NÁVRHOVÉ VZORY

Přednáška 5 – Čistý kód

ČISTÝ KÓD

- KISS
- DRY
- SOLID
- YAGNI

- „Clean Code“ označuje psaní kódu, který je snadno pochopitelný, udržovatelný a upravitelný.
- Zaměřuje se na čitelnost, jednoduchost a přehlednost, což usnadňuje spolupráci vývojářů nad společnou kódovou základnou.
- Clean Code se řídí principy a postupy, jako jsou smysluplné názvy proměnných, konzistentní formátování, správné odsazování a modulární návrh.
- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Prentice Hall.

SOLID

- Každý vývojář by měl znát těchto pět návrhových principů, které mají za cíl usměrňovat návrh a vývoj softwaru tak, aby vznikal udržitelnější, flexibilnější a lépe škálovatelný kód.
- Každý princip se zaměřuje na jiný aspekt návrhu softwaru, ale dohromady směřují k tvorbě softwarových systémů, které jsou časem snáze pochopitelné, upravitelné a rozšiřitelné.

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

- „Každá třída by měla mít pouze jeden důvod ke změně.“
- To znamená, že třída by neměla nést více odpovědností najednou a jedna odpovědnost by neměla být rozptýlena mezi více tříd nebo smíchaná s jinými odpovědnostmi.
- Je to podobné jako ve firmě – je efektivnější, když má každý svou vlastní roli, než aby šéf, zaměstnanec, řidič a kuchař byla jedna a tatáž osoba.

SRP PŘÍKLAD

```java
// Třída pro generování reportu
class ReportGenerator {
    public String generateReport() {
        return "Report content";
    }
}

// Třída pro ukládání reportu
class ReportSaver {
    public void saveToFile(String content) {
        System.out.println("Saving report: " + content);
    }
}

// Třída pro tisk reportu
class ReportPrinter {
    public void printReport(String content) {
        System.out.println("Printing report: " + content);
    }
}

// Použití
public class Main {
    public static void main(String[] args) {
        ReportGenerator generator = new ReportGenerator();
        ReportSaver saver = new ReportSaver();
        ReportPrinter printer = new ReportPrinter();

        String report = generator.generateReport();
        saver.saveToFile(report);
        printer.printReport(report);
    }
}
```

```java
class Report {
    public String generateReport() {
        return "Report content";
    }

    public void saveToFile(String content) {
        // logika pro uložení do souboru
        System.out.println("Saving report: " + content);
    }

    public void printReport(String content) {
        // logika pro tisk
        System.out.println("Printing report: " + content);
    }
}
```

OPEN/CLOSED PRINCIPLE, OCP

- Princip otevřenosti/uzavřenosti (Open/Closed Principle, OCP): „Softwarové entity by měly být otevřené pro rozšíření, ale uzavřené pro změny.“
- Tento princip říká, že třída by měla být snadno rozšiřitelná, aniž by bylo nutné měnit její základní implementaci.
- Musím snad kopat a měnit základy celého domu, abych si k němu přidal balkon?

```java
// Abstrakce
interface Shape {
    double calculateArea();
}

// Konkrétní implementace
class Circle implements Shape {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

class Rectangle implements Shape {
    private double width, height;
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public double calculateArea() {
        return width * height;
    }
}

// Kalkulátor, který neporušuje OCP
class AreaCalculator {
    public double calculateArea(Shape shape) {
        return shape.calculateArea();
    }
}

// Použití
public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle(5);
        Shape rectangle = new Rectangle(4, 6);

        AreaCalculator calculator = new AreaCalculator();
        System.out.println("Circle area: " + calculator.calculateArea(circle)
        System.out.println("Rectangle area: " + calculator.calculateArea(rect
    }
}
```

OCP - PŘÍKLAD

```java
class AreaCalculator {
    public double calculateArea(Object shape) {
        if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            return Math.PI * c.getRadius() * c.getRadius();
        } else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            return r.getWidth() * r.getHeight();
        }
        return 0;
    }
}
```

- Problém: Pokud přidáme nový tvar (např. trojúhelník), musíme upravit metodu calculateArea, čímž porušujeme OCP.
- Výhody:
- Snadné rozšíření – přidání nového tvaru nevyžaduje úpravu existujícího kódu
- Lepší udržovatelnost – méně rizika při změnách
- Čistší architektura – každá třída má jasnou odpovědnost

LISKOV SUBSTITUTION PRINCIPLE, LSP

- Princip substituce podle Liskovové (Liskov Substitution Principle, LSP): „Podtypy by měly být zaměnitelné za své základní typy, aniž by se narušila správnost programu.“
- Pokud program nebo modul používá základní třídu, pak by odvozená třída měla být schopna tuto třídu rozšířit, aniž by měnila její původní implementaci.
- Pokud to vypadá jako kachna, kváká jako kachna, ale potřebuje baterky, pravděpodobně máte špatnou abstrakci.

LSP - PŘÍKLAD

```java
class Rectangle {
    protected int width;
    protected int height;

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

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

    public int getArea() {
        return width * height;
    }
}

class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width; // nutí výšku být stejná jako šířka
    }

    @Override
    public void setHeight(int height) {
        this.height = height;
        this.width = height; // nutí šířku být stejná jako výška
    }
}
```

- Problém: Pokud použijeme Square tam, kde očekáváme Rectangle, může dojít k nečekanému chování. Například:

```java
Rectangle r = new Square();
r.setWidth(5);
r.setHeight(10);
System.out.println(r.getArea()); // Očekáváme 50, ale dostaneme 100
```

LSP - PŘÍKLAD

```
interface Shape {
    int getArea();
}

class Rectangle implements Shape {
    private int width;
    private int height;

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

    public int getArea() {
        return width * height;
    }
}

class Square implements Shape {
    private int side;

    public Square(int side) {
        this.side = side;
    }

    public int getArea() {
        return side * side;
    }
}
```

INTERFACE SEGREGATION PRINCIPLE, ISP

- Princip segregace rozhraní (Interface Segregation Principle, ISP): „Klienti by neměli být nuceni záviset na rozhraních, která nepoužívají.“

- Žádný klient by neměl být nucen implementovat metody, které nepotřebuje, a smlouvy (rozhraní) by měly být rozděleny na menší a užší.
- Pták umí létat, pták je zvíře, takže zvířata umí létat.
- Znamená to tedy, že všechna zvířata umí létat?

ISP - PŘÍKLAD

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

class Developer implements Worker {
    public void work() {
        System.out.println("Developer is coding.");
    }

    public void eat() {
        System.out.println("Developer is eating.");
    }
}

class Robot implements Worker {
    public void work() {
        System.out.println("Robot is working.");
    }

    public void eat() {
        throw new UnsupportedOperationException("Robot does not eat.");
    }
}
```

- Problém: Robot je nucen implementovat metodu eat(), kterou nepotřebuje. To porušuje ISP.

ISP - PŘÍKLAD

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

interface Eatable {
    void eat();
}

class Developer implements Workable, Eatable {
    public void work() {
        System.out.println("Developer is coding.");
    }

    public void eat() {
        System.out.println("Developer is eating.");
    }
}

class Robot implements Workable {
    public void work() {
        System.out.println("Robot is working.");
    }
}
```

- Výhoda:
- Každá třída implementuje jen to, co potřebuje
- Rozhraní jsou specifická a přehledná
- Kód je čistší, flexibilnější a lépe testovateln

DEPENDENCY INVERSION PRINCIPLE (DIP):

- Princip inverze závislostí (Dependency Inversion Principle – DIP):
  „Moduly na vysoké úrovni by neměly záviset na modulech na nízké úrovni; oba typy by měly záviset na abstrakcích.“
- Tento princip říká, že mezi komponentami softwaru by neměla být těsná vazba.
- Aby se tomu předešlo, měly by komponenty záviset na abstrakci (např. rozhraní nebo obecné definici chování), nikoli na konkrétní implementaci.
- Příklad s autem:
  Při návrhu auta se nepočítá s konkrétními příslušenstvími (např. typem rádia nebo konkrétní značkou pneumatik).
- Místo toho se auto navrhuje tak, aby bylo kompatibilní s různými typy těchto doplňků – tedy závisí na abstrakci, ne na konkrétním provedení.
- To umožňuje flexibilitu a snadnou výměnu komponent bez zásahu do samotného návrhu auta.

DIP - PŘÍKLAD

```
class LightBulb {
    public void turnOn() {
        System.out.println("LightBulb is turned on");
    }

    public void turnOff() {
        System.out.println("LightBulb is turned off");
    }
}

class Switch {
    private LightBulb bulb;

    public Switch(LightBulb bulb) {
        this.bulb = bulb;
    }

    public void operate(boolean on) {
        if (on) bulb.turnOn();
        else bulb.turnOff();
    }
}
```

- Problém: Třída Switch přímo závisí na třídě LightBulb.
- Pokud byste chtěli přepnout na jiné zařízení (například ventilátor), museli byste upravit třídu Switch, čímž by došlo k porušení principu otevřenosti/uzavřenosti (Open/Closed Principle).

```java
// Abstraction
interface Switchable {
    void turnOn();
    void turnOff();
}

// Low-level module
class LightBulb implements Switchable {
    public void turnOn() {
        System.out.println("LightBulb is turned on");
    }

    public void turnOff() {
        System.out.println("LightBulb is turned off");
    }
}

// High-level module
class Switch {
    private Switchable device;

    public Switch(Switchable device) {
        this.device = device;
    }

    public void operate(boolean on) {
        if (on) device.turnOn();
        else device.turnOff();
    }
}

// Main
public class Main {
    public static void main(String[] args) {
        Switchable bulb = new LightBulb();
        Switch lightSwitch = new Switch(bulb);
        lightSwitch.operate(true);  // Output: LightBulb is turned on
        lightSwitch.operate(false); // Output: LightBulb is turned off
    }
}
```

DIP - ŘEŠENÍ

- Výhody principu závislosti na abstrakcích (DIP)
- Volné propojení: Třída Switch nezávisí na konkrétním zařízení, které ovládá.
- Flexibilita: LightBulb lze snadno nahradit jiným zařízením, které implementuje rozhraní Switchable.
- Testovatelnost: Rozhraní Switchable lze snadno nahradit falešnou implementací (mockem) při jednotkovém testování.
- Škálovatelnost: Nová zařízení lze přidávat bez nutnosti měnit logiku ve vyšších vrstvách aplikace.

```java
public interface Engine {
    void start();
}
```

```java
public class DieselEngine implements Engine {
    @Override
    public void start() {
        System.out.println("Diesel engine starting ... ");
    }
}
```

```java
public class Car {
    private Engine engine;

    public Car(Engine engine) {
        this.engine = engine;
    }

    public void startCar() {
        engine.start();
        System.out.println("Car is running.");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Engine diesel = new DieselEngine();
        Car myCar = new Car(diesel);
        myCar.startCar();
    }
}
```

DIP - PŘÍKLAD

- Car neví nic o konkrétním typu motoru – závisí jen na rozhraní Engine.
- Můžeme snadno přidat další typ motoru (např. ElectricEngine) bez změny třídy Car.
- To je přesně to, co DIP doporučuje: záviset na abstrakcích, ne na konkrétních třídách.

YAGNI (YOU AIN'T GONNA NEED IT – NEBUDEŠ TO POTŘEBOVAT)

- YAGNI doporučuje neimplementovat funkce, dokud nejsou skutečně potřeba, aby se předešlo zbytečné složitosti a zachoval se důraz na aktuální požadavky.
- Příklad:
- „Chystám se do Velké Británie, ale balím si kufr pro Španělsko – pro případ, že bych tam někdy jel. Potřebuji ho teď?“
- → Ne. A právě to je pointa YAGNI.
- Je to jako připravovat se na scénář, který možná nikdy nenastane, místo toho, abys řešil to, co je před tebou. Chceš si to přeložit i do programátorského kontextu, nebo naopak do každodenního života?

KISS (KEEP IT SIMPLE, STUPID) – „DRŽ TO JEDNODUŠE, HLUPÁKU

- Princip KISS podporuje jednoduchost v návrhu a vývoji.
- Upřednostňuje přímočará řešení před složitými, aby se zlepšila srozumitelnost a udržovatelnost kódu.
- Přirovnání:
- „Je jednodušší letět do Velké Británie letadlem než jet autem, že?“
- → Stejně tak je lepší napsat jednoduchý kód, než se ztratit ve zbytečně složitém řešení.

KISS PŘÍKLAD

```
class Calculator {
    public int calculate(String operation, int a, int b) {
        if ("add".equals(operation)) {
            return a + b;
        } else if ("subtract".equals(operation)) {
            return a - b;
        } else {
            throw new IllegalArgumentException("Unsupported operation");
        }
    }
}
```

- Snadno pochopitelné
- Snadno testovatelné
- Dělá přesně to, co má – bez zbytečných podmínek

```
class SimpleAdder {
    public int add(int a, int b) {
        return a + b;
    }
}
```

DRY (DON'T REPEAT YOURSELF – „NEOPAKUJ SE“

- Princip DRY podporuje znovupoužitelnost kódu tím, že se vyhýbá duplicitě kódu nebo logiky. Výsledkem je čistší a lépe udržovatelný kód.
- Přirovnání:
- „Máme si kupovat nové kufry na každou cestu?“
- → Ne, samozřejmě ne.
- Stejně tak bychom neměli psát stejný kód znovu a znovu – stačí ho napsat jednou a používat opakovaně.

DRY - PŘÍKLAD

```java
class InvoicePrinter {
    public void printHeader() {
        System.out.println("=== Invoice ===");
    }

    public void printInvoiceA() {
        System.out.println("=== Invoice ===");
        System.out.println("Customer: Alice");
        System.out.println("Amount: $100");
    }

    public void printInvoiceB() {
        System.out.println("=== Invoice ===");
        System.out.println("Customer: Bob");
        System.out.println("Amount: $200");
    }
}
```

```java
class InvoicePrinter {
    private void printHeader() {
        System.out.println("=== Invoice ===");
    }

    public void printInvoice(String customer, int amount) {
        printHeader();
        System.out.println("Customer: " + customer);
        System.out.println("Amount: $" + amount);
    }
}
```

- Kód je čistší a přehlednější
- Snadno se udržuje – změna hlavičky se provede jen na jednom místě
- Znovupoužitelnost bez zbytečné duplicity

BEHAVIOURAL PATTERNS

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

OBSERVER

- Observer je behaviorální návrhový vzor, který Vám umožňuje definovat mechanismus přihlášení k odběru, aby bylo možné upozornit více objektů na jakékoliv události, ke kterým došlo u objektu, který sledují

K ČEMU VYUŽÍT

- Představte si, že máte dva typy objektů: Zákazníka a Obchod. Zákazník má velký zájem o konkrétní značku produktu (například nový model iPhonu), který by se měl brzy objevit v obchodě.
- Zákazník by mohl chodit do obchodu každý den a kontrolovat dostupnost produktu. Ale dokud je produkt stále na cestě, většina těchto návštěv by byla zcela zbytečná.
- Na druhou stranu by obchod mohl posílat spoustu e-mailů (což by mohlo být považováno za spam) všem zákazníkům pokaždé, když je nový produkt dostupný.
- Tím by někteří zákazníci byli ušetřeni nekonečných cest do obchodu.
- Zároveň by to však rozčílilo ostatní zákazníky, kteří o nové produkty nemají zájem.
- Vypadá to, že máme konflikt. Buď zákazník ztrácí čas kontrolou dostupnosti produktu, nebo obchod plýtvá prostředky oznámením nesprávným zákazníkům.

ŘEŠENÍ

- Objekt, který má nějaký zajímavý stav, se často nazývá subject (subjekt), ale protože bude také informovat ostatní objekty o změnách svého stavu, budeme jej nazývat publisher (vydavatel).
- Všechny ostatní objekty, které chtějí sledovat změny stavu publisheru, se nazývají subscribers (odběratelé).
- Vzor Observer navrhuje, abyste přidali do třídy publisher mechanismus přihlášení k odběru, aby se jednotlivé objekty mohly přihlásit k odběru nebo odhlásit od toku událostí přicházejících od publisheru.
- Ve skutečnosti tento mechanismus sestává z:
  - Pole (array) pro uložení seznamu referencí na objekty subscriberů.
  - Několika veřejných metod, které umožňují přidávat subscribery do seznamu a odebírat je ze seznamu

- Teď, kdykoli se u publisheru stane důležitá událost, projde si seznam svých subscriberů a zavolá na jejich objektech konkrétní notifikační metodu.
- V reálných aplikacích může existovat desítky různých tříd subscriberů, které mají zájem sledovat události téže třídy publisheru.
- Nechtěli byste, aby byl publisher svázán se všemi těmito třídami.
- Navíc o některých z nich nemusíte ani vědět předem, pokud má být třída publisheru používána jinými lidmi.
- Proto je zásadní, aby všichni subscriberi implementovali stejný interface a aby publisher s nimi komunikoval pouze přes tento interface.
- Tento interface by měl deklarovat notifikační metodu spolu se sadem parametrů, které může publisher použít k předání nějakých kontextových dat spolu s notifikací.

- Pokud má vaše aplikace několik různých typů publisherů a chcete, aby byli vaši subscriberi kompatibilní se všemi, můžete zajít ještě dál a donutit všechny publishery, aby dodržovaly stejný interface.
- Tento interface by potřeboval pouze popsat několik metod pro přihlášení k odběru.
- Interface by umožnil subscriberům sledovat stav publisherů, aniž by byli svázáni s jejich konkrétními třídami.

VYUŽITÍ

- Použijte vzor Observer, pokud změny stavu jednoho objektu mohou vyžadovat změnu jiných objektů a skutečná množina těchto objektů není předem známa nebo se dynamicky mění.
- S tímto problémem se často setkáte při práci s třídami grafického uživatelského rozhraní (GUI).
- Například jste vytvořili vlastní třídy tlačítek a chcete umožnit klientům připojit vlastní kód k tlačítkům, aby se spustil vždy, když uživatel stiskne tlačítko.
- Vzor Observer umožňuje jakémukoli objektu, který implementuje subscriber interface, přihlásit se k notifikacím událostí od publisher objektů.
- Mechanismus přihlášení k odběru můžete přidat i do svých tlačítek, čímž klientům umožníte připojit vlastní kód prostřednictvím vlastních tříd subscriberů.
- Použijte tento vzor, pokud některé objekty ve vaší aplikaci musí sledovat jiné objekty, ale pouze po omezenou dobu nebo ve specifických případech.
- Seznam subscriberů je **dynamický**, takže se mohou k seznamu připojit nebo z něj odejít kdykoli podle potřeby.

JAK IMPLEMENTOVAT

- Projděte svou obchodní logiku a pokuste se ji rozdělit do dvou částí:
  - Jádrová funkčnost, nezávislá na ostatním kódu, bude fungovat jako publisher.
  - Zbytek se promění v sadu tříd subscriberů.
- Deklarujte subscriber interface. Minimálně by měl deklarovat jednu metodu **update**.
- Deklarujte publisher interface a popište dvojici metod pro přidání subscriberu do seznamu a jeho odstranění ze seznamu. Pamatujte, že publishery musí pracovat se subscribery pouze přes subscriber interface.
- Rozhodněte, kde umístit skutečný seznam subscriberů a implementaci metod pro přihlášení k odběru. Obvykle tento kód vypadá stejně pro všechny typy publisherů, takže nejlogičtější místo je abstraktní třída, přímo odvozená od publisher interface. Konkrétní publishery pak tuto třídu rozšiřují a dědí chování přihlášení k odběru.
- Pokud však aplikujete vzor na existující hierarchii tříd, zvažte přístup založený na kompozici: dejte logiku přihlášení do samostatného objektu a nechte všechny skutečné publishery jej používat.
- Vytvořte konkrétní třídy publisherů. Pokaždé, když se u publisheru stane něco důležitého, musí upozornit všechny své subscribery.
- Implementujte metody notifikace update v konkrétních třídách subscriberů. Většina subscriberů bude potřebovat nějaká kontextová data o události, která lze předat jako argument notifikační metody.
- Existuje však i jiná možnost: po obdržení notifikace si může subscriber vyžádat data přímo z publisheru. V takovém případě musí publisher předat sám sebe metodou update. Méně flexibilní možností je trvale propojit publishera se subscriberem přes konstruktor.
- Klient musí vytvořit všechny potřebné subscribery a registrovat je u příslušných publisherů.

POROVNÁNÍ

Výhoda

- Princip otevřenosti/uzavřenosti (Open/Closed Principle) – můžete zavádět nové třídy subscriberů, aniž byste museli měnit kód publisheru (a naopak, pokud existuje publisher interface).
- Můžete navazovat vztahy mezi objekty za běhu programu.

Nevýhoda

- Subscribery jsou upozorněny v náhodném pořadí.

VZTAHY S JINÝMI VZORY

- Řetězec zodpovědnosti (Chain of Responsibility), Command, Mediator a Observer se zabývají různými způsoby propojení odesílatelů a příjemců požadavků:
  - Chain of Responsibility předává požadavek postupně podél dynamického řetězce potenciálních příjemců, dokud jej některý z nich nezpracuje.
  - Command vytváří jednosměrné propojení mezi odesílateli a příjemci.
  - Mediator eliminuje přímé propojení mezi odesílateli a příjemci, čímž je nutí komunikovat nepřímo přes objekt mediátora.
  - Observer umožňuje příjemcům dynamicky se přihlašovat a odhlašovat k přijímání požadavků.
- Rozdíl mezi Mediátorem a Observerem je často nejasný. Ve většině případů můžete implementovat kterýkoli z těchto vzorů, ale někdy je možné použít oba současně.
- Hlavním cílem Mediatoru je odstranit vzájemné závislosti mezi sadou komponent systému. Místo toho se tyto komponenty stávají závislými na jednom objektu mediátora. Cílem Observeru je navázat dynamická jednosměrná propojení mezi objekty, kde některé objekty fungují jako podřízené vůči jiným.
- Existuje populární implementace vzoru Mediator, která využívá Observer. Objekt mediátora hraje roli publisheru, a komponenty fungují jako subscribery, které se přihlašují a odhlašují k událostem mediátora. Pokud je Mediator implementován tímto způsobem, může vypadat velmi podobně jako Observer.
- Pamatujte, že Mediator lze implementovat i jinými způsoby. Například můžete trvale propojit všechny komponenty se stejným objektem mediátora. Tato implementace nebude připomínat Observer, ale stále bude příkladem vzoru Mediator.

PŘÍKLAD