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, zaškrtávací 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 zkontrolovat a ověřit hodnoty všech polí.
- Pokud by tato 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 zaškrtávacího 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 zkontrolovat 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.
  Čím 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, **aniž 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 znovu 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 **Facade**.
- 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 extrahovat 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 znovu 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 funkcionalitu. 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 rekurzi 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 uzlů. Tvrdil, 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 uzlů. 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 znovu 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 verzí 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 předefinovat 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 souvisel 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 otevírání/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 třídy, 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 nadtřídy.

- 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 obsahují.

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