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 stavy.
- 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 podmínkách se projeví tehdy, jakmile začnete do třídy Document přidávat více stavů a více stavově záviselného chování.
- Většina metod začne obsahovat monstruózní 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 stavy 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 stavy **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 stavy se často mění.
  - Vzor navrhuje přesunout veškerý kód související se stavy do samostatných tříd. Díky tomu můžete přidávat nové stavy 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 jejích 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 stavy a přechody v podmínkami řízeném stavovém stroji.
  - Tento vzor vám umožní vytvořit hierarchie 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 umožňuje.
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 stavy, aniž byste měnili stávající 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 stavy 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 stavy, 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 přijímalo 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 znovu 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 reference 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 nevzhledně.

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 funkcionalitu 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 funkcionality.
- Například aplikace, která nahrává krátká zábavná videa s kočkami na sociální sítě, by mohla potenciálně používat profesionální knihovnu pro konverzi videa.
- Nicméně vše, co opravdu potřebuje, je třída s jedinou metodou encode(filename, format). Po vytvoření takové třídy a jejím propojení s knihovnou pro konverzi videa 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 znovu 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 podsystém nabízí. Jste na správné cestě, pokud toto rozhraní činí klientský kód nezávislým na mnoha třídách podsystému.
- Deklarujte a implementujte toto rozhraní v nové třídě fasády. Fasáda by měla přesměrovat volání z klientského kódu na příslušné objekty podsystému. Fasáda by měla být zodpovědná za inicializaci podsystému a správu jeho životního cyklu, pokud to klientský kód již nedělá.
- Pro plné využití vzoru zajistěte, aby veškerý klientský kód komunikoval s podsystémem pouze prostřednictvím fasády. Klientský kód je nyní chráněn před jakýmikoli změnami v kódu podsystému. Například pokud se podsystém 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 podsystémem objektů.
- **Abstract Factory** může sloužit jako alternativa k Fasádě, pokud chcete jen skrýt způsob, jakým jsou objekty podsystému **vytvářeny**, před klientským kódem.
- **Flyweight** ukazuje, jak vytvořit mnoho malých objektů, zatímco Fasáda ukazuje, jak vytvořit jeden objekt, který reprezentuje celý podsystém.

- 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 funkcionalitu. 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 jediný 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