# Принципы объектно-ориентированного дизайна

- SOLID:
  - **S**ingle responsibility
  - **O**pen-closed
  - **L**iskov substitution
  - **I**nterface segregation
  - **D**ependency inversion

# Что такое SOLID

**SOLID** - это аббревиатура пяти основных принципов дизайна классов в объектно-ориентированном проектировании.

Аббревиатура была введена Робертом Мартином в начале 2000-х.

Рекомендую почитать:

**Чистый код. Роберт Мартин**

# Основные принципы

- **S**ingle responsibility - **Принцип единственной обязанности**
- **O**pen-closed - **Принцип открытости/закрытости**
- **L**iskov substitution - **Принцип подстановки Барбары Лисков**
- **I**nterface segregation - **Принцип разделения интерфейса**
- **D**ependency inversion - **Принцип инверсии зависимостей**

# Single responsibility
## Принцип единственной обязанности

- Класс или модуль должны иметь одну и только одну причину измениться.

# Пример нарушения принципа SRP

```
class Order
{
    public void calculate(){ ... }
    public void addItem(Product product){ ... }
    public List<Product> getItems(){ ... }
    ...
    public void load(){ ... }
    public void save(){ ... }
    public void print(){ ... }
}
```

# Как исправить

```
class Order
{
    public void calculate();
    public void addItem(Product product){ ... }
    public List<Product> getItems(){ ... }
}

class OrderRepository
{
    public Order load(int orderId){ ... }
    public void save(Order order){ ... }
}

class OrderPrintManager
{
    public void print(Order order){ ... }
}
```

# Но...

Существует, например, паттерн Active Record, который нарушает принцип SRP

*Active Record может быть успешно использован в небольших проектах с простой бизнес-логикой*.

```
ActiveRecord post = Post.newRecord();
post.setData("title", "Happy Java Programming");
post.setData("body", "Java programming is fun.");
post.create();
```

# Open-closed
# Принцип открытости/закрытости

- Объекты проектирования (классы, функции, модули и т.д.) должны быть открыты для расширения, но закрыты для модификации.
- Это означает, что новое поведение должно добавляться только добавлением новых сущностей, а не изменением старых.

# Пример нарушения OCP

```
class MessageSender {

    …

    public void send(String message, MessageType type){
              if(type == MessageType.SMS)
  sendSMS(msg);
              else
                 if(type == MessageType.EMAIL)
  sendEmail(msg);
   }
}
```

# Как исправить

- Воспользуемся паттерном “Стратегия”

```
interface SendingStrategy {
    void send(String message);
}

class MessageSender {
    private SendingStrategy strategy;

    public MessageSender(SendingStrategy strategy) {
        this.strategy = strategy;
    }

    public void send(String message) {
        this.strategy.send(message);
    }

}
```

# Как исправить(продолжение)

- Конкретные стратегии отправки

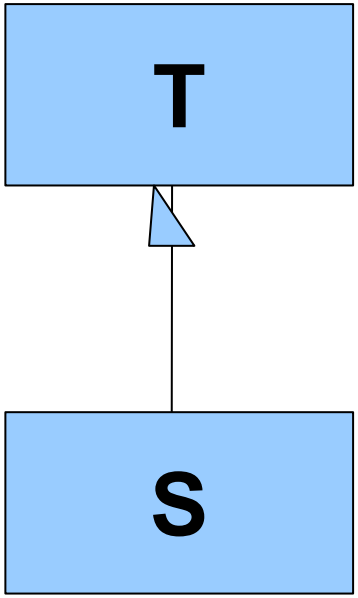
```
class EmailSendingStrategy implements SendingStrategy {

    @Override
    public void send(String message) {
        System.out.println("Sending Email: " + message);
    }

}

class SMSSendingStrategy implements SendingStrategy {

    @Override
    public void send(String message) {
        System.out.println("Sending SMS: " + message);
    }

}
```

# Liskov substitution
## Принцип подстановки Барбары Лисков

Роберт С. Мартин определил этот принцип так:

*Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом.*

# Замещение



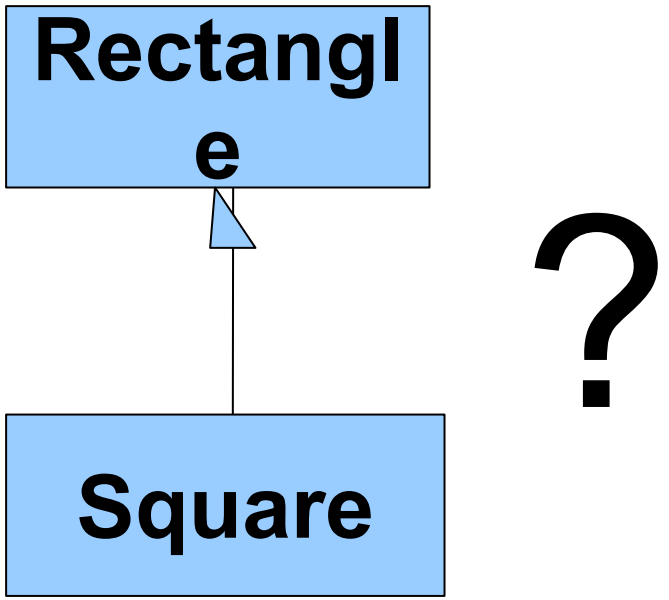

Объекты типа **T** могут быть замещены объектами типа **S** *без каких-либо изменений желательных свойств этой программы*

# Нарушение принципа LSP

- Circle-ellipse problem
- Square-rectangle problem

# Square-rectangle problem

- Является ли класс Квадрат подклассом класса Прямоугольник?



# Класс Rectangle

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

    public double getWidth() {
        return width;
    }
    public void setWidth(double width) {
        this.width = width;
    }
    public double getHeight() {
        return height;
    }
    public void setHeight(double height) {
        this.height = height;
    }
    public String toString() {
        return this.width + "x" + this.height;
    }
}
```

# Класс Square

```
class Square extends Rectangle {

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

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

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

}
```

# В чем же проблема?

```
public class LiskovViolation {


   public static void main(String[] args) {
      Rectangle rectangle = new Square();
      rectangle.setWidth(10);
      System.out.println(rectangle); // 10.0x10.0
      rectangle.setHeight(20);
      System.out.println(rectangle); // 20.0x20.0 !!!! Should be 10.0x20.0
   }


}
```

# Как исправить

- Если использовать концепцию неизменяемого объекта (immutable object), то принцип не будет нарушаться.
- Необходимо убрать возможность изменения объекта после его создания.

# Interface segregation
## Принцип разделения интерфейса

- Слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе.

# “Толстый” интерфейс

- Если среди методов интерфейса можно выделить группы методов, которые нужны определенным пользователям интерфейса, то скорее всего интерфейс “толстый”.
- Такой интерфейс нужно разбить на более мелкие, которые будут выражать потребности конкретной группы пользователей интерфейса.

# Пример нарушения ISP

```
interface Person {
    void goToWork();
    void withdrawSalary();
    void eat();
}
```

Если мы захотим сделать реализацию, в которой единственным необходимым методом будет eat() придется реализовывать и все остальные методы

# Как исправить

```
public interface Person {
    void eat();
}

public interface Worker {
    void goToWork();
    void withdrawSalary();
}
```

# Dependency inversion
## Принцип инверсии зависимостей

- Все взаимосвязи в программе должны поддерживаться с помощью абстракных классом или интерфейсов.

# Нарушение принципа DIP

```
public class Crawler {
    public void saveHtmlDocument() {
     DomBasedHtmlParser parser = new DomBasedHtmlParser();
     HtmlDocument document = parser.parseUrl("http://example.com/");
     save(document, "index.html");
    }

    public void save(HtmlDocument htmlDocument, String pageName) {
     // сохранение документа в файл
    }
}
```

Экземпляр класса парсер создается внутри метода `saveHtmlDocument()` и не использует интерфейс, что делает невозможным использования другой реализации парсера и затрудняет тестирование.

# Как исправить. Вариант 1

```
public class Crawler {
    private HtmlParser parser;

    public Crawler(HtmlParser parser) {
        this.parser = parser;
    }

    public void saveHtmlDocument() {
     HtmlDocument document = parser.parseUrl("http://example.com/");
     save(document, "index.html");
    }

    public void save(HtmlDocument htmlDocument, String pageName) {
     // сохранение документа в файл
    }
}
```

# Как исправить. Вариант 2

```
public class Crawler{

    private HtmlParser parser = ParserFactory.getHtmlParser();


    public void saveHtmlDocument() {

     HtmlDocument document = parser.parseUrl("http://example.com/");
     save(document, "index.html");
    }

    public void save(HtmlDocument htmlDocument, String pageName) {
     // сохранение документа в файл
    }
}
```