Association, Aggregation and Composition

Software Engineering
CITS1220
Lecture Outline

1. Associations in UML
2. Aggregation and Composition
3. Examples in Java
Reuse

- A major motivation for OOP is reusability
- Take advantage of the previous work of yourself and others
- Reuse tried and tested designs and debugged and code
- Build classes/objects from “software components”
Ways of reusing

- Cut and paste (sometimes called cloning) is dangerous as bugs are duplicated
- We look at associations today
  - General associations
  - Aggregation
  - Composition
- Inheritance later in semester
Associations

- An association is a relationship between two classes
  - Describes how instances of one class refer to instances of another class
  - Indicated in UML by connecting two classes with a line (optionally labelled)

![Diagram]

```
Employee <-> worksFor <-> Company
```
Multiplicities

- Associations should indicate their multiplicities
- Symbol near a class B indicates “this many instances of B” involved in relationship
Multiplicities II

- The symbol * means “any number including 0”
  - Each employee works for exactly 1 company
  - Each company has 0 or more employees
Ranges

- The symbol .. is used to indicate ranges
  - Managers may have 0, 1 or more PAs
  - Each PA must work for at least one manager, but maybe more
In Java

- In Java, an association is reflected by one class having variables referring to other classes

```java
public class Company {
    private String companyName;
    private Employee[] staff;
}
```
An Association

- A `MobilePhoneGUI` displays exactly one `MobilePhone`
- A `MobilePhone` may (or may not) be displayed by a `MobilePhoneGUI`

 nhấndiagram

(Uni)directional association
Aggregation & Composition

- Special forms of association reflecting the “is composed of” or “is built from” relationship

```
Vehicle              1
                  *           VehiclePart
                  
MobilePhoneGUI      1
                  *           JButton
```
What’s the difference?

- Book gives a very subtle difference between aggregation and composition
  - In aggregation the “parts” have an independent existence and can be shared or reassigned between aggregates
  - In composition, the parts BELONG to exactly one aggregate, are created when the aggregate is created and destroyed when the aggregate is destroyed
Terminology

- Association, aggregation and composition overlap and authors often blur the differences
  - Many would use “aggregation” for the MobilePhoneGUI example
- It is common to simply use “composition”
  - We will not make fine distinctions in this unit
Java Example

- Consider a class to represent a cylinder
  - A circular base and a height

```
Cylinder
  base: Circle
  height: double
data volume(): double

Circle
  radius: double
data area(): double
```
Existing Circle code

```java
public class Circle {

    private double radius;

    public Circle(double r) {
        radius = r;
    }

    public double area() {
        return (Math.PI * radius * radius);
    }
}
```
public class Cylinder {

    private Circle base;
    private double height;

    // constructor with two arguments
    public Cylinder(double r, double h) {
        base = new Circle(r);
        height = h;
    }
}
.. and uses its methods

// method for computing volume of cylinder
public double volume() {
    // call the area() method
    return (base.area() * height);
}
}
Constructing aggregate objects

- If a class A has instance variables referring to another class B, then when and how do the objects of class B get constructed?
  - Internally - The constructor for A constructs all required instances of B
  - Externally - The client passes instances of B via constructor arguments
Internal Construction

- Usual model for the “filled diamond” type of composition
  - Cylinder calls constructor for Circle
- In Java there is no need for the “parts” to be destroyed explicitly as this happens automatically
  - In C++ the destructor for Cylinder would need to call the destructor for Circle
External Construction

```java
public class MobilePhoneGUI {
    private MobilePhone myPhone;
    // Lots of GUI stuff
}
```

- How is `myPhone` set?
External Construction II

// Constructor
public MobilePhoneGUI (MobilePhone mp) {
    myPhone = mp;
}

- Recall
  - This saves the value of the parameter variable \texttt{mp} into the instance variable \texttt{myPhone}
  - There must be an already-created instance of \texttt{MobilePhone} available when the \texttt{MobilePhoneGUI} is constructed
Standard Java cliché

- It is normal to just use the same name for the parameter and instance variable

```java
// Constructor
public MobilePhoneGUI (MobilePhone myPhone) {
    this.myPhone = myPhone;
}
```

- Avoids proliferation of “single-use” variable names
Mutual knowledge

How do we ensure that two objects each know about the other?

This is a particular issue for GUI design

- The GUI object constructs UI components such as buttons
- Users click buttons
- GUI object needs to respond and so the button needs to know about the GUI object
Listeners

- The standard mechanism for this is the use of listeners - each UI component keeps a list of “who is interested in me”

```java
JButton close = new JButton(“C”);
closeButton.addMouseListListener(this);
```
GUI object knows about the JButton because it created the button, and then passes the button a reference to itself! Now the button knows who created it - more importantly it knows who is interested in mouse events!
Events

- A `MouseEvent` occurs whenever the user does something mouse-related with a component, for example clicks the button

- When this happens:
  - The button *notifies* all of its registered listeners - in this case it notifies the GUI object that it has been clicked