Outline of this Class

• What is UML?
• A more detailed view on
  ✓ Use case diagrams
  ✓ Class diagrams
  ✓ Sequence diagrams
  ✓ Activity/Statecharts diagrams
UML Basic Notation: First Summary

- UML provides a wide variety of notations for modeling many aspects of software systems
- UML diagrams cover the three fundamental models for software design:
  - Functional model: Use case diagrams
  - Object model: Class diagrams
  - Dynamic model: Sequence diagrams, statechart diagram

- Now we go into a little bit more detail...

UML First Pass (covered in Last Lecture)

- Use case diagrams
  - Describe the functional behavior of the system as seen by the user
- Class diagrams
  - Describe the static structure of the system: Objects, attributes, associations
- Sequence diagrams
  - Describe the dynamic behavior between objects of the system
- Statechart diagrams
  - Describe the dynamic behavior of an individual object
- Activity diagrams
  - Describe the dynamic behavior of a system, in particular the workflow.
UML first pass: Use case diagrams

Use case diagrams represent the functionality of the system from user’s point of view.
**UML Use Case Diagrams**

Used during requirements elicitation and analysis to represent external behavior (“visible from the outside of the system”)

An *Actor* represents a role, that is, a type of user of the system

A *use case* represents a class of functionality provided by the system

**Use case model:** The set of all use cases that completely describe the functionality of the system.

---

**Actors**

- An actor is a model for an external entity which interacts (communicates) with the system:
  - User
  - External system (Another system)
  - Physical environment (e.g. Weather)

- An actor has a unique name and an optional description

- Examples:
  - *Passenger*: A person in the train
  - *GPS satellite*: An external system that provides the system with GPS coordinates.
Use Case

- A use case represents a class of functionality provided by the system
- Use cases can be described textually, with a focus on the event flow between actor and system
- The textual use case description consists of 6 parts:
  1. Unique name
  2. Participating actors
  3. Entry conditions
  4. Exit conditions
  5. Flow of events
  6. Special requirements.

**PurchaseTicket**

Use Case Description Example

1. **Name**: Purchase ticket
2. **Participating actor**: Passenger
3. **Entry condition**:
   - *(GOOD)* Passenger selects an option from the display
   - *(WRONG)* Passenger stands in front of ticket distributor
   - *(Very WRONG)* Passenger has sufficient money to purchase ticket
4. **Exit condition**:
   - Passenger has ticket
   - *(Better)*: System delivered ticket
5. **Flow of events**:
   1. Passenger selects the number of zones to be traveled
   2. Ticket Distributor displays the amount due
   3. Passenger inserts money, at least the amount due
   4. Ticket Distributor returns change
   5. Ticket Distributor issues ticket
6. **Special requirements**: None.
Uses Cases can be related

- **Extends Relationship**
  - To represent seldom invoked use cases or exceptional functionality
- **Includes Relationship**
  - To represent functional behavior common to more than one use case.

The `<<extends>>` Relationship

- `<<extends>>` relationships model exceptional or seldom invoked cases
- The exceptional event flows are factored out of the main event flow for clarity
- The direction of an `<<extends>>` relationship is to the extended use case
- Use cases representing exceptional flows can extend more than one use case.
The <<includes>> Relationship

- <<includes>> relationship represents common functionality needed in more than one use case
- <<includes>> behavior is factored out for reuse, not because it is an exception
- The direction of a <<includes>> relationship is to the using use case (unlike the direction of the <<extends>> relationship).

Use Case Models can be packaged

- Classifier
- Use Case
- System boundary
Historical Remark: UML 1 used packages

UML Class Diagram
UML first pass: Class diagrams

Class diagrams represent the structure of the system
Class Diagrams

- Class diagrams represent the structure of the system
- Used
  - during requirements analysis to model application domain concepts
  - during system design to model subsystems
  - during object design to specify the detailed behavior and attributes of classes.

<table>
<thead>
<tr>
<th>TarifSchedule</th>
<th>Trip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumeration getZones()</td>
<td>zone: Zone</td>
</tr>
<tr>
<td>Price getPrice(Zone)</td>
<td>Price: Price</td>
</tr>
</tbody>
</table>

Classes

- A class represents a concept
- A class encapsulates state (attributes) and behavior (operations)
  - Each attribute has a type
  - Each operation has a signature

The class name is the only mandatory information
Actor vs Class vs Object

- **Actor**
  - An entity outside the system to be modeled, interacting with the system ("Passenger")

- **Class**
  - An abstraction modeling an entity in the application or solution domain
  - The class is part of the system model ("User", "Ticket distributor", "Server")

- **Object**
  - A specific instance of a class ("Joe, the passenger who is purchasing a ticket from the ticket distributor").

Instances

<table>
<thead>
<tr>
<th>tarif2006:TarifSchedule</th>
<th>:TarifSchedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>zone2price = {</td>
<td>zone2price = {</td>
</tr>
<tr>
<td>{'1', 0.20},</td>
<td>{'1', 0.20},</td>
</tr>
<tr>
<td>{'2', 0.40},</td>
<td>{'2', 0.40},</td>
</tr>
<tr>
<td>{'3', 0.60}}</td>
<td>{'3', 0.60}}</td>
</tr>
</tbody>
</table>

- An *instance* represents a phenomenon
- The attributes are represented with their *values*
- The name of an instance is underlined
- The name can contain only the class name of the instance (anonymous instance)
Associations

Associations denote relationships between classes

The multiplicity of an association end denotes how many objects the instance of a class can legitimately reference.

1-to-1 and 1-to-many Associations

1-to-1 association

1-to-many association
Many-to-many Associations

- A stock exchange lists many companies.
- Each company is identified by a ticker symbol

From Problem Statement To Object Model

Problem Statement: A stock exchange lists many companies. Each company is uniquely identified by a ticker symbol

Class Diagram:
From Problem Statement to Code

**Problem Statement** : A stock exchange lists many companies. Each company is identified by a ticker symbol

**Class Diagram:**

```
private Vector m_Company = new Vector();
public int m_tickerSymbol;
private Vector m_StockExchange = new Vector();
```

**Java Code**

```
public class StockExchange
{
    private Vector m_Company = new Vector();
};
public class Company
{
    public int m_tickerSymbol;
    private Vector m_StockExchange = new Vector();
};
```

Associations are mapped to Attributes!

**Qualifiers**

Without qualification

```
Directory 1 *
File
filename
```

With qualification

```
Directory filename 1 0..1 File
```

- Qualifiers can be used to reduce the multiplicity of an association
Qualification: Another Example

StockExchange * Lists * Company

StockExchange * Lists 1 Company

tickerSymbol

tickerSymbol

Aggregation

- An aggregation is a special case of association denoting a “consists-of” hierarchy
- The aggregate is the parent class, the components are the children classes

A solid diamond denotes composition: A strong form of aggregation where the life time of the component instances is controlled by the aggregate. That is, the parts don’t exist on their own (“the whole controls/destroys the parts”)
**Inheritance**

- **Inheritance** is another special case of an association denoting a “kind-of” hierarchy
- Inheritance simplifies the analysis model by introducing a taxonomy
- The children classes inherit the attributes and operations of the parent class.

![Inheritance Diagram]

**Packages**

- Packages help you to organize UML models to increase their readability
- We can use the UML package mechanism to organize classes into subsystems

![Packages Diagram]

- Any complex system can be decomposed into subsystems, where each subsystem is modeled as a package.
Object Modeling in Practice

Class Identification: Name of Class, Attributes and Methods

Is Foo the right name?

Object Modeling in Practice: Brainstorming

Is Foo the right name?
Object Modeling in Practice: More classes

1) Find New Classes
2) Review Names, Attributes and Methods

Object Modeling in Practice: Associations

1) Find New Classes
2) Review Names, Attributes and Methods
3) Find Associations between Classes
4) Label the generic associations
5) Determine the multiplicity of the associations
6) Review associations
Practice Object Modeling: Find Taxonomies

![Diagram showing the relationships between Bank, Account, Customer, Savings Account, Checking Account, and Mortgage Account.]

Practice Object Modeling: Simplify, Organize

![Diagram showing a simplified version of the relationships between Account, Savings Account, Checking Account, and Mortgage Account, with a note to show taxonomies separately.]
Practice Object Modeling: Simplify, Organize

Bank
Name

Account
Amount
AccountId
Deposit()
Withdraw()
GetBalance()

Customer
Name
CustomerId()

* Has

Use the 7+2 heuristics
or better 5+2!

UML Sequence Diagram
Sequence diagrams represent the behavior of a system as messages ("interactions") between different objects.

**Sequence Diagrams**

- **Focus on Controlflow**
  - Used during analysis
    - To refine use case descriptions
    - To find additional objects ("participating objects")
  - Used during system design
  - Used to refine subsystem interfaces

- **Messages** are represented by lines
- **Operations on participating Object** are represented by rectangles
- **Activations** are represented by narrow rectangles.

Message

- pressButton1()
- pressButton1()
- pressButton2()
- pressButton1and2()

Lifeline

- :WatchUser
- :Watch
- :LCDDisplay
- :Time

Activation

- blinkHours()
- blinkMinutes()
- incrementMinutes()
- refresh()
- commitNewTime()
- stopBlinking()
Scenarios, use case and sequence diagrams

• A scenario is an instance of a use case describing a concrete set of actions (no alternative paths are in it)
• A use case is an abstraction that describes all possible scenarios involving the described functionality.
• Scenarios are used as examples for illustrating common cases;
  • their focus is on understandability.
• Use cases are used to describe all possible cases;
  • their focus is on completeness.

How to describe scenarios

• We describe a scenario using a template with three fields:
  • The name of the scenario enables us to refer to it unambiguously. The name of a scenario is underlined to indicate that it is an instance.
  • The participating actor instances field indicates which actor instances are involved in this scenario. Actor instances also have underlined names.
  • The flow of events of a scenario describes the sequence of events step by step.
Scenario: an example

<table>
<thead>
<tr>
<th>Scenario name</th>
<th>warehouseOnFire</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating actor instances</td>
<td>bob, alice:FieldOfficer</td>
</tr>
<tr>
<td>Flow of events</td>
<td>1. Bob, driving down main street in his patrol car, notices smoke coming out of a warehouse. His partner, Alice, activates the &quot;Report Emergency&quot; function from her FRIEND laptop.</td>
</tr>
<tr>
<td></td>
<td>2. Alice enters the address of the building, a brief description of its location (i.e., northwest corner), and an emergency level. In addition to a fire unit, she requests several paramedic units on the scene given that area appears to be relatively busy. She confirms her input and waits for an acknowledgment.</td>
</tr>
<tr>
<td></td>
<td>3. John, the Dispatcher, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He allocates a fire unit and two paramedic units to the incident site and sends their estimated arrival time (ETA) to Alice.</td>
</tr>
<tr>
<td></td>
<td>4. Alice receives the acknowledgment and the ETA.</td>
</tr>
</tbody>
</table>

Sequence Diagrams can also model the Flow of Data

- The source of an arrow indicates the activation which sent the message
- Horizontal dashed arrows indicate data flow, for example return results from a message

...continued on next slide...
Sequence Diagrams: Iteration & Condition

- Iteration is denoted by a * preceding the message name
- Condition is denoted by boolean expression in [ ] before the message name

Creation and destruction

- Creation is denoted by a message arrow pointing to the object
- Destruction is denoted by an X mark at the end of the destruction activation
  - In garbage collection environments, destruction can be used to denote the end of the useful life of an object.
Message Types

- Asynchronous
- Synchronous
- Call and Object creation
- Reply
- Lost
- Found

Sequence Diagram Properties

- UML sequence diagram represent behavior in terms of interactions
- Useful to identify or find missing objects
- Time consuming to build, but worth the investment
- Complement the class diagrams (which represent structure).
UML first pass: Statechart diagrams

Represent behavior of a single object with interesting dynamic behavior.
State machine diagram for 2Bwatch

Internal transitions in 2BWatch statechart
UML Activity Diagrams

An activity diagram consists of nodes and edges

- **Nodes describe activities and objects**
  - Control nodes
  - Executable nodes
    - Most prominent: *Action*
  - Object nodes
    - E.g. a document

- **Edge** is a directed connection between nodes
  - There are two types of edges
    - Control flow edges
    - Object flow edges
Example: Structure of the Text Book

Object Node

Problem Statement

Requirements elicitation (Ch. 4)

nonfunctional requirements

functional model

use case diagram

Analysis (Ch. 5)

class diagram

analysis object model

dynamic model

statechart diagram

sequence diagram

System design (Ch. 6 & 7)

Example: Structure of the Text Book (2)

System design (Ch. 6 & 7)

subsystem decomposition

design goals

Object design (Ch. 8 & 9)

class diagram

object design model

source code

Implementation (Ch. 10)
deliverable system

Test (Ch. 11)
Activity Diagrams: Grouping of Activities

- Activities may be grouped into *swimlanes* to denote the object or subsystem that implements the activities.

![Activity Diagram](image)

State Chart Diagrams vs Activity Diagrams

- An activity diagram that contains only activities can be seen as a special case of a state chart diagram.
- Such an activity diagram is useful to describe the overall workflow of a system.

![State Chart](image)
Statechart Diagram vs Activity Diagram

Statechart Diagram for Incident
Focus on the set of attributes of a single abstraction (object, system)

Activity Diagram for Incident
(Focus on dataflow in a system)

What should be done first? Coding or Modeling?

- It depends....
- Forward Engineering
  - Creation of code from a model
  - Start with modeling
  - Greenfield projects
- Reverse Engineering
  - Creation of a model from existing code
  - Interface or reengineering projects
- Roundtrip Engineering
  - Move constantly between forward and reverse engineering
  - Reengineering projects
  - Useful when requirements, technology and schedule are changing frequently.
Additional References

• Martin Fowler

• Grady Booch, James Rumbaugh, Ivar Jacobson

• Open Source UML tools
  • Astah Community:
    http://astah.net/editions/community
  • http://java-source.net/open-source/uml-modeling

UML Summary

• UML provides a wide variety of notations for representing many aspects of software development
  • Powerful, but complex

• UML is a programming language
  • Can be misused to generate unreadable models
  • Can be misunderstood when using too many exotic features

• We concentrated on a few notations:
  • Functional model: Use case diagram
  • Object model: class diagram
  • Dynamic model: sequence diagrams, statechart and activity diagrams.