Building the Analysis Model 4

Suradet Jitprapaikulsarn

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Class-based Modeling

- Identify analysis classes by examining the problem statement
- Use a "grammatical parse" to isolate potential classes
- Identify the attributes of each class
- Identify operations that manipulate the attributes

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Analysis Classes

- **External entities** (printer, user, sensor)
- **Things** (report, display, signal)
- **Occurrences or events** (alarm, telephone call)
- **Roles** (manager, clerk)
- **Organization units** (Accounting Dept, R & D)
- **Places** (building, manufacturing floor)
- **Structures** (employee records)

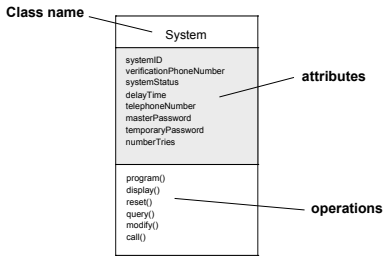
Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Selecting Classes—Criteria

- retained information
- needed services
- multiple attributes
- common attributes
- common operations
- essential requirements

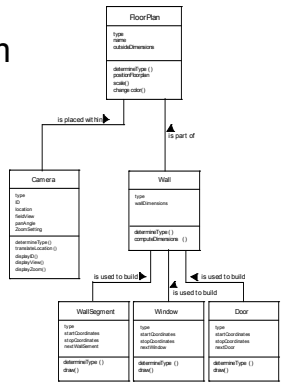
Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Class Diagram



Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Class Diagram



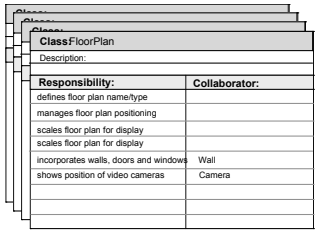
Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

CRC Modeling

- Analysis classes have “responsibilities”
  - ***Responsibilities*** are the attributes and operations encapsulated by the class
- Analysis classes collaborate with one another
  - ***Collaborators*** are those classes that are required to provide a class with the information needed to complete a responsibility.
  - In general, a collaboration implies either a request for information or a request for some action.

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

CRC Modeling



Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Class Types

- ***Entity classes***, also called *model* or *business* classes, are extracted directly from the statement of the problem (e.g., FloorPlan and Sensor).
- ***Boundary classes*** are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
- ***Controller classes*** manage a “unit of work” [UML03] from start to finish. That is, controller classes can be designed to manage
  - the creation or update of entity objects;
  - the instantiation of boundary objects as they obtain information from entity objects;
  - complex communication between sets of objects;
  - validation of data communicated between objects or between the user and the application.

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Responsibilities

- System intelligence should be distributed across classes to best address the needs of the problem
- Each responsibility should be stated as generally as possible
- Information and the behavior related to it should reside within the same class
- 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

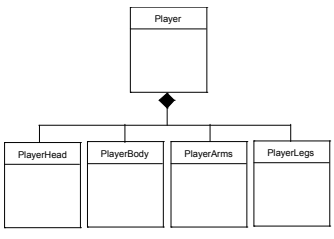
Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Collaborations

- Classes fulfill their responsibilities in one of two ways:
  - A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
  - a class can collaborate with other classes.
- Collaborations identify relationships between classes
- Collaborations are identified by determining whether a class can fulfill each responsibility itself
- three different generic relationships between classes [WIR90]:
  - the ***is-part-of*** relationship
  - the ***has-knowledge-of*** relationship
  - the ***depends-upon*** relationship

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Composite Aggregate Class



Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Reviews of CRC model

- All participants in the review (of the CRC model) are given a subset of the CRC model index cards.
  - Cards that collaborate should be separated (i.e., no reviewer should have two cards that collaborate).
- All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
- 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.
- 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.
- If the responsibilities and collaborations noted on the index cards cannot accommodate the use-case, modifications are made to the cards.
  - This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

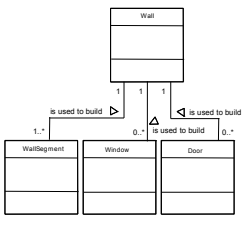
Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Associations and Dependencies

- Two analysis classes are often related to one another in some fashion
  - In UML these relationships are called ***associations***
  - Associations can be refined by indicating ***multiplicity*** (the term ***cardinality*** is used in data modeling
- 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

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Multiplicity



Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Dependency

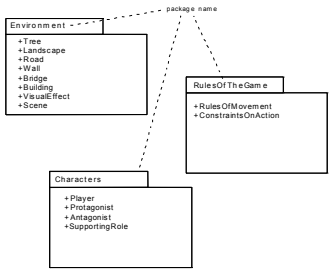
Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Analysis Package

- Various elements of the analysis model (e.g., use-cases, analysis classes) are categorized in a manner that packages them as a grouping
- The plus sign preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages.
- Other symbols can precede an element within a package. A minus sign indicates that an element is hidden from all other packages and a # symbol indicates that an element is accessible only to packages contained within a given package.

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Analysis Package



Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Behavioral Modeling

- The behavioral model indicates how software will respond to external events or stimuli. To create the model, the analyst must perform the following steps:
  - Evaluate all use-cases to fully understand the sequence of interaction within the system.
  - Identify events that drive the interaction sequence and understand how these events relate to specific objects.
  - Create a sequence for each use-case.
  - Build a state diagram for the system.
  - Review the behavioral model to verify accuracy and consistency.

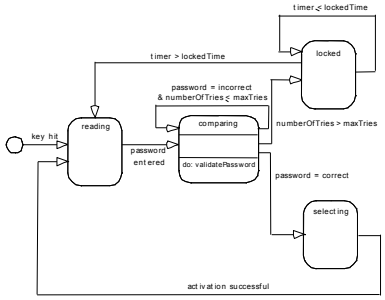
Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

State Representation

- 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 and
  - 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 [CHA93].
  - A ***passive state*** is simply the current status of all of an object's attributes.
  - The ***active state*** of an object indicates the current status of the object as it undergoes a continuing transformation or processing.

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

State Diagram for the ControlPanel Class



Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

The State of a System

- **state**—a set of observable circum-stances that characterizes the behavior of a system at a given time
- **state transition**—the movement from one state to another
- **event**—an occurrence that causes the system to exhibit some predictable form of behavior
- **action**—process that occurs as a consequence of making a transition

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Behavioral Modeling

- make a list of the different states of a system (How does the system behave?)
- indicate how the system makes a transition from one state to another (How does the system change state?)
  - indicate event
  - indicate action
- draw a **state diagram** or a **sequence diagram**

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Sequence Diagram

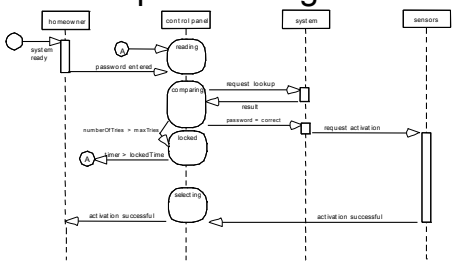
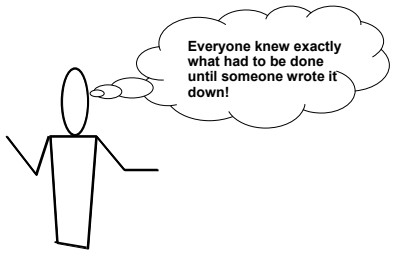


Figure 8.27 Sequence diagram (partial) for *SafeHome* security function

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Writing the Software Specification



Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Specification Guideline

- ❑ use a layered format that provides increasing detail as the "layers" deepen
- ❑ use consistent graphical notation and apply textual terms consistently (stay away from aliases)
- ❑ be sure to define all acronyms
- ❑ be sure to include a table of contents; ideally, include an index and/or a glossary
- ❑ write in a simple, unambiguous style (see "editing suggestions" on the following pages)
- ❑ always put yourself in the reader's position, "Would I be able to understand this if I wasn't intimately familiar with the system?"

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Specification Guideline

Be on the lookout for persuasive connectors, ask why?
keys: *certainly, therefore, clearly, obviously, it follows that ...*

Watch out for vague terms
keys: *some, sometimes, often, usually,ordinarily, most, mostly ...*

When lists are given, but not completed, be sure all items are understood
keys: *etc., and so forth, and so on, such as*

Be sure stated ranges don't contain unstated assumptions
e.g., *Valid codes range from 10 to 100.* Integer? Real? Hex?

Beware of vague verbs such as *handled, rejected, processed, ...*

Beware "passive voice" statements
e.g., *The parameters are initialized.* By what?

Beware "dangling" pronouns
e.g., *The I/O module communicated with the data validation module and its contol flag is set.* Whose control flag?

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005

Specification Guideline

When a term is explicitly defined in one place, try substituting the definition forother occurrences of the term

When a structure is described in words, draw a picture

When a structure is described with a picture, try to redraw the picture to emphasize different elements of the structure

When symbolic equations are used, try expressing their meaning in words

When a calculation is specified, work at least two examples

Look for statements that imply certainty, then ask for proof
keys; always, every, all, none, never

Search behind certainty statements—be sure restrictions or limitations are realistic

Derived from Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th Edition, McGraw-Hill, 2005