Softwarové inženýrství

6. přednáška

Radek Janoštík

Univerzita Palackého v Olomouci

30. 10. 2025

Outline

Návrhové vzory – úvod

- Dnes budeme (většinou) hovořit o objektově orientovaném návrhu aplikací
- Vychází z (přirozeného) náhledu na svět, modelujeme reálné objekty
- Co je to objekt? Co je to třída?
  - $\Rightarrow$ Rozdělení programu na ucelené funkční celky
- Jaké jsou základní principy OOP?
  - Dědičnost
  - Polymorfismus
  - Zapouzdření
  - Rozhraní
- Snaha o abstrakci, hledání společné funkcionality tříd
  - $\Rightarrow$ hierarchie tříd
  - Problém: (abstraktní) nadtřídy nemusí mít protějšek v realitě $\Rightarrow$ hůře uchopitelné

Návrhové vzory – úvod

- Vytvoření rozumné hierarchie tříd není triviální záležitost
- Jak moc „obsáhlé“ třídy vytvářet? Kde udělat hranici?
- Jak správně navrhnout rozhraní tříd?
- Čím konkrétnější návrh bude, tím méně bude znovupoužitelný
- Přehnaná míra abstrakce může být (zbytečně) zmatečná
- Možností je opravdu mnoho a nezkušený se snadno ztratí (a tíhne k neobjektovému návrhu)
- Jistě už daný problém řešil někdo před námi
  - Jak dlouho je tady s námi OOP?
  - Proč bychom měli neustále stavět na zelené louce
  - Pojďme se poučit z chyb ostatních
  - Existuje nějaké ověřené řešení dané problematiky?

Návrhové vzory

- Přebírat ověřená řešení je přirozené a životem ověřené řešení:
  - Filmy a literatura (ošklivé káčátko, tajný super-hrdina, zakázaná láska, . . .)
  - Architektura (sedlová střecha, átrium veřejných budov, jižní a severní strana)
  - Automobilový průmysl (modulární platformy, fyzická tlačítka, dálkové ovládání)
  - Jistě každého napadne nějaký ověřený *vzor*, který se používá ve vaší doméně
- *Návrhový vzor (design pattern)* = pojmenování a způsob řešení opakujícího se problému v OOP
  - Žádná novinka, ale popis reálného problému
  - Způsob řešení není konkrétní kód, ale popis rozdělení tříd, jejich spolupráce a tvaru
- Znalost vzorů urychlí a usnadní návrh OOP programů
  - Zvyšuje pravděpodobnost úspěchu řešení
  - Vzory ustanovují i terminologii $\Rightarrow$ snadnější komunikace, konzistence
- Různě „velké“ vzory – od vzorů pro komponenty až po jejich složitou spolupráci

Návrhový vzor

bývá definován 4 základními prvky:

1. *Název vzoru* – zapamatovatelné, krátké. Plně charakterizuje vzor
   - pojmenování je důležité pro pozdější práci, komunikaci
   - usnadňuje přemýšlení (pojmenování problému jménem)
   - často se ustálí až praxí, více paralelních názvů
2. *Popis problému* – vysvětlení problému a jeho kontextu
   - kdy se může vzor použít
   - struktura tříd a objektů
   - podmínky, při kterých má vzor smysl použít
3. *Popis řešení* – detailní, ucelené. Ne úplně konkrétní
   - vzor, jak problém řešit
   - nutné prvky ve vzorech, vztahy, podmínky
   - uspořádání prvků
4. *Možné důsledky* – kompromisy při použití vzoru
   - důležité znát při rozhodování zda použít, či ne
   - jak půjde výsledek znovu použít, ušetříme čas?

Návrhové vzory – úroveň abstrakce

- Vzory nejsou o konkrétních třídách, které lze (znovu)použít celé
  - Např. datové struktury – množina, fronta, zásobník, ...
- Jsou „pod“ vzory pro architektury a „nad“ třídami
- Popis (způsobu) komunikace více tříd a objektů
- Řešení jejich vytváření (a správy)
- Rozšiřování funkcionality
- V popisu vzorů budeme používat UML
- Případně příklad v programovacích jazycích, či pseudokódech

Rozdělení návrhových vzorů

- Podle úrovně abstrakce
  - *Třídní* – vztahy mezi třídami a podtřídami (dědičnost, compile-time)
  - *Objektové* – vztahy mezi objekty – jde měnit za běhu
- Podle účelu:
  - *Vytvářející* – jakým způsobem se vytvářejí nové objekty
  - *Strukturální* – z jakých prvků jsou objekty složeny
  - *Vzory chování* – jak spolu třídy/objekty komunikují, co dělají
- Užitečné ještě mohou být *konkurentní vzory* – jak řešit paralelizační problémy
  - Probírali jste v jiných předmětech?
  - Monitor, bariéra, Thread pool, scheduler, ...
- Jednotlivé vzory se mohou doplňovat (spolupráce) nebo vylučovat (alternativní vzory)

Přehled návrhových vzorů – vytvářející

Název	Anglický název	Stručný popis	Četnost využití
Abstraktní továrna	Abstract factory	Vytváří instance vzájemně souvisejících nebo vzájemně závislých tříd, aniž bychom specifikovali konkrétní třídy těchto objektů	vysoká (5/5)
Stavitel	Builder	Odděluje konstrukci složitých objektů od jejich prezentace, čímž stejným postupem konstrukce lze vytvářet odlišné prezentace	nižší (2/5)
Továrna	Factory method	Vytváří objekty, přičemž potomci třídy rozhodují, z kterých tříd objekty budou vytvořeny	vysoká (5/5)
Prototyp	Prototype	Specifická instance třídy. Nové objekty se vytvářejí klonováním či kopírováním prototypu	střední (3/5)
Jedináček	Singleton	Třída, která má jen jednu instanci (objekt)	vyšší (4/5)

Přehled návrhových vzorů – strukturální (1 / 2)

Název	Anglický název	Stručný popis	Četnost využití
Adaptér	Adapter	Konvertuje rozhraní jedné třídy na rozhraní druhé třídy. Vytváří společné rozhraní mezi třídami, jejichž rozhraní jsou vzájemně nekompatibilní	vyšší (4/5)
Most	Bridge	Odděluje rozhraní a implementaci objektu, čímž se obojí může měnit nezávisle na sobě	střední (3/5)
Kompozit	Composite	Vytváří stromovou strukturu z uzlů. Přitom interní a listové uzly ačkoliv jsou z rozdílných tříd, mají jednotné rozhraní	vyšší (4/5)
Dekorátor	Decorator	Dynamicky přidává k objektu zodpovědnosti. Flexibilní způsob, jak prostřednictvím dědících tříd rozšířit funkcionalitu objektu	střední (3/5)

Přehled návrhových vzorů – strukturální (2 / 2)

Název	Anglický název	Stručný popis	Četnost využití
Fasáda	Facade	Třída vytvářející rozhraní po celý subsystém. Fasáda definuje rozhraní, prostřednictvím kterého se se subsystémem snadněji pracuje	vysoká (5/5)
Muší váha	Flyweight	Efektivní správa většího množství objektů s podobnou strukturou. Principem je sdílení co největší části dat těchto podobných objektů	nízká (1/5)
Zástupce	Proxy	Objekt zajišťující přístup k jinému objektu	vyšší (4/5)

Přehled návrhových vzorů – vzory chování (1 / 3)

Název	Anglický název	Stručný popis	Četnost využití	
Řetěz zodpovědnosti	Chain of Responsibility	Předání požadavku do řetězce objektů. Místo aby se požadavek předal příslušnému objektu, je předán do řetězce, ve kterém je mezi objekty předáván, dokud se nedostane k objektu, který ho vyřídí	nižší (2/5)	
Příkaz	Command	Odděluje zadání požadavku a jeho vykonání. Tedy třída zadávající požadavek je oddělena (klient) od třídy vykonávající požadavek, která rozhoduje, jak požadavek bude vykonán	vyšší (4/5)	
Interpreter	Interpreter	Interpretace vět jednoduchého jazyka (jazyk je zadán jednoduchou gramatikou)	nízká (1/5)	
Iterátor	Iterator	Poskytuje přístup pro sekvenční procházení datové struktury, aniž bychom znali interní reprezentaci struktury.	vysoká (5/5)	

Přehled návrhových vzorů – vzory chování (2 / 3)

Název	Anglický název	Stručný popis	Četnost využití
Mediator	Mediator	Zajišťuje komunikaci mezi dvěma třídami. Třídy nekomunikují tímto přímo spolu, čímž jedna třída nemusí znát metody druhé třídy	nižší (2/5)
Memento	Memento	Uchová stav objektu, čímž ho případně umožňuje později vrátit zpět	nízká (1/5)
Pozorovatel	Observer	Popis závislosti *one-to-many*, ve které při změně stavu *jednoho* jsou *ostatní* o změně informování	vysoká (5/5)
Stav	State	Změní chování objektu, když se změní jeho stav. Z vnějšku to vypadá, jako by se z něho stal objekt jiné třídy	střední (3/5)

Přehled návrhových vzorů – vzory chování (3 / 3)

Název	Anglický název	Stručný popis	Četnost využití
Strategie	Strategy	Zapouzdřuje více algoritmů, čímž umožňuje jejich výběr nezávisle na klientovi, který je používá	vyšší (4/5)
Šablona	Template method	Definuje skelet algoritmu, přičemž některé jeho kroky ponechává na dědících třídách. To umožňuje algoritmus modifikovat bez změny jeho struktury pomocí dědících tříd	střední (3/5)
Návštěvník	Visitor	Umožňuje pro třídy definovat novou operaci, aniž bychom tyto třídy měnili. Třída realizující novou operaci tyto třídy „navštíví“ a operaci na datech v nich uložených vykoná	nízká (1/5)

Popis návrhového vzoru

- Nespokojíme se s UML diagramem, je potřeba vzor důkladně popsat, jeho problém, motivace, řešení, výhody nevýhody
- Ucelený (doporučený formát):
  - *Název* – výstižný, ihned představitelný. Udává názvosloví, v názvech tříd.
  - *Klasifikace* – hovoří o třídách či objektech? Vytváření? Chování? Struktuře?
  - *Účel* – popis co vzor dělá, jaký je jeho účel
  - *Alternativní názvy* – jaká jiná terminologie se „uchytila“?
  - *Motivace* – konkrétní scénář, který vzor řeší
  - *Popis struktur* – diagramy
  - *Popis součástí* – jaké třídy jsou ve vzoru obsaženy, co která dělá
  - *Popis spolupráce* – jak třídy a objekty spolu fungují
  - *Důsledky* – jaký má vzor dopad na systém?
  - *Implementace* – co bychom měli zohlednit v implementaci? + ukázka
  - *Reálné použití* – kde byl vzor již využit
  - *Související vzory* – které mohou vzor doplnit, případně alternativy

Vytvářející vzory – úvod

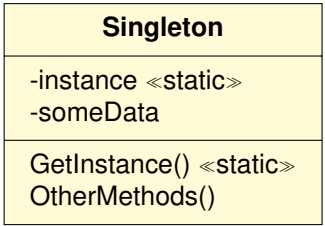
- Řeší, jak se vytváří nové instance
- Třídní vytvářející vzory – využití dědičnosti
- Objektové vytvářející vzory – *jiný* objekt vytváří nové objekty
- Důležité u systémů, kde převažují složené objekty (méně dědičnosti)
- Základní rysy vzorů
  - Skrývají *konkrétnost* třídy, které systém používá
  - Skrývají *jak* jsou třídy vytvářeny a „skládány do sebe“
  - Zbytek systému pouze pracuje s rozhraními/abstraktními třídami
- V *compiletime* nemusíme vědět, jaké konkrétní objekty dostaneme

Jedináček (Singleton)

- *Klasifikace* – objektový vytvářející vzor
- *Účel* – zajistí, aby třída měla pouze jednu instanci. Instance bude dostupná „globálně“
- *Motivace* – potřeba zajistit, aby funkcionalitu měla pod kontrolou právě jedna instance
  - Jeden přístupový bod pro funkcionalitu
  - Logger – potřebujeme zapisovat postupně, do jednoho souboru
  - Databáze – řešení transakcí, rollback
  - Funkce operačního systému (plánovač)
- Jak to udělat? Nějakou globální proměnnou?
  - Lepší je nechat vytváření a unikátnost na třídě samotné
- *Součásti* – pouze jedna třída *Singleton*
  - Umožňuje přístup k unikátní instanci
  - Odpovědná i za vytvoření

Jedináček (Singleton)

- *Struktura*



- *Důsledky* – kontrolovaný přístup – jak a kdy budou mít klienti přístup
  - Nezanáší do aplikací globální proměnné
  - Nemusí být pouze jedna instance – kontrola nad omezením množství
- *Implementace* – `private` konstruktor
- *Související vzory* – používá více dalších vzorů (Abstraktní továrna, Stavitel, Prototyp)

Jedináček (Singleton) – zdrojový kód

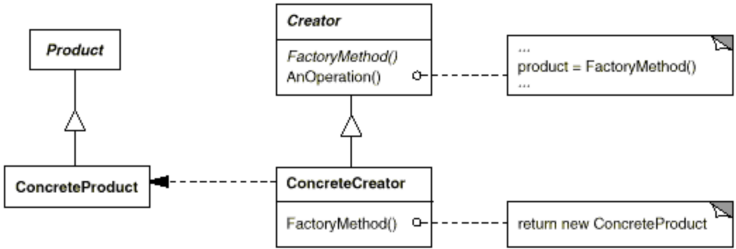
```
1  class Singleton {
2    private static Singleton instance=null; // private staticky field
         pro ulozeni instance
3    private Singleton() {} // private konstruktor
4
5    public static Singleton GetInstance() {
6      if (instance == null) {
7        instance = new Singleton();
8      }
9      return instance;
10   }
11 }
```

- Když neznáte vzor, na první pohled netušíte, co se v kódu děje

Továrna (Factory Method)

- *Klasifikace* – Třídní vytvářecí vzor
- *Účel* – Definuje rozhraní pro vytváření objektu, ale vytvoření ponecháno podtřídám
- *Alternativní název* – Virtuální konstruktor (Virtual Constructor)
- *Motivace* – Mějme *framework* pro aplikace, které zobrazují různé typy dokumentů uživateli
  - Dvě abstrakce (třídy) `Document` a `Application` – abstraktní třídy
  - Pro kreslící aplikaci vytvoříme podtřídy `DrawingApplication` a `DrawingDocument`
  - `Application` by měla být schopná vytvořit nový `Document`, kdykoliv si to uživatel bude přát (klikne)
  - Jenže konkrétní reprezentace `Document` závisí na typu, abstraktní `Application` neví, jaký konkrétní objekt vytvořit
  - $\Rightarrow$ Je potřeba přenechat vytvoření objektu na podtřídách
- *Využití* – když třída nezná/nemůže předvídat jaké třídy objektů má vytvářet

Továrna (Factory Method)

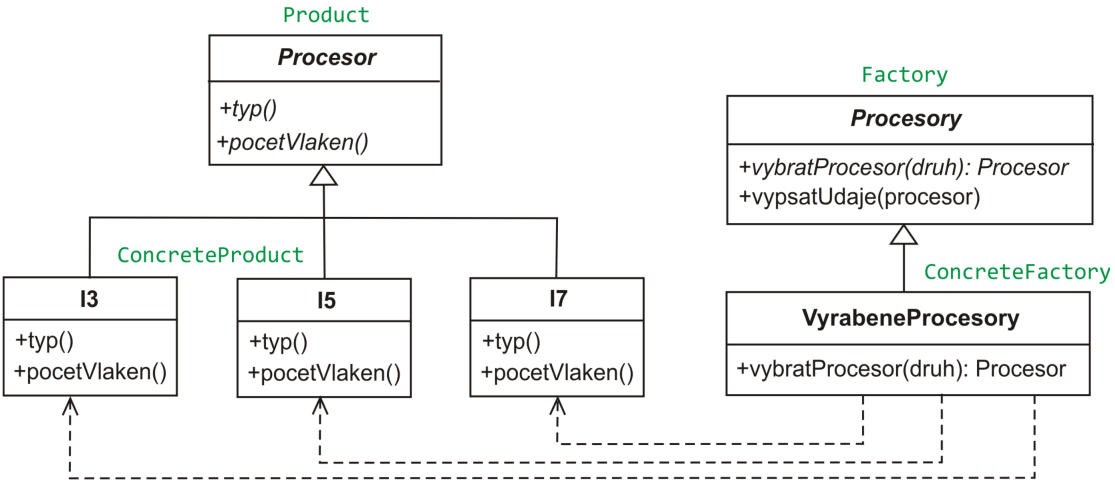


- *Součásti*
  - Product (Dokument) – definuje rozhraní objektů, které továrna vytváří
  - ConcreteProduct (DrawingDocument) – konkrétní, implementuje rozhraní Product
  - Creator (Application) – deklaruje abstraktní operaci pro vytváření Product
    - ★ Případně může implicitně vytvořit *nějaký* ConcreteProduct
  - ConcreteFactory (DrawingApplication) – implementuje tovární metodu, vrací ConcreteProduct

Továrna (Factory Method)

- *Spolupráce* – Creator spoléhá na podtřídy, že korektně vytvoří instance
- *Důsledky* – Kód (frameworku) nemusí obsahovat aplikačně specifické třídy. Pracuje jen s rozhraním
  - *hooks* pro podtřídy – např. vytvoření dialogu pro uložení souboru. Creator může dodat obecný dialog, ConcreteFactory dodá detaily.
- *Implementace* – Dvě možné varianty – Creator abstraktní, potomci musí implementovat
  - Creator vytvoří výchozí objekt, potomci můžou modifikovat
  - Parametrizovaná factory method – specifikace, který produkt vytvořit (ale splňuje rozhraní)
- *Související vzory* – Abstraktní továrna (Abstract Factory), Šablona (Template Method), Prototyp

Továrna (Factory Method) – ukázka



Továrna (Factory Method) – zdrojový kód

```
1  abstract class Procesor { // Product
2    public abstract string typ();
3    public abstract int pocetVlaken();
4  }
5
6  class I3: Procesor { // ConcreteProduct
7    public override string typ() { return "I3"; }
8    public override int pocetVlaken() { return 4; }
9  }
10 class I5: Procesor { // ConcreteProduct
11   public override string typ() { return "I5"; }
12   public override int pocetVlaken() { return 4; }
13 }
14 class I7: Procesor { // ConcreteProduct
15   public override string typ() { return "I7"; }
16   public override int pocetVlaken() { return 8; }
17 }
```

Továrna (Factory Method) – zdrojový kód

```
1  abstract class Procesory { // Factory(Creator)
2    public abstract Procesor vybratProcesor(string druh);
3    public void vypsatUdaje(Procesor proc) {
4      Console.WriteLine(\$"typ: {proc.typ()} pocet
           vlaken:{proc.pocetVlaken()}");
5    }}
6  class VyrabeneProcesory: Procesory { // ConcreteFactory
7    public override Procesor vybratProcesor(string druh) {
8      if (druh=="usporny") return new I3();
9      if (druh=="stredni") return new I5();
10     if (druh=="rychly") return new I7();
11     return null;
12   }}
13 // Main
14 Procesory procesory = new VyrabeneProcesory();
15 Procesor proc=procesory.vybratProcesor("usporny");
16 procesory.vypsatUdaje(proc);
```

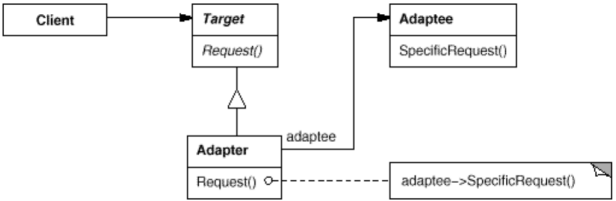
Strukturální vzory – úvod

- Řeší, jakým způsobem jsou třídy a objekty složeny „do sebe“
- *Třídní vzory* – nástrojem je dědičnost (jak rozhraní, tak implementací)
- *Objektové vzory* – jak skládat objekty, aby se jim dala rozšiřovat funkcionalita
  - Rozšiřování funkcionality za běhu
- Strukturální vzory často využívány v kombinaci

Adaptér (Adapter)

- *Klasifikace* – Třídní strukturální vzor
- *Účel* – Konvertuje rozhraní třídy na jiné, které klient umí.
  - Umožňuje „slepit“ vzájemně nekompatibilní třídy (překlad rozhraní)
- *Alternativní název* – Wrapper
- *Motivace* – Ve znovupoužití třídy často může bránit rozdílné rozhraní
  - Mějme grafický editor. Pracuje s jednoduchými prvky čáry, trojúhelníky, ...
  - Mají společné jednoduché rozhraní (vykresli, otoč, posuň, ...)
  - Chtěli bychom mít i tvar typu text, vykreslování značně komplikované
  - Využijeme knihovnu pro render textu, ale text má jiné rozhraní
  - $\Rightarrow$ potřeba překlenout rozhraní $\Rightarrow$ adaptér

Adaptér (Adapter)

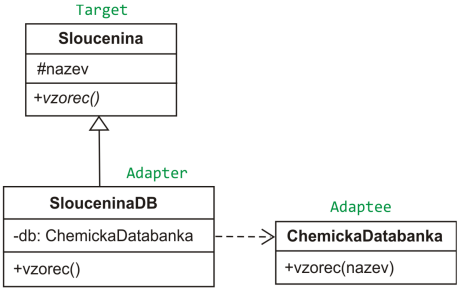


- *Součásti*
  - Target (Shape) – definice rozhraní v dané doméně
  - Client (DrawingEditor) – používá (volá metody) rozhraní Target
  - Adaptee (TextView) – existující, nekompatibilní rozhraní
  - Adapter (TextShape) – samotný adaptér „překladač rozhraní“

Adaptér (Adapter)

- *Spolupráce* – Client používá rozhraní Target – volá metody Adapter
  - Adapter se postará o překlad a korektní funkcionalitu
- *Důsledky* – kolik práce Adapter dělá?
  - Překládá pouze volání metod?
  - Musí dělat nějakou větší práci navíc (konverze formátů, . . .)
  - Nedělá víc práce než Adaptee?

Adaptér (Adapter) – ukázka



- Rozhraní `ChemickaDatabanka` není kompatibilní s rozhraním `Sloucenina`

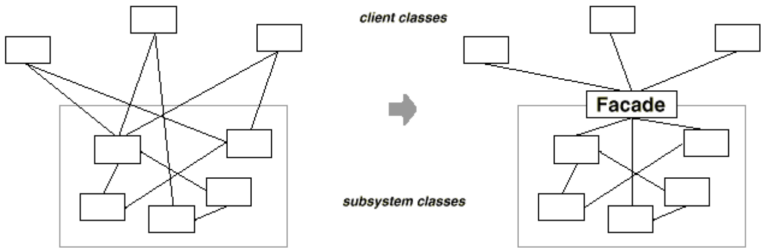
```
1 abstract class Sloucenina { // Target
2   protected string nazev;
3   public Sloucenina(string nazev) { this.nazev=n; }
4   public abstract string vzorec();
5 }
```

Adaptér (Adapter) – Zdrojový kód

```
1  class ChemickaDatabanka { // Adaptee
2    public string vzorec(string nazev) {
3      // ziskame vzorec z webove sluzby
4    }
5  }
6  class SlouceninaDB: Sloucenina { // Adapter
7    private ChemickaDatabanka db;
8    public SlouceninaDB(string nazev): base(nazev) {
9      db=new ChemickaDatabanka(); // Hodil by se Singleton?
10   }
11   public override string vzorec() { return db.vzorec(nazev); }
12 }
13 // Main
14 Sloucenina voda = new SlouceninaDB("voda");
15 Console.WriteLine(voda.vzorec());
16 Sloucenina benzen = new SlouceninaDB("benzen");
17 Console.WriteLine(benzen.vzorec());
18 Sloucenina etanol = new SlouceninaDB("etanol");
```

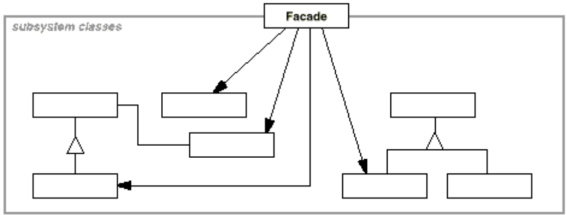
Fasáda (Facade)

- *Klasifikace* – Objektový strukturální vzor
- *Účel* – Zajišťuje jednotné rozhraní pro množinu rozhraní v subsystému objektů
  - Definuje rozhraní na „vyšší úrovni“ ⇒ snadnější použití subsystému
- *Motivace* – Máme subsystém obsahující více tříd. Potřebujeme komunikovat se všemi.
  - ⇒ Jsme na nich závislí ⇒ Hůře se mění struktura subsystému



Fasáda (Facade)

- *Motivace – příklad*
  - Řešíme překlad zdrojového kódu programu a jeho spuštění
  - Musíme kódu nejprve přeložit (samotný obrovský subsystém)
  - Dále výsledné binární soubory sestavit „do sebe“
  - Program spustit. Budeme závislí (minimálně) na 3 třídách subsystému
  - $\Rightarrow$ Bylo by dobré mít jedno rozhraní, které vyřeší vše $\Rightarrow$ Fasáda

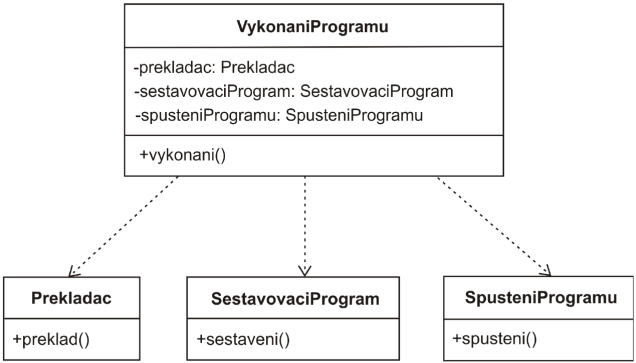


- *Účastníci* – Facade (Kompilátor) – zná třídy subsystému, vyřizuje požadavky na subsystém
  - Třídy subsystému (Scanner, Parser, Linker, Runner, . . .) – implementují funkcionalitu, nevědí o fasádě

Fasáda (Facade)

- *Spolupráce* – Klient komunikuje se subsystémem pouze přes Fasádu
  - Fasáda požadavky předává dál, případně upravuje rozhraní
  - Klient nemá přímý přístup ke třídám subsystému
- *Důsledky* – Odstínění komponent subsystému
  - Snižuje závislosti mezi klientem a subsystémem $\Rightarrow$ snadnější nahrazení
  - Snižuje množství rekompilací zdrojových kódů
- *Implementace* – Možnost Fasády jako abstraktní třídy
  - Konkrétní implementace v podtřídách (možnost volby fasády)
  - Přístup k subsystémům

Fasáda (Facade) – ukázka



- Zdrojový kód ukazovat nebudeme – implementace přímočará

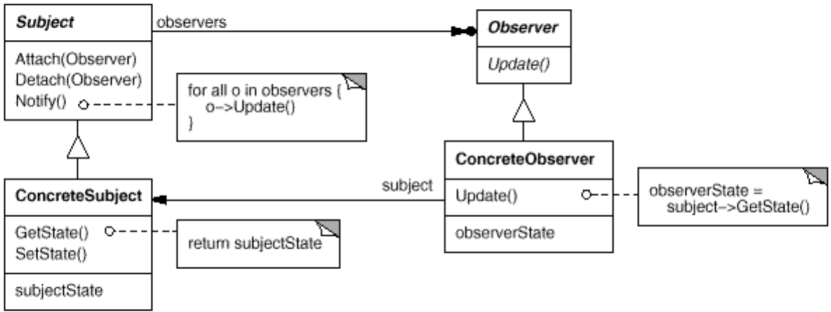
Vzory chování – úvod

- Řeší výpočty, rozdělení odpovědnosti a **komunikaci** mezi objekty
- Řeší složitý proces řízení (control flow), který je těžký sledovat za běhu
- *Třídní vzory* – chování rozděleno dědičností
  - *Šablona (Template method)* a *Interpretr*
- *Objektové vzory* – skládání objektů k vykonání složitých funkcionalit
  - Každý sám celou funkcionalitu nezvládne
  - Při vhodném složení ano (rozdělení odpovědnosti, jednodušší menší celky)
- Musí o sobě spolupracující objekty „vědět“ (=mít referenci)
- Nebo o sobě neví vůbec? $\Rightarrow$ vzory řeší různé kombinace

Pozorovatel (Observer)

- *Klasifikace* – Objektový vzor chování
- *Účel* – Popis závislosti *one-to-many*, ve které při změně stavu *jednoho* jsou *ostatní* o změně informováni.
- *Alternativní názvy* – Dependents, Publish-Subscribe
- *Motivace* – Při rozdělení systému na třídy je potřeba udržovat systém v konzistentním stavu.
  - Pro zvýšení použitelnosti je dobré mít obecný mechanismus, který umožňuje reakci na změny
  - Mějme grafickou aplikaci. Zobrazení dat (tabulka, grafy, . . .) závisí na datech, které zobrazují
  - Data se změní. Jakým způsobem zajistíme aktualizaci zobrazení dat?
  - Musí data znát své zobrazení? Dobré by bylo mít systém „notifikací o změně“
- *Použití* – když změna jednoho objektu vyžaduje změnu ostatních
  - Měnící objekt nepotřebuje znát (konkrétně) objekty, které potřebují na změnu reagovat

Pozorovatel (Observer)

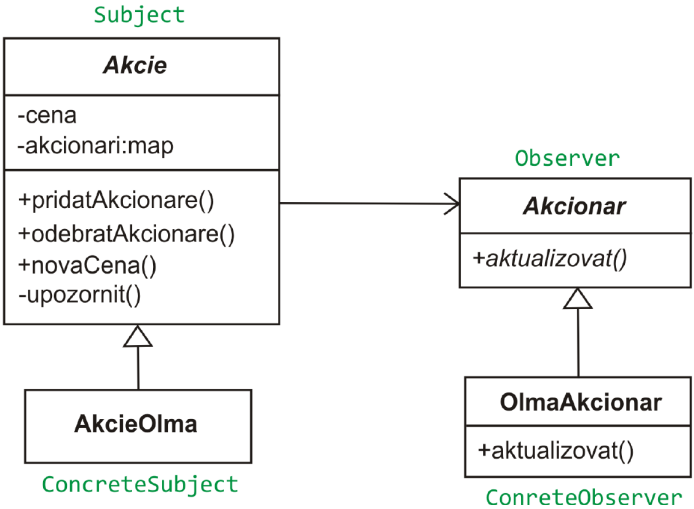


- *Účastníci*
  - Subject – zná své pozorovatele (dynamické, libovolné množství). Poskytuje rozhraní pro „přihlášení a odhlášení“ odběru změn
  - Observer – definuje rozhraní pro informování o změně
  - ConcreteSubject – uložení sledovaného stavu. Informuje o změně stavu.
  - ConcreteObserver – zná pozorovaný objekt, udržuje si stav pozorovaného, reaguje na změny

Pozorovatel (Observer)

- *Spolupráce* – ConcreteSubject informuje o změně stavu své pozorovatele
  - ConcreteObserver po obdržení notifikace se může dotázat na nový stav Subjectu
- *Důsledky* – nezanáší do kódu konkrétní závislosti. Vše řešeno dynamicky
  - Umožňuje „všesměrové vysílání“ (broadcast) o změně stavu (nezajímá nás vyslyšení)
  - Problém: Pozorovatelé se neznají. Může dojít ke kaskádě změn stavů
- *Implementace* – může zkomplikovat i jednoduché třídy
  - Ve spoustě jazyků/frameworků elegantně vyřešeno (připraveno v hierarchii)
  - Generické „observable“ datové typy implementující vzor

Pozorovatel (Observer) – ukázka



Pozorovatel (Observer) – zdrojový kód

```
1  class Akcie { // Subject
2    protected int cena;
3    private Dictionary<string, Akcionar> akcionari;
4    public Akcie(int c) {
5      akcionar= new Dictionary<string, Akcionar>();
6      cena=c;
7    }
8    public void pridatAkcionare(Akcionar a) {
         akcionari.Add(a.jmen(),a); }
9
10   public void odebratAkcionare(string j) { akcionari.Remove(j); }
11   private void upozornit() {
12     foreach (KeyValuePair<string,Akcionar> a in akcionari)
13     a.Value.aktualizovat(this);
14   }
15   public void novaCena(int c) { cena=c; upozornit(); }
16   public int cen() { return cena; }
17 }
```

Pozorovatel (Observer) – zdrojový kód

```
1  class AkcieOlma: Akcie {
2    public AkcieOlma(int c): base(c) { }
3  } // ConcreteSubject
4
5  abstract class Akcionar { // Observer
6    protected string jmeno;
7    public abstract void aktualizovat(Akcie a);
8    public string jmen() { return jmeno; }
9   }
10
11 class OlmaAkcionar: Akcionar {// ConcreteObserver
12   public OlmaAkcionar(string j) { jmeno=j; }
13   public override void aktualizovat(Akcie akcie) {
14     Console.Write("Upozorneni pro akcionare: ");
15     Console.Write(jmeno);
16     Console.Write(", nova cena akcie = ");
17     Console.WriteLine(akcie.cen()); }
18 }
```

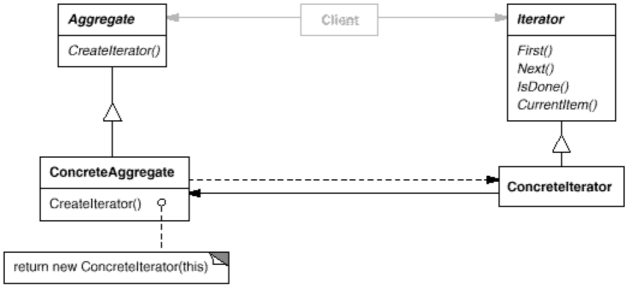
Pozorovatel (Observer) – zdrojový kód

```
1 // Main
2 // vytvorime objekt akciove spolecnosti
3 AkcieOlma olma =new AkcieOlma(1200);
4 // vytvorime objekty jejich akcionaru
5 olma.pridatAkcionare(new OlmaAkcionar("Jan Sova"));
6 olma.pridatAkcionare(new OlmaAkcionar("Petr Bukovec"));
7 // pohyby cen akcii
8 olma.novaCena(1201);
9 olma.novaCena(1209);
```

Iterator

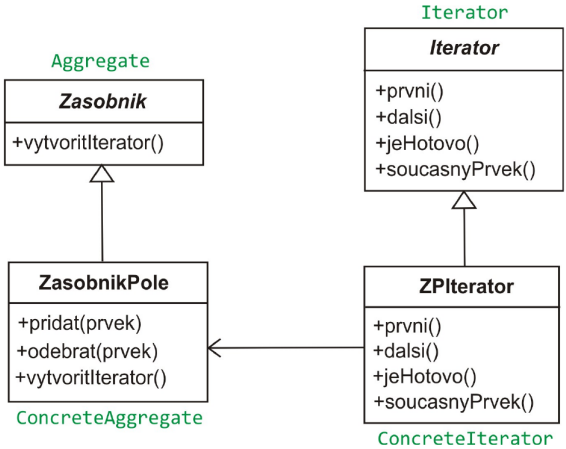
- *Klasifikace* – Objektový vzor chování
- *Účel* – Poskytuje přístup pro sekvenční procházení datové struktury, aniž bychom znali interní reprezentaci struktury.
- *Alternativní názvy* – Cursor, Enumerator
- *Motivace* – Máme různé datové struktury (seznamy, stromy, . . .) a chtěli bychom je sekvenčně procházet.
  - Mohli bychom funkcionalitu procházení dát do datových struktur $\Rightarrow$ komplikace kódu
  - Vytvořme nový objekt, který zajistí funkcionalitu pro sekvenční procházení složených objektů.
  - Mějme strom. Chtěli bychom projít jeho prvky. Jednou „do hloubky“, jednou „do šířky“, zleva, zprava, . . .
  - Pro každý typ průchodu můžeme vytvořit iterátor
  - Můžeme pak procházet „stejným kódem“ různými způsoby

Iterator



- *Účastníci* – Iterator – definuje rozhraní pro průchod složené struktury
  - ConcreteIterator – implementace rozhraní Iterator. Udržuje pozici „kde jsme“
  - Aggregate – Rozhraní pro vytvoření objektu Iterator
  - ConcreteAggregate – Konkrétní datová struktura, umožňující sekvenční průchod

Iterator – příklad



Iterator – zdrojový kód

```
1  abstract class Zasobnik { // Agregate
2    public abstract Iterator vytvoritIterator();
3    // Push, Pop, Peek, IsEmpty
4  }
5  class ZasobnikPole: Zasobnik { ConcreteAggregate
6    private int n;
7    private int [] zasob;
8    private int indx;
9    public ZasobnikPole(int nn) { zasob=new int [n=nn]; indx=0; }
10   public override Iterator vytvoritIterator() {
11     return new ZPIterator(this);
12   }
13   public void pridat(int prvek) { zasob[indx++]=prvek; }
14   public int odebrat() { return zasob[--indx]; }
15   public int index() { return indx; }
16   public int zasobnik(int i) { return zasob[i]; }
17 }
```

Iterator – zdrojový kód

```
1  abstract class Iterator { // Iterator
2    public abstract void prvni();
3    public abstract void dalsi();
4    public abstract bool jeHotovo();
5    public abstract int soucasnyPrvek();
6  }
7  class ZPIterator: Iterator { // ConcreteIterator
8    private ZasobnikPole zasobnik;
9    private int indx;
10   public ZPIterator(ZasobnikPole z) { zasobnik=z; }
11   public override void prvni() { indx=0; }
12   public override void dalsi() { ++indx; }
13   public override bool jeHotovo() {
14     return indx == zasobnik.index();
15   }
16   public override int soucasnyPrvek() {
17     return zasobnik.zasobnik(indx);
18   }
```

Iterator – zdrojový kód

```
1  // Main
2  ZasobnikPole zasob=new ZasobnikPole(10);
3  // Vlozeni prvku
4  for (int i=1; i<=5; ++i) {
5    zasob.pridat(i);
6    }
7  // Vytvoreni iteratoru
8  Iterator it=zasob.vytvoritIterator();
9  // Sekvencni pruchod
10 for (it.prvni(); !it.jeHotovo(); it.dalsi()) {
11   Console.WriteLine(it.soucasnyPrvek());
12 }
```

Návrhové vzory – závěr (1 / 2)

- Cílem dnešní přednášky nebylo ukázat důkladně všechny vzory
  - Není ani cílem předmětu (zabralo by to snad polovinu přednášek)
- Snaha nastínit jejich existenci, význam, použitelnost
- Znalost výše zmíněných vzorů budu požadovat u zkoušky
- Zbylé vzory ponechám na samostudium
  - Vypadá děsivě, že to je na více než 200 stránkách v doporučené literatuře
  - Ale čte se velmi dobře, některé pasáže zdlouhavé, text často „řídký“
- U zkoušky budu požadovat přehledovou znalost vzorů:
  - Abstraktní továrna (Abstract Factory), Stavitel (Builder)
  - Dekorátor (Decorator), Zástupce (Proxy)
  - Strategie, Příkaz (Command) a Memento

Návrhové vzory – závěr (2 / 2)

- Návrhové vzory byly důkladně probírány v předmětu KMI/OOT
  - Dříve pro 1. ročník navazujícího studia
- Bohužel pro nové studijní programy již není
  - Není ani alternativa
- Důkladnější probrání návrhových vzorů se do našeho předmětu již nevejde
- Dávám k dispozici materiály dr. Večerky s přehledem všech zmíněných vzorů:
  - `https://apollo.inf.upol.cz/~janostik/data/VytvarejiciVzory.pdf`
  - `https://apollo.inf.upol.cz/~janostik/data/StrukturalniVzory.pdf`
  - `https://apollo.inf.upol.cz/~janostik/data/VzoryChovani.pdf`
- (přehledová) znalost návrhových vzorů je důležitá
  - Při návrhu se k důkladnému nastudování můžete kdykoliv vrátit
  - Když nevíte, že něco takového existuje, tak pochopení kódu trvá dlouho

Doporučená literatura

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns, Elements of reusable Object-Oriented Software. Addison-Wesley, 1995.

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Návrh programů pomocí vzorů: stavební kameny objektově orientovaných programů. Grada 2003. ISBN: 80-247-0302-5