NÁVRHOVÉ VZORY

Přednáška 1 - Úvod

TÉMATA

- Clean code
- Antipaterny
- Návrhové vzory
- Refactoring

CO JSOU TO NÁVRHOVÉ VZORY

- Návrhové vzory jsou typická řešení běžně se vyskytujících problémů v softwarovém designu.
- Jsou to jako předpřipravené plány, které si můžete přizpůsobit, abyste vyřešili opakující se problém ve svém kódu.
- Nemůžete vzít vzor a jednoduše jej zkopírovat do svého programu tak, jak to jde s hotovými funkcemi nebo knihovnami.
- Vzor není konkrétní kus kódu, ale obecný koncept řešení určitého problému.
- Můžete se řídit podrobnostmi vzoru a implementovat řešení, které odpovídá realitě vašeho programu.
- Vzory jsou často zaměňovány s algoritmy, protože oba pojmy popisují typická řešení známých problémů.
- Zatímco algoritmus vždy definuje jasnou posloupnost kroků k dosažení cíle, vzor je spíše vyšší úrovní popisu řešení. Kód téhož vzoru může být v různých programech odlišný.
- Přirovnání:
  - algoritmus je jako kuchařský recept – oba mají jasně dané kroky k dosažení výsledku.
  - vzor je spíše jako technický výkres – ukazuje, jak má výsledek vypadat a jaké má mít vlastnosti, ale přesný postup realizace je na vás.

ZÁKLADNÍ STAVEBNÍ ČÁSTI

- Většina vzorů je popsána velmi formálně, aby je lidé mohli znovu použít v různých kontextech. Zde jsou části, které se obvykle ve specifikaci vzoru nacházejí:
  - **Záměr vzoru** stručně popisuje jak problém, tak i řešení.
  - **Motivace** blíže vysvětluje problém a ukazuje, jaké řešení vzor umožňuje.
  - **Struktura tříd** ukazuje jednotlivé části vzoru a jejich vzájemné vztahy.
  - **Ukázka kódu** v jednom z populárních programovacích jazyků usnadňuje pochopení podstaty vzoru.

HISTORIE VZORŮ

- Návrhové vzory jsou typická řešení běžných problémů v **objektově orientovaném** návrhu.
- Když se nějaké řešení opakuje znovu a znovu v různých projektech, někdo mu nakonec dá jméno a podrobně ho popíše - „objeví“ vzor.
- Koncept vzorů byl poprvé popsán Christopherem Alexanderem v knize *A Pattern Language: Towns, Buildings, Construction*.
- Kniha popisuje „jazyk“ pro navrhování městského prostředí.
- Jednotkami tohoto jazyka jsou vzory.
- Ty mohou určovat například, jak vysoko mají být okna, kolik pater má mít budova, jak velké mají být zelené plochy v sousedství a podobně.
- Myšlenku převzali čtyři autoři: Erich Gamma, John Vlissides, Ralph Johnson a Richard Helm. V roce 1994 vydali knihu *Design Patterns: Elements of Reusable Object-Oriented Software*, v níž koncept návrhových vzorů aplikovali na programování.
- Kniha představila 23 vzorů řešících různé problémy objektově orientovaného návrhu a velmi rychle se stala bestsellerem.
- Kvůli jejímu dlouhému názvu se jí začalo říkat „kniha od gangu čtyř“, což se brzy zkrátilo jen na „GoF knihu“.
- Od té doby byly objeveny desítky dalších objektově orientovaných vzorů.
- „Přístup pomocí vzorů“ se stal velmi populárním i v jiných oblastech programování, takže dnes existuje spousta dalších vzorů mimo objektově orientovaný návrh.

PROČ SE ZAJÍMAT O VZORY?

- Pravda je taková, že můžete pracovat jako programátor i mnoho let, aniž byste znali jediný návrhový vzor.
- I v takovém případě ale možná některé vzory používáte, aniž byste o tom věděli.
- Proč byste tedy měli trávit čas jejich studiem?
- Návrhové vzory jsou sada ověřených řešení běžných problémů v návrhu softwaru.
- I když se s těmito problémy nikdy nesetkáte, znalost vzorů je stále užitečná, protože vás naučí řešit různé druhy problémů pomocí principů objektově orientovaného návrhu.
- Návrhové vzory také definují společný jazyk, který můžete používat vy i váš tým pro efektivnější komunikaci.
- Můžete říct: „Na to použijme Singleton,“ a všichni budou chápat, co tím myslíte.
- Není třeba vysvětlovat, co Singleton je, pokud znáte vzor a jeho název.
- Velmi často se pak setkáváme s dotazy na vzory při vstupních pohovorech a testech

JE TO VŽDY DOBRÉ?

- Zdá se, že jen líní lidé ještě nekritizovali návrhové vzory.
- Podívejme se na nejtypičtější argumenty proti jejich používání.
- **Berličky pro slabý programovací jazyk**
  - Potřeba vzorů obvykle vyvstává tehdy, když si lidé zvolí programovací jazyk nebo technologii, která postrádá potřebnou úroveň abstrakce.
  - V takovém případě se vzory stávají jakousi berličkou, která jazyku dodává tolik potřebné „superschopnosti“.
  - Například vzor *Strategy* lze v mnoha moderních programovacích jazycích realizovat prostě pomocí anonymní (lambda) funkce.
- **Neefektivní řešení**
  - Vzory se snaží systematizovat přístupy, které jsou už beztak široce využívané.
  - Toto sjednocení mnozí vnímají jako dogma a vzory pak implementují „doslova“, aniž by je přizpůsobili kontextu svého projektu.
- **Neopodstatněné použití**
  - Když máte jen kladivo, všechno vypadá jako hřebík.
- To je problém, který pronásleduje mnoho začátečníků, kteří se s návrhovými vzory právě seznámili.
- Jakmile se o vzorech dozvědí, snaží se je aplikovat všude, dokonce i v situacích, kde by naprosto stačil jednodušší kód.

KLASIFIKACE NÁVRHOVÝCH VZORŮ

- Návrhové vzory se liší svou složitostí, úrovní podrobnosti a měřítkem použitelnosti na celý navrhovaný systém.
- Líbí se mi přirovnání ke stavbě silnic: křižovatku můžete udělat bezpečnější buď instalací semaforů, nebo vybudováním celého vícepodlažního uzlu s podchody pro chodce.
- Nejzákladnější a nízkoúrovňové vzory se často nazývají *idiomy*.
- Obvykle platí jen pro jeden konkrétní programovací jazyk.
- Nejuniverzálnější a nejvýše postavené vzory jsou *architektonické vzory*.
- Vývojáři je mohou implementovat prakticky v libovolném jazyce.
- Na rozdíl od ostatních vzorů mohou sloužit k návrhu architektury celé aplikace.
- Kromě toho lze všechny vzory kategorizovat podle jejich záměru, neboli účelu.
- Budeme se bavit o třech hlavní skupinách vzorů:
  - **Creational patterns** poskytují mechanismy pro vytváření objektů, které zvyšují flexibilitu a znovupoužitelnost existujícího kódu.
  - **Structural patterns** vysvětlují, jak sestavovat objekty a třídy do větších celků, přičemž tyto struktury zůstávají flexibilní a efektivní.
  - **Behavioral patterns** se starají o efektivní komunikaci a rozdělení odpovědností mezi objekty.

KATALOG VZORŮ

Creational patterns

Abstract Factory

Builder

Factory Method

Prototype

Singleton

Structural patterns

Adapter

Bridge

Composite

Decorator

Facade

Flyweight

Proxy

Behavioural patterns

Chain of Responsibility

Command

Interpreter

Iterator

Mediator

Memento

Observer

State

Strategy

Template Method

Visitor

ANTI-PATTERN

- Andrew Koenig popsal antipatterny ve své práci *"Patterns and Antipatterns"* z roku 1995 následovně:
- *Antipattern je podobný vzoru, jen s tím rozdílem, že místo řešení nabízí něco, co na první pohled vypadá jako řešení, ale ve skutečnosti jím není.*
- Antipatterny jsou opakem osvědčených postupů (*best practice*), tedy řešení, která se ukázala jako účinná.
- Často se používají proto, že se zdají fungovat, ale širší kontext nebo dlouhodobé důsledky nebývají brány v úvahu.
- Mohou se vyskytovat v návrhu softwaru, řízení projektů i v organizačním chování.
- Antipatternům bychom se měli vyhýbat.

I. PROGRAMOVACÍ ANTI-PATTERN

- Programátorské antipatterny jsou běžné chyby nebo špatné postupy při psaní zdrojového kódu.
- Tyto typy antipatternů mohou vést k problémům, jako je zvýšená složitost a snížená udržovatelnost.

SPAGHETTI CODE

- Antipattern *spaghetti code* nastává, když je kód špatně strukturovaný a obtížně srozumitelný.
- Takový kód postrádá modularitu, oddělení odpovědností a čitelnost.
- Je obtížné ho udržovat nebo upravovat, protože je tak těžko pochopitelný.
- Dobrou prevencí a nápravou jsou code review a refaktoring.
- Větší bloky kódu bychom měli rozdělit na menší znovupoužitelné části.
- Metody by měly být malé a měly by dělat jen jednu věc.
- Mnoho technik, jak předcházet nebo odstranit spaghetti code, popisuje kniha Roberta C. Martina *Clean Code: A Handbook of Agile Software Craftsmanship*.

SPAGHETTI CODE - PŘÍKLAD

```java
public class ShoppingCart {
    public static void main(String[] args) {
        double total = 0;
        String[] items = {"apple", "banana", "orange"};
        int[] quantities = {3, 2, 5};

        for (int i = 0; i < items.length; i++) {
            if (items[i].equals("apple")) {
                total += 0.5 * quantities[i];
            } else if (items[i].equals("banana")) {
                total += 0.2 * quantities[i];
            } else if (items[i].equals("orange")) {
                total += 0.8 * quantities[i];
            }
        }

        System.out.println("Total price: " + total);
    }
}
```

- Business logika, data a prezentace jsou promíchané dohromady.
- Přidání nového produktu vyžaduje úpravy na více místech v podmínkách if-else.
- Chybí oddělení odpovědností → obtížná údržba (klasický Spaghetti Code).

ŘEŠENÍ

```java
// Represents a product
class Product {
    private String name;
    private double price;

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public double getPrice() {
        return price;
    }

    public String getName() {
        return name;
    }
}
```

```java
// Represents an item in the cart
class CartItem {
    private Product product;
    private int quantity;

    public CartItem(Product product, int quantity) {
        this.product = product;
        this.quantity = quantity;
    }

    public double getTotalPrice() {
        return product.getPrice() * quantity;
    }
}
```

```java
// Represents the shopping cart
class ShoppingCart {
    private List<CartItem> items = new ArrayList<>();

    public void addItem(Product product, int quantity) {
        items.add(new CartItem(product, quantity));
    }

    public double calculateTotal() {
        return items.stream().mapToDouble(CartItem::getTotalPrice).sum();
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        Product apple = new Product("apple", 0.5);
        Product banana = new Product("banana", 0.2);
        Product orange = new Product("orange", 0.8);

        cart.addItem(apple, 3);
        cart.addItem(banana, 2);
        cart.addItem(orange, 5);

        System.out.println("Total price: " + cart.calculateTotal());
    }
}
```

CO JSME ZÍSKALI

- Princip jediné odpovědnosti – Každá třída má jasně definovanou roli.
- Udržovatelnost – Přidání nových produktů nevyžaduje žádné změny v logice košíku.
- Čitelnost – Snadno pochopíte, co která část dělá.
- Rozšiřitelnost – Lze snadno přidat slevy, daně nebo nové typy produktů.

LAVA FLOW

- Antipattern *lava flow* nastává, když v kódu zůstává část, která už není potřeba.
- Takový kód je obtížné chápat i udržovat.
- Často nevíme, proč tam kód je a k čemu slouží.
- Nepoužívaný kód bychom měli co nejdříve odstraňovat, abychom se antipatternu lava flow vyhnuli.
- Bohužel není vždy snadné nevyužitý kód rozpoznat.
- Pravidelný refaktoring však může množství takového kódu snížit a výskyt tohoto antipatternu omezit.

LAVA FLOW - PŘÍKLAD

```java
public class UserService {

    // Current working method
    public void createUser(String name) {
        System.out.println("Creating user: " + name);
        // some database logic here
    }

    // Old method no longer used
    public void createUserOld(String name, String email) {
        System.out.println("Creating user (old method): " + name + ", " + email);
        // old database logic, never called
    }

    // Another obsolete method
    public void deleteUserOld(int userId) {
        System.out.println("Deleting user with ID: " + userId);
    }
}
```

Problémy:

- createUserOld a deleteUserOld už se nepoužívají → zbytečně znečišťují kód.
- Noví vývojáři mohou být zmatení a neví, která metoda je správná.
- Zvyšuje se riziko chyb při úpravách nebo refaktoringu.

ŘEŠENÍ

```java
public class UserService {

    public void createUser(String name) {
        System.out.println("Creating user: " + name);
        // some database logic here
    }
}
```

- Čistý, přehledný kód.
- Menší riziko chyb.
- Snadnější údržba a rozšiřování.

ACCIDENTAL COMPLEXITY

- Antipattern *accidental complexity* nastává, když je řešení problému zbytečně složité.
- Může k tomu dojít z různých důvodů, například z nedostatku zkušeností nebo znalostí, snahy o „přeinženýrování“ řešení, nebo z nedostatečného důrazu na jednoduchost.
- Princip návrhu *Keep it simple, stupid (KISS)* říká, že bychom měli řešení udržovat co nejjednodušší, aby byla lépe použitelná a pochopitelná.
- To je jeden z přístupů, jak antipatternu accidental complexity předcházet.

ACCIDENTAL COMPLEXITY - PŘÍKLAD

```java
public class TextConverter {

    public String convertToUpperCase(String input) {
        if (input == null) {
            return null;
        }

        // unnecessary complexity: using multiple loops and conditions
        char[] chars = input.toCharArray();
        StringBuilder sb = new StringBuilder();

        for (int i = 0; i < chars.length; i++) {
            char c = chars[i];
            if (c >= 'a' && c <= 'z') {
                sb.append((char) (c - ('a' - 'A')));
            } else if (c >= 'A' && c <= 'Z') {
                sb.append(c);
            } else {
                sb.append(c); // non-alphabetic characters
            }
        }

        return sb.toString();
    }
}
```

Problémy:

- Příliš složité řešení pro jednoduchý úkol.
- Zvyšuje riziko chyb a obtížně se čte.
- Není nutné používat vlastní smyčky a podmínky, protože Java má vestavěné metody.

ŘEŠENÍ

```java
public class TextConverter {

    public String convertToUpperCase(String input) {
        if (input == null) {
            return null;
        }
        return input.toUpperCase();
    }
}
```

- Jednoduché a čitelné řešení.
- Menší riziko chyb.
- Snadná údržba a rozšiřitelnost.

GOD OBJECT

- Antipattern *God Object* nastává, když se jeden objekt nebo třída snaží dělat příliš mnoho, což vede k silnému provázání a snížené udržovatelnosti.
- *God Object* má obvykle příliš mnoho odpovědností a porušuje princip jediné odpovědnosti (*single responsibility principle*) v objektově orientovaném programování.
- Takový objekt bychom měli rozdělit do několika menších tříd s jasně vymezenými odpovědnostmi.

```java
public class ApplicationManager {

    private List<String> users = new ArrayList<>();
    private List<String> orders = new ArrayList<>();

    // User management
    public void addUser(String user) {
        users.add(user);
    }

    public void removeUser(String user) {
        users.remove(user);
    }

    // Order management
    public void addOrder(String order) {
        orders.add(order);
    }

    public void removeOrder(String order) {
        orders.remove(order);
    }

    // Reporting
    public void generateReport() {
        System.out.println("Users: " + users);
        System.out.println("Orders: " + orders);
    }
}
```

GOD OBJECT - PŘÍKLAD

- Třída má příliš mnoho odpovědností.
- Jakákoli změna v jedné oblasti (uživatelé, objednávky, reporty) ovlivní celou třídu.
- Těsné provázání → těžká údržba a testování.

ŘEŠENÍ

```java
// Správa uživatelů
class UserManager {
    private List<String> users = new ArrayList<>();

    public void addUser(String user) {
        users.add(user);
    }

    public void removeUser(String user) {
        users.remove(user);
    }

    public List<String> getUsers() {
        return users;
    }
}
```

```java
// Správa objednávek
class OrderManager {
    private List<String> orders = new ArrayList<>();

    public void addOrder(String order) {
        orders.add(order);
    }

    public void removeOrder(String order) {
        orders.remove(order);
    }

    public List<String> getOrders() {
        return orders;
    }
}
```

```java
// Reporting
class ReportGenerator {
    public void generateReport(List<String> users, List<String> orders) {
        System.out.println("Users: " + users);
        System.out.println("Orders: " + orders);
    }
}
```

```java
// Demo
public class Main {
    public static void main(String[] args) {
        UserManager userManager = new UserManager();
        OrderManager orderManager = new OrderManager();
        ReportGenerator reportGenerator = new ReportGenerator();

        userManager.addUser("Alice");
        userManager.addUser("Bob");

        orderManager.addOrder("Order1");
        orderManager.addOrder("Order2");

        reportGenerator.generateReport(userManager.getUsers(), orderManager.getOrders());
    }
}
```

ŘEŠENÍ

- Využili jsme principu jedné odpovědnosti
- Každá třída má jasně vymezenou odpovědnost.
- Snadnější údržba a testování.
- Lepší čitelnost a rozšiřitelnost – můžeme přidávat nové funkce bez ovlivnění ostatních tříd.

HARD CODE

- *Hard coding* je antipattern, kdy hodnoty nebo konfigurace zapisujeme přímo do zdrojového kódu programu, místo abychom je uložili do samostatného konfiguračního souboru nebo databáze.
- To ztěžuje úpravu chování programu bez změny zdrojového kódu.
- Důsledkem mohou být vyšší náklady na údržbu a menší flexibilita.
- Tomuto antipatternu se můžeme vyhnout ukládáním konfigurace do samostatných konfiguračních souborů nebo databází.
- Nástroje pro statickou analýzu kódu, jako například Sonar, také pomáhají odhalovat pevně zakódované hodnoty ve zdrojovém kódu.

HARD CODE - PŘÍKLAD

```java
public class EmailService {

    public void sendEmail(String recipient) {
        String smtpServer = "smtp.example.com";  // hard-coded
        int port = 587;                          // hard-coded
        String username = "user@example.com";    // hard-coded
        String password = "password123";         // hard-coded

        System.out.println("Sending email to " + recipient + " via " + smtpServer + ":" + port);
        // logika odeslání e-mailu
    }
}
```

- Obtížná změna serveru, portu nebo přihlašovacích údajů → musíme měnit zdrojový kód.
- Riziko bezpečnostních problémů, pokud jsou hesla v kódu.
- Nízká flexibilita → těžké použít v jiném prostředí.

```java
public class EmailService {

    private String smtpServer;
    private int port;
    private String username;
    private String password;

    public EmailService() {
        Properties props = new Properties();
        try {
            props.load(new FileInputStream("config.properties"));
            smtpServer = props.getProperty("smtp.server");
            port = Integer.parseInt(props.getProperty("smtp.port"));
            username = props.getProperty("smtp.username");
            password = props.getProperty("smtp.password");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void sendEmail(String recipient) {
        System.out.println("Sending email to " + recipient + " via " + smtpServer + ":" + port);
        // logika odeslání e-mailu
    }
}
```

properties

```
smtp.server=smtp.example.com
smtp.port=587
smtp.username=user@example.com
smtp.password=password123
```

ŘEŠENÍ

- Flexibilita – změna serveru nebo portu bez úpravy kódu.
- Bezpečnost – hesla a citlivé údaje nejsou pevně zakódované.
- Udržovatelnost – snadno upravit konfiguraci pro různá prostředí (test, produkce).

MAGIC NUMBERS

- Antipattern *magic numbers* je špatná programátorská praxe, kdy se ve zdrojovém kódu používají číselné hodnoty bez odpovídajícího pojmenování.
- Magic numbers snižují čitelnost kódu a zvyšují náchylnost k chybám, protože není jasné, co dané hodnoty představují.
- Abychom se tomuto antipatternu vyhnuli, je třeba dát těmto hodnotám smysluplný název nebo připojit jasné vysvětlení.
- Podrobnější informace lze najít v příslušných odborných článcích.

MAGIC NUMBERS - PŘÍKLAD

```
public class ShippingCostCalculator {

    public double calculateCost(double weight) {
        if (weight <= 5) {
            return weight * 10;   // 10 = price per kg for light packages?
        } else if (weight <= 20) {
            return weight * 8;    // 8 = price per kg for medium packages?
        } else {
            return weight * 5;    // 5 = price per kg for heavy packages?
        }
    }
}
```

- Není jasné, co čísla 10, 8 a 5 znamenají.
- Těžko se upravují ceny, pokud se změní pravidla.
- Zvyšuje riziko chyb při úpravách.

```java
public class ShippingCostCalculator {

    private static final double LIGHT_PACKAGE_RATE = 10.0;
    private static final double MEDIUM_PACKAGE_RATE = 8.0;
    private static final double HEAVY_PACKAGE_RATE = 5.0;

    private static final double LIGHT_PACKAGE_LIMIT = 5.0;
    private static final double MEDIUM_PACKAGE_LIMIT = 20.0;

    public double calculateCost(double weight) {
        if (weight <= LIGHT_PACKAGE_LIMIT) {
            return weight * LIGHT_PACKAGE_RATE;
        } else if (weight <= MEDIUM_PACKAGE_LIMIT) {
            return weight * MEDIUM_PACKAGE_RATE;
        } else {
            return weight * HEAVY_PACKAGE_RATE;
        }
    }
}
```

ŘEŠENÍ

- Čitelnost – jasně víme, co jednotlivé hodnoty znamenají.
- Údržba – změna cen nebo limitů je jednoduchá, stačí upravit konstanty.
- Snížení chybovosti – méně nejasností a rizika chyb při úpravách kódu.