UML: Unified Modeling Language
Modeling

• Describing a system at a high level of abstraction
  - A model of the system
  - Used for requirements and specification

• Many notations over time
  - State machines
  - Entity-relationship diagrams
  - Dataflow diagrams
Recent History: 1980's

- The rise of object-oriented programming
- New class of OO modeling languages
- By early '90's, over fifty OO modeling languages
Recent History: 1990’s

• Three leading OO notations decide to combine
  – Grady Booch (BOOCH)
  – Jim Rumbaugh (OMT: Object Modeling Technique)
  – Ivar Jacobsen (OOSE: OO Soft. Eng)

• Why?
  – Natural evolution towards each other
  – Effort to set an industry standard
UML

- UML stands for Unified Modeling Language

- Design by committee
  - Many interest groups participating
  - Everyone wants their favorite approach to be “in”
UML

• UML stands for **Unified Modeling Language**

• Design by committee
  - Many interest groups participating
  - Everyone wants their favorite approach to be “in”
UML (Cont.)

- Resulting design is huge
  - Many features
  - Many loosely unrelated styles under one roof

- Could also be called
  Union of all Modeling Languages
Objectives of UML

• UML is a general purpose notation that is used to
  • visualize
  • specify
  • construct and
  • document
  the artifacts of a software system
This and Next Lectures

• We discuss
  - Use Case Diagrams for functional models
  - Class Diagrams for structural models
  - Object Diagrams
  - Sequence Diagrams for dynamic models
  - Activity Diagrams
  - State Diagrams

• This is a subset of UML
  - But probably the most used subset
Development Process

• Requirements elicitation - High level capture of user/system requirements
  - Use Case Diagram
• Identify major objects and relationships
  - Object and class diagrams
• Create scenarios of usage
  - Class, Sequence and Collaboration diagrams
• Generalize scenarios to describe behavior
  - Class, State and Activity Diagrams
• Refine and add implementation details
  - Component and Deployment Diagrams
Structural Diagrams

- **Class Diagram** - set of classes and their relationships. Describes interface to the class (set of operations describing services)
- **Object Diagram** - set of objects (class instances) and their relationships
- **Component Diagram** - logical groupings of elements and their relationships
- **Deployment Diagram** - set of computational resources (nodes) that host each component
Behavioral Diagram

- **Use Case Diagram** - high-level behaviors of the system, user goals, external entities: actors
- **Sequence Diagram** - focus on time ordering of messages
- **Collaboration Diagram** - focus on structural organization of objects and messages
- **State (Machine) Diagram** - event driven state changes of system
- **Activity Diagram** - flow of control between activities
Use Case Diagram

- Elements
  - Actors
  - Use cases
  - Relations
- Use case diagram shows relationship between actors and use cases
Use Case Diagram Example

- **Ride**
  - **Diagnose**
    - **Repair**
      - **Business Class Ride**
      - **Economy Class Ride**
Example:

Project and Resource Management System

- A resource manager manages resources
- A project manager manages projects
- A system administrator is responsible for administrative functions of the system
- A backup system houses backup data for the system
Figure 4-1: High-Level Use Case Diagram
Do these Use Cases Pass the Tests?

- Boss test?
- EBP test?
- Size test?
Manage Project Use Case

• A project manager can add, remove, and update a project
• Remove and update project requires to find project
• A project update may involve
  - Add, remove, or update activity
  - Add, remove, or update task
  - Assign resource to a task or unassign resource from a task
Figure 4-3: Manage Projects Use Case Diagram
Class Diagrams

- Describe classes
  - In the OO sense
- Class diagrams are static -- they display what interacts but not what happens when they do interact
- Each box is a class
  - List fields
  - List methods

<table>
<thead>
<tr>
<th>Train</th>
</tr>
</thead>
<tbody>
<tr>
<td>lastStop</td>
</tr>
<tr>
<td>nextStop</td>
</tr>
<tr>
<td>velocity</td>
</tr>
<tr>
<td>doorsOpen?</td>
</tr>
<tr>
<td>addStop(stop);</td>
</tr>
<tr>
<td>startTrain(velocity);</td>
</tr>
<tr>
<td>stopTrain();</td>
</tr>
<tr>
<td>openDoors();</td>
</tr>
<tr>
<td>closeDoors();</td>
</tr>
</tbody>
</table>
Class Diagrams: Relationships

- Many different kinds of edges to show different relationships between classes
- Any examples?
## Relationships in UML

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Function</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>association</td>
<td>A description of a connection among instances of classes</td>
<td></td>
</tr>
<tr>
<td>dependency</td>
<td>A relationship between two model elements</td>
<td>- - - ≫</td>
</tr>
<tr>
<td>generalization</td>
<td>A relationship between a more specific and a more general description, used for inheritance and polymorphic type declarations</td>
<td></td>
</tr>
<tr>
<td>realization</td>
<td>Relationship between a specification and its implementation</td>
<td>- - - ≫</td>
</tr>
<tr>
<td>usage</td>
<td>A situation in which one element requires another for its correct functioning</td>
<td>«kind» ≫</td>
</tr>
</tbody>
</table>
Association

- Association between two classes
  - if an instance of one class must know about the other in order to perform its work.
- Label endpoints of edge with cardinalities
  - Use * for arbitrary
- Can be directional (use arrows in that case)
Association

```
[Diagram showing relationship between Subscription, Priority, and Reservation]
```

- Subscription has a binary association with Reservation
- Subscription has a reflexive association with Priority
- Priority has an association name
- Rolename: source
- Multiplicity: 0..1
- Participating class: Reservation
Examples of Association

person

Works For

company

person

-employee

1..*

-employer

*
Link Attributes

• Associations may have properties in the same manner as objects/classes

• Salary and job title can be represented as
Association

participating class

Organization

*                      donor                      *

Person

DonationLevel

yearAmount: Money
lifeAmount: Money

association class (all one element)

association attributes
Types of Association

Aggregation Composition

- **Subscription** (aggregate)
  - Performance (parts)

- **Order** (composite)
  - **CustomerInfo**
  - **LineItem** (parts)
Aggregation

- An association in which one class belongs to a collection
  - Shared: An object can exist in more than one collections
  - No ownership implied
- Denoted by hollow diamond on the “contains” side

Composition

- An association in which one class belongs to a collection
  - No Sharing: An object cannot exist in more than one collections
  - Strong “has a” relationship
  - Ownership
- Denoted by filled diamond on the “contains” side
Car

1

Wheels

4

Project

1

Consultant

1..*
Aggregation

Composition

CS435

Student

Millington

Classroom

1..*

1
Generalization

- Inheritance between classes
- Denoted by open triangle

Diagram:
- Button
  - RequestButton
  - EmergencyButton
Generalization

- **Shape**
  - **subclasses**
    - Polygon
    - Ellipse
    - Spline
    - \(\ldots\)
  - **generalizations**
  - **unseen subclasses**
Generalization

- **Abstract class**: `Order` with `date: Date` and `confirm()`
- **Concrete class**: `MailOrder` with `dateFilled: Date` and `confirm()`, and `BoxOfficeOrder` with `hold: Boolean` and `confirm()`
- **Superclass (parent)**: `Order`
- **Abstract operation**: `confirm()`
- **Generalization**: `MailOrder` and `BoxOfficeOrder` are derived from `Order`
Generalization

• (Think subclassing)

Doctor

Hospital Doctor

Cardiologist

General Practitioner
Generalization

Reservation
  targetDate: Date
  number: Integer
  confirm()

TimeStampedTransaction
  received: Time
  stamp()

TimeStampedReservation

This class inherits the attributes and operations of both of its parents.

No new features are needed by the child.
Generalization

• An is-a relationship
• Abstract class
Realization

```
ChoiceBlock
  setDefault(choice: Choice)
  getChoice(): Choice

Possible choices:
  Choice

specifie's

realization

implementation

PopupMenu
  setDefault (choice: Button)
  getChoice(): Button

RadioButtonArray
  setDefault (choice: Button)
  getChoice(): Button
  1..* choices

Button

String

1..* choices
```
Dependency

We use term dependencies for other relationships that do not fit sharper categories.
<table>
<thead>
<tr>
<th>Dependency</th>
<th>Function</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>access</td>
<td>A private import of the contents of another package</td>
<td>access</td>
</tr>
<tr>
<td>binding</td>
<td>Assignment of values to the parameters of a template to generate a new model element</td>
<td>bind</td>
</tr>
<tr>
<td>call</td>
<td>Statement that a method of one class calls an operation of another class</td>
<td>call</td>
</tr>
<tr>
<td>creation</td>
<td>Statement that one class creates instances of another class</td>
<td>create</td>
</tr>
<tr>
<td>derivation</td>
<td>Statement that one instance can be computed from another instance</td>
<td>derive</td>
</tr>
<tr>
<td>instantiation</td>
<td>Statement that a method of one class creates instances of another class</td>
<td>instantiate</td>
</tr>
<tr>
<td>permission</td>
<td>Permission for an element to use the contents of another element</td>
<td>permit</td>
</tr>
<tr>
<td>realization</td>
<td>Mapping between a specification and an implementation of it</td>
<td>realize</td>
</tr>
<tr>
<td>Concept</td>
<td>Definition</td>
<td>Action</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>instantiation</td>
<td>Statement that a method of one class creates instances of another class</td>
<td>instantiate</td>
</tr>
<tr>
<td>permission</td>
<td>Permission for an element to use the contents of another element</td>
<td>permit</td>
</tr>
<tr>
<td>realization</td>
<td>Mapping between a specification and an implementation of it</td>
<td>realize</td>
</tr>
<tr>
<td>refinement</td>
<td>Statement that a mapping exists between elements at two different semantic levels</td>
<td>refine</td>
</tr>
<tr>
<td>send</td>
<td>Relationship between the sender of a signal and the receiver of the signal</td>
<td>send</td>
</tr>
<tr>
<td>substitution</td>
<td>Statement that the source class supports the interfaces and contracts of the target class and may be substituted for it</td>
<td>substitute</td>
</tr>
<tr>
<td>trace dependency</td>
<td>Statement that some connection exists between elements in different models, but less precise than a mapping</td>
<td>trace</td>
</tr>
<tr>
<td>usage</td>
<td>Statement that one element requires the presence of another element for its correct functioning (includes call, creation, instantiation, send, and potentially others)</td>
<td>use</td>
</tr>
</tbody>
</table>
Example class diagram?

```
<table>
<thead>
<tr>
<th>window</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name : string(idl)</td>
</tr>
<tr>
<td>+open()</td>
</tr>
<tr>
<td>+close()</td>
</tr>
<tr>
<td>+display()</td>
</tr>
</tbody>
</table>

consolewindow

dialogbox

event

control
```
Which Relation is Right?

- **Aggregation** – aka is-part-of, is-made-of, contains
- Use **association** when specific (persistent) objects have multiple relationships (e.g., there was only one Bill Gates at MS and Steve Jobs at Apple)
- Use **dependency** when working with static objects, or if there is only one instance
- Do not confuse part-of with is-a
## Relationships in UML

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Function</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>association</td>
<td>A description of a connection among instances of classes</td>
<td></td>
</tr>
<tr>
<td>dependency</td>
<td>A relationship between two model elements</td>
<td></td>
</tr>
<tr>
<td>generalization</td>
<td>A relationship between a more specific and a more general description, used for inheritance and polymorphic type declarations</td>
<td></td>
</tr>
<tr>
<td>realization</td>
<td>Relationship between a specification and its implementation</td>
<td></td>
</tr>
<tr>
<td>usage</td>
<td>A situation in which one element requires another for its correct functioning</td>
<td></td>
</tr>
</tbody>
</table>
Figure 4-5: High-Level Project Class Diagram
Figure 4-7: Detailed Project Class Diagram
Figure 4-8: Detailed Activities and Tasks Class Diagram
Object Diagram

- Object diagram is an instantiation of a class diagram
- Represents a static structure of a system at a particular time
Figure 4-11: Project Object Diagram
Sequence Diagrams

• Sequence diagrams
  - Refine use cases
  - *Gives view of dynamic behavior of classes*
    • Class diagrams give the static class structure

• Not orthogonal to other diagrams
  - *Overlapping functionality*
  - True of all UML diagrams
Development Process

- Requirements elicitation – High level capture of user/system requirements
  - Use Case Diagram
- Identify major objects and relationships
  - Object and class diagrams
- Create scenarios of usage
  - Class, Sequence and Collaboration diagrams
- Generalize scenarios to describe behavior
  - Class, State and Activity Diagrams
- Refine and add implementation details
  - Component and Deployment Diagrams
UML Driven Process

Requirements Elicitation → Analysis → Specification → Design → Implementation

- Use Case Diagram
- Object Diagram
- Sequence Diagram
- Class Diagram
- State Chart
- Collaboration Diagram
- Deployment Diagram
- Activity Diagram
UML Driven Process Model
Work Products

- Functional Model – Use Case diagrams
- Analysis Object Model – simple object/class diagram
- Dynamic Model – State and Sequence diagrams
- Object Design Model – Class diagrams
- Implementation Model – Deployment, and Activity diagrams
Acknowledgements

• Many slides courtesy of Rupak Majumdar