Requirements Engineering

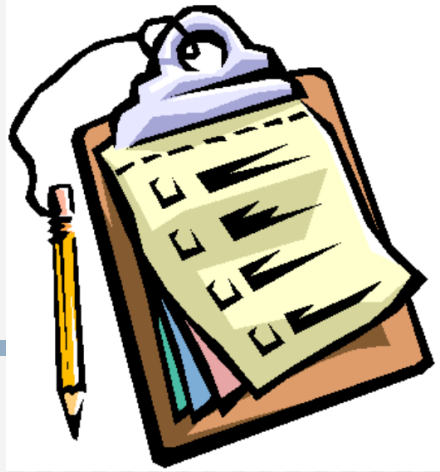
- **Modeling: Chapters 8-11**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 8/e

by Roger S. Pressman and Bruce R. Maxim

Agenda



- Understanding Requirements
- Requirements Modeling
  - Scenario-Based Methods
  - Class-Based Methods
  - Behavioral Methods
  - Requirements Modeling for Web and Mobile Apps

Understanding Requirements

Requirements Engineering-I

- Inception—ask a set of questions that establish …
  - basic understanding of the problem
  - the people who want a solution
  - the nature of the solution that is desired, and
  - effective communication and collaboration between the customer and the developer
- Elicitation—elicit requirements from all stakeholders
- Elaboration—expand and refine the information obtained during inception and elicitation
  - **Requirements modeling**: create an analysis model that identifies data, function and behavioral requirements
- Negotiation—agree on a deliverable system that is realistic for developers and customers

Requirements Engineering-II

- Specification—can be any one (or more) of the following:
  - A written document
  - A set of models
  - A formal mathematical
  - A collection of user scenarios (use-cases)
  - A prototype
- Validation—a review mechanism that looks 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 or unrealistic (unachievable) requirements.
- Requirements management— control and track changes to requirements at any time
  - Many of these tasks are supported by the software configuration management (SCM)

A Sample Template for Requirements Specification

INFO

Software Requirements Specification Template

A *software requirements specification* (SRS) is a work product that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified.

Karl Wiegers [Wie03] of Process Impact Inc. has developed a worthwhile template (available at **www.processimpact.com/process_assets/srs_template.doc**) that can serve as a guideline for those who must create a complete SRS. A topic outline follows:

Table of Contents

Revision History

1. **Introduction**
   - 1.1 Purpose
   - 1.2 Document Conventions
   - 1.3 Intended Audience and Reading Suggestions
   - 1.4 Project Scope
   - 1.5 References
2. **Overall Description**
   - 2.1 Product Perspective
   - 2.2 Product Features
   - 2.3 User Classes and Characteristics
   - 2.4 Operating Environment
   - 2.5 Design and Implementation Constraints
   - 2.6 User Documentation
   - 2.7 Assumptions and Dependencies
3. **System Features**
   - 3.1 System Feature 1
   - 3.2 System Feature 2 (and so on)
4. **External Interface Requirements**
   - 4.1 User Interfaces
   - 4.2 Hardware Interfaces
   - 4.3 Software Interfaces
   - 4.4 Communications Interfaces
5. **Other Nonfunctional Requirements**
   - 5.1 Performance Requirements
   - 5.2 Safety Requirements
   - 5.3 Security Requirements
   - 5.4 Software Quality Attributes
6. **Other Requirements**

Appendix A: Glossary

Appendix B: Analysis Models

Appendix C: Issues List

A detailed description of each SRS topic can be obtained by downloading the SRS template at the URL noted in this sidebar.

A Sample Checklist for Requirements Validation

INFO

Requirements Validation Checklist

It is often useful to examine each requirement against a set of checklist questions. Here is a small subset of those that might be asked:

- Are requirements stated clearly? Can they be misinterpreted?
- Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?
- Is the requirement bounded in quantitative terms?
- What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?
- Does the requirement violate any system domain constraints?
- Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?
- Is the requirement traceable to any system model that has been created?
- Is the requirement traceable to overall system/product objectives?
- Is the specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
- Has an index for the specification been created?
- Have requirements associated with performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

Inception

- Identify stakeholders
  - “who else do you think I should talk to?”
- Recognize multiple points of view
- Work toward collaboration
- The first questions
  - 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?
    - possible alternatives to custom software development

Eliciting Requirements

- meetings are conducted and attended by both software engineers and customers
- rules for preparation and participation are established
- an agenda is suggested
- a "facilitator" (can be a customer, a developer, or an outsider) controls the meeting
- the goal is
  - to explore the problem
  - propose elements of the solution
  - specify a preliminary set of solution requirements

Quality Function Deployment

- QFD is a quality management technique that translates unspoken customer needs into system requirements
- QFD emphasizes an understanding of what is **valuable** to the customer and then deploys these values throughout the engineering process

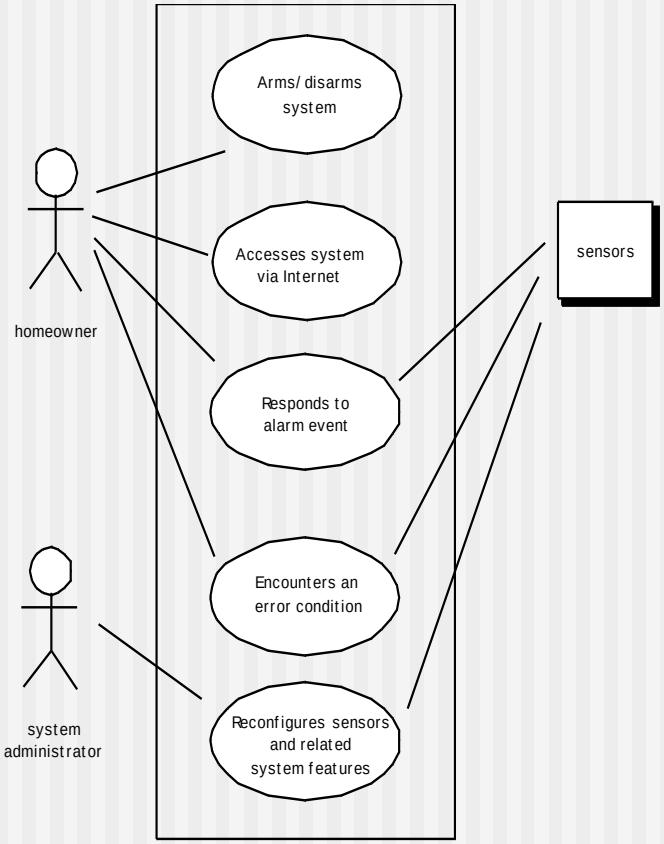
1. Normal requirements—objectives and goals that are stated for a product to satisfy customers
2. Expected requirements—implicit requirements that their absence will be a cause for significant dissatisfaction
3. Exciting requirements—beyond customer expectations and prove to be very satisfying when present

- Customer voice table

Usage Scenarios (Use-Cases)

- As requirements are gathered, an overall vision of system functions and features begin to materialize
- However, it is difficult to move into more technical SE activities until you understand how these functions and features will be used by different classes of users
- To do this, developers and users can create a set of scenarios that identify a thread of usage for the system
- The scenarios, often called use cases
  - provide a description of how the system will be used
- Each scenario is described from the point-of-view of an “actor”—a person or device that interacts with the software in some way

Use-Case Diagram



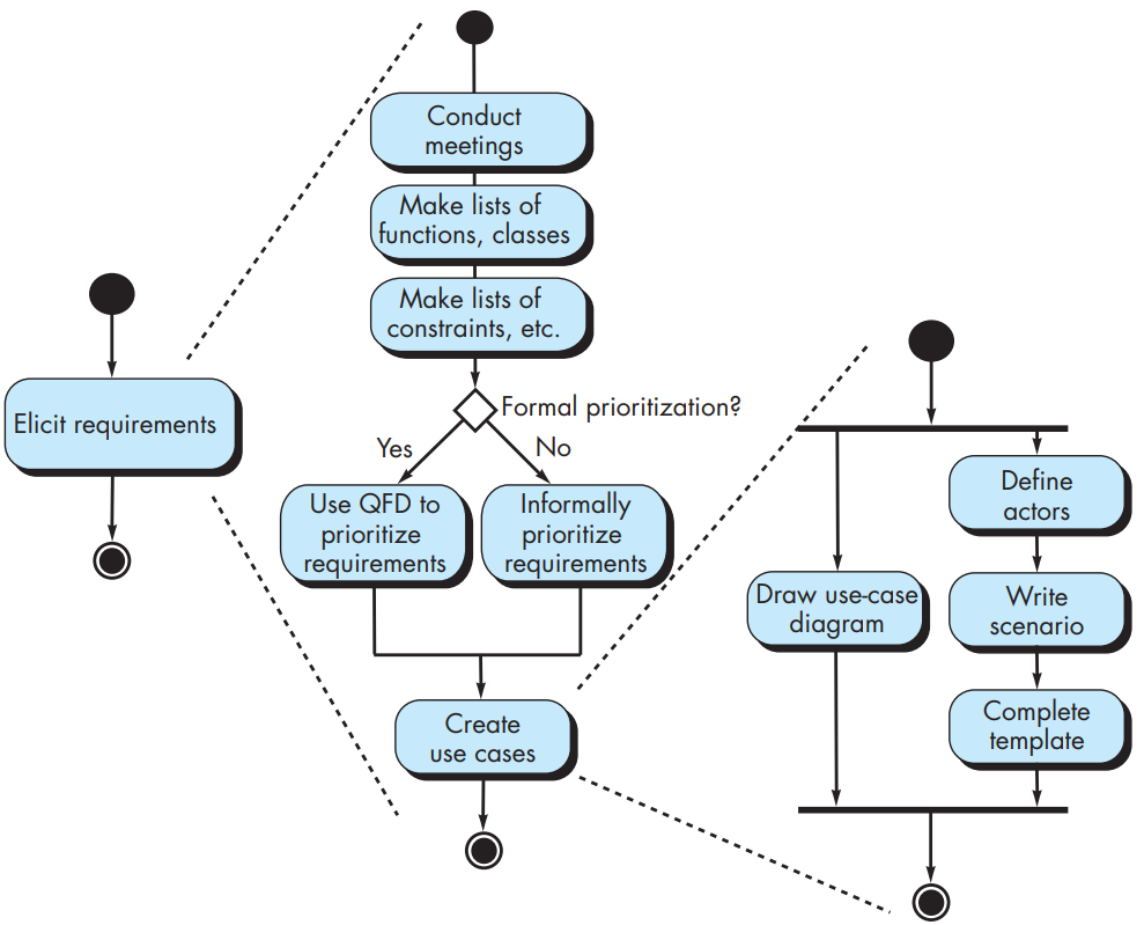
Non-Functional Requirements

- Non-Functional Requirement (NFR) – quality attribute, performance attribute, security attribute, or general system constraint. A two phase process is used to determine which NFR's are compatible:
  - The first phase is to create a matrix using NFRs as column labels and SE guidelines as row labels
    - classifying each NFR and guideline pair as complementary, overlapping, conflicting, or independent
  - The second phase is for the team to prioritize and create a homogeneous set of NFRs using a set of rules to decide which to implement

Agile Requirements Elicitation

- Within the context of an agile process, requirements are elicited by asking all stakeholders to create user stories
- Although the agile approach to requirements elicitation is attractive for many software teams, critics argue that:
  - overall business goals and nonfunctional requirements are often lacking
- In some cases, rework is required to accommodate performance and security issues
- In addition, user stories may not provide a sufficient basis for system evolution over time

Activity Diagram of Requirements Elicitation



Building the Analysis Model

- Intent: provides a description of the required **informational**, **functional**, and **behavioral** domains of a system
- a snapshot of requirements at any given time
- changes dynamically as you learn more about the system
- As the analysis model evolves, certain elements will become relatively stable
  - providing a solid foundation for the design tasks that follow

Elements of the Analysis Model

- Scenario-based elements
  - Use-case—descriptions of the interaction between an “actor” and the system
  - Activity Diagram—processing narratives for software functions
- Class-based elements
  - Implied by scenarios
- Behavioral elements
  - State diagram

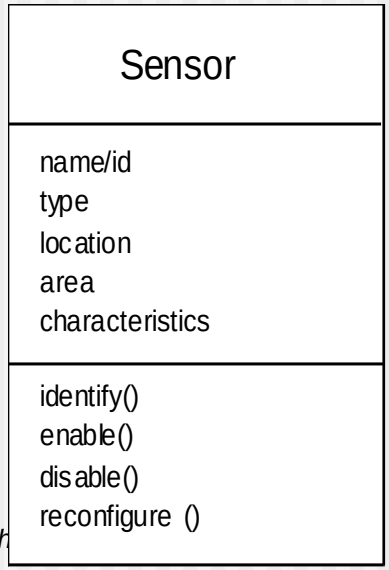
Scenario-Based Elements

- The system is described from the user's point of view using a scenario
- Often the first part of the analysis model
- Serve as input for the creation of other modeling elements
- Use-case diagram
- Use-case description
- Activity diagram

Class-Based Elements

- Each usage scenario implies a set of objects that are manipulated during the scenario
- UML class diagram depicts the structures of classes

From the *SafeHome* system …



Behavioral Elements

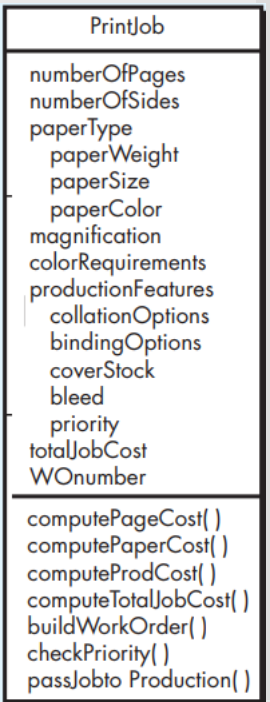
- The behavior of a system can have a profound effect on the design that is chosen and the implementation approach
- The state diagram is **one method** for representing the behavior of a system
  - Depicting its states and the events that cause the system to change state
- **State diagram**
- Events (or transitions) represented by:

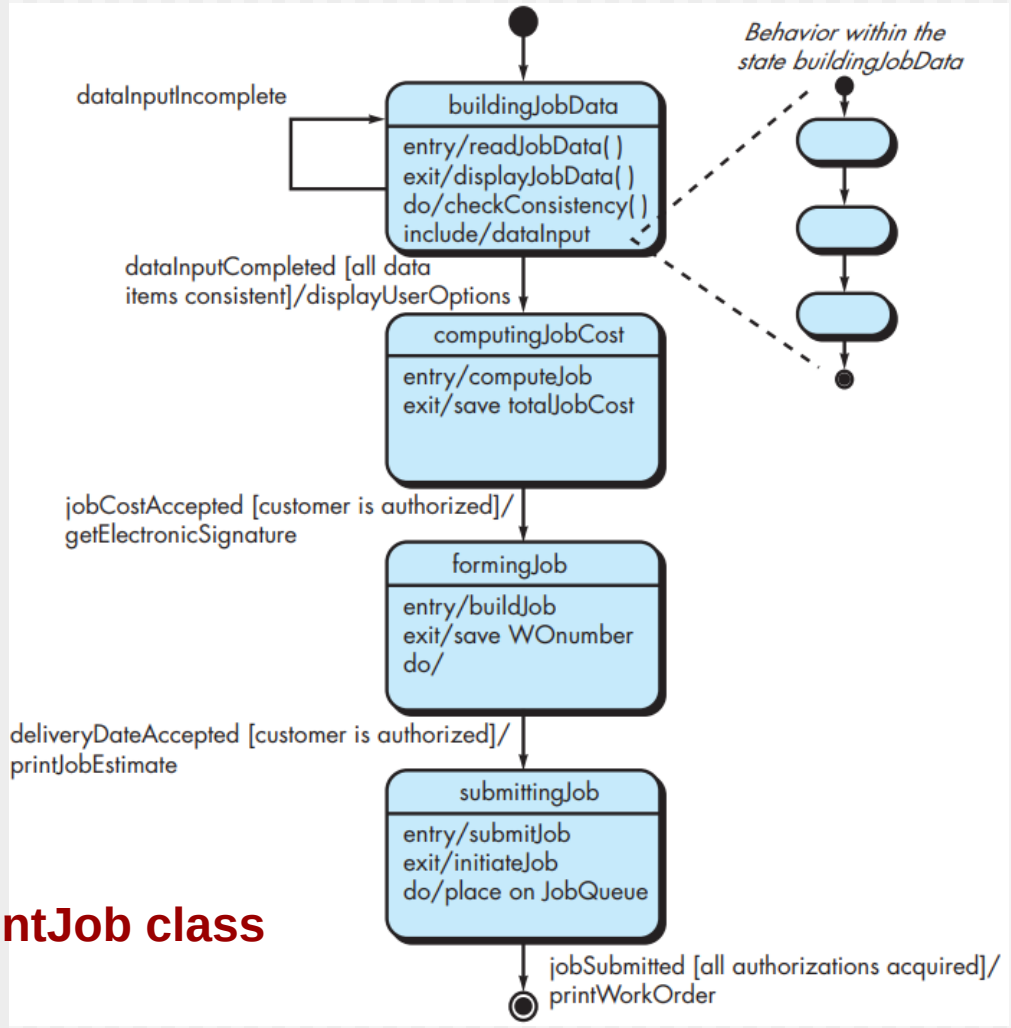
  Event-name [guard-condition] / action

- Each state may define the following actions:
  - *entry/*, exit/, *do/*, and *include/*

State Diagram Example

Consider a software to be built for a sophisticated print shop. The overall intent of the software is to collect the customer's requirements at the front counter, cost a print job, and then pass the job on to an automated production facility





Statechart for PrintJob class

Negotiating Requirements

- Asking stakeholders to balance functionality, performance, and other requirements against cost and time-to-market
- The best negotiations strive for a "**win-win**" result
- Stakeholders win by getting a 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

Negotiation Activities

- Identify the key stakeholders
  - These are the people who will be involved in the negotiation
- Determine each of the stakeholders “win conditions”
  - Win conditions are not always obvious
- Negotiate
  - Work toward a set of requirements that lead to “win-win”

Validating Requirements - I

- A review of the requirements model addresses the following questions:
  - Is each requirement consistent with the overall objective 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?
  - Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
  - Is each requirement bounded and unambiguous?

Validating Requirements - II

- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- 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?
- ….

Requirements Monitoring

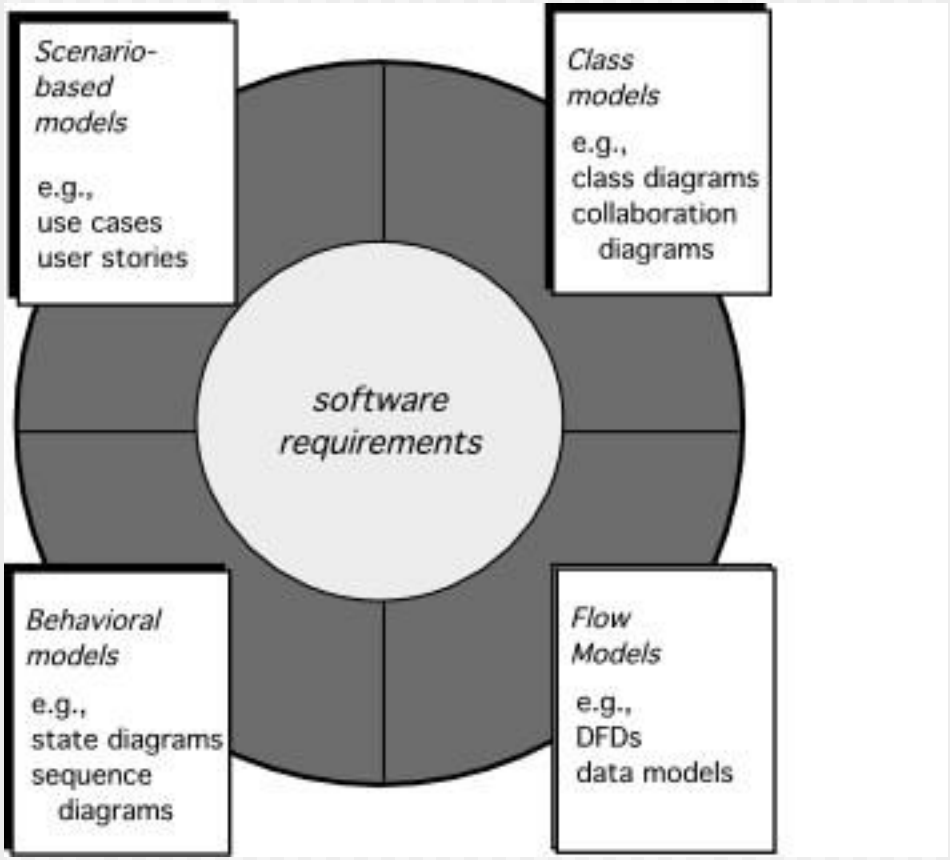
- Extremely needed in **incremental** development
- Incremental development implies the need for incremental validation
- Requirements monitoring supports continuous validation
  - *Run-time verification* – determines whether software matches its specification.
  - *Run-time validation* – assesses whether evolving software meets user goals.
  - *Distributed debugging* – uncovers the root cause of errors.
  - *Business activity monitoring* – evaluates whether a system satisfies business goals.
  - *Evolution and co-design* – provides information to stakeholders as the system evolves.

Requirements Modeling

Requirements Analysis

- Requirements analysis allows the software engineer (called an *analyst* or *modeler* in this role) to:
  - elaborate on basic requirements established during earlier requirement engineering tasks
  - build models that depict
    - user scenarios
    - functional activities
    - problem classes and their relationships
    - system and class behavior
    - and the flow of data as it is transformed

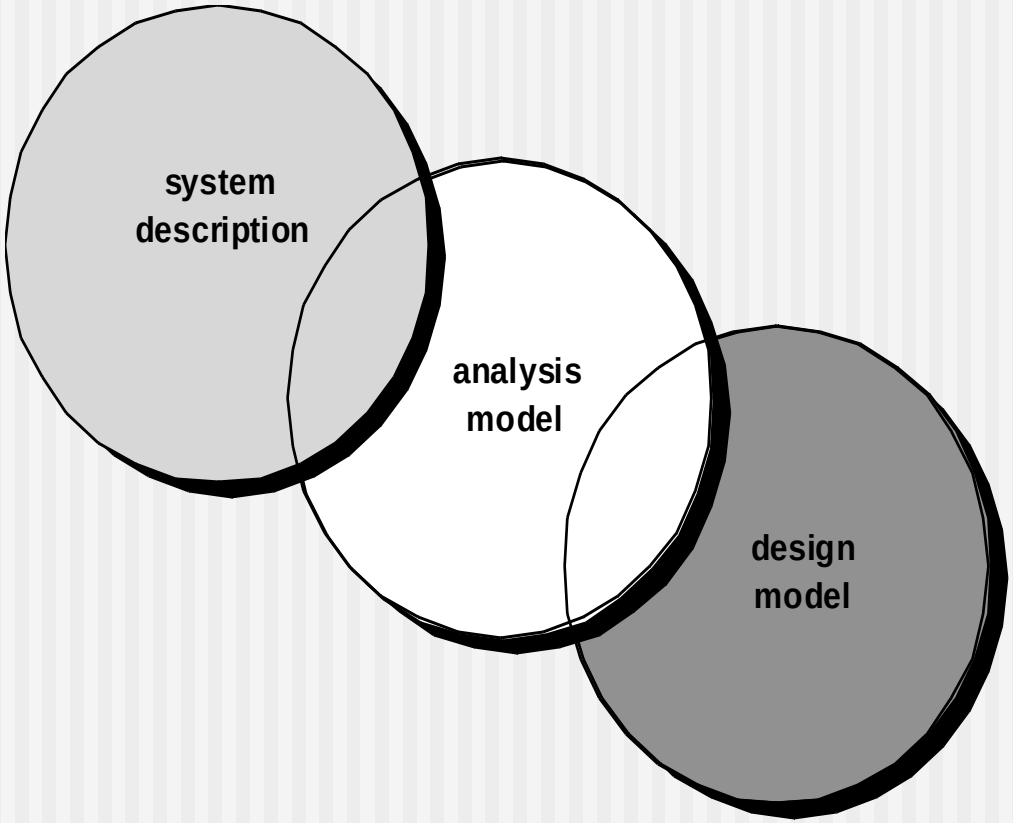
Elements of Requirements Analysis



Requirements Modeling

- Scenario-based models
  - Depict the system from the user's point of view
- Class-oriented models
  - Define classes (attributes and operations) and their relationships
- Behavioral models
  - Show how the software behaves as a consequence of external "events"
- Data models
  - Depict the information domain for the problem
- Flow-oriented models
  - Show how data are transformed inside the system

A Bridge



Domain Analysis

- Some analysis problems often reoccur across many applications within a specific business domain

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 . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks . . .

Donald Firesmith

Domain Analysis

- Application domain can range from avionics to banking, from multimedia video games to software embedded within medical devices
- The goal of domain analysis:
  - find or create those analysis classes that are broadly applicable so that they may be reused
- Steps:
  - Define the domain to be investigated.
  - Collect a representative sample of applications in the domain.
  - Analyze each application in the sample.
  - Develop an analysis model for the objects.

Requirements Modeling: Scenario-Based Methods

Scenario-Based Modeling

"[Use-cases] are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use-cases)."

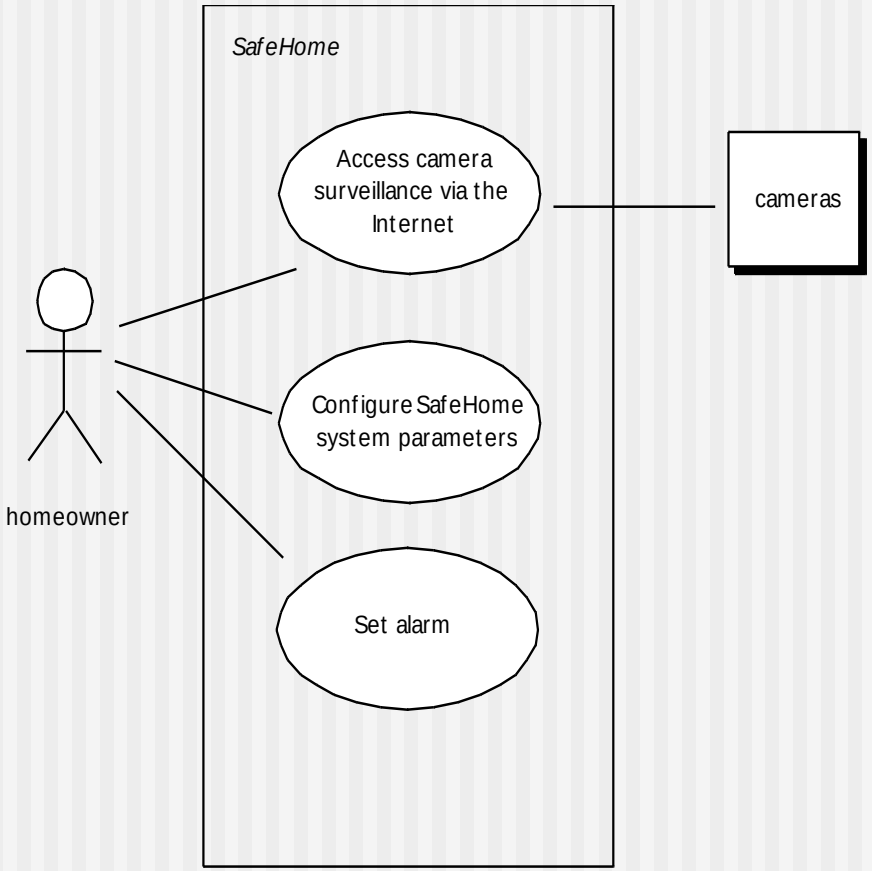
Ivar Jacobson

- Typical elements for scenario-based modeling:
  - Use-case diagram
  - Use-case description
  - Activity diagram

Developing a Use-Case

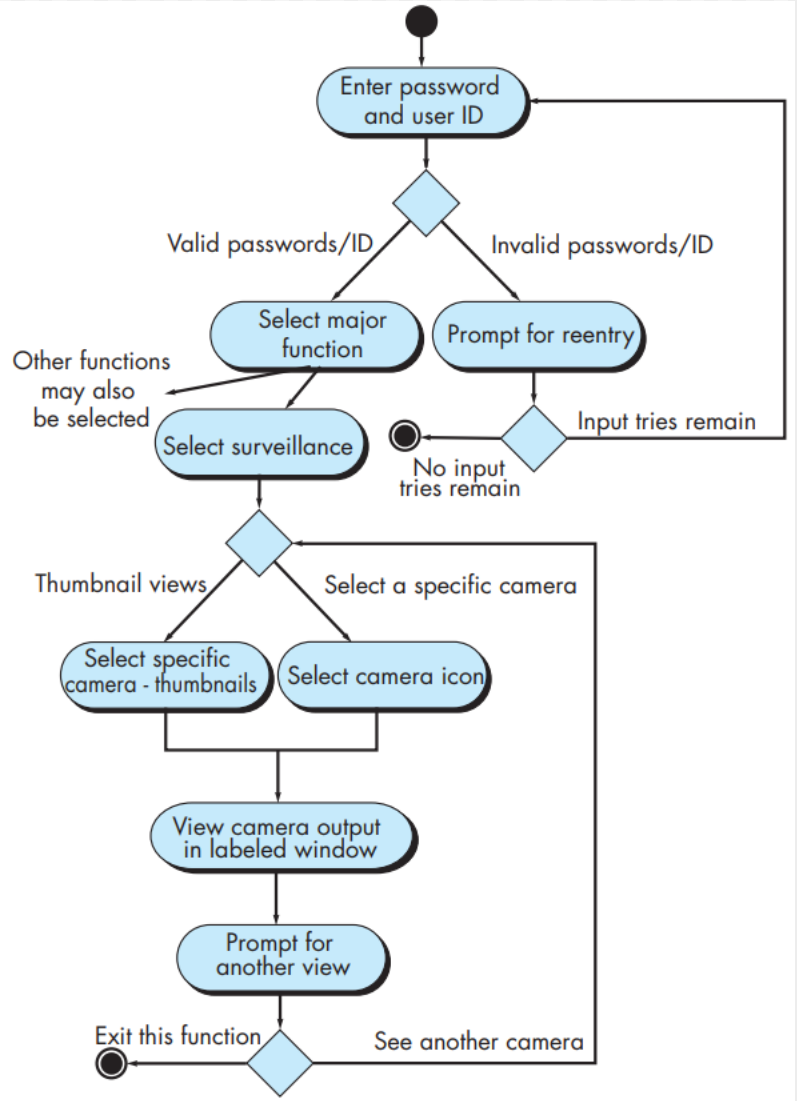
- When looking for use cases, ask the following questions:
  - What are the main tasks of the actor?
  - What information does the actor need from the system?
  - What information does the actor provide to the system?
  - Does the system need to inform the actor of any changes or events that have occurred?
  - Does the actor need to inform the system of any changes or events that have occurred?

Use-Case Diagram



Activity Diagram

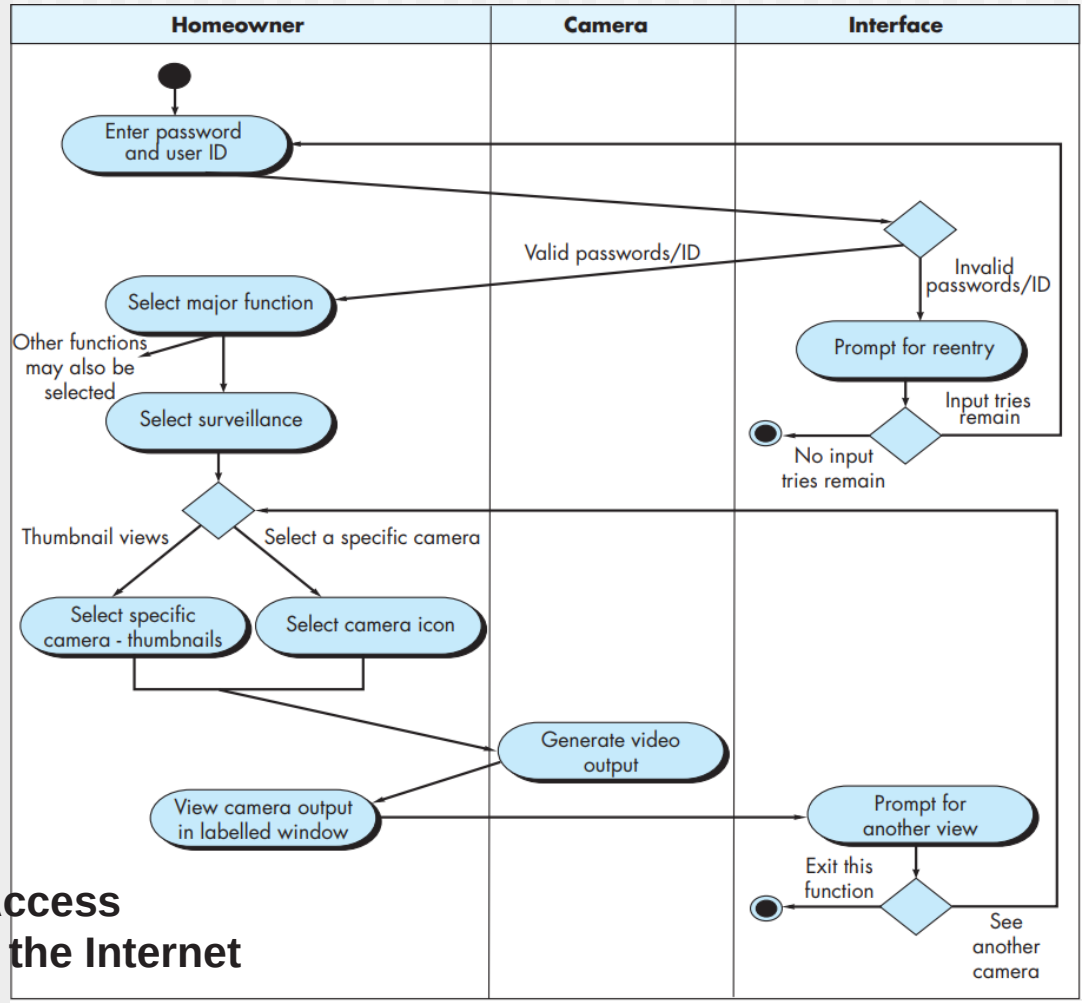
Supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario



Activity diagram for Access camera surveillance via the Internet—display camera view function

Swimlane Diagrams

A variation of the activity diagram that allows the modeler 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



Swimlane diagram for Access camera surveillance via the Internet

Requirements Modeling: Class-Based Methods

Class-Based Modeling

- Class-based modeling represents:
  - objects that the system will manipulate
  - operations (also called methods or services) that will be applied to the objects to effect the manipulation
  - relationships (some hierarchical) between the objects
  - collaborations that occur between the classes that are defined.
- The elements of a class-based model include classes and objects, attributes, operations, CRC models, collaboration diagrams and packages.

Identifying Analysis Classes

- Examining the usage scenarios developed as part of the requirements model and perform a "grammatical parse" [Abb83]
  - 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
  - Example:

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.

- But what should we look for once all of the nouns have been identified?

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
  - *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
  - *Structures* (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects

Defining Attributes

- Study each use case and select those “things” that reasonably “belong” to the class
- Ask this question for each class:
  - *What data items fully define this class in the context of the problem at hand?*
- Example:
  - Consider the **System** class defined for SafeHome
  - A homeowner can configure the system to reflect sensor information, alarm information, activation or deactivation information, identification information, …

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

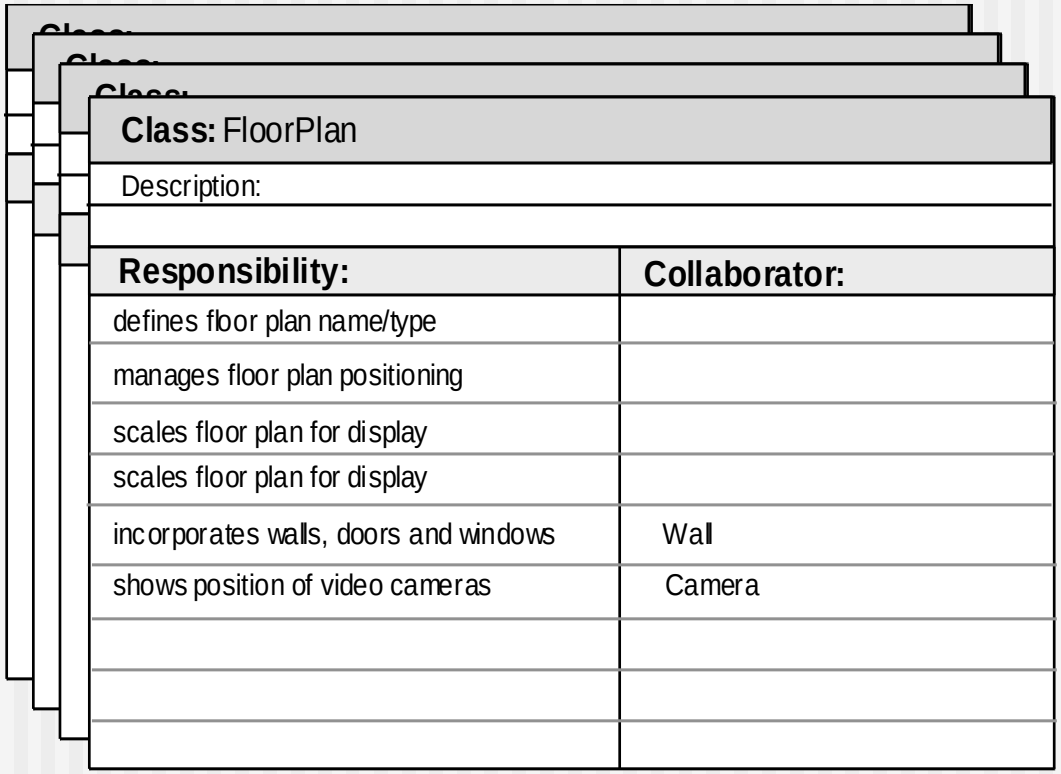
Defining Operations

- Do a grammatical parse of a processing narrative and look at the **verbs**
- For example, from the SafeHome processing narrative:
  - “sensor is assigned a number and type”
  - “a master password is programmed for arming and disarming the system.”
  - These phrases indicate a number of things:
    - assign() operation is relevant for the Sensor class.
    - program() operation will be applied to the System class.
    - arm() and disarm() are operations that apply to System class

CRC Models

- *Class-responsibility-collaborator (CRC) modeling* [Wir90] provides a **simple** means for identifying and organizing the classes
- Ambler [Amb95] 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.

CRC Modeling



Class: FloorPlan

Description:

Responsibility:	Collaborator:
defines floor plan name/type	
manages floor plan positioning	
scales floor plan for display	
scales floor plan for display	
incorporates walls, doors and windows	Wall
shows position of video cameras	Camera

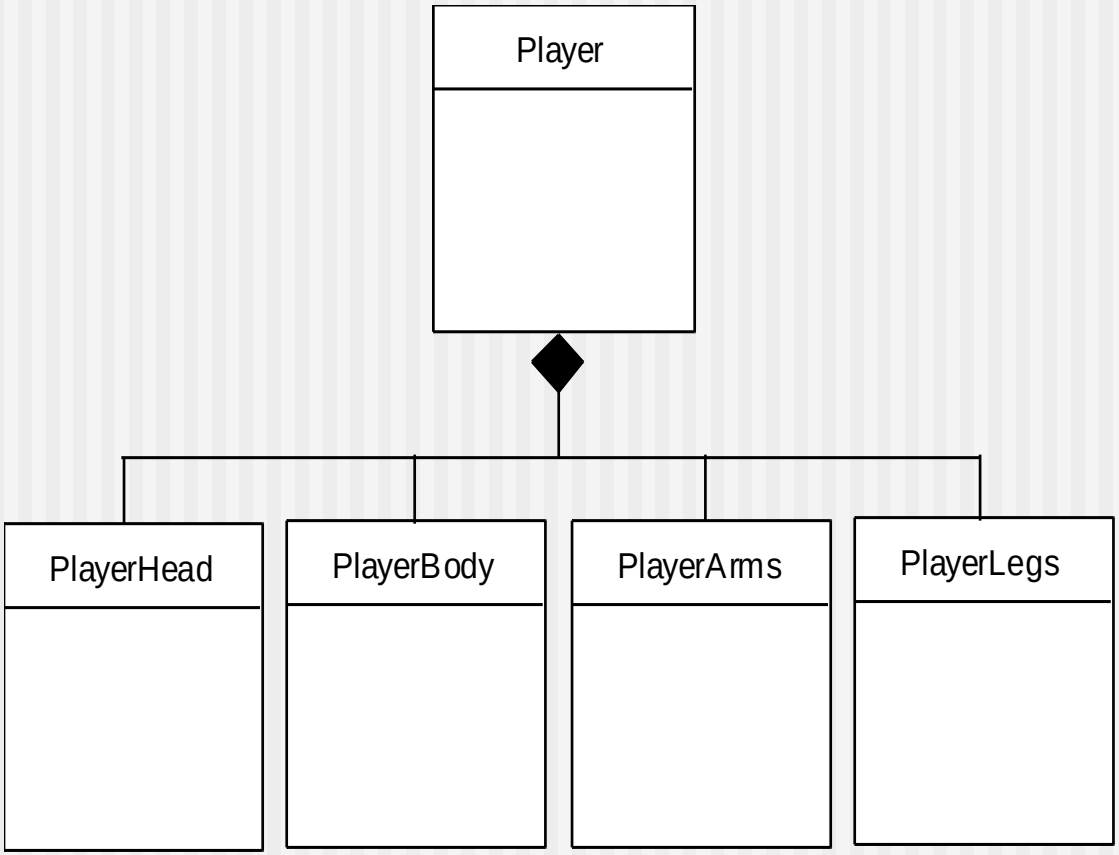
Responsibilities

- Each responsibility should be stated as generally as possible
  - general responsibilities (both attributes and operations) should reside high in the class hierarchy
- Information and the behavior related to it should reside within the same class
  - encapsulation
- Information about one thing should be localized with a single class, not distributed across multiple classes.
- Responsibilities should be shared among related classes, when appropriate.
  - Consider a video game with four classes: **Player**, **PlayerBody**, **PlayerArms**, **PlayerLegs**, **PlayerHead**.
  - Each class has its own attributes (e.g., position, orientation, …) and must be updated and displayed as the user manipulates
  - The responsibilities update and display must therefore be shared

Collaborations

- Collaborations are identified by determining whether a class can fulfill each responsibility itself
- If it cannot, then it needs to interact with another class.
- Collaborations identify relationships between classes
- Three different generic relationships between classes [WIR90]:
  - the *is-part-of* relationship (for example, **PlayerHead** and **Player**)
  - the *has-knowledge-of* relationship
    - one class must acquire information from another class
  - the *depends-upon* relationship
    - Not achieved *by has-knowledge-of* or *is-part-of*.
    - For example, **PlayerHead** must always be connected to **PlayerBody**
    - Yet each object could exist without direct knowledge of the other
    - An attribute of the **PlayerHead** object called center-position is determined from the center position of **PlayerBody**
    - This information is obtained via a third object, **Player**, that acquires it from **PlayerBody**. Hence, **PlayerHead** depends-upon **PlayerBody**

Composite Aggregate Class



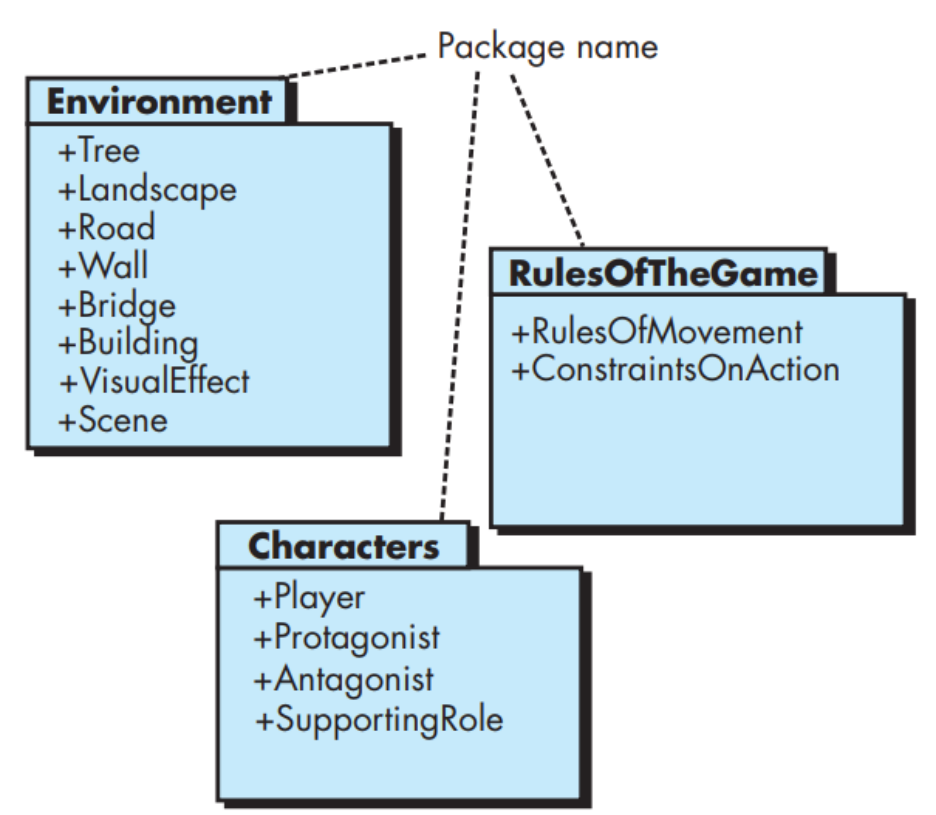
UML Class Diagram

- Remind:
  - Standard Notations
  - Associations
    - Multiplicity
  - Dependencies
  - Composition and Aggregation
  - Specialization/Generalization
  - …

Analysis Packages

- An important part of analysis modeling is **categorization**
- Various elements of the analysis model (e.g., use-cases, analysis classes) are categorized as a grouping
  - Called an *analysis package*
- Given a representative name
- The plus sign preceding the analysis class name in each package indicates that
  - the classes have public visibility and accessible from other packages
- A minus sign indicates that an element is hidden from all other packages
- # symbol indicates that an element is accessible only to packages contained within a given package

Analysis Packages



Requirements Modeling: Behavior

Behavioral Modeling

- The behavioral model indicates how software will respond to external events.
- To create the model, the analyst must perform the following steps:
    1. Evaluate all use-cases to fully understand the sequence of interaction within the system.
    2. Identify events that drive 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 Use-Cases

- In general, an event occurs whenever the system and an actor **exchange information**
- 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.

  - "homeowner uses the keypad to key in a four-digit password"
    - Event: *password entered* (generated by **Homeowner**)
  - "password is compared with the valid password stored in the system"
    - Event: *password compared* (recognized by **ControlPanel**)

State Representations

- In the context of behavioral modeling, two different characterizations of states must be considered:
  - The state of each class as the system performs its function
  - 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**:
  - A *passive state* is simply the current status of all of an object's attributes
    - The passive state of the class **Player** would include the current value of position, orientation, …
  - The *active state* of an object indicates the current status of the object as it undergoes a continuing transformation
    - The active states of the class **Player**: moving, at rest, injured, being cured, trapped, lost, …
    - An event (sometimes called a trigger) must occur to force an object to make a transition from one active state to another

State and Sequence Diagrams

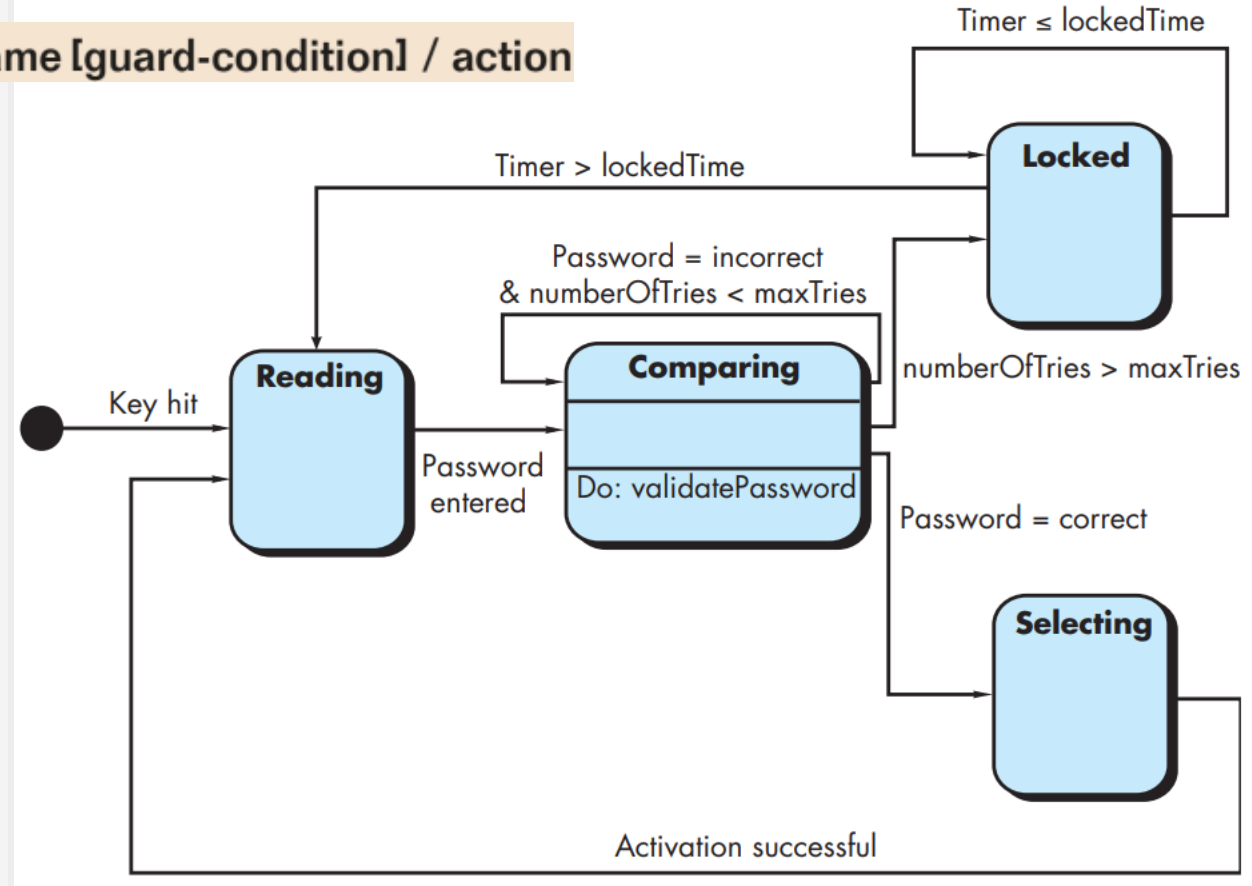
- Two different behavioral representations:
  1. State diagram—indicates how **an individual class** changes state based on external events
     - The active state model
     - Shows the "life history" of an object
  2. Sequence diagram—shows **the behavior of the system** as a function of time
     - Shows how events cause transitions from one object to another object

       (key classes in a use-case and the events that cause behavior to flow from class to class)

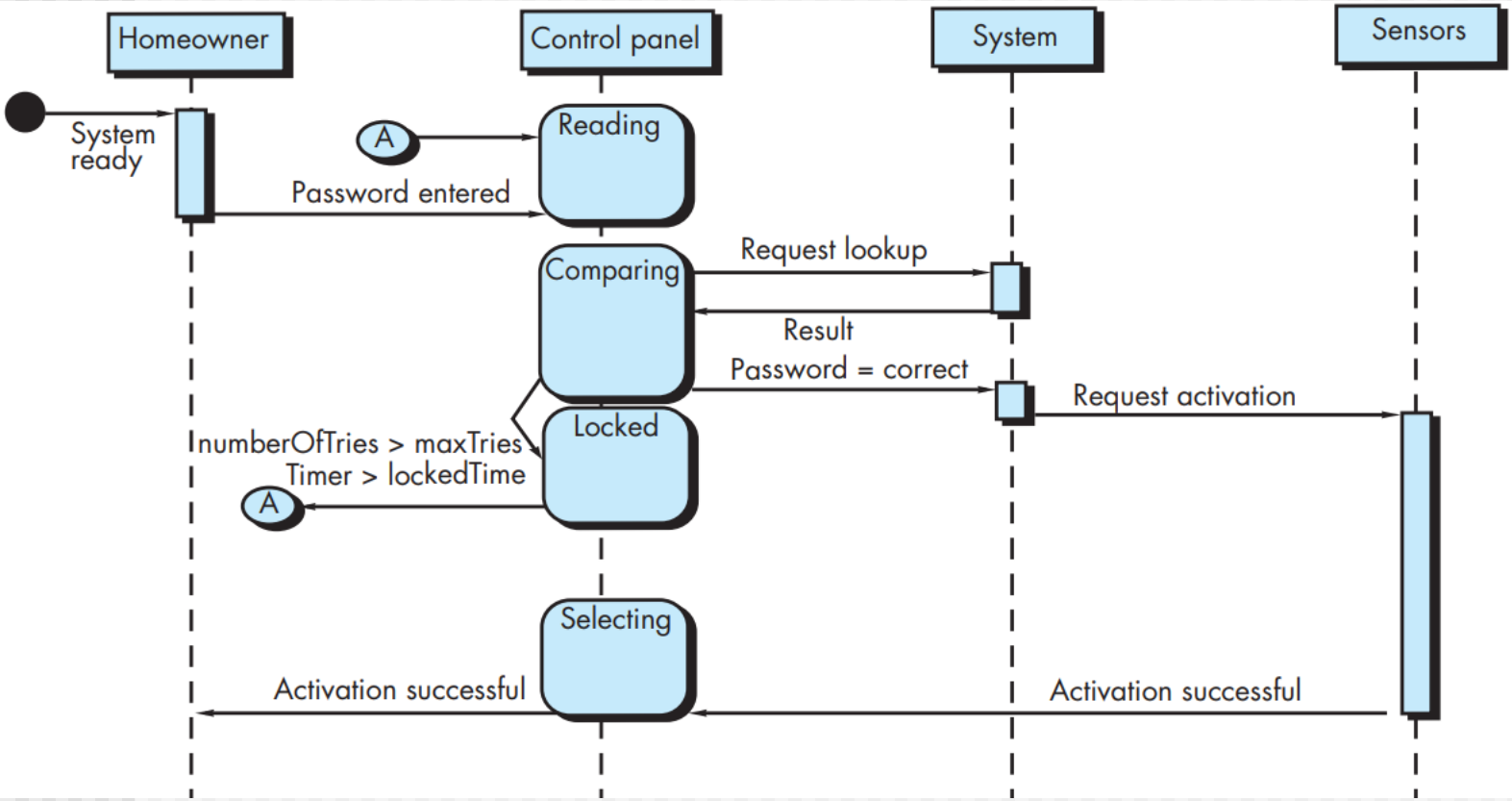
State Diagram Elements

- state—active states for a class
- state transition—the movement from one state to another (represented by arrows)
- event (or trigger)—causes change between active states
  - represented by a label for the arrow
- guard—a Boolean condition that must be satisfied in order to occur the transition
- action—occurs concurrently with the state transition or as a consequence of it
  - generally involves one or more operations (responsibilities) of the object

State Diagram for the ControlPanel Class (use-case: “SafeHome security function”)



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



Requirements Modeling for Web and Mobile Apps

Requirements Modeling for Web/Mobile Apps

Content Analysis—identifies the full spectrum of content to be provided by the app (e.g., text, images, video, audio data,..)

Interaction Analysis—describes the manner in which the user interacts with the app

Functional Analysis—defines the operations that will be applied to manipulate content and describes all processing functions in detail

Configuration Analysis—describes the environment and infrastructure in which the app resides

Navigation Analysis—defines the overall navigation strategy for the app

When Do We Perform Analysis?

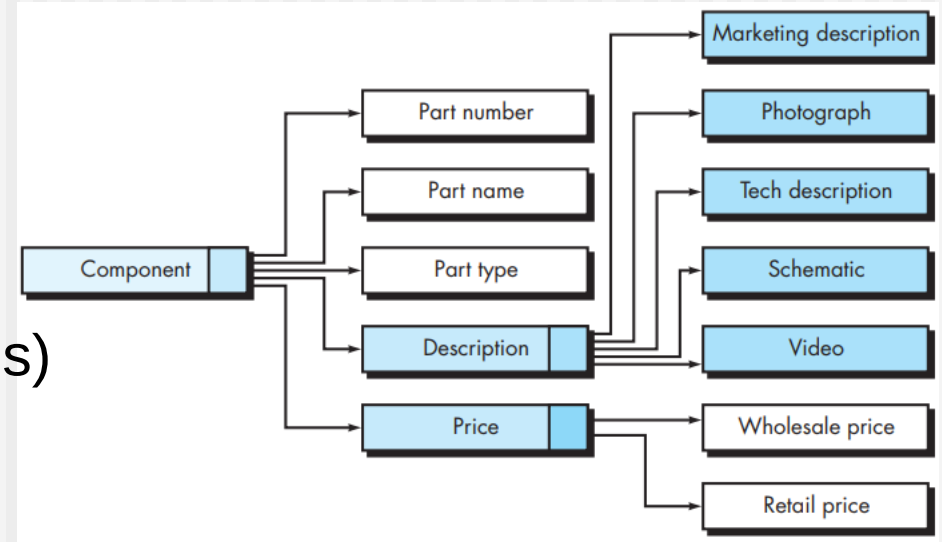
- In some Web/Mobile App situations, analysis and design merge. However, an explicit analysis activity occurs when …
  - the Web or Mobile App to be built is large and/or complex
  - the number of stakeholders is large
  - the number of developers is large
  - the development team members have not worked together before
  - the success of the app will have a strong bearing on the success of the business

The Content Model

- Content objects: **user-visible** entities that are created or manipulated as a user interacts with app
  - E.g., a textual description of a product, an article, a photograph, a user's response on a forum, an animated logo, a video of speech, …
- Can be extracted from use-cases
  - examine the scenario description for direct and indirect references to content
- Attributes of each content object are identified
- The relationships among content objects and/or the hierarchy of content maintained by an app
  - Relationships—entity-relationship diagram or UML
  - Hierarchy—data tree or UML

Example: Data Tree

- A use case, *Purchasing SafeHome Components,* describes this scenario:
  - I will be able to get descriptive and pricing information for each product component
- Data tree shows a hierarchy of information that describes a content object (here, product component)
- In this example: There are eight content objects (shaded rectangles)



The Interaction Model

- Web and mobile apps enable a “conversation” between an end user and the app
- This conversation can be described using an interaction model
- Composed of four elements:
  - use-cases
  - sequence diagrams
  - state diagrams
  - and/or user interface prototypes
- *In many instances, **a set of use cases** is sufficient to describe the interaction at an analysis level (further refinement and detail is introduced during design)*

Sequence Diagram

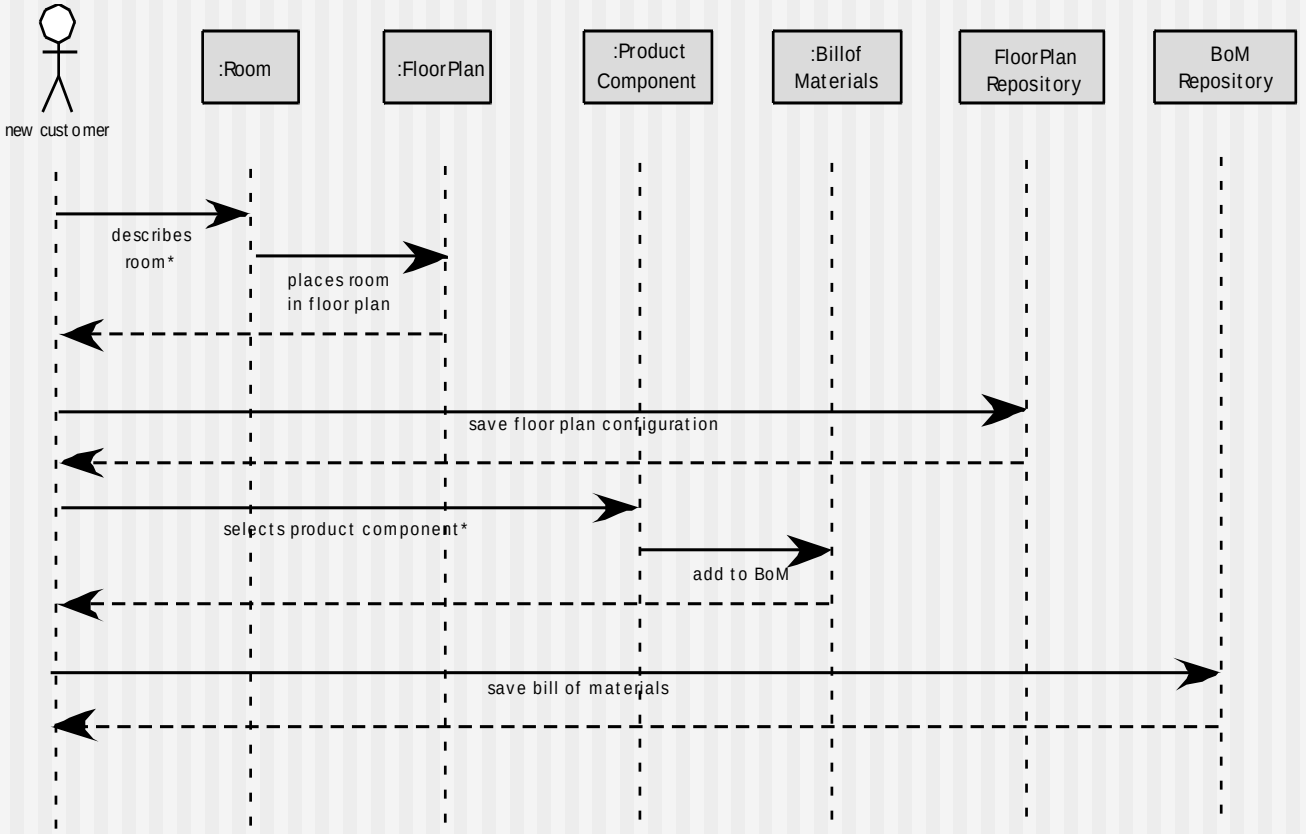


Figure 18.5 Sequence diagram for use-case: *select SafeHome components*

State Diagram

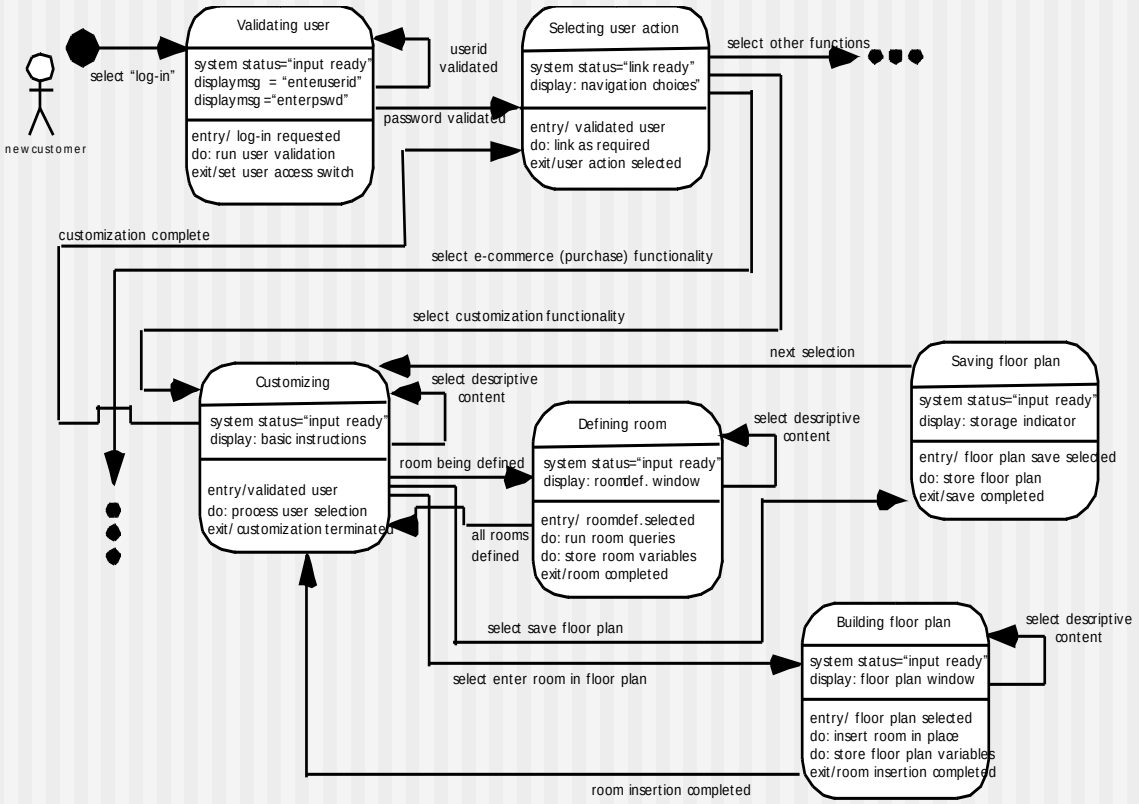


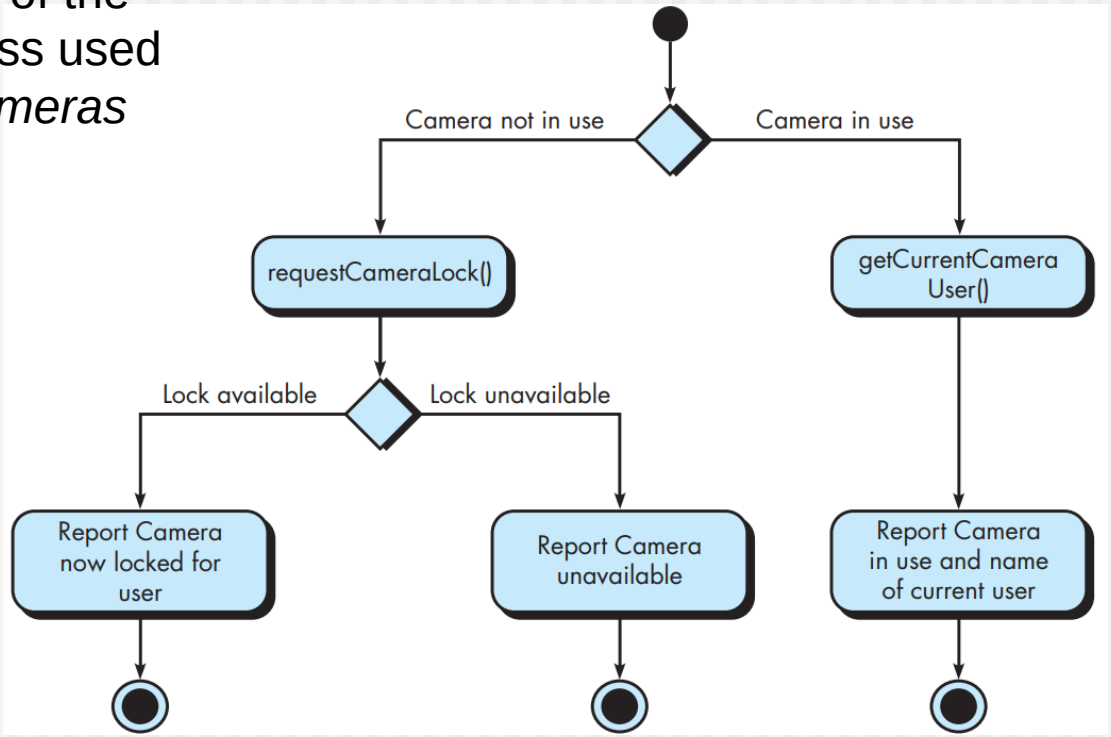
Figure 18.6 Partial state diagram for **new customer** interaction

The Functional Model

- The functional model addresses two processing elements of an app:
  1. user-observable functionality that is delivered by the app to end-users
     - Any processing functions that are **initiated directly by the user**
  2. operations contained within analysis classes that implement behaviors associated with the class
- For example, a financial mobile app might provide a financial function: computing mortgage payment
  - From the user's point of view, this is a visible process (it has a visible outcome)
  - This process may actually be implemented using different operations within analysis classes
- An activity diagram can be used to show processing flow

Activity Diagram

An activity diagram for the *takeControlOfCamera()* operation that is part of the **Camera** analysis class used within the *Control cameras* use case



The Configuration Model

- In some cases, the configuration model is nothing more than a list of server-side and client-side attributes
- Server-side
  - Server hardware and operating system environment must be specified
  - Interoperability considerations on the server-side must be considered
  - Appropriate interfaces and communication protocols must be specified
- Client-side
  - Browser configuration issues must be identified
  - Testing requirements should be defined
- For complex apps, the UML deployment diagram can be used to model configuration architectures

The Navigation Model

- In most mobile applications that reside on smartphone platforms:
  - Navigation is generally constrained to relatively simple button lists and icon-based menus
  - The depth of navigation is relatively shallow
  - So, navigation modeling is relatively simple
- For WebApps, navigation modeling is more complex
  - How each user category will navigate from one WebApp element (e.g., content object) to another
- The mechanics of navigation are defined at design
- At this stage, we focus on overall navigation requirements

Navigation Requirements

- Should certain elements be easier to reach (require fewer navigation steps) than others? What is the priority for presentation?
- How should navigation errors be handled?
- Should navigation be accomplished via links, via search-based access, or by some other means?
- Should a navigation log be maintained for users?
- Should a full navigation map or menu (as opposed to a single “back” link) be available at every point in a user’s interaction?
- Can a user “store” his previous navigation through the WebApp to expedite future usage?
- Can a user take a “guided tour” that highlight the most important available elements (content objects and functions)?
- …

Further Reading

The End