Modelling with Classes

CITS1220 Software Engineering
Lecture Overview

- **Classes and UML**
- Associations between classes
- Special types of association: is-a, has-a, is-part-of
- Modelling Example
- Implementing associations in Java
What is UML?

- Unified Modelling Language
- a standard graphical language for modelling object-oriented software
  - 1980s / 1990s, the first object-oriented development processes
  - Rumbaugh and Booch merged their approaches in 1994
    - They worked together at the Rational Software Corporation
  - Jacobson joined team in 1995
    - His work focused on use cases
  - Object Management Group (OMG) started the process of UML standardization in 1997
UML diagrams

- **Class diagrams**
  - describe classes and their relationships
  - in CITS1220 we will focus on class diagrams

- **Interaction diagrams**
  - show the behaviour of systems in terms of how objects interact with each other

- **State diagrams and activity diagrams**
  - show how systems behave internally

- **Component and deployment diagrams**
  - show how the various components of systems are arranged logically and physically
UML class diagrams

- describe classes and their relationships

More UML next year in SE design:

- Interaction diagrams
  - show the behaviour of systems in terms of how objects interact with each other

- State diagrams and activity diagrams
  - show how systems behave internally

- Component and deployment diagrams
  - show how the various components of systems are arranged logically and physically
What constitutes a good model?

- A model should
  - use a standard notation
  - be understandable by clients and users
  - lead software engineers to have insights about the system
  - provide abstraction

- Models are used
  - to help create designs
  - to permit analysis and review of those designs.
  - as the core documentation describing the system
Essentials of UML Class Diagrams

- **Classes**
  - represent the types of data in the system

- **Associations**
  - represent links between instances of classes

- **Attributes**
  - are simple data found within classes and their instances

- **Operations**
  - represent the functions performed by the classes and their instances

- **Generalizations**
  - group classes into inheritance hierarchies
Classes

- A class is simply represented as a box with the name of the class inside
  - The diagram may also show the attributes and operations
  - The complete signature of an operation is:
    
    operationName(parameterName: parameterType ...): returnType

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

- Classes and UML
- Associations between classes
- Special types of association: is-a, has-a, is-part-of
- Modelling Example
- Implementing associations in Java
Associations and Multiplicity

- An association is used to show how two classes are related to each other (A has a B)
  - Symbols indicating multiplicity are shown at each end of the association
Labelling associations

Each association can be labelled, to make explicit the nature of the association.
Many-to-one associations

- A company has many employees,
- An employee can only work for one company.
  - This company will not store data about the moonlighting activities of employees!
- A company can have zero employees
  - E.g. a ‘shell’ company
- It is not possible to be an employee unless you work for a company
Many-to-many associations

- An assistant can work for many managers
- A manager can have many assistants
- Assistants can work in pools
- Managers can have a group of assistants
- Some managers might have zero assistants.
- Is it possible for an assistant to have, perhaps temporarily, zero managers?

Assistant  *  Manager

1..* supervisor
One-to-one associations

- For each company, there is exactly one board of directors
- A board is the board of only one company
- A company must always have a board
- A board must always be of some company
Analyzing and validating associations

- Avoid unnecessary one-to-one associations

- Avoid this                   do this

```
Person
 name 1

PersonInfo
 address
 email
 birthdate 1

Person
 name
 address
 email
 birthdate
```
A more complex example

- A booking is always for exactly one passenger
  - no booking with zero passengers
  - a booking could *never* involve more than one passenger.
- A Passenger can have any number of Bookings
  - a passenger could have no bookings at all
  - a passenger could have more than one booking

The *frame* around this diagram is an optional feature that any UML 2.0 may possess.
Theatre Company Example

- One play (eg Hamlet) is performed at many different performances (eg 10 Oct 2007 at 8pm and 10 Oct 07 at 2pm)
- Alternatively, more than one play may be performed at a single performance (not modelled here)
- Each performance takes place in a particular theatre (eg His Majesty’s)
- The same theatre hosts many different performances (eg Hamlet on Monday, Romeo and Juliet on Tuesday)
- There is a many to many relationship between plays and theatres, but this is implied through performances, not modelled separately
- Implementation: Performance needs to know about Play but not vice versa
- Each theatre keeps a list of performances, each performance refers to the theatre it takes place in
Association classes

- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes.
- The following are equivalent (we will use the 2nd).
Reflexive associations

- It is possible for an association to connect a class to itself
Directionality in associations

- Associations are by default *bi-directional*
- It is possible to limit the direction of an association by adding an arrow at one end

![Diagram](image)
Lecture Overview

- Classes and UML
- Associations between classes
- Special types of association: is-a, has-a, is-part-of
- Modelling Example
- Implementing associations in Java
Generalization

- Specializing a superclass into two or more subclasses *(A is a B)*
  - A *generalization set* is a labelled group of generalizations with a common superclass
  - The label (sometimes called the *discriminator*) describes the criteria used in the specialization
Avoiding unnecessary generalizations

Inappropriate hierarchy of classes, which should be instances

- Recording
  - VideoRecording
    - MusicVideo
  - AudioRecording
    - JazzRecording
    - ClassicalRecording
    - BluesRecording
    - RockRecording
Avoiding unnecessary generalizations (cont)

Improved class diagram, with its corresponding instance diagram
More Advanced Features: Aggregation

- Aggregations are special associations that represent ‘part-whole’ relationships.
  - The ‘whole’ side is often called the \textit{assembly} or the \textit{aggregate}
  - This symbol is a shorthand notation association named \texttt{isPartOf}

\begin{center}
\begin{tikzcd}
\text{Vehicle} & \text{VehiclePart} \\
\text{Country} & \text{Region}
\end{tikzcd}
\end{center}
When to use an aggregation

- As a general rule, you can mark an association as an aggregation if the following are true:
  - You can state that
    - the parts ‘are part of’ the aggregate
    - or the aggregate ‘is composed of’ the parts
  - When something owns or controls the aggregate, then they also own or control the parts
Lecture Overview

- Classes and UML
- Associations between classes
- Special types of association: is-a, has-a, is-part-of
- Modelling Example
- Implementing associations in Java
Detailed Example: A Class Diagram for Genealogy

- Problems
  - A person must have two parents
  - Marriages not properly accounted for
Two possible solutions
A simple technique for discovering domain classes

- Look at a source material such as a description of requirements
- Extract the *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 represent *types of users* or other actors
Identifying associations and attributes

- Start with classes you think are most central and important.
- Decide on the clear and obvious data it must contain and its relationships to other classes.
- Work outwards towards the classes that are less important.
- Avoid adding many associations and attributes to a class.

- A system is simpler if it manipulates less information.
Tips for identifying and specifying valid associations

- An association should exist if a class:
  - possesses
  - controls
  - is connected to
  - is related to
  - is a part of
  - has as parts
  - is a member of, or
  - has as members

  some other class in your model

- Specify the multiplicity at both ends
- Label it clearly
A common mistake is to represent *actions* as if they were associations.

**Bad**: The *borrow* operation creates a *Loan*, and the *return* operation sets the *returnedDate* attribute.

**Better**: The *borrow* operation creates a *Loan*, and the *return* operation sets the *returnedDate* attribute.
Identifying attributes

- Look for information that must be maintained about each class
- Several nouns rejected as classes, may now become attributes
- An attribute should generally contain a simple value
  - E.g. string, number
Tips for identifying and specifying valid attributes

- It is not good to have many duplicate attributes

| 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. |

Person
- name
- addresses

Address
- street
- municipality
- provOrState
- country
- postalCode
- type

Person
- name

1 addresses

Person
- name

*
Attributes and associations

- Passenger
  - name
  - number
  - Booking
    - seatNumber

- Employee
  - name
  - employeeNumber
  - jobFunction
  - "0..1 supervisor"
  - SpecificFlight
    - date

- RegularFlight
  - time
  - flightNumber
Identifying generalizations

- **bottom-up**
  - Group together similar classes creating a new superclass

- **top-down**
  - Look for more general classes first, specialize them if needed
An example (generalization)
Lecture Overview

- Classes and UML
- Associations between classes
- Special types of association: is-a, has-a, is-part-of
- Modelling Example
- Implementing associations in Java
Implementation in Java

- Attributes are implemented as instance variables
- Generalizations are implemented using extends
- Interfaces are implemented using implements
- Associations are normally implemented using instance variables
Implementing Associations

- Divide each two-way association into two one-way associations
  - so each associated class has an instance variable.
- For a one-way association where the multiplicity at the other end is ‘one’ or ‘optional’
  - declare a variable of that class (a reference)
- For a one-way association where the multiplicity at the other end is ‘many’:
  - use a collection class implementing List, such as Vector
Implementation Examples

Two way, many to many

One way, one to one

```java
class union { ... person wife; }
```

One way, one to many

```java
class union { ...
  ArrayList<person> children;
}
```
Classes A and B are associated if:

- An object of class A creates an object of class B
- An object of class A has an attribute whose values are objects of class B, or collections of objects of class B
- An object of class A sends a message to an object of class B
- An object of class A receives a message with an object of class B as an argument
Example: Genealogy

```java
public class person {
    public static final String male="male";
    public static final String female="female";

    private String name;
    private String sex;
    private String placeOfBirth;
    private Date dateOfBirth;
    private Date dateOfDeath;

    private union parents; //birth parents
    private union adoptiveparents;
    private ArrayList<union> marriages;

    public person(String name, String sex, String placeOfBirth,
                   Date dateOfBirth, Date dateOfDeath) {...}
}
```
Example: Genealogy

```java
public class union {

    private String placeOfMarriage;
    private Date dateOfMarriage;
    private Date dateOfDivorce;
    private person husband;
    private person wife;
    private ArrayList<person> children;

    public union (String placeOfMarriage,
                 Date dateOfMarriage,
                 person husband,
                 person wife) {…}

    public void addChild(person child) { …}
    public void makeDivorce(Date dateOfDivorce) {…}
}
```