UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SOFTWARE ENGINEERING – LECTURE 28

SOFTWARE DESIGN EXERCISES

Prof. Luigi Libero Lucio Starace

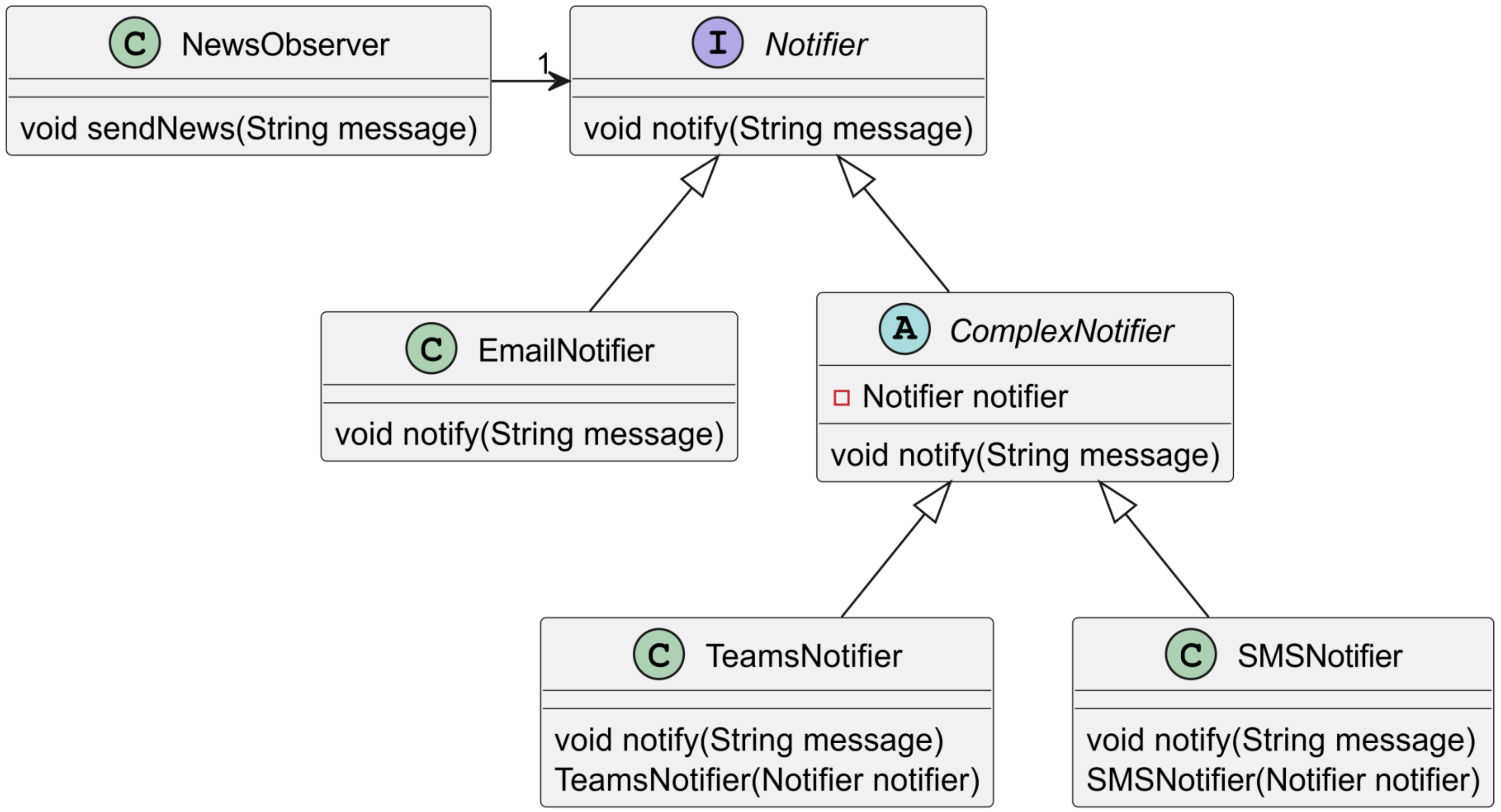
luigiliberolucio.starace@unina.it

https://www.docenti.unina.it/luigiliberolucio.starace

MIDTERM COMING UP!

- Midterm exam will take place next **Thursday, November 13**
- Tentatively starting at **09.00** in **Room CL-II-1**
- **Did you sign up for the midterm?**
- Facsimile published on Teams
  - Have you checked that out?

EXERCISE #1: IS THERE ANY KNOWN PATTERN?


C NewsObserver
void sendNews(String message)
1
I Notifier
void notify(String message)
C EmailNotifier
void notify(String message)
A ComplexNotifier
Notifier notifier
void notify(String message)
C TeamsNotifier
void notify(String message)
TeamsNotifier(Notifier notifier)
C SMSNotifier
void notify(String message)
SMSNotifier(Notifier notifier)

EXERCISE #1: IS THERE ANY KNOWN PATTERN?

- What if I gave you some source code?

```
public class Main {
    public static void main(String[] args) {
        Notifier notifier = new EmailNotifier();
        notifier = new SMSNotifier(notifier);
        notifier = new TeamsNotifier(notifier);


        NewsObserver observer = new NewsObserver(notifier);
        observer.sendNews("University exams postponed due to soccer match.");
    }
}
```

EXERCISE #2: IS THIS CODE SOLID?

- Which SOLID principles, if any, are violated by the following code?

```java
class GameCharacter {
    public void attack() {
        System.out.println("Character attacks!");
    }
    public void heal() {
        System.out.println("Character heals!");
    }
}
```

```java
class Healer extends GameCharacter {
    @Override
    public void attack() {
        throw new UnsupportedOperationException("Healers cannot attack!");
    }
    @Override
    public void heal() {
        System.out.println("Character heals whole party!");
    }
}
```

EXERCISE #2: IS THIS CODE SOLID?

- This code violates Liskov's Substitution Principle!

```java
class GameCharacter {
    public void attack() {
        System.out.println("Character attacks!");
    }
    public void heal() {
        System.out.println("Character heals!");
    }
}
```

```java
class Healer extends GameCharacter {
    @Override
    public void attack() {
        throw new UnsupportedOperationException("Healers cannot attack!");
    }
    @Override
    public void heal() {
        System.out.println("Character heals whole party!");
    }
}
```

EXERCISE #2: IS THIS CODE SOLID?

- This code violates Liskov's Substitution Principle!

```
public class Game {
  public static void main(String[] args) {
    GameCharacter character = new Healer(); // LSP violation: Healer not substitutable
    character.attack(); // Raises exception
}
```

- If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering the correctness of the program.

EXERCISE #3: IS THIS CODE SOLID?

- Which SOLID principles, if any, are violated by the following code?

```java
public class DamageCalculator {
    public int calculateDamage(String characterType, int baseDamage) {
        if (characterType.equals("Warrior")) {
            return baseDamage + 10;
        } else if (characterType.equals("Mage")) {
            return baseDamage + 5;
        } else if (characterType.equals("Archer")) {
            return baseDamage + 7;
        } else {
            return baseDamage;
        }
    }
}
```

EXERCISE #3: IS THIS CODE SOLID?

- This code violates the Open-closed Principle!

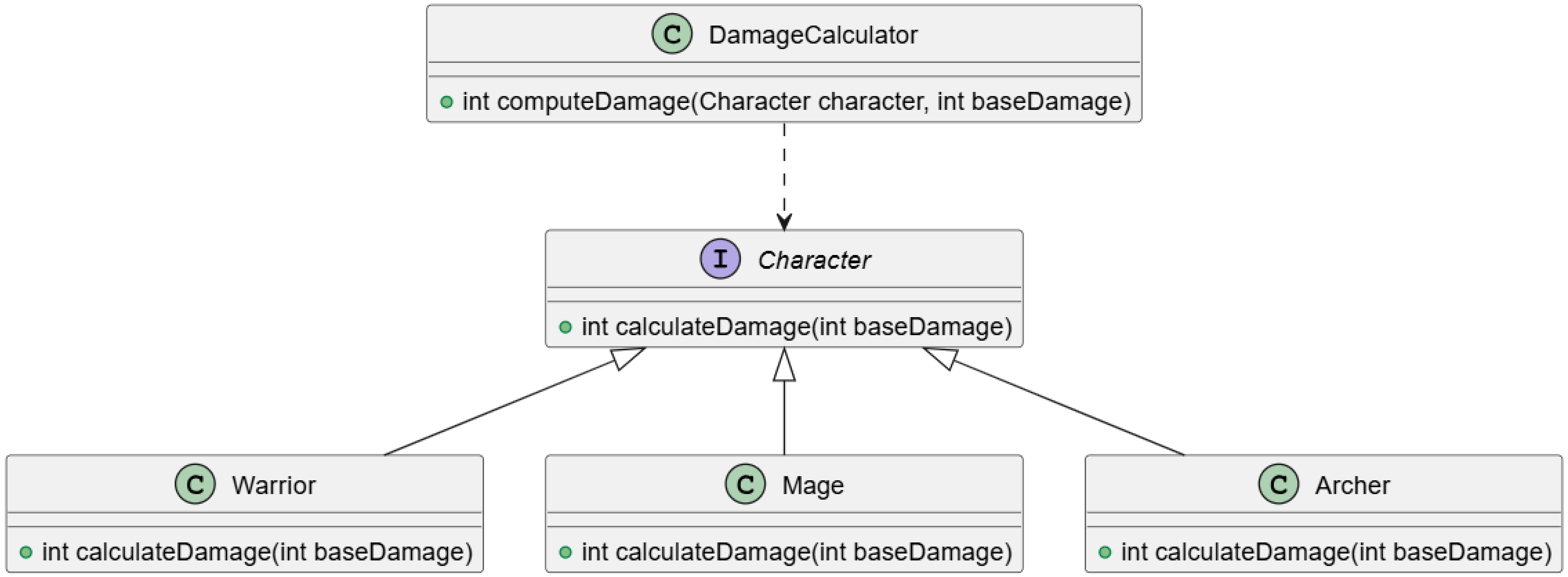
```java
public class DamageCalculator {
    public int calculateDamage(String characterType, int baseDamage) {
        if (characterType.equals("Warrior")) {
            return baseDamage + 10;
        } else if (characterType.equals("Mage")) {
            return baseDamage + 5;
        } else if (characterType.equals("Archer")) {
            return baseDamage + 7;
        } else {
            return baseDamage;
        }
    }
}
```

Software should be:

- **Open for extension**: You should be able to add new behavior or capabilities to a module.
- **Closed for modification**: You should not have to change existing source code to do so.

EXERCISE #3: IS THIS CODE SOLID?

- How can we improve the code?



DamageCalculator
int computeDamage(Character character, int baseDamage)
Character
int calculateDamage(int baseDamage)
Warrior
int calculateDamage(int baseDamage)
Mage
int calculateDamage(int baseDamage)
Archer
int calculateDamage(int baseDamage)

EXERCISE #3: IS THIS CODE SOLID?

- How can we improve the code?

```
public class DamageCalculator {
  public int computeDamage(Character character, int baseDamage) {
    return character.calculateDamage(baseDamage);
  }

  public static void main(String[] args) {
    DamageCalculator calculator = new DamageCalculator();
    Character warrior = new Warrior();
    Character mage = new Mage();
    Character archer = new Archer();

    System.out.println("Warrior damage: " + calculator.calculate(warrior, 50));
    System.out.println("Mage damage: " + calculator.calculate(mage, 50));
    System.out.println("Archer damage: " + calculator.calculate(archer, 50));
  }
}
```

EXERCISE #4: IS THIS CODE SOLID?

- Which SOLID principles, if any, are violated by the following code?

```java
public class PizzaDeliveryCat {
    private Scooter scooter = new Scooter();

    public void deliver(String address) {
        scooter.driveTo(address);
    }
}


class Scooter {
    public void driveTo(String address) {
        System.out.println("Scooter zooms to " + address);
    }
}
```

EXERCISE #4: IS THIS CODE SOLID?

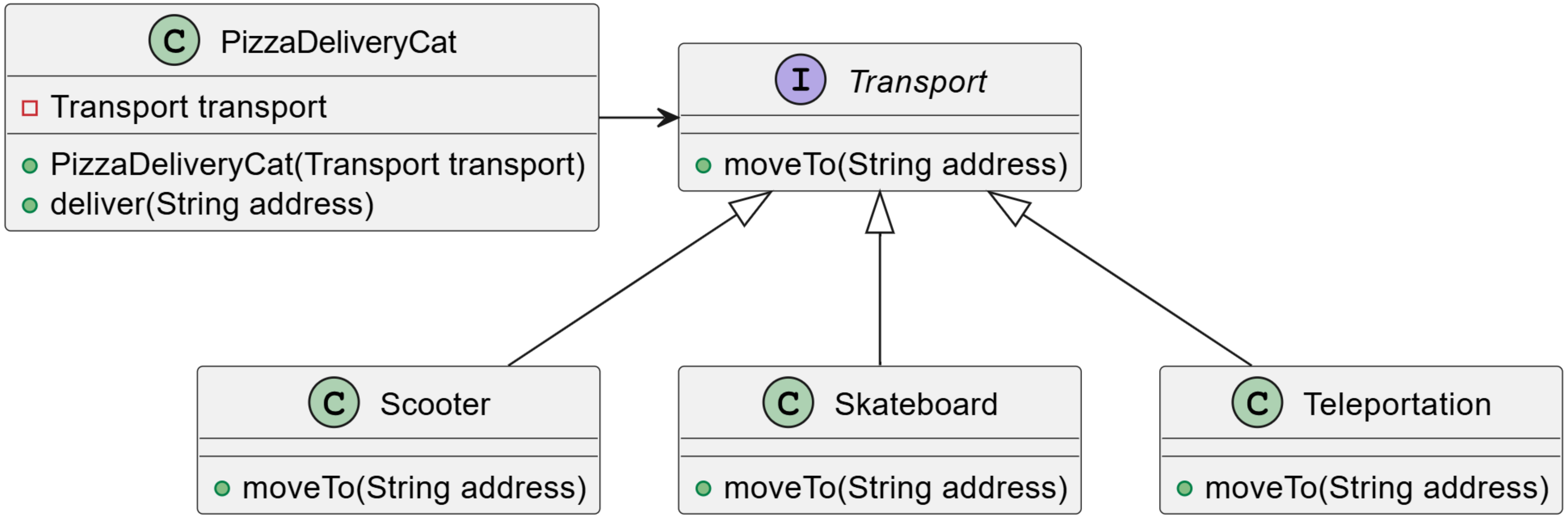
- The code violates the Dependency Inversion Principle!

```java
public class PizzaDeliveryCat {
    private Scooter scooter = new Scooter(); // tight coupling!

    public void deliver(String address) {
        scooter.driveTo(address);
    }
}


class Scooter {
    public void driveTo(String address) {
        System.out.println("Scooter zooms to " + address);
    }
}
```

High-level modules should not depend on low-level modules. Both should depend on abstractions.

EXERCISE #4: IS THIS CODE SOLID?

- How can we improve the code?



FISHY :)

EXERCISE #5: DOES THIS CODE SMELL?

```java
public class KittyCashier {
  public void printFishDetails(Fish fish) {
    System.out.println("Fish name: " + fish.getName());
    System.out.println("Fish weight: " + fish.getWeight() + "kg");
    System.out.println("Fish price: $" + fish.getPricePerKg() * fish.getWeight());
  }
}

public class Fish {
    private String name;
    private double weight;
    private double pricePerKg;
    // constructors, getters and setters omitted...
}
```

FISHY :)

EXERCISE #5: DOES THIS CODE SMELL?

- How would you refactor it?
  - KittyCashier is a little too much interested in internal details of Fish
  - Access multiple fields (**feature envy**) and performs some logic that belongs to Fish
  - Violates Single Responsibility Principle
  - Might start by moving code to a new method in Fish (e.g.: Fish.getDetails())

```java
public class KittyCashier {
  public void printFishDetails(Fish fish) {
    System.out.println(fish.getDetails());
  }
}
```

```java
public class Fish {
  public String getDetails() {
    return "Fish name: " + name + "\n" +
      "Fish weight: " + weight + "kg\n" +
      "Fish price: $" + (pricePerKg * weight);
  }
}
```

FISHY :)

EXERCISE #5: DOES THIS CODE SMELL?

- How would you refactor it?
  - getDetails still has multiple responsibilities
  - Might want to perform another extract method to extract logic related to price

```java
public class Fish {
  public String getDetails() {
    return "Fish name: " + name + "\n" +
      "Fish weight: " + weight + "kg\n" +
      "Fish price: $" + (pricePerKg * weight);
  }
}
```

```java
public class Fish {
  public String getDetails() {
    return "Fish name: " + name + "\n" +
     "Fish weight: "+ weight + "kg\n" +
     "Fish price: $"+ this.calculatePrice();
  }

  public double calculatePrice() {
    return pricePerKg * weight;
  }
}
```