NÁVRHOVÉ VZORY

Přednáška 2 – Factory, Abstract Factory, Singleton

CREATIONAL DESIGN PATTERNS

- Creational Design Patterns poskytují různé mechanismy pro vytváření objektů, které zvyšují flexibilitu a umožňují znovupoužití existujícího kódu

FACTORY

- Factory Method je kreacionální návrhový vzor, který poskytuje rozhraní pro vytváření objektů v nadtřídě, ale umožňuje podtřídám upravit typ objektů, které budou vytvořeny.
- Jinými slovy: nadtřída definuje „jak“ se objekt vytvoří, ale konkrétní podtřída rozhoduje „jaký“ objekt to bude.

FACTORY

- Představ me si, že vytváříme aplikaci pro správu logistiky.
- První verze naší aplikace dokáže zpracovávat pouze přepravu pomocí nákladních vozidel, takže většina kódu se nachází ve třídě Truck.
- Po nějaké době se naše aplikace stane poměrně populární.
- Každý den začínáš dostávat desítky žádostí od námořních přepravních společností, které chtějí, aby aplikace podporovala i námořní logistiku.

- Skvělé zprávy, že? Ale co kód?
- V současnosti je většina našeho kódu pevně svázaná s třídou Truck.
- Přidání lodí (Ship) do aplikace by vyžadovalo úpravy napříč kódem.
- A pokud se později rozhodneme přidat další typ dopravy, budeme pravděpodobně muset všechny tyto změny opakovat znovu.
- Výsledkem bude poměrně nepřehledný kód, plný podmínek, které mění chování aplikace podle toho, jaký typ dopravního objektu se právě používá.

FACTORY

- Řešení:
- Návrhový vzor Factory Method navrhuje, abysme místo přímého vytváření objektů pomocí operátoru new používali volání speciální tovární metody.
- Nemějte obavy – objekty se stále vytvářejí pomocí new, ale toto volání probíhá uvnitř tovární metody.
- Objekty, které tato metoda vrací, se často označují jako produkty.

FACTORY

- Na první pohled se tato změna může zdát zbytečná: pouze jsme přesunuli volání konstruktoru z jedné části programu do jiné.
- Ale zamysleme se nad tím: nyní můžeme překrýt tovární metodu v podtřídě a změnit typ produktů, které tato metoda vytváří.
- Existuje však drobné omezení: podtřídy mohou vracet různé typy produktů pouze tehdy, pokud tyto produkty mají společného otce nebo rozhraní.
- Tovární metoda v otci by navíc měla mít návratový typ deklarovaný právě jako toto rozhraní.

FACTORY

- Například třídy Truck a Ship by měly implementovat rozhraní Transport, které deklaruje metodu s názvem deliver.
- Každá třída tuto metodu implementuje odlišně: nákladní auta doručují náklad po souši, zatímco lodě doručují náklad po moři.
- Tovární metoda ve třídě RoadLogistics vrací objekty typu Truck, zatímco tovární metoda ve třídě SeaLogistics vrací objekty typu Ship.
- Kód, který používá tovární metodu (často označovaný jako klientský kód), nevidí rozdíl mezi konkrétními produkty, které vracejí různé podtřídy.
- Klient zachází se všemi produkty jako s abstraktním typem Transport.
- Klient ví, že všechny dopravní objekty mají mít metodu deliver, ale jak přesně funguje, ho nezajímá.

FACTORY – KDY POUŽÍT

- Použij návrhový vzor Factory Method, když dopředu nevíš přesné typy a závislosti objektů, se kterými má tvůj kód pracovat.
- Factory Method odděluje kód pro vytváření produktů od kódu, který produkty skutečně používá. Díky tomu je snazší rozšiřovat logiku vytváření objektů nezávisle na zbytku aplikace.
- Například: pokud chceš do aplikace přidat nový typ produktu, stačí vytvořit novou podtřídu tvůrce a překrýt tovární metodu.
- Použij Factory Method také tehdy, když chceš uživatelům své knihovny nebo frameworku umožnit rozšíření jeho vnitřních komponent.
- Dědičnost je pravděpodobně nejjednodušší způsob, jak rozšířit výchozí chování knihovny nebo frameworku. Ale jak framework pozná, že má použít právě tvou podtřídu místo standardní komponenty?
- Řešením je centralizovat kód pro vytváření komponent do jediné tovární metody a umožnit komukoliv tuto metodu překrýt, spolu s rozšířením samotné komponenty.

FACTORY – KDY POUŽÍT

- Představme si, že vytváříme aplikaci pomocí open-source UI frameworku.
- Naše aplikace má mít kulatá tlačítka, ale framework nabízí pouze hranatá.
- Rozšíříme tedy standardní třídu Button a vytvoříme podtřídu RoundButton.
- Teď ale potřebujeme frameworku říct, že má používat tvou novou třídu místo výchozí.
- Řešení:
- Vytvoříme podtřídu UIWithRoundButtons z hlavní třídy frameworku a překryjeme metodu createButton.
- Zatímco tato metoda ve výchozí třídě vrací objekty typu Button, naše podtřída bude vracet RoundButton.
- Nakonec použijeme třídu UIWithRoundButtons místo UIFramework.

FACTORY – KDY POUŽÍT

- Použij návrhový vzor Factory Method, když chceme šetřit systémové prostředky tím, že budeme znovu používat existující objekty místo toho, abychom je pokaždé vytvářeli znovu.
- Tuto potřebu často vídáme při práci s velkými a náročnými objekty, jako jsou databázová spojení, systémy souborů nebo síťové zdroje.
- Co je potřeba udělat pro opětovné použití existujícího objektu:
  - Nejprve musíme vytvořit úložiště, které bude sledovat všechny vytvořené objekty.
  - Když někdo požádá o objekt, program by měl vyhledat volný objekt v tomto „poolu“.
  - … a pak ho vrátit klientskému kódu.
  - Pokud žádný volný objekt není, program by měl vytvořit nový (a přidat ho do poolu).
- To je poměrně hodně kódu! A měl by být centralizovaný na jednom místě, aby se program nezanesl duplicitní logikou.
- Nejzřejmějším a nejpohodlnějším místem, kam tento kód umístit, by mohl být konstruktor třídy, jejíž objekty se snažíme znovu použít.
- Jenže konstruktor ze své podstaty vždy vrací nový objekt – nemůže vracet již existující instanci.
- Proto potřebuješ běžnou metodu, která dokáže vytvářet nové objekty i znovu používat existující.
- A to zní přesně jako tovární metoda (Factory Method).

FACTORY – JAK IMPLMENTOVAT

1. Je třeba zajistit, aby všechny produkty implementovaly stejné rozhraní. Toto rozhraní by mělo deklarovat metody, které dávají smysl pro každý produkt.
2. Do třídy factory se přidá prázdná tovární metoda. Návratový typ této metody by měl odpovídat společnému rozhraní produktů.
3. V kódu tvůrce je nutné najít všechna místa, kde se používají konstruktory produktů. Postupně se nahrazují voláním tovární metody a logika vytváření produktů se přesouvá právě do této metody.
    1. V této fázi může kód tovární metody působit nevzhledně – může obsahovat rozsáhlý switch blok, který vybírá, jakou třídu produktu vytvořit. To však lze později vylepšit.
4. Následně se vytvoří sada podtříd tvůrce – jedna pro každý typ produktu, který byl uveden v tovární metodě. V těchto podtřídách se tovární metoda přepíše a odpovídající části konstrukčního kódu se přesunou z původní metody.
5. Pokud je typů produktů příliš mnoho a nedává smysl vytvářet podtřídu pro každý z nich, lze znovu použít řídicí parametr z původní třídy i v podtřídách.
6. Pokud po všech těchto přesunech zůstane základní tovární metoda prázdná, může být označena jako abstraktní. Pokud v ní něco zůstane, může být považováno za výchozí chování metody.

FACTORY – PRO A PROTI

- **Pro**:
- Vyhnete se **silnému svázání** mezi tvůrcem a konkrétními produkty.
- **Princip jediné odpovědnosti (SRP)**: Kód pro vytváření produktů lze přesunout na jedno místo v programu, což usnadňuje jeho údržbu.
- **Princip otevřenosti/uzavřenosti (OCP)**: Do programu lze zavádět nové typy produktů, aniž by bylo nutné měnit existující klientský kód.

- **Proti:**
- Kód se může stát složitějším, protože je potřeba zavést velké množství nových podtříd pro implementaci návrhového vzoru.
- Nejideálnějším scénářem je situace, kdy se vzor zavádí do již existující hierarchie tříd tvůrců.

FACTORY PŘÍKLAD JAVA

- Logistics

SINGLETON

- Singleton je kreacionální návrhový vzor, který umožňuje zajistit, že daná třída bude mít pouze jednu instanci, a zároveň poskytuje globální přístupový bod k této instanci.
- Tento vzor se často používá například u konfigurací aplikace, správy připojení k databázi, nebo logovacích služeb, kde je žádoucí, aby existoval právě jeden sdílený objekt v rámci celého systému.

SINGLETON

- Návrhový vzor **Singleton** řeší **dva problémy současně**, čímž porušuje **princip jediné odpovědnosti (Single Responsibility Principle)**:
- **Zajištění, že třída má pouze jednu instanci**
  - Jedním z hlavních důvodů, proč je žádoucí kontrolovat počet instancí třídy, je **řízení přístupu ke sdílenému zdroji** – například k databázi nebo souboru.
  - Princip fungování je následující: pokud je objekt jednou vytvořen a později se aplikace pokusí vytvořit nový, místo nové instance obdrží tu, která již byla vytvořena.
  - Takové chování **nelze dosáhnout běžným konstruktorem**, protože ten ze své podstaty **vždy vrací novou instanci**.

SINGLETON

- Poskytnutí globálního přístupového bodu k instanci
  - V minulosti se často používaly globální proměnné k uchování důležitých objektů. Tyto proměnné byly sice praktické, ale zároveň velmi nebezpečné – jakýkoliv kód mohl jejich obsah přepsat a způsobit pád aplikace.
  - Podobně jako globální proměnná umožňuje vzor Singleton přístup k určitému objektu odkudkoliv v programu. Na rozdíl od globální proměnné však Singleton chrání tuto instanci před přepsáním jiným kódem.
  - Existuje i další aspekt tohoto problému: není žádoucí, aby kód, který řeší kontrolu nad jedinou instancí, byl roztříštěný po celém programu. Mnohem vhodnější je mít tuto logiku centralizovanou v jedné třídě, zejména pokud na ni zbytek systému spoléhá.
  - V dnešní době je vzor Singleton natolik rozšířený, že se za „singleton“ často označuje i řešení, které pokrývá pouze jeden z výše uvedených problémů.

SINGLETON - ŘEŠENÍ

- Všechny implementace návrhového vzoru Singleton mají společné dva kroky:
  - Výchozí konstruktor se nastaví jako privátní, aby se zabránilo ostatním objektům ve vytváření instancí pomocí operátoru new u třídy Singleton.
  - Vytvoří se statická metoda, která funguje jako konstruktor. Interně tato metoda volá privátní konstruktor pro vytvoření objektu a uloží ho do statického pole. Všechny následující volání této metody pak vracejí uloženou (kešovanou) instanci.
- Pokud má kód přístup ke třídě Singleton, může volat její statickou metodu.
- Při každém volání této metody se vždy vrací stejný objekt.

SINGLETON - VYUŽITÍ

- Vzor Singleton se používá tehdy, když má být od určité třídy v programu k dispozici pouze jedna instance, sdílená všemi klienty.
- Typickým příkladem je jediný objekt databáze, který využívají různé části aplikace.
- Tento vzor zakazuje všechny ostatní způsoby vytváření objektů dané třídy, kromě speciální metody pro vytvoření instance. Tato metoda buď vytvoří nový objekt, nebo vrátí již existující, pokud byl dříve vytvořen.
- Singleton se také hodí v případech, kdy je potřeba přísnější kontrola nad globálními proměnnými.
- Na rozdíl od globálních proměnných zaručuje Singleton, že existuje pouze jedna instance třídy.
- Tuto uloženou instanci nelze přepsat žádným jiným kódem, kromě samotné třídy Singleton.
- Je však důležité poznamenat, že toto omezení lze kdykoliv upravit – například tak, aby bylo možné vytvářet více instancí.
- Jediné místo, které je potřeba změnit, je tělo metody getInstance.

SINGLETON – JAK IMPLEMENTOVAT

- Do třídy se přidá privátní statické pole pro uložení instance Singletonu.
- Deklaruje se veřejná statická metoda pro získání instance Singletonu.
- Uvnitř této statické metody se implementuje „lazy inicializace“ – při prvním volání se vytvoří nový objekt a uloží se do statického pole.
- Při všech dalších voláních se vrací právě tato uložená instance.
- Konstruktor třídy se nastaví jako privátní.
- Statická metoda třídy bude mít stále přístup ke konstruktoru, ale ostatní objekty nikoliv.
- V klientském kódu se nahradí všechna přímá volání konstruktoru Singletonu voláním jeho statické metody pro vytvoření instance.

SINGLETON – PRO A PROTI

- Lze si být jistý, že daná třída má **pouze jednu instanci**.
- Získává se **globální přístupový bod** k této instanci.
- Objekt typu Singleton je **inicializován až při prvním požadavku**, tedy při prvním volání metody, která jej poskytuje.

- Porušuje princip Single Responsibility Principle, protože vzor řeší dva problémy současně.
- Vzor Singleton může zakrývat špatný návrh, například když jednotlivé komponenty programu vědí příliš mnoho jedna o druhé.
- Vyžaduje speciální zacházení v prostředí s více vlákny, aby nedošlo k tomu, že více vláken vytvoří instanci Singletonu vícekrát.
- Může být obtížné jednotkově testovat klientský kód, který Singleton používá, protože mnoho testovacích frameworků spoléhá na dědičnost při vytváření mock objektů.
- Jelikož konstruktor třídy Singleton je privátní a přepisování statických metod není ve většině jazyků možné, je nutné vymyslet kreativní způsob, jak Singleton otestovat.
- Nebo se testy prostě nebudou psát.
- Nebo se Singleton vůbec nepoužije.

SINGLETON

- příklad