CS 307: Software Engineering

Lecture 7: Unified Modeling Language (UML)

Prof. Jeff Turkstra
Announcements

- Code repository, project name, and product backlog
  - Due Friday, January 27
Lecture 07

- Unified Modeling Language (UML)
What is UML?

- A “general purpose” modeling language
- Provides a standardized method for designing a system
- Adopted by OMG (Object Management Group) in 1997
- Latest standard, 2.0, published in 2005
UML

- ...has detailed semantics
- has extension mechanisms
- has an associated textual language
  - Object Constraint Language (OCL)
- Not a methodology
  - Assists in software development
What is a “good” model?

- A model that
  - Uses a standard notation
  - Is understandable by all stakeholders
  - Helps software engineers gain insight into the system
  - Provides abstraction

- When do you use it?
  - To help create designs
  - To analyze and review designs
  - As documentation for the system
Diagram essentials

- Classes: represent the types of data themselves
- Associations: linkages between class instances
- Attributes: simple data found in classes and instances
- Operations: represent the functions (methods) performed by classes
- Generalizations: grouping of classes into inheritance hierarchies
Diagrams

- Static (structural) view
  - Class diagrams
  - Composite structure diagrams

- Dynamic (behavioral) view
  - Sequence diagrams
  - Activity diagrams
  - State machine diagrams
Class diagrams

- Describes the system’s classes
  - Attributes
  - Operations (methods)
  - Relationship among objects

<table>
<thead>
<tr>
<th>Rectangle</th>
<th>Rectangle</th>
<th>Rectangle</th>
<th>Rectangle</th>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>width</td>
<td>height</td>
<td>width</td>
<td>- height: int</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- width: int</td>
</tr>
<tr>
<td>getArea()</td>
<td>resize()</td>
<td>getArea()</td>
<td>resize()</td>
<td>+ getArea(): int</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+ resize(int,int)</td>
</tr>
</tbody>
</table>
Composite structure diagrams

- Shows internal structure of a class
- Parts: runtime role of a classifier (e.g., class) or a collection of instances
- Port: connects classifiers with their parts and environment
- Connector: binds two or more entities together
Activity diagrams

- Graphical workflow showing stepwise activity and actions
  - Supports choice, iteration, and concurrency
- i.e., a flowchart
  - Petri-net (later)
State machine diagrams

- Recall: mealy/moore FSMs
Associations and multiplicity

- An association shows how two classes relate to each other.
- Symbols indicating multiplicity are shown at each end of the association.
Labeling associations

- Employee * \( \rightarrow \) worksFor 1 \( \rightarrow \) Company
- AdministrativeAssistant * \( \rightarrow \) Manager 1..* \( \rightarrow \) supervisor
- Company 1 \( \rightarrow \) 1 \( \rightarrow \) BoardOfDirectors
- Office 0..1 \( \rightarrow \) allocatedTo * \( \rightarrow \) Employee
- Person 0..8 \( \rightarrow \) boardMember * \( \rightarrow \) BoardOfDirectors
Many-to-one

- A company has many employees
- An employee can work for only one company
- A company could have zero employees
  - “Shell” company
- Cannot be an employee unless you work for a company
Many-to-many

- An assistant can work for many managers
- A manager can have many assistants
- Assistants may work in pools
- Managers may have a group of assistants
- Some managers may have zero assistants
- Could an assistant have zero managers?

Diagram:

```
Assistant -> supervisor Manager
*            1..* 
```
One-to-one

- For a given company, exactly one board of directors
- Board oversees only one company
- Company must always have a board
- Board must always be with a company
Analysis and validation

- Avoid unnecessary one-to-one associations

Not this

<table>
<thead>
<tr>
<th>Person</th>
<th>1</th>
<th>PersonInfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
<td>address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>email</td>
</tr>
<tr>
<td></td>
<td></td>
<td>birthdate</td>
</tr>
</tbody>
</table>

This

<table>
<thead>
<tr>
<th>Person</th>
<th>name</th>
<th>address</th>
<th>email</th>
<th>birthdate</th>
</tr>
</thead>
</table>

More complex example

- Booking is always for exactly one passenger
  - Never zero passengers
  - Never more than one
- Passenger can have any number of bookings
  - None
  - More than one

Frame is an optional feature of any UML diagram
Association classes

- Sometimes, an attribute concerning two classed cannot be placed in either

- These are equivalent:

![Diagram showing association classes](image)

![Diagram showing alternative association](image)
Reflexive associations

- An association can connect a class to itself
Generalization

- A generalization set is a labeled group of generalizations with a common superclass
- The label (discriminator) describes the criteria used for specializing
Multiple discriminators

![Diagram showing the hierarchy of animal types]

- Animal
  - Habitat
  - AquaticAnimal
    - TypeOfFood
      - AquaticCarnivore
      - AquaticHerbivore
  - LandAnimal
    - TypeOfFood
      - LandCarnivore
      - LandHerbivore
Objects do not change class

- For example, a student may be part time sometimes or full time others.
Notes and descriptive text

- Descriptive text and other diagrams
  - Embed your diagrams in a larger document
  - Text can explain aspects of the system using any notation you like
  - Highlight and expand on important features
    - Give rationales

- Notes
  - Small block of text embedded in the UML diagram
  - Think comment
Developing class diagrams

- UML models can be created at different stages
  - Different purposes?
  - Different levels of detail
- Exploratory domain model
  - Domain analysis, represents aspects of the domain
- System domain model
  - Models aspects of domain represented by the system
- System model
  - Includes classes for UI and system architecture
System domain model vs. system model

- The system domain model omits many classes needed for a complete system
  - Could contain less than half of them
  - Developed and used independently of
    - UI classes
    - Architectural classes

- Complete system model includes
  - System domain model
  - UI Classes
  - Architectural classes
  - Utility classes
Suggested sequence

- Identify a first set of candidate classes
- Add associations and attributes
- Find generalizations
- List the main responsibilities of each class
- Decide on specific operations
- Iterate over the entire process until model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities, operations, etc
  - Identify interfaces
  - Apply design patterns (later)
Identifying classes

- Developing a domain model of leads to discovery of classes
- When you work on the UI or architecture, one tends to invent classes to solve a design problem
- Reuse should always be kept in mind
Discovering domain classes

- Look at source material such as description of requirements
- Extract nouns and noun phrases
- Eliminate nouns that
  - ...are redundant
  - ...represent instances
  - ...are vague or highly general
  - ...not needed in the application
- Pay attention to classes in a domain model that represents types of users or other actors
Identifying associations and attributes

- Start with classes you think are most central and important
- Decide on clear and obvious data each must contain in its relationships to other classes
- Work outwards towards classes that are less important
- Avoid adding many associations and attributes to a class
  - Systems are simpler if they manipulate less information
Attributes

- Look for information about each class that must be maintained
- Nouns rejected as classes may make good attributes
- Attribute should generally contain a simple value
  - String, number, etc
Tips

- Avoid duplicates if possible
- If a subset can be formed, create a distinct class

Bad, due to a plural attribute  
Bad, due to too many attributes, and the inability to add more addresses  
Good solution. The type indicates whether it is a home address, business address etc.
Example

```
+----------------+      +----------------+      +----------------+
| Passenger      |      | Employee       |      | RegularFlight  |
| name           |      | name           |      | time           |
| number         |      | employeeNumber |      | flightNumber   |
|                |      | jobFunction    |      |                |
+----------------+      +----------------+      +----------------+
| 1               |      | *              |      | 1              |
| *               |      |                |      |                |
+----------------+      +----------------+      +----------------+
| Booking        |      | SpecificFlight |
| seatNumber     |      | date           |
+----------------+      +----------------+      +----------------+
```
Allocating responsibilities

- A responsibility is something that the system is required to do
- Each functional requirement should be attributed to a class
  - All responsibilities for a given class should be clearly related
  - Too many responsibilities? Consider splitting into two or more classes
  - No responsibilities? Likely useless
  - Trouble attributing a responsibility to a class? Create a new class
Responsibilities

- **Perform use case analysis**
  - Look at user stories
- **Look for verbs and nouns describing actions in the system description**
Example

- Creating a new regular flight
- Searching for a flight
- Modifying attributes of a flight
- Creating a specific flight
- Booking a passenger
- Canceling a booking
Prototyping class diagrams

- Write names on small cards as you identify classes
- List attributes and responsibilities as they become known
  - If you cannot fit everything on one card, maybe it should be split
- Move cards around on a whiteboard to create a class diagram
- Draw lines among the cards for associations and generalizations
Identifying operations

- Operations are needed to realize the responsibilities of each class
  - May be several operations for one responsibility
  - Main operations are normally public
  - Other methods that collaborate to perform the responsibility should be private
Class diagrams in Java

- Attributes are instance variables
- Generalizations → extends
- Interfaces → implements
- Associations are normally implemented using instance variables
  - Divide two-way association into two one-way associations (each class has an instance variable)
  - Multiplicity one or optional, declare a variable of that class (a reference)
  - Multiplicity of many, use a collection class implementing List (e.g., vector)
Questions?