

# Object Oriented Design

Thursday, October 4

# Announcements

# S.O.L.I.D. Design Principles

**S**ingle Responsibility Principle (SRP)

**O**pen-Closed Principle (OCP)

**L**iskov Substitution Principle (LSP)

**I**nterface Segregation Principle (ISP)

**D**ependency Inversion Principle (DIP)

# Single Responsibility Principle

*A class should have only one responsibility.*

Responsibility = Reason for change

***A class should have only one reason to change.***

# Single Responsibility Principle



```
public class Person {  
    public void save() {...}  
    public double calculateHours() {...}  
    public double calculateSalary() {...}  
    public double calculateTax() {...}  
}
```

The person class has at least 3 reasons to change:

If the database schema changes

If the salary calculation rules change

If the tax code changes

# The solution

Extract separate responsibilities into their own classes.

**Only when the changes occur.**

```
public class Person {  
    public void save();  
    public double calculateHours() {...}  
    public double calculateSalary() {...}  
    public double  
    calculateTax(TaxCalculator c) {...}  
}
```

```
public class TaxCalculator {  
    public double calculateTax(...) { ... }  
}
```



# Open Closed Principle

*Software entities (classes) should be closed for modification, but open for extension.*

# Open Closed Principle

**Closed for modification:** the source code doesn't need to be modified when requirements change

**Open for extension:** You can easily add new features, change the behavior to meet new requirements

```
public class Shape {  
    private String shape;  
    public void draw() {  
        if (shape.equals("circle")) {  
            ... // draw a circle  
        } else if (shape.equals("square")) {  
            ... // draw a square  
        }  
    }  
}
```

*What if I want to  
add a triangle?*

```
public abstract class Shape() {  
    public abstract void draw();  
}
```

```
public class Circle extends Shape { ... }
```

```
public class Square extends Shape { ... }
```

```
public class Triangle extends Shape {...}
```

# Known uses

Chrome extensions

Any tool that allows plugins (VS Code, Atom, IDEA, Eclipse)

Operating systems drivers

# Strategic closure

Can a system be 100% closed?

No.

Closure must be **strategic**.

Places in the code that **change often** are generally good candidates.

```
public class NotificationSender {  
    public send(User user, String text) {  
        val message = new EmailMessage(user);  
        message(text);  
    }  
}
```

*What if I want to send a text message?*

```
public class NotificationSender {  
    public send(String text, EmailMessage  
        message) {  
        message.send(text);  
    }  
}
```

```
notificationSender.send(new  
    EmailMessage(bob), "Hello");
```

*Will this work?*



```
public class NotificationSender {  
    public send(String text, Message  
        message) {  
        message.send(text);  
    }  
}
```

```
public abstract class Message {  
    public Message(User user) { ... }  
    public abstract void send(String text);  
}
```

```
public class EmailMessage extends  
    Message {  
    public send(String text) { ... }  
}
```

```
public class TextMessage extends  
    Message {  
    public send(String text) { ... }  
}
```

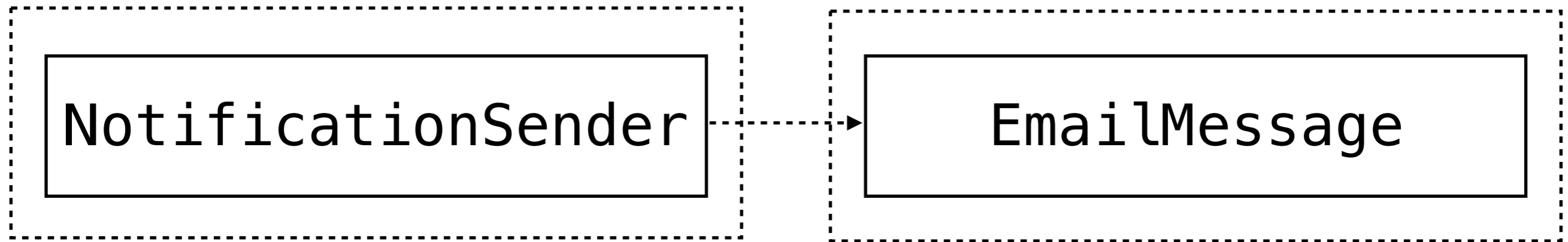
```
notificationSender.send(new  
    EmailMessage(bob), "Hello");  
notificationSender.send(new  
    TextMessage(bob), "Hello");
```

# Dependency Inversion Principle

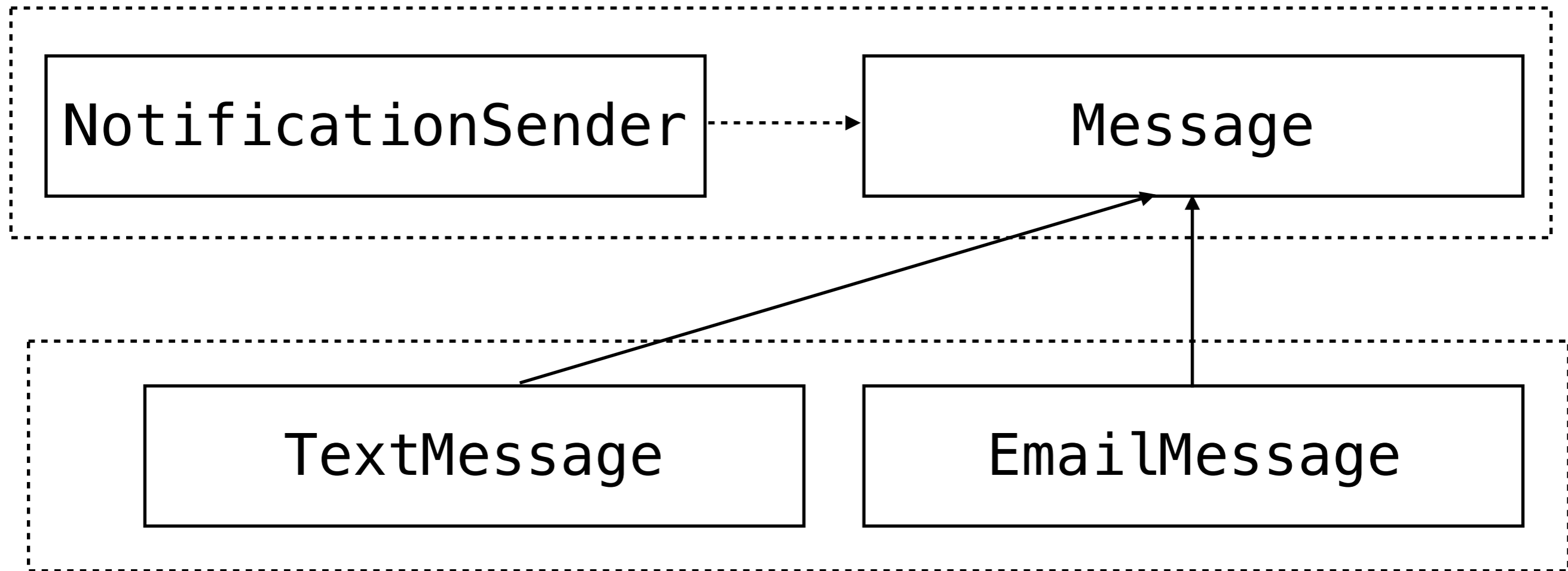
High level modules (classes) should not depend on low level modules (classes). Both should depend on **abstractions**.

Abstractions should not depend on details. Details should depend on abstractions.

# Dependency Inversion Principle



# Dependency Inversion Principle

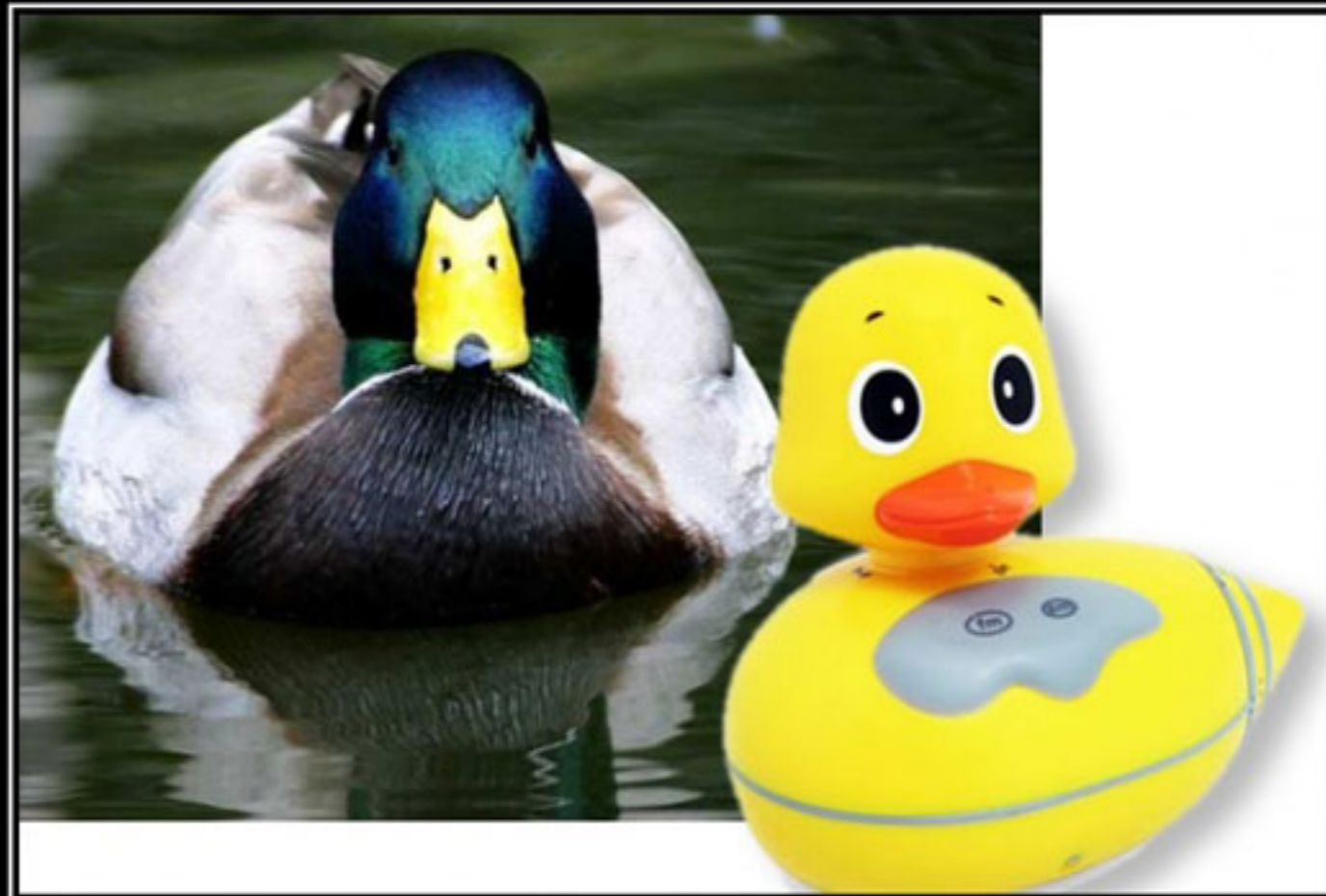


# Liskov Substitution Principle

Named after Barbara Liskov, who defined it

She was the Turing Award in 2008 for her work in the design of programming languages.

# Liskov Substitution Principle



## LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

# Liskov Substitution Principle

Functions that use references to base classes must be able to use objects of derived classes (subclasses) without knowing it.

You should always be able to substitute a derived class for its base class.



```
public abstract class Bird{  
    public abstract void fly();  
}
```

```
public class Pigeon extends Bird {  
    public void fly() {  
        // fly like a pigeon  
    }  
}
```

```
public class Duck extends Bird {  
    public void fly() {  
        // fly like a duck  
    }  
}
```

```
public class Penguin extends Bird {  
    public void fly() {  
        // fly like a penguin  
    }  
}
```

Penguins don't fly? What are our options:

1. leave the method empty
2. throw an exception
3. A penguin isn't a bird

# The contract

*A bird must fly.*

When we call `Bird.fly()`; we expect something to happen; e.g. the bird's location changed, and it's at a higher altitude

If nothing happens, I've **broken** the contract.

# The contract

Each class (abstract or not), or interface, defines a contract.

Bertrand Meyer's **design by contract** expresses this in a formal way, with pre-, post-conditions and invariants.

Every sub-class must respect the contract

# The contract

Why is breaking the contract bad?

Everywhere I use a bird, I must check, to see if's a Penguin, and act accordingly.

```
if (bird instanceof Penguin) {  
    //do penguin stuff  
}
```

This breaks OCP, among others.

# The solution

```
public abstract class FlightlessBird {  
    public abstract void waddle();  
}
```

```
public abstract class Bird extends FlightlessBird {  
    public abstract void fly();  
}
```

```
public class Pigeon extends Bird {  
    public void fly() { ... }  
}
```

```
public class Penguin extends FlightlessBird {  
    public void waddle() { ... }  
}
```

# Interface Segregation Principle

*Clients should not be forced to depend on methods that they do not use.*

This applies to abstract classes and interfaces

```
public interface CoffeeMachine {  
    public void addGroundCoffee(  
        GroundCoffee c);  
    public Drink brewFilterCoffee();  
}
```

```
public class MrCoffee  
    implements CoffeeMachine {  
    public void addGroundCoffee(  
        GroundCoffee c){ ... }  
    public Drink brewFilterCoffee(){ ... }  
}
```



```
public interface CoffeeMachine {  
    public void addGroundCoffee(  
        GroundCoffee c);  
    public Drink brewFilterCoffee();  
    public Drink brewEspresso();  
}
```

```
public class FancyEspressoMachine  
    implements CoffeeMachine {  
    public void addGroundCoffee(  
        GroundCoffee c) { ... }  
    public Drink brewFilterCoffee() { ... }  
    public Drink brewEspresso() { ... }  
}
```

# The problem

The `CoffeeMachine` interface is too broad

Implementing classes are forced to no-op at least some of the methods.

This breaks the contract, and makes things ambiguous.

# The solution

```
public interface CoffeeMachine {  
    public void addGroundCoffee(  
        GroundCoffee c);  
    public Drink brewFilterCoffee();  
    public Drink brewEspresso();  
}
```

```
public interface FilterCoffeeMachine  
    extends CoffeeMachine {  
    public Drink brewFilterCoffee();  
}
```

```
public interface EspressoCoffeeMachine  
    extends CoffeeMachine {  
    public Drink brewEspresso();  
}
```

# The solution

```
public class MrCoffee  
  implements FilterCoffeeMachine {  
  public void addGroundCoffee(  
    GroundCoffee c){ ... }  
  public Drink brewFilterCoffee(){ ... }  
}
```

```
public class FancyEspressoMachine  
  implements EspressoMachine {  
  public Drink brewEspresso(){ ... }  
}
```

# The result

Small, composable interfaces

Each interface does one thing

Complex objects will implement multiple interfaces

**S**ingle Responsibility Principle (SRP)

**O**pen-Closed Principle (OCP)

**L**iskov Substitution Principle (LSP)

**I**nterface Segregation Principle (ISP)

**D**ependency Inversion Principle (DIP)