SOLID

Maurice Müller

2022-10-25

SRP

- Single Responsibility Principle

Definition

> *There should never be more than one reason for a class to change.*
>
> *— Robert C. Martin*

- Jede Klasse sollte genau eine Aufgabe erfüllen.

Generelles Prinzip für:

- Module
- Klassen
- Methoden
- Variablen

Beispiel SRP

Wieviele Aufgaben hat die folgende Klasse/Methode?

```
class UserService {

    private final DBConnection dbConnection;

    UserService(DBConnection connection) {
        this.dbConnection = connection;
    }

    public void register(String email, String password) {
        if(!email.contains("@")) {
            throw new EmailValidationException("Email address is invalid.")
        }
        dbConnection.save(new User(email, password));
        Email smtpEmail = new Email("info@example.com", email, "Successfull
        SmtpClient smtpClient = SmtpClient.get("IP", "account", "password")
        smtpClient.send(smtpEmail);
    }

}
```

- Benutzer registrieren
- E-Mail-Adressen prüfen
- Benutzer in DB speichern
- SMTP-Client erzeugen
- E-Mail verschicken

Lösung

- Refactoring
  - dt. *Umgestaltung / Neuordnung*
  - Methoden / Klassen auslagern / umschreiben

```java
class UserService {

    private final DBConnection dbConnection;

    UserService(DBConnection connection) {
        this.dbConnection = connection;
    }

    public void register(String email, String password) {
        validateEmail(email);
        persistUser(new User(email, password));
        sendEmail(email);
    }

    private void sendEmail(String email) {
        Email smtpEmail = new Email("info@example.com", email, "Successfull
        SmtpClient smtpClient = SmtpClient.get("IP", "account", "password")
        smtpClient.send(smtpEmail);
    }

    private void persistUser(User user) {
        dbConnection.save(user);
    }

    private void validateEmail(String email) {
        if(!email.contains("@")) {
            throw new EmailValidationException("Email address is invalid.")
        }
    }
}
```

```java
class UserService {

    private final DBConnection dbConnection;
    private final EmailService emailService;

    UserService(DBConnection connection, EmailService emailService) {
        this.dbConnection = connection;
        this.emailService = emailService;
    }

    public void register(String email, String password) {
        EmailValidator.validateEmail(email);
        persistUser(new User(email, password));
        emailService.sendEmail(email, "Successfully registered.");
    }

    private void persistUser(User user) {
        dbConnection.save(user);
    }
}
```

Aus der Praxis

Beispiel: src/java/principles/srp/reallife/ObjectManipulator

Vorteile von SRP

- höherer Grad an Wiederverwendbarkeit
- kleinere Klassen, Methoden, Module, …
  - bessere Wartbarkeit
- einfacher zu testen

Erinnerungshilfe

Finding your one thing

https://www.youtube.com/watch?v=2k1uOqRb0HU

OCP

- Open / Closed Principle

Definition

- Offen für Erweiterungen, geschlossen für Änderungen

> *Module sollten sowohl offen (für Erweiterungen) als auch verschlossen (für Modifikationen) sein.*
>
> — *Bertrand Meyer*

Was hängt von Klassen / Modulen ab?

- andere Klassen / Module
- Dokumentation
- Tests

⇒ Änderungen an (öffentlichen) Stellen führen zwangsläufig zu Änderungen an anderen Stellen.

OCP Beispiel

von www.joelabrahamsson.com/a-simple-example-of-the-openclosed-principle

```
public class Rectangle {
    public final Double width;
    public final Double height;

    public Rectangle(Double width, Double height) {
        this.width = width;
        this.height = height;
    }
}
```

Neues Feature: die Fläche von einer beliebigen Anzahl an Rechtecken berechnen

```
public class AreaCalculator {
    public static Double area(List<Rectangle> rectangleList) {
        return rectangleList.stream().mapToDouble(rectangle → rectangle.he
    }
}
```

Neues Feature: ein Kreis und die Flächenberechnung von Kreisen und Rechtecken

```java
public class Circle {
    public final Double radius;

    public Circle(Double radius) {
        this.radius = radius;
    }
}
```

```java
public class AreaCalculatorExt {
    public static Double area(List<Object> rectangleList) {
        return rectangleList.stream().mapToDouble(object -> {
            if (object instanceof Rectangle)
                return ((Rectangle) object).height * ((Rectangle) object).w
            else
                return ((Circle) object).radius * ((Circle) object).radius
        }).sum();
    }
}
```

Neue Feature

ein Dreieck

ein Stern

ein Kreuz

...

Lösung?

Abstraktion

```java
public interface Shape {
    Double area();
}
```

```java
public class AreaCalculator {
    public static Double area(List<Shape> shapes) {
        return shapes.stream().mapToDouble(Shape::area).sum();
    }
}
```

- Erweiterungen (neue Formen) lassen sich hinzufügen, ohne den *AreaCalculator* anpassen zu müssen

Vorteile OCP

- modularer
- Erweiterungen sind möglich, ohne bestehendes (viel) anzupassen
  - Dokumentation muss nicht geändert werden
- Schnittstellen / Ansatzpunkte sind klarer

LSP

Liskov Substitution Principle

Definition

> *If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g. correctness).*
>
> — *https://en.wikipedia.org/wiki/Liskov_substitution_principle*

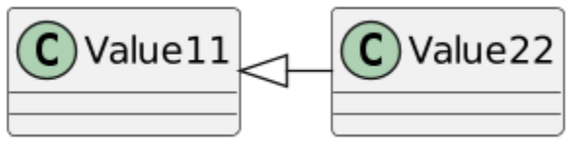
Eine abgeleitete Klasse soll an jeder beliebigen Stelle ihre Basisklasse ersetzen können, ohne, dass es zu unerwünschten Nebeneffekten kommt.

Varianz

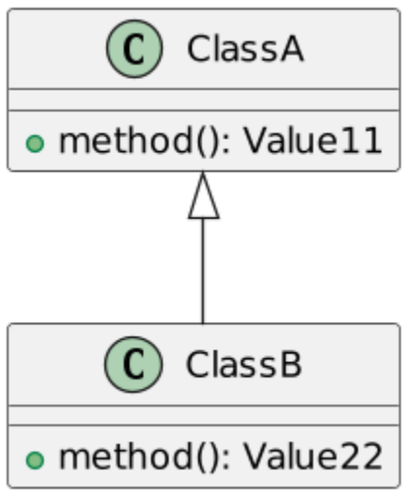
- Varianzregeln beschreiben, wann ein Objekt durch Objekte der Ober- oder Unterklassen ersetzbar ist
- drei Arten:
  - **Kovarianz**: Typhierarchie und Vererbungshierarchie haben die gleiche Richtung
  - **Kontravarianz**: Typhierarchie entgegengesetzt zur Vererbungshierarchie
  - **Invarianz**: Typhierarchie bleibt unverändert

Varianz: Beispiele

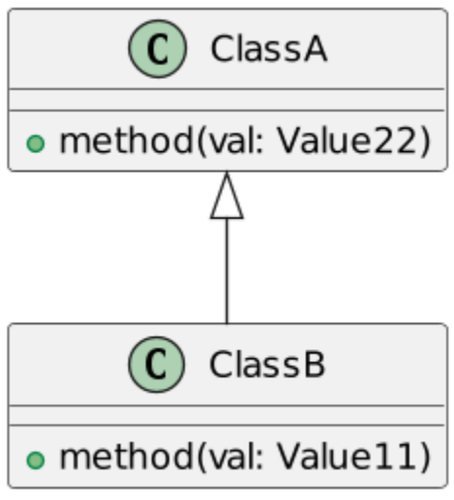
Gegeben:



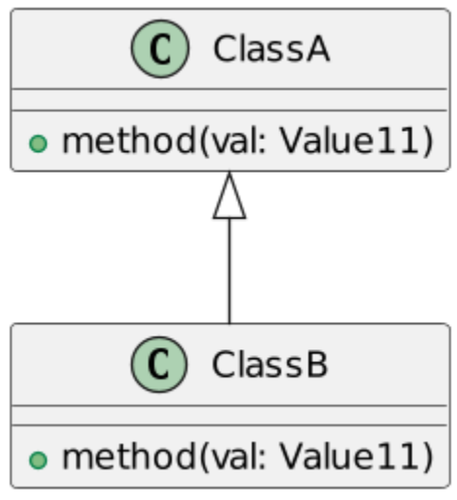
Kovarianz



Kontravarianz



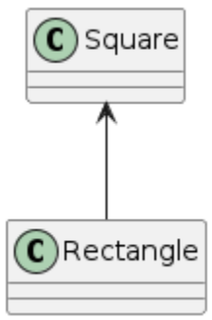
Invarianz



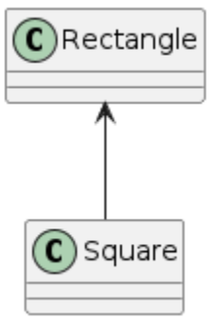
Rechteck / Quadrat

Wer leitet von wem ab nach LSP?

Rechteck → Quadrat



Quadrat → Rechteck



Option 1: Quadrat → Rechteck

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

    public Rectangle(double height, double width) {
        this.height = height;
        this.width = width;
    }

    public double getHeight() {
        return height;
    }

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

    public double getWidth() {
        return width;
    }

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

```java
@Test
public void testSetter() {
    Rectangle rectangle = new Rectangle(0.0, 0.0);
```

```java
public class Square extends Rectangle {
    public Square(double size) {
        super(size, size);
    }

    @Override
    public void setHeight(double size) {
        super.setHeight(size);
        super.setWidth(size);
    }

    @Override
    public void setWidth(double size) {
        super.setHeight(size);
        super.setWidth(size);
    }
}
```

```java
@Test
public void testSquare() {
    Rectangle rectangle = new Square(0.0);
    rectangle.setHeight(111.1);
    rectangle.setWidth(222.2);
    Assertions.assertEquals(111.1, rectangle.getHeight(), 0.0);
    Assertions.assertEquals(222.2, rectangle.getWidth(), 0.0);
}
```

Option 2: Rechteck → Quadrat

```
public class Square {
    private double width;

    public Square(double width) {
        this.width = width;
    }

    public double getWidth() {
        return width;
    }

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

```
@Test
public void testSquare() {
    Square square = new Square(0.0);
    square.setWidth(222.2);
    Assertions.assertEquals(222.2, square.getWidth(), 0.0);
}
```

```java
public class Rectangle extends Square {

    private double height;

    public Rectangle(double width, double height) {
        super(width);
        this.height = height;
    }

    public double getHeight() {
        return height;
    }

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

```java
@Test
public void testSetter() {
    Square square = new Rectangle(0.0, 0.0);
    square.setWidth(222.2);
    Assertions.assertEquals(222.2, square.getWidth(), 0.0);
}
```

Option 2: Fläche berechnen

```java
public class SquareWithArea {
    private double width;

    public SquareWithArea(double width) {
        this.width = width;
    }

    public double area() {
        return width * width;
    }

    public double getWidth() {
        return width;
    }

    // skipped setter
}
```

```java
public class RectangleWithArea extends SquareWithArea {

    private double height;

    public RectangleWithArea(double width, double height) {
        super(width);
        this.height = height;
    }

    @Override
    public double area() {
        return height * super.getWidth();
    }

    // skipped getter/setter
}
```

- man muss dran denken, bestimmte Methoden zu überschreiben
- um auf eigentliche Basisfunktionalität zugreifen zu können, benötigt man Aufrufe auf **super**
- Lösung?

Bessere Abstraktion

```java
public abstract class Shape {
    public abstract double area();
}
```

```java
public class Square extends Shape {

    private double width;

    // skipped constructor

    @Override
    public double area() {
        return width * width;
    }
}
```

```java
public class Rectangle extends Shape {

    private double width;
    private double height;

    // skipped constructor

    @Override
    public double area() {
        return width * height;
    }
}
```

Beispiel Entenrennen

```java
public static void main(String[] args) throws InterruptedException {
    List<RaceDuck> ducks = getRaceDucks();
    ducks.forEach(RaceDuck::swim);
    raceLoop(ducks);
}

private static void raceLoop(List<RaceDuck> ducks) throws InterruptedExcept
    boolean raceFinished = false;
    while(!raceFinished) {
        Thread.sleep(500);
        raceFinished = ducks.stream().allMatch(RaceDuck::finishedRace);
    }
}
```

```java
public abstract class RaceDuck {
    /**
     * Calling this method lets the duck swim immediately.
     */
    public abstract void swim();

    public abstract boolean finishedRace();
}
```

```java
public static void main(String[] args) throws InterruptedException {
    List<RaceDuck> ducks = getRaceDucks();
    ducks.forEach(RaceDuck::swim);
    raceLoop(ducks);
}

private static void raceLoop(List<RaceDuck> ducks) throws InterruptedExcept
    boolean raceFinished = false;
    while(!raceFinished) {
        Thread.sleep(500);
        raceFinished = ducks.stream().allMatch(RaceDuck::finishedRace);
    }
}
```

```java
public class RealDuck extends RaceDuck {

    private final AtomicBoolean finishedRace = new AtomicBoolean(false);
    private final String name;

    public RealDuck(String name) {
        this.name = name;
    }

    @Override
    public void swim() {
        System.out.println(name + " started swimming...");
        new CompletableFuture<Boolean>()
            .completeOnTimeout(true, ThreadLocalRandom.current().nextInt(5,
            .thenAcceptAsync(finishedRace → {
                System.out.println(name + " finished.");
                this.finishedRace.set(finishedRace);
            });
    }
```

```java
public static void main(String[] args) throws InterruptedException {
    List<RaceDuck> ducks = getRaceDucks();
    ducks.forEach(RaceDuck::swim);
    raceLoop(ducks);
}

private static void raceLoop(List<RaceDuck> ducks) throws InterruptedExcept
    boolean raceFinished = false;
    while(!raceFinished) {
        Thread.sleep(500);
        raceFinished = ducks.stream().allMatch(RaceDuck::finishedRace);
    }
}
```

```java
public class EDuck extends RaceDuck {

    private final String name;
    private boolean batteriesApplied = false;
    private final AtomicBoolean finishedRace = new AtomicBoolean(false);

    public EDuck(String name) {
        this.name = name;
    }

    @Override
    public void swim() {
        if(!batteriesApplied)
            return;
        System.out.println(name + " started swimming...");
        new CompletableFuture<Boolean>()
            .completeOnTimeout(true, ThreadLocalRandom.current().nextInt(1,
            .thenAcceptAsync(finishedRace → {
                System.out.println(name + " finished.");
```

Analyse Entenrennen:

- für *RealDuck* funktioniert alles
- bei *EDuck* hängt das Programm in einer Endlosschleife
  - *EDuck* verhält sich nicht so, wie es *RaceDuck* vorgibt

Lösung:

- *EDuck* abändern
  - automatisch Batterien einsetzen
  - Exception werfen
- bessere Abstraktion

Entenrennen Lösung

Batterien automatisch einsetzen

```
@Override
public void swim() {
    if(!batteriesApplied)
        applyBatteries();
    System.out.println(name + " started swimming...");
    new CompletableFuture<Boolean>()
        .completeOnTimeout(true, ThreadLocalRandom.current().nextInt(1, 5),
        .thenAcceptAsync(finishedRace → {
            System.out.println(name + " finished.");
            this.finishedRace.set(finishedRace);
        });
}

private void applyBatteries() {
    batteriesApplied = true;
}
```

Exception werfen

```java
/**
 * Warning: You have to apply batteries before calling this method.
 */
@Override
public void swim() {
    if(!batteriesApplied)
        throw new RuntimeException("You forgot to apply batteries.");
    System.out.println(name + " started swimming...");
    new CompletableFuture<Boolean>()
        .completeOnTimeout(true, ThreadLocalRandom.current().nextInt(1, 5),
        .thenAcceptAsync(finishedRace → {
            System.out.println(name + " finished.");
            this.finishedRace.set(finishedRace);
        });
}
```

bessere Abstraktion

```java
public abstract class RaceDuck {
    /**
     * Calling this method lets the duck swim immediately if it was prepare
     */
    public abstract void swim();

    public abstract void prepareForRace();

    public abstract boolean finishedRace();
}
```

Angepasster Aufruf der main-Methode

```
public static void main(String[] args) throws InterruptedException {
    List<RaceDuck> ducks = getRaceDucks();
    ducks.forEach(RaceDuck::prepareForRace);
    ducks.forEach(RaceDuck::swim);
    raceLoop(ducks);
}
```

Vorteile LSP

- bessere Abstraktionen
- weniger Fehler durch Polymorphie / *bessere* Polymorphie

ISP

Interface Segregation Principle

Definition

> *Many client-specific interfaces are better than one general-purpose interface.*
>
> — *Robert C. Martin*

Beispiel ISP

basierend auf http://www.oodesign.com/interface-segregation-principle.html

```java
public interface Worker {
    void work();
    void eat();
}
```

```java
public class SimpleWorker implements Worker {
    @Override
    public void work() {
        // TODO: implement method
    }

    @Override
    public void eat() {
        // TODO: implement method
    }
}
```

```java
public class SuperWorker implements Worker {
    @Override
    public void work() {
        // TODO: implement method
    }

    @Override
    public void eat() {
        // TODO: implement method
```

```java
public class Manager {
    private List<Worker> workers = new ArrayList<>();

    void manage() {
        workers.forEach(Worker::work);
    }
}
```

- neuer Mitarbeiter: ein Roboter
- ein Roboter isst nicht
  - damit der *Manager* den Roboter verwalten kann, muss Roboter das Interface *Worker* implementieren und damit die *eat()*-Methode
- Lösung?

Bessere Abstraktion

```java
public interface Workable {
    void work();
}
```

```java
public interface Eatable {
    void eat();
}
```

```java
public class HumanWorker implements Eatable, Workable {
    @Override
    public void eat() {} //skipped implemenation

    @Override
    public void work() {} //skipped implemenation
}
```

```java
public class RobotWorker implements Workable {
    @Override
    public void work() {} //skipped implementation
}
```

```java
public class Manager {
    private List<Workable> workers = new ArrayList<>();

    public void manager() {
        workers.forEach(Workable::work);
    }
}
```

Vorteile ISP

- modularer
- wartbarer
- Aufgaben sind klarer verteilt
- unterstützt SRP
- (-) unter Umständen zu viele Interfaces

DIP

Dependency Inversion Principle

Definition

> *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
>
> *Abstractions should not depend on details. Details should depend on abstractions.*

— *https://en.wikipedia.org/wiki/Dependency_inversion_principle*

Beispiel DIP

basierend auf https://de.wikipedia.org/wiki/Dependency-Inversion-Prinzip

```
public class Lamp {
    private boolean glowing = false;

    public void turnOn() {
        glowing = true;
    }

    public void turnOff() {
        glowing = false;
    }
}
```

```
public class PushSwitch {
    private boolean pushed = false;
    private final Lamp lamp;

    public PushSwitch(Lamp lamp) {
        this.lamp = lamp;
    }

    void push() {
        if(!pushed) {
            lamp.turnOn();
            pushed = true;
            return;
        }
        lamp.turnOff();
```

- Schalter hängt direkt von Lampe ab
  - ändert sich die Lampenimplementierung, muss man Schalter ebenfalls ändern
- Schalter kann nur für diese eine Lampe verwendet werden
- Lösung?

Bessere Abstraktion

```
public interface Switchable {
    void turnOn();
    void turnOff();
}
```

```
public class PushSwitch {
    private boolean pushed = false;
    private final Switchable switchable;

    public PushSwitch(Switchable switchable) {
        this.switchable = switchable;
    }

    public void push() {
        if(!pushed) {
            switchable.turnOn();
            pushed = true;
            return;
        }
        switchable.turnOff();
        pushed = false;
    }
}
```

Vorteile DIP

- modularer und damit besser wiederzuverwenden
- leichter zu erweitern
- wartbarer
- testbarer
- unterstützt OCP und SRP