Advanced Programming

CSE 201

Instructor: Sambuddho

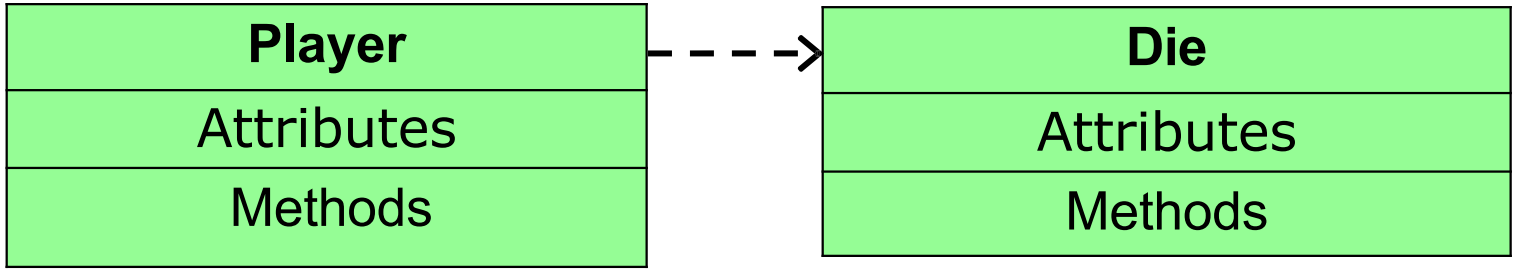
(Semester: Monsoon 2025)

Week 4 - Relationships

UML: Quick Introduction

- UML stands for the Unified Modeling Language
  - We will cover this in depth in later lectures
- Much detailed than sequence diagrams
- UML diagrams show relationships among classes and objects
  - Lines connecting the classes
- A UML class diagram consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)

A Sample UML Class Diagram



Class Relationships

- *The whole point of OOP is that your code replicates real world objects, thus making your code readable and maintainable.*
- *When we say real world, the real world has relationships.*
- *When writing a program, need to keep in mind "big picture"—how are different classes related to each other?*

Most Common Class Relationships

- **Composition**
  - *A “contains” B - “part-of” relationship*
- **Association/Aggregation**
  - *A “knows-about” B - “uses-a” or “has-a” relationship*
- **Dependency**
  - *A “depends on” B - Sort of “depends-on” relationship*
- **Inheritance**
  - *HarleyDavidson “is-a” Bike*

Composition Relationship

- *Class A* **contains** *object of class B*
  - *A* **instantiate** *B*
- *Thus A knows about B and can call methods on it*
- *But this is not symmetrical!*
  - *B can't automatically call methods on A*

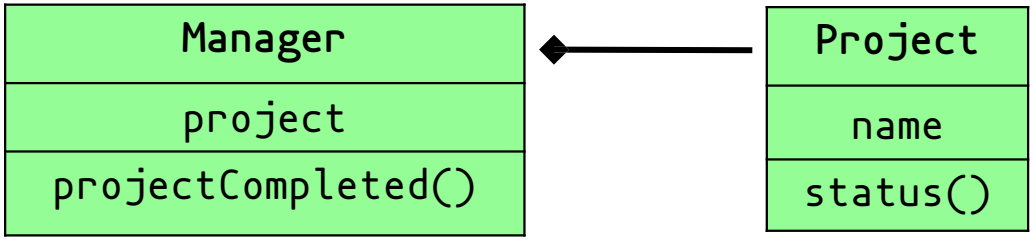
- *Lifetime?*
  - *The death relationship*
  - *Garbage collection of A means B also gets garbage collected*

Composition in UML

- Represented by a solid arrow with diamond head
- In below UML diagram, A is composed of B

Composition Example (1/2)

- Manager is fixed for a project and is responsible for the timely completion of the project. If manager leaves, project is ruined




```
class Project { private String name;
   public boolean status() { ... }
   .....
}
// A manager is fixed for a project
class Manager {
   private Project project;
   public Manager() {
     this.project = new Project("ABC");
   }
   public boolean projectCompleted() {
     return project.status();
   }


}
```

Composition Example (2/2)

- Pecause PetShop itself instantiates a DogGroomer with
  - "new DogGroomer();"
- Since PetShop created a DogGroomer and stored it in an instance variable, all PetShop's methods "know" about the _groomer and can access it _groomer and can access it

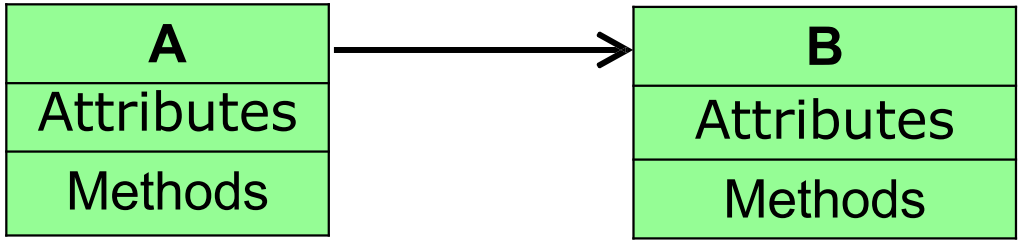
```
public class PetShop {

  private DogGroomer _groomer;

  public PetShop() {
    _groomer = new DogGroomer();
    this.testGroomer();
  }

  public void testGroomer() {
    Dog django = new Dog();//local var
    _groomer.groom(django);
  }

}
```

Association Relationship

- Association is a relationship between two objects
- Class A and class B are **associated** if A “knows about” B, but B is not a component of A
- But this is not symmetrical!
- **Class A holds a class level reference to class B**
- Lifetime?
  - Objects of class A and B have their own lifetime, i.e., they can exist without each other

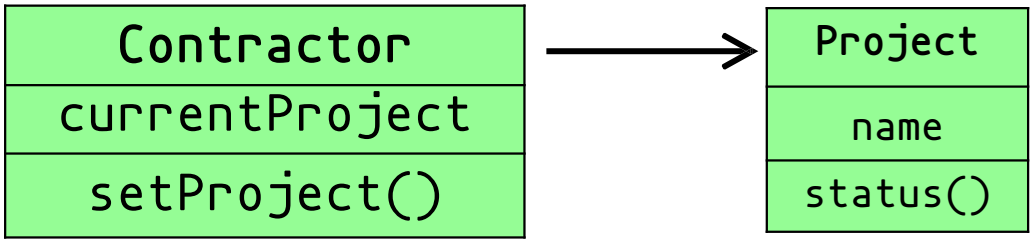
Association in UML

- Represented by a solid arrow
- In below UML diagram, A holds a reference of B



Association Example (1/4)

- A contractor's project keep's changing as per company's policy and contractor's performance




```
class Project { private String name;
   public boolean status() { ... }
   .....
}
// Contractor's project keep changing
class Contractor {
   private Project currentProject;
   public Contractor(Project proj) {
     this.currentProject = proj;
   }
   public void setProject(Project proj){
     this.currentProject = proj;
   }

}
```

Associations Example (2/4)

- **Association** *means that one object knows about another object that is not one of its components.*

```
public class DogGroomer {


  public DogGroomer() {
    // this is the constructor!
  }

  public void groom(Dog shaggyDog) {
    shaggyDog.setHairLength(1);
  }
}
```

Associations Example (2/4)

- As noted, PetShop contains a DogGroomer, so it can send messages to the DogGroomer
- But what if the DogGroomer needs to send messages to the PetShop she works in?
  - The DogGroomer probably needs to know several things about her PetShop: for example, operating hours, grooming supplies in stock, customers currently in the shop...

```
public class DogGroomer {


  public DogGroomer() {
    // this is the constructor!
  }

  public void groom(Dog shaggyDog) {
    shaggyDog.setHairLength(1);
  }
}
```

Associations Example (2/4)

- The PetShop keeps track of such information in its properties.
- Can set up an **association** so that DogGroomer can send her PetShop messages to retrieve information she needs.

```
public class DogGroomer {


  public DogGroomer() {
    // this is the constructor!
  }

  public void groom(Dog shaggyDog) {
    shaggyDog.setHairLength(1);
  }
}
```

Associations Example (2/4)

- *This is what the full association looks like*
- *Let's break it down line by line*
- **But note we're not yet making use of the association in this fragment**

```
public class DogGroomer {

   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
     _petShop = myPetShop; // store the assoc.
   }

   public void groom(Dog shaggyDog)
     { shaggyDog.setHairLength(1);
   }
}
```

Associations Example (2/4)

- We declare an instance variable named `_petShop`
- We want this variable to record the instance of `PetShop` that the `DogGroomer` belongs to

```
public class DogGroomer {

   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
     _petShop = myPetShop; // store the assoc.
   }

   public void groom(Dog shaggyDog)
     { shaggyDog.setHairLength(1);
   }
}
```

Associations Example (2/4)

- Modified DogGroomer's constructor to take in a parameter of type PetShop
- Constructor will refer to it by the name myPetShop
- Whenever we instantiate a DogGroomer, we'll need to pass it an instance of PetShop as an argument. Which? The PetShop instance that created the DogGroomer, hence use this

```
public class DogGroomer { private PetShop
   _petShop;

   public DogGroomer(PetShop myPetShop) {
      _petShop = myPetShop; // store the
      assoc.
   }
}  //groom method elided
```

```
public class PetShop {
   private DogGroomer _groomer;

   public PetShop() {
      _groomer = new DogGroomer(this);
      this.testGroomer();
   }

}  //testGroomer() elided
```

Associations Example (2/4)

- Now store myPetShop in instance variable _petShop
- _petShop now points to same PetShop instance passed to its constructor
- After constructor has been executed and can no longer reference myPetShop, any DogGroomer method can still access same PetShop instance by the name _petShop

```
public class DogGroomer {

   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
     _petShop = myPetShop; // store the assoc.
   }

   public void groom(Dog shaggyDog)
     { shaggyDog.setHairLength(1);
   }
}
```

Associations Example (2/4)

- Let's say we've written an accessor method and a mutator method in the PetShop class: getClosingTime() and setNumCustomers(int customers)
- If the DogGroomer ever needs to know the closing time, or needs to update the number of customers, she can do so by calling
  - getClosingTime()
  - setNumCustomers(int customers)

```
public class DogGroomer {

   private PetShop _petShop;
   private Time _closingTime;

   public DogGroomer(PetShop myPetShop) {
     _petShop = myPetShop; // store assoc.
     _closingTime = myPetShop.getClosingTime();
     _petShop.setNumCustomers(10);
   }
}
```

Association: Under the Hood (1/5)

```
public class PetShop {
   private DogGroomer _groomer;

   public PetShop() {
     _groomer = new DogGroomer(this);
     this.testGroomer();
   }

   public void testGroomer() {
     Dog django = new Dog();
     _groomer.groom(django);
                }
}
```

```
public class DogGroomer {
   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
     _petShop = myPetShop;
   }

   /* groom and other methods elided for this
   example */
}
```

Somewhere in memory...

Association: Under the Hood (2/5)

```
public class PetShop {
   private DogGroomer _groomer;

   public PetShop() {
     _groomer = new DogGroomer(this);
     this.testGroomer();
   }

   public void testGroomer() {
     Dog django = new Dog();
     _groomer.groom(django);
                 }
}
```

```
public class DogGroomer {
   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
     _petShop = myPetShop;
   }

   /* groom and other methods elided for this
   example */
}
```

Somewhere in memory...

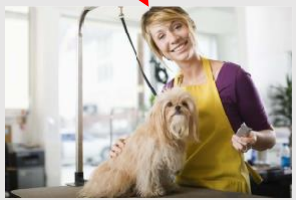
Somewhere else in our code, someone calls new PetShop(). An instance of PetShop is created somewhere in memory and PetShop's constructor initializes all its instance variables (just a DogGroomer here).

Association: Under the Hood (3/5)

```
public class PetShop {
   private DogGroomer _groomer;

   public PetShop() {
      _groomer = new DogGroomer(this);
      this.testGroomer();
   }

   public void testGroomer() {
      Dog django = new Dog();
      _groomer.groom(django);
                  }
}
```

```
public class DogGroomer {
   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
      _petShop = myPetShop;
   }

   /* groom and other methods elided for this
   example */
}
```

Somewhere in memory...



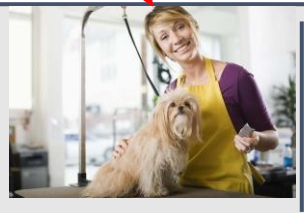
The PetShop instantiates a new DogGroomer, passing itself in as an argument to the DogGroomer's constructor (remember the this keyword?)

Association: Under the Hood (4/5)

```
public class PetShop {
   private DogGroomer _groomer;

   public PetShop() {
      _groomer = new DogGroomer(this);
      this.testGroomer();
   }

   public void testGroomer() {
      Dog django = new Dog();
      _groomer.groom(django);
                  }
}
```

```
public class DogGroomer {
   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
      _petShop = myPetShop;
   }

   /* groom and other methods elided for this
   example */
}
```

Somewhere in memory...



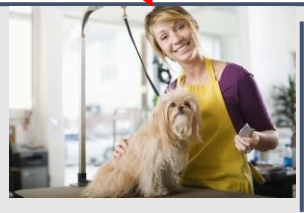
When the DogGroomer's constructor is called, its parameter, myPetShop, points to the same PetShop that was passed in as an argument.

Association: Under the Hood (5/5)

```
public class PetShop {
   private DogGroomer _groomer;

   public PetShop() {
      _groomer = new DogGroomer(this);
      this.testGroomer();
   }

public void testGroomer()
      { Dog django = new
      Dog();
      _groomer.groom(django);
   }
```

```
public class DogGroomer {
   private PetShop _petShop;

   public DogGroomer(PetShop myPetShop) {
      _petShop = myPetShop;
   }

   /* groom and other methods elided for this
   example */
}
```

Somewhere in memory...



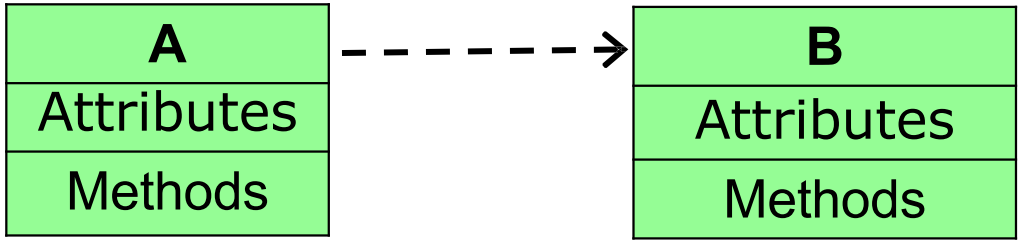
The DogGroomer sets its _petShop instance variable to point to the same PetShop it received as an argument. Now it "knows about" the petShop that instantiated it! And therefore so do all its methods.

Dependency

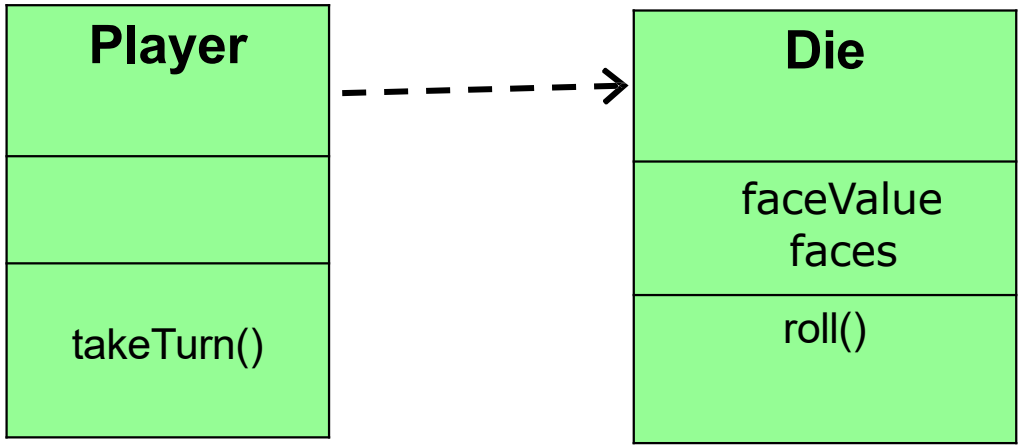
- Class A **depends** on class B if A cannot carry out its work without B, but B is neither a component of A nor it has association with A
- **A is requesting service from an object of class B**
  - A or B **"doesn't know"** about each other (no association)
  - A or B **"doesn't contain"** each other (no composition)
- But this is not symmetrical! B doesn't depends on A

Dependency in UML

- Represented by a dashed arrow starting from the dependent class to its dependency
  - A is dependent on B
  - A is requesting service from B

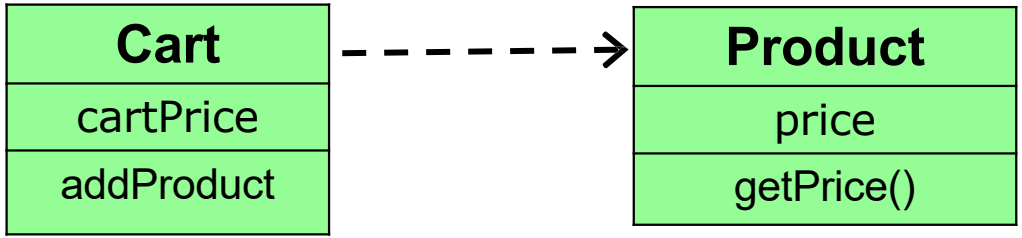


Dependency Example (1/3)



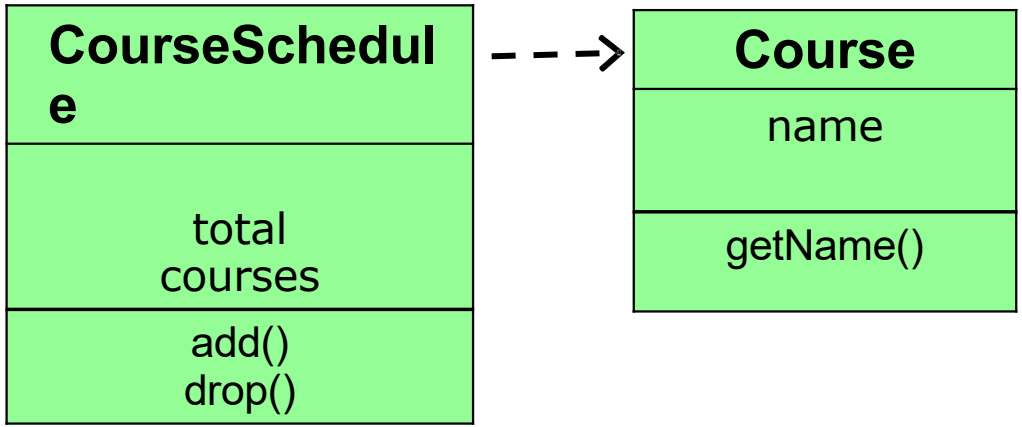

```
class Die {
   private int faceValue, faces;
   .....
   public void roll() { ..... }
}

class Player {
   public void takeTurn(Die die) {
     die.roll();
   }
}
```

Dependency Example (2/3)




```
class Product {
   private double price;
   .....
   public double getPrice() { ..... }
}

class Cart {
   private double cartPrice;
   public void addProduct(Product p) {
     cartPrice += p.getPrice();
   }
}
```

Dependency Example (3/3)




```
class Course {
   private String name;
   .....
   public String getName() { ..... }
}

class CourseSchedule { private int
   total; private String courses[];
   public void addCourse(Course c) {
     courses[total++] = c.getName();
   }
   .....
}
```