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 znovu 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 tuto krabici.
- Pokud by jeden z těchto předmětů byla menší krabice, tato 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 (nadtřídy). Tento vztah je tzv. „**is-a**“ vztah, kdy podtřída je konkrétnější instancí nadtřídy.
- 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

```java
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 demonstruje 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 navrhnout 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 refaktoring 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

```java
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.
- **Vyhnout 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 náhradu 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 znovu 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