# 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 **filtrovala 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 zpracovatelé (handlery).
- 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 handlery 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 handlery, které mají možnost jej zpracovat.
- A tady přichází ta nejlepší 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 handlery 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 handleru**, 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 **přijmout 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 příjemců 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 trasy.
- 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 contextu 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 neřeší, 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 umožňuje 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 tolik 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 extrahujete 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 contextu.

## 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 poli 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 stavy, což jim umožňuje měnit stav contextu podle libosti.

# PŘÍKLAD