

NÁVRHOVÉ VZORY

Přednáška 1 - Úvod

TÉMATA

- Clean code
- Antipatterny
- 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 zkopirovat 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 znova a znova 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 nešetkáte, 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 chápout, 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.
 - Tož 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: křížovatku 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 nejvíše 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, neboť účelu.
- Budeme se bavit o třech hlavních 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 refactoring.
- 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

```
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

```
// 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;
    }
}
```

```
// 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;
    }
}
```

```
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());
    }
}
```

```
// 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();
    }
}
```

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, aby se antipatternu lava flow vyhnuli.
- Bohužel není vždy snadné nevyužitý kód rozpoznat.
- Pravidelný refactoring však může množství takového kódu snížit a výskyt tohoto antipatternu omezit.

LAVA FLOW - PŘÍKLAD

```
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 refactoringu.

ŘEŠENÍ

```
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

```
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

```
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 rozšířitelnost.

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.

```
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.

```
// 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;
    }
}

// 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;
    }
}

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

// 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

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

```
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.

```
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.

```
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.

NÁVRHOVÉ VZORY

Přednáška 2 – Factory, Abstract Factory, Singleton

CREATIONAL DESIGN PATTERNS

- Creational Design Patterns poskytují různé mechanismy pro vytváření objektů, které zvyšují flexibilitu a umožňují znovupoužití existujícího kódu

FACTORY

- Factory Method je kreacionální návrhový vzor, který poskytuje rozhraní pro vytváření objektů v nadřídě, ale umožňuje podřídám upravit typ objektů, které budou vytvořeny.
- Jinými slovy: nadřída definuje „jak“ se objekt vytvoří, ale konkrétní podřída rozhoduje „jaký“ objekt to bude.

FACTORY

- Představ me si, že vytváříme aplikaci pro správu logistiky.
- První verze naší aplikace dokáže zpracovávat pouze přepravu pomocí nákladních vozidel, takže většina kódu se nachází ve třídě Truck.
- Po nějaké době se naše aplikace stane poměrně populární.
- Každý den začínáš dostávat desítky žádostí od námořních přepravních společností, které chtějí, aby aplikace podporovala i námořní logistiku.

- Skvělé zprávy, že? Ale co kód?
- V současnosti je většina našeho kódu pevně svázaná s třídou Truck.
- Přidání lodí (Ship) do aplikace by vyžadovalo úpravy napříč kódem.
- A pokud se později rozhodneme přidat další typ dopravy, budeme pravděpodobně muset všechny tyto změny opakovat znovu.
- Výsledkem bude poměrně nepřehledný kód, plný podmínek, které mění chování aplikace podle toho, jaký typ dopravního objektu se právě používá.

FACTORY

- Řešení:
- Návrhový vzor Factory Method navrhuje, abysme místo přímého vytváření objektů pomocí operátoru new používali volání speciální tovární metody.
- Nemějte obavy – objekty se stále vytvářejí pomocí new, ale toto volání probíhá uvnitř tovární metody.
- Objekty, které tato metoda vrací, se často označují jako produkty.

FACTORY

- Na první pohled se tato změna může zdát zbytečná: pouze jsme přesunuli volání konstruktoru z jedné části programu do jiné.
- Ale zamysleme se nad tím: nyní můžeme překrýt tovární metodu v podtřídě a změnit typ produktů, které tato metoda vytváří.
- Existuje však drobné omezení: podtřídy mohou vracet různé typy produktů pouze tehdy, pokud tyto produkty mají společného otce nebo rozhraní.
- Tovární metoda v otci by navíc měla mít návratový typ deklarovaný právě jako toto rozhraní.

FACTORY

- Například třídy Truck a Ship by měly implementovat rozhraní Transport, které deklaruje metodu s názvem deliver.
- Každá třída tuto metodu implementuje odlišně: nákladní auta doručují náklad po souši, zatímco lodě doručují náklad po moři.
- Tovární metoda ve třídě RoadLogistics vrací objekty typu Truck, zatímco tovární metoda ve třídě SeaLogistics vrací objekty typu Ship.
- Kód, který používá tovární metodu (často označovaný jako klientský kód), nevidí rozdíl mezi konkrétními produkty, které vracejí různé podtřídy.
- Klient zachází se všemi produkty jako s abstraktním typem Transport.
- Klient ví, že všechny dopravní objekty mají mít metodu deliver, ale jak přesně funguje, ho nezajímá.

FACTORY – KDY POUŽÍT

- Použij návrhový vzor Factory Method, když dopředu nevíš přesné typy a závislosti objektů, se kterými má tvůj kód pracovat.
- Factory Method odděluje kód pro vytváření produktů od kódu, který produkty skutečně používá. Díky tomu je snazší rozšiřovat logiku vytváření objektů nezávisle na zbytek aplikace.
- Například: pokud chceš do aplikace přidat nový typ produktu, stačí vytvořit novou podtřídu tvůrce a překrýt tovární metodu.
- Použij Factory Method také tehdy, když chceš uživatelům své knihovny nebo frameworku umožnit rozšíření jeho vnitřních komponent.
- Dědičnost je pravděpodobně nejjednodušší způsob, jak rozšířit výchozí chování knihovny nebo frameworku. Ale jak framework pozná, že má použít právě tvou podtřídu místo standardní komponenty?
- Řešením je centralizovat kód pro vytváření komponent do jediné tovární metody a umožnit komukoliv tuto metodu překrýt, spolu s rozšířením samotné komponenty.

FACTORY – KDY POUŽÍT

- Představme si, že vytváříme aplikaci pomocí open-source UI frameworku.
- Naše aplikace má mít kulatá tlačítka, ale framework nabízí pouze hranatá.
- Rozšíříme tedy standardní třídu Button a vytvoříme podtřídu RoundButton.
- Teď ale potřebujeme frameworku říct, že má používat tvou novou třídu místo výchozí.
- Řešení:
- Vytvoříme podtřídu UIWithRoundButtons z hlavní třídy frameworku a překryjeme metodu createButton.
- Zatímco tato metoda ve výchozí třídě vrací objekty typu Button, naše podtřída bude vracet RoundButton.
- Nakonec použijeme třídu UIWithRoundButtons místo UIFramework.

FACTORY – KDY POUŽÍT

- Použij návrhový vzor Factory Method, když chceme šetřit systémové prostředky tím, že budeme znovu používat existující objekty místo toho, abychom je pokaždé vytvářeli znova.
- Tuto potřebu často yídáme při práci s velkými a náročnými objekty, jako jsou databázová spojení, systémy souborů nebo síťové zdroje.
- Co je potřeba udělat pro opětovné použití existujícího objektu:
 - Nejprve musíme vytvořit úložiště, které bude sledovat všechny vytvořené objekty.
 - Když někdo požadá o objekt, program by měl vyhledat volný objekt v tomto „poolu“.
 - ... a pak ho vrátit klientskému kódu.
 - Pokud žádný volný objekt není, program by měl vytvořit nový (a přidat ho do poolu).
- To je poměrně hodně kódu! A měl by být centralizovaný na jednom místě, aby se program nezanesl duplicitní logikou.
- Nejzřejmějším a nejpohodlnějším místem, kam tento kód umístit, by mohl být konstruktor třídy, jejíž objekty se snážíme znova použít.
- Jenže konstruktor ze své podstaty vždy vrací nový objekt – nemůže vracet již existující instanci.
- Proto potřebuješ běžnou metodu, která dokáže vytvářet nové objekty i znova používat existující.
- A to zní přesně jako tovární metoda (Factory Method).

FACTORY – JAK IMPLIMENTOVAT

1. Je třeba zajistit, aby všechny produkty implementovaly stejné rozhraní. Toto rozhraní by mělo deklarovat metody, které dávají smysl pro každý produkt.
2. Do třídy factory se přidá prázdná tovární metoda. Návratový typ této metody by měl odpovídat společnému rozhraní produktů.
3. V kódu tvůrce je nutné najít všechna místa, kde se používají konstruktory produktů. Postupně se nahrazují voláním tovární metody a logika vytváření produktů se přesouvá právě do této metody.
 1. V této fázi může kód tovární metody působit nevhledně – může obsahovat rozsáhlý switch blok, který vybírá, jakou třídu produktu vytvořit. To však lze později vylepšit.
4. Následně se vytvoří sada podtříd tvůrce – jednou pro každý typ produktu, který byl uveden v tovární metodě. V těchto podtřídách se tovární metoda přepíše a odpovídající části konstrukčního kódu se přesunou z původní metody.
5. Pokud je typů produktů příliš mnoho a nedává smysl vytvářet podtřídu pro každý z nich, lze znovu použít řídicí parametr z původní třídy i v podtřídách.
6. Pokud po všech těchto přesunech zůstane základní tovární metoda prázdná, může být označena jako abstraktní. Pokud v ní něco zůstane, může být považováno za výchozí chování metody.

FACTORY – PRO A PROTI

- **Pro:**
 - Vyhnete se **silnému svázání** mezi tvůrcem a konkrétními produkty.
 - **Princip jediné odpovědnosti (SRP):** Kód pro vytváření produktů lze přesunout na jedno místo v programu, což usnadňuje jeho údržbu.
 - **Princip otevřenosti/uzavřenosti (OCP):** Do programu lze zavádět nové typy produktů, aniž by bylo nutné měnit existující klientský kód.
- **Proti:**
 - Kód se může stát složitějším, protože je potřeba zavést velké množství nových podtříd pro implementaci návrhového vzoru.
 - Nejideálnějším scénářem je situace, kdy se vzor zavádí do již existující hierarchie tříd tvůrců.

FACTORY PŘÍKLAD JAVA

- Logistics

SINGLETON

- Singleton je kreacionální návrhový vzor, který umožňuje zajistit, že daná třída bude mít pouze jednu instanci, a zároveň poskytuje globální přístupový bod k této instanci.
- Tento vzor se často používá například u konfigurací aplikace, správy připojení k databázi, nebo logovacích služeb, kde je žádoucí, aby existoval právě jeden sdílený objekt v rámci celého systému.

SINGLETON

- Návrhový vzor **Singleton** řeší **dva problémy současně**, čímž porušuje **princip jediné odpovědnosti (Single Responsibility Principle)**:
- **Zajištění, že třída má pouze jednu instanci**
 - Jedním z hlavních důvodů, proč je žádoucí kontrolovat počet instancí třídy, je **řízení přístupu ke sdílenému zdroji** – například k databázi nebo souboru.
 - Princip fungování je následující: pokud je objekt jednou vytvořen a později se aplikace pokusí vytvořit nový, místo nové instance obdrží tu, která již byla vytvořena.
 - Takové chování **nelze dosáhnout běžným konstruktorem**, protože ten ze své podstaty **vždy vrací novou instanci**.

SINGLETON

- Poskytnutí globálního přístupového bodu k instanci
 - V minulosti se často používaly globální proměnné k uchování důležitých objektů. Tyto proměnné byly sice praktické, ale zároveň velmi nebezpečné – jakýkoliv kód mohl jejich obsah přepsat a způsobit pád aplikace.
 - Podobně jako globální proměnná umožňuje vzor Singleton přístup k určitému objektu odkudkoliv v programu. Na rozdíl od globální proměnné však Singleton chrání tuto instanci před přepsáním jiným kódem.
 - Existuje i další aspekt tohoto problému: není žádoucí, aby kód, který řeší kontrolu nad jedinou instancí, byl roztržštěný po celém programu. Mnohem vhodnější je mít tuto logiku centralizovanou v jedné třídě, zejména pokud na ni zbytek systému spoléhá.
 - V dnešní době je vzor Singleton natolik rozšířený, že se za „singleton“ často označuje i řešení, které pokrývá pouze jeden z výše uvedených problémů.

SINGLETON - ŘEŠENÍ

- Všechny implementace návrhového vzoru Singleton mají společné dva kroky:
 - Výchozí konstruktor se nastaví jako privátní, aby se zabránilo ostatním objektům ve vytváření instancí pomocí operátoru new u třídy Singleton.
 - Vytvoří se statická metoda, která funguje jako konstruktor. Interně tato metoda volá privátní konstruktor pro vytvoření objektu a uloží ho do statického pole. Všechny následující volání této metody pak vracejí uloženou (kešovanou) instanci.
- Pokud má kód přístup ke třídě Singleton, může volat její statickou metodu.
- Při každém volání této metody se vždy vrací stejný objekt.

SINGLETON - VYUŽITÍ

- Vzor Singleton se používá tehdy, když má být od určité třídy v programu k dispozici pouze jedna instance, sdílená všemi klienty.
- Typickým příkladem je jediný objekt databáze, který využívají různé části aplikace.
- Tento vzor zakazuje všechny ostatní způsoby vytváření objektů dané třídy, kromě speciální metody pro vytvoření instance. Tato metoda buď vytvoří nový objekt, nebo vrátí již existující, pokud byl dříve vytvořen.
- Singleton se také hodí v případech, kdy je potřeba přesnější kontrola nad globálními proměnnými.
- Na rozdíl od globálních proměnných zaručuje Singleton, že existuje pouze jedna instance třídy.
- Tuto uloženou instanci nelze přepsat žádným jiným kódem, kromě samotné třídy Singleton.
- Je však důležité poznamenat, že toto omezení lze kdykoliv upravit – například tak, aby bylo možné vytvářet více instancí.
- Jediné místo, které je potřeba změnit, je tělo metody getInstance.

SINGLETON – JAK IMPLEMENTOVAT

- Do třídy se přidá privátní statické pole pro uložení instance Singletonu.
- Deklaruje se veřejná statická metoda pro získání instance Singletonu.
- Uvnitř této statické metody se implementuje „lazy inicializace“ – při prvním volání se vytvoří nový objekt a uloží se do statického pole.
- Při všech dalších voláních se vrací právě tato uložená instance.
- Konstruktor třídy se nastaví jako privátní.
- Statická metoda třídy bude mít stále přístup ke konstruktoru, ale ostatní objekty nikoliv.
- V klientském kódu se nahradí všechna přímá volání konstruktoru Singletonu voláním jeho statické metody pro vytvoření instance.

SINGLETON – PRO A PROTI

- Lze si být jistý, že daná třída má **pouze jednu instanci**.
- Získává se **globální přístupový bod** k této instanci.
- Objekt typu Singleton je **inicializován až při prvním požadavku**, tedy při prvním volání metody, která jej poskytuje.
- Porušuje princip Single Responsibility Principle, protože vzor řeší dva problémy současně.
- Vzor Singleton může zakrývat špatný návrh, například když jednotlivé komponenty programu vědí příliš mnoho jedna o druhé.
- Vyžaduje speciální zacházení v prostředí s více vlákny, aby nedošlo k tomu, že více vláken vytvoří instanci Singletonu vícekrát.
- Může být obtížné jednotkově testovat klientský kód, který Singleton používá, protože mnoho testovacích frameworků spoléhá na dědičnost při vytváření mock objektů.
- Jelikož konstruktor třídy Singleton je privátní a přepisování statických metod není ve většině jazyků možné, je nutné vymyslet kreativní způsob, jak Singleton otestovat.
- Nebo se testy prostě nebudou psát.
- Nebo se Singleton vůbec nepoužije.

SINGLETON

- příklad

NÁVRHOVÉ VZORY

Přednáška 3

CREATIONAL DESIGN PATTERNS

- Creational Design Patterns poskytují různé mechanismy pro vytváření objektů, které zvyšují flexibilitu a umožňují znovupoužití existujícího kódu

BUILDER

- Builder je tvořivý- creational návrhový vzor, který umožňuje vytvářet složité objekty krok za krokem.
- Tento vzor dovoluje vyrábět různé typy a reprezentace objektu pomocí stejného konstrukčního kódu.

CO ŘEŠÍ?

- Představte si složitý objekt, který vyžaduje zdlouhavou, krok za krokem prováděnou inicializaci mnoha polí a vnořených objektů.
- Takový inicializační kód bývá obvykle ukrytý uvnitř **obrovského konstruktora** s mnoha parametry — nebo ještě hůře, roztríštěný napříč klientským kódem.
- Například si představme, jak bychom mohli vytvořit objekt House (dům).
 - K postavení jednoduchého domu potřebujete vytvořit čtyři stěny a podlahu, nainstalovat dveře, zasadit pár oken a postavit střechu.
 - Ale co když chcete větší, světlejší dům se zahradou a dalšími výmožnostmi (například topením, vodovodem a elektrickým rozvodem)?
 - Nejjednodušším řešením by bylo **rozšířit základní třídu House** a vytvořit **sadu podtříd**, které by pokrývaly všechny možné kombinace parametru.
 - Nakonec byste však skončili s **velkým množstvím potomků** a každý nový parametr (například typ verandy) by vyžadoval, aby se tato hierarchie dále rozrůstala.
- Existuje ale i jiný přístup, který nezahrnuje množení potomků.
- Můžete vytvořit obrovský konstruktor přímo v základní třídě House, který bude mít všechny možné parametry určující podobu domu.
- Tento přístup sice odstraní potřebu podtříd, ale přináší jiný problém — ve **většině případů většina parametrů nebude využita**, což způsobí, že volání konstruktoru bude velmi těžkopádné.
- Například jen malé procento domů má bazén, takže parametry týkající se bazénu budou v devíti z deseti případů zbytečné.

ŘEŠENÍ

- Návrhový vzor Builder (Stavitel) navrhuje, abyste **oddělili kód pro vytváření objektu z jeho vlastní třídy** a přesunuli ho do samostatných objektů nazývaných stavitele (builders).
- Tento vzor **organizuje** konstrukci objektu do **sady kroků** (např. buildWalls, buildDoor atd.).
- Pro vytvoření objektu pak spustíte sérii těchto kroků na objektu stavitele.
- Důležité je, že **nemusíte volat všechny kroky** — můžete volat pouze ty, které jsou potřeba pro vytvoření konkrétní konfigurace objektu.
- Některé kroky konstrukce mohou vyžadovat různou implementaci, pokud je potřeba vytvořit různé podoby výsledného produktu.
- Například stěny chaty mohou být ze dřeva, zatímco stěny hradu musí být z kamene.
- V takovém případě můžete vytvořit několik různých tříd stavitele, které implementují stejnou sadu stavebních kroků, ale každý svým vlastním způsobem.
- Tyto stavitele pak můžete použít v konstrukčním procesu (tedy ve stanoveném pořadí volání stavebních kroků) k vytvoření různých typů objektů.
- Například si představte jednoho stavitele, který staví vše ze dřeva a skla, druhého, který staví z kamene a železa, a třetího, který používá zlato a diamanty.
- Voláním stejné sady kroků získáte z prvního běžný dům, z druhého malý hrad a z třetího palác.
- Tento přístup však funguje pouze tehdy, pokud klientský kód, který volá jednotlivé stavební kroky, umí komunikovat se stavitelem prostřednictvím společného rozhraní.

DIRECTOR

- Můžete zajít ještě dál a vytáhnout sekvenci volání stavebních kroků, které používáte k vytvoření produktu, do samostatné třídy nazývané ředitel (Director).
- Třída Director určuje pořadí, v jakém se mají stavební kroky provádět, zatímco stavitel (Builder) poskytuje jejich konkrétní implementaci.
- Mít třídu Director v programu není povinné – stavební kroky můžete vždy volat ve správném pořadí přímo z klientského kódu.
- Nicméně třída Director je vhodné místo pro uložení různých stavebních postupů, které pak můžete znova použít napříč programem.
- Kromě toho třída Director zcela **skryje detaily konstrukce** produktu před klientským kódem.
- Klient pouze přiřadí stavitele řediteli, spustí konstrukci pomocí ředitele a poté získá výsledek od stavitele.

VYUŽITÍ

- Použijte návrhový vzor Builder (Stavitel) k odstranění tzv. „teleskopického konstruktoru“.
- Řekněme, že máte konstruktor s deseti volitelnými parametry.
- Volání takového „monstra“ je velmi nepraktické, proto konstruktor přetížíte a vytvoříte několik kratších verzí s menším počtem parametrů.
- Tyto konstruktory stále odkazují na hlavní konstruktor a předávají výchozí hodnoty pro všechny vynechané parametry.

```
class Pizza {  
    Pizza(int size) { ... }  
    Pizza(int size, boolean cheese) { ... }  
    Pizza(int size, boolean cheese, boolean pepperoni) { ... }  
    // ...  
}
```

- Vytvoření takového „monstra“ je možné pouze v jazycích, které podporují přetěžování metod, jako jsou C# nebo Java.
- Návrhový vzor Builder umožňuje stavět objekty krok za krokem, přičemž použijete pouze ty kroky, které opravdu potřebujete. Po implementaci tohoto vzoru už nemusíte cpát desítky parametrů do konstruktorů.

VYUŽITÍ 2

- Použijte Builder, když chcete, aby váš kód dokázal vytvářet různé podoby nějakého produktu (například kamenné a dřevěné domy).
- Vzor Builder se hodí, když konstrukce různých podob produktu zahrnuje podobné kroky, lišící se jen detaily.
- Základní rozhraní stavitele (base builder interface) definuje všechny možné konstrukční kroky a konkrétní stavitelé (concrete builders) tyto kroky implementují pro vytvoření konkrétních podob produktu.
- Mezitím třída Director určuje pořadí konstrukce.

VYUŽITÍ 3

- Použijte Builder k vytvoření Composite stromů nebo jiných složitých objektů.
- Vzor Builder umožňuje stavět produkty krok za krokem.
- Některé kroky lze odložit bez toho, aby se poškodil výsledný produkt.
- Dokonce můžete kroky volat i rekurzivně, což se hodí, pokud potřebujete vytvořit strom objektů.
- Stavitel neodhaluje nedokončený produkt během provádění kroků konstrukce.
- Tím se zabraňuje tomu, aby klientský kód získal neúplný výsledek.

JAK IMPLEMENTOVAT?

1. Ujistěte se, že dokážete jasně definovat společné konstrukční kroky pro vytváření všech dostupných podob produktu. Jinak nebude možné pokračovat v implementaci vzoru.
2. Tyto kroky deklarujte v základním rozhraní stavitele (base builder interface).
3. Vytvořte konkrétní třídu stavitele pro každou podobu produktu a implementujte v ní jejich konstrukční kroky.
 1. Nezapomeňte implementovat metodu pro získání výsledku konstrukce.
 2. Důvod, proč tuto metodu nelze deklarovat přímo v rozhraní stavitele, je ten, že různí stavitelé mohou vytvářet produkty, které nemají společné rozhraní.
 3. Proto nevíte, jaký by měl být návratový typ takové metody.
 4. Pokud ale pracujete s produkty ze společné hierarchie, lze metodu pro získání výsledku bezpečně přidat do základního rozhraní.
4. Zvažte vytvoření třídy Director. Ta může zapouzdřit různé způsoby konstrukce produktu při použití stejného objektu stavitele.
5. Klientský kód vytváří objekty stavitele i ředitele.
 1. Před zahájením konstrukce musí klient předat objekt stavitele řediteli.
 2. Obvykle klient toto provede pouze jednou, prostřednictvím parametrů konstruktoru třídy ředitele.
 3. Ředitel pak používá stavitele ve všech dalších krocích konstrukce.
 4. Existuje také alternativní přístup, kdy je stavitel předán konkrétní metodě pro konstrukci produktu v ředidle.
6. Výsledek konstrukce lze získat přímo od ředitele pouze pokud všechny produkty následují stejné rozhraní. V opačném případě by měl klient výsledek získat přímo od stavitele.

PRO A PROTI

Výhody

- Můžete stavět objekty krok za krokem, odkládat některé kroky konstrukce nebo volat kroky rekursivně.
- Stejný konstrukční kód lze znova použít při vytváření různých podob produktu.
- Princip jediné odpovědnosti (Single Responsibility Principle): můžete izolovat složitý konstrukční kód od obchodní logiky produktu.

Nevýhody

- Celková složitost kódu však narůstá, protože vzor vyžaduje vytvoření několika nových tříd.

VZTAHY S JINÝMI VZORY

- Mnoho návrhů začíná použitím Factory Method (méně složité a snadněji přizpůsobitelné pomocí podtříd) a postupně se vyvíjí směrem k Abstract Factory, Prototype nebo Builder (flexibilnější, ale složitější).
- Builder se zaměřuje na konstrukci složitých objektů krok za krokem. Abstract Factory se specializuje na vytváření rodin souvisejících objektů. Abstract Factory vrací produkt okamžitě, zatímco Builder vám umožňuje spustit další konstrukční kroky před získáním produktu.
- Builder můžete použít při vytváření složitých Composite stromů, protože můžete naprogramovat jeho konstrukční kroky tak, aby fungovaly rekurzivně.
- Builder lze kombinovat s Bridge: třída Director hraje roli abstrakce, zatímco různí stavitele (builders) působí jako implementace.
- Abstract Factory, Builder i Prototype lze implementovat jako Singletony.

PŘÍKLAD

- Postupná výroba automobilů V tomto příkladu návrhový vzor Builder umožňuje krok za krokem stavět různé modely automobilů.
- Příklad také ukazuje, jak Builder produkuje produkty různých druhů (např. návod k obsluze auta) pomocí stejných stavebních kroků.
- Ředitel (Director) řídí pořadí konstrukce.
- Ví, které kroky volat, aby vznikl konkrétní model auta.
- Pracuje se staviteli pouze prostřednictvím jejich společného rozhraní, což umožňuje předávat řediteli různé typy stavitelů.
- Konečný výsledek se získává z objektu stavitele, protože ředitel nemůže vědět, jaký typ produktu vznikne.
- Pouze objekt Builder přesně ví, co staví.

PROTOTYPE

- Prototype je tvořivý (creační) návrhový vzor, který vám umožňuje kopírovat existující objekty, aníž by byl váš kód závislý na jejích třídách.

CO ŘEŠÍ?

- Řekněme, že máte nějaký objekt a chcete vytvořit jeho přesnou kopii.
- Jak byste to udělali?
- Nejprve musíte vytvořit nový objekt stejné třídy.
- Poté musíte projít všechna pole původního objektu a zkopírovat jejich hodnoty do nového objektu.
- Ale je tu jeden háček.
- Ne všechny objekty lze tímto způsobem zkopírovat, protože některá pole objektu mohou být privátní a nepřístupná zvenčí.
- A to není vše — přímý přístup má ještě další problém.
- Protože musíte znát třídu objektu, abyste mohli vytvořit jeho kopii, váš kód se stává závislým na této třídě.
- další komplikace: Někdy znáte pouze rozhraní, které objekt implementuje, ale ne jeho konkrétní třídu — například v situaci, kdy parametr metody přijímá libovolné objekty, které dodržují určité rozhraní.

ŘEŠENÍ

- Vzor Prototype (Prototyp) přenáší proces klonování na samotné objekty - delegace
- Tento vzor definuje společné rozhraní pro všechny objekty, které podporují klonování.
- Díky tomuto rozhraní můžete klonovat objekt, aniž by byl váš kód závislý na jeho konkrétní třídě.
- Obvykle toto rozhraní obsahuje pouze jednu metodu – clone().
- Implementace metody clone je ve všech třídách velmi podobná.
- Metoda vytvoří nový objekt aktuální třídy a zkopiuje všechny hodnoty polí ze starého objektu do nového.
- Můžete dokonce kopírovat i privátní pole, protože většina programovacích jazyků umožňuje objektům přistupovat k privátním polím jiných objektů stejné třídy.
- Objekt, který podporuje klonování, se nazývá prototyp.
- Pokud mají vaše objekty desítky polí a stovky možných konfigurací, může být klonování vhodnou alternativou k dědičnosti.
- Jak to funguje: Vytvoříte sadu objektů, nakonfigurovaných různými způsoby.
- Kdykoli potřebujete nový objekt, který se některému z nich podobá, jednoduše zkopírujete prototyp místo toho, abyste vytvářeli nový objekt od začátku.

POUŽITÍ VZORU

- Pokud váš kód nemá záviset na konkrétních třídách objektů, které potřebujete kopírovat.
 - To nastává často, když váš kód pracuje s objekty, které jsou vám předávány z externího (3rd-party) kódu prostřednictvím nějakého rozhraní.
 - Konkrétní třídy těchto objektů nejsou známé a ani na nich nemůžete být závislí, i kdybyste chtěli.
 - Vzor Prototype poskytuje klientskému kódu obecné rozhraní pro práci se všemi objekty, které podporují klonování.
 - Toto rozhraní dělá klientský kód nezávislým na konkrétních třídách objektů, které klonuje.
- Pokud chcete snížit počet podtříd, které se liší pouze způsobem, jakým inicializují své objekty.
 - Představte si složitou třídu, která vyžaduje náročnou konfiguraci před použitím.
 - Existuje několik běžných způsobů, jak tuto třídu nakonfigurovat, a tento kód je roztržtěn po celé aplikaci.
 - Abyste odstranili duplicitu, vytvoříte několik podtříd, do jejichž konstruktorů umístíte potřebný inicializační kód.
 - Tím sice problém s duplicitou vyřešíte, ale zároveň získáte spoustu zbytečných podtříd.
 - Vzor Prototype vám umožní použít sadu předem vytvořených objektů, nakonfigurovaných různými způsoby, jako prototypy.
 - Místo vytváření nové instance podtřidy odpovídající konkrétní konfiguraci může klient jednoduše vyhledat vhodný prototyp a zkopirovat ho.

JAK IMPLEMENTOVAT?

1. Vytvořte rozhraní prototypu a deklarujte v něm metodu `clone()`. Nebo prostě přidejte tuto metodu do všech tříd existující hierarchie, pokud nějakou máte.
2. Třída prototypu musí definovat alternativní konstruktor, který přijímá objekt téže třídy jako argument. Konstruktor musí zkopirovat hodnoty všech polí definovaných v třídě z předaného objektu do nově vytvořené instance. Pokud měníte podtřídu, musíte volat konstruktor nadřazené třídy, aby nadřída mohla zpracovat klonování svých privátních polí.
3. Pokud váš programovací jazyk nepodporuje přetěžování metod, nebudete schopni vytvořit samostatný „prototypový“ konstruktor. Kopírování dat objektu do nově vytvořeného klounu se pak musí provést uvnitř metody `clone()`. Přesto je lepší mít tento kód v běžném konstruktoru, protože výsledný objekt je ihned po zavolení operátoru `new` plně nakonfigurován.
4. Metoda klonování obvykle sestává z jediné řádky: volání operátoru `new` s prototypovou verzí konstruktoru. Každá třída musí explicitně přepsat metodu klonování a použít své vlastní jméno třídy spolu s `new`. Jinak by metoda klonování mohla vytvořit objekt nadřazené třídy.
5. Volitelně můžete vytvořit centralizovaný registr prototypů, který bude uchovávat katalog často používaných prototypů.
 1. Registr můžete implementovat jako novou třídu továrny nebo jej umístit do základní třídy prototypu s statickou metodou pro získání prototypu.
 2. Tato metoda by měla vyhledávat prototyp podle kritérií, která klientský kód předá metodě. Kritéria mohou být buď jednoduchý textový tag, nebo komplexní sada parametrů.
 3. Po nalezení vhodného prototypu by registr měl zkopirovat jeho instanci a vrátit kopii klientovi.
6. Nakonec nahraďte přímé volání konstruktorů podtříd voláním tovární metody registru prototypů.

PRO A PROTI

PRO

- Můžete klonovat objekty bez závislosti na jejich konkrétních třídách.
- Můžete odstranit opakující se inicializační kód a místo něj využít klonování předem vytvořených prototypů.
- Můžete snáze vytvářet složité objekty.
- Získáte alternativu k dědičnosti, když pracujete s přednastavenými konfiguracemi složitých objektů.

PROTI

- Klonování složitých objektů s kruhovými odkazy může být velmi komplikované.

VZTAHY S JINÝMI VZORY

- Mnoho návrhů začíná použitím Factory Method (méně složité a snadněji přizpůsobitelné pomocí podtříd) a postupně se vyvíjí směrem k Abstract Factory, Prototype nebo Builder (flexibilnější, ale složitější).
- Třídy Abstract Factory jsou často založeny na sadě Factory Methods, ale můžete také použít Prototype pro skládání metod v těchto třídách.
- Prototype se hodí, když potřebujete uchovávat kopie příkazů (Commands) do historie.
- Návrhy, které hojně využívají Composite a Decorator, často profitují z použití Prototype.
- Použití tohoto vzoru umožňuje klonovat složité struktury místo toho, abyste je znova stavěli od začátku.
- Prototype není založen na dědičnosti, takže nemá její nevýhody.
- Na druhou stranu vyžaduje složitou inicializaci klonovaného objektu.
- Factory Method je založena na dědičnosti, ale nevyžaduje inicializační krok.
- Někdy může být Prototype jednodušší alternativou k Memento.
- To funguje, pokud je objekt, jehož stav chcete uložit do historie, poměrně jednoduchý a nemá odkazy na externí zdroje, nebo pokud je snadné tyto odkazy znova navázat.
- Abstract Factory, Builder i Prototype lze implementovat jako Singletony.

PŘÍKLAD

- Kopírování grafických prvků

NÁVRHOVÉ VZORY

Přednáška 4 – Antipatterns II

II. METODOLOGICKÉ ANTI-PATTERNS

- Metodologické antipatterny jsou běžné úskalí nebo problémy, které mohou nastat při navrhování a implementaci projektu či řešení.
- **Premature Optimization**
 - Předčasná optimalizace je antipattern, kdy optimalizujeme řešení pro výkon ještě před tím, než víme, zda tuto optimalizaci skutečně potřebujeme.
 - Takové optimalizace mají jen náklady a žádné výhody.
 - Mohou vést k několika problémům, včetně zvýšené složitosti, snížené čitelnosti a horší udržovatelnosti.
 - Optimalizovat řešení pro výkon bychom měli jen tehdy, pokud je to skutečně potřeba.
- **Reinventing the Wheel**
 - Antipattern *reinventing the wheel* nastává, když je řešení problému zbytečně znova vynalézáno místo využití existujícího řešení nebo rozšíření již existující práce.
 - To vede ke zvýšení času vývoje a nákladů.
 - Nové řešení navíc nemusí být tak dobře otestované jako existující řešení.
- **Copy and Paste Programming**
 - Antipattern *copy and paste programming* nastává, když je zdrojový kód kopírován z jednoho místa na jiné místo toho, aby byl znova použit prostřednictvím abstrakce.
 - To vede k vyšším nákladům na údržbu, snížené čitelnosti kódu a menší znovupoužitelnosti.
 - Místo kopírování kódu ho můžeme **refaktorovat a znova použít**.
 - Například lze kód vložit do pomocné (*utility*) třídy, ke které mají přístup různé části programu.

PREMATURE OPTIMIZATION - PŘÍKLAD

```
public class StringConcatenator {  
  
    public String concatenate(String[] words) {  
        String result = "";  
        for (int i = 0; i < words.length; i++) {  
            // Předčasná optimalizace: snažíme se ušetřit paměť tím, že nepo  
            result += words[i]; // Vytváří se nový objekt při každé iteraci  
        }  
        return result;  
    }  
}
```

- Používání operátoru `+` v cyklu s `String` vede k neefektivnímu vytváření nových objektů.
- Vývojář se snažil „optimalizovat“ tím, že nepoužil další třídu jako `StringBuilder`, ale výsledek je horší výkon.
- Navíc, pokud by se ukázalo, že metoda se volá zřídka nebo s malým polem, optimalizace by byla zbytečná.

ŘEŠENÍ

```
public class StringConcatenator {  
    public String concatenate(String[] words) {  
        StringBuilder result = new StringBuilder();  
        for (String word : words) {  
            result.append(word);  
        }  
        return result.toString();  
    }  
}
```

- Použití `StringBuilder` je standardní a efektivní způsob spojování řetězců.
 - Kód je čitelnější, jednodušší na údržbu a výkonnější.
 - Optimalizace je provedena až poté, co je jasné, že výkon je důležitý (např. na základě profilování)
-
- **Neoptimalizujme dřív, než víme, že je to potřeba.**
 - **Nejprve pišme čistý, srozumitelný kód.**
 - **Optimalizujme na základě dat – např. pomocí profilování nebo benchmarků**

REINVENTING THE WHEEL - PŘÍKLAD

- Vývojář vytváří vlastní řešení pro problém, který už má kvalitní, ověřené řešení – často ve formě knihovny nebo frameworku.
- Místo toho, aby použil existující nástroj, „vynalézá kolo znova“ – což vede ke zbytečné práci, chybám a horší údržbě.

```
public class CustomJsonParser {

    public Map<String, String> parse(String json) {
        Map<String, String> result = new HashMap<>();
        json = json.replace("{", "").replace("}", "");
        String[] pairs = json.split(",");
        for (String pair : pairs) {
            String[] keyValue = pair.split(":");
            result.put(keyValue[0].trim(), keyValue[1].trim());
        }
        return result;
    }
}
```

```
import com.google.gson.Gson;
import java.util.Map;

public class JsonParserUsingGson {

    public Map<String, String> parse(String json) {
        Gson gson = new Gson();
        return gson.fromJson(json, Map.class);
    }
}
```

ŘEŠENÍ

- Kód je kratší, čitelnější a spolehlivější.
- Gson řeší složité případy, typovou bezpečnost, validaci a je dobře testovaný.
- Vývojář se může soustředit na logiku aplikace, ne na znovuvynalézání základních nástrojů.
- „**Nepišme vlastní knihovnu, pokud ji už někdo napsal lépe.**“
- Vždy se nejprve podívejme, zda už neexistuje kvalitní řešení.
- Používejme knihovny, které jsou aktivně udržované, dokumentované a komunitou ověřené.
- Vlastní implementace má smysl jen tehdy, když máme specifické požadavky, které žádná knihovna nesplňuje.

```
public class OrderProcessor {  
    public void processOrder(Order order) {  
        double totalPrice = order.getItemPrice() * order.getQuantity();  
        totalPrice += totalPrice * 0.1; // Přidání daně  
        System.out.println("Order ID: " + order.getId());  
        System.out.println("Total Price: " + totalPrice);  
    }  
  
    public class InvoiceGenerator {  
        public void generateInvoice(Order order) {  
            double totalPrice = order.getItemPrice() * order.getQuantity();  
            totalPrice += totalPrice * 0.1; // Přidání daně  
            System.out.println("Invoice ID: " + order.getId());  
            System.out.println("Total Price: " + totalPrice);  
        }  
    }  
}
```

COPY AND PASTE PROGRAMMING

- Stejný výpočet ceny a výpis se opakuje.
- Pokud se změní logika výpočtu (např. sazba daně), musí se upravit na více místech.
- Vede to k chybám, nejasnostem a vyšší náročnosti na údržbu

ŘEŠENÍ

```
public class OrderUtils {  
  
    public static double calculateTotalPrice(Order order) {  
        double totalPrice = order.getItemPrice() * order.getQuantity();  
        return totalPrice + totalPrice * 0.1;  
    }  
  
    public static void printOrderDetails(String label, Order order) {  
        double totalPrice = calculateTotalPrice(order);  
        System.out.println(label + " ID: " + order.getId());  
        System.out.println("Total Price: " + totalPrice);  
    }  
  
}  
  
public class OrderProcessor {  
    public void processOrder(Order order) {  
        OrderUtils.printOrderDetails("Order", order);  
    }  
}  
  
public class InvoiceGenerator {  
    public void generateInvoice(Order order) {  
        OrderUtils.printOrderDetails("Invoice", order);  
    }  
}
```

- Kód je modulární, přehledný a snadno udržovatelný.
- Změna výpočtu se provede na jednom místě.
- Dodržuje princip DRY – Don't Repeat Yourself.
- „Když kopírujeme kód, kopírujeme i chyby.“
- Vždy se zamysleme, zda by se kód neměl extrahovat do metody nebo třídy.
- IDE jako IntelliJ nebo Eclipse mají funkce jako „Extract Method“, které nám s tím pomůžou.
- Používejme utility třídy, design patterns (např. Strategy, Template Method) nebo funkcionální přístup, pokud se hodí.

III. SOFTWARE TESTING ANTI-PATTERNS

- Antipatterny softwarového testování jsou běžné chyby nebo špatné postupy ve fázi testování softwaru. Tyto antipatterny mohou vést ke sníženému pokrytí testy, zvýšeným nákladům na údržbu a nižší spolehlivosti.
- **Wrong Kind of Test**
 - Když používáme nesprávný typ testu, neúmyslně aplikujeme antipattern *wrong kind of test* na naši kódovou bázi.
 - Je zásadní porozumět různým typům testů a vědět, kdy který použít.
 - Použití nesprávného typu testu může vést ke sníženému pokrytí, vyšším nákladům na údržbu a nižší spolehlivosti.
- **Testing Internal Implementation**
 - Antipattern *testing internal implementation* nastává, když jsou testy napsány tak, že jsou vázány na interní implementaci softwarové komponenty místo na její externí chování.
 - To zvyšuje náklady na údržbu, protože testy je nutné aktualizovat při každé změně interní implementace.
- **Happy Path**
 - Když se při testování kódu zaměřujeme pouze na nejběžnější a očekávané případy, aplikujeme antipattern *happy path*.
 - Je důležité testovat i neočekávané případy a scénáře, jinak je vysoké riziko chyb a nepředvídatelného chování.

WRONG KIND OF TEST - PŘÍKLAD

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

```
public class CalculatorTest {  
  
    @Test  
    public void testAdd() {  
        Calculator calculator = new Calculator();  
  
        // Špatný test: kontroluje vnitřní stav nebo konkrétní implementaci  
        int result = calculator.add(2, 3);  
        assertEquals(5, result); // OK, ale ...  
  
        // Zbytečný test: kontroluje, že metoda používá operátor +  
        // (např. pomocí reflection nebo mockování, což je zbytečné pro tuto  
    }  
}
```

- Test je příliš jednoduchý, ale v praxi se často stává, že testy kontrolují konkrétní implementaci místo chování.
- Pokud by se metoda změnila (např. použila jiný způsob výpočtu), test by selhal, i když výsledek by byl správný.
- Testy by měly být odolné vůči refactoringu.

```
public class CalculatorTest {  
  
    @Test  
    public void shouldReturnSumOfTwoNumbers() {  
        Calculator calculator = new Calculator();  
        assertEquals(5, calculator.add(2, 3));  
        assertEquals(0, calculator.add(-2, 2));  
        assertEquals(-5, calculator.add(-2, -3));  
    }  
}
```

ŘEŠENÍ

- Testuje se chování metody – různé vstupy a očekávané výstupy.
- Test je jednoduchý, čitelný a nezávislý na konkrétní implementaci.
- Pokud se změní způsob výpočtu, ale výsledek zůstane stejný, test stále projde
- „Dobrý test je jako bezpečnostní pás – chrání tě, ale neomezuje v pohybu.“
- Piš testy, které ověřují chování, ne konkrétní implementaci.
- Testy by měly být jednoduché, izolované a smysluplné.
- Refactoring by neměl nutit k přepisování testů, pokud se nezměnilo chování.

WRONG KIND OF TEST – PŘÍKLAD 2

```
// Třída, kterou testujeme
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

- Tento test je křehký — testuje implementační detaily, ne funkční chování.
- Pokud by se metoda interně změnila (např. používala long a přetypovala výsledek), test by zbytečně selhal, i když výsledek by byl správný.

```
// Test – špatný typ testu
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {
    @Test
    public void testInternalImplementation() {
        Calculator calc = new Calculator();

        // Předpokládáme, že metoda add používá konkrétní interní proměnnou "sum"
        // (což vůbec není garantováno rozhraním)
        int a = 3;
        int b = 5;
        int expected = 8;

        int result = calc.add(a, b);
        assertEquals(expected, result);

        // ❌ Navíc testujeme implementační detail (např. že metoda vrací přesně int)
        assertTrue(((Object)result) instanceof Integer); // zbytečné
    }
}
```

ŘEŠENÍ 2

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {
    @Test
    public void testAddReturnsCorrectSum() {
        Calculator calc = new Calculator();

        assertEquals(8, calc.add(3, 5));
        assertEquals(0, calc.add(-3, 3));
        assertEquals(-10, calc.add(-7, -3));
    }
}
```

- Tento test kontroluje chování, nikoli způsob, jakým je metoda implementována.
- Pokud se v budoucnu implementace změní (např. použije BigDecimal, nebo jinou interní logiku), test stále zůstane platný.

WRONG KIND OF TEST – PŘÍKLAD 3

```
// REST controller
@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/{id}")
    public ResponseEntity<User> getUser(@PathVariable Long id) {
        return ResponseEntity.ok(userService.findUserById(id));
    }
}
```

```
// Špatný test
@SpringBootTest
@AutoConfigureMockMvc
class UserControllerWrongTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void testInternalServiceIsCalled() throws Exception {
        UserController controller = new UserController();
        UserService mockService = Mockito.mock(UserService.class);
        controller.userService = mockService;

        controller.getUser(1L);

        // ❌ Testujeme interní implementaci - že metoda "findUserById" byla zavolána
        // místo aby nás zajímal výsledek API
        Mockito.verify(mockService).findUserById(1L);
    }
}
```

ŘEŠENÍ 3

- testuje chování API z pohledu klienta, nezajímá se o to, jak je služba implementována
- bude stále platný i při vnitřní změně (např. jiná služba, jiná databáze).

```
@SpringBootTest
@AutoConfigureMockMvc
class UserControllerGoodTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void test GetUser Returns UserData() throws Exception {
        mockMvc.perform(get("/users/1"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.id").value(1))
            .andExpect(jsonPath("$.name").value("Alice"));
    }
}
```

TESTING INTERNAL IMPLEMENTATION - PŘÍKLAD

```
public class UserService {  
    private int retryCount = 3;  
  
    public boolean login(String username, String password) {  
        // nějaká logika  
        return true;  
    }  
}
```

```
public class UserServiceTest {  
  
    @Test  
    public void testRetryCountInternalState() throws Exception {  
        UserService service = new UserService();  
  
        // Špatný test: přístup k privátnímu poli pomocí reflexe  
        Field field = UserService.class.getDeclaredField("retryCount");  
        field.setAccessible(true);  
        int value = (int) field.get(service);  
  
        assertEquals(3, value); // Testuje vnitřní stav, ne chování  
    }  
}
```

- Test porušuje zapouzdření (encapsulation).
- Změna názvu pole nebo jeho typu rozbije test, i když login stále funguje.
- Test neověřuje, zda login funguje správně – jen že existuje konkrétní implementace.

ŘEŠENÍ

```
public class UserService {  
  
    private int retryCount = 3;  
  
    public boolean login(String username, String password) {  
        for (int i = 0; i < retryCount; i++) {  
            if (authenticate(username, password)) {  
                return true;  
            }  
        }  
        return false;  
    }  
  
    private boolean authenticate(String username, String password) {  
        return "admin".equals(username) && "secret".equals(password);  
    }  
}
```

```
public class UserServiceTest {  
  
    @Test  
    public void shouldLoginWithCorrectCredentials() {  
        UserService service = new UserService();  
        assertTrue(service.login("admin", "secret"));  
    }  
  
    @Test  
    public void shouldFailLoginWithWrongCredentials() {  
        UserService service = new UserService();  
        assertFalse(service.login("user", "wrong"));  
    }  
}
```

- Testy ověřují funkční chování, ne detaily implementace.
- Pokud se změní počet pokusů nebo způsob autentizace, testy stále fungují, pokud chování zůstane stejné.
- Kód je odolný vůči refactoringu.
- „Test by měl být jako zákazník – zajímá ho výsledek, ne jak jsi ho dosáhl.“
- Testujme co třída dělá, ne jak to dělá.
- Vyhýbejme se reflexi, přístupu k privátním polím nebo metodám.
- Pokud potřebujeme testovat vnitřní logiku, zvažme refactoring do samostatné třídy nebo metody s věřejným rozhraním.

HAPPY PATH - PŘÍKLAD

```
public class PaymentService {  
    public String processPayment(String cardNumber, double amount) {  
        // Předpokládáme, že karta je vždy platná a částka je vždy kladná  
        return "Payment of $" + amount + " processed for card " + cardNumber;  
    }  
}
```

```
public class PaymentServiceTest {  
  
    @Test  
    public void testSuccessfulPayment() {  
        PaymentService service = new PaymentService();  
        String result = service.processPayment("4111111111111111", 100.0);  
        assertEquals("Payment of $100.0 processed for card 4111111111111111",  
    }  
}
```

- Test pokrývá pouze „šťastnou cestu“ – validní vstup, žádné chyby.
- Co když je částka záporná? Co když je číslo karty neplatné nebo null?
- Kód neobsahuje žádné ošetření výjimek nebo validaci vstupů.

```
public class PaymentService {  
    public String processPayment(String cardNumber, double amount) {  
        if (cardNumber == null || cardNumber.isEmpty()) {  
            throw new IllegalArgumentException("Card number is invalid");  
        }  
        if (amount <= 0) {  
            throw new IllegalArgumentException("Amount must be positive");  
        }  
        return "Payment of $" + amount + " processed for card " + cardNumber;  
    }  
}
```

```
public class PaymentServiceTest {  
  
    @Test  
    public void testSuccessfulPayment() {  
        PaymentService service = new PaymentService();  
        String result = service.processPayment("4111111111111111", 100.0);  
        assertEquals("Payment of $100.0 processed for card 4111111111111111",  
    }  
  
    @Test  
    public void testInvalidCardNumber() {  
        PaymentService service = new PaymentService();  
        assertThrows(IllegalArgumentException.class, () -> {  
            service.processPayment("", 100.0);  
        });  
    }  
  
    @Test  
    public void testNegativeAmount() {  
        PaymentService service = new PaymentService();  
        assertThrows(IllegalArgumentException.class, () -> {  
            service.processPayment("4111111111111111", -50.0);  
        });  
    }  
}
```

ŘEŠENÍ

- Kód je robustní, ošetřuje chybové stavy.
- Testy pokrývají jak „happy path“, tak „sad path“.
- Snižuje se riziko výpadků v produkci kvůli neosetřeným vstupům.
- „Testovat jen šťastnou cestu je jako jezdit autem bez rezervy – funguje, dokud nepřijde problém.“
- Piš testy, které pokrývají validní vstupy, nevalidní vstupy, okrajové případy a výjimky.
- V produkčním kódu vždy ošetřuj vstupy, které mohou být neplatné nebo nečekané.
- Používej princip defenzivního programování – předpokládej, že vstupy mohou být chybné.

HAPPY PATH - PŘÍKLAD 2

```
// Jednoduchá služba pro převod měny
public class CurrencyConverter {
    public double convert(double amount, String from, String to) {
        if (from.equals("USD") && to.equals("EUR")) {
            return amount * 0.9;
        } else if (from.equals("EUR") && to.equals("USD")) {
            return amount * 1.1;
        }
        throw new IllegalArgumentException("Unsupported currency pair");
    }
}
```

```
// Špatný test – testuje jen happy path
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CurrencyConverterTest {
    @Test
    public void testUsdToEurConversion() {
        CurrencyConverter converter = new CurrencyConverter();
        double result = converter.convert(100, "USD", "EUR");
        assertEquals(90.0, result);
    }
}
```

- Testuje jen jeden ideální případ, kdy vše funguje správně.
- Netestuje:
 - Neplatné měny ("XXX"),
 - Převod mezi stejnými měnami,
 - Záporné hodnoty,
 - Výjimky.

```
public class CurrencyConverterTest {  
  
    private final CurrencyConverter converter = new CurrencyConverter();  
  
    @Test  
    public void testUsdToEurConversion() {  
        assertEquals(90.0, converter.convert(100, "USD", "EUR"));  
    }  
  
    @Test  
    public void testEurToUsdConversion() {  
        assertEquals(110.0, converter.convert(100, "EUR", "USD"));  
    }  
  
    @Test  
    public void testUnsupportedCurrencyThrowsException() {  
        assertThrows(IllegalArgumentException.class, () -> converter.convert(100, "USD", "GBP"));  
    }  
  
    @Test  
    public void testNegativeAmountHandled() {  
        double result = converter.convert(-100, "USD", "EUR");  
        assertEquals(-90.0, result);  
    }  
  
    @Test  
    public void testSameCurrencyHandledGracefully() {  
        // třeba později metoda bude rozšířena, tady můžeme testovat současné chování  
        assertThrows(IllegalArgumentException.class, () -> converter.convert(100, "USD", "USD"));  
    }  
}
```

ŘEŠENÍ 2

- Pokrývá i chybové a hraniční případy.
- Testuje výjimky a chování systému při neočekávaných vstupech,
- Poskytuje reálnou jistotu, že kód obstojí i mimo „šťastnou cestu“.

IV. JAK ROZPOZNAT A VYHNOUT SE ANTI PATERNS

- Je zásadní antipatterny rozpoznat a vyhnout se jím, protože mohou vést k několika negativním důsledkům:
 - Neefektivní nebo neúčinná řešení problémů
 - Zvýšená složitost a náklady na údržbu
 - Snížená čitelnost a srozumitelnost kódu
 - Snížená produktivita a morálka týmu
- Rozpoznáním a vyhýbáním se antipatternům můžeme zlepšit naše řešení a zajistit, že budou **efektivní, účinná a udržitelná**.
- To může vést k lepším výsledkům projektů a příznivějšímu pracovnímu prostředí.

IDENTIFIKACE ANTIPATTERNŮ

- Při identifikaci antipatternů v našem kódu nebo návrhu je důležité **mít otevřenou mysl a zpochybňovat své předpoklady**.
- Někdy se můžeme přichytit k řešení, které si vyžádalo hodně času a úsilí, přesto může existovat lepší řešení.
- Abychom tomu předešli, je užitečné **požádat o zpětnou vazbu od ostatních**.
- Ostatní lidé často vidí náš problém z jiné perspektivy a mohou odhalit problémy nebo neefektivnosti, které jsme přehlédli.
- Další způsob, jak antipatterny identifikovat, je **hledat varovné signály**.
- Příliš složitá nebo těžko pochopitelná řešení mohou být známkou antipatternu.
- Také **studium osvědčených postupů a běžných úskalí** nám může pomoci vyhnout se chybám.
- Je rovněž důležité **nebát se zrušit kód a začít znovu**.
- Pokud zjistíme, že jsme uvízli v antipatternu, nové začátky mohou být tím nejlepším řešením.
- Dodržováním těchto tipů můžeme zvýšit šance na **rozpoznání antipatternů** v naší práci a zlepšit naše celkové dovednosti v programování a návrhu.

VYHÝBÁNÍ SE POUŽITÍ ANTIPATERNŮ

- Při práci na softwarových projektech je zásadní **být si vědom běžných úskalí**, která mohou vést k antipatternům.
- Jednou ze strategií, jak se těmto úskalím vyhnout, je **udělat krok zpět a zvážit širší kontext problému**.
- Porozumění problému jako celku pomůže při hledání dobrého řešení.
- Další strategií je **používat osvědčené návrhové vzory a nejlepší praktiky**.
- Řešení, která se osvědčila u ostatních s podobnými problémy, jsou dobrou volbou a mohou ušetřit spoustu času.
- Užitek může přinést i **seznámení se s běžnými návrhovými vzory**.
- Další strategií je **rozložit velké problémy na menší části**.
- To pomáhá vyhnout se pocitu zahlcení a usnadňuje odhalování problémů a neefektivnosti.
- Také se **nemáme bát požádat o pomoc**, pokud si s něčím nejsme jisti.
- Vždy existuje někdo zkušenější, kdo je ochotný pomoci a sdílet své znalosti.
- Je rovněž důležité **průběžně kontrolovat náš kód**, aby bylo zajištěno, že používáme nejlepší možné řešení.
- Jakmile se o problému a jeho řešení dozvímě více, můžeme změnit svůj přístup a vyvinout lepší řešení.

ČISTÝ KÓD

- KISS
- DRY
- SOLID
- YAGNI
- „Clean Code“ označuje psaní kódu, který je snadno pochopitelný, udržovatelný a úpravitelný.
- 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 KONEC

- 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

```
// 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);
    }
}
```

```
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?

```

// 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(rectangle));
    }
}

```

OCP - PŘÍKLAD

```

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

```
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:

```
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

```
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

```
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).

```

// 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.

DIP - PŘÍKLAD

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

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

```
public class Car {  
    private Engine engine;  
  
    public Car(Engine engine) {  
        this.engine = engine;  
    }  
  
    public void startCar() {  
        engine.start();  
        System.out.println("Car is running.");  
    }  
}
```

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

- Car neví nic o konkrétním typu motoru – závisí jen na rozhraní Engine.
- Můžeš 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 tedy?“
- → 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 kódů.
- 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 znova a znova – stačí ho napsat jednou a používat opakováně.

```
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");  
    }  
}
```

```
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);  
    }  
}
```

DRY - PŘÍKLAD

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

NÁVRHOVÉ VZORY

Přednáška 5 – Čistý kód

ČISTÝ KÓD

- KISS
- DRY
- SOLID
- YAGNI
- „Clean Code“ označuje psaní kódu, který je snadno pochopitelný, udržovatelný a úpravitelný.
- 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

```
// 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);
    }
}
```

```
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?

```

// 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(rectangle));
    }
}

```

OCP - PŘÍKLAD

```

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

```
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:

```
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

```
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

```
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).

```

// 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.

DIP - PŘÍKLAD

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

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

```
public class Car {  
    private Engine engine;  
  
    public Car(Engine engine) {  
        this.engine = engine;  
    }  
  
    public void startCar() {  
        engine.start();  
        System.out.println("Car is running.");  
    }  
}
```

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

- 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 tedy?“
- → 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 kódů.
- 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 znova a znova – stačí ho napsat jednou a používat opakováně.

```
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");  
    }  
}
```

```
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);  
    }  
}
```

DRY - PŘÍKLAD

- 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 jakékoli 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 iPhone), 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

- Tedy, 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 sadou 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 tlačítka 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 třidy 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

- Subscribers 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 požadavku:
 - 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 Observera 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 subscribeři, 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

NÁVRHOVÉ VZORY

Přednáška 6

BEHAVIOURAL PATTERNS

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

CHAIN OF RESPONSIBILITY

- Řetězec zodpovědnosti (Chain of Responsibility) je behaviorální návrhový vzor, který Vám umožňuje předávat požadavky podél řetězce zpracovatelů (handlerů).
- Po obdržení požadavku se každý zpracovatel rozhodne, zda požadavek zpracuje, nebo jej předá dalšímu zpracovateli v řetězci.

CO ŘEŠÍ?

- Představte si, že pracujete na **systému pro online objednávky**.
- Chcete omezit přístup do systému tak, aby **pouze ověření uživatelé** mohli vytvářet objednávky.
- Zároveň by **uživatelé s administrátorskými oprávněními** měli mít **plný přístup ke všem objednávkám**.
- Po krátkém plánování jsme si uvědomili, že tyto kontroly je nutné provádět **postupně**.
- Aplikace se může pokusit **ověřit uživatele** pokaždé, když obdrží požadavek obsahující jeho přihlašovací údaje.
- Pokud však tyto údaje **nejsou správné** a **ověření selže, nemá smysl pokračovat** v dalších kontrolách.

- Během následujících několika měsíců jste implementoval(a) **několik dalších postupných kontrol**.
- Jeden z Vašich kolegů navrhl, že **není bezpečné předávat neupravená data přímo do systému objednávek**. Proto jste přidal(a) **další validační krok**, který **očišťuje data v požadavku**.
- Později si někdo všiml, že je systém **zranitelný vůči útokům hrubou silou (brute force)** při hádání hesel. Aby se tomu zabránilo, **rychle jste přidal(a) kontrolu**, která **filrovala opakované neúspěšné požadavky pocházející ze stejné IP adresy**.
- Další kolega navrhl, že by bylo možné **zrychlit systém** tím, že se při opakovaných požadavcích se stejnými daty **vrátí uložené (cache) výsledky**.
- Proto jste přidal(a) **další kontrolu**, která **propustí požadavek do systému pouze tehdy, pokud pro něj neexistuje vhodná odpověď v mezipaměti**.

- Kód jednotlivých kontrol, který už dříve působil chaoticky, se s každým novým rozšířením stal ještě objemnějším a nepřehlednějším.
- Změna jedné kontroly mohla ovlivnit ostatní.
- Nejhorší však bylo, že když jste se pokusil(a) znovu použít tyto kontroly pro ochranu jiných částí systému, musel(a) jste část kódu duplikovat, protože některé komponenty vyžadovaly jen určité kontroly, ale ne všechny.
- Systém se stal velmi obtížně pochopitelným a nákladným na údržbu.
- S kódem jste se nějakou dobu trápil(a), až jste se jednoho dne rozhodl(a) celé řešení přepracovat (**refaktorovat**).

ŘEŠENÍ

- Stejně jako mnoho jiných behaviorálních návrhových vzorů, i Řetězec zodpovědnosti (Chain of Responsibility) je založen na přeměně jednotlivých chování do samostatných objektů, které se nazývají zpracovatele (handlers).
- V našem případě by měla být každá kontrola oddělena do vlastní třídy, která bude mít jedinou metodu provádějící danou kontrolu.
- Požadavek spolu se svými daty je této metodě předán jako argument. Vzor navrhuje, abyste tyto handlers propojil(a) do řetězce.
- Každý propojený handler obsahuje referenci na další handler v řetězci.
- Kromě samotného zpracování požadavku handler také předává požadavek dál v řetězci.
- Požadavek tak putuje řetězcem, dokud jej neprojdou všechny handlers, které mají možnost jej zpracovat.
- A tady přichází ta nejlépší část: handler se může rozhodnout, že požadavek dále nepředá, a tím zastaví další zpracování.
- V našem příkladu se systémem objednávek handler zpracuje požadavek a poté se rozhodne, zda jej pošle dál.
- Pokud požadavek obsahuje správná data, všechny handlers mohou vykonat své hlavní úlohy – ať už jde o ověření uživatele, nebo práci s mezipamětí (cache).

- Existuje však trochu odlišný přístup, při kterém se handler po obdržení požadavku rozhodne, zda jej dokáže zpracovat.
- Pokud ano, požadavek dále nepředává.
- Takže požadavek zpracuje buď jen jeden handler, nebo žádný.
- Tento přístup je velmi běžný při zpracování událostí v hierarchii prvků grafického uživatelského rozhraní (GUI).
- Například když uživatel klikne na tlačítko, událost se šíří řetězcem GUI prvků, který začíná tlačítkem, pokračuje přes jeho kontejnery (např. formuláře nebo panely) a končí hlavním oknem aplikace.
- Událost zpracuje první prvek v řetězci, který je schopen ji obsloužit.

- Je zásadní, aby všechny třídy handlerů implementovaly stejný interface.
- Každý konkrétní handler by se měl starat pouze o to, aby následující handler měl metodu `execute`.
- Tímto způsobem můžete sestavovat řetězce za běhu programu, používat různé handlery a přitom nespojovat kód s jejich konkrétními třídami.

POUŽITÍ

- Použijte vzor Chain of Responsibility, pokud se od vašeho programu očekává, že bude zpracovávat různé typy požadavků různými způsoby, ale přesné typy požadavků a jejich pořadí nejsou předem známy.
- Vzor vám umožňuje propojit několik handlerů do jednoho řetězce a při obdržení požadavku „se zeptat“ každého handleru, zda jej dokáže zpracovat.
- Tímto způsobem dostanou všichni handlery možnost požadavek zpracovat.
- Použijte tento vzor, pokud je nezbytné provést několik handlerů v konkrétním pořadí.
- Protože můžete handlery v řetězci propojit v libovolném pořadí, všechny požadavky projdou řetězcem přesně podle vašeho plánu.
- Použijte vzor CoR, pokud se množina handlerů a jejich pořadí mají měnit za běhu programu.
- Pokud poskytnete settery pro referenční pole uvnitř tříd handlerů, budete moci handlery dynamicky vkládat, odstraňovat nebo měnit jejich pořadí.

JAK IMPLEMENTOVAT

- Deklarujte **interface handleru** a popište **signaturu metody pro zpracování požadavků**.
- Rozhodněte se, **jak klient předá data požadavku metodě**. Nejsnadnější a nejflexibilnější způsob je **převést požadavek na objekt** a předat jej metodě jako argument.
- Aby se odstranil **duplicitní boilerplate kód** v konkrétních handlerech, může být vhodné **vytvořit abstraktní základní třídu handlu**, odvozenou od interface handleru.
- Tato třída by měla obsahovat **pole pro uložení reference na další handler v řetězci**. Zvažte, zda třídu učinit **neměnnou (immutable)**. Pokud však plánujete **měnit řetězce za běhu programu**, musíte definovat **setter**, který umožní změnu hodnoty tohoto referenčního pole.
- Můžete také **implementovat výchozí chování** pro metodu zpracování požadavku – například **předat požadavek dalšímu objektu**, pokud nějaký existuje. Konkrétní handlery mohou toto chování využít **voláním metody rodiče**.
- Postupně vytvořte **konkrétní podtřídy handlerů** a implementujte jejich metody pro zpracování požadavků. Každý handler by měl při obdržení požadavku učinit **dvě rozhodnutí**:
 - Zda požadavek **zpracuje**.
 - Zda požadavek **předá dál řetězcem**.
- Klient může **řetězce sestavit sám**, nebo **přjmout předpřipravené řetězce od jiných objektů**. V druhém případě je nutné implementovat **některé tovární třídy**, které sestaví řetězce podle konfigurace nebo nastavení prostředí.
- Klient může **spustit libovolný handler v řetězci**, nejen první. Požadavek bude **putovat řetězcem**, dokud **některý handler odmítne předat jej dál**, nebo dokud **nedojde na konec řetězce**.
- Vzhledem k **dynamické povaze řetězce** by měl být klient připraven na následující scénáře:
- Řetězec může sestávat z **jediného členu**.
- Některé požadavky nemusí **dosáhnout konce řetězce**.
- Jiné požadavky mohou **dosáhnout konce řetězce nezpracované**.

POROVNÁNÍ

Přínosy

- Můžete řídit pořadí zpracování požadavků.
- Princip jediné odpovědnosti (Single Responsibility Principle) – můžete oddělit třídy, které vyvolávají operace, od tříd, které tyto operace provádějí.
- Princip otevřenosti/uzavřenosti (Open/Closed Principle) – můžete do aplikace zavádět nové handlery, aniž byste porušili stávající kód klienta.

Zápory

- Některé požadavky mohou zůstat nezpracované.

VZTAHY S JINÝMI VZORY

- Chain of Responsibility, Command, Mediator a Observer se zabývají různými způsoby propojení odesílatelů a příjemcu 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.
- Řetězec zodpovědnosti se často používá ve spojení s Composite. V takovém případě, když listová komponenta obdrží požadavek, může jej předat řetězcem všech rodičovských komponent až k kořeni objektového stromu.
- Handlery v Chain of Responsibility lze implementovat jako Commandy. V takovém případě můžete provádět různé operace nad stejným kontextovým objektem, reprezentovaným požadavkem.
- Existuje však i jiný přístup, kde požadavek sám je objektem Command. V tomto případě můžete provádět stejnou operaci v řadě různých kontextů, propojených do řetězce.
- Chain of Responsibility a Decorator mají velmi podobnou strukturu tříd. Oba vzory spoléhají na rekurzivní kompozici pro předávání vykonávání přes řadu objektů. Nicméně existuje několik zásadních rozdílů:
 - Handlery v CoR mohou nezávisle provádět libovolné operace. Také mohou kdykoliv zastavit předávání požadavku dál.
 - Naproti tomu různé dekorátory mohou rozšiřovat chování objektu, aniž by porušily jeho základní rozhraní. Navíc dekorátory nesmějí přerušit tok požadavku.

PŘÍKLAD

STRATEGY

- Strategie je behaviorální návrhový vzor, který Vám umožňuje definovat rodinu algoritmů, umístit každý z nich do samostatné třídy a učinit jejich objekty vzájemně zaměnitelnými.

K ČEMU JE DOBRÝ?

- Jednoho dne jste se rozhodli vytvořit navigační aplikaci pro příležitostné cestovatele. Aplikace byla postavena kolem krásné mapy, která uživatelům pomáhala rychle se zorientovat v jakémkoli městě.
- Jednou z nejžádanějších funkcí aplikace bylo automatické plánování tras.
- Uživatel by měl být schopen zadat adresu a na mapě vidět nejrychlejší trasu do cíle.
- První verze aplikace uměla vytvářet trasy pouze po silnicích. Lidé, kteří cestovali autem, byli nadšení. Ale zjevně ne každý rád řídí během dovolené.
- Proto jste v další aktualizaci přidali možnost vytvářet trasy pro pěší. Hned poté jste přidali další možnost – zahrnout do tras veřejnou dopravu.
- To však byl pouze začátek. Později jste plánovali přidat i vytváření tras pro cyklisty. A ještě později další možnost – plánování tras, které povedou přes všechna turisticky zajímavá místa ve městě.

- Z obchodního hlediska byla aplikace úspěchem, ale po technické stránce vám způsobila mnoho starostí. Pokaždé, když jste přidali nový algoritmus pro plánování trasy, hlavní třída navigátoru se zdvojnásobila ve velikosti.
- V určitém okamžiku se tento „monstrózní“ kód stal příliš obtížným na údržbu.
- Jakákoli změna v některém z algoritmů – ať už šlo o jednoduchou opravu chyby nebo drobnou úpravu hodnocení ulic – ovlivnila celou třídu, čímž se zvýšilo riziko, že vzniknou chyby v již funkčním kódu.
- Navíc se práce v týmu stala neefektivní. Vaši kolegové, kteří byli přijati po úspěšném vydání aplikace, si stěžovali, že tráví příliš mnoho času řešením konfliktů při slučování kódu (merge conflicts).
- Implementace nové funkce totiž vyžadovala úpravu té samé obrovské třídy, což vedlo ke konfliktům s kódem, na kterém pracovali ostatní.

ŘEŠENÍ

- Vzor Strategy navrhuje, abyste vzali třídu, která něco dělá různými způsoby, a všechny tyto algoritmy extrahovali do samostatných tříd, nazývaných strategie.
- Původní třída, nazývaná context, musí mít pole (list) pro uchování reference na jednu ze strategií. Context **deleguje** práci na připojený objekt strategie místo toho, aby ji vykonával sám.
- Context není zodpovědný za výběr vhodného algoritmu pro daný úkol.
- Místo toho klient předává požadovanou strategii do contextu.
- Ve skutečnosti context o strategiích neví mnoho.
- Pracuje se všemi strategiemi přes stejné generické rozhraní, které pouze vystavuje jedinou metodu pro spuštění algoritmu zabaleného ve vybrané strategii.

- Tím se context stává nezávislým na konkrétních strategiích, takže můžeme přidávat nové algoritmy nebo upravovat stávající, aniž byste měnili kód kontextu nebo jiných strategií.
- V naší navigační aplikaci lze každý algoritmus pro plánování tras extrahovat do samostatné třídy s jedinou metodou buildRoute.
- Tato metoda přijímá počáteční bod a cíl a vrací kolekci kontrolních bodů trasy.
- I když každý routingový algoritmus může při stejných argumentech vytvořit odlišnou trasu, hlavní třída navigátoru nereší, který algoritmus je vybrán, protože jejím hlavním úkolem je vykreslit sadu kontrolních bodů na mapě. Třída má také metodu pro přepínání aktivní strategie routingu, takže její klienti, například tlačítka v uživatelském rozhraní, mohou nahradit aktuálně vybranou strategii plánování tras jinou.

VYUŽITÍ

- Použijte vzor Strategy, pokud chcete v rámci objektu používat různé varianty algoritmu a mít možnost přepínat mezi algoritmy během běhu programu.
- Vzor Strategy Vám umožnuje nepřímo měnit chování objektu za běhu tím, že ho spojíte s různými podobjekty, které mohou provádět specifické úkoly různými způsoby.
- Použijte Strategy, pokud máte mnoho podobných tříd, které se liší pouze způsobem, jakým provádějí určité chování.
- Vzor Strategy umožňuje extrahovat proměnlivé chování do samostatné hierarchie tříd a původní třídy zkombinovat do jedné, čímž se snižuje duplikace kódu.
- Použijte tento vzor, abyste oddělili obchodní logiku třídy od implementačních detailů algoritmů, které nemusí být v kontextu té logiky také důležité.
- Vzor Strategy umožňuje izolovat kód, interní data a závislosti různých algoritmů od zbytku kódu. Různí klienti získávají jednoduché rozhraní pro spuštění algoritmů a jejich přepínání za běhu.
- Použijte vzor, pokud Vaše třída obsahuje rozsáhlý podmíněný příkaz (if/else nebo switch), který přepíná mezi různými variantami stejného algoritmu.
- Vzor Strategy Vám umožní odstranit takový podmíněný příkaz tím, že všechny algoritmy extrahuje do samostatných tříd, které všechny implementují stejné rozhraní. Původní objekt pak deleguje provedení jednomu z těchto objektů, místo aby implementoval všechny varianty algoritmu.

JAK IMPLEMENTOVAT

1. V třídě context identifikujte algoritmus, který je náchylný k častým změnám. Může to být také rozsáhlý podmíněný příkaz, který za běhu programu vybírá a vykonává variantu stejného algoritmu.
2. Deklarujte strategy interface, které bude společné pro všechny varianty algoritmu.
3. Postupně extrahujte všechny algoritmy do samostatných tříd. Všechny by měly implementovat strategy interface.
4. V třídě context přidejte pole pro uchování reference na objekt strategie. Zajistěte setter pro nahrazení hodnoty tohoto pole. Context by měl s objektem strategie pracovat pouze přes strategy interface. Context může definovat rozhraní, které umožní strategii přístup k jeho datům.
5. Klienti contextu musí přiřadit vhodnou strategii, která odpovídá způsobu, jak očekávají, že context bude vykonávat svou primární funkci.

PRO A PROTI

Pro

- Můžete za běhu programu měnit algoritmy, které objekt používá.
- Můžete izolovat implementační detaily algoritmu od kódu, který jej používá.
- Můžete nahradit dědičnost kompozicí.
- Princip otevřenosti/uzavřenosti (Open/Closed Principle) – můžete zavádět nové strategie, aniž byste museli měnit kód kontextu.

Proti

- Pokud máte jen několik algoritmů a ty se málo mění, není skutečný důvod program zbytečně komplikovat novými třídami a rozhraními, která vzor přináší.
- Klienti musí být informováni o rozdílech mezi strategiemi, aby byli schopni vybrat tu správnou.
- Mnoho moderních programovacích jazyků podporuje funkcionální typy, které umožňují implementovat různé verze algoritmu v sadě anonymních funkcí. Tyto funkce pak můžete používat stejně, jako byste používali objekty strategií, ale bez toho, abyste kód zatěžovali dalšími třídami a rozhraními.

VZTAHY S DALŠÍMI VZORY

- Bridge, State, Strategy (a do určité míry i Adapter) mají velmi podobnou strukturu. Všechny tyto vzory jsou založeny na kompozici, tedy na delegování práce jiným objektům. Přesto řeší různé problémy. Vzor není jen recept na strukturování kódu určitým způsobem; může také komunikovat ostatním vývojářům, jaký problém tento vzor řeší.
- Command a Strategy mohou vypadat podobně, protože oba umožňují parametrizovat objekt nějakou akcí. Nicméně mají velmi odlišné cíle.
 - Command slouží k tomu, aby převedl libovolnou operaci na objekt. Parametry operace se stávají polí tohoto objektu. Tato konverze umožňuje odložit provedení operace, zařadit ji do fronty, ukládat historii příkazů, posílat příkazy na vzdálené služby atd.
 - Naopak Strategy obvykle popisuje různé způsoby, jak dělat stejnou věc, což umožňuje vyměňovat algoritmy uvnitř jedné třídy contextu.
- Decorator umožňuje změnit „vzhled“ objektu, zatímco Strategy umožňuje změnit jeho „vnitřní chování“.
- Template Method je založena na dědičnosti: umožňuje měnit části algoritmu jejich rozšířením v podtřídách. Strategy je založena na kompozici: umožňuje měnit části chování objektu tím, že mu poskytnete různé strategie, které odpovídají tomuto chování. Template Method funguje na úrovni třídy, je tedy statická. Strategy funguje na úrovni objektu, což umožňuje přepínat chování za běhu programu.
- State lze považovat za rozšíření Strategy. Oba vzory jsou založeny na kompozici: mění chování contextu tím, že část práce delegují pomocným objektům. Strategy činí tyto objekty zcela nezávislé a nevědomé o sobě navzájem, zatímco State neomezuje závislosti mezi konkrétními stavami, což jim umožňuje měnit stav contextu podle libosti.

PŘÍKLAD

NÁVRHOVÉ VZORY

Přednáška 7

BEHAVIORAL PATTERNS

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

ITERATOR

- Iterator je behaviorální návrhový vzor, který Vám umožňuje procházet prvky kolekce, aniž byste odhalovali její vnitřní reprezentaci (např. seznam, zásobník, strom atd.).

K ČEMU SE HODÍ?

- Kolekce patří mezi nejčastěji používané datové typy v programování.
- Přesto je kolekce jen kontejner pro skupinu objektů.
- Většina kolekcí uchovává své prvky v jednoduchých seznamech.
- Některé jsou však založeny na zásobnících, stromech, grafech a dalších složitějších datových strukturách.
- Bez ohledu na to, jak je kolekce strukturovaná, musí poskytovat způsob přístupu k jejím prvkům, aby je mohl využívat jiný kód.
- Měl by existovat způsob, jak projít každý prvek kolekce, aniž byste opakovaně přistupovali ke stejným prvkům.

- To může znít jako jednoduchý úkol, pokud máte kolekci založenou na seznamu – stačí projít všechny prvky cyklem.
- Ale jak sekvenčně procházet prvky složité datové struktury, například stromu?
- Například jednoho dne vám může stačit procházení stromu do hloubky (depth-first traversal).
- Další den můžete potřebovat procházení do šířky (breadth-first traversal).
- A další týden třeba náhodný přístup k prvkům stromu.
- Přidávání stále více algoritmů pro procházení do kolekce postupně rozmažává její primární odpovědnost, kterou je efektivní ukládání dat.
- Navíc některé algoritmy mohou být přizpůsobeny specifické aplikaci, takže jejich začlenění do generické třídy kolekce by bylo neobvyklé.
- Na druhou stranu klientský kód, který má pracovat s různými kolekcemi, se nemusí vůbec zajímat o to, jak kolekce uchovávají své prvky.
- Protože však různé kolekce poskytují různé způsoby přístupu k prvkům, nemáte jinou možnost, než svázat svůj kód s konkrétními třídami kolekci.

ŘEŠENÍ

- Hlavní myšlenkou vzoru Iterátor je extrahovat chování procházení kolekce do samostatného objektu, nazývaného iterator.
- Kromě implementace samotného algoritmu objekt iterator zapouzdřuje všechny detaily procházení, například aktuální pozici a počet zbývajících prvků do konce.
- Díky tomu může několik iteratorů současně procházet stejnou kolekcí, nezávisle na sobě.
- Obvykle iterátory poskytují jednu hlavní metodu pro získávání prvků kolekce. Klient může tuto metodu volat opakovaně, dokud nevrátí žádný prvek, což znamená, že iterator prošel všechny prvky.
- Všechny iterátory musí implementovat stejné rozhraní.
- To činí klientský kód kompatibilní s libovolným typem kolekce nebo algoritmem procházení, pokud je k dispozici vhodný iterator.
- Pokud potřebujete speciální způsob procházení kolekce, stačí vytvořit novou třídu iteratoru, aniž byste museli měnit kolekci nebo klientský kód.

VYUŽITÍ

- Použijte vzor Iterator, pokud vaše kolekce obsahuje složitou datovou strukturu, ale chcete skrýt její složitost před klienty (ať už z důvodu pohodlí nebo bezpečnosti).
- Iterator zapouzdřuje detaily práce se složitou datovou strukturou a klientovi poskytuje několik jednoduchých metod pro přístup k prvkům kolekce. Tento přístup je velmi pohodlný pro klienta a zároveň chrání kolekci před neopatrnými nebo škodlivými akcemi, které by klient mohl provést při přímé práci s kolekcí.
- Vzor použijte také k snížení duplikace kódu procházení napříč vaší aplikací.
- Kód pro ne-triviální algoritmy procházení bývá často velmi rozsáhlý. Pokud je umístěn přímo v obchodní logice aplikace, může rozmažat odpovědnost původního kódu a snížit jeho udržovatelnost. Přesunutím kódu procházení do určených iteratorů můžete učinit kód aplikace čistším a přehlednějším.
- Použijte vzor Iterator, pokud chcete, aby váš kód procházel různé datové struktury, nebo pokud jsou typy těchto struktur předem neznámé.
- Vzor poskytuje několik generických rozhraní pro kolekce i iterátory. Pokud váš kód tato rozhraní používá, bude fungovat i s různými typy kolekcí a iterátorů, které tato rozhraní implementují.

JAK IMPLEMENTOVAT?

1. Deklarujte rozhraní iterátoru. Minimálně by mělo obsahovat metodu pro získání dalšího prvku kolekce. Pro větší pohodlí můžete přidat i několik dalších metod, například získání předchozího prvku, sledování aktuální pozice nebo kontrolu konce iterace.
2. Deklarujte rozhraní kolekce a definujte metodu pro získání iterátoru. Návratový typ by měl odpovídat typu iterátoru. Podobné metody můžete deklarovat i v případě, že plánujete mít několik odlišných skupin iterátorů.
3. Implementujte konkrétní třídy iterátorů pro kolekce, které chcete zpřístupnit procházení pomocí iterátorů. Objekt iterátoru musí být propojen s jedinou instancí kolekce. Obvykle se toto propojení zajišťuje přes konstruktor iterátoru.
4. Implementujte rozhraní kolekce ve svých třídách kolekcí. Hlavní myšlenkou je poskytnout klientovi zkratku pro vytvoření iterátoru, přizpůsobeného konkrétní třídě kolekce. Objekt kolekce musí předat sám sebe konstruktoru iterátoru, aby bylo propojení mezi nimi zajištěno.
5. Projděte klientský kód a nahraďte všechny části kódu procházení kolekce použitím iterátorů. Klient získává nový objekt iterátoru pokaždé, když potřebuje projít prvky kolekce.

PRO A PROTI

Pro

- Princip jediné odpovědnosti (Single Responsibility Principle): Můžete učinit klientský kód a kolekce přehlednějšími tím, že rozsáhlé algoritmy procházení extrahujete do samostatných tříd.
- Princip otevřenosti/uzavřenosti (Open/Closed Principle): Můžete implementovat nové typy kolekcí a iteratorů a předávat je stávajícímu kódu, aniž byste cokoli porušili.
- Můžete procházet stejnou kolekcí paralelně, protože každý objekt iteratoru obsahuje svůj vlastní stav iterace.
- Ze stejného důvodu můžete iteraci odložit a pokračovat v ní později, kdy je potřeba.

Proti

- Použití vzoru může být přehnané, pokud vaše aplikace pracuje pouze s jednoduchými kolekcemi.
- Použití iteratoru může být méně efektivní, než přímo procházet prvky některých specializovaných kolekcí.

VZTAHY S DALŠÍMI VZORY

- Iterator můžete použít pro procházení stromů Composite.
- Můžete použít Factory Method společně s iterátorem, aby podtřídy kolekcí vracely různé typy iteratorů, které jsou kompatibilní s kolekcemi.
- Můžete použít Memento spolu s iterátorem k zachycení aktuálního stavu iterace a jeho případnému vrácení zpět, pokud to bude potřeba.
- Můžete použít Visitor společně s iterátorem k procházení složité datové struktury a provádění nějaké operace nad jejími prvky, i když všechny mají různé třídy.

PŘÍKLAD

COMMAND

- Command (Příkaz) je behaviorální návrhový vzor, který převádí požadavek (request) na samostatný objekt, jenž obsahuje všechny potřebné informace o daném požadavku.
- Tato transformace umožňuje:
 - předávat požadavky jako argumenty metod,
 - odkládat nebo řadit jejich vykonání do fronty,
 - a také podporovat operace, které lze vrátit zpět (undo).

CO ŘEŠÍ?

- Představte si, že pracujete na nové aplikaci textového editoru. Vaším aktuálním úkolem je vytvořit panel nástrojů s řadou tlačítek pro různé operace editoru. Vytvořili jste elegantní třídu `Button`, kterou lze použít jak pro tlačítka na panelu nástrojů, tak i pro běžná tlačítka v různých dialogových oknech.
- Ačkoli všechna tato tlačítka vypadají podobně, mají dělat různé věci. Kam ale umístit kód, který se má vykonat po kliknutí na jednotlivá tlačítka?
- Nejjednodušším řešením by bylo vytvořit mnoho podtříd — pro každé použití tlačítka jednu. Tyto podtřidy by obsahovaly kód, který se má spustit při kliknutí.
- Brzy si ale uvědomíte, že tento přístup je velmi problematický:
 - Máte obrovské množství podtříd.
 - Každá změna v základní třídě `Button` může snadno rozbít kód v těchto podtřídách.
 - Jinými slovy, váš GUI kód se stává závislým na proměnlivém kódu business logiky.

- A teď to nejhorší: některé operace (např. kopírování/vkládání textu) musí být vyvolány z více míst.
- Uživatel může například: kliknout na malé tlačítko „Kopírovat“ na panelu nástrojů, použít kontextové menu,nebo stisknout Ctrl+C na klávesnici.
- Zpočátku, když aplikace obsahovala pouze panel nástrojů, bylo v pořádku umístit implementaci jednotlivých operací přímo do podtříd tlačítek — například kód pro kopírování textu do třídy CopyButton.
- Jakmile však přidáte kontextová menu, klávesové zkratky a další ovládací prvky, musíte buď duplikovat kód operace v mnoha třídách, nebo — což je ještě horší — učinit menu závislými na tlačítkách.

ŘEŠENÍ

- Dobrý návrh softwaru je často založen na principu oddělení odpovědností (separation of concerns), což obvykle vede k rozdělení aplikace do vrstev.
- Nejčastějším příkladem je rozdělení na:
 - vrstvu grafického uživatelského rozhraní (GUI)
 - vrstvu obchodní logiky.
- Vrstva GUI je zodpovědná za zobrazení přehledného a atraktivního rozhraní, zachytávání vstupu uživatele a zobrazování výsledků toho, co uživatel a aplikace dělají.
- Když však dojde na něco důležitého — například výpočet dráhy Měsíce nebo vytváření roční zprávy —, GUI vrstva předává práci podkladové vrstvě business logiky.
- V kódu to může vypadat takto: objekt GUI volá metodu objektu business logiky a předává jí argumenty. Tento proces se obvykle popisuje jako odeslání požadavku z jednoho objektu na druhý.

- Návrhový vzor Command navrhoje, aby objekty GUI neodesílaly tyto požadavky přímo. Místo toho byste měli vytáhnout všechny detaily požadavku — například objekt, na který je volána metoda, název metody a seznam argumentů — do samostatné třídy příkazu (command) s jedinou metodou, která tento požadavek spustí.
- Objekty Command slouží jako spojovací článek mezi různými objekty GUI a objekty obchodní logiky.
- Od této chvíle objekt GUI nemusí vědět, který objekt obchodní logiky požadavek obdrží a jak bude zpracován.
- Objekty GUI jednoduše spustí příkaz, který se postará o všechny detaily.

- Vraťme se k našemu textovému editoru. Po použití návrhového vzoru Command již nepotřebujeme všechny ty podtřídy tlačítka, aby implementovaly různé chování po kliknutí.
- Stačí vložit jedno pole do základní třídy Button, které bude uchovávat odkaz na objekt příkazu (Command), a nechat tlačítko tento příkaz vykonat při kliknutí.
- Pro každou možnou operaci implementujete sadu tříd příkazů a přiřadíte je konkrétním tlačítkům podle toho, jaké chování mají tlačítka vykonávat.
- Ostatní prvky GUI, jako jsou menu, klávesové zkratky nebo celé dialogy, mohou být implementovány stejným způsobem. Budou propojeny s příkazem, který se spustí, když uživatel s prvkem GUI interaguje. Jak jste si pravděpodobně už všimli, prvky související se stejnou operací budou propojeny se stejnými příkazy, což zabraňuje duplicitě kódu.
- Výsledkem je, že příkazy tvoří praktickou střední vrstvu, která snižuje závislost mezi vrstvou GUI a vrstvou business logiky. A to je jen malá část výhod, které návrhový vzor Command nabízí!

POUŽITÍ

- Použijte návrhový vzor Command, pokud chcete parametrizovat objekty operacemi.
- Návrhový vzor Command může převést konkrétní volání metody na samostatný objekt. Tato změna otevírá řadu zajímavých možností: můžete příkazy předávat jako argumenty metod, ukládat je do jiných objektů, měnit propojené příkazy za běhu aplikace atd. Příklad: vyvíjíte komponentu GUI, například kontextové menu, a chcete, aby uživatelé mohli konfigurovat položky menu, které spustí operace, když uživatel klikne na položku.
- Použijte vzor Command, pokud chcete řadit operace do fronty, plánovat jejich vykonání nebo je provádět vzdáleně. Stejně jako jakýkoli jiný objekt, může být příkaz serializován, což znamená, že lze převést na řetězec, který je snadno zapisovatelný do souboru nebo databáze. Později lze tento řetězec obnovit jako původní objekt příkazu. Tímto způsobem můžete odložit nebo naplánovat vykonání příkazu. A to není vše! Stejným způsobem můžete příkazy řadit do fronty, logovat nebo odesílat po síti.

- Použijte vzor Command, pokud chcete implementovat vratné operace (undo/redo). Ačkoli existuje mnoho způsobů, jak implementovat undo/redo, návrhový vzor Command je pravděpodobně nejpopulárnější.
- Pro možnost vrácení operací je potřeba implementovat historii provedených příkazů. Historie příkazů je zásobník obsahující všechny provedené příkazy spolu s příslušnými zálohami stavu aplikace.
- Tato metoda má dvě nevýhody: Uložení stavu aplikace není snadné, protože část stavu může být soukromá. Tento problém lze zmírnit použitím vzoru Memento. Zálohy stavu mohou spotřebovat hodně paměti RAM. Proto někdy můžete použít alternativní přístup: místo obnovy minulého stavu příkaz provede opačnou operaci. Tato reverzní operace má také svou cenu — může být obtížná nebo dokonce nemožná k implementaci.

JAK IMPLEMENTOVAT?

- Deklarujte rozhraní příkazu (Command interface) s jedinou metodou pro vykonání.
- Začněte extrahovat požadavky do konkrétních tříd příkazů, které implementují toto rozhraní. Každá třída musí obsahovat sadu polí pro uchování argumentů požadavku spolu s odkazem na skutečný objekt příjemce (receiver). Všechny tyto hodnoty musí být inicializovány přes konstruktor příkazu.
- Identifikujte třídy, které budou působit jako odesílatelé (senders). Přidejte do těchto tříd pole pro uchování příkazů. Odesílatelé by měli komunikovat se svými příkazy pouze přes rozhraní příkazu.
- Odesílatelé obvykle nevytvářejí objekty příkazů sami, ale dostávají je od klientského kódu. Změňte odesílatele tak, aby vykonávali příkaz, místo aby odesílali požadavek přímo příjemci.
- Klient by měl inicializovat objekty v následujícím pořadí:
 - Vytvořit příjemce (receivers).
 - Vytvořit příkazy a přiřadit je příjemcům, pokud je to potřeba.
 - Vytvořit odesílatele (senders) a přiřadit jím konkrétní příkazy.

PROROVNÁNÍ

Pro

- Princip jediné odpovědnosti (Single Responsibility Principle): Můžete oddělit třídy, které volají operace, od tříd, které tyto operace provádějí.
- Princip otevřenosti/uzavřenosti (Open/Closed Principle): Můžete přidávat nové příkazy do aplikace, aniž byste porušili existující klientský kód.
- Můžete implementovat undo/redo.
- Můžete implementovat odložené vykonání operací.
- Můžete složit sadu jednoduchých příkazů do jednoho komplexního příkazu.

Proti

- Kód se může stát složitějším, protože zavádíte zcela novou vrstvu mezi odesílateli a příjemci.

VZTAH S DALŠÍMI VZORY

- **Chain of Responsibility, Command, Mediator a Observer** řeší různé způsoby propojení odesílatelů a příjemců požadavků:
- **Chain of Responsibility** předává požadavek sekvenčně podél dynamického řetězce potenciálních příjemců, dokud jej některý z nich nezpracuje.
- **Command** vytváří **jednosměrná propojení** mezi odesílateli a příjemci.
- **Mediator** eliminuje přímá propojení mezi odesílateli a příjemci, nutí je komunikovat **nepřímo přes objekt mediátora**.
- **Observer** umožňuje příjemcům **dynamicky se přihlašovat a odhlašovat** k přijímání požadavků.
- **Handlery v Chain of Responsibility** lze implementovat jako **Commandy**. V takovém případě můžete nad stejným kontextovým objektem (reprezentovaným požadavkem) vykonat mnoho různých operací.
- Existuje yšak i jiný přístup, kdy je **požadavek sám o sobě objektem Command**. V tom případě můžete **vykonat stejnou operaci** v řadě různých kontextů propojených do řetězce.
- **Command a Memento** lze použít společně při implementaci „**undo**“. V takovém případě jsou příkazy zodpovědné za provedení různých operací nad cílovým objektem, zatímco memento ukládají stav tohoto objektu těsně před vykonáním příkazu.
- **Command a Strategy** mohou vypadat podobně, protože oba umožňují **parametrizovat objekt nějakou akcí**. Mají však velmi odlišný záměr:
- **Command** převádí libovolnou operaci na objekt. Parametry operace se stávají poli objektu, což umožňuje odložené vykonání, řazení do fronty, uchovávání historie příkazů, odesílání příkazů na vzdálené služby atd.
- **Strategy** obvykle popisuje různé způsoby provedení téže činnosti a umožňuje **vyměňovat algoritmy** uvnitř jedné kontextové třídy.
- **Prototype** se hodí, pokud potřebujete ukládat **kopie příkazů do historie**.
- **Visitor** lze považovat za **rozšířenou verzi vzoru Command**. Jeho objekty mohou vykonávat operace nad různými objekty různých tříd.

PŘÍKLAD

MEMENTO

- Memento je návrhový vzor, který umožňuje uložit a obnovit předchozí stav objektu, aniž by odhalil detaily jeho vnitřní implementace.

K ČEMU SE HODÍ?

- Představte si, že vytváříte aplikaci textového editoru. Kromě jednoduchého úprav textu může váš editor text také formátovat, vkládat obrázky apod.
- V určitém okamžiku se rozhodnete, že uživatelům umožníte vracet zpět jakékoli provedené operace. Tato funkce se během let stala natolik běžnou, že dnes už ji lidé očekávají v každé aplikaci.
- Pro implementaci zvolíte přímý přístup: před provedením jakékoli operace aplikace uloží stav všech objektů do nějakého úložiště.
- Později, když se uživatel rozhodne akci vrátit zpět, aplikace načte poslední snímek (snapshot) z historie a použije jej k obnovení stavu všech objektů.

- Zamysleme se nad těmito snímky stavů. Jak byste takový snímek vlastně vytvořili? Pravděpodobně by bylo potřeba projít všechna pole objektu a jejich hodnoty zkopirovat do úložiště. To by ovšem fungovalo jen v případě, že by objekt umožňoval volný přístup ke svým datům. Bohužel většina skutečných objektů to neumožňuje, protože ukrývá svá důležitá data v soukromých (private) polích.
- Ignorujme tento problém a předpokládejme, že naše objekty se chovají „jako hippies“ — mají otevřené vztahy a udržují svůj stav veřejný. Tento přístup by sice okamžitý problém vyřešil a umožnil vytvářet snímky stavů dle libosti, ale přináší vážné nevýhody.
- V budoucnu se totiž můžete rozhodnout refaktorovat některé třídy editoru, nebo přidat či odebrat některá pole. To by sice znělo jednoduše, ale zároveň by bylo nutné upravit i všechny třídy, které jsou zodpovědné za kopírování stavů dotčených objektů.

- Ale to není všechno. Zamysleme se nad samotnými „**snímky stavu editoru**“. Jaká data vlastně obsahují? V tom nejzákladnějším případě by měly obsahovat **aktuální text, pozici kurzoru, aktuální posun (scroll)**, a podobné údaje.
- Aby bylo možné takový snímek vytvořit, bylo by potřeba tyto hodnoty **shromáždit a uložit** do nějakého kontejneru.
- S největší pravděpodobností budete chtít tyto kontejnery ukládat do nějakého **seznamu (historie)**, který bude představovat historii změn. Tyto kontejnery by tedy pravděpodobně byly **objekty jedné třídy**. Tato třída by měla jen minimum metod, ale **spoustu polí odpovídajících stavu editoru**.
- Aby ostatní objekty mohly data do snímku zapisovat nebo je z něj číst, musela by být taž pole **veřejná (public)**. Tím by se ale **odhalil celý vnitřní stav editoru**, včetně soukromých částí. Ostatní třídy by se tak staly **závislé na každé malé změně** ve třídě snímku — i na těch, které by se jinak děly jen uvnitř editoru bez vlivu na ostatní části programu.
- Vypadá to, že jsme se dostali do slepé uličky:
 - buď **odhalíme všechny vnitřní detaily tříd** a učiníme je tak velmi křehkými,
 - nebo **omezíme přístup k jejich stavu**, čímž znemožníme vytváření snímků.
- Existuje tedy vůbec nějaký jiný způsob, jak implementovat funkci „**Zpět**“ (**undo**)?

ŘEŠENÍ

- Všechny problémy, které jsme právě popsali, jsou způsobeny **porušením zapouzdření (encapsulation)**. Některé objekty se snaží dělat víc, než by měly. Aby získaly potřebná data pro vykonání určité akce, **vstupují do soukromého prostoru jiných objektů**, místo aby nechaly tyto objekty provést akci samy.
- **Návrhový vzor Memento** svěřuje vytváření snímků stavu **samotnému vlastníkovi tohoto stavu** – tedy tzv. originatoru (původnímu objektu). Místo toho, aby se jiné objekty snažily kopírovat stav editoru „zvenčí“, může **třída editoru** sama vytvořit snímek, protože má **plný přístup ke svému vlastnímu stavu**.
- Vzor navrhuje ukládat kopii stavu objektu do **speciálního objektu zvaného memento**. Obsah tohoto mementa **není přístupný žádnému jinému objektu** kromě toho, který ho vytvořil. Ostatní objekty mohou s mementy komunikovat pouze přes **omezené rozhraní**, které může například umožnit získání **metadat snímku** (čas vytvoření, název provedené operace apod.), ale **ne samotný uložený stav původního objektu**.

- Tako přísná pravidla umožňují **ukládat mamenta do jiných objektů**, které se obvykle nazývají **správci (caretakers)**. Protože správce pracuje s mementem **pouze prostřednictvím omezeného rozhraní, nemůže nijak manipulovat se stavem**, který je uvnitř mamenta uložen.
Zároveň však **originátor** má přístup ke všem polím uvnitř mamenta, což mu umožňuje **obnovit svůj předchozí stav kdykoli potřebuje**.
- V našem příkladu s textovým editorem můžeme vytvořit **samostatnou třídu pro historii**, která bude působit jako správce. **Zásobník mement** uložený ve správci se bude rozrůstat pokaždé, když se editor chystá provést nějakou operaci. Tento zásobník lze dokonce **zobrazit v uživatelském rozhraní aplikace**, aby uživatel viděl **historii provedených akcí**.
- Když uživatel spustí „**Undo**“ (**zpět**), historie vezme **nejnovější memento** ze zásobníku a předá ho zpět editoru s požadavkem na **vrácení změn**. Protože má editor **plný přístup k mementu**, může **obnovit svůj stav** podle hodnot uložených v tomto mementu.

VYUŽITÍ

- Použijte **návrhový vzor Memento**, když chcete vytvářet **snímky stavu objektu**, abyste mohli **obnovit jeho předchozí stav**.
- Vzor Memento umožňuje **vytvořit úplné kopie stavu objektu**, včetně **soukromých polí**, a uložit je **odděleně od samotného objektu**. I když si většina lidí tento vzor pamatuje díky případu použití „undo“, je rovněž **nezbytný při práci s transakcemi** (např. pokud je potřeba v případě chyby operaci vrátit zpět).
- Použijte tento vzor, když **přímý přístup k polím/getterům/setterům objektu porušuje jeho zapouzdření**.
- Memento činí **objekt samotný zodpovědným za vytvoření snímku svého stavu**. Žádný jiný objekt **nemůže tento snímek čist**, což zajíšťuje, že data stavu původního objektu jsou **bezpečná a chráněná**.

JAK IMPLEMENTOVAT?

1. Určete, která třída bude hrát roli **originátora**. Je důležité vědět, zda program používá **jeden centrální objekt tohoto typu**, nebo **více menších objektů**.
2. Vytvořte třídu **Memento**. Postupně deklarujte **sadu polí**, která **odpovídají polím** deklarovaným uvnitř třídy originátora.
3. Udělejte třídu Memento **neměnnou (immutable)**. Memento by mělo přijmout data **jen jednou**, přes konstruktor. Třída **by neměla mít žádné settery**.
4. Pokud váš programovací jazyk podporuje **vnořené třídy**, vložte Memento dovnitř originátora. Pokud ne, **vytvořte prázdné rozhraní (interface)** z třídy Memento a nechte všechny ostatní objekty, aby na Memento odkazovaly přes toto rozhraní. Do rozhraní můžete přidat nějaké **operace s metadaty**, ale nic, co by **odhalovalo stav originátora**.
5. Přidejte do třídy originátora **metodu pro vytvoření mementa**. Originátor by měl předat svůj stav Mementu přes **jeden nebo více argumentů konstruktoru Mementa**.
Návratový typ metody by měl být typu rozhraní, které jste v předchozím kroku vytvořili (pokud jste ho vytvořili). Interně by metoda pro tvorbu mementa měla **pracovat přímo s třídou Memento**.
6. Přidejte do třídy originátora **metodu pro obnovení stavu originátora**. Měla by přijímat **objekt Memento** jako argument. Pokud jste v předchozím kroku vytvořili rozhraní, použijte ho jako typ parametru. V tomto případě bude třeba **přetykovat příchozí objekt na třídu Memento**, protože originátor potřebuje **plný přístup k objektu**.
7. Správce (caretaker), ať už představuje **objekt příkazu, historii nebo něco úplně jiného**, by měl vědět, **kdy si vyžádat nové memento od originátora, jak ho uložit a kdy obnovit originátora s konkrétním mementem**.
8. Vazba mezi **správci a originátory** může být přesunuta do třídy Memento. V tomto případě musí být každé memento **spojeno s originátorem**, který ho vytvořil. Metoda pro obnovu stavu by se také přesunula do třídy Memento. To však dává smysl **pouze pokud je třída Memento vložena do originátora** nebo pokud třída originátora poskytuje **dostatečné settery pro přepsání svého stavu**.

PRO A PROTI

Pro

- Můžete vytvářet **snímky stavu objektu**, aniž byste porušili jeho zapouzdření.
- Můžete **zjednodušit kód originátora** tím, že necháte správce (caretaker) spravovat historii stavů originátora.

Proti

- Aplikace může spotřebovat hodně paměti RAM, pokud klienti vytvářejí mementa příliš často.
- Správci (caretakers) by měli sledovat životní cyklus originátora, aby mohli zničit zastaralá mementa.
- Většina dynamických programovacích jazyků, jako PHP, Python a JavaScript, nemůže zaručit, že stav uvnitř memento zůstane nedotčený.

VZTAHY S JINÝMI VZORY

- Můžete použít **Command a Memento společně** při implementaci funkce „undo“. V tomto případě jsou příkazy zodpovědné za provádění různých operací nad cílovým objektem, zatímco mementa ukládají stav tohoto objektu těsně před vykonáním příkazu.
- Můžete použít **Memento spolu s Iterator** k zachycení aktuálního stavu iterace a jeho případnému obnovení.
- Někdy může být **Prototype jednodušší alternativou k Memento**. To funguje, pokud je objekt, jehož stav chcete uložit do historie, poměrně jednoduchý a nemá odkazy na externí zdroje, nebo jsou tyto odkazy snadno znova navázatelné.

PŘÍKLAD

NÁVRHOVÉ VZORY

Přednáška 8

BEHAVIORAL PATTERNS

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

MEDIATOR

- **Mediátor** je behaviorální (chování popisující) návrhový vzor, který umožňuje omezit chaotické závislosti mezi objekty.
- Tento vzor zakazuje přímou komunikaci mezi objekty a nutí je spolupracovat pouze prostřednictvím **objektu mediátoru**.

CO ŘEŠÍ?

- Představte si, že máte dialogové okno pro vytváření a úpravu zákaznických profilů. Skládá se z různých ovládacích prvků, jako jsou textová pole, zaškrťávací políčka, tlačítka atd.
- Některé z těchto prvků mohou mezi sebou interagovat.
- Například zaškrtnutí políčka „**Mám psa**“ může zobrazit skryté textové pole pro zadání jména psa.
Dalším příkladem je tlačítko **Odeslat**, které musí před uložením dat zkontolovat a ověřit hodnoty všech polí.
- Pokud by taž logika byla implementována přímo v kódu jednotlivých prvků formuláře, výrazně by to ztížilo jejich opětovné použití v jiných formulářích aplikace.
Například byste nemohli použít třídu daného zaškrťávacího políčka v jiném formuláři, protože by byla pevně svázána s textovým polem pro jméno psa.
- Buď byste museli použít všechny třídy, které se podílejí na vykreslení formuláře profilu, nebo vůbec žádnou.

ŘEŠENÍ

- Návrhový vzor **Mediator** navrhuje, abyste zastavili veškerou přímou komunikaci mezi komponentami, které chcete oddělit a učinit na sobě nezávislými.
- Místo toho by tyto komponenty měly spolupracovat **nepřímo** — voláním speciálního objektu zvaného **mediátor**, který přesměrovává volání na odpovídající komponenty.
- Díky tomu jsou komponenty závislé pouze na jedné třídě mediátoru, místo aby byly propojené s desítkami svých „kolegyň“.
- V našem příkladu s formulářem pro úpravu profilu může jako mediátor fungovat samotná třída **dialogového okna**.
- Tato třída už s největší pravděpodobností zná všechny své podřízené prvky, takže do ní nemusíme přidávat žádné nové závislosti.

- Největší změna nastává u samotných prvků formuláře.
- Vezměme si například tlačítko **Odeslat**. Dříve při každém kliknutí uživatelem muselo zkонтrolovat hodnoty všech jednotlivých prvků formuláře.
- Nyní má pouze **jediný úkol** — oznámit dialogovému oknu, že došlo ke kliknutí. Po obdržení této notifikace pak **dialog** sám provede validaci nebo úkol předá konkrétním prvkům formuláře.
- Tím pádem už tlačítko není závislé na desítkách jiných prvků, ale pouze na třídě dialogu.
- Závislost lze ještě více omezit tím, že vytvoříme **společné rozhraní** pro všechny typy dialogů.
- Toto rozhraní bude deklarovat metodu pro zasílání notifikací, kterou mohou všechny prvky formuláře používat k oznamování událostí. Díky tomu může naše tlačítko **Odeslat** spolupracovat s jakýmkoli dialogem, který toto rozhraní implementuje.
- Tímto způsobem vzor **Mediator** umožňuje zapouzdřit složitou síť vztahů mezi různými objekty do jednoho mediátoru.
Cím méně závislostí má třída, tím snadněji ji lze **upravit, rozšířit nebo znovu použít**.

VYUŽITÍ

- Použijte **vzor Mediator**, když je obtížné měnit některé třídy, protože jsou **silně provázány** s množstvím jiných tříd.
- Tento vzor umožňuje **vytáhnout všechny vztahy mezi třídami do samostatné třídy**, čímž se změny v jedné komponentě **izolují** od zbytku systému.
- Použijte tento vzor také tehdy, **když nelze komponentu znovu použít v jiném programu**, protože je příliš závislá na ostatních komponentách.
- Po aplikaci vzoru **Mediator** si jednotlivé komponenty **nejsou vědomy existence jiných komponent**. Stále však mohou mezi sebou komunikovat – i když **nepřímo**, prostřednictvím objektu mediátoru.
- Chcete-li komponentu znovu použít v jiné aplikaci, stačí jí poskytnout novou implementaci mediátoru.
- Vzor **Mediator** použijte také tehdy, když zjišťujete, že vytváříte velké množství **podtříd komponent** jen proto, abyste mohli znovu použít základní chování v různých kontextech.
- Protože všechny vztahy mezi komponentami jsou obsaženy v mediátoru, je snadné definovat **zcela nové způsoby spolupráce** těchto komponent – stačí vytvořit novou třídu mediátoru, **aníž by bylo nutné měnit samotné komponenty**.

JAK IMPLEMENTOVAT?

- Identifikujte skupinu **silně provázaných tříd**, které by mohly těžit z větší **nezávislosti** (např. kvůli jednodušší údržbě nebo snadnější znovupoužitelnosti těchto tříd).
- Deklarujte **rozhraní mediátoru** a popište požadovaný **komunikační protokol** mezi mediátorem a různými komponentami. Ve většině případů stačí jedna metoda pro příjem oznámení od komponent.
- Toto rozhraní je klíčové, pokud chcete třídy komponent znova použít v jiných kontextech. Dokud komponenta komunikuje se svým mediátorem přes **obecné rozhraní**, můžete ji propojit s jinou implementací mediátoru.
- Implementujte **konkrétní třídu mediátoru**. Zvažte uložení odkazů na všechny komponenty uvnitř mediátoru – díky tomu může mediátor **volat metody libovolné komponenty** ve svých vlastních metodách.
- Můžete zajít ještě dál a nechat mediátor **zodpovídat i za vytváření a ničení** objektů komponent. V takovém případě se mediátor může začít podobat vzoru **Factory** nebo **Fascade**.
- Komponenty by měly uchovávat **odkaz na objekt mediátoru**. Toto propojení se obvykle nastavuje v **konstruktoru komponenty**, kam se objekt mediátoru předává jako argument.
- Upravte kód komponent tak, aby **volaly notifikační metodu mediátoru** místo přímého volání metod jiných komponent.
Kód, který volá jiné komponenty, **přesuňte do třídy mediátoru** a spouštějte jej vždy, když mediátor obdrží oznámení od dané komponenty.

PRO A PROTI

Pro

- **Princip jediné odpovědnosti (Single Responsibility Principle).** Můžete extražovat komunikaci mezi různými komponentami do jednoho místa, což usnadňuje porozumění a údržbu.
- **Princip otevřenosti/uzavřenosti (Open/Closed Principle).** Můžete zavádět nové mediátory, aniž byste museli měnit samotné komponenty.
- Můžete **snížit provázanost** mezi jednotlivými komponentami programu.
- Můžete **snáze znova použít** jednotlivé komponenty.

Proti

- Postupem času se z mediátoru může stát **God Object**

POROVNÁNÍ S OSTATNÍMI VZORY

- Chain of Responsibility, Command, Mediator a Observer řeší různé způsoby propojení odesílatelů a příjemců požadavků:
- **Chain of Responsibility** předává požadavek postupně podél dynamického řetězce potenciálních příjemců, dokud jej někdo nezpracuje.
- **Command** vytváří jednosměrné spojení mezi odesílateli a příjemci.
- **Mediator** odstraňuje přímá spojení mezi odesílateli a příjemci a nutí je komunikovat nepřímo prostřednictvím objektu mediátoru.
- **Observer** umožňuje příjemcům dynamicky se přihlašovat k odběru a odhlašovat se z přijímání požadavků.
- **Facade** a **Mediator** mají podobnou roli: snaží se organizovat spolupráci mezi mnoha úzce propojenými třídami.
- **Facade** definuje zjednodušené rozhraní k podsystému objektů, ale nepřidává žádnou novou funkci. Samotný podsystém o existenci fasády neví a objekty uvnitř podsystému mohou komunikovat přímo.
- **Mediator** centralizuje komunikaci mezi komponentami systému. Komponenty znají pouze mediátor a nepřistupují přímo k ostatním komponentám.

- Rozdíl mezi **Mediator** a **Observer** je často nejasný. Ve většině případů můžete implementovat kterýkoli z těchto vzorů; někdy je však možné je použít současně. Podívejme se, jak na to.
- Hlavním cílem **Mediatoru** je odstranit vzájemné závislosti mezi sadou komponent systému. Tyto komponenty se místo toho stanou závislými na jediném objektu mediátoru. Cílem **Observeru** je vytvořit dynamická jednosměrná propojení mezi objekty, kde některé objekty působí jako podřízené jiným.
- Existuje populární implementace vzoru **Mediator**, která využívá **Observer**. Objekt mediátoru zde hraje roli vydavatele (publisher) a komponenty fungují jako odběratelé (subscribers), kteří se přihlašují k událostem mediátoru a odhlašují se z nich. Když je Mediator implementován tímto způsobem, může vypadat velmi podobně jako Observer.
- Když si nejste jisti, pamatujte, že vzor Mediator lze implementovat i jinak. Například můžete trvale propojit všechny komponenty se stejným objektem mediátoru. Tato implementace nebude připomínat Observer, ale stále bude instancí vzoru Mediator.
- Představte si nyní program, kde se všechny komponenty staly vydavateli, což umožňuje dynamické propojení mezi sebou. Nebude existovat centralizovaný objekt mediátoru, pouze distribuovaná sada pozorovatelů (observers).

PŘÍKLAD

VISITOR

- Visitor je behaviorální návrhový vzor, který vám umožňuje oddělit algoritmy od objektů, na kterých tyto algoritmy pracují.

CO ŘEŠÍ?

- Představte si, že váš tým vyvíjí aplikaci, která pracuje s geografickými informacemi strukturovanými jako jeden obrovský graf. Každý uzel grafu může reprezentovat složitou entitu, například město, ale také detailnější prvky, jako jsou průmyslové oblasti, turistické atrakce apod. Uzel je propojen s jinými uzly, pokud mezi objekty, které reprezentují, vede cesta. Ve skutečnosti je každý typ uzlu reprezentován vlastní třídou, zatímco každý konkrétní uzel je objektem.
- V určitém okamžiku jste dostali úkol implementovat export grafu do formátu XML. Zpočátku se práce jevila jako poměrně jednoduchá. Plánovali jste přidat metodu export do každé třídy uzlu a pak využít řekurzi k průchodu každým uzlem grafu, přičemž se volá metoda export. Řešení bylo jednoduché a elegantní: díky polymorfismu nebyl kód, který volal metodu export, svázán s konkrétními třídami uzlů.

- Bohužel, systémový architekt odmítl umožnit vám měnit existující třídy uzelů. Tvrzal, že kód je již v produkci a nechtěl riskovat jeho porušení kvůli možné chybě ve vašich změnách.
- Navíc zpochybnil, zda má smysl mít kód pro export do XML uvnitř tříd uzelů. Primární úlohou těchto tříd byla práce s geodaty. Chování pro export do XML by tam působilo cize.
- Byl zde i další důvod pro odmítnutí. Bylo velmi pravděpodobné, že po implementaci této funkce někdo z marketingového oddělení požádá o možnost exportu do jiného formátu nebo o nějakou jinou zvláštní funkci. To by vás přinutilo znova měnit tyto cenné a křehké třídy.

ŘEŠENÍ

- Vzorec Visitor navrhuje umístit nové chování do samostatné třídy zvané visitor, místo toho, abyste se jej pokoušeli integrovat do existujících tříd. Původní objekt, který měl chování vykonat, je nyní předán jako argument jedné z metod visitoru, čímž tato metoda získá přístup ke všem potřebným datům obsaženým v objektu.
- Co když však toto chování může být vykonáno nad objekty různých tříd? Například v našem případě s exportem do XML bude pravděpodobně implementace mírně odlišná pro různé třídy uzlů. Visitor třída tak může definovat nejen jednu, ale celou sadu metod, z nichž každá přijímá argumenty odlišného typu, například takto:

```
class ExportVisitor implements Visitor {  
    void doForCity(City c) { ... }  
    void doForIndustry(Industry f) { ... }  
    void doForSightSeeing(SightSeeing ss) { ... }  
}
```

- Ale jak přesně bychom tyto metody volali, zejména při práci s celým grafem? Tyto metody mají různé signatury, takže nemůžeme využít polymorfismus. Abychom vybrali správnou metodu visitoru, která dokáže zpracovat daný objekt, museli bychom kontrolovat jeho třídu. Nepůsobí to jako noční můra?

```
for (Node node : graph) {  
    if (node instanceof City)  
        exportVisitor.doForCity((City) node);  
    if (node instanceof Industry)  
        exportVisitor.doForIndustry((Industry) node);  
    // ...  
}
```

- Můžete se ptát, proč nepoužijeme přetížení metod (method overloading)? To znamená, že všechny metody mají stejné jméno, i když podporují různé sady parametrů.
- Bohužel, i když náš jazyk přetížení podporuje (např. Java či C#), nebude nám to stačit.
- Protože přesná třída objektu uzlu není dopředu známá, mechanismus přetížení nedokáže určit správnou metodu k vykonání a použije výchozí metodu, která přijímá objekt základní třídy Node.

- Visitor vzorec tento problém řeší. Používá techniku zvanou **Double Dispatch**, která umožňuje vykonat správnou metodu na objektu bez složitých podmínek.
- Místo toho, abychom nechali klienta vybírat správnou verzi metody, delegujeme tuto volbu na objekty, které předáváme visitoru jako argument. Protože objekty znají své vlastní třídy, dokážou správně vybrat metodu ve visitoru méně neprakticky.
- Objekty „přijímají“ visitor a říkají mu, která metoda návštěvy má být vykonána.

```
for (Node node : graph) {  
    node.accept(exportVisitor);  
}  
  
class City extends Node {  
    void accept(Visitor v) {  
        v.doForCity(this);  
    }  
}
```

```
class Industry extends Node {  
    void accept(Visitor v) {  
        v.doForIndustry(this);  
    }  
}
```

- Pokud nyní extrahujeme společné rozhraní pro všechny visitory, všechny existující uzly mohou pracovat s jakýmkoli visitor objektem, který do aplikace zavedete.
- Když budete chtít přidat nové chování související s uzly, stačí implementovat novou třídu visitoru.

VYUŽITÍ

- Použijte vzorec Visitor, když potřebujete provést operaci nad všemi prvky složité objektové struktury (například stromu objektů).
- Vzorec Visitor vám umožňuje vykonat operaci nad množinou objektů různých tříd tím, že objekt visitor implementuje několik variant téže operace, které odpovídají všem cílovým třídám.
- Použijte Visitor k vyčištění obchodní logiky pomocných chování.
- Vzorec umožňuje, aby hlavní třídy vaší aplikace byly více zaměřené na své primární úkoly, tím, že všechna ostatní chování extrahujete do sady tříd visitorů.
- Použijte vzorec, když má nějaké chování smysl pouze u některých tříd hierarchie, ale ne u ostatních.
- Toto chování můžete extrahovat do samostatné třídy visitor a implementovat pouze ty metody návštěvy, které přijímají objekty relevantních tříd, zbytek můžete nechat prázdný.

JAK IMPLEMENTOVAT

1. Deklarujte rozhraní visitor s množinou „návštěvních“ metod, po jedné pro každou konkrétní třídu elementu, která existuje v programu.
2. Deklarujte rozhraní elementu. Pokud pracujete s existující hierarchií tříd elementů, přidejte abstraktní metodu „accept“ do základní třídy hierarchie. Tato metoda by měla přijímat objekt visitoru jako argument.
3. Implementujte metodu accept ve všech konkrétních třídách elementů. Tyto metody musí jednoduše přesměrovat volání na odpovídající návštěvní metodu přijatého visitor objektu, která odpovídá třídě aktuálního elementu.
4. Třídy elementů by měly pracovat s visitory pouze přes rozhraní visitoru. Visitři však musí být obeznámeni se všemi konkrétními třídami elementů, uvedenými jako typy parametrů návštěvních metod.
5. Pro každé chování, které nelze implementovat uvnitř hierarchie elementů, vytvořte novou konkrétní třídu visitor a implementujte všechny návštěvní metody.
Můžete se setkat se situací, kdy visitor bude potřebovat přístup k některým privátním členům třídy elementu. V takovém případě můžete buď tyto pole nebo metody zpřístupnit (což by porušilo zapouzdření elementu), nebo vložit třídu visitor do třídy elementu. Druhá možnost je možná pouze tehdy, pokud programovací jazyk podporuje vnořené třídy.
6. Klient musí vytvořit objekty visitoru a předat je elementům pomocí metod „accept“.

VÝHODY A NEVÝHODY

- **Princip otevřenosti/zavřenosti (Open/Closed Principle):** Můžete zavést nové chování, které může pracovat s objekty různých tříd, aniž byste měnili tyto třídy.
- **Princip jediné odpovědnosti (Single Responsibility Principle):** Můžete přesunout více verzi stejného chování do jedné třídy.
- Objekt visitor může při práci s různými objekty akumulovat užitečné informace. To může být vhodné, pokud chcete projít nějakou složitou strukturu objektů, například strom objektů, a aplikovat visitor na každý objekt této struktury.
- Je třeba aktualizovat všechny visitory vždy, když je do hierarchie elementů přidána nebo z ní odstraněna nějaká třída.
- Visitory mohou postrádat potřebný přístup k privátním polím a metodám elementů, se kterými mají pracovat.

VZTAHY S JINÝMI VZORY

- Visitor lze považovat za silnější verzi vzoru Command. Jeho objekty mohou provádět operace nad různými objekty různých tříd.
- Visitor můžete použít k provedení operace nad celým stromem typu Composite.
- Visitor lze kombinovat s Iterator vzorem k procházení složité datové struktury a provádění operace nad jejími prvky, i když všechny patří do různých tříd.

PŘÍKLAD

TEMPLATE METHOD

- Template Method je behaviorální návrhový vzor, který definuje kostru algoritmu v nadřazené třídě, ale umožňuje podtřídám pře definovat konkrétní kroky algoritmu, aniž by měnily jeho strukturu.

CO ŘEŠÍ?

- Představte si, že vytváříte aplikaci pro datamining, která analyzuje firemní dokumenty. Uživatelé dodávají aplikaci dokumenty v různých formátech (PDF, DOC, CSV) a aplikace se snaží z těchto dokumentů extrahovat smysluplná data ve jednotném formátu.
- První verze aplikace dokázala pracovat pouze s DOC soubory. V následující verzi byla schopná podporovat CSV soubory. O měsíc později jste ji „naučili“ extrahovat data z PDF souborů.
- V určitém bodě jste si všimli, že všechny tři třídy mají hodně podobného kódu. I když byl kód pro práci s různými datovými formáty v každé třídě odlišný, kód pro zpracování a analýzu dat je téměř identický. Nebylo by skvělé zbavit se duplikace kódu a zároveň zachovat strukturu algoritmu?
- Další problém souvisejí s klientským kódem, který tyto třídy používal. Obsahoval spoustu podmínek, které vybíraly správný postup v závislosti na třídě objektu pro zpracování. Kdyby všechny tři třídy zpracování měly společné rozhraní nebo nadřazenou třídu, bylo by možné odstranit podmínky v klientském kódu a využít polymorfismus při volání metod na objektu pro zpracování.

ŘEŠENÍ

- Vzor Template Method naznačuje, že algoritmus rozdělíte na řadu kroků, tyto kroky převedete na metody a vložíte sérii volání těchto metod do jediné šablonové (template) metody. Krok může být buď abstraktní, nebo mít nějakou výchozí implementaci. Aby klient mohl algoritmus použít, měl by poskytnout vlastní podtřídu, implementovat všechny abstraktní kroky a případně přepsat některé volitelné (ale nikoli samotnou šablonovou metodu).
- Podívejme se, jak by to fungovalo v naší aplikaci pro dolování dat. Můžeme vytvořit základní třídu pro všechny tři algoritmy parsování. Tato třída definuje šablonovou metodu sestávající ze série volání různých kroků zpracování dokumentu. Zpočátku můžeme všechny kroky deklarovat jako abstraktní, čímž přinutíme podtřídy poskytnout vlastní implementace těchto metod. V našem případě mají podtřídy již všechny potřebné implementace, takže jediná věc, kterou možná bude třeba upravit, jsou podpisy metod tak, aby odpovídaly metodám nadřazené třídy.

- Nyní se podívejme, co můžeme udělat pro odstranění duplicitního kódu. Zdá se, že kód pro otevřání/zavírání souborů a extrakci/parsing dat se liší pro různé datové formáty, takže tyto metody není třeba měnit. Implementace ostatních kroků, jako je analýza surových dat a tvorba zpráv, je však velmi podobná, takže ji lze přesunout do základní třídy, kde ji mohou sdílet podtřídy.
- Jak vidíte, máme dva typy kroků:
- abstraktní kroky musí implementovat každá podtřída
- volitelné kroky již mají nějakou výchozí implementaci, ale stále je lze přepsat, pokud je to potřeba
- Existuje ještě další typ kroku, nazývaný hook. Hook je volitelný krok s prázdným tělem. Šablonová metoda funguje i v případě, že hook není přepsán. Obvykle jsou hooky umístěny před a po klíčových krocích algoritmu, čímž poskytují podtřídám dodatečné body pro rozšíření algoritmu.

VYUŽITÍ

- Použijte vzor Template Method, když chcete umožnit klientům rozšiřovat pouze konkrétní kroky algoritmu, ale ne celý algoritmus nebo jeho strukturu.
- Template Method umožňuje převést monolitický algoritmus na sérii jednotlivých kroků, které mohou být snadno rozšiřovány podtřídami, přičemž struktura definovaná v nadřazené třídě zůstává zachována.
- Použijte tento vzor, pokud máte několik tříd, které obsahují téměř identické algoritmy s menšími odlišnostmi. V důsledku toho byste museli upravovat všechny třídy, pokud se algoritmus změní.
- Když takový algoritmus převedete na šablonovou metodu, můžete také přesunout kroky s podobnou implementací do nadřazené třidy, čímž odstraníte duplicitní kód. Kód, který se liší mezi podtřídami, může zůstat v podtřídách.

JAK IMPLEMENTOVAT

1. Analyzujte cílový algoritmus a zjistěte, zda ho lze rozdělit na jednotlivé kroky. Zvažte, které kroky jsou společné pro všechny podtřídy a které budou vždy unikátní.
2. Vytvořte abstraktní základní třídu a deklarujte šablonovou metodu (template method) a sadu abstraktních metod představujících jednotlivé kroky algoritmu. Nastíněte strukturu algoritmu v šablonové metodě voláním odpovídajících kroků. Zvažte označení šablonové metody jako final, aby podtřídy nemohly tuto metodu přepisovat.
3. Je v pořádku, pokud všechny kroky budou nakonec abstraktní. Některé kroky však mohou mít prospěch z výchozí implementace. Podtřídy nemusí tyto metody implementovat.
4. Zvažte přidání háčků (hooks) mezi klíčové kroky algoritmu.
5. Pro každou variantu algoritmu vytvořte novou konkrétní podtřídu. Musí implementovat všechny abstraktní kroky, ale může také přepsat některé volitelné kroky.

VÝHODY A NEVÝHODY

- Můžete umožnit klientům přepisovat pouze určité části rozsáhlého algoritmu, takže jsou méně ovlivněni změnami, které se týkají ostatních částí algoritmu.
- Můžete vytáhnout duplicitní kód do nadřídy.
- Někteří klienti mohou být omezeni poskytnutou kostrou algoritmu.
- Můžete porušit princip Liskovovy substituce tím, že podtřídou potlačíte výchozí implementaci kroku.
- Šablonové metody bývají těžší na údržbu, čím více kroků obsahuje.

VZTAHY S JINÝMI VZORY

- Factory Method je specializací šablonové metody (Template Method). Zároveň může Factory Method sloužit jako krok ve velké šablonové metodě.
- Template Method je založena na dědičnosti: umožňuje měnit části algoritmu rozšiřováním těchto částí v podtřídách. Strategy je založen na kompozici: můžete měnit části chování objektu tím, že mu poskytnete různé strategie odpovídající tomuto chování. Template Method funguje na úrovni třídy, takže je statická. Strategy funguje na úrovni objektu, což umožňuje přepínat chování za běhu programu.

PŘÍKLAD

NÁVRHOVÉ VZORY

Přednáška 9

BEHAVIORAL PATTERNS

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

STATE

- Stav (State) je návrhový vzor chování, který umožňuje objektu měnit své chování podle toho, v jakém je vnitřním stavu.
- Navenek to vypadá, jako by objekt změnil svou třídu.

- Vzor **State** úzce souvisí s konceptem **konečného automatového stroje (Finite-State Machine)**.
- Hlavní myšlenkou je, že v daném okamžiku může být program pouze v **jednom ze zadaného (konečného) počtu stavů**.
- V každém stavu se program chová jinak a může okamžitě přepínat mezi jednotlivými stavami.
- Avšak podle aktuálního stavu může (nebo nemusí) být povoleno přepnutí do některých jiných stavů.
- Tato pravidla přepínání, nazývaná **přechody**, jsou také konečná a předem daná.

- Stejný přístup můžete použít i na objekty.
- Představme si, že máme třídu Document.
- Dokument může být v jednom ze tří stavů:
 - Draft,
 - Moderation a
 - Published.
- Metoda publish se v každém stavu chová jinak:
 - v Draft přesune dokument do stavu Moderation,
 - v Moderation zveřejní dokument, ale jen pokud je aktuální uživatel administrátor,
 - v Published nedělá vůbec nic.

- Konečné automaty bývají obvykle implementovány pomocí množství podmíněných příkazů (if nebo switch), které vybírají správné chování podle aktuálního stavu objektu.
- Obvykle je tento „stav“ jednoduše sada hodnot ve vlastnostech objektu.
- I když jste o konečných automatech nikdy neslyšeli, pravděpodobně jste nějaký stav už implementovali.
- Zní vám povědomě následující struktura?

```
class Document is
    field state: string
    // ...
    method publish() is
        switch (state)
            "draft":
                state = "moderation"
                break
            "moderation":
                if (currentUser.role == "admin")
                    state = "published"
                break
            "published":
                // Do nothing.
                break
        // ...
    
```

- Největší slabina stavového stroje založeného na podmírkách se projeví tehdy, jakmile začnete do třídy Document přidávat více stavů a více stavově záviselného chování.
- Většina metod začne obsahovat monstrózní podmínky, které určují správné chování podle aktuálního stavu.
- Takový kód je velmi obtížně udržovatelný, protože každá změna v logice přechodů může vyžadovat úpravy podmínek ve všech metodách.
- Problém se obvykle zvětšuje s tím, jak projekt roste.
- Je těžké předpovědět všechny možné stavy a přechody už při návrhu.
- Původně jednoduchý stavový stroj s několika málo podmínkami se tak snadno může časem změnit v nepřehledný a nafouknutý chaos.

ŘEŠENÍ

- Vzor **State** navrhuje, abyste pro **všechny možné stavы objektu** vytvořili samostatné třídy a veškeré chování závislé na stavu přesunuli do těchto tříd.
- Místo toho, aby původní objekt (tzv. *context*) implementoval všechna chování sám, **uchovává odkaz na jeden z objektů představujících aktuální stav** a veškerou práci související se stavem deleguje na tento objekt.
- Pokud chcete přepnout context do jiného stavu, jednoduše **nahradíte aktivní objekt stavu** jiným objektem, který představuje nový stav. To je možné pouze tehdy, pokud všechny třídy stavů dodržují stejné rozhraní a context s nimi pracuje prostřednictvím tohoto rozhraní.
- Tato struktura může na první pohled připomínat vzor **Strategy**, ale existuje jeden zásadní rozdíl:
 - Ve vzoru *State* mohou jednotlivé stavы **vědět o existenci jiných stavů** a mohou **iniciovat přechody mezi sebou**, zatímco strategie si obvykle mezi sebou vůbec neuvědomují, že jiné strategie existují.

VYUŽITÍ

- Použijte vzor State, pokud máte objekt, který se **chová různě v závislosti na svém aktuálním stavu**, počet stavů je velký a kód specifický pro jednotlivé stavы se často mění.
 - Vzor navrhuje přesunout veškerý kód související se stavы do samostatných tříd. Díky tomu můžete přidávat nové stavы nebo měnit existující nezávisle na ostatních, což snižuje náklady na údržbu.
- Použijte tento vzor, pokud máte třídu **zanesenu rozsáhlými podmínkami**, které mění chování podle aktuálních hodnot jejich atributů.
 - State umožňuje tyto větve podmínek přesunout do metod příslušných tříd stavů. Přitom můžete z hlavní třídy odstranit dočasná pole a pomocné metody používané pouze pro logiku stavů.
- Použijte State, pokud máte **množství duplicitního kódu** mezi podobnými stavы a přechody v podmínkami řízeném stavovém stroji.
 - Tento vzor vám umožní vytvořit hierarchii stavových tříd a omezit duplicitu díky přesunutí společného kódu do abstraktních bázových tříd.

JAK IMPLEMENTOVAT

1. Rozhodněte se, která třída bude vystupovat jako kontext.

Může to být existující třída, která už nyní obsahuje kód závislý na stavu; nebo nová třída, pokud je stavový kód roztroušený přes více tříd.

2. Deklarujte rozhraní pro stavy.

Ačkoli může kopírovat všechny metody kontextu, zaměřte se pouze na ty, které skutečně mohou obsahovat chování závislé na stavu.

3. Pro každý skutečný stav vytvořte třídu, která bude dané rozhraní implementovat. Poté projděte metody kontextu a veškerý kód týkající se konkrétního stavu přesuňte do vytvořených tříd.

4. Při přesunu kódu můžete zjistit, že potřebuje přístup k privátním členům kontextu. Existuje několik řešení:
 - Zveřejněte potřebná pole nebo metody (public).
 - Přesuňte extrahované chování do veřejné metody kontextu a volejte ji ze stavové třídy. (Rychlé, ale ne úplně elegantní — lze později zlepšit.)
 - Vnořte stavové třídy přímo do třídy kontextu, pokud to jazyk umožnuje.
5. V kontextu přidejte proměnnou typu stavového rozhraní a veřejný setter, který umožní tuto hodnotu měnit.
6. Projděte metody kontextu znovu a nahraďte prázdné podmínky pro jednotlivé stavy voláním odpovídajících metod aktuálního stavového objektu.
7. K přepnutí stavu kontextu vytvořte instanci jedné z tříd stavů a předejte ji kontextu. To lze provést:
 - přímo v kontextu,
 - ve stavových třídách,
 - nebo v kódu klienta.
- Třída, která vytvoří novou instanci stavu, pak bude záviset na konkrétní stavové třídě.

VÝHODY A NEVÝHODY

Pro

- Princip jediné odpovědnosti (Single Responsibility Principle): Kód týkající se konkrétních stavů je organizován do samostatných tříd.
- Princip otevřenosti/uzavřenosti (Open/Closed Principle): Je možné zavést nové stavы, aniž byste měnili stavající stavové třídy nebo kontext.
- Zjednodušení kódu kontextu: Odstraněním objemných podmínek stavového automatu se kód stává přehlednějším.

Proti

- Použití tohoto vzoru může být zbytečně komplikované, pokud má stavový automat jen několik stavů nebo se stavы mění jen zřídka.

VZTAHY S JINÝMI VZORY

- Bridge, State, Strategy (a do jisté míry Adapter) mají velmi podobnou strukturu. Všechny tyto vzory jsou založeny na **kompozici**, tedy na **delegování** práce na jiné objekty. Nicméně každý řeší jiný problém. Vzor není jen recept, jak strukturovat kód určitým způsobem, ale také může ostatním vývojářům sdělit problém, který daný vzor řeší.
- State lze považovat za rozšíření Strategy. Oba vzory jsou založeny na kompozici: mění chování kontextu delegováním části práce na pomocné objekty. Strategy činí tyto objekty zcela nezávislými a neinformovanými o sobě navzájem. State však neomezuje závislosti mezi konkrétními stavami, což jim umožňuje libovolně měnit stav kontextu.

PŘÍKLAD



STRUCTURAL DESIGN PATTERNS

STRUKTURÁLNÍ NÁVRHOVÉ VZORY

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

DECORATOR

- Decorator je strukturální návrhový vzor, který vám umožňuje přidávat nové chování objektům tím, že tyto objekty vložíte do speciálních obalových objektů (wrapperů), které obsahují tato chování.

CO ŘEŠÍ?

- Představte si, že pracujete na knihovně pro notifikace, která umožňuje jiným programům informovat uživatele o důležitých událostech.
- První verze knihovny byla založena na třídě Notifier, která měla jen několik polí, konstruktor a jedinou metodu send.
- Tato metoda mohla přijmout zprávu od klienta a odeslat ji na seznam e-mailů, který byl předán do notifikátoru prostřednictvím konstruktoru.
- Aplikace třetí strany, která fungovala jako klient, měla vytvořit a nakonfigurovat objekt notifikátoru jednou a poté jej používat vždy, když nastala důležitá událost.

- V určitém okamžiku si uvědomíte, že uživatelé knihovny očekávají víc než jen e-mailové notifikace.
- Mnozí by chtěli dostávat SMS při kritických problémech, jiní by chtěli být informováni přes Facebook a korporátní uživatelé by jistě ocenili notifikace přes Slack.
- Jak těžké by to mohlo být?
- Rozšířili jste tedy třídu Notifier a přidali nové notifikační metody do nových podtříd.
- Nyní měl klient instanciovat požadovanou třídu notifikátoru a používat ji pro všechny další notifikace.

- Pak se však někdo rozumně zeptal: „Proč nelze použít více typů notifikací současně? Pokud vám hoří dům, pravděpodobně chcete být informováni všemi dostupnými kanály.“
- Snažili jste se tento problém řešit vytvořením speciálních podtříd, které kombinovaly několik notifikačních metod v jedné třídě.
- Brzy se však ukázalo, že tento přístup způsobí obrovský nárůst kódu, nejen v knihovně, ale i na straně klienta.
- Je třeba najít jiný způsob, jak strukturovat třídy notifikací, aby jejich počet náhodou nepřekonal nějaký Guinnessův rekord.

ŘEŠENÍ

- Rozšíření třídy je první věc, která vás napadne, když potřebujete změnit chování objektu. Nicméně dědičnost má několik vážných omezení, o kterých byste měli vědět.
- Dědičnost je **statická**. Chování existujícího objektu nelze měnit za běhu programu. Lze pouze nahradit celý objekt jiným, vytvořeným z jiné podtřídy. Podtřídy mohou mít jen jednu rodičovskou třídu. Ve většině jazyků dědičnost neumožňuje, aby třída dědila chování více tříd najednou.
- Jedním ze způsobů, jak tato omezení obejít, je použití **agregace** nebo **kompozice** místo dědičnosti. Obě alternativy fungují téměř stejně: jeden objekt má referenci na jiný objekt a deleguje mu část práce, zatímco při dědičnosti je objekt sám schopný tuto práci vykonávat díky chování zděděnému od své nadtřídy.

- S tímto přístupem můžete snadno nahradit „pomocný“ objekt jiným, čímž změníte chování kontejneru za běhu programu. Objekt může využívat chování různých tříd tím, že má reference na více objektů a deleguje jim různé úkoly. **Agregace/kompozice** je klíčovým principem mnoha návrhových vzorů, včetně dekorátoru.
- „Wrapper“ je alternativní název pro vzor Decorator, který jasně vyjadřuje hlavní myšlenku vzoru.
- Wrapper je objekt, který může být propojen s cílovým objektem. Wrapper obsahuje stejnou sadu metod jako cílový objekt a deleguje mu všechny požadavky, které přijme. Wrapper však může změnit výsledek tím, že před nebo po předání požadavku cílovému objektu něco provede.

- Kdy se jednoduchý wrapper stane skutečným dekorátorem? Jak již bylo zmíněno, wrapper implementuje stejný interface jako objekt, který obaluje.
- Proto jsou tyto objekty z pohledu klienta identické.
- Nastavte referenční pole wrapperu tak, aby přijímal libovolný objekt, který tento interface dodržuje.
- Tím můžete objekt obalit do více wrapperů a přidat mu kombinované chování všech wrapperů.

- V našem příkladu s notifikacemi necháme jednoduché chování e-mailových notifikací uvnitř základní třídy Notifier, ale všechny ostatní notifikační metody převedeme na dekorátory.
- Kód klienta by pak obalil základní objekt notifikátoru do sady dekorátorů podle svých preferencí. Výsledné objekty budou strukturovány jako zásobník. Poslední dekorátor v zásobníku bude objektem, se kterým klient skutečně pracuje.
- Vzhledem k tomu, že všechny dekorátory **implementují stejný interface** jako základní notifikátor, zbytek kódu klienta nebude řešit, zda pracuje s „čistým“ objektem notifikátoru nebo s dekorovaným objektem.
- Stejný přístup lze použít i pro jiná chování, například formátování zpráv nebo sestavování seznamu příjemců. Klient může objekt obalit libovolnými vlastními dekorátory, pokud dodržují stejný interface jako ostatní.

KDY JEJ POUŽÍT

- Použijte vzor Decorator, když potřebujete být schopni přiřadit objektům další **chování za běhu programu**, aniž byste narušili kód, který tyto objekty používá.
- Decorator vám umožňuje strukturovat obchodní logiku do vrstev, vytvořit dekorátor pro každou vrstvu a skládat objekty s různými kombinacemi této logiky za běhu. Kód klienta může se všemi těmito objekty zacházet stejně, protože všechny dodržují společný interface.
- Použijte vzor, když je rozšíření chování objektu pomocí dědičnosti nepraktické nebo nemožné.
- Mnoho programovacích jazyků má klíčové slovo final, které lze použít k zabránění dalšímu dědění třídy. U finální třídy je jediný způsob, jak znova použít existující chování, obalit třídu vlastním wrapperem pomocí vzoru Decorator.

JAK IMPLEMENTOVAT

- Ujistěte se, že váš obchodní doménový model lze reprezentovat jako primární komponentu s více volitelnými vrstvami nad ní.
- Zjistěte, které metody jsou společné jak pro primární komponentu, tak pro volitelné vrstvy. Vytvořte rozhraní komponenty a deklarujte tam tyto metody.
- Vytvořte konkrétní třídu komponenty a definujte v ní základní chování.
- Vytvořte základní dekorátorovou třídu. Měla by mít pole pro uchování referenčního bodu na obalený objekt. Pole by mělo být deklarováno typem komponentního rozhraní, aby umožňovalo propojení s konkrétními komponentami i dekorátory. Základní dekorátor musí veškerou práci delegovat na obalený objekt.
- Ujistěte se, že všechny třídy implementují rozhraní komponenty.
- Vytvořte konkrétní dekorátory rozšířením základního dekorátoru. Konkrétní dekorátor musí vykonat své chování před nebo po volání metody rodiče (která vždy deleguje na obalený objekt).
- Kód klienta je zodpovědný za vytváření dekorátorů a jejich skládání způsobem, který klient potřebuje.

VÝHODY A NEVÝHODY

Výhody

- Můžete rozšířit chování objektu, aniž byste museli vytvářet novou podtřídu.
- Můžete přidávat nebo odebírat odpovědnosti objektu za běhu programu.
- Můžete kombinovat několik chování tím, že objekt obalíte do více dekorátorů.
- Princip jediné odpovědnosti (Single Responsibility Principle).
- Můžete rozdělit monolitickou třídu, která implementuje mnoho možných variant chování, do několika menších tříd.

Nevýhody

- Je obtížné odstranit konkrétní obal (wrapper) ze zásobníku obalů.
- Je obtížné implementovat dekorátor tak, aby jeho chování nezáviselo na pořadí v zásobníku dekorátorů.
- Počáteční konfigurační kód vrstev může vypadat docela nevhledně.

VZTAHY S DALŠÍMI VZORY

- **Adapter** poskytuje zcela odlišné rozhraní pro přístup k existujícímu objektu. Naproti tomu u vzoru Decorator rozhraní zůstává stejné nebo se rozšiřuje. Navíc Decorator podporuje rekurzivní kompozici, což není možné při použití Adapteru.
- S Adapterem přistupujete k existujícímu objektu přes jiné rozhraní. S **Proxy** zůstává rozhraní stejné. S Decoratorem přistupujete k objektu přes rozšířené rozhraní.
- **Chain of Responsibility** a Decorator mají velmi podobnou strukturu tříd. Oba vzory spoléhají na rekurzivní kompozici k předávání vykonávání přes řadu objektů. Nicméně existuje několik zásadních rozdílů.
 - Handlery v CoR mohou provádět libovolné operace nezávisle na sobě a mohou kdykoli zastavit předávání požadavku dál. Naproti tomu různí dekorátoři mohou rozšířit chování objektu, aniž by porušili základní rozhraní. Dekorátorům není dovoleno přerušit tok požadavku.
- **Composite** a Decorator mají podobné struktury, protože oba spoléhají na rekurzivní kompozici k organizaci otevřeného počtu objektů.
 - Decorator je podobný Composite, ale má pouze jednoho potomka. Další významný rozdíl: Decorator přidává další odpovědnosti k obalenému objektu, zatímco Composite pouze „sčítá“ výsledky svých potomků.
 - Vzory mohou také spolupracovat: můžete použít Decorator k rozšíření chování konkrétního objektu v Composite stromu.

- Návrhy, které intenzivně využívají Composite a Decorator, mohou často profitovat z použití **Prototype**. Aplikace tohoto vzoru umožňuje klonovat složité struktury místo jejich opětovné konstrukce od začátku.
- **Decorator** vám umožňuje změnit „vzhled“ objektu, zatímco **Strategy** umožňuje změnit jeho „vnitřní chování“.
- Decorator a **Proxy** mají podobné struktury, ale velmi odlišné účely. Oba vzory jsou založeny na principu kompozice, kdy jeden objekt deleguje část práce jinému. Rozdíl je v tom, že Proxy obvykle spravuje životní cyklus svého servisního objektu sama, zatímco kompozice dekorátorů je vždy řízena klientem.

PŘÍKLAD

FACADE

- Fasáda (Facade) je strukturální návrhový vzor, který poskytuje zjednodušené rozhraní k nějaké knihovně, frameworku nebo jiné složité sadě tříd.

K ČEMU JE DOBRÝ

- Představte si, že musíte svůj kód pracovat s rozsáhlou sadou objektů, které patří do sofistikované knihovny nebo frameworku. Obvykle byste museli všechny tyto objekty inicializovat, sledovat jejich závislosti, volat metody ve správném pořadí a podobně.
- Výsledkem by bylo, že by se obchodní logika vašich tříd stala těsně propojenou s implementačními detaily třetích stran, což by ztěžovalo její porozumění a údržbu.

ŘEŠENÍ

- Fasáda je třída, která poskytuje jednoduché rozhraní ke složitému podsystému obsahujícímu mnoho pohyblivých částí.
- Fasáda může poskytovat omezenou funkciálnost ve srovnání s přímou prací s podsystémem.
- Nicméně zahrnuje pouze ty funkce, o které klienti skutečně stojí.
- Použití fasády je užitečné, když potřebujete integrovat svou aplikaci se sofistikovanou knihovnou, která nabízí desítky funkcí, ale vy potřebujete jen malou část její funkciálnosti.
- Například aplikace, která nahrává krátká zábavná video s kočkami na sociální síťě, by mohla potenciálně používat profesionální knihovnu pro konverzi video.
- Nicméně vše, co opravdu potřebuje, je třída s jedinou metodou `encode(filename, format)`. Po vytvoření takové třídy a jejím propojení s knihovnou pro konverzi video získáte svou první fasádu.

- Použijte vzor Fasáda, když potřebujete mít omezené, ale přehledné rozhraní ke složitému podsystému.
- Často se podsystémy časem stávají složitějšími. I použití návrhových vzorů obvykle vede k vytvoření většího množství tříd. Podsystém může být flexibilnější a snáze znova použitelný v různých kontextech, ale množství konfiguračního a podpůrného kódu, který vyžaduje od klienta, stále roste. Fasáda se snaží tento problém vyřešit tím, že poskytuje zkratku k nejčastěji používaným funkcím podsystému, které vyhovují většině požadavků klienta.

- Použijte Fasádu, pokud chcete strukturovat podsystém do vrstev.
- Vytvořte fasády pro definování vstupních bodů do každé úrovně podsystému. Můžete snížit propojení mezi více podsystémy tím, že budete vyžadovat, aby komunikovaly pouze přes fasády.
 - Například se můžeme vrátit k našemu frameworku pro konverzi videa. Lze ho rozdělit do dvou vrstev: video- a audio-vrstva. Pro každou vrstvu můžete vytvořit fasádu a poté zajistit, aby třídy každé vrstvy komunikovaly mezi sebou prostřednictvím těchto fasád. Tento přístup se velmi podobá návrhovému vzoru Mediátor.

JAK IMPLEMENTOVAT

- Zkontrolujte, zda je možné poskytnout jednodušší rozhraní než to, které již existující pod systém nabízí. Jste na správné cestě, pokud toto rozhraní činí klientský kód nezávislým na mnoha třídách pod systému.
- Deklarujte a implementujte toto rozhraní v nové třídě fasády. Fasáda by měla přesměrovat volání z klientského kódu na příslušné objekty pod systému. Fasáda by měla být zodpovědná za inicializaci pod systému a správu jeho životního cyklu, pokud to klientský kód již nedělá.
- Pro plné využití vzoru zajistěte, aby veškerý klientský kód komunikoval s pod systémem pouze prostřednictvím fasády. Klientský kód je nyní chráněn před jakýmkoli změnami v kódu pod systému. Například pokud se pod systém aktualizuje na novou verzi, budete muset upravit pouze kód ve fasádě.
- Pokud fasáda naroste příliš velká, zvažte vytažení části jejího chování do nové, specializované fasády.

VÝHODY A NEVÝHODY

Výhody

- Můžete izolovat svůj kód od složitosti podsystému.

Nevýhody

- Fasáda se může stát „God objektem“, který je propojen se všemi třídami aplikace.

VZTAHY S DALŠÍMI VZORY

- Fasáda definuje nové rozhraní pro existující objekty, zatímco **Adapter** se snaží udělat existující rozhraní použitelným. Adapter obvykle obaluje jen jeden objekt, zatímco Fasáda pracuje s celým pod systémem objektů.
- **Abstract Factory** může sloužit jako alternativa k Fasádě, pokud chcete jen skrýt způsob, jakým jsou objekty pod systému **vytvářeny**, před klientským kódem.
- **Flyweight** ukazuje, jak vytvořit mnoho malých objektů, zatímco Fasáda ukazuje, jak vytvořit jeden objekt, který reprezentuje celý pod systém.

- Fasáda a **Mediátor** mají podobnou úlohu:
 - snaží se zorganizovat spolupráci mezi mnoha úzce propojenými třídami. Fasáda definuje zjednodušené rozhraní k podsystému objektů, ale nezavádí žádnou novou funkctionalitu. Samotný podsystém o fasádě neví. Objekty uvnitř podsystému mohou komunikovat přímo.
 - Mediátor centralizuje komunikaci mezi komponentami systému. Komponenty znají pouze objekt mediátora a nekomunikují přímo.
- Třída Fasáda může být často převedena na **Singleton**, protože v většině případů stačí jedený objekt fasády.
- Fasáda je podobná **Proxy** v tom, že oba obalují složitou entitu a inicializují ji samostatně. Na rozdíl od Fasády má Proxy stejné rozhraní jako její servisní objekt, což je činí zaměnitelnými.

PŘÍKLAD

NÁVRHOVÉ VZORY

Přednáška 10



STRUCTURAL DESIGN PATTERNS

STRUKTURÁLNÍ NÁVRHOVÉ VZORY

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

COMPOSITE - KOMPOZICE

- Composite je strukturální návrhový vzor, který vám umožňuje skládat objekty do stromových struktur a následně s těmito strukturami pracovat, jako by šlo o jednotlivé objekty.
- **Kompozice je metoda používaná k psaní znova použitelného kódu.**
Je jí dosaženo tehdy, když jsou objekty složeny z jiných menších objektů se specifickými chováními. Jinými slovy: větší objekty s širší funkčností jsou tvořeny menšími objekty, které poskytují konkrétní funkcionalitu.
- Cílem kompozice je stejný výsledek jako u dědičnosti, avšak místo toho, aby objekt získával vlastnosti z rodičovské třídy, je větší objekt **složen** z jiných objektů a může využívat jejich funkcionalitu.

CO ŘEŠÍ?

- Použití návrhového vzoru Composite má smysl pouze tehdy, když hlavní model vaší aplikace lze reprezentovat jako strom.
- Například si představte, že máte dva typy objektů:
- Produkty a Krabice. Krabice může obsahovat několik Produktů, stejně jako řadu menších Krabic.
- Tyto malé Krabice mohou také držet nějaké Produkty nebo dokonce ještě menší Krabice, a tak dále.

- Řekněme, že se rozhodnete vytvořit objednávkový systém využívající tyto třídy.
- Objednávky by mohly obsahovat jednoduché produkty bez jakéhokoli balení, stejně jako krabice naplněné produkty... a další krabice.
- Jak byste určili celkovou cenu takové objednávky? Můžete zkusit přímý přístup: rozbalit všechny krabice, projít všechny produkty a spočítat celkovou cenu.
- To by bylo možné v reálném světě; ale v programu to není tak jednoduché jako spustit smyčku.
- Musíte předem znát třídy Produktů a Krabic, kterými procházíte, úroveň vnoření krabic a další nepříjemné detaily.
- To vše činí přímý přístup buď příliš neohrabaným, nebo dokonce nemožným.

ŘEŠENÍ

- Návrhový vzor Composite navrhuje pracovat s Produkty a Krabicemi prostřednictvím **společného rozhraní**, které deklaruje metodu pro výpočet celkové ceny.
- Jak by tato metoda fungovala? U produktu by jednoduše vrátila cenu produktu. U krabice by procházela každý předmět, který krabice obsahuje, zjistila jeho cenu a poté vrátila celkovou cenu pro tuhú krabici.
- Pokud by jeden z těchto předmětů byla menší krabice, tažo krabice by také začala procházet svůj obsah, a tak dále, dokud by nebyly vypočítány ceny všech vnitřních komponent.
- Krabice by dokonce mohla přidat nějaké další náklady k celkové ceně, například náklady na balení.
- Největší výhodou tohoto přístupu je, že se nemusíte starat o konkrétní třídy objektů, které tvoří strom.
- Nemusíte vědět, zda je objekt jednoduchý produkt nebo složitá krabice.
- Všechny je můžete zacházet stejně prostřednictvím společného rozhraní.
- Když zavoláte metodu, objekty samy předají požadavek dále stromem.

KDY POUŽÍT?

- Použijte návrhový vzor Composite, pokud potřebujete implementovat stromovou strukturu objektů.
- Návrhový vzor Composite vám poskytuje dva základní typy prvků, které sdílejí společné rozhraní: jednoduché listy a složité kontejnery. **Kontejner může být složen jak z listů, tak z jiných kontejnerů**. To vám umožňuje vytvořit zanořenou rekurzivní strukturu objektů, která připomíná strom.
- Použijte tento vzor, pokud chcete, aby klientský kód zacházel s jednoduchými i složitými prvky jednotně.
- Všechny prvky definované vzorem Composite sdílejí společné rozhraní. Díky tomuto rozhraní se klient nemusí starat o konkrétní třídu objektů, se kterými pracuje.

JAK IMPLEMENTOVAT

- Ujistěte se, že základní model vaší aplikace lze reprezentovat jako stromovou strukturu. Snažte se ji rozdělit na jednoduché prvky a kontejnery. Pamatujte, že kontejnery musí být schopny obsahovat jak jednoduché prvky, tak i jiné kontejnery.
- Deklarujte komponentní rozhraní s metodami, které dávají smysl pro jednoduché i složité komponenty.
- Vytvořte třídu listu (leaf) pro reprezentaci jednoduchých prvků. Program může mít více různých tříd listů.
- Vytvořte třídu kontejneru (container) pro reprezentaci složitých prvků. V této třídě poskytujte pole pro ukládání referencí na podprvky. Pole musí být schopné ukládat jak listy, tak kontejnery, proto jej deklarujte s typem komponentního rozhraní.

- Při implementaci metod komponentního rozhraní mějte na paměti, že kontejner by měl delegovat většinu práce na podprvky.
- Nakonec definujte metody pro přidávání a odstraňování podprvků v kontejneru.
- Mějte na paměti, že tyto operace mohou být deklarovány v komponentním rozhraní. To by porušovalo princip Interface Segregation, protože metody budou prázdné u listové třídy. Klient však bude schopen zacházet se všemi prvky jednotně, i když tvoříte strom.

POROVNÁNÍ

pro

- Můžete pohodlněji pracovat se složitými stromovými strukturami: využijte polymorfismus a rekurzi ve svůj prospěch.
- Princip Open/Closed: můžete do aplikace zavádět nové typy prvků, aniž byste porušili existující kód, který nyní pracuje se stromem objektů.

proti

- Může být obtížné poskytnout společné rozhraní pro třídy, jejichž funkčnost se příliš liší. V některých případech byste museli rozhraní komponenty příliš zgeneralizovat, což ztěžuje jeho pochopení.

DĚDIČNOST VS KOMPOZICE

- V oblasti objektově orientovaného programování, zejména v Javě, hrají klíčovou roli dva základní koncepty – kompozice a dědičnost.
- Oba slouží k opětovnému využití kódu, ale každý zcela odlišným způsobem a s vlastními výhodami a důsledky pro návrh softwaru.

DĚDIČNOST

- Dědičnost je základní kámen objektově orientovaného programování, který umožňuje jedné třídě (podtřídě) dědit pole a metody z jiné třídy (nadřídy). Tento vztah je tzv. „**is-a**“ vztah, kdy podtřída je konkrétnější instancí nadřídy.
- Zde je myšlenková mapa ilustrující koncept dědičnosti v OOP.
- Rozebírá definici, výhody, typy a klíčové koncepty a poskytuje příklady dědičnosti, ukazující, jak jsou třídy propojeny a jak mohou dědit vlastnosti a chování od jiných tříd.

VÝHODY DĚDIČNOSTI

- Opětovné využití kódu:
 - Dědičnost umožňuje podtřídám znovu použít kód svých nadtříd, což snižuje duplicitu.
- Jednoduchost:
 - Poskytuje přímočarý způsob, jak navázat vztahy mezi třídami, což usnadňuje tvorbu a pochopení hierarchií tříd.

NEVÝHODY DĚDIČNOSTI

- Těsné provázání:
 - Podtřídy jsou úzce provázány se svými nadtřídami, což činí kód méně flexibilní a obtížněji modifikovatelný bez dopadu na podtřídy.
- Hierarchie dědičnosti:
 - Nadměrné používání dědičnosti může vést ke složitým hierarchiím tříd, které se stávají obtížně spravovatelnými a pochopitelnými.

DĚDIČNOST - PŘÍKLAD

```
class Animal {  
    void eat() {  
        System.out.println("This animal eats food.");  
    }  
  
class Dog extends Animal {  
    void bark() {  
        System.out.println("The dog barks.");  
    }  
  
public class TestInheritance {  
    public static void main(String[] args) {  
        Dog myDog = new Dog();  
        myDog.eat(); // Inherited method  
        myDog.bark();  
    }  
}
```

- V tomto příkladu je třída Dog podtřídou Animal a dědí metodu eat.
- To demonstруje vztah „is-a“, kdy pes „je“ zvíře.

KOMPOZICE

- Kompozice v objektově orientovaném programování (OOP) je návrhový princip, který se používá k modelování vztahu „**has-a**“ mezi objekty.
- Umožňuje kombinovat jednoduché objekty nebo datové typy a vytvářet složitější struktury.
- Kompozice je způsob, jak navrhнуть nebo strukturovat třídy tak, aby bylo možné znovu využívat kód a zároveň zachovat flexibilitu návrhu.
- V kompozici třída známá jako **kompozit** obsahuje objekt jiné třídy, známé jako **komponenta**.
- Místo dědičnosti z komponenty drží kompozitní třída odkaz na tuto komponentu.
- Tento přístup umožňuje delegovat odpovědnosti komponentní třídě, což je způsob, jak vyjádřit, že kompozitní třída „má“ komponentní třídu.

VÝHODY KOMPOZICE

- Flexibilita:
 - Kompozice poskytuje větší flexibilitu tím, že umožňuje měnit chování třídy za běhu skládáním s různými objekty.
- Volné provázání:
 - Podporuje volné provázání tříd, což zjednodušuje refactoring a údržbu systému.

NEVÝHODY KOMPOZICE

- Komplexita návrhu:
 - Může vést k složitějším návrhům, protože vyžaduje explicitní definování a správu vztahů mezi třídami.
- Vývojová režie:
 - Počátečně může být větší režie při vývoji, protože je třeba navrhnout více rozhraní a tříd.

KOMPOZICE

```
class Engine {  
    void start() {  
        System.out.println("Engine is starting");  
    }  
  
}  
  
class Car {  
    private Engine engine; // Car "has-a" Engine  
  
    Car() {  
        engine = new Engine();  
    }  
  
    void start() {  
        engine.start();  
        System.out.println("Car is starting");  
    }  
  
}  
  
public class TestComposition {  
    public static void main(String[] args) {  
        Car myCar = new Car();  
        myCar.start();  
    }  
}
```

- V tomto příkladu **auto (Car) má motor (Engine)**. Místo toho, aby auto dědilo vlastnosti a metody motoru, auto obsahuje objekt motoru, což demonstruje vztah „has-a“ (má).

KDY POUŽÍT KOMPOZICI MÍSTO DĚDIČNOSTI

- **Potřeba flexibility:** Použijte kompozici, když potřebujete dynamicky měnit chování systému nebo očekáváte, že komponenty se mohou měnit nezávisle na sobě.
- **Vyhnut se těsnému propojení:** Pokud chcete předejít silnému provázání a zachovat vysokou koherenci, kompozice je vhodná volba.
- **Pro vztah „is-a“:** Použijte dědičnost, pokud jsou vaše třídy v jasném hierarchickém vztahu, ale mějte na paměti hloubku stromu dědičnosti.

PŘÍKLAD

PROXY

- Proxy je strukturální návrhový vzor, který umožňuje poskytnout nahradu nebo zástupný objekt pro jiný objekt.
- Proxy řídí přístup k původnímu objektu a umožňuje provést určitou akci buď před tím, než požadavek dorazí k původnímu objektu, nebo po jeho zpracování.

CO ŘEŠÍ?

- Proč byste chtěli kontrolovat přístup k nějakému objektu?
- Zde je příklad: máte masivní objekt, který spotřebovává velké množství systémových prostředků. Potřebujete ho jen občas, ale ne vždy. Mohli byste použít *lazy initialization* (odloženou inicializaci): vytvořit tento objekt až ve chvíli, kdy je skutečně potřeba.
- Všichni klienti tohoto objektu by však museli spouštět určitý odložený inicializační kód. Bohužel by to pravděpodobně vedlo k velké duplikaci kódu.
- V ideálním světě bychom chtěli tento kód umístit přímo do třídy objektu, ale to není vždy možné. Například když je třída součástí uzavřené knihovny třetí strany.

ŘEŠENÍ

- Proxy vzor navrhuje, abyste vytvořili novou **proxy třídu** se stejným rozhraním jako původní objekt služby. Poté upravíte aplikaci tak, aby všem klientům původního objektu předávala objekt proxy.
- Když proxy obdrží požadavek od klienta, vytvoří skutečný objekt služby a deleguje na něj veškerou práci. Jaký je v tom ale přínos?
- Pokud potřebujete provést něco **před** nebo **po** hlavní logice třídy, proxy vám to umožní, aniž byste museli měnit danou třídu. Protože proxy implementuje stejné rozhraní jako původní třída, lze ji předat jakémukoli klientovi, který očekává skutečný objekt služby.

VYUŽITÍ

- **Lazy initialization (virtuální proxy)**

- Používá se tehdy, když máte těžkotonážní objekt, který spotřebovává mnoho systémových prostředků tím, že je neustále vytvořený, i když ho potřebujete jen občas.
- Místo vytvoření objektu při startu aplikace můžete jeho inicializaci odložit až na okamžik, kdy je skutečně potřeba.

- **Řízení přístupu (protection proxy)**

- Používá se tehdy, když chcete, aby službu mohli využívat jen určité typy klientů; například když jsou vaše objekty klíčovou součástí operačního systému a klienti jsou různé spuštěné aplikace (včetně škodlivých).
- Proxy předá požadavek službě pouze tehdy, pokud klient splňuje určité přístupové podmínky.

- **Lokální reprezentace vzdálené služby (remote proxy)**

- Používá se tehdy, když se objekt služby nachází na vzdáleném serveru.
- Proxy v tomto případě předává požadavky klienta přes síť a řeší všechny nepříjemné detaily komunikace.

- **Logování požadavků (logging proxy)**
 - Používá se tehdy, když chcete uchovávat historii požadavků zaslaných službě.
 - Proxy může každý požadavek zaznamenat, ještě než jej předá službě.
- **Caching výsledků požadavků (caching proxy)**
 - Používá se tehdy, když je potřeba ukládat výsledky požadavků a řídit životní cyklus této cache — zejména pokud jsou výsledky velké.
 - Proxy může implementovat cache pro opakující se požadavky, které vracejí stejné výsledky. Jako klíče může používat parametry požadavků.
- **Chytrá reference (smart reference)**
 - Používá se tehdy, když potřebujete uvolnit těžkotonážní objekt, pokud jej už žádný klient nepoužívá.
 - Proxy může sledovat klienty, kteří získali odkaz na službu nebo její výsledky. Čas od času může projít seznam klientů a ověřit, zda jsou stále aktivní. Pokud je seznam prázdný, proxy může objekt zrušit a uvolnit systémové prostředky.
 - Proxy může také sledovat, zda klient objekt změnil. Nezměněné objekty lze pak znova použít pro jiné klienty.

JAK IMPLEMENTOVAT

- Pokud neexistuje žádné rozhraní služby, vytvořte ho, aby byly proxy a služba zaměnitelné. Extrahování rozhraní ze třídy služby ale nemusí být vždy možné, protože byste museli změnit všechny klienty služby, aby toto rozhraní používali.
Plán B je vytvořit proxy jako podtřídu třídy služby — tím zdědí její veřejné rozhraní.
- **1. Vytvořte proxy třídu**
 - Proxy by měla mít pole pro uložení odkazu na objekt služby.
Obvykle proxy vytváří a spravuje životní cyklus služby. Ve vzácných případech je služba proxy předána přes konstruktor klientem.
- **2. Implementujte metody proxy podle jejich účelu**
 - Ve většině případů by proxy měla po provedení vlastní práce delegovat volání na objekt služby.
- **3. Zvažte vytvoření tovární (factory) metody**
 - Ta může rozhodovat, zda klient dostane proxy nebo skutečnou službu.
Může to být jednoduchá statická metoda v proxy třídě, nebo plnohodnotná tovární metoda
- **4. Zvažte implementaci lazy inicializace**
 - Proxy může službu vytvořit až v okamžiku, kdy je skutečně potřeba.

POROVNÁNÍ

Pro

- Můžete řídit objekt služby, aniž by o tom klienti věděli.
- Můžete spravovat životní cyklus objektu služby, když klienti o něj nestojí.
- Proxy funguje i v případě, že objekt služby není připraven nebo není dostupný.
- Princip otevřenosti/uzavřenosti (Open/Closed Principle). Můžete zavést nové proxy, aniž byste měnili službu nebo klienty.

Proti

- Kód se může stát složitějším, protože je třeba zavést mnoho nových tříd.
- Odezva od služby může být zpožděná.

VZTAH S DALŠÍMI VZORY

- U **Adapteru** přistupujete k existujícímu objektu přes odlišné rozhraní. U **Proxy** zůstává rozhraní stejné. U **Decoratoru** přistupujete k objektu přes rozšířené rozhraní.
- **Facade** je podobný **Proxy**, protože oba „bufrují“ složitou entitu a inicializují ji samostatně. Na rozdíl od Facade má Proxy stejné rozhraní jako jeho servisní objekt, což je činí zaměnitelnými.
- **Decorator** a **Proxy** mají podobnou strukturu, ale velmi odlišný účel. Oba vzory jsou založeny na principu kompozice, kde jeden objekt deleguje část práce jinému. Rozdíl je v tom, že Proxy obvykle spravuje životní cyklus svého servisního objektu samostatně, zatímco kompozice u Decoratoru je vždy řízena klientem.

PŘÍKLAD

NÁVRHOVÉ VZORY

Přednáška 11

STRUCTURAL DESIGN PATTERNS

STRUKTURÁLNÍ NÁVRHOVÉ VZORY

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

ADAPTER

- Adapter je strukturální návrhový vzor, který umožňuje spolupráci objektů s nekompatibilními rozhraními.

CO ČEŠÍ?

- Představte si, že vytváříte aplikaci pro sledování akciového trhu. Aplikace stahuje burzovní data z několika zdrojů ve formátu XML a poté z nich uživateli zobrazuje grafy a diagramy.
- V určitém okamžiku se rozhodnete aplikaci vylepšit integrací chytré analytické knihovny třetí strany. Má to ale háček: tato analytická knihovna pracuje pouze s daty ve formátu JSON.
- Můžete sice upravit knihovnu tak, aby pracovala s XML. To by však mohlo rozbití existující kód, který na knihovně závisí.
- A co je ještě horší — možná ani nemáte přístup ke zdrojovému kódu knihovny, takže takový přístup je zcela nemožný.

ŘEŠENÍ

- Můžete vytvořit adaptér. Jedná se o speciální objekt, který převádí rozhraní jednoho objektu tak, aby mu jiný objekt rozuměl.
- Adaptér obalí jeden z objektů, aby skryl složitost převodu, který probíhá „v zákulisí“. Obalený objekt si ani neuvědomuje, že je adaptér použit. Například můžete obalit objekt, který pracuje v metrech a kilometrech, adaptérem, jenž všechna data převádí na imperiální jednotky, jako jsou stopy a míle.
- Adaptéry neumí jen převádět data do různých formátů, ale také pomáhají objektům s odlišnými rozhraními spolupracovat. Funguje to následovně:
 1. Adaptér získá rozhraní kompatibilní s jedním z existujících objektů.
 2. Pomocí tohoto rozhraní může existující objekt bezpečně volat metody adaptéru.
 3. Když adaptér obdrží volání, předá požadavek druhému objektu — ale ve formátu a pořadí, které tento druhý objekt očekává.
- Někdy je dokonce možné vytvořit obousměrný adaptér, který umí převádět volání oběma směry.
- Vraťme se zpět k naší aplikaci pro sledování akciového trhu. Abychom vyřešili problém nekompatibilních formátů, můžete vytvořit adaptéry XML-to-JSON pro každou třídu analytické knihovny, se kterou váš kód pracuje přímo. Poté upravíte svůj kód tak, aby komunikoval s knihovnou pouze přes tyto adaptéry. Když adaptér obdrží volání, převede příchozí XML data na JSON strukturu a předá volání odpovídajícím metodám obaleného analytického objektu.

KDY POUŽÍT

- Použijte třídu Adapter, pokud chcete použít nějakou existující třídu, ale její rozhraní není kompatibilní se zbytkem vašeho kódu.
- Návrhový vzor Adapter vám umožňuje vytvořit prostřední vrstvu — třídu, která funguje jako překladač mezi vaším kódem a starším (legacy) kódem, externí knihovnou nebo jakoukoli třídou s neobvyklým rozhraním.
- Použijte tento vzor také v případě, že chcete znova použít několik existujících podtříd, které postrádají určitou společnou funkциálnost, kterou nelze přidat do jejich předka.
- Mohli byste rozšířit každou podtřídu a doplnit chybějící funkciálnost do nových potomků. Museli byste však duplikovat kód ve všech těchto nových třídách, což je velmi špatná praxe.
- Mnohem elegantnějším řešením je vložit chybějící funkciálnost do třídy adaptéra. Poté můžete objekty s chybějícími funkcemi obalit adaptérem a dynamicky tak získat potřebnou funkciálnost. Aby to fungovalo, musí mít cílové třídy společné rozhraní a atribut (pole) adaptéra by měl toto rozhraní dodržovat. Tento přístup je velmi podobný návrhovému vzoru Decorator.

JAK IMPLEMENTOVAT

- Ujistěte se, že máte alespoň dvě třídy s nekompatibilními rozhraními:
 - **Užitečnou servisní třídu**, kterou nemůžete změnit (často třída třetí strany, legacy kód nebo třída s mnoha existujícími závislostmi).
 - **Jednu nebo více klientských tříd**, které by měly prospěch z použití této servisní třídy.
1. **Deklarujte klientské rozhraní** a popište, jak klienti komunikují se servisní třídou.
 2. **Vytvořte adapterovou třídu** a zajistěte, aby implementovala klientské rozhraní. Prozatím nechte všechny metody prázdné.
 3. Přidejte do adapterové třídy **pole pro uložení reference na servisní objekt**. Obyklá praxe je inicializovat foto pole přes konstruktor, ale někdy je pohodlnější předat objekt adapteru až při volání jeho metod.
 4. **Postupně implementujte všechny metody klientského rozhraní** v adapterové třídě. Adapter by měl delegovat většinu skutečné práce na servisní objekt a řešit pouze konverzi rozhraní nebo formátu dat.
 5. **Klienti by měli používat adapter přes klientské rozhraní**. Tímto způsobem můžete měnit nebo rozšiřovat adaptory, aniž byste ovlivnili klientský kód.

PROVNÁNÍ

Pro

- **Princip jediné odpovědnosti (Single Responsibility Principle).** Můžete oddělit kód rozhraní nebo konverze dat od primární obchodní logiky programu.
- **Princip otevřenosti/uzavřenosti (Open/Closed Principle).** Můžete do programu zavádět nové typy adapterů, aniž byste porušili existující klientský kód, pokud s adaptery pracují prostřednictvím klientského rozhraní.

proti

- Celková složitost kódu se zvyšuje, protože je potřeba zavést sadu nových rozhraní a tříd. Někdy je jednodušší jednoduše upravit služební třídu tak, aby odpovídala zbytku vašeho kódu.

VZTAH S DALŠÍMI VZORY

- **Bridge** je obvykle navržen předem, což umožňuje vyvíjet části aplikace nezávisle na sobě. Na druhou stranu, Adapter se běžně používá u již existující aplikace, aby některé jinak nekompatibilní třídy mohly spolupracovat.
- Adapter poskytuje zcela jiné rozhraní pro přístup k existujícímu objektu. Naproti tomu u vzoru **Decorator** rozhraní buď zůstává stejné, nebo se rozšiřuje. Navíc Decorator podporuje rekurzivní kompozici, což není možné při použití Adapteru.
- S Adapterem přistupujete k existujícímu objektu přes jiné rozhraní. U **Proxy** rozhraní zůstává stejné. U **Decoratoru** přistupujete k objektu přes rozšířené rozhraní.
- **Facade** definuje nové rozhraní pro existující objekty, zatímco **Adapter** se snaží učinit existující rozhraní použitelné. **Adapter** obvykle obaluje jen jeden objekt, zatímco Facade pracuje s celým pod systémem objektů.
- **Bridge, State, Strategy** (a do jisté míry i Adapter) mají velmi podobné struktury. Všechny tyto vzory jsou založeny na kompozici, tedy delegování práce jiným objektům. Nicméně každý řeší jiný problém. Vzor není jen recept na strukturování kódu určitým způsobem; fakté komunikuje ostatním vývojářům, jaký problém daný vzor řeší.

PŘÍKLAD

BRIDGE

- Bridge je strukturální návrhový vzor, který umožňuje rozdělit velkou třídu nebo sadu úzce propojených tříd do dvou samostatných hierarchií — abstrakce a implementace — které mohou být vyvíjeny nezávisle na sobě.

CO ŘEŠÍ?

- Řekněme, že máte geometrickou třídu Shape se dvěma podtřídami:
 - Circle a Square.
- Chcete tuto hierarchii tříd rozšířit o barvy, takže plánujete vytvořit podtřídy Red a Blue pro tvary.
- Avšak protože již máte dvě podtřídy, budete muset vytvořit čtyři kombinace tříd, například BlueCircle a RedSquare.
- Přidávání nových typů tvarů a barev do hierarchie způsobí, že se počet tříd exponenciálně zvýší.
- Například pro přidání trojúhelníku byste museli zavést dvě podtřídy, jednu pro každou barvu.
- A poté by přidání nové barvy vyžadovalo vytvoření tří podtříd, jednu pro každý typ tvaru.
- Čím dál pokračujeme, tím horší to bude.

ŘEŠENÍ

- Tento problém vzniká, protože se snažíme rozšířit třídy tvarů ve dvou nezávislých dimenzích: podle tvaru a podle barvy. To je velmi běžný problém při dědičnosti tříd.
- Vzor **Bridge** se snaží tento problém vyřešit **přechodem od dědičnosti k kompozici objektů**. To znamená, že jednu z dimenzi vyextrahujeme do samostatné hierarchie tříd, takže původní třídy budou odkazovat na objekt nové hierarchie, místo aby veškerý svůj stav a chování obsahovaly v jedné třídě.
- Podle tohoto přístupu můžeme kód související s barvou vyčlenit do vlastní třídy se dvěma podtřídami: Red a Blue.
- Třída Shape pak získá referenční pole ukazující na jeden z objektů barvy. Tvar může nyní delegovat veškerou práci související s barvou na propojený objekt barvy. Tato reference bude fungovat jako most (bridge) mezi třídami Shape a Color.
- Od tohoto okamžiku přidávání nových barev nebude vyžadovat změny v hierarchii tvarů a naopak.

VYUŽITÍ

- Vzor **Bridge** použijte, když chcete rozdělit a zorganizovat **monolitickou třídu**, která má několik variant nějaké funkcionality (například třída, která může pracovat s různými databázovými servery).
- Čím větší třída je, tím těžší je pochopit, jak funguje, a tím déle trvá provedení změny. Změny provedené v jedné z variant funkcionality mohou vyžadovat úpravy v celé třídě, což často vede k chybám nebo opomenutí některých kritických vedlejších efektů.
- Vzor **Bridge** umožňuje rozdělit monolitickou třídu do několika hierarchií tříd. Po tomto rozdělení můžete měnit třídy v každé hierarchii nezávisle na třídách v ostatních hierarchiích. Tento přístup zjednodušuje údržbu kódu a minimalizuje riziko narušení existujícího kódu.

- Použijte tento vzor, pokud potřebujete rozšířit třídu ve **více ortogonálních (nezávislých) dimenzích**.
- Bridge doporučuje vyčlenit **samostatnou hierarchii tříd** pro každou dimenzi. Původní třída deleguje související práci na objekty patřící do téhoto hierarchií místo toho, aby vše prováděla sama.
- Použijte Bridge, pokud potřebujete být schopni **přepínat implementace za běhu programu**.
- I když je to volitelné, vzor Bridge vám umožňuje nahradit objekt implementace uvnitř abstrakce. Stačí jednoduše přiřadit nové hodnoty do pole.
- Mimochodem, právě tento bod je hlavním důvodem, proč mnoho lidí zaměňuje Bridge s **Strategy**. Pamatujte, že vzor není jen způsob, jak strukturovat třídy – může také komunikovat **účel a řešený problém**.

JAK IMPLEMENTOVAT

1. Identifikujte ortogonální dimenze ve vašich třídách. Tyto nezávislé koncepty mohou být například: abstrakce/platforma, doména/infrastruktura, front-end/back-end, nebo rozhraní/implementace.
2. Zjistěte, jaké operace klient potřebuje, a definujte je v základní třídě abstrakce.
3. Určete operace dostupné na všech platformách. Deklarujte ty, které abstrakce potřebuje, v obecné implementační rozhraní.
4. Pro všechny platformy ve vaší doméně vytvořte konkrétní implementační třídy, ale ujistěte se, že všechny dodržují implementační rozhraní.
5. V rámci třídy abstrakce přidejte referenční pole pro typ implementace. Abstrakce deleguje většinu práce na objekt implementace, na který foto pole odkazuje.
6. Pokud máte několik variant vyšší logiky, vytvořte upřesněné abstrakce pro každou variantu děděním ze základní abstrakce.
7. Klientský kód by měl předat objekt implementace konstruktoru abstrakce, aby je propojil. Poté může klient zapomenout na implementaci a pracovat pouze s objektem abstrakce.

POROVNÁNÍ

Pro

- Můžete vytvářet platformně nezávislé třídy a aplikace.
- Klientský kód pracuje s vysokou úrovní abstrakcí a není vystaven detailům platformy.
- Open/Closed Principle: Můžete zavádět nové abstrakce a implementace nezávisle na sobě.
- Single Responsibility Principle: Můžete se soustředit na vysokou logiku v abstrakci a na detaily platformy v implementaci.

Proti

- Použití vzoru na **vysoce kohezivní třídu** může kód zkomplikovat.

PŘÍKLAD

FLYWEIGHT

- Flyweight je strukturální návrhový vzor, který umožňuje umístit do dostupné paměti RAM více objektů tím, že sdílí společné části stavu mezi více objekty místo toho, aby každý objekt obsahoval všechna data samostatně.

CO ČEŠÍ?

- Aby sis po dlouhých pracovních hodinách trochu odpočinul, rozhodl jsi se vytvořit jednoduchou videohru: hráči by se pohybovali po mapě a stříleli na sebe navzájem. Rozhodl jsi se implementovat realistický systém částic, který měl být charakteristickým prvkem hry. Obrovské množství kulí, raket a střepin z výbuchů mělo létat po celé mapě a přinášet hráči napínavý zážitek.
- Po dokončení jsi provedl poslední commit, sestavil hru a poslal ji kamarádovi k testování. I když hra běžela bez problémů na tvém počítači, tvůj kamarád si s ní dlouho nezahrál. Na jeho počítači se hra po několika minutách hrani neustále zhroutila. Po několika hodinách pátrání v debug logách jsi zjistil, že k pádu hry došlo kvůli nedostatečné paměti RAM. Ukázalo se, že kamarádův počítač byl mnohem méně výkonný než tvůj, a proto se problém projevil tak rychle.
- Skutečný problém souvisel s tvým systémem částic. Každá částice, například kulka, raketa nebo střepina, byla reprezentována samostatným objektem obsahujícím spoustu dat. V určitém okamžiku, když chaos na obrazovce hráče dosáhl vrcholu, se nově vytvořené částice už nevešly do zbývající RAM, a program se zhroutil.

ŘEŠENÍ

- Při bližším zkoumání třídy Particle si možná všimnete, že pole color a sprite zabírají mnohem více paměti než ostatní pole.
- Co je horší, tato dvě pole uchovávají téměř identická data napříč všemi částicemi. Například všechny kulky mají stejnou barvu a sprite.
- Ostatní části stavu částice, jako jsou souřadnice, vektor pohybu a rychlosť, jsou unikátní pro každou částici.
- Koneckonců, hodnoty těchto polí se v čase mění. Tato data představují neustále se měnící kontext, ve kterém částice existuje, zatímco barva a sprite zůstávají pro každou částici konstantní. Tato konstantní data objektu se obvykle nazývají intrinsic state (vnitřní stav).
- Žijí uvnitř objektu; ostatní objekty je mohou pouze číst, nikoliv měnit. Zbytek stavu objektu, často měněný „zvenčí“ jinými objekty, se nazývá extrinsic state (vnější stav).

- Návrhový vzor **Flyweight** doporučuje přestat ukládat extrinsic state uvnitř objektu. Místo toho by měl být tento stav předáván konkrétním metodám, které na něm závisí. Pouze intrinsic state zůstává uvnitř objektu, což umožňuje jeho opětovné použití v různých kontextech. Výsledkem je, že potřebujete méně těchto objektů, protože se liší pouze ve vnitřním stavu, který má mnohem méně variant než stav vnější.
- Vrátíme-li se ke hře, pokud bychom extrinsic state extrahovali z třídy Particle, stačily by pouze tři různé objekty k reprezentaci všech částic ve hře: kulka, raketa a střepina. Jak jste si pravděpodobně už všimli, objekt, který uchovává pouze intrinsic state, se nazývá **flyweight**.

UKLÁDÁNÍ VNĚJŠÍHO STAVU

- Kam se přesune extrinsic state? Nějaká třída ho přece musí stále uchovávat, že?
- Ve většině případů se přesune do kontejnerového objektu, který agreguje objekty ještě před aplikací vzoru.
- V našem případě je to hlavní objekt Game, který uchovává všechny částice ve svém poli particles. Abychom přesunuli extrinsic state do této třídy, je potřeba vytvořit několik polí pro ukládání souřadnic, vektorů a rychlosti každé jednotlivé částice.
- Ale to není všechno. Potřebujete další pole pro ukládání referencí na konkrétní flyweight, který reprezentuje částici. Tato pole musí být synchronizována, aby bylo možné přistupovat ke všem datům jedné částice pomocí stejného indexu.

- Elegantnější řešení je vytvořit samostatnou **context class**, která bude uchovávat extrinsic state spolu s referencí na flyweight objekt. Tento přístup vyžaduje mít pouze jedno pole v kontejnerové třídě.
- Nebudeme potřebovat tolik těchto kontextových objektů, kolik jsme jich měli původně? Technicky ano. Ale problém je v tom, že tyto objekty jsou mnohem menší než předtím.
- Nejvíce paměťově náročná pole byla přesunuta do pouhých několika flyweight objektů. Nyní může tisíc malých kontextových objektů znova používat jeden těžký flyweight objekt místo toho, aby ukládaly tisíc kopií jeho dat.

FLYWEIGHT A IMMUTABILITY

- Protože stejný flyweight objekt může být použit v různých kontextech, musíte zajistit, aby jeho stav nemohl být měněn.
- Flyweight by měl inicializovat svůj stav pouze jednou, přes konstruktor.
- Neměl by poskytovat žádné settery ani veřejná pole pro ostatní objekty.

FLYWEIGHT TOVÁRNA

- Pro pohodlnější přístup k různým flyweightům můžete vytvořit **factory method**, která spravuje pool existujících flyweight objektů. Metoda přijímá intrinsic state požadovaného flyweightu od klienta, hledá existující flyweight objekt odpovídající tomuto stavu a vrací ho, pokud byl nalezen. Pokud ne, vytvoří nový flyweight a přidá ho do poolu.
- Existuje několik možností, kam tuto metodu umístit. Nejlogičtější je do kontejneru flyweightů. Alternativně můžete vytvořit novou tovární třídu. Nebo můžete metodu udělat statickou a umístit ji přímo do samotné třídy flyweight

CO ČEŠÍ

- Vzorec **Flyweight** používejte pouze tehdy, pokud váš program musí podporovat obrovské množství objektů, která se sotva vejdou do dostupné RAM.
- Výhoda použití vzoru silně závisí na tom, jak a kde je použit. Nejvíce se hodí, když:
 1. aplikace potřebuje vytvořit obrovské množství podobných objektů
 2. toto spotřebovává veškerou dostupnou RAM na cílovém zařízení
 3. objekty obsahují duplicitní stavy, které lze extrahovat a sdílet mezi více objekty

JAK IMPLEMENTOVAT

- Rozdělte pole třídy, která se má stát flyweightem, na dvě části:
 - **intrinsic state**: pole, která obsahují neměnná data duplicitně uložená v mnoha objektech
 - **extrinsic state**: pole, která obsahují kontextová data unikátní pro každý objekt
- Pole představující intrinsic state nechte ve třídě, ale zajistěte, aby byla **neměnná**. Počáteční hodnoty by měla získat pouze v konstruktoru.
- Projděte metody, které používají pole extrinsic state. Pro každé pole použité v metodě zaveděte nový parametr a použivejte ho místo pole.
- Volitelně vytvořte **tovární třídu** (factory class) pro správu poolu flyweightů. Měla by zkontrolovat existující flyweight před vytvořením nového. Jakmile je továrna připravena, klienti by měli flyweighty vyžadovat pouze přes ni. Popisují požadovaný flyweight předáním jeho intrinsic state továrně.
- Klient musí uchovávat nebo počítat hodnoty extrinsic state (kontext), aby mohl volat metody flyweight objektů. Pro pohodlí může být extrinsic state spolu s polem odkazujícím na flyweight přesunut do samostatné **context třídy**.

POROVNÁNÍ

Pro

- Můžete ušetřit spoustu RAM, pokud váš program obsahuje velké množství podobných objektů.

Proti

- Může dojít k výměně úspory RAM za CPU cykly, pokud je třeba kontextová data pokaždé přepočítávat při volání metody flyweightu.
- Kód se stává mnohem složitějším. Noví členové týmu se budou vždy ptát, proč byl stav entity oddělen tímto způsobem.

VZTAHY S JINÝMI VZORY

- Sdílené listové uzly stromu **Composite** můžete implementovat jako **Flyweight**, abyste ušetřili RAM.
- Flyweight ukazuje, jak vytvořit spoustu malých objektů, zatímco **Facade** ukazuje, jak vytvořit jediný objekt, který reprezentuje celý podsystém.
- Flyweight by mohl připomínat **Singleton**, pokud byste dokázali redukovat všechny sdílené stavy objektů na jeden flyweight objekt. Ale existují dva zásadní rozdíly mezi těmito vzory:
 - U **Singletonu** by měla existovat pouze jedna instance, zatímco třída Flyweight může mít více instancí s různými intrinsic stavy.
 - Objekt Singleton může být **mutable** (měnitelný). Flyweight objekty jsou **immutable** (neměnné).

PŘÍKLAD

NÁVRHOVÉ VZORY

Přednáška 12

SHRNUTÍ

- Probraná téma:
 - Čistý kód
 - SOLID, KISS, Antipatterny
 - Návrhové vzory
 - Refactoring a návrh struktury softwarového díla

REFACTORING

- Clean code (čistý kód)
- Hlavním cílem refactoringu je boj s technickým dluhem. Refactoring přeměňuje nepořádek na čistý kód a jednoduchý design.
- **Čistý kód je pro ostatní programátory srozumitelný.**
 - A nemáme na mysli extrémně sofistikované algoritmy. Špatné pojmenování proměnných, nafouklé třídy a metody, magická čísla — to vše dělá kód chaotickým a těžko pochopitelným.
- **Čistý kód neobsahuje duplicity.**
 - Kdykoli potřebujete změnit duplicitní kus kódu, musíte si pamatovat, že musíte provést stejnou změnu ve všech jeho výskyttech. To zvyšuje kognitivní zátěž a zpomaluje vývoj.
- **Čistý kód obsahuje minimální počet tříd a dalších „pohyblivých částí“.**
 - Méně kódu znamená méně věcí, které musíte udržovat v hlavě. Méně kódu znamená méně údržby. Méně kódu znamená méně chyb. Kód je závazek — držte ho krátký a jednoduchý.
- **Čistý kód prochází všemi testy.**
 - Víte, že máte špinavý kód, když projde jen 95 % testů. A víte, že jste v háji, když je pokrytí testy 0 %.
- **Čistý kód je snazší a levnější na údržbu!**
 - Pokud chceme, mohu také přeložit pokračování, nebo připravit příklad kódu ukazující refactoring do čistší podoby pomocí návrhových vzorů.

TECHNICKÝ DLUH

- Každý programátor se snaží psát vynikající kód hned od začátku. Pravděpodobně neexistuje vývojář, který by úmyslně psal nečistý kód na úkor projektu. Ale kdy se vlastně čistý kód stává nečistým?
- Metafora „**technického dluhu**“ ve vztahu k nečistému kódu byla původně navržena **Wardem Cunninghamem**.
- Pokud si někdo vezme půjčku v bance, umožní mu to rychleji realizovat nákupy. Za toto urychlení však zaplatí – nejenže musí splatit jistinu, ale také úroky navíc. Není nutné dodávat, že úroky mohou narůst natolik, že mohou převýšit celkový příjem, a tím učinit splacení nemožným.
- Úplně stejná situace může nastat u kódu. Je možné dočasně zrychlit vývoj tím, že se nenapíšou testy pro nové funkce, ale tato strategie postupně zpomalí další práci, dokud nebude dluh splacen – například právě dopsáním testů.

PŘÍČINY TECHNICKÉHO DLUHU

- **Tlak byznysu**
 - Někdy obchodní okolnosti donutí tým nasadit nové funkce dříve, než jsou zcela dokončené. V takovém případě se v kódu mohou začít objevovat různé provizoria a improvizace, aby se zakryly nedokončené části projektu.
- **Nedostatečné porozumění následkům technického dluhu**
 - Někteří zaměstnavatelé nemusí rozumět tomu, že technický dluh má „úrok“, který zpomaluje vývoj, jak dluh narůstá. Kvůli tomu může být složité alokovat čas týmu na refactoring, protože vedení nevidí jeho okamžitou hodnotu.
- **Neschopnost bojovat proti silné provázanosti komponent**
 - Tato situace nastává, když projekt připomíná monolit místo souboru jednotlivých modulů. V takovém případě má změna jedné části dopad na části ostatní. Týmová práce se stává obtížnou, protože není možné jednoduše izolovat práci jednotlivých vývojářů.

PŘÍČINY TECHNICKÉHO DLUHU II

- **Nedostatek testů**

- Absence okamžité odezvy podporuje rychlá, ale riskantní řešení. V nejhorších případech mohou být tyto změny nasazeny přímo do produkce bez řešení. Následky mohou být katastrofální – třeba zdánlivě neškodný hotfix může odeslat testovací e-mail tisícům zákazníků, nebo ještě hůře, poškodit celou databázi.

- **Nedostatek dokumentace**

- To zpomaluje začlenění nových členů týmu a může zcela zastavit vývoj, pokud z projektu odejdou klíčoví lidé.

- **Nedostatečná komunikace mezi členy týmu**

- Pokud není znalostní báze sdílena napříč firmou, lidé pracují se zastaralým porozuměním projektových procesů a informací. Situace se může ještě zhoršit, pokud jsou juniorskí vývojáři špatně vedeni mentory.

- **Dlouhodobý paralelní vývoj v několika větvích**

- To může vést k akumulaci technického dluhu, který se po sloučení změn ještě zvýší. Cím více se změny provádějí izolovaně, tím větší bude výsledný dluh.

PŘÍČINY TECHNICKÉHO DLUHU III

- **Odkládání refactoringu**
 - Požadavky na projekt se neustále vyvíjejí a v určitém okamžiku může být zřejmé, že části kódu jsou zastaralé a je třeba je přepracovat. Na druhou stranu vývojář každý den doplňuje nový kód, který na těchto zastaralých částech závisí. Čím déle se refactoring odkládá, tím více kódu bude potřeba v budoucnu přepracovat.
- **Nedodržování společných standardů**
 - To se stává, pokud každý člen týmu píše kód podle sebe – například stejným způsobem, jako psal při předchozím projektu.
- **Inkompetence**
 - To nastává, když vývojář jednoduše neví, jak psát kvalitní a udržovatelný kód.

KDY REFAKTOROVAT?

- Pravidlo tří
 - Když něco děláte poprvé, prostě to udělejte. Když děláte něco podobného podruhé, zakruťte se nevolí, že to musíte zopakovat, ale udělejte to stejně. Když něco děláte potřetí, začněte refaktorovat.
- Při přidávání funkce
 - Refactoring pomáhá pochopit kód ostatních. Pokud se musíte vypořádat s cizím nepořádným kódem, snažte se jej nejdřív refaktorovat. Čistý kód se mnohem lépe chápe. Zlepšíte ho nejen sami pro sebe, ale i pro ty, kteří jej budou používat po vás. Refactoring také usnadňuje přidávání nových funkcí. Změny se mnohem snáz provádějí v čistém kódu.

KDY REFAKTOROVAT? II

- Při opravě chyby
 - Chyby v kódu se chovají stejně jako v reálném světě: žijí v nejtemnějších, nejšpinavějších koutech kódu. Uklidíte-li kód, chyby se prakticky odhalí samy. Manažeři oceňují proaktivní refaktorování, protože eliminuje potřebu vytvářet speciální úkoly na refactoring později. Šťastní šéfové dělají šťastné programátory!
- Během code review
 - Code review může být poslední šancí uklidit kód předtím, než se stane veřejně přístupným. Nejlepší je provádět takové kontroly ve dvojici s autorem kódu. Tak můžete rychle opravit jednoduché problémy a zároveň odhadnout čas na opravu těch složitějších.

JAK REFAKTOROVAT?

- Refactoring by měl být prováděn jako série malých změn, z nichž každá existující kód nepatrně zlepší, přičemž program zůstává stále funkční.

KONTROLNÍ SEZNAM SPRÁVNĚ PROVEDENÉHO REFAKTORINGU

- **Kód by měl být čistší.**
 - Pokud kód zůstane po refactoringu stejně nečitelný... pak bohužel právě promarnili hodinu života. Je potřeba zjistit, proč k tomu došlo.
 - To se často stává, když se od ustálených malých kroků odkloní a smíchá se mnoho změn do jedné velké úpravy, ve které se lze snadno ztratit – zvláště pokud je málo času.
 - Může se to stát i u extrémně špatného kódu. Ať se vylepší sebevíc, celek zůstane nepořádek.
 - V takovém případě stojí za to zvážit úplné přepsání části kódu. Ale až poté, co jsou napsané testy a vyhrazena dostatečně velká časová rezerva. Jinak se dostaví výsledky, o kterých byla řeč v prvním odstavci.

KONTROLNÍ SEZNAM SPRÁVNĚ PROVEDENÉHO REFAKTORINGU

- **Během refactoringu by neměla vznikat nová funkctionalita.**
 - Refactoring a vývoj nových funkcí by se neměly míchat. Tyto procesy je vhodné oddělit alespoň na úrovni jednotlivých commitů.

KONTROLNÍ SEZNAM SPRÁVNĚ PROVEDENÉHO REFAKTORINGU

- **Všechny existující testy musí po refactoringu projít.**
 - Nastávají dvě situace, kdy testy po refactoringu selžou:
 - 1. Při refactoringu nastala chyba.**
Zřejmé řešení: chybu opravit.
 - 2. Testy byly příliš nízkoúrovňové.**
Například testovaly privátní metody tříd.
 - V takovém případě jsou na vině samotné testy. Je potřeba je zrefaktorovat nebo napsat novou, vyšší vrstvu testů.
 - Skvělým způsobem, jak se takové situaci vyhnout, je psát testy ve stylu BDD (Behavior-driven development).

TO JE VŠE ☺

ZKOUŠKA

- Témata:
 - Odhalení prohřešků v existujícím kódu
 - Refactoring existujícího kódu
 - Návrh struktury SW
- Jak bude zkouška probíhat?