

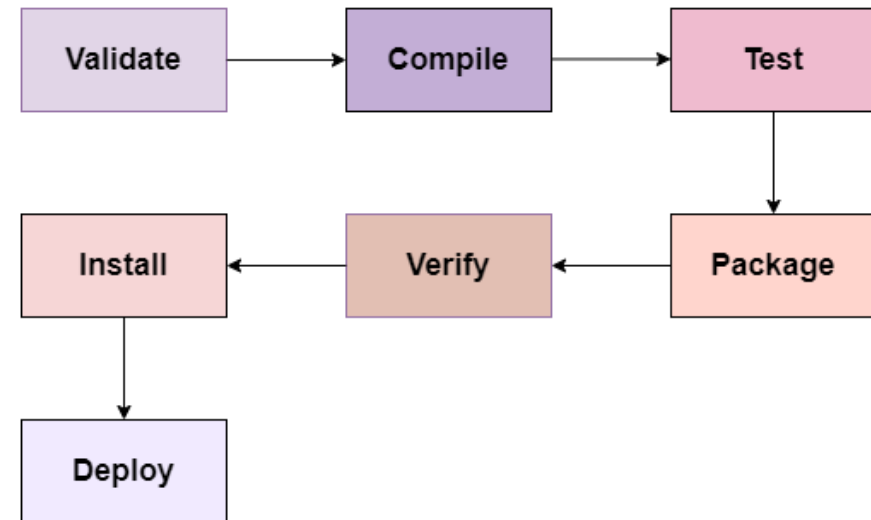
Objektovo Orientované programovanie

10. Prednáška
LS 2024/2025
Juraj Petrík

Lifecycle

- mvn clean: deletes the target directory.
- mvn compile: compiles source code.
- mvn test: runs unit tests.
- mvn package: creates JAR/WAR file.
- mvn install: installs artifact in local repository.
- mvn dependency:tree: shows dependency hierarchy.

Maven Build Lifecycle



pom.xml

- <project>
- <modelVersion>4.0.0</modelVersion>
- <groupId>com.example</groupId>
- <artifactId>my-app</artifactId>
- <version>1.0.0</version>
-
- <dependencies>
- <dependency>
- <groupId>junit</groupId>
- <artifactId>junit</artifactId>
- <version>4.12</version>
- <scope>test</scope>
- </dependency>
- </dependencies>
- </project>

Maven

- `mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.5 -DinteractiveMode=false`

Maven phases

- Vykonaju sa aj vsetky “predchadzajuce”, t.j. pri mvn compile:
 - 1.validate
 - 2.generate-sources
 - 3.process-sources
 - 4.generate-resources
 - 5.process-resources
 - 6.compile

Maven repozitare

- Local
- Central - <https://mvnrepository.com/>
- Remote

Maven zavislosti

- Direct vs transitive

```
<dependency>  
  <groupId>org.junit.jupiter</groupId>  
  <artifactId>junit-jupiter-api</artifactId>  
  <version>5.12.2</version>  
  <scope>test</scope>  
</dependency>
```

```
<dependency>  
  <groupId>com.google.code.gson</groupId>  
  <artifactId>gson</artifactId>  
  <version>2.13.0</version>  
</dependency>
```

SOLID

- **S**ingle Responsibility Principle
- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle

- Trieda ma riešiť a zodpovedať za jednu vec, nie za “všetko”

```
class Employee {  
    private String name;  
    private String position;  
  
    public void saveToDatabase(Employee employee) {  
    }  
  
    public void calculateTax(Employee employee) {  
    }  
  
    public void generateReport(Employee employee) {  
    }  
}
```

```
class Employee {  
    private String name;  
    private String position;  
}  
  
class EmployeeRepository {  
    public void save(Employee employee) {  
    }  
}  
  
class TaxCalculator {  
    public double calculateTax(Employee employee) {  
    }  
}  
  
class ReportGenerator {  
    public void generateReport(Employee employee) {  
    }  
}
```

Open-Closed Principle

- Otvorenosť pre rozšírenie, ale uzavretosť pre úpravu

```
class Shape {  
    private String type;  
  
    public double calculateArea() {  
        if (type.equals("circle")) {  
        } else if (type.equals("rectangle")) {  
        }  
    }  
}
```

```
interface Shape {  
    double calculateArea();  
}  
  
class Circle implements Shape {  
    private double radius;  
  
    @Override  
    public double calculateArea() {  
        return Math.PI * radius * radius;  
    }  
}
```

```
class Rectangle implements Shape {  
    private double width;  
    private double height;  
  
    @Override  
    public double calculateArea() {  
        return width * height;  
    }  
}  
  
class Triangle implements Shape {  
    private double base;  
    private double height;  
  
    @Override  
    public double calculateArea() {  
        return 0.5 * base * height;  
    }  
}
```

Liskov Substitution Principle

- Objekty nadtried majú byť zameniteľné s objektami podtried (nadtriedy) bez fatálnych dôsledkov na aplikáciu

```
class Bird {  
    public void fly() {  
        System.out.println("Flying");  
    }  
}  
  
class Ostrich extends Bird {  
    @Override  
    public void fly() {  
        throw new  
        UnsupportedOperationException("Ostriches can't fly!");  
    }  
}  
  
public class Main {  
    public static void makeBirdFly(Bird bird) {  
        bird.fly(); // exception!  
    }  
}
```

```
class Bird {  
}  
  
class FlyingBird extends Bird {  
    public void fly() {  
        System.out.println("Flying");  
    }  
}  
  
class Ostrich extends Bird {  
}  
  
class Sparrow extends FlyingBird {  
    @Override  
    public void fly() {  
        System.out.println("Sparrow flying");  
    }  
}  
  
public class Main {  
    public static void makeBirdFly(FlyingBird bird) {  
        bird.fly();  
    }  
}
```

Interface Segregation Principle

- Klient nemá byť nútený byť závislý od rozhraní ktoré nepoužíva

```
interface Worker {  
    void work();  
    void eat();  
    void sleep();  
}  
  
class HumanWorker implements Worker {  
    public void work() {}  
    public void eat() {}  
    public void sleep() {}  
}  
  
class RobotWorker implements Worker {  
    public void work() {}  
    public void eat() {}  
    public void sleep() {}  
}
```

```
interface Workable {  
    void work();  
}  
  
interface Eatable {  
    void eat();  
}  
  
interface Sleepable {  
    void sleep();  
}  
  
class HumanWorker implements Workable, Eatable, Sleepable {  
    public void work() {}  
    public void eat() {}  
    public void sleep() {}  
}  
  
class RobotWorker implements Workable {  
    public void work() {}  
}
```

Dependency Inversion Principle

- Vysoko úrovňové moduly nemajú byť závislé od nízko úrovňových. Implementujte abstrakciu.

```
class LightBulb {  
    public void turnOn() {}  
    public void turnOff() {}  
}  
  
class Switch {  
    private LightBulb bulb;  
  
    public Switch() {  
        this.bulb = new LightBulb();  
    }  
  
    public void operate() {  
    }  
}
```

```
interface Switchable {  
    void turnOn();  
    void turnOff();  
}  
  
class LightBulb implements Switchable {  
    public void turnOn() {}  
    public void turnOff() {}  
}  
  
class Fan implements Switchable {  
    public void turnOn() {}  
    public void turnOff() {}  
}  
  
class Switch {  
    private Switchable device;  
  
    public Switch(Switchable device) {  
        this.device = device;  
    }  
  
    public void operate() {  
    }  
}  
  
Switch lightSwitch = new Switch(new LightBulb());  
Switch fanSwitch = new Switch(new Fan());
```

Other important principles

DRY (Don't Repeat Yourself)

- Neduplikujeme kód

```
class OrderProcessor {  
    public void processDomesticOrder(Order order) {  
        validate(order);  
        calculateDomesticTax(order);  
        saveToDatabase(order);  
        sendEmail(order, "domestic");  
    }  
  
    public void processInternationalOrder(Order order)  
{  
        validate(order);  
        calculateInternationalTax(order);  
        saveToDatabase(order);  
        sendEmail(order, "international");  
    }  
}
```

```
class OrderProcessor {  
    public void processOrder(Order order, TaxCalculator taxCalculator, String type)  
    {  
        validate(order);  
        taxCalculator.calculateTax(order);  
        saveToDatabase(order);  
        sendEmail(order, type);  
    }  
}
```

KISS (Keep It Simple Stupid)

- Jednoduchší kód == čitateľnejší a lepšie udržiavateľný

```
public class ComplexTemperatureConverter {  
    public double convertTemperature(double value,  
    String fromUnit, String toUnit) {  
        if (fromUnit.equals("Celsius") &&  
toUnit.equals("Fahrenheit")) {  
            return (value * 9/5) + 32;  
        } else if (fromUnit.equals("Fahrenheit") &&  
toUnit.equals("Celsius")) {  
            return (value - 32) * 5/9;  
        } else if (fromUnit.equals("Kelvin") &&  
toUnit.equals("Celsius")) {  
            return value - 273.15;  
        }  
    }  
}
```

```
public class TemperatureConverter {  
    public double celsiusToFahrenheit(double celsius) {  
        return (celsius * 9/5) + 32;  
    }  
  
    public double fahrenheitToCelsius(double fahrenheit) {  
        return (fahrenheit - 32) * 5/9;  
    }  
}
```


YAGNI (You Aren't Gonna Need It)

- Neimplementujte niečo len preto, že možno to v budúcnosti bude potrebné

```
interface EmployeeRepository {  
    Employee findById(long id);  
    List<Employee> findAll();  
    void save(Employee employee);  
    void delete(long id);  
    // I will need these in the future, for sure!  
    List<Employee> findByDepartment(String  
department);  
    List<Employee> findBySalaryRange(double min,  
double max);  
}
```

```
interface EmployeeRepository {  
    Employee findById(long id);  
    List<Employee> findAll();  
    void save(Employee employee);  
}
```

Law of Demeter (Principle of Least Knowledge)

- Objekt komunikuje iba s najbližšími kamarátmi:
 1. Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
 2. Each unit should only talk to its friends; don't talk to strangers.
 3. Only talk to your immediate friends.

Law of Demeter (Principle of Least Knowledge)

```
class Customer {  
    private Wallet wallet;  
  
    public Wallet getWallet() {  
        return wallet;  
    }  
}  
  
class Wallet {  
    private double money;  
  
    public double getMoney() {  
        return money;  
    }  
}  
  
double money = customer.getWallet().getMoney();
```

```
class Customer {  
    private Wallet wallet;  
  
    public double getPayment(double amount) {  
        return wallet.subtractMoney(amount);  
    }  
}  
  
class Wallet {  
    private double money;  
  
    public double subtractMoney(double amount) {  
        if (money >= amount) {  
            money -= amount;  
            return amount;  
        }  
        return 0;  
    }  
}  
  
double money = customer.getPayment(100);
```

Composition Over Inheritance

- Ak je možné uprednostňujte kompozíciu (skladanie) pred dedením
- Čo keď chcem robopsa, čo vie štekať, ale nedýcha?

```
class Animal {  
    void breathe() {}  
}
```

```
class Mammal extends Animal {  
    void nurse() {}  
}
```

```
class Dog extends Mammal {  
    void bark() {}  
}
```

```
class Cat extends Mammal {  
    void meow() {}  
}
```

```
interface Breathable {  
    void breathe();  
}
```

```
interface Nursable {  
    void nurse();  
}
```

```
interface Barkable {  
    void bark();  
}
```

```
class Animal implements Breathable {  
    public void breathe() {}  
}
```

```
class Dog {  
    private Breathable breathable;  
    private Barkable barkable;  
  
    public Dog(Breathable b, Barkable bark) {  
        this.breathable = b;  
        this.barkable = bark;  
    }  
  
    public void bark() {  
        barkable.bark();  
    }  
}
```

```
class RobotDog {  
    private Barkable barkable;  
  
    public RobotDog(Barkable bark) {  
        this.barkable = bark;  
    }  
  
    public void bark() {  
        barkable.bark();  
    }  
}
```

Principle of Least Astonishment

- Snažíme sa čo najmenej (negatívne) prekvapiť používateľov (klientov)

```
class DateUtils {  
    public static Date addMonths(Date date, int months) {  
        Calendar cal = Calendar.getInstance();  
        cal.setTime(date);  
        cal.add(Calendar.MONTH, months);  
        return cal.getTime();  
    }  
}
```

```
class DateUtils {  
    public static Date addMonths(Date date, int months) {  
        if (date == null) {  
            throw new IllegalArgumentException("Date cannot be null");  
        }  
  
        Calendar cal = Calendar.getInstance();  
        cal.setTime(date);  
        cal.add(Calendar.MONTH, months);  
  
        if (cal.get(Calendar.DAY_OF_MONTH) != date.getDay()) {  
            cal.set(Calendar.DAY_OF_MONTH,  
                Math.min(cal.getActualMaximum(Calendar.DAY_OF_MONTH),  
                    date.getDay()));  
        }  
  
        return cal.getTime();  
    }  
}
```

Quiz time