SOFTWARE ENGINEERING

Week 6
Requirement Modeling

Course ID 06016410,
06016321

Nont Kanungsukkasem, B.Eng., M.Sc., Ph.D.
nont@it.kmitl.ac.th

Quick Look

Quick Look

What is it? Requirements modeling uses a combination of text and diagrammatic forms to depict requirements in a way that is relatively easy to understand, and more important, straightforward to review for correctness, completeness, and consistency.

Who does it? A software engineer (sometimes called an analyst) builds these models from requirements elicited from various stakeholders.

Why is it important? Requirements models can be readily evaluated by all stakeholders, resulting in useful feedback at the earliest possible time. Later, as the model is refined, it becomes the basis for software design.

What are the steps? Requirements modeling combines three steps: scenario-based modeling, class modeling, and behavioral modeling.

What is the work product? Usage scenarios, called use cases, describe software functions and usage. In addition, a series of UML diagrams can be used to represent system behavior and other aspects.

How do I ensure that I've done it right? Requirements modeling work products must be reviewed for correctness, completeness, and consistency.

Requirement Analysis

- Requirements analysis
  - results in the specification of software's operational characteristics,
  - indicates software's interface with other system elements
  - establishes constraints that software must meet
- Requirements analysis allows you (regardless of whether you're called a software engineer, an analyst, or a modeler) to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering.

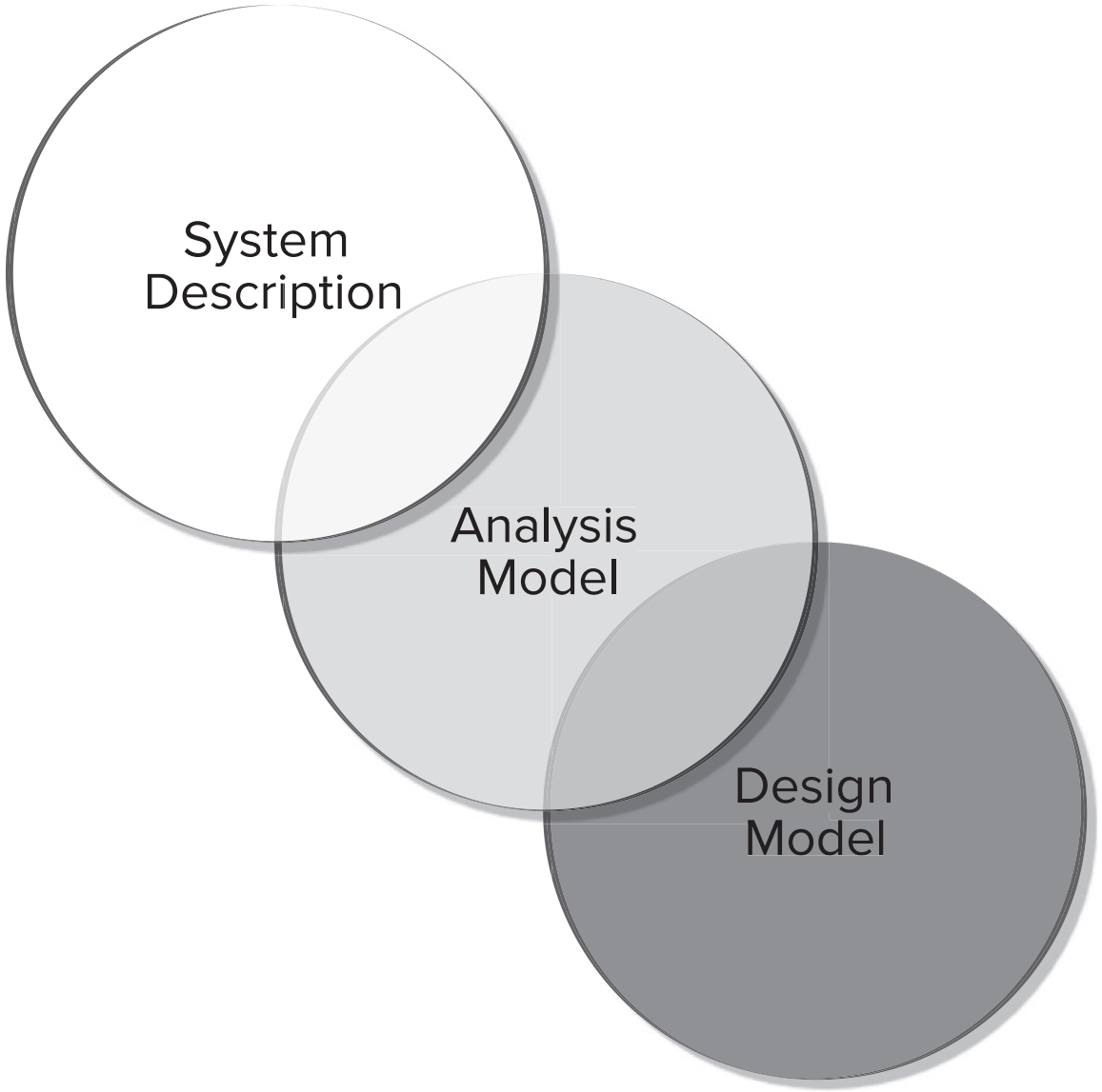
Requirement Models

- Scenario-Based models
- Class-Based models.
- Behavioral models.
- Functional models
- Data models
- Flow-oriented models

Objectives and Philosophy

- Throughout analysis modeling, your primary focus is on *what*, not *how*.
  - What user interaction occurs, 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?
- 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
  3. to define a set of requirements that can be validated once the software is built.

A Bridge

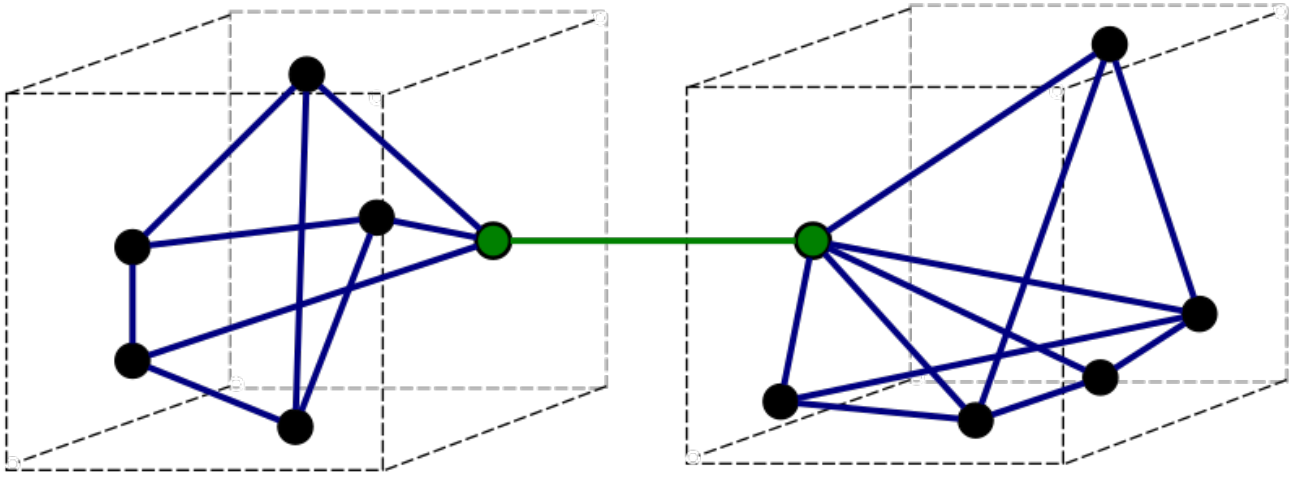


Analysis Rules of Thumb

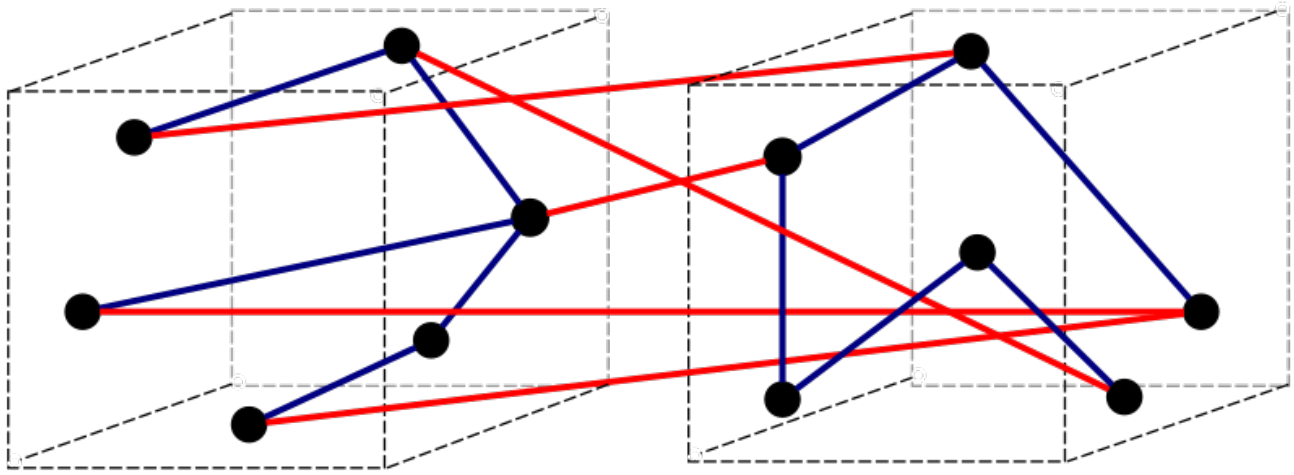
- First, focus on the problem or business domain while keeping the level of abstraction high.
- Second, recognize that an analysis model should provide insight into the information domain, the function, and the behavior of the software.
- Third, delay a consideration of software architecture and nonfunctional details until later in the modeling activity.
- Also, it's important to be aware of the ways in which elements of the software are interconnnected with other elements.

Coupling vs Cohesion

	Cohesion	Coupling
1	Cohesion is the degree to which the elements inside a module belong together.	Coupling is the degree of interdependence between the modules.
2	A module with high cohesion contains elements that are tightly related to each other and united in their purpose.	Two modules have high coupling (or tight coupling) if they are closely connected and dependent on each other.
3	A module is said to have low cohesion if it contains unrelated elements.	Modules with low coupling among them work mostly independently of each other.
4	Highly cohesive modules reflect higher quality of software design	Loose coupling reflects the higher quality of software design



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

Requirements Modeling Principles

1. The information domain of a problem must be represented and understood.
2. The functions that the software performs must be defined.
3. The behavior of the software (as a consequence of external events) must be represented.
4. The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.
5. The analysis task should move from essential information toward implementation detail.

1 Scenario-Based Modeling

Creating Use Cases

- "Contract for behavior"
- The "contract" defines the way in which an actor uses a computer-based system to accomplish some goal.
- In other words, a use case captures the interactions that occur between producers and consumers of information within the system itself.
- A use case describes a specific usage scenario in straightforward language from the point of view of a defined actor.
- If use cases are to provide value as a modeling tool, the following questions must be answered:
  - what to write about?
  - how much to write about it?
  - how detailed to make your description?
  - how to organize the description?

What to Write About?

- Inception and elicitation provide you with the information you'll need to begin writing use cases.
- Requirements gathering meetings and other requirements engineering mechanisms are used to
  - identify stakeholders
  - define the scope of the problem
  - specify overall operational goals
  - establish priorities, outline all known functional requirements
  - describe the things (objects) that will be manipulated by the system.
- To begin developing a set of use cases, list the functions or activities performed by a specific actor.
  - You can obtain these from a list of required system functions, through conversations with stakeholders, or by an evaluation of activity diagrams developed as part of requirements modeling.

Example: Abbreviated list of functions

- An abbreviated list of functions in SafeHome that are performed by the homeowner actor:
  - Select camera to view.
  - Request thumbnails from all cameras.
  - Display camera views in a device window.
  - Control pan and zoom for a specific camera.
  - Selectively record camera output.
  - Replay camera output.
  - Access camera surveillance via the Internet.

How much to write about?

- As further conversations with the stakeholder (who plays the role of a homeowner) progress, the requirements gathering team develops use cases for each of the functions noted.
- In general, use cases are written first in an informal narrative fashion.
- If more formality is required, the same use case is rewritten using a structured format.

Example: Use case (User Story)

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

If I'm at a remote location, I can use any mobile device with appropriate browser software to log on to the *SafeHome Products* website. I enter my user ID and two levels of passwords and once I'm validated, I have access to all functionality for my installed *SafeHome* system. To access a specific camera view, I select "surveillance" from the major function buttons displayed. I then select "pick a camera" and the floor plan of the house is displayed. I then select the camera that I'm interested in. Alternatively, I can look at thumbnail snapshots from all cameras simultaneously by selecting "all cameras" as my viewing choice. Once I choose a camera, I select "view" and a one-frame-per-second view appears in a viewing window that is identified by the camera ID. If I want to switch cameras, I select "pick a camera," the original viewing window disappears, and the floor plan of the house is displayed again. I then select the camera that I'm interested in. A new viewing window appears.

Example: Use case (User Actions)

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Actor: homeowner

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Alternative Interactions

- A description of alternative interactions is essential for a complete understanding of the function that is being described by a use case.
- Therefore, 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?

Example: Use case (Secondary Scenarios)

- Consider steps 6 and 7 in the primary scenario presented earlier:

> 6. The homeowner selects "pick a camera."
> 7. The system displays the floor plan of the house.

- Can the actor take some other action at this point?
  - "View thumbnail snapshots for all cameras."
- Is it possible that the actor will encounter some error condition at this point? If so, what might it be?
  - "No floor plan configured for this house."
- 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?
  - The system may encounter an alarm condition.

Alternative Interactions: Additional Questions

- In addition to the three generic questions, the following issues should also be explored:
  - Are there cases in which some "validation function" occurs during this use case?
    - This implies that the 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 or mobile 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.

Documenting Use Cases

- A typical outline for formal use cases.
  - 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" [Coc01b].
  - 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.

Example: Use Case Template

Use case: Access camera surveillance via the Internet—display camera views (ACS-DCV)

Iteration: 2, last modification: January 14 by V. Raman.

Primary actor: Homeowner.

Goal in context: To view output of cameras placed throughout the house from any remote location via the Internet.

Preconditions: System must be fully configured; appropriate user ID and passwords must be obtained.

Trigger: The homeowner decides to take a look inside the house while away.

Scenario:

1. The homeowner logs onto the *SafeHome Products* website.
2. The homeowner enters his or her user ID.
3. The homeowner enters two passwords (each at least eight characters in length).
4. The system displays all major function buttons.
5. The homeowner selects the "surveillance" button from the major function buttons.
6. The homeowner selects "pick a camera."
7. The system displays the floor plan of the house.
8. The homeowner selects a camera icon from the floor plan.
9. The homeowner selects the "view" button.
10. The system displays a viewing window that is identified by the camera ID.
11. The system displays video output within the viewing window at one frame per second.

Exceptions:

1. ID or passwords are incorrect or not recognized—see use case **Validate ID and passwords.**
2. Surveillance function not configured for this system—system displays appropriate error message; see use case **Configure surveillance function.**
3. Homeowner selects "View thumbnail snapshots for all camera"—see use case **View thumbnail snapshots for all cameras.**
4. A floor plan is not available or has not been configured—display appropriate error message and see use case **Configure floor plan.**
5. An alarm condition is encountered—see use case **Alarm condition encountered.**

Priority: Moderate priority, to be implemented after basic functions.

When available: Third increment.

Frequency of use: Infrequent.

Channel to actor: Via PC-based browser and Internet connection.

Secondary actors: System administrator, cameras.

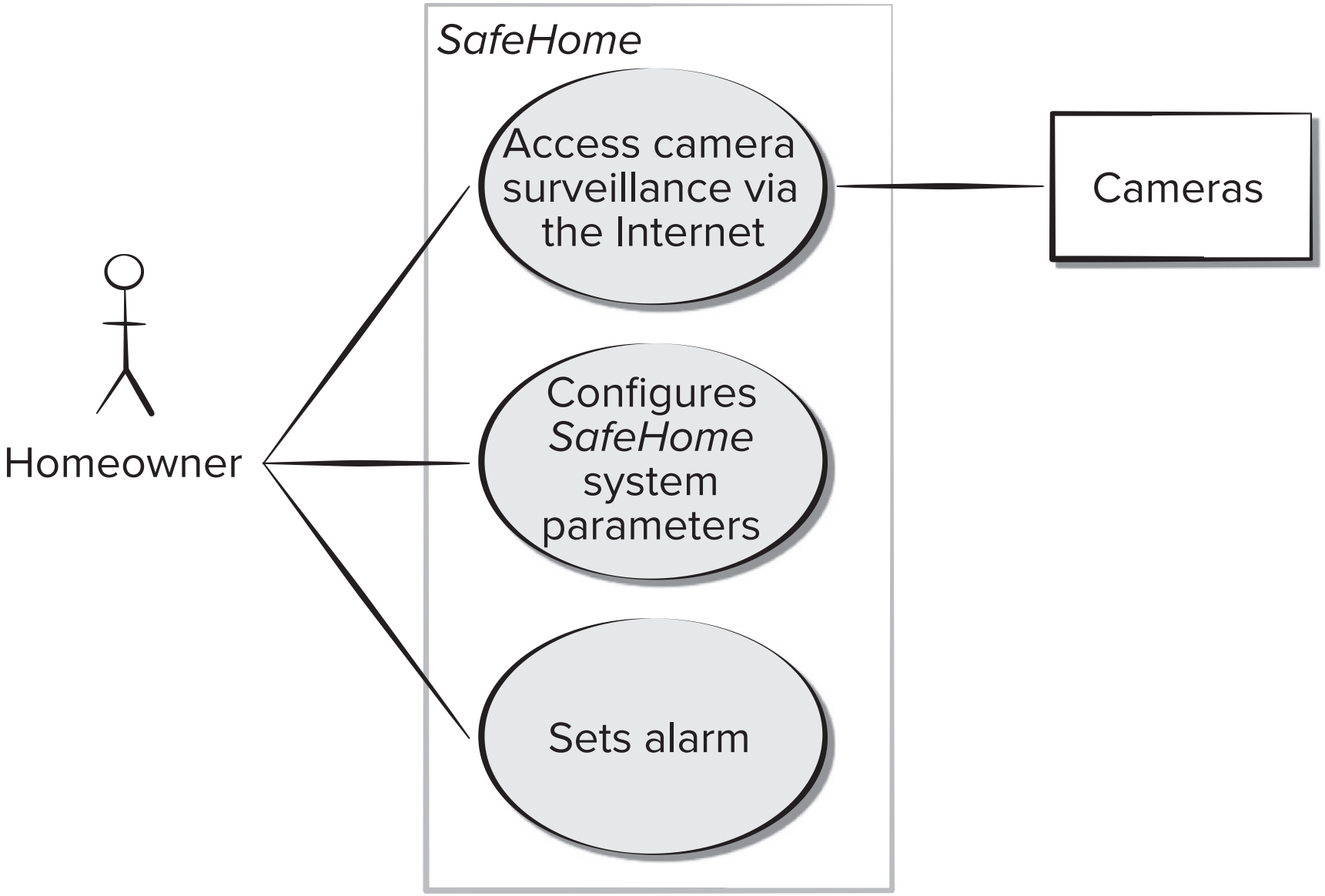
Channels to secondary actors:

1. System administrator: PC-based system.
2. Cameras: wireless connectivity.

Open issues:

1. What mechanisms protect unauthorized use of this capability by employees of *SafeHome Products*?
2. Is security sufficient? Hacking into this feature would represent a major invasion of privacy.
3. Will system response via the Internet be acceptable given the bandwidth required for camera views?
4. Will we develop a capability to provide video at a higher frames-per-second rate when high-bandwidth connections are available?

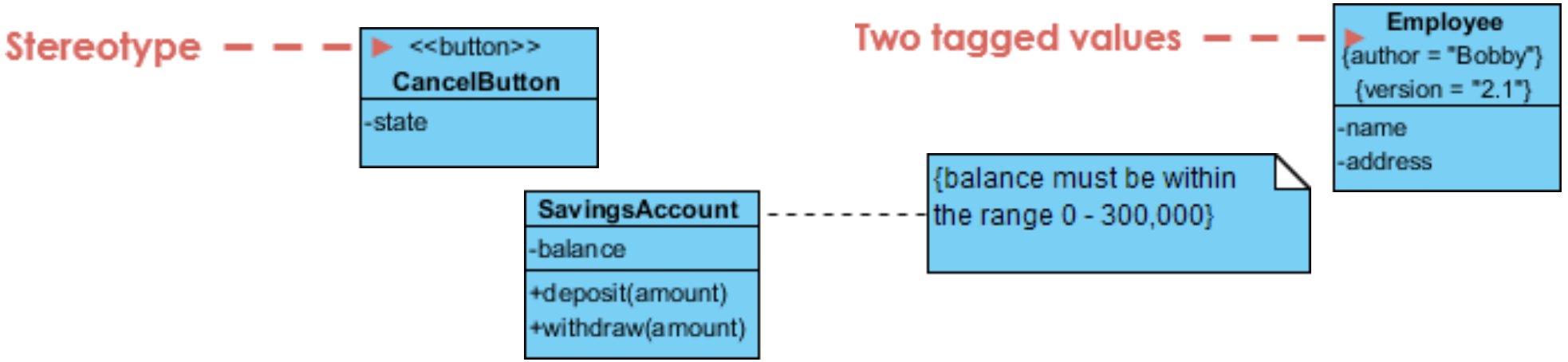
Example: Use Case Diagram



Actors and User Profiles

- A UML actor models an entity that interacts with a system object.

- A UML profile provides a way of extending an existing model to other domains or platforms.



Use Case

- Each modeling notation has limitations, and the UML use case is no exception.
- Like any other form of written description, a use case is only as good as its author(s). If the description is unclear, the use case can be misleading or ambiguous.
- A use case focuses on functional and behavioral requirements and is generally inappropriate for nonfunctional requirements.
- For situations in which the requirements model must have significant detail and precision (e.g., safety critical systems), a use case may not be sufficient.
- However, scenario-based modeling is appropriate for a significant majority of all situations that you will encounter as a software engineer.
- If developed properly, the use case can provide substantial benefit as a modeling tool.

2 Class-Based Modeling

Identifying Analysis Classes

- We can begin to identify classes by
  - examining the usage scenarios developed as part of the requirements model
  - 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 table.
  - Synonyms should be noted.
  - If the class (noun) is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a solution, it is part of the problem space.
- But what should we look for once all the nouns have been isolated?

Manifestations of 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.

Example: Defining Analysis Classes

The SafeHome security function *enables* the homeowner to *configure* the security system when it is *installed, monitors* all sensors *connected* to the security system, and *interacts* with the homeowner through the Internet, a PC or a control panel.

During installation, the SafeHome PC is used to *program* and *configure* the system. Each sensor is assigned a number and type, a master password is programmed for *arming* and *disarming* the system, and telephone number(s) are *input* for *dialing* when a sensor event occurs.

When a sensor event is *recognized*, the software *invokes* an audible alarm attached to the system. After a delay time that is *specified* by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, *provides* information about the location, *reporting* the nature of the event that has been detected. The telephone number will be *redialed* every 20 seconds until telephone connection is *obtained.*

The homeowner *receives* security information via a control panel, the PC, or a browser, collectively called an interface. The interface *displays* prompting messages and system status information on the control panel, the PC, or the browser window. Homeowner interaction takes the following form . . .

Example: Potential Classes

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

Selection characteristics of Potential Class

- Selection characteristics of 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.

Example: Decision for inclusion of potential classes

Potential Class	Characteristic Number That Applies
homeowner	rejected: 1, 2 fail even though 6 applies
sensor	accepted: all apply
control panel	accepted: all apply
installation	rejected
system (alias security function)	accepted: all apply
number, type	rejected: 3 fails, attributes of sensor
master password	rejected: 3 fails
telephone number	rejected: 3 fails
sensor event	accepted: all apply
audible alarm	accepted: 2, 3, 4, 5, 6 apply
monitoring service	rejected: 1, 2 fail even though 6 applies

Defining Attributes

- **Attributes** describe a class that has been selected for inclusion in the analysis model.
- 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.
- In addition, the following question should be answered for each class:
  - What data items (composite and/or elementary) fully define this class in the context of the problem at hand?

Example

- Build two different classes for professional baseball players
  - For Playing Statistics software: name, position, batting average, fielding percentage, years played, and games played might be relevant
  - For Pension Fund software: name, average salary, credit toward full vesting, pension plan options chosen, mailing address, and the like.

Example

- A homeowner can configure the security function to reflect sensor information, alarm response information, activation and deactivation information, identification information, and so forth.
- We can represent these composite data items in the following manner:
  - identification information = system ID + verification phone number + system status
  - alarm response information = delay time + telephone number
  - activation/deactivation information = master password + number of allowable tries + temporary password

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
  4. Operations that monitor an object for the occurrence of a controlling event
- These functions are accomplished by operating on attributes and/or associations.
- Therefore, an operation must have "knowledge" of the nature of the class attributes and associations.

Defining Operations

- As a first iteration at deriving a set of operations for an analysis class, you can again study a processing narrative (or use case) and select those operations that reasonably belong to the class.
- To accomplish this, the grammatical parse is again studied, and *verbs* are isolated.
- Some of these verbs will be legitimate operations and can be easily connected to a specific class.

Example

- "sensor is *assigned* a number and type"
- "a master password is *programmed* for *arming* and *disarming* the system."

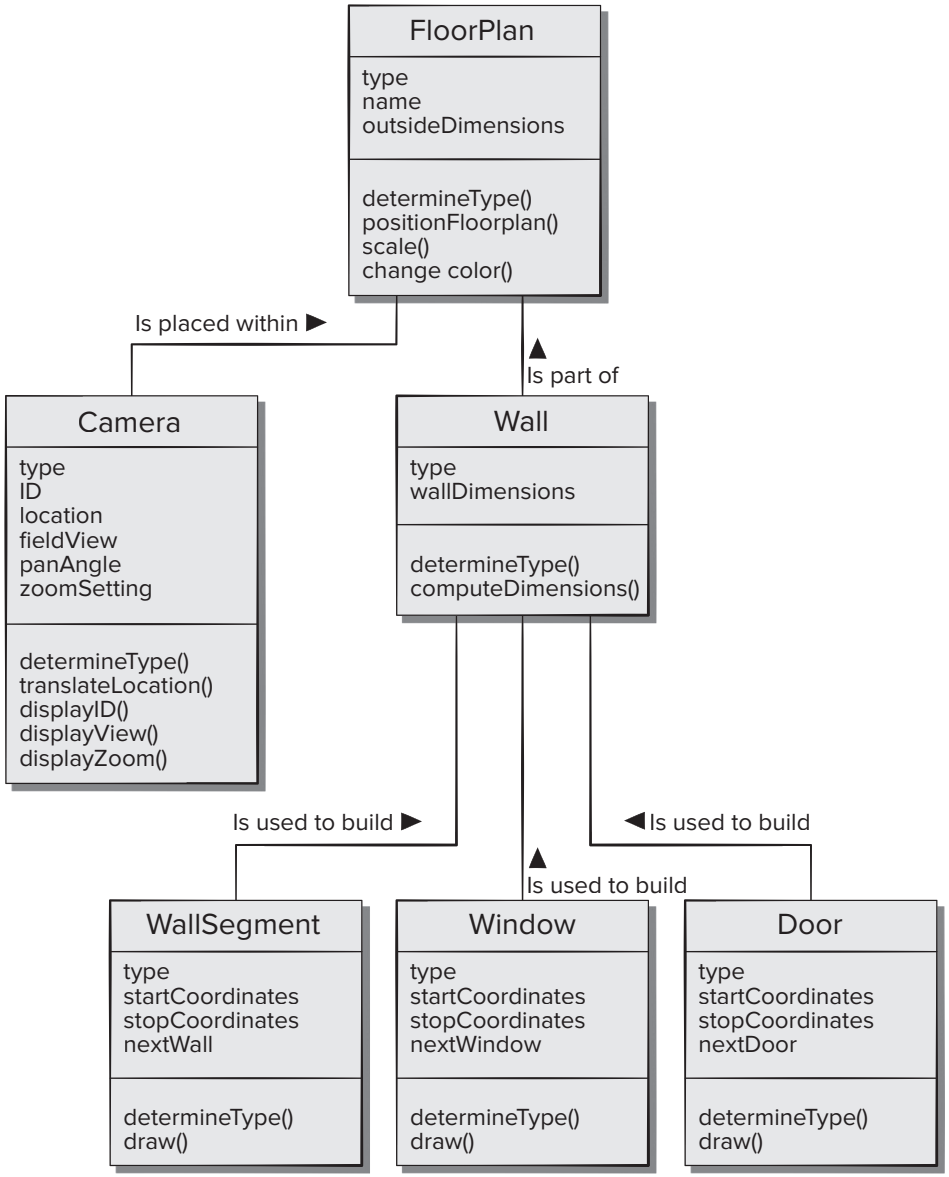
- These phrases indicate several things:
  - That an *assign()* operation is relevant for the Sensor class.
  - That a *program()* operation will be applied to the System class.
  - That *arm()* and *disarm()* are operations that apply to System class.

Association

- In many instances, two analysis classes are related to one another in some fashion.
- In UML, these relationships are called *associations*.
- An association describes a collection of links with common semantics.

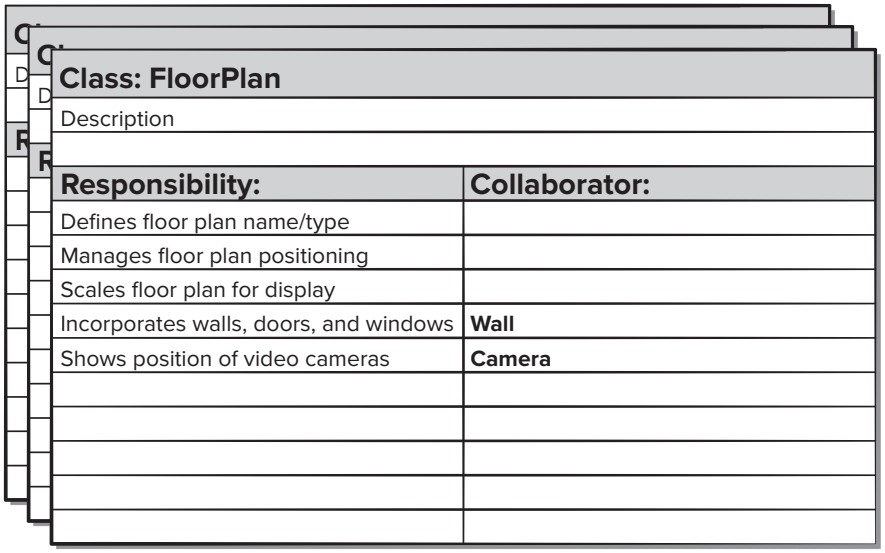
Example: Association

- When the homeowner wants to pick a camera, he or she must pick it from a floor plan.
- FloorPlan is an object that is put together with walls, doors, windows, and cameras.



Class-Responsibility-Collaborator Modeling

- Class-responsibility-collaborator (CRC) modeling provides a simple means for identifying and organizing the classes that are relevant to system or product requirements.
- A CRC model can be viewed as a collection of index cards.

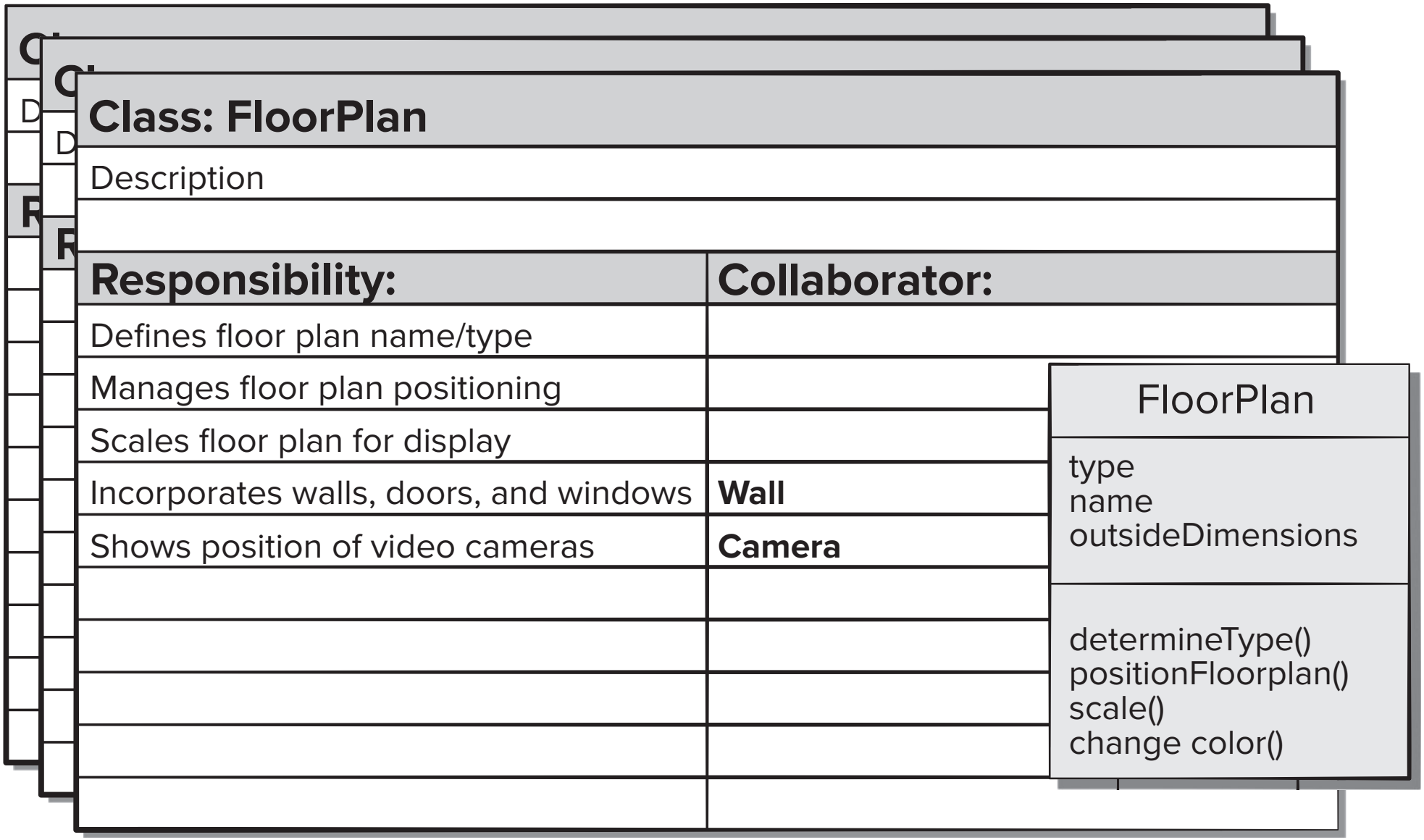


Class: FloorPlan	
Description	
Responsibility:	**Collaborator:**
Defines floor plan name/type	
Manages floor plan positioning	
Scales floor plan for display	
Incorporates walls, doors, and windows	**Wall**
Shows position of video cameras	**Camera**

Class-Responsibility-Collaborator Modeling

- **Classes**
  - A class can use its own operations to manipulate its own attributes, thereby fulfilling a responsibility itself
  - a class can collaborate with other classes.
- **Responsibilities**
  - Responsibilities are the *attributes* and *operations* that are relevant for the class.
- **Collaborations.**
  - Collaborators are those classes that provide a class with the information needed or action required to complete a responsibility.
  - Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class.

Example: A CRC model index card



Review CRC model

- Once a complete CRC model has been developed, the representatives from the 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. No reviewer should have two cards that collaborate.
  2. 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.
  3. When the token is passed, the holder of the class 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.
  4. If an error is found, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or revising lists of responsibilities or collaborations on existing cards.

3 Functional Modeling

Functional Model

- The functional model addresses two application processing elements, each representing a different level of procedural abstraction:
  1. user-observable functionality that is delivered by the app to end users
     - User-observable functionality encompasses any processing functions that are initiated directly by the user.
  2. the operations contained within analysis classes that implement behaviors associated with the class
     - At a lower level of procedural abstraction, the requirements model describes the processing to be performed by analysis class operations.
     - These operations manipulate class attributes and are involved as classes collaborate with one another to accomplish some required behavior.

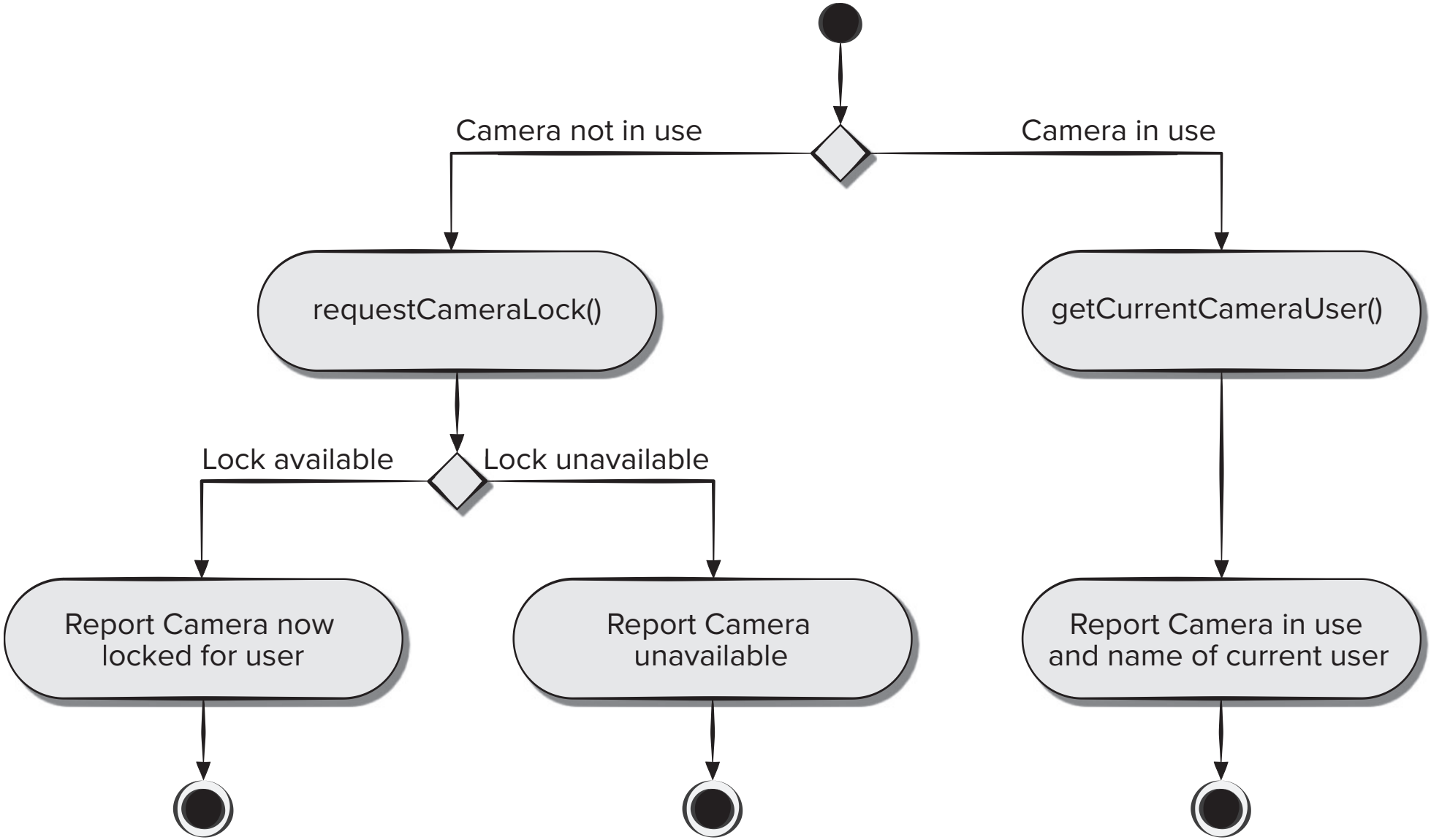
A Procedural View

- Regardless of the level of procedural abstraction, the UML activity diagram can be used to represent processing details.
- At the analysis level, activity diagrams should be used only where the functionality is relatively complex.
- The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario.

Example

- **Control cameras**
- The interaction is relatively simple, but there is the potential for complex functionality, given that this "simple" operation requires complex communication with devices located remotely and accessible across the Internet.
- A further possible complication relates to negotiation of control when multiple authorized people attempt to monitor and/or control a single sensor at the same time.

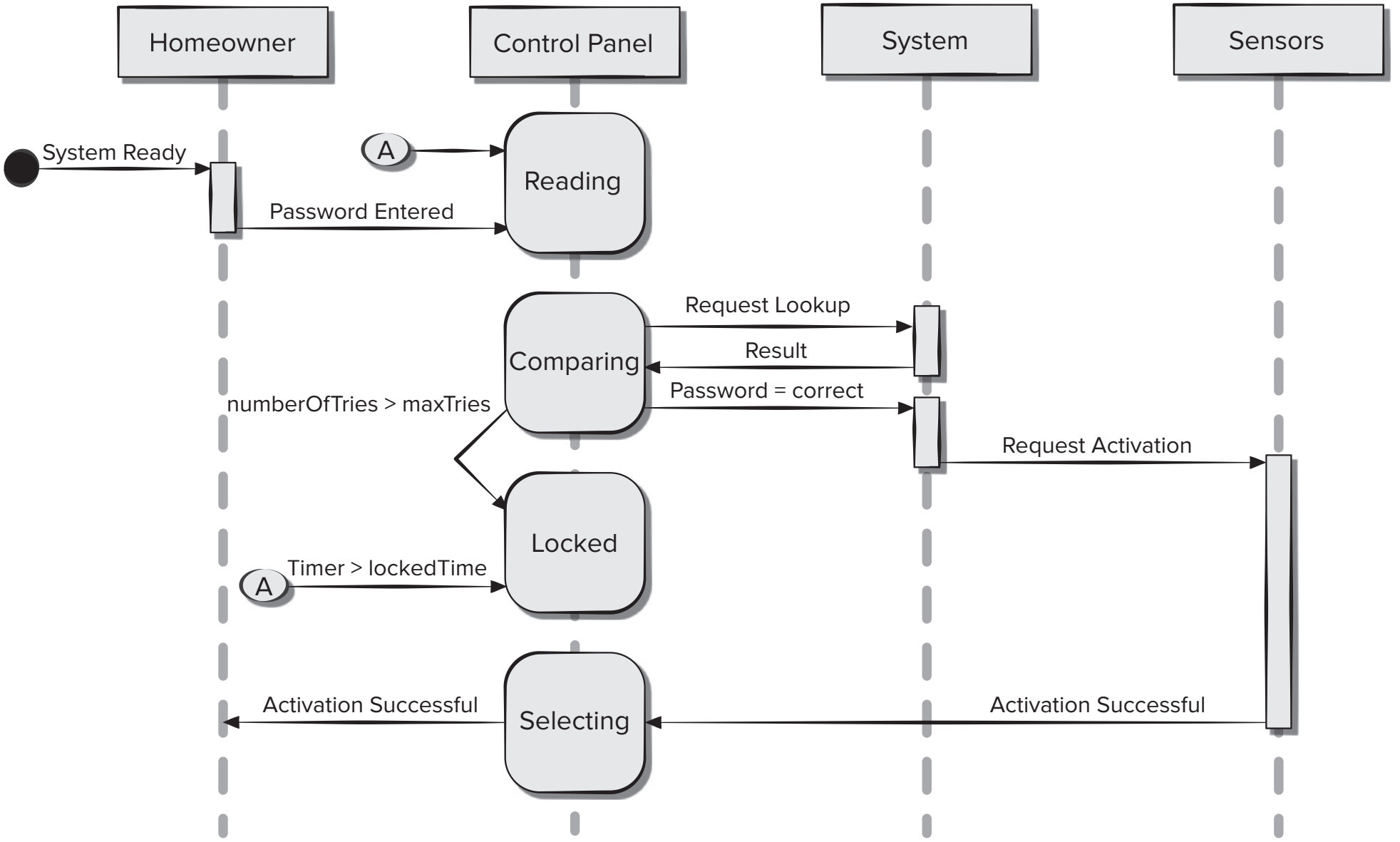
Example: Activity Diagram



UML Sequence Diagrams

- The UML sequence diagram can be used for behavioral modeling. Sequence diagrams can also be used to show how events cause transitions from object to object.
- Once events have been identified by examining a use case, the modeler creates a sequence diagram
- 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.
- Once a complete sequence diagram has been developed, all the events that cause transitions between system objects can be collated into a set of input events and output events (from an object).
- This information is useful in the creation of an effective design for the system to be built.

Example: Sequence Diagram



4 Behavioral Modeling

Behavioral Model

- A **behavioral** model indicates how software will respond to internal or external events or stimuli.
- This information is useful in the creation of an effective design for the system to be built.
- UML *activity* diagrams can be used to model how system elements respond to internal events.
- UML *state* diagrams can be used to model how system elements respond to external events.

Create Behavioral Model

- 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

- In general, an event occurs whenever the system and an actor exchange information.
- An event is not the information that has been exchanged, but rather the fact that information has been exchanged.
- A use case is examined for points of information exchange.

Example

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.
  - Any conditions or constraints should be listed.

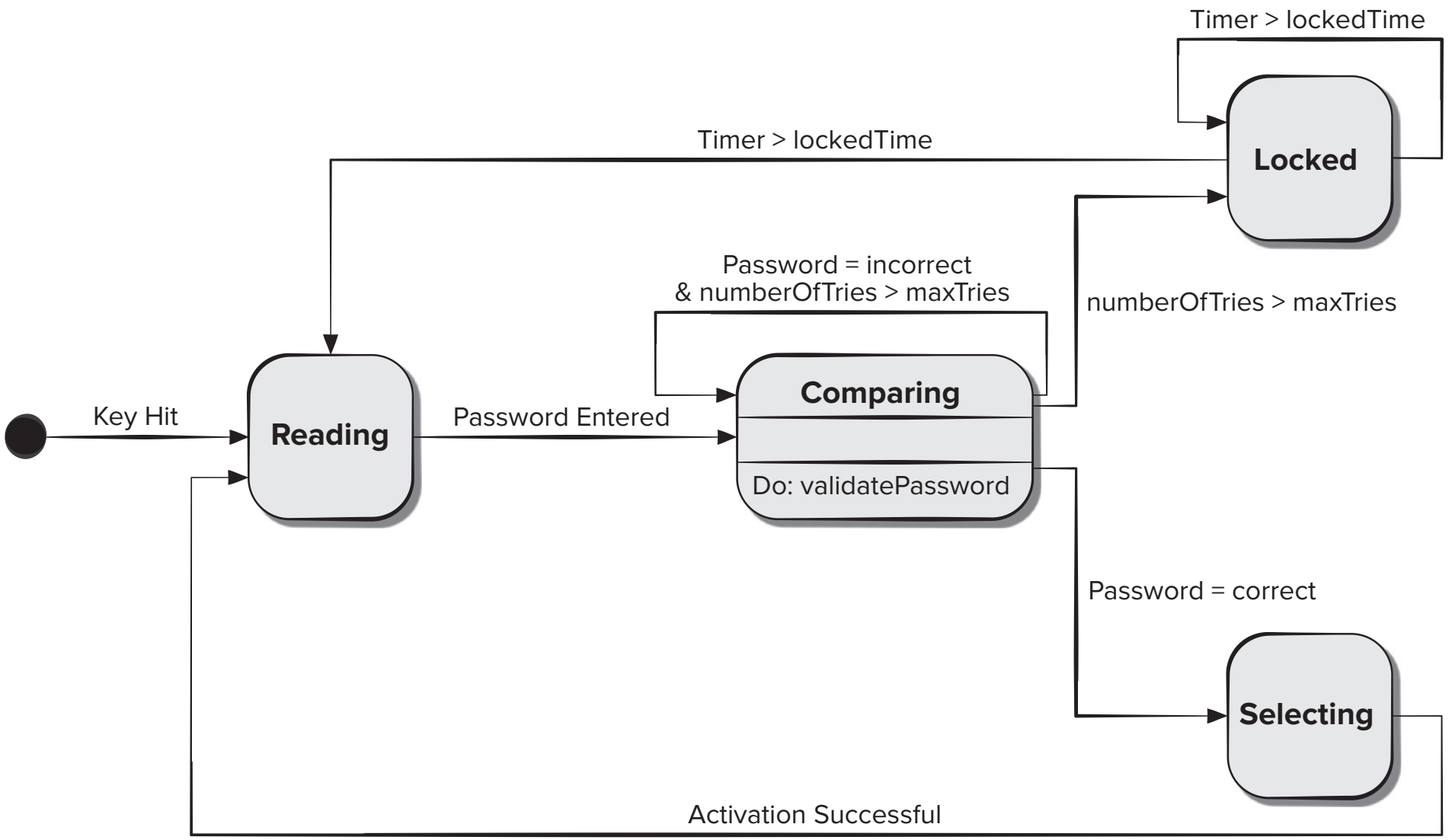
Objects

- Once all events have been identified, they are allocated to the objects involved.
- Objects can be responsible for
  - generating events
    (e.g., Homeowner generates the password entered event)
  - recognizing events that have occurred elsewhere
    (e.g., ControlPanel recognizes the binary result of the password compared event).

UML State Diagrams

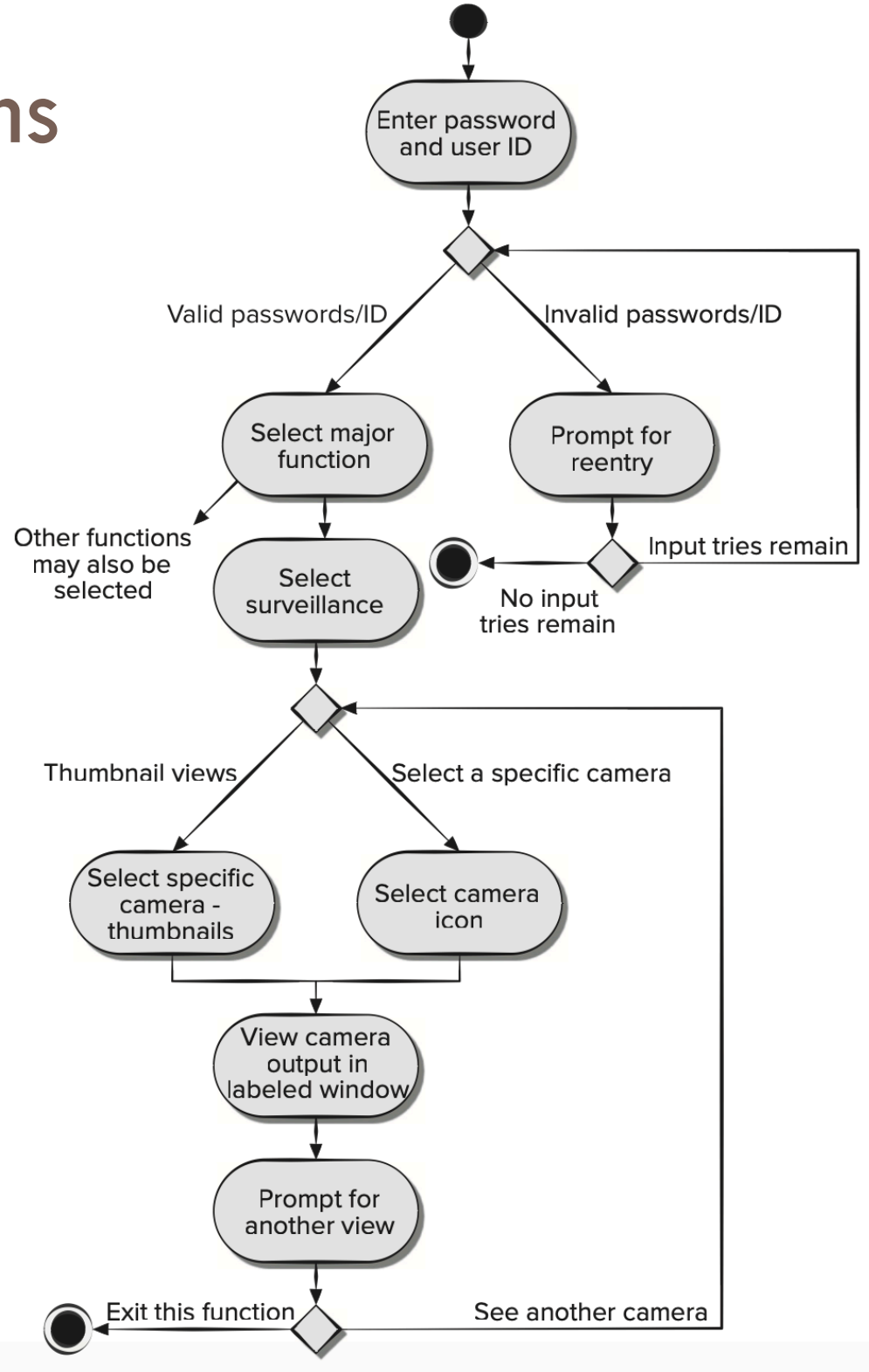
- In the context of behavioral modeling, two different characterizations of states must be considered:
  1. The state of each class as the system performs its function
  2. The state of the system as observed from the outside as the system performs its function.
- The state of a class takes on both passive and active characteristics.
  - A *passive* state is simply the current values assigned to an object's attributes.
  - An *active* state of an object indicates the status of the object as it undergoes a continuing transformation or processing.
    - An *event* (sometimes called a *trigger*) must occur to force an object to make a transition from one active state to another.
- 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.

Example: State diagram



UML Activity Diagrams

- The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario.
- Many software engineers like to describe activity diagrams as a way of representing how a system reacts to internal events.



UML Swimlane Diagram

- 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 (if there are multiple actors involved in a specific use case) 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.

Example: Swimlane Diagram