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 rozbít 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 znovu použít několik existujících podtříd, které postrádají určitou společnou funkcionalitu, kterou nelze přidat do jejich předka.
- Mohli byste rozšířit každou podtřídu a doplnit chybějící funkcionalitu do nových potomků. Museli byste však duplikovat kód ve všech těchto nových třídách, což je velmi špatná praxe.
- Mnohem elegantnějším řešením je vložit chybějící funkcionalitu do třídy adaptéru. Poté můžete objekty s chybějícími funkcemi obalit adaptérem a dynamicky tak získat potřebnou funkcionalitu. Aby to fungovalo, musí mít cílové třídy společné rozhraní a atribut (pole) adaptéru 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**. Obvyklá praxe je inicializovat toto 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 adaptery, 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 podsystémem 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; také 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 dimenzí 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 těchto 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 toto 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 hraní 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 rychlost, 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 znovu 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 zaveďte nový parametr a používejte 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 jediný 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