Návrhové vzory

- cílem objektového návrhu je struktura, která má:
 - o snadnou rozšiřitelnost
 - účelnost
 - testovatelnost
 - o dokumentovatelnost
- každý vzor:
 - o popisuje
 - často se vyskytující problém
 - jádro řešení tohoto problému
 - o tak aby:
 - bylo možno řešení opakovaně využívat...
 - ... bez toho, že bychom dělali stejnou věc dvakrát
- typy vzorů v softwarovém inženýrství:
 - o analytické pro řešení analýzy a popisu zadání
 - o architektonické pro určení vhodné architektury řešení
 - o **integrační** pro řešení integrace
 - o návrhové pro implementaci součástí řešení
 - o konvence kódu konvence, vzory a standardy jak psát zdrojový kód

Analytické vzory

- konceptuální model
- zachycuje abstrakci často modelované situace
- reprezentován jako skupina generických objektů, vztahy a interakce mezi nimi
- doménově neutrální
- popisuje konceptuální struktury na business úrovni, ne SW implementace
- např. Accountable, Měření a pozorování,...

Architektonické vzory

- reagují na nutnost řešení v SW inženýrství limitovaný výkon HW, vysoká dostupnost, rozšiřitelnost,...
- vzory pro konstrukci např. snaha o konstrukci dostatečně pružné aplikace
- Layered pattern; Client-Server; Peer-to-Peer; MicroServices; MVC pattern,...

Layered pattern (vzor vrstev)

- systém členěn do skupin (vrstev), které jsou řazeny pod sebe
- každá vrstva poskytuje služby vrstvě nad sebou
- časté členění na:
 - o prezentační vrstvu
 - o servisní vrstvu
 - vrstvu business logiky
 - o datovou vrstvu (DAO)

MVC pattern

- oddělení správy dat, jejich prezentace a vlastní řízení aplikace
- **model** představuje data a servisní vrstvu přístupu k datům
- **view** zajišťuje UI a prezentaci dat
- **controller** koordinuje view a model

Návrhové vzory

- vzor pro řešení v SW návrhu
- snaha o podchycení typických problémů, které máme rozhlednit při návrhu, a jejich řešení
- GoF "Gang of Four" vzory
 - katalog vzorů
 - o jednotná struktura popisu vzoru
 - jméno vzoru
 - problém
 - řešení
 - důsledky
 - o různé typy:
 - Creational (pro vytváření instancí)
 - Structural (pro organizování objektů do struktur)
 - Behavioral (pro provádění programové logiky)

Creational patterns

- Factory method umožňuje jednotně vytvářet instance z vícero odvozených tříd
- Abstract factory umožňuje jednotně vytvářet skupiny spolusouvisejících objektů
- Builder odděluje konstrukci objektu od jeho reprezentace
- **Prototype** plně inicializovaná instance, slouží jako vzor pro kopírování
- Singleton třída, která má mít jen jedinou instanci

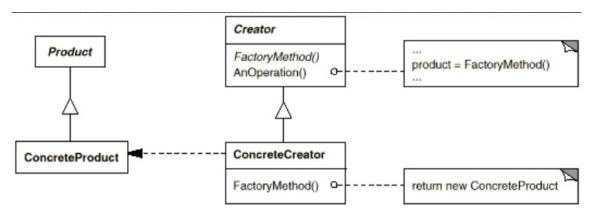
Factory method

- motivace:

- o potřebuji skrýt podrobnosti vytváření konkrétního objektu
- o mám sadu tříd se společnými metodami, ale detaily instancování zná až aplikace

příklad:

- o logistická aplikace pracuje s různými druhy dopravy (vlak, loď,...)
- potřebuji mít jednotný způsob pro vytvoření metoda createTransport, která vytvoří konkrétní objekt odpovídající dané logistice



- o neznáme přesné typy a vztahy mezi objekty
- způsob, jak uživatelé knihovny/frameworku mohou rozšiřovat interní komponenty
- o chceme šetřit systémové zdroje a znovupoužít existující objekty

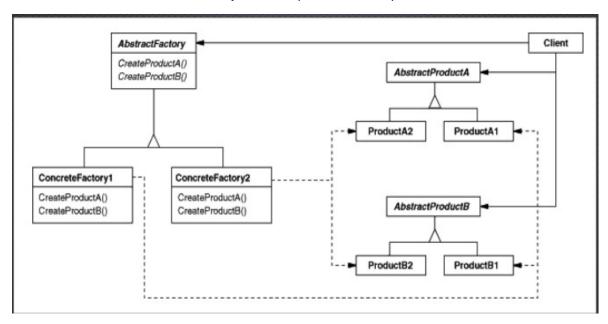
Výhody	Nevýhody
odstranění vazby mezi implementací a kódem, který ji využívá	potřeba zděděných tříd
zjednodušení kódu – přesunutí vytváření objektů na jedno místo	
zjednodušení přidávání nových tříd do programu	

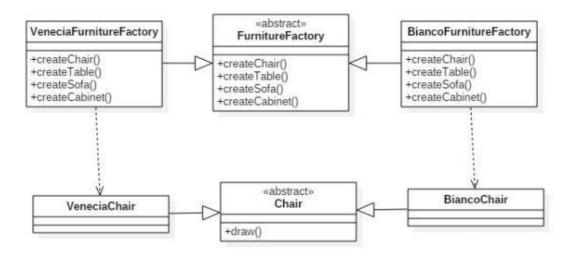
Abstract factory

- interface pro vytváření kolekcí objektů bez podrobné specifikace jejich struktury

- příklad:

- o sada skupin souvisejících objektů např. nábytková souprava (stůl, židle, pohovka,...)
- o sady jsou různé nechceme kombinovat nábytek různých sad, ale prvky se opakují
- o chceme v budoucnu jednoduše přidat nové sady





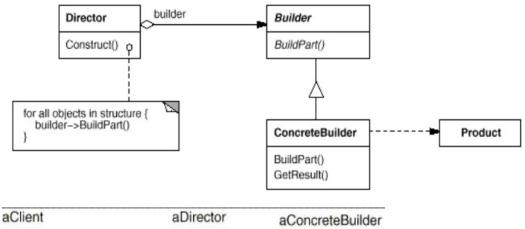
- práce s rozdílnými variantami produktů z jedné sady, ale nechceme se vázat na konkrétní třídy
- o třída má několik factory metod, které nesouvisí s její hlavní zodpovědností

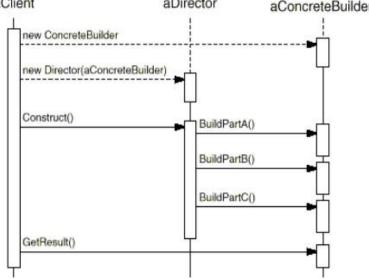
Výhody	Nevýhody
možnost vytvářet nové sady objektů, kde máme zaručenu jejich kompatibilitu	zvýšení počtu tříd v řešení
odstranění těsné vazby mezi implementací produktu a využívajícím kódem	
určuje zodpovědnost tak, aby třída měla jen jednu	

Builder

http://voho.eu/wiki/informatika/oop/navrhovy-vzor/builder/

- oddělení konstrukce složitého objektu od jeho reprezentace
- nezávislost algoritmu pro vytváření složitého objektu na vytváření částí
- reprezentace objektu různými způsoby

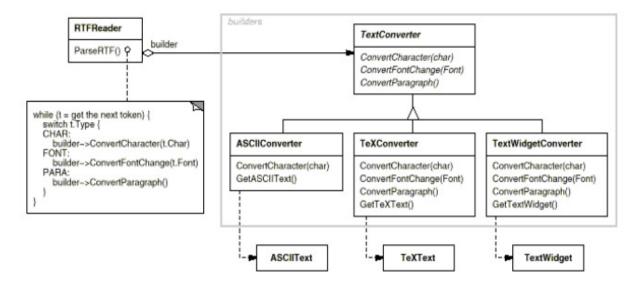




Příklad – export z RTF editoru

- editor RTF dokumentů
- měl by umět exportovat do ASCII, TEX, TextWidget,...
- postupným čtením RTF dokumentu dává příkazy konverteru do výstupního formátu

- o přečte znak
- o pošle znak do konverteru
- o přečte začátek nového odstavce
- zadá příkaz do konverteru na založení nového odstavce...



participanti:

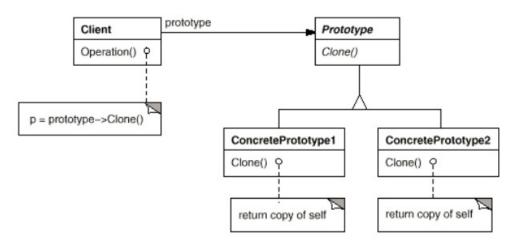
- Builder (TextConverter)
 - specifiguje abstraktní interface pro vytváření částí objektů typu Product
- ConcreteBuilder (ASCIIConverter, TeXConverter,...)
 - konstruuje a sestavuje části implementuje interface třídy builder
 - udržuje si vytvořenou reprezentaci
 - poskytuje služby pro vyzvednutí objektu (GetASCIIText, GetTeXText,...)
- Director (RTFReader)
 - konstruuje objekty pomocí služeb třídy Builder
- Product (ASCIIText, TeXText, TextWidget)
 - reprezentuje složitý objekt, který je konstruován
 - třída ConcreteBuilder vytváří konkrétní vnitřní reprezentaci
 - zahrnuje třídy pro definici částí a služby pro konstrukci celku

- výskyt konstruktoru s mnoha volitelnými parametry
- o kód produkuje různé reprezentace jednoho produktu stejné kroky, různé detaily
- o práce s kompozitními stromovými strukturami

Výhody	Nevýhody
stavba produktu krok po kroku	zvyšuje členitost (další třídy)
stejný kód může stavět rozdílné produkty	
izolace složitého vytvářecího kódu od business logiky	

Prototype

- potřebuji vyrobit kopii členitého objektu s privátními proměnnými, ke kterým se zvnějšku nedostanu
- → delegace vytvoření kopie na objekt samotný (vytvoří kopii sebe sama)



- použití:

- nemá-li kód záviset na konkrétních třídách objektů (např. když kód pracuje jen s interface)
- o chceme redukovat hierarchii tříd podobných objektů, jen jinak konfigurovaných

Výhody	Nevýhody
klonování objektů bez vazby na konkrétní třídy	těžké klonování členitých cross-reference objektů
redukce opakujících se inicializačních kódů	
rychlejší výroba nových instancí	
alternativa k dědění pro velké objekty s mnoha volbami konfigurace	

Singleton

- zajištění jediné instance třídy v aplikaci
- konkrétní typ instance lze určit za běhu
- poskytovanou instanci lze měnit bez změn klientského kódu
- snadná změna logiky tvorby (povolení více instancí apod.)

- o využití jediné instance (např. databázový objekt sdílený ostatními objekty)
- o potřeba více řídit přístup ke globálním proměnným

Výhody	Nevýhody
zajištění max. jedné instance třídy	možná souvislost se špatným návrhem
možnost odložené inicializace	nutnost speciálního ošetření ve vícevláknovém zpracování
	nutnost neustále mockovat v unit testech

Object Pool pattern

- držení objeků v pool a jejich znovupoužití
- použití:
 - o velké množství vytvářených objektů s krátkou životností
 - o časově/zdrojově náročná inicializace objektu
 - když chceme zajistit predikci času pro získání objektu (a není možné predikovat čas pro vytvoření nového objektu)
 - o potřebujeme-li řídit maximální počet aktuálně používaných objektů

Inversion of Control (IoC)

- návrhový princip, kdy kód programu je řízen z obecného frameworku
- původní klasický přístup kód volá znovupoužitelné knihovny starající se o obecné úlohy
 → inverzní přístup
- obecný framework řídí chod programu a volá specificky vytvořený kód pro řešení konkrétních úloh

Dependency injection (vkládání závislostí)

- speciální případ IoC
- technika pro vkládání závislostí mezi jednolivé komponenty programu
 - tak, aby jedna komponenta mohla využívat druhou, aniž by na ni měla v době sestavování programu referenci
- zahrnuje nejméně **tři spolupracující objekty**:
 - o konzument (objekt požadující služby)
 - poskytovatel (objekt vykonávající služby)
 - o **Dependency Provider** (dodá konzumentovi poskytovatele)
- chce-li objekt využívat služeb jiného bez použití DI, zodpovídá za jeho lifecycle (inicializace,...)
- při aplikaci DI je objekt od správy lifecycle odstíněn tu zařižuje Dependency Provider
 - o objekt tedy potřebuje jen referenci na DP
 - DP může být implementován jako lokátor služeb, abstraktní továrna, tovární metoda, pomocí frameworků,...
- existují tři různé **způsoby vkládání** (jak do objektu přidat externí referenci jiného objektu)
 - o **interface injection** externí modul, který je do objektu přidán, implementuje rozhraní, jež objekt očekává v době sestavení programu
 - o **setter injection** objekt má setter metodu, pomocí níž lze závislost injektovat
 - o **constructor injection** závislost je do objektu injektována v parametru konstruktoru již při jeho zrodu

Výhody	Nevýhody
provider může poskytovat řadu komponent bez zásahu do kódu programu	menší přehlednost kódu
o přetypování se stará IoC kontejner	náročnější správa kódu
závislosti komponent jsou definovány explicitně – jednodušší orientace v závislostech	
IoC kontejner lze většinou konfigurovat mimo kód – jednotlivé komponenty lze využít v jiných programech	

DI frameworky - Spring; Java EE WebBeans; Google Guice;...