiation, and specification.**
- In order to encourage a collaborative, team-oriented approach to requirements gathering, stakeholders work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements

Eliciting Requirements

➤ Collaborative Requirements Gathering:-

- Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:
 - Meetings are conducted and attended by both software engineers and other stakeholders.
 - Rules for preparation and participation are established.
 - An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
 - A “facilitator” (can be a customer, a developer, or an outsider) controls the meeting.
 - A “definition mechanism” (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

Eliciting Requirements

➤ Collaborative Requirements Gathering:-

- The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements.
- During inception basic questions and answers establish the scope of the problem and the overall perception of a solution. Out of these initial meetings, the developer and customers write a one- or two-page "product request."
- A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate.
- The product request is distributed to all attendees before the meeting.

Eliciting Requirements

➤ Collaborative Requirements Gathering:-

- each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions.
- In addition, each attendee is asked to make another list of services (processes or functions) that manipulate or interact with the objects.
- Finally, lists of constraints (e.g., cost, size, business rules) and performance criteria (e.g., speed, accuracy) are also developed.

Eliciting Requirements

➤ Collaborative Requirements Gathering:-

- As an example, consider an expert from a product request written by a marketing person involved in the *SafeHome* project.
- This person writes the following narrative about the home security function that is to be part of *SafeHome*:

Our research indicates that the market for home management systems is growing at a rate of 40 percent per year. The first *SafeHome* function we bring to market should be the home security function. Most people are familiar with “alarm systems” so this would be an easy sell.

The home security function would protect against and/or recognize a variety of undesirable “situations” such as illegal entry, fire, flooding, carbon monoxide levels, and others. It’ll use our wireless sensors to detect each situation. It can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected.

Eliciting Requirements

➤ Collaborative Requirements Gathering:-

- Objects described for **SafeHome** might include the control panel, smoke detectors, window and door sensors, motion detectors, an alarm, an event (a sensor has been activated), a display, a PC, telephone numbers, a telephone call, and so on.
- The list of services might include *configuring the system, setting the alarm, monitoring the sensors, dialing the phone, programming the control panel, and reading the display.*
- In a similar fashion, each attendee will develop lists of constraints (e.g., the system must recognize when sensors are not operating, must be user-friendly, must interface directly to a standard phone line) and performance criteria (e.g., a sensor event should be recognized within one second, and an event priority scheme should be implemented).

Eliciting Requirements

➤ Collaborative Requirements Gathering:-

- The lists of objects can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive-backed sheets, or written on a wall board.
- Alternatively, the lists may have been posted on an electronic bulletin board, at an internal website, or posed in a chat room environment for review prior to the meeting.



Conducting a Requirements Gathering Meeting

The scene: A meeting room. The first requirements gathering meeting is in progress.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator (pointing at whiteboard): So that's the current list of objects and services for the home security function.

Marketing person: That about covers it from our point of view.

Vinod: Didn't someone mention that they wanted all *SafeHome* functionality to be accessible via the Internet? That would include the home security function, no?

Marketing person: Yes, that's right . . . we'll have to add that functionality and the appropriate objects.

Facilitator: Does that also add some constraints?

Jamie: It does, both technical and legal.

Production rep: Meaning?

Jamie: We better make sure an outsider can't hack into the system, disarm it, and rob the place or worse. Heavy liability on our part.

Doug: Very true.

Marketing: But we still need that . . . just be sure to stop an outsider from getting in.

Ed: That's easier said than done and . . .

Facilitator (interrupting): I don't want to debate this issue now. Let's note it as an action item and proceed.

(Doug, serving as the recorder for the meeting, makes an appropriate note.)

Facilitator: I have a feeling there's still more to consider here.

(The group spends the next 20 minutes refining and expanding the details of the home security function.)

Eliciting Requirements

➤ Quality Function Deployment

- *Quality function deployment* (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software.
- QFD “concentrates on maximizing customer satisfaction from the software engineering process”.
- To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.

Eliciting Requirements

➤ Quality Function Deployment

QFD identifies three types of requirements:

1. Normal requirements.

- The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied.
- Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

Eliciting Requirements

➤ Quality Function Deployment

QFD identifies three types of requirements:

2. Expected requirements.

- These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them.
- Their absence will be a cause for significant dissatisfaction.
- Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

Eliciting Requirements

➤ Quality Function Deployment

QFD identifies three types of requirements:

3. **Exciting requirements.**

- These features go beyond the customer's expectations and prove to be very satisfying when present.
- For example, software for a new mobile phone comes with standard features, but is coupled with a set of unexpected capabilities (e.g., multitouch screen, visual voice mail) that delight every user of the product.

Eliciting Requirements

➤ Usage Scenarios

- As requirements are gathered, an overall vision of system functions and features begins to materialize.
- However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users.
- To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed.
- The scenarios, often called *use cases* provide a description of how the system will be used.

Eliciting Requirements

SAFEHOME



Developing a Preliminary User Scenario

The scene: A meeting room, continuing the first requirements gathering meeting.

The players: Jamie Lazar, software team member; Vinod Raman, software team member; Ed Robbins, software team member; Doug Miller, software engineering manager; three members of marketing; a product engineering representative; and a facilitator.

The conversation:

Facilitator: We've been talking about security for access to *SafeHome* functionality that will be accessible via the Internet. I'd like to try something. Let's develop a usage scenario for access to the home security function.

Jamie: How?

Facilitator: We can do it a couple of different ways, but for now, I'd like to keep things really informal. Tell us (he points at a marketing person) how you envision accessing the system.

Marketing person: Um . . . well, this is the kind of thing I'd do if I was away from home and I had to let someone into the house, say a housekeeper or repair guy, who didn't have the security code.

Facilitator (smiling): That's the reason you'd do it . . . tell me how you'd actually do this.

Marketing person: Um . . . the first thing I'd need is a PC. I'd log on to a website we'd maintain for all users of *SafeHome*. I'd provide my user id and . . .

Vinod (interrupting): The Web page would have to be secure, encrypted, to guarantee that we're safe and . . .

Facilitator (interrupting): That's good information, Vinod, but it's technical. Let's just focus on how the end user will use this capability. OK?

Vinod: No problem.

Marketing person: So as I was saying, I'd log on to a website and provide my user ID and two levels of passwords.

Eliciting Requirements

Jamie: What if I forget my password?

Facilitator (interrupting): Good point, Jamie, but let's not address that now. We'll make a note of that and call it an *exception*. I'm sure there'll be others.

Marketing person: After I enter the passwords, a screen representing all *SafeHome* functions will appear. I'd select the home security function. The system might request that I verify who I am, say, by asking for my address or phone number or something. It would then display a picture of the security system control panel

along with a list of functions that I can perform—arm the system, disarm the system, disarm one or more sensors. I suppose it might also allow me to reconfigure security zones and other things like that, but I'm not sure.

(As the marketing person continues talking, Doug takes copious notes; these form the basis for the first informal usage scenario. Alternatively, the marketing person could have been asked to write the scenario, but this would be done outside the meeting.)

Eliciting Requirements

➤ Elicitation Work Products

- The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include
 - A statement of need and feasibility.
 - A bounded statement of scope for the system or product.
 - A list of customers, users, and other stakeholders who participated in requirements elicitation.
 - A description of the system's technical environment.
 - A list of requirements and the domain constraints that apply to each.
 - A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
 - Any prototypes developed to better define requirements.

Developing Use Cases

- An use case tells a stylized story about how an end user interacts with the system under a specific set of circumstances.
- The story may be narrative text, an outline of tasks or interactions, a template-based description, or a diagrammatic representation.
- The first step in writing a use case is to define the set of “actors” that will be involved in the story.
- *Actors* are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described.
- Actors represent the roles that people (or devices) play as the system operates.
- An actor is anything that communicates with the system or product and that is external to the system itself.

Developing Use Cases

- Once actors have been identified, use cases can be developed.
- Jacobson suggests a number of questions that should be answered by a use case:
 - Who is the primary actor, the secondary actor(s)?
 - What are the actor's goals?
 - What preconditions should exist before the story begins?
 - What main tasks or functions are performed by the actor?
 - What exceptions might be considered as the story is described?
 - What variations in the actor's interaction are possible?
 - What system information will the actor acquire, produce, or change?
 - Will the actor have to inform the system about changes in the external environment?
 - What information does the actor desire from the system?
 - Does the actor wish to be informed about unexpected changes?

Developing Use Cases

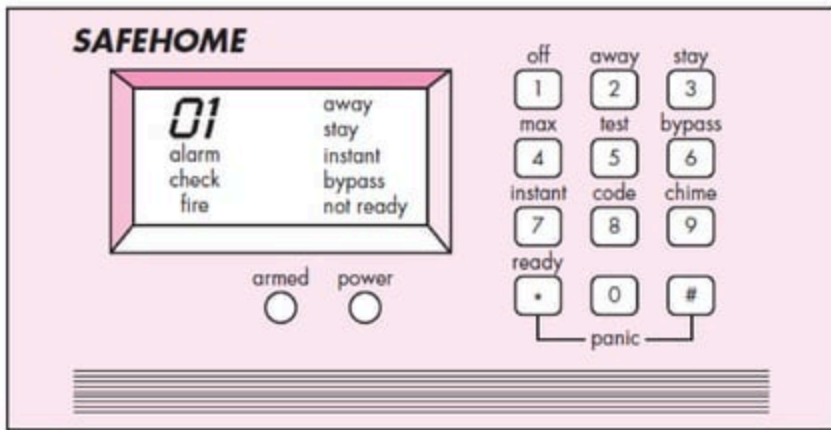
- Recalling basic *SafeHome* requirements, we define four actors:
 1. **homeowner** (a user),
 2. **setup manager** (likely the same person as **homeowner**, but playing a different role),
 3. **sensors** (devices attached to the system), and
 4. the **monitoring and response subsystem** (the central station that monitors the *SafeHome* home security function).

Developing Use Cases

- For the purposes of this example, we consider only the **homeowner** actor. The **homeowner** actor interacts with the home security function in a number of different ways using either the alarm control panel or a PC:
 - Enters a password to allow all other interactions.
 - Inquires about the status of a security zone.
 - Inquires about the status of a sensor.
 - Presses the panic button in an emergency.
 - Activates/deactivates the security system.

Developing Use Cases

Considering the situation in which the homeowner uses the control panel, the basic use case for system activation will be



Developing Use Cases

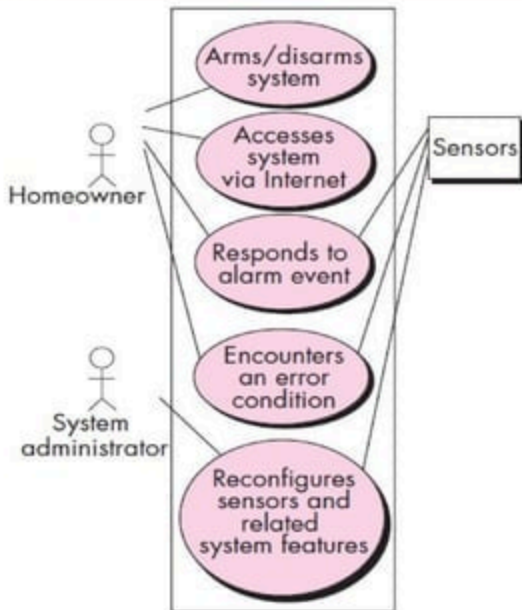
1. The homeowner observes the *SafeHome* control panel (Figure) to determine if the system is ready for input. If the system is not ready, a *not ready* message is displayed on the LCD display, and the homeowner must physically close windows or doors so that the *not ready* message disappears. [A *not ready* message implies that a sensor is open; i.e., that a door or window is open.]
2. 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.

Developing Use Cases

3. The homeowner selects and keys in stay or away (see Figure) to activate the system. Stay activates only perimeter sensors (inside motion detecting sensors are deactivated). Away activates all sensors.
4. 4. When activation occurs, a red alarm light can be observed by the homeowner.

Developing Use Cases

UML use case diagram for
SafeHome home security function



Building the Requirements Model

- The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system.
- The model changes dynamically as you learn more about the system to be built, and other stakeholders understand more about what they really require.

Building the Requirements Model

➤ Elements of the Requirement Model

- Three types of elements we are going to use in Requirement Model
 1. Scenario-based elements
 2. Class-based elements
 3. Behavioral-based elements

Building the Requirements Model

1. Scenario-based elements

- The system is described from the user's point of view using a scenario-based approach.
- Scenario-based elements of the requirements model are often the first part of the model that is developed.
- As such, they serve as input for the creation of other modeling elements.

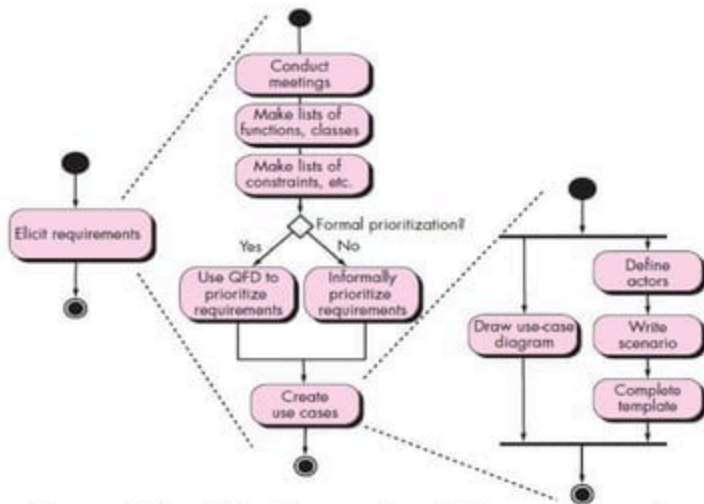


Figure. UML activity diagrams for eliciting requirements

Building the Requirements Model

2. Class-based elements

- Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system.
- These objects are categorized into classes—a collection of things that have similar attributes and common behaviors.
- For example, a UML **class diagram** can be used to depict a **Sensor** class for the *SafeHome* security function

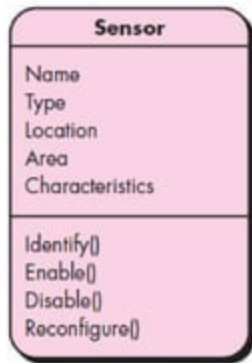


Figure. Class diagram for sensor

Building the Requirements Model

Class diagrams:-

- Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.
- The class diagrams are widely used in the modeling of object-oriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.
- Class diagram describes the attributes and operations of a class and also the constraints imposed on the system.

Building the Requirements Model

Components of Class diagrams:-

- class diagram is made up of three sections:

1. Upper section:-

- The upper section encompasses the name of the class.
- Some of the following rules that should be taken into account while representing a class are given below:
 - a) Capitalize the initial letter of the class name.
 - b) Place the class name in the center of the upper section.
 - c) A class name must be written in bold format.
 - d) The name of the abstract class should be written in italics format.

Building the Requirements Model

Components of Class diagrams:-

2. Middle section:-

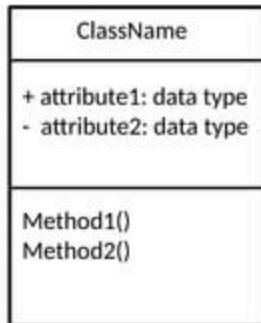
- The middle section constitutes the attributes, which describe the quality of the class.
- The attributes have the following characteristics:
 - a) The attributes are written along with its visibility factors, which are public (+), private (-), protected (#), and package (~).
 - b) The accessibility of an attribute class is illustrated by the visibility factors.
 - c) A meaningful name should be assigned to the attribute, which will explain its usage inside the class.

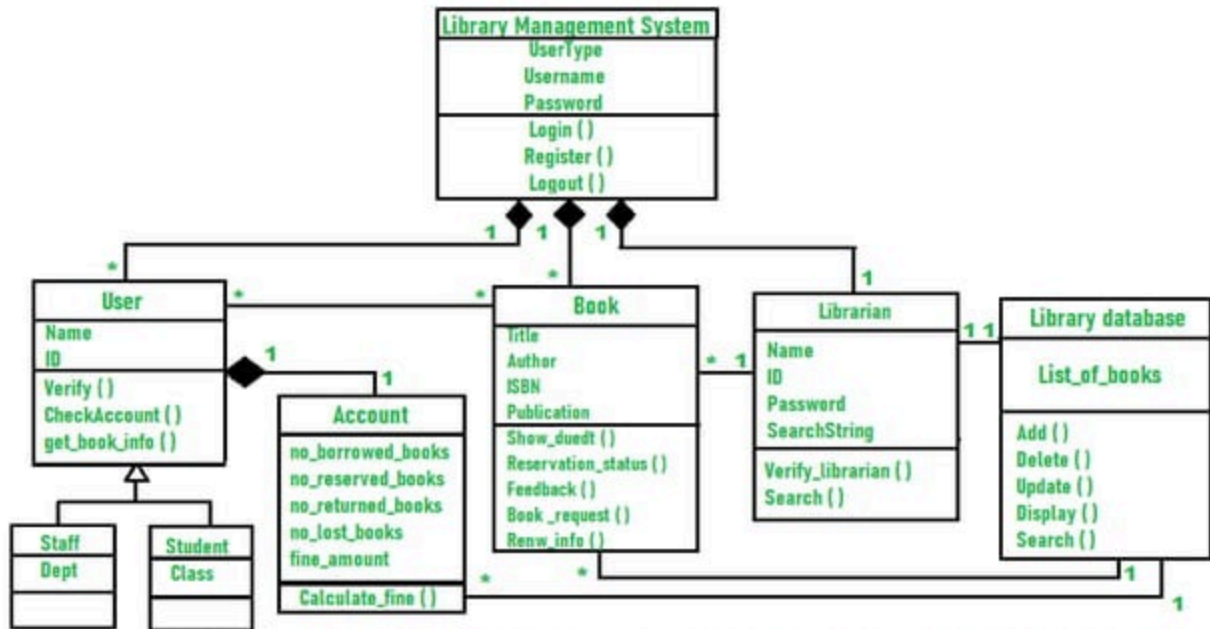
Building the Requirements Model

Components of Class diagrams:-

3. Lower section:-

- The lower section contain methods or operations. The methods are represented in the form of a list, where each method is written in a single line. It demonstrates how a class interacts with data.





CLASS DIAGRAM FOR LIBRARY MANAGEMENT SYSTEM

Building the Requirements Model

3. Behavioral elements

- The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied.
- Therefore, the requirements model must provide modeling elements that depict behavior.
- The **state diagram** is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state.
- A state is any externally observable mode of behavior.
- In addition, the state diagram indicates actions (e.g., process activation) taken as a consequence of a particular event.

Building the Requirements Model

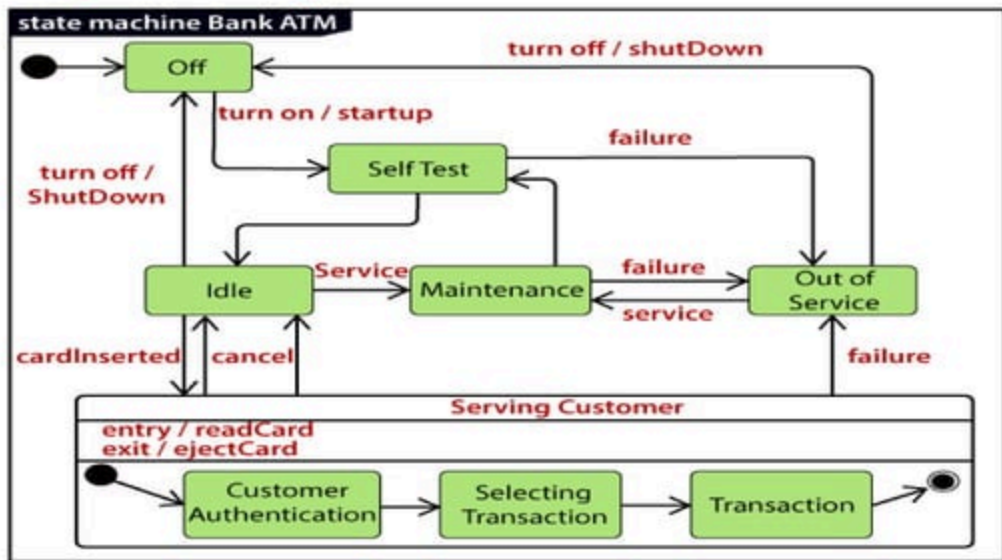


Figure. State Diagram for ATM Machine

Negotiating Requirements

- In an ideal requirements engineering context, the inception, elicitation, and elaboration tasks determine customer requirements in sufficient detail to proceed to subsequent software engineering activities.
- you may have to enter into a negotiation with one or more stakeholders. In most cases, stakeholders are asked to balance functionality, performance, and other product or system characteristics against cost and time-to-market.
- The intent of this negotiation is to develop a project plan that meets stakeholder needs while at the same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team.

Negotiating Requirements

- The best negotiations strive for a “win-win” result. That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you (as a member of the software team) win by working to realistic and achievable budgets and deadlines.
- Boehm defines a set of negotiation activities at the beginning of each software process iteration.

Negotiating Requirements

- Rather than a single customer communication activity, the following activities are defined:
 1. Identification of the system or subsystem's key stakeholders.
 2. Determination of the stakeholders' "win conditions."
 3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software team).

Negotiating Requirements

INFO



The Art of Negotiation

Learning how to negotiate effectively can serve you well throughout your personal and technical life. The following guidelines are well worth considering:

1. *Recognize that it's not a competition.* To be successful, both parties have to feel they've won or achieved something. Both will have to compromise.
2. *Map out a strategy.* Decide what you'd like to achieve; what the other party wants to achieve, and how you'll go about making both happen.
3. *Listen actively.* Don't work on formulating your response while the other party is talking. Listen to her. It's likely you'll gain knowledge that will help you to better negotiate your position.
4. *Focus on the other party's interests.* Don't take hard positions if you want to avoid conflict.
5. *Don't let it get personal.* Focus on the problem that needs to be solved.
6. *Be creative.* Don't be afraid to think out of the box if you're at an impasse.
7. *Be ready to commit.* Once an agreement has been reached, don't waffle; commit to it and move on.

Negotiating Requirements

- Rather than a single customer communication activity, the following activities are defined:
 1. Identification of the system or subsystem's key stakeholders.
 2. Determination of the stakeholders' "win conditions."
 3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software team).

Validating Requirements

- As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity.
- The requirements represented by the model are prioritized by the stakeholders and grouped within requirements packages that will be implemented as software increments.
- A review of the requirements model addresses the following questions:
 - Is each requirement consistent with the overall objectives for the system/product?
 - Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?

Validating Requirements

- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?
- Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?

Requirements Analysis

- Requirements analysis allows you to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering.
- Requirement models provide a software designer with information that can be translated to architectural, interface, and component-level designs.
- Finally, the requirements model (and the software requirements specification) provides the developer and the customer with the means to assess quality once software is built.

Requirements Analysis

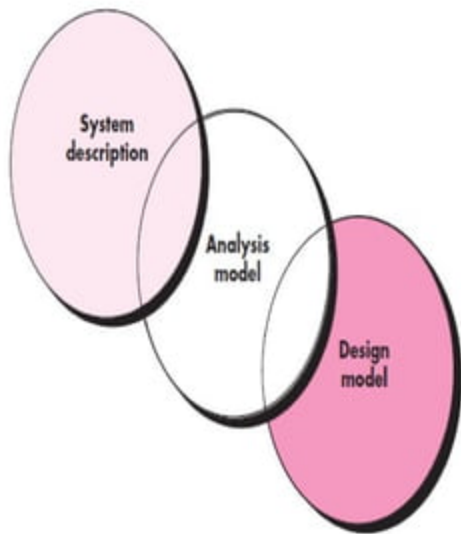
Overall Objectives and Philosophy

- Throughout requirements modeling, your 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

Requirements Analysis

Overall Objectives and Philosophy

- 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 and a software design



Requirements Analysis

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.*
 - *Delay consideration of infrastructure and other nonfunctional models until design.*
 - *Minimize coupling throughout the system.*
 - *Be certain that the requirements model provides value to all stakeholders.*
 - *Keep the model as simple as it can be.*

Requirements Analysis

Domain Analysis

- Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain. . . .

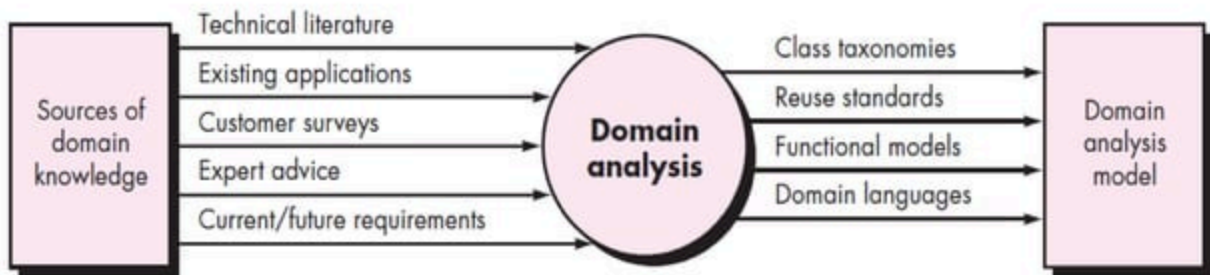


Figure. Input and Output for domain Analysis

Requirements Analysis

Requirements Modeling Approaches

- One view of requirements modeling, called *structured analysis*, considers data and the processes that transform the data as separate entities.
- 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.

Requirements Analysis

Requirements Modeling Approaches

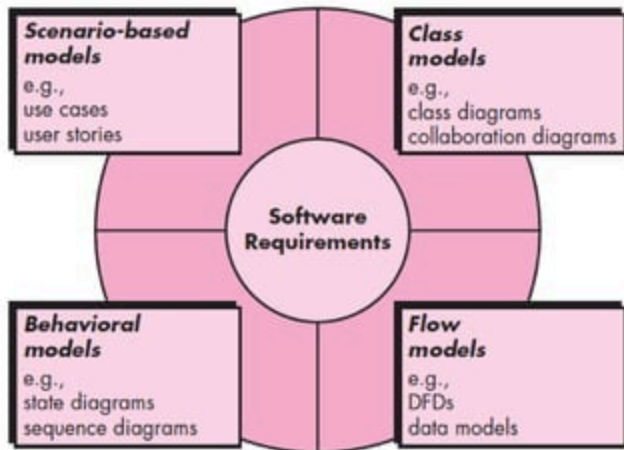


Figure. Elements of Analysis Model

UML Models that Supplement the Use Case

- There are many requirements modeling situations in which a text-based model—even one as simple as a use case—may not impart information in a clear and concise manner.
- In such cases, you can choose from a broad array of UML graphical models.




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
 - and solid horizontal lines to indicate that parallel activities are occurring.



Activity Diagram

➤ Notations used in Activity diagrams

Symbol	Name	Description
	Initial State/ start Symbol	Represents the beginning of a process or workflow in an activity diagram. It can be used by itself or with a note symbol that explains the starting point.
	Activity/ Action Box	It represents the set of actions that are to be performed by system
	Decision Box	It is used to make decision and follow the single path depends upon the decision.




Activity Diagram

➤ Notations used in Activity diagrams

Symbol	Name	Description
	Connector Symbol	Shows the directional flow, or control flow, of the activity. An incoming arrow starts a step of an activity; once the step is completed, the flow continues with the outgoing arrow.
	Join Symbol	Combines two concurrent activities and re-introduces them to a flow where only one activity occurs at a time. Represented with a thick vertical or horizontal line.

Activity Diagram

➤ Notations used in Activity diagrams

Symbol	Name	Description
	Fork Symbol	Splits a single activity flow into two concurrent activities. Symbolized with multiple arrowed lines from a join.
	Note Symbol	Allows the diagram creators or collaborators to communicate additional messages that don't fit within the diagram itself. Leave notes for added clarity and specification.
	End Symbol	Marks the end state of an activity and represents the completion of all flows of a process.

UML Models that Supplement the Use Case

Developing an Activity Diagram

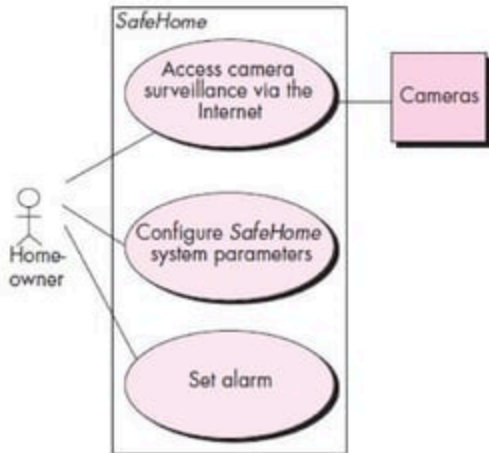


Figure. Preliminary use-case diagram for the *SafeHome* system

UML Models that Supplement the Use Case

Developing an Activity Diagram

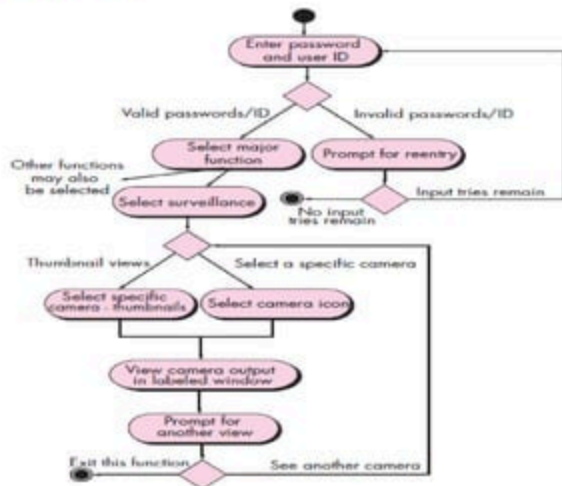


Figure. Activity Diagram for Access Camera surveillance via the internet display camera views function

UML Models that Supplement the Use Case

Developing an Activity Diagram

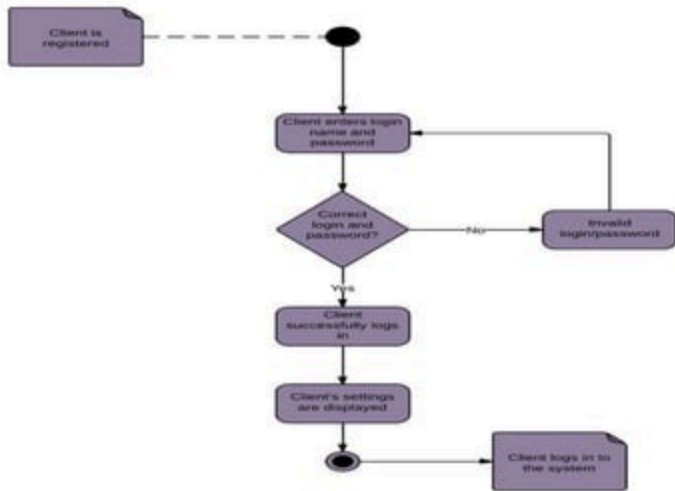


Figure. Activity Diagram for Login Page

UML Models that Supplement the Use Case

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.

UML Models that Supplement the Use Case

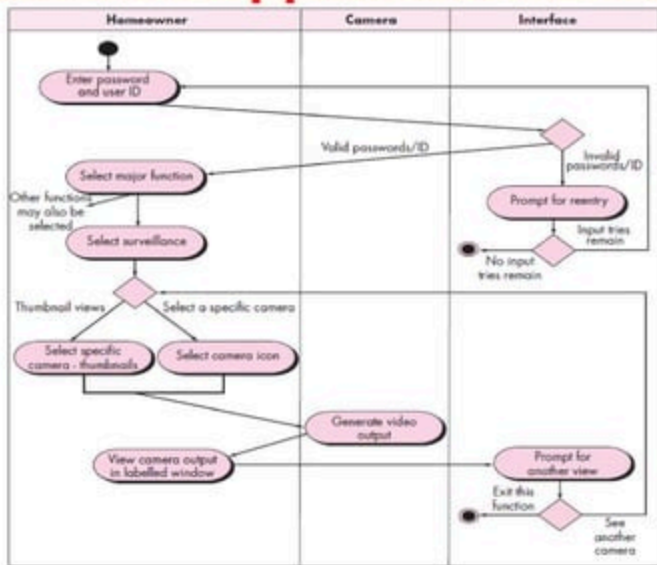


Figure. Swimlane diagram for Access camera surveillance via the Internet—display camera views function

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

Class-Based Modeling

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.
- Classes are determined by underlining each noun or noun phrase and entering it into a simple

Class-Based Modeling

Identifying Analysis classes

- *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.

Class-Based Modeling

Identifying Analysis classes

- For *SafeHome* project if we Extract the nouns, we can propose a number of potential classes as follows

Potential Class	General Classification
homeowner	role or external entity
sensor	external entity
control panel	external entity
installation	occurrence
system (alias security system)	thing
number, type	not objects, attributes of sensor
master password	thing
telephone number	thing
sensor event	occurrence
audible alarm	external entity
monitoring service	organizational unit or external entity

Class-Based Modeling

Specifying Attributes

- *Attributes* describe a class that has been selected for inclusion in the requirements model.
- Attributes define the characteristics of class
- Attributes essentially characterize what the class represents in the problem space.
- Example, a system that tracks baseball statistics for professional baseball players, the attributes of the class **Player** would be name, position, batting average, and years played are relevant.
- whereas in a professional baseball pension system, attributes may include average salary, credit toward full vesting, pension plan options, and mailing address.

Class-Based Modeling

Specifying Attributes

- *Composite attributes:- Those attributes that can be divided into more simple attribute*
 - Address : Street + City + State + Zip

Class-Based Modeling

Define 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:
 - Adding: Adding a new book to the library catalog.
 - Deleting: Removing a book from the library inventory.
 - Reformatting: Reformatting the due date of a borrowed book.
 - Selecting: Selecting multiple books to check out at once.

Class-Based Modeling

Define Operations

2. Operations that perform a computation:
 - Calculating: Calculating the late fee for overdue books.
 - Sorting: Sorting the list of books by title, author, or category.
 - Summing: Summing up the total number of available books in the library.
3. Operations that inquire about the state of an object:
 - Checking: Checking if a particular book is available for borrowing.
 - Retrieving: Retrieving the details of a specific library member's borrowing history.
 - Verifying: Verifying if a library card is valid and active.
4. Operations that monitor an object for the occurrence of a controlling event:
 - Listening: Listening for a request to reserve a book that is currently unavailable.
 - Watching: Watching for the return of a borrowed book to update the inventory.

Class-Responsibility-Collaborator (CRC) Modeling

- CRC stands for Class Responsibility Collaborator Cards.
- Introduced at OOPSLA in 1989 by Kent Beck and Ward Cunningham as an approach for teaching object-oriented design.
- In 1995, CRC cards are used extensively in teaching and exploring early design ideas.
- A CRC Card is an index card used to represent:
 - a class of objects
 - their behavior
 - their interactions
- The cards are created based on System requirement that model the behavior of the system through scenario.

Class-Responsibility-Collaborator (CRC) Modeling

➤ CRC Card format

Class Name	
Superclasses	
Subclasses	
Responsibilities	Collaborators

Class-Responsibility-Collaborator (CRC) Modeling

➤ CRC Card Definitions

- **Name:-**
 - The name located at the top of the card, describes the class that the CRC card represents.
- **Responsibility:-**
 - Responsibilities describes the behaviors of class and are represented along the left side of the card. Each distinct responsibility is shown at each row.
- **Collaborators:-**
 - Some responsibilities will collaborate with one or more other classes to fulfill one or more scenarios. Collaborators are listed on the right hand side of the CRC card, next to Responsibilities.

Class-Responsibility-Collaborator (CRC) Modeling

- A small library system as example.
- There are four roles:
 - Book: the information about the book, include title, author, register code...etc.
 - Librarian: the role who manage books.
 - Borrower: the one borrow books include their contact.
 - Date: to record which day the book been borrow and return.

Class-Responsibility-Collaborator (CRC) Modeling

- A small library system as example.

Class: <i>Book</i>	
Responsibilities	Collaborators
knows whether on loan	
knows due date	
knows its title	
knows its author(s)	
knows its registration code	
knows if late	Date
check out	

Class: <i>Librarian</i>	
Responsibilities	Collaborators
check in book	Book
check out book	Book, Borrower
search for book	Book
knows all books	
search for borrower	Borrower
knows all borrowers	

Class: <i>Borrower</i>	
Responsibilities	Collaborators
knows its name	
keeps track of borrowed items	
keeps track of overdue fines	

Class: <i>Date</i>	
Responsibilities	Collaborators
knows current date	
can compare two dates	
can compute new dates	

Association and Dependencies

- 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.
- It also describes how many objects are taking part in that relationship.
- Association represented by a dotted line with or without arrows on both sides.
- The two ends represents two associated elements as shown in the following figure. The multiplicity is also mentioned at the ends (1, *, etc.) to show how many objects are associated.



Association and Dependencies

- In many instances, 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* are represented in double angle brackets (e.g., <<stereotype>>).

