# NÁVRHOVÉ VZORY

Přednáška 7

# BEHAVIORAL PATTERNS

- ~~Chain of Responsibility~~
- Command
- Iterator
- Mediator
- Memento
- ~~Observer~~
- State
- ~~Strategy~~
- Template Method
- Visitor

# ITERATOR

- Iterator je behaviorální návrhový vzor, který Vám umožňuje procházet prvky kolekce, aniž byste odhalovali její vnitřní reprezentaci (např. seznam, zásobník, strom atd.).

# K ČEMU SE HODÍ?

- Kolekce patří mezi nejčastěji používané datové typy v programování.
- Přesto je kolekce jen kontejner pro skupinu objektů.
- Většina kolekcí uchovává své prvky v jednoduchých seznamech.
- Některé jsou však založeny na zásobnících, stromech, grafech a dalších složitějších datových strukturách.
- Bez ohledu na to, jak je kolekce strukturovaná, musí poskytovat způsob přístupu k jejím prvkům, aby je mohl využívat jiný kód.
- Měl by existovat způsob, jak projít každý prvek kolekce, aniž byste opakovaně přistupovali ke stejným prvkům.

- To může znít jako jednoduchý úkol, pokud máte kolekci založenou na seznamu – stačí projít všechny prvky cyklem.
- Ale jak sekvenčně procházet prvky složité datové struktury, například stromu?
- Například jednoho dne vám může stačit procházení stromu do hloubky (depth-first traversal).
- Další den můžete potřebovat procházení do šířky (breadth-first traversal).
- A další týden třeba náhodný přístup k prvkům stromu.
- Přidávání stále více algoritmů pro procházení do kolekce postupně rozmazává její primární odpovědnost, kterou je efektivní ukládání dat.
- Navíc některé algoritmy mohou být přizpůsobeny specifické aplikaci, takže jejich začlenění do generické třídy kolekce by bylo neobvyklé.
- Na druhou stranu klientský kód, který má pracovat s různými kolekcemi, se nemusí vůbec zajímat o to, jak kolekce uchovávají své prvky.
- Protože však různé kolekce poskytují různé způsoby přístupu k prvkům, nemáte jinou možnost, než svázat svůj kód s konkrétními třídami kolekcí.

# ŘEŠENÍ

- Hlavní myšlenkou vzoru Iterator je extrahovat chování procházení kolekce do samostatného objektu, nazývaného iterator.
- Kromě implementace samotného algoritmu objekt iterator zapouzdřuje všechny detaily procházení, například aktuální pozici a počet zbývajících prvků do konce.
- Díky tomu může několik iteratorů současně procházet stejnou kolekci, nezávisle na sobě.
- Obvykle iterátory poskytují jednu hlavní metodu pro získávání prvků kolekce. Klient může tuto metodu volat opakovaně, dokud nevrátí žádný prvek, což znamená, že iterator prošel všechny prvky.
- Všechny iterátory musí implementovat stejné rozhraní.
- To činí klientský kód kompatibilní s libovolným typem kolekce nebo algoritmem procházení, pokud je k dispozici vhodný iterátor.
- Pokud potřebujete speciální způsob procházení kolekce, stačí vytvořit novou třídu iteratoru, aniž byste museli měnit kolekci nebo klientský kód.

# VYUŽITÍ

- Použijte vzor Iterator, pokud vaše kolekce obsahuje složitou datovou strukturu, ale chcete skrýt její složitost před klienty (ať už z důvodu pohodlí nebo bezpečnosti).
- Iterator zapouzdřuje detaily práce se složitou datovou strukturou a klientovi poskytuje několik jednoduchých metod pro přístup k prvkům kolekce. Tento přístup je velmi pohodlný pro klienta a zároveň chrání kolekci před neopatrnými nebo škodlivými akcemi, které by klient mohl provést při přímé práci s kolekcí.
- Vzor použijte také k snížení duplikace kódu procházení napříč vaší aplikací.
- Kód pro ne-triviální algoritmy procházení bývá často velmi rozsáhlý. Pokud je umístěn přímo v obchodní logice aplikace, může rozmazat odpovědnost původního kódu a snížit jeho udržovatelnost. Přesunutím kódu procházení do určených iteratorů můžete učinit kód aplikace čistším a přehlednějším.
- Použijte vzor Iterator, pokud chcete, aby váš kód procházel různé datové struktury, nebo pokud jsou typy těchto struktur předem neznámé.
- Vzor poskytuje několik generických rozhraní pro kolekce i iterátory. Pokud váš kód tato rozhraní používá, bude fungovat i s různými typy kolekcí a iterátorů, které tato rozhraní implementují.

# JAK IMPLEMENTOVAT?

1. Deklarujte rozhraní iteratoru. Minimálně by mělo obsahovat metodu pro získání dalšího prvku kolekce. Pro větší pohodlí můžete přidat i několik dalších metod, například získání předchozího prvku, sledování aktuální pozice nebo kontrolu konce iterace.
2. Deklarujte rozhraní kolekce a definujte metodu pro získání iteratoru. Návratový typ by měl odpovídat typu iteratoru. Podobné metody můžete deklarovat i v případě, že plánujete mít několik odlišných skupin iteratorů.
3. Implementujte konkrétní třídy iteratorů pro kolekce, které chcete zpřístupnit procházení pomocí iteratorů. Objekt iteratoru musí být propojen s jedinou instancí kolekce. Obvykle se toto propojení zajišťuje přes konstruktor iteratoru.
4. Implementujte rozhraní kolekce ve svých třídách kolekcí. Hlavní myšlenkou je poskytnout klientovi zkratku pro vytvoření iteratoru, přizpůsobeného konkrétní třídě kolekce. Objekt kolekce musí předat sám sebe konstruktoru iteratoru, aby bylo propojení mezi nimi zajištěno.
5. Projděte klientský kód a nahraďte všechny části kódu procházení kolekce použitím iteratorů. Klient získává nový objekt iteratoru pokaždé, když potřebuje projít prvky kolekce.

# PRO A PROTI

## Pro

- Princip jediné odpovědnosti (Single Responsibility Principle): Můžete učinit klientský kód a kolekce přehlednějšími tím, že rozsáhlé algoritmy procházení extrahujete do samostatných tříd.
- Princip otevřenosti/uzavřenosti (Open/Closed Principle): Můžete implementovat nové typy kolekcí a iteratorů a předávat je stávajícímu kódu, aniž byste cokoli porušili.
- Můžete procházet stejnou kolekci paralelně, protože každý objekt iteratoru obsahuje svůj vlastní stav iterace.
- Ze stejného důvodu můžete iteraci odložit a pokračovat v ní později, kdy je potřeba.

## Proti

- Použití vzoru může být přehnané, pokud vaše aplikace pracuje pouze s jednoduchými kolekcemi.
- Použití iteratoru může být méně efektivní, než přímo procházet prvky některých specializovaných kolekcí.

# VZTAHY S DALŠÍMI VZORY

- Iterator můžete použít pro procházení stromů Composite.
- Můžete použít Factory Method společně s iterátorem, aby podtřídy kolekcí vracely různé typy iteratorů, které jsou kompatibilní s kolekcemi.
- Můžete použít Memento spolu s iterátorem k zachycení aktuálního stavu iterace a jeho případnému vrácení zpět, pokud to bude potřeba.
- Můžete použít Visitor společně s iterátorem k procházení složité datové struktury a provádění nějaké operace nad jejími prvky, i když všechny mají různé třídy.

# PŘÍKLAD

# COMMAND

- Command (Příkaz) je behaviorální návrhový vzor, který převádí požadavek (request) na samostatný objekt, jenž obsahuje všechny potřebné informace o daném požadavku.
- Tato transformace umožňuje:
  - předávat požadavky jako argumenty metod,
  - odkládat nebo řadit jejich vykonání do fronty,
  - a také podporovat operace, které lze vrátit zpět (undo).

# CO ŘEŠÍ?

- Představte si, že pracujete na nové aplikaci textového editoru. Vaším aktuálním úkolem je vytvořit panel nástrojů s řadou tlačítek pro různé operace editoru.Vytvořili jste elegantní třídu Button, kterou lze použít jak pro tlačítka na panelu nástrojů, tak i pro běžná tlačítka v různých dialogových oknech.
- Ačkoli všechna tato tlačítka vypadají podobně, mají dělat různé věci. Kam ale umístit kód, který se má vykonat po kliknutí na jednotlivá tlačítka?
- Nejjednodušším řešením by bylo vytvořit mnoho podtříd — pro každé použití tlačítka jednu. Tyto podtřídy by obsahovaly kód, který se má spustit při kliknutí.
- Brzy si ale uvědomíte, že tento přístup je velmi problematický:
  - Máte obrovské množství podtříd.
  - Každá změna v základní třídě Button může snadno rozbít kód v těchto podtřídách.
  - Jinými slovy, váš GUI kód se stává závislým na proměnlivém kódu business logiky.

- A teď to nejhorší: některé operace (např. kopírování/vkládání textu) musí být vyvolány z více míst.
- Uživatel může například: kliknout na malé tlačítko „Kopírovat“ na panelu nástrojů, použít kontextové menu,nebo stisknout Ctrl+C na klávesnici.
- Zpočátku, když aplikace obsahovala pouze panel nástrojů, bylo v pořádku umístit implementaci jednotlivých operací přímo do podtříd tlačítek — například kód pro kopírování textu do třídy CopyButton.
- Jakmile však přidáte kontextová menu, klávesové zkratky a další ovládací prvky, musíte buď duplikovat kód operace v mnoha třídách, nebo — což je ještě horší — učinit menu závislými na tlačítkách.

# ŘEŠENÍ

- Dobrý návrh softwaru je často založen na principu oddělení odpovědností (separation of concerns), což obvykle vede k rozdělení aplikace do vrstev.
- Nejčastějším příkladem je rozdělení na:
  - vrstvu grafického uživatelského rozhraní (GUI)
  - vrstvu obchodní logiky.
- Vrstva GUI je zodpovědná za zobrazení přehledného a atraktivního rozhraní, zachytávání vstupu uživatele a zobrazování výsledků toho, co uživatel a aplikace dělají.
- Když však dojde na něco důležitého — například výpočet dráhy Měsíce nebo vytváření roční zprávy —, GUI vrstva předává práci podkladové vrstvě business logiky.
- V kódu to může vypadat takto: objekt GUI volá metodu objektu business logiky a předává jí argumenty.Tento proces se obvykle popisuje jako odeslání požadavku z jednoho objektu na druhý.

- Návrhový vzor Command navrhuje, aby objekty GUI neodesílaly tyto požadavky přímo. Místo toho byste měli vytáhnout všechny detaily požadavku — například objekt, na který je volána metoda, název metody a seznam argumentů — do samostatné třídy příkazu (command) s jedinou metodou, která tento požadavek spustí.
- Objekty Command slouží jako spojovací článek mezi různými objekty GUI a objekty obchodní logiky.
- Od této chvíle objekt GUI nemusí vědět, který objekt obchodní logiky požadavek obdrží a jak bude zpracován.
- Objekty GUI jednoduše spustí příkaz, který se postará o všechny detaily.

- Vraťme se k našemu textovému editoru. Po použití návrhového vzoru Command již nepotřebujeme všechny ty podtřídy tlačítek, aby implementovaly různé chování po kliknutí.
- Stačí vložit jedno pole do základní třídy Button, které bude uchovávat odkaz na objekt příkazu (Command), a nechat tlačítko tento příkaz vykonat při kliknutí.
- Pro každou možnou operaci implementujete sadu tříd příkazů a přiřadíte je konkrétním tlačítkům podle toho, jaké chování mají tlačítka vykonávat.
- Ostatní prvky GUI, jako jsou menu, klávesové zkratky nebo celé dialogy, mohou být implementovány stejným způsobem. Budou propojeny s příkazem, který se spustí, když uživatel s prvkem GUI interaguje. Jak jste si pravděpodobně už všimli, prvky související se stejnou operací budou propojeny se stejnými příkazy, což zabraňuje duplicitě kódu.
- Výsledkem je, že příkazy tvoří praktickou střední vrstvu, která snižuje závislost mezi vrstvou GUI a vrstvou business logiky. A to je jen malá část výhod, které návrhový vzor Command nabízí!

# POUŽITÍ

- Použijte návrhový vzor Command, pokud chcete parametrizovat objekty operacemi.
- Návrhový vzor Command může převést konkrétní volání metody na samostatný objekt. Tato změna otevírá řadu zajímavých možností: můžete příkazy předávat jako argumenty metod, ukládat je do jiných objektů, měnit propojené příkazy za běhu aplikace atd.Příklad: vyvíjíte komponentu GUI, například kontextové menu, a chcete, aby uživatelé mohli konfigurovat položky menu, které spustí operace, když uživatel klikne na položku.
- Použijte vzor Command, pokud chcete řadit operace do fronty, plánovat jejich vykonání nebo je provádět vzdáleně.Stejně jako jakýkoli jiný objekt, může být příkaz serializován, což znamená, že lze převést na řetězec, který je snadno zapisovatelný do souboru nebo databáze. Později lze tento řetězec obnovit jako původní objekt příkazu. Tímto způsobem můžete odložit nebo naplánovat vykonání příkazu. A to není vše! Stejným způsobem můžete příkazy řadit do fronty, logovat nebo odesílat po síti.

- Použijte vzor Command, pokud chcete implementovat vratné operace (undo/redo).Ačkoli existuje mnoho způsobů, jak implementovat undo/redo, návrhový vzor Command je pravděpodobně nejpopulárnější.
- Pro možnost vrácení operací je potřeba implementovat historii provedených příkazů. Historie příkazů je zásobník obsahující všechny provedené příkazy spolu s příslušnými zálohami stavu aplikace.
- Tato metoda má dvě nevýhody: Uložení stavu aplikace není snadné, protože část stavu může být soukromá. Tento problém lze zmírnit použitím vzoru Memento. Zálohy stavu mohou spotřebovat hodně paměti RAM. Proto někdy můžete použít alternativní přístup: místo obnovy minulého stavu příkaz provede opačnou operaci. Tato reverzní operace má také svou cenu — může být obtížná nebo dokonce nemožná k implementaci.

# JAK IMPLEMENTOVAT?

- Deklarujte rozhraní příkazu (Command interface) s jedinou metodou pro vykonání.
- Začněte extrahovat požadavky do konkrétních tříd příkazů, které implementují toto rozhraní. Každá třída musí obsahovat sadu polí pro uchování argumentů požadavku spolu s odkazem na skutečný objekt příjemce (receiver). Všechny tyto hodnoty musí být inicializovány přes konstruktor příkazu.
- Identifikujte třídy, které budou působit jako odesílatelé (senders). Přidejte do těchto tříd pole pro uchování příkazů. Odesílatelé by měli komunikovat se svými příkazy pouze přes rozhraní příkazu.
- Odesílatelé obvykle nevytvářejí objekty příkazů sami, ale dostávají je od klientského kódu. Změňte odesílatele tak, aby vykonávali příkaz, místo aby odesílali požadavek přímo příjemci.
- Klient by měl inicializovat objekty v následujícím pořadí:
  - Vytvořit příjemce (receivers).
  - Vytvořit příkazy a přiřadit je příjemcům, pokud je to potřeba.
  - Vytvořit odesílatele (senders) a přiřadit jim konkrétní příkazy.

# PROROVNÁNÍ

## Pro

- Princip jediné odpovědnosti (Single Responsibility Principle): Můžete oddělit třídy, které volají operace, od tříd, které tyto operace provádějí.
- Princip otevřenosti/uzavřenosti (Open/Closed Principle): Můžete přidávat nové příkazy do aplikace, aniž byste porušili existující klientský kód.
- Můžete implementovat undo/redo.
- Můžete implementovat odložené vykonání operací.
- Můžete složit sadu jednoduchých příkazů do jednoho komplexního příkazu.

## Proti

- Kód se může stát složitějším, protože zavádíte zcela novou vrstvu mezi odesílateli a příjemci.

# VZTAH S DALŠÍMI VZORY

- **Chain of Responsibility, Command, Mediator a Observer** řeší různé způsoby propojení odesílatelů a příjemců požadavků:
- **Chain of Responsibility** předává požadavek sekvenčně podél dynamického řetězce potenciálních příjemců, dokud jej některý z nich nezpracuje.
- **Command** vytváří **jednosměrná propojení** mezi odesílateli a příjemci.
- **Mediator** eliminuje přímá propojení mezi odesílateli a příjemci, nutí je komunikovat **nepřímo přes objekt mediátora**.
- **Observer** umožňuje příjemcům **dynamicky se přihlašovat a odhlašovat** k přijímání požadavků.
- **Handlery v Chain of Responsibility** lze implementovat jako **Commandy**. V takovém případě můžete nad stejným kontextovým objektem (reprezentovaným požadavkem) vykonat mnoho různých operací.
- Existuje však i jiný přístup, kdy je **požadavek sám o sobě objektem Command**. V tom případě můžete **vykonat stejnou operaci** v řadě různých kontextů propojených do řetězce.
- **Command a Memento** lze použít společně při implementaci **„undo“**. V takovém případě jsou příkazy zodpovědné za provedení různých operací nad cílovým objektem, zatímco mementa ukládají stav tohoto objektu těsně před vykonáním příkazu.
- **Command a Strategy** mohou vypadat podobně, protože oba umožňují **parametrizovat objekt nějakou akcí**. Mají však velmi odlišný záměr:
- **Command** převádí libovolnou operaci na objekt. Parametry operace se stávají poli objektu, což umožňuje odložené vykonání, řazení do fronty, uchovávání historie příkazů, odesílání příkazů na vzdálené služby atd.
- **Strategy** obvykle popisuje různé způsoby provedení téže činnosti a umožňuje **vyměňovat algoritmy** uvnitř jedné kontextové třídy.
- **Prototype** se hodí, pokud potřebujete ukládat **kopie příkazů do historie**.
- **Visitor** lze považovat za **rozšířenou verzi vzoru Command**. Jeho objekty mohou vykonávat operace nad různými objekty různých tříd.

# PŘÍKLAD

# MEMENTO

- Memento je návrhový vzor, který umožňuje uložit a obnovit předchozí stav objektu, aniž by odhalil detaily jeho vnitřní implementace.

# K ČEMU SE HODÍ?

- Představte si, že vytváříte aplikaci textového editoru. Kromě jednoduchého úprav textu může váš editor text také formátovat, vkládat obrázky apod.
- V určitém okamžiku se rozhodnete, že uživatelům umožníte vracet zpět jakékoli provedené operace. Tato funkce se během let stala natolik běžnou, že dnes už ji lidé očekávají v každé aplikaci.
- Pro implementaci zvolíte přímý přístup: před provedením jakékoli operace aplikace uloží stav všech objektů do nějakého úložiště.
- Později, když se uživatel rozhodne akci vrátit zpět, aplikace načte poslední snímek (snapshot) z historie a použije jej k obnovení stavu všech objektů.

- Zamysleme se nad těmito snímky stavů. Jak byste takový snímek vlastně vytvořili? Pravděpodobně by bylo potřeba projít všechna pole objektu a jejich hodnoty zkopírovat do úložiště. To by ovšem fungovalo jen v případě, že by objekt umožňoval volný přístup ke svým datům. Bohužel většina skutečných objektů to neumožňuje, protože ukrývá svá důležitá data v soukromých (private) polích.
- Ignorujme tento problém a předpokládejme, že naše objekty se chovají „jako hippies“ — mají otevřené vztahy a udržují svůj stav veřejný. Tento přístup by sice okamžitý problém vyřešil a umožnil vytvářet snímky stavů dle libosti, ale přináší vážné nevýhody.
- V budoucnu se totiž můžete rozhodnout refaktorovat některé třídy editoru, nebo přidat či odebrat některá pole. To by sice znělo jednoduše, ale zároveň by bylo nutné upravit i všechny třídy, které jsou zodpovědné za kopírování stavů dotčených objektů.

- Ale to není všechno. Zamysleme se nad samotnými **„snímky“ stavu editoru**. Jaká data vlastně obsahují? V tom nejzákladnějším případě by měly obsahovat **aktuální text, pozici kurzoru, aktuální posun (scroll)**, a podobné údaje.
- Aby bylo možné takový snímek vytvořit, bylo by potřeba tyto hodnoty **shromáždit a uložit** do nějakého kontejneru.
- S největší pravděpodobností budete chtít tyto kontejnery ukládat do nějakého **seznamu (historie)**, který bude představovat historii změn. Tyto kontejnery by tedy pravděpodobně byly **objekty jedné třídy**. Tato třída by měla jen minimum metod, ale **spoustu polí odpovídajících stavu editoru**.
- Aby ostatní objekty mohly data do snímku zapisovat nebo je z něj číst, musela by být tato pole **veřejná (public)**. Tím by se ale **odhalil celý vnitřní stav editoru**, včetně soukromých částí. Ostatní třídy by se tak staly **závislé na každé malé změně** ve třídě snímku — i na těch, které by se jinak děly jen uvnitř editoru bez vlivu na ostatní části programu.
- Vypadá to, že jsme se dostali do slepé uličky:
  - buď **odhalíme všechny vnitřní detaily tříd** a učiníme je tak velmi křehkými,
  - nebo **omezíme přístup k jejich stavu**, čímž znemožníme vytváření snímků.
- Existuje tedy vůbec nějaký jiný způsob, jak implementovat funkci **„Zpět“ (undo)**?

# ŘEŠENÍ

- Všechny problémy, které jsme právě popsali, jsou způsobeny **porušením zapouzdření (encapsulation)**. Některé objekty se snaží dělat víc, než by měly. Aby získaly potřebná data pro vykonání určité akce, **vstupují do soukromého prostoru jiných objektů**, místo aby nechaly tyto objekty provést akci samy.
- **Návrhový vzor Memento** svěřuje vytváření snímků stavu **samotnému vlastníkovi tohoto stavu –** tedy tzv. *originatoru* (původnímu objektu). Místo toho, aby se jiné objekty snažily kopírovat stav editoru „zvenčí“, může **třída editoru** sama vytvořit snímek, protože má **plný přístup ke svému vlastnímu stavu**.
- Vzor navrhuje ukládat kopii stavu objektu do **speciálního objektu zvaného *memento***. Obsah tohoto mementa **není přístupný žádnému jinému objektu** kromě toho, který ho vytvořil. Ostatní objekty mohou s mementy komunikovat pouze přes **omezené rozhraní**, které může například umožnit získání **metadat snímku** (čas vytvoření, název provedené operace apod.), ale **ne samotný uložený stav původního objektu**.

- Takto přísná pravidla umožňují **ukládat mementa do jiných objektů**, které se obvykle nazývají **správci (caretakers)**. Protože správce pracuje s mementem **pouze prostřednictvím omezeného rozhraní**, **nemůže nijak manipulovat se stavem**, který je uvnitř mementa uložen.
  Zároveň však **originátor** má přístup ke všem polím uvnitř mementa, což mu umožňuje **obnovit svůj předchozí stav kdykoli potřebuje**.
- V našem příkladu s textovým editorem můžeme vytvořit **samostatnou třídu pro historii**, která bude působit jako správce. **Zásobník mement** uložený ve správci se bude rozrůstat pokaždé, když se editor chystá provést nějakou operaci. Tento zásobník lze dokonce **zobrazit v uživatelském rozhraní aplikace**, aby uživatel viděl **historii provedených akcí**.
- Když uživatel spustí **„Undo“ (zpět)**, historie vezme **nejnovější memento** ze zásobníku a předá ho zpět editoru s požadavkem na **vrácení změn**. Protože má editor **plný přístup k mementu**, může **obnovit svůj stav** podle hodnot uložených v tomto mementu.

# VYUŽITÍ

- Použijte **návrhový vzor Memento**, když chcete vytvářet **snímky stavu objektu**, abyste mohli **obnovit jeho předchozí stav**.
- Vzor Memento umožňuje **vytvořit úplné kopie stavu objektu**, včetně **soukromých polí**, a uložit je **odděleně od samotného objektu**. I když si většina lidí tento vzor pamatuje díky případu použití „undo“, je rovněž **nezbytný při práci s transakcemi** (např. pokud je potřeba v případě chyby operaci vrátit zpět).
- Použijte tento vzor, když **přímý přístup k polím/getterům/setterům objektu porušuje jeho zapouzdření**.
- Memento činí **objekt samotný zodpovědným za vytvoření snímku svého stavu**. Žádný jiný objekt **nemůže tento snímek číst**, což zajišťuje, že data stavu původního objektu jsou **bezpečná a chráněná**.

# JAK IMPLEMENTOVAT?

1. Určete, která třída bude hrát roli **originátora**. Je důležité vědět, zda program používá **jeden centrální objekt tohoto typu**, nebo **více menších objektů**.
2. Vytvořte třídu **Memento**. Postupně deklarujte **sadu polí**, která **odpovídají polím** deklarovaným uvnitř třídy originátora.
3. Udělejte třídu Memento **neměnnou (immutable)**. Memento by mělo přijmout data **jen jednou**, přes konstruktor. Třída **by neměla mít žádné settery**.
4. Pokud váš programovací jazyk podporuje **vnořené třídy**, vložte Memento dovnitř originátora. Pokud ne, **vytvořte prázdné rozhraní (interface)** z třídy Memento a nechte všechny ostatní objekty, aby na Memento odkazovaly přes toto rozhraní. Do rozhraní můžete přidat nějaké **operace s metadaty**, ale nic, co by **odhalovalo stav originátora**.
5. Přidejte do třídy originátora **metodu pro vytvoření mementa**. Originátor by měl předat svůj stav Mementu přes **jeden nebo více argumentů konstruktoru Mementa**.
   Návratový typ metody by měl být typu rozhraní, které jste v předchozím kroku vytvořili (pokud jste ho vytvořili). Interně by metoda pro tvorbu mementa měla **pracovat přímo s třídou Memento**.
6. Přidejte do třídy originátora **metodu pro obnovení stavu originátora**. Měla by přijímat **objekt Memento** jako argument. Pokud jste v předchozím kroku vytvořili rozhraní, použijte ho jako typ parametru. V tomto případě bude třeba **přetypovat příchozí objekt na třídu Memento**, protože originátor potřebuje **plný přístup k objektu**.
7. Správce (caretaker), ať už představuje **objekt příkazu, historii nebo něco úplně jiného**, by měl vědět, **kdy si vyžádat nové memento od originátora**, **jak ho uložit** a **kdy obnovit originátora s konkrétním mementem**.
8. Vazba mezi **správci a originátory** může být přesunuta do třídy Memento. V tomto případě musí být každé memento **spojeno s originátorem**, který ho vytvořil. Metoda pro obnovu stavu by se také přesunula do třídy Memento. To však dává smysl **pouze pokud je třída Memento vložena do originátora** nebo pokud třída originátora poskytuje **dostatečné settery pro přepsání svého stavu**.

# PRO A PROTI

## Pro

- Můžete vytvářet **snímky stavu objektu**, aniž byste porušili jeho zapouzdření.
- Můžete **zjednodušit kód originátora** tím, že necháte správce (caretaker) spravovat historii stavů originátora.

## Proti

- Aplikace může spotřebovat hodně paměti RAM, pokud klienti vytvářejí mementa příliš často.
- Správci (caretakers) by měli sledovat životní cyklus originátora, aby mohli zničit zastaralá mementa.
- Většina dynamických programovacích jazyků, jako PHP, Python a JavaScript, nemůže zaručit, že stav uvnitř mementa zůstane nedotčený.

# VZTAHY S JINÝMI VZORY

- Můžete **použít Command a Memento společně** při implementaci funkce „undo“. V tomto případě jsou příkazy zodpovědné za provádění různých operací nad cílovým objektem, zatímco mementa ukládají stav tohoto objektu těsně před vykonáním příkazu.
- Můžete **použít Memento spolu s Iterator** k zachycení aktuálního stavu iterace a jeho případnému obnovení.
- Někdy může být **Prototype jednodušší alternativou k Memento**. To funguje, pokud je objekt, jehož stav chcete uložit do historie, poměrně jednoduchý a nemá odkazy na externí zdroje, nebo jsou tyto odkazy snadno znovu navázatelné.

# PŘÍKLAD