PRINCIPLES OF DETAILED DESIGN

Lecture Notes

Software Design (SE324)

(Prepared by Dr. Khaldoon Al-Zoubi)

Review…

- **Class**
  - template definition
- **Object**
  - a specific instance of a class
- **Interface**
  - Class is an interface implementation
  - Agreed protocol between various implementations (classes)
- **Polymorphism**
  - The ability to take on many forms.

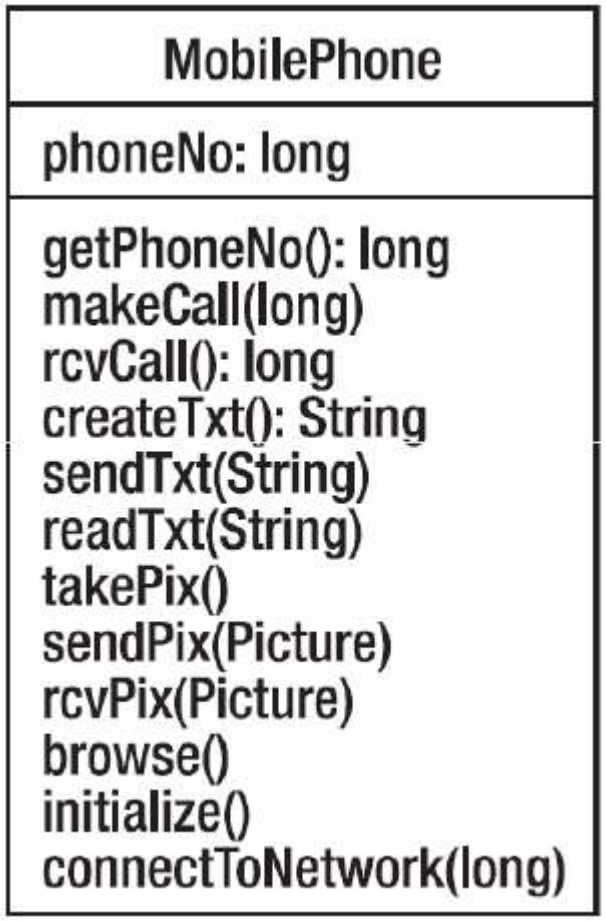
```
class A
{
	void method()
	{
		System.out.println("From Class A");
	}
}

class B extends A
{
	@Override
	void method()
	{
		System.out.println("From Class B");
	}
}
```

Design Principles for Internal Component Design

- Maintainability: Component designs that evolve gracefully over time are hard to achieve.
  - Modifiability → Minimizing the degree of complexity involved when changing the system code (classes structure and behavior) to fit current or future needs.
  - Reusability → Ability of code (classes, methods, etc.) to be used repeatedly
- OO Design principles for internal component design include (SOLID):
  - **S**ingle responsibility principle
  - **O**pen-Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface-Segregation Principle
  - **D**ependency-Inversion Principle

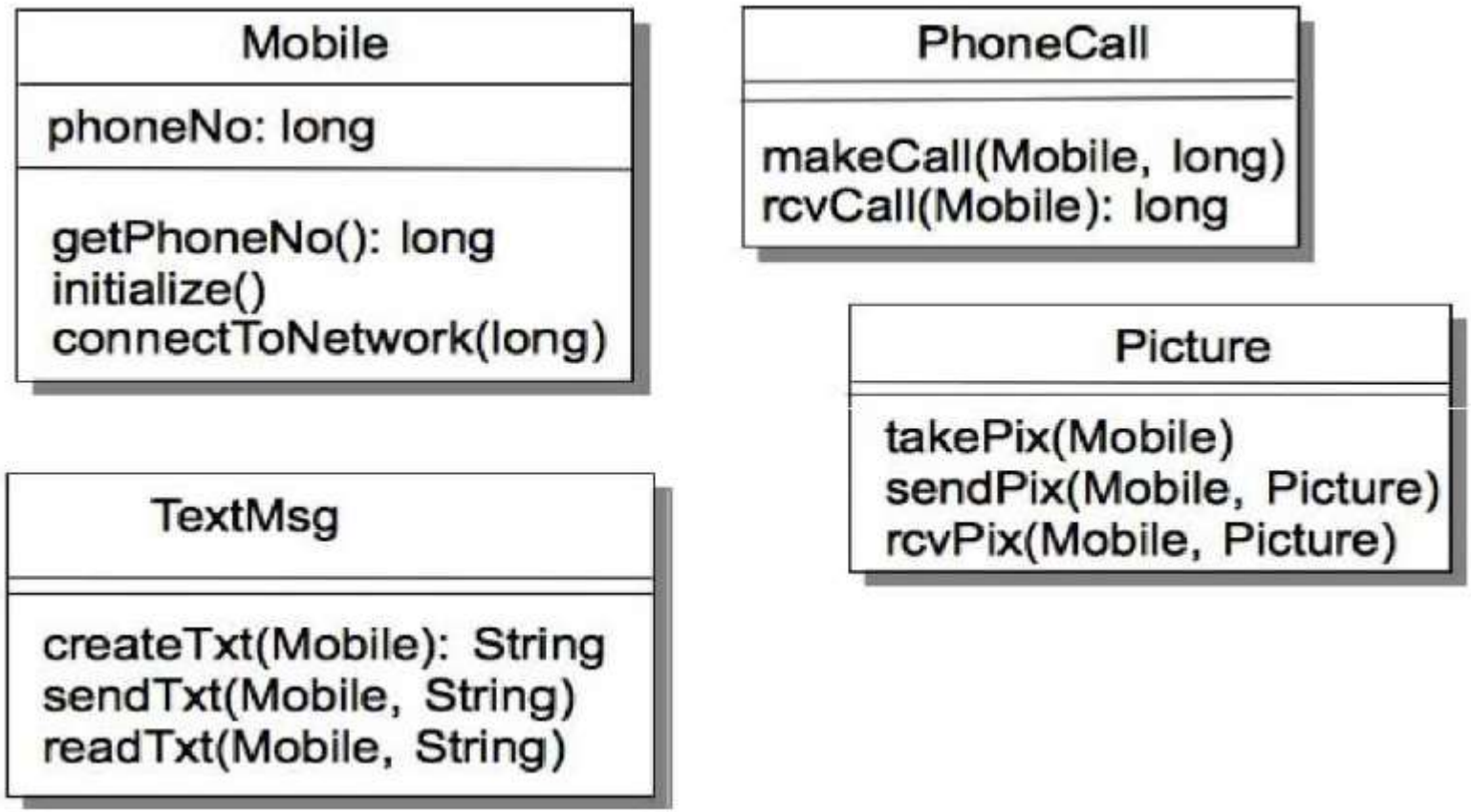
Single Responsibility Principle

- Single Responsibility means One reason to change
  - This implies that **higher cohesion** is better.
- All services should be focused on carrying out that single responsibility.
- This principle states that if we have 2 reasons to change for a class, we must split the functionality in two classes



Single Responsibility Principle

- Each Class with single responsibility



Open-Closed principle (OCP)

- What it means:
  - software designs should be open to extension but closed for modification.
  - The main idea behind the OCP is that code that works should remain untouched and that new additions should be extensions of the original work.
  - Being close to modifications does not mean that designs cannot be modified; it means that modifications should be done by adding new code, and incorporating this new code in the system in ways that does not require old code to be changed!
- Why OCP?
  - an essential principle for creating **reusable and modifiable** systems that evolve gracefully with time.

Open-Closed principle (OCP)

Consider a fictional **gaming system** that includes **several types of terrestrial characters**, ones that can roam freely over land. *It is anticipated that new characters will be added in the future.*

Note:
This is really not the code for a gaming system! The code is for illustration purpose.

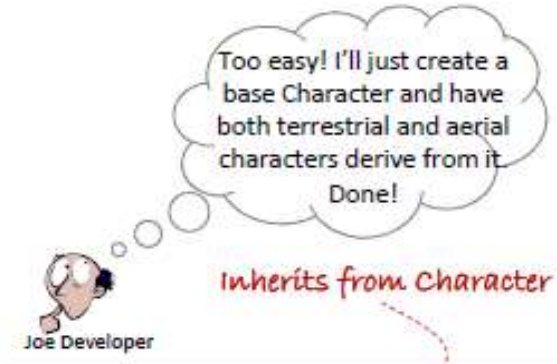
```
// The terrestrial character.
class TerrestrialCharacter {

public:
  // Draw the character on the screen.
  virtual void draw() { /*Code to draw the terrestrial character.*/ }

  // Make the character run!
  virtual void run() { /* Code to make the character run.*/
};

// The game engine responsible for managing the game.
class GameEngine {

public:
  // Add the character to the screen.
  void add(TerrestrialCharacter* pCharacter) {

    // Display the character.
    pCharacter->draw();

    // Make the character move!
    pCharacter->run();
  }
};
```

What can you tell me about the add(...) function?

What happens if we add a new requirement to support other types of characters, e.g., an AerialCharacter that can fly?

Yes, that is right, we would have to change the code inside the add(...) method. This violates the OCP! Let's see an improved version in the next slide...

Open-Closed principle (OCP)




```
class Character {

public:
  // Get the type of character.
  virtual string getType() = 0;

  // Draw the character on the screen.
  virtual void draw() = 0;
};
```

Joe Developer decided to abstract the Character concept and separate it from more specific Character types

Inherits from Character

```
class AerialCharacter : public Character {

public:
  // Get the type of character.
  virtual string getType() {

    // Return the type of character.
    return "aerial";
  }

  // Draw the character on the screen.
  virtual void draw() {

    // Code to draw the aerial character.
    cout<<"drawing aerial character!\n";
  }

  // Make the character fly!
  virtual void fly() {

    // Code to make the character fly.
    cout<<"character flying!\n";
  }
};
```

Since Terrestrial characters run and Aerial ones fly, Joe decided to delegate creation of these functions to subtypes, namely, TerrestrialCharacter and AerialCharacter

Are we done? Not really! The getType(...) function should give you an indication why we're still violating the OCP. Let's take a closer look in the next slide...

Inherits from Character

```
class TerrestrialCharacter : public Character {

public:
  // Get the type of character.
  virtual string getType() {

    // Return the type of character.
    return "terrestrial";
  }

  // Draw the character on the screen.
  virtual void draw() {

    // Code to draw the terrestrial character.
    cout<<"drawing terrestrial character!\n";
  }

  // Make the character run!
  virtual void run() {

    // Code to make the character run.
    cout<<"character running!\n";
  }
};
```

Note: Character is really an interface, so instead of "Inherits from Character" it (more precisely) realizes the Character interface.

Open-Closed principle (OCP)

Design Principle:
Encapsulate Variation

```
class GameEngine {

public:
  // Add a character to the game.
  void add( Character* pCharacter ) {

    // Draw the character on the screen.
    pCharacter->draw();

    // If aerial, make it fly, otherwise, make it run.
    if( pCharacter->getType().compare("aerial") == 0 ) {

      // Downcast the pointer to an aerial character.
      AerialCharacter* pAerial = dynamic_cast<AerialCharacter*>(pCharacter);

      // Assume a valid pointer and make the character fly!
      pAerial->fly();
    }
    else {

      // Downcast the pointer to a terrestrial character.
      TerrestrialCharacter* pTerrestrial =
                            dynamic_cast<TerrestrialCharacter*>(pCharacter);

      // Make the character run!
      pTerrestrial->run();

    } // end if statement.
  } // end add function.
};
```

This code will always vary, depending on the characters in the game!

Notice how the GameEngine client needs to know the type of Character before it can activate it. This is a side-effect of a violation of the O

Sample test driver code

```
int _tmain(int argc, _TCHAR* argv[])
{
    // create the mad rabbit character.
    TerrestrialCharacter madRabbit;
    // create the killer bee character.
    AerialCharacter killerBee;

    // create the game engine.
    GameEngine engine;

    // add characters to the game.
    engine.add(&madRabbit);
    engine.add(&killerBee);
    system("pause");

    return 0;
}
```

Sample output

```
drawing terrestrial character!
character running!
drawing aerial character!
character flying!
Press any key to continue . . .
```

It works! We're done!
Not really, we've improved the design, but are we OCP-Compliant?

The Character design still requires clients to know too much about Characters.
What would happen if we now need to support an Aquatic Character?

Let's see in the next slide how to make this design OCP-Compliant...

Open-Closed principle (OCP)

```
class Character {

public:
  // Draw the character on the screen.
  virtual void draw() = 0;

  // Make the character move.
  virtual void move() = 0;
};
```

Encapsulate the movement behavior, so that move(...) works for all characters in the game!

```
// The aerial character.
class AerialCharacter : public Character {

public:
  // Draw the character on the screen.
  virtual void draw() { /* Code to draw the aerial character. */ }

  // Make the character fly.
  virtual void move() { /* Code to make the character fly! */ }
};
```

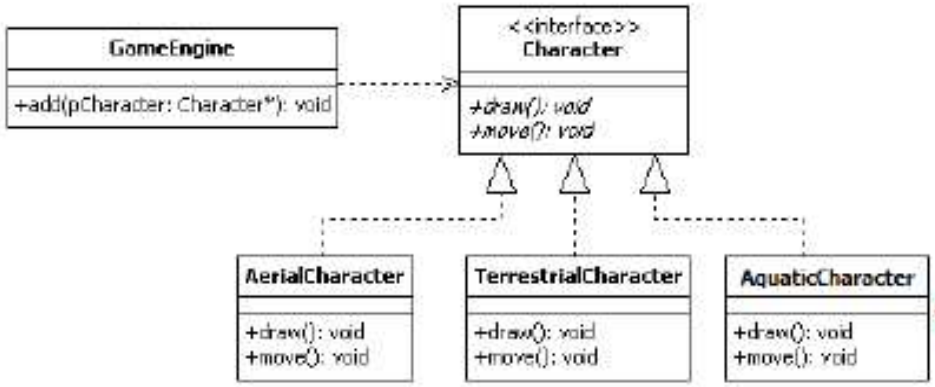
Per the interface contract, these must provide the implementation for both draw and fly services

```
// The terrestrial character.
class TerrestialCharacter : public Character {

public:
  // Draw the character on the screen.
  virtual void draw() { /* Code to draw the terrestrial character. */ }

  // Make the character run.
  virtual void move() { /* Code to make the character run! */ }
};
```

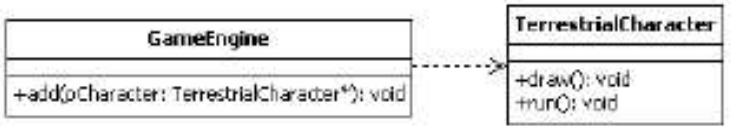
In the next slide, let's see how the code for the GameEngine class looks now based on this new design...

Open-Closed principle (OCP)

New redesign! Adheres to OCP!



Old design! Violates OCP!



New Aquatic Character added by extension and not by modifying existing working code!

```
// The game engine responsible for managing the game.
class GameEngine {

public:
  // Add the character to the screen.
  void add(Character* pCharacter) {

    // Display the character.
    pCharacter->draw();

    // Activate the character... make it move!
    pCharacter->move();

  } // end add function.
};
```

With this design, GameEngine can draw and activate current and future Characters in the game without modification!

Liskov Substitution Principle

- Subclasses must be substitutable for their base class
  - Child classes should never break the parent class' type definitions.
  - Subtypes must be substitutable for their base types.
- The above means: Make sure your generalization (inheritance) or realization does not break classes behavior

Liskov Substitution Principle Java Example

```
public class Rectangle
{
    private double width;
    private double height;

    /**
     * Constructor for objects of class Rectangle
     */
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    public double getHeight() {
        return this.height;
    }

    public double getWidth() {
        return this.width;
    }
}
```

```
public class Square extends Rectangle
{
    /**
     * Constructor for objects of class Square
     */
    public Square(double side) {
        super(side, side);
    }

    public void setSide(double side) {
        super.setWidth(side);
        super.setHeight(side);
    }

    public double getSide() {
        return super.getWidth();
    }
}
```

```
void myFunc(Rectangle r, double newWidth) {
        r.setWidth(newWidth);
}
```

Liskov Substitution Principle Java Example

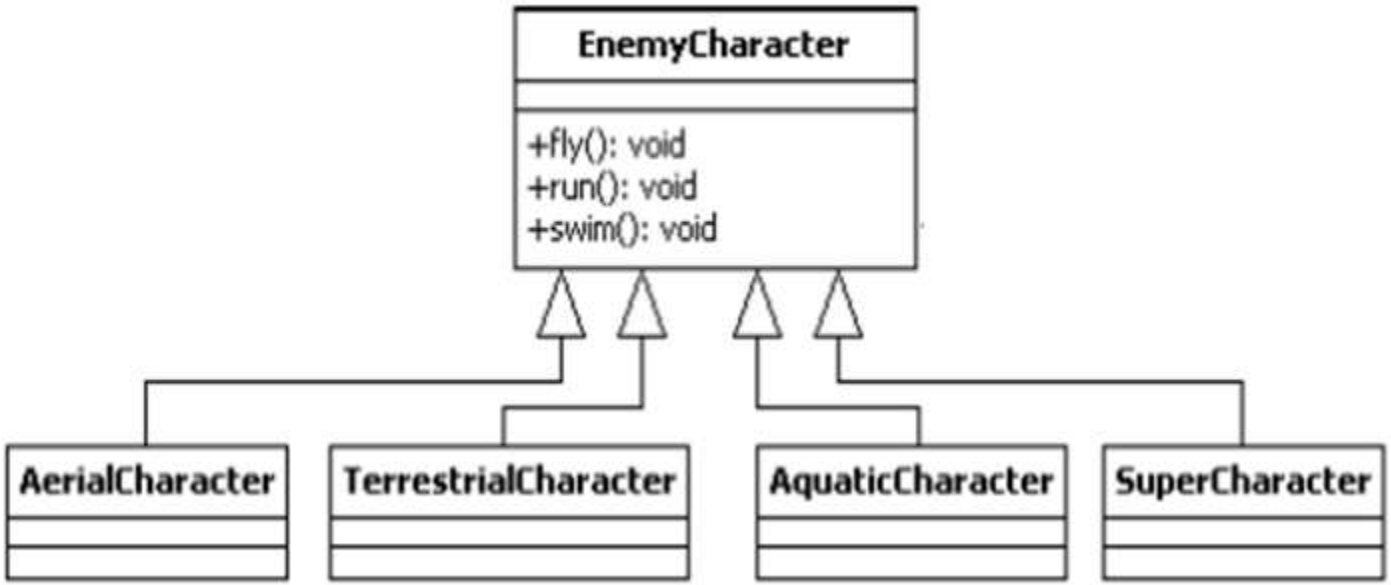
- Now we can get around this. We can override the Rectangle class' setWidth() and setHeight() methods in Square like this:

```
public void setWidth(double w) {
        super.setWidth(w);
        super.setHeight(w);
}

    public void setHeight(double h) {
        super.setWidth(h);
        super.setHeight(h);
}
```

- What is wrong with this solution?
- What other possible solutions?

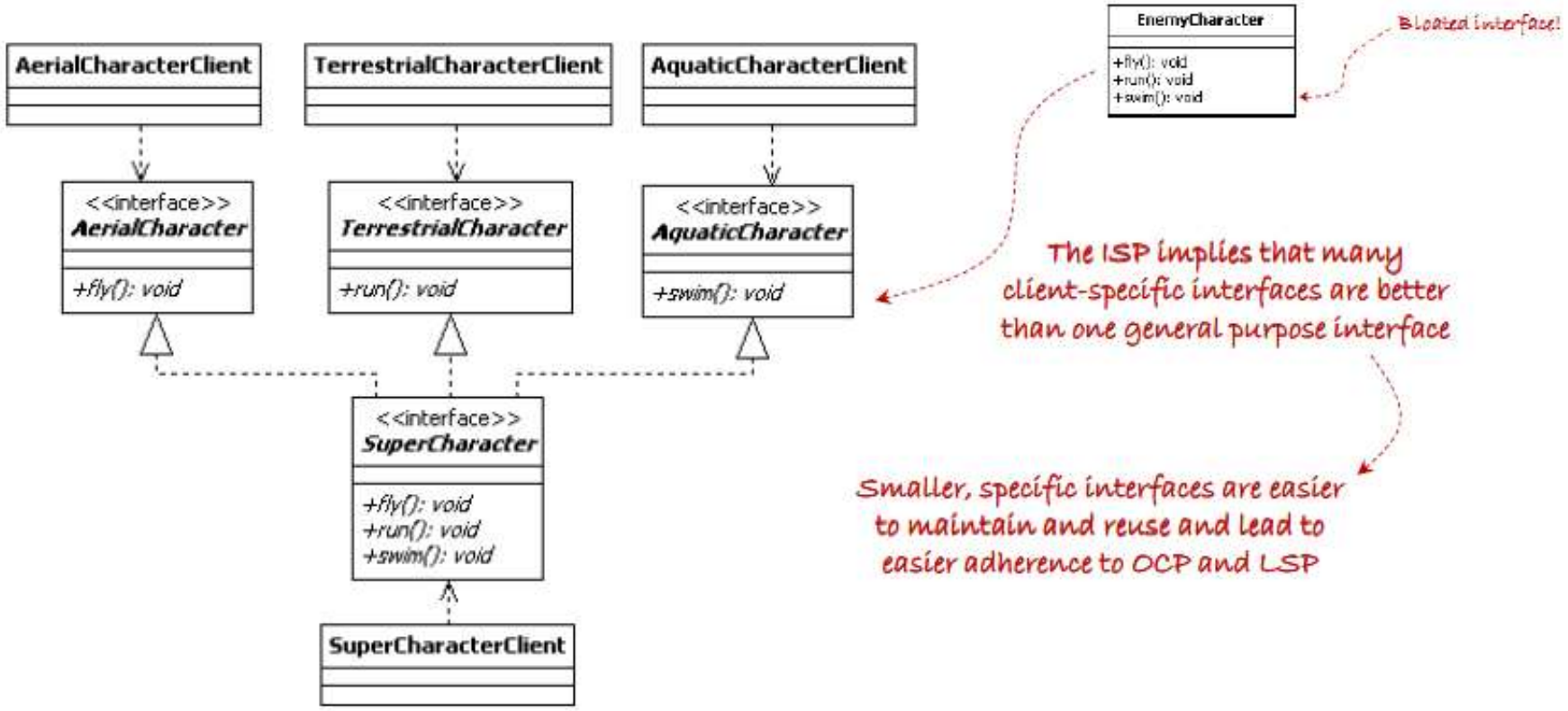
Interface segregation principle (ISP)

- Is this good design? Why?



Interface segregation principle (ISP)

- **ISP means: Clients shouldn't have to depend on interfaces they don't need**
- **Remember high *cohesion is good***



The Dependency Inversion Principle (DIP)

```
// Dependency Inversion Principle - Bad example

class Worker {

        public void work() {

                // ....working

        }

}



class Manager {

        Worker worker;



        public void setWorker(Worker w) {
                worker = w;
        }

        public void manage() {
                worker.work();
        }
}

class SuperWorker {
        public void work() {
                //.... working much more
        }
}
```

```
// Dependency Inversion Principle - Good example
interface IWorker {
        public void work();
}

class Worker implements IWorker{
        public void work() {
                // ....working
        }
}

class SuperWorker  implements IWorker{
        public void work() {
                //.... working much more
        }
}

class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
                worker = w;
        }

        public void manage() {
                worker.work();
        }
}
```

The Dependency Inversion Principle (DIP)

- High-level modules should not depend on low-level modules
- Don't depend on concrete classes depend on abstractions.
- Abstractions (or interfaces) should not depend on details (or implementations).
  - Details (implementations) should depend on abstractions (interfaces).
- Effectively, the DIP reduces coupling between different pieces of code
  - **Low coupling is good**