Otázka 05 - Y36SIN

Zadání

Návrh - návrhové třídy a jejich transformace z analytických tříd, použití návrhových vzorů, návrh rozhraní, komponenty a nasazení, odpovídající diagramy v UML. (Y36SIN)

Jen na začátek bych chtěl upozornit, že slidy nejsou na moodlu v předmětu Y36SIN k dispozici, protože předmět není teď otevřen. Napsal jsem panu Komárkovi, jestli lze slidy někde získat. Jakmile odpoví tak toto aktualizuji.

Odpověd od Komárka

Dobrý den.

Pokud budete vycházet pouze z prednaskovych prusvitek, tak se Vam muze stat, ze stanicema neprojdete. Spousta informací zaznívala pouze ustne na prednaskach nebo je uvedena v doporucene odborne literature. Např:

Arlow, J., Neustat, I.: UML2 a unifikovaný proces vývoje aplikací. Computer Press, Praha 2007.

Pecinovský, R.: Návrhové vzory – 33 vzorových postupů pro objektové programování. Computer Press, Brno 2007.

Gunderloy, M.: Z kodéra vývojářem. Computer Press, Brno 2007.

Nejnovejsi prednasky z predmetu jsou dostupne na odkazu
http://ocw.cvut.cz/moodle/course/view.php?id=203. Prihaste se jako host s heslem x36sin.

S pozdravem Martin Komarek

Vojtěch Král napsal(a):

Dobry den,
pred rokem jsem mel predmet Y36SIN a tedka na nej vypracovavam statnicovou otazku. Pouzil jsem odkaz na predmet, ktery jsem uz mel, abych stahl prednaskove slidy, ale nahlasilo mi to, ze predmet neni aktivni. Chtel bych se zeptat, jestli je nekde mozno ziskat prednaskove slidy nejlepe ze ZS 2008 (nebo z toho minuleho) ??

S podzravem
Vojtěch Král

Návrh - návrhové třídy a jejich transformace z analytických tříd

- Třídá : předpis podle kterého se vytvářejí její instance neboli objekty
- Analytická třída : třída na vysoké úrovní abstrakce, která modeluje problematickou doménu
- Návrhová třída : třída s takovým stupňem specifikace, že podle ní programátor je schopen napsat program a nebo ze které rovnou může být generován zdrojový kód

Analytická třída poskytuje informaci o své povaze funkčnosti na velmi vysoké úrovni, které by měli rozumně i normální lidé. Vůbec se v analytické třídě nezabýváme typy (int, double, String a další), viditelností (public, private, protected, default = package friendly) atributů třídy a zároveň metody třídy nejsou klasickými metodami, které znáte například z programovacího prostředí. U těchto metoda nás netrápí návratové hodnoty či přesně definované atributy. Tyto metody by meli být obecné. Rovněž dědení by mělo být zobrazeno tak, že odpovídá i specializaci (dědění) v realite, např syn je potomkem otce nebo židle je specializací nábytku (je to totiž jeho speciální případ)

Přechod na návrhovou třídu, jak jsem již řekl, se uskutečnuje zpřesnováním analytické třídy. Při tomto zpresňování je samozřejmě již nutné specifikovat typy atributů (int, double, String …) a jejich viditelnost (public, private, protected, default = package friendly). Rovněž je nutné specifikovat přesně metody třídy s jejich parametry (včetně typu) a samozřejmě návratovou hodnotu a také viditelnost. Při tomto upřesňování se většinou stane, že všeobjímající metody z analytické třídy se rozpadnout na více menších metod (není totiž správné psát metody s tuny řádky kódu).

Příklad upřesnění metody

Mějme letiště a jeho zákazníky, kteří chtěji někam letět. **Analytická** metoda terminálu letiště by například mohla být *odbavit()*. Jak již jsem řekl analytická, tak by měla mít odraz v reálném světe pro obyčejného člověka neprogramátora. To ona ale má. Před letem musíte být odbaveni, musí být zkontrolovány vaše doklady, letenky a musí vám být přiděleno místo v letadle. Toto všechno může zahrnovat metoda *odbavit()*. Tato **analytická** metoda by samozřejmě mohla být i rozdrobena na více **analytických** metod, ale to už záleží na na doméně kterou modelujete, jestli je potřeba už v analytickém modelu uvést přesně jednotlivé úkony, které se při odbavování dějí a nebo jestli vám stačí jedna větší metoda. Následně musíte metodu pro **návrhovou** třídu specifikovat. Tuto jednu metodu můžete nechat a další jednotlivé činosti rozdělit do metod *zkontrolujVíza() odejmiZavazadla() přidelSedadlo()* atd. Metoda *odbavit()* může být veřejná natož ostatní dílčí metody mohou být soukromé a budou volány z metody *odbavit()*. Samozřejmě by bylo nutno ještě u jednotlivých metod vyřešit, jestli objekt *cestující* se bude předávat jako parametr metodě *odbavit()*, což se mi jeví v tomto příkladu jako dobrý nápad, a nebo budete chtít ho předávat jinak. Rovněž by jste jeste museli specifikovat návratové hodnoty metod, takže třeba metoda *zkontrolujViza()* by mohla vracet hodnotu typu *boolean* jestli je vízum platné či nikoliv.

Samozřejmě návrhové třídy mají zásady pro svoji formulaci. Návrhová třída by měla být:

- Úplná a dostačující
- Jednoduchá
- Vysoce soudržná
- Bez těsných vazeb

Úplná a dostačující

Úplností je míněn fakt, že by třída měla poskytovat vše co od ní klienti očekávájí. Klienti mohou rovněž podle jména třídy již odhadovat, které funkce by jim třída mohla poskytnout.

Příklad úplnosti

Třída *BankovníÚčet*, která poskytuje metodu *vybrat()* by rovněž měla logicky poskytovat metodu *vložit()*. Dalším příkladem, by mohla být třída *KatalogProduktů*, u které nejspíš budete předpokládat, že bude umožnovat přidání, mazání a vyhledávání produktů.

Dostatečnost slouží k ujištění, že všechny metody třídy jsou zcela zaměřeny na realizaci zamýšleného účelu třídy. Třída neposkytuje nic navíc, co nesouvísí se zodpovědností třídy.

Příklad dostatečnosti

Dříve zmíněná třída *BánkovníÚčet* jistě **nebude** splňovat kritérium dostatečnosti, pokud bude poskytovat krom vybírání a vkládání na účet, ještě umožnovat poskytování půjček, úvěrů, či vydávání kreditních karet.

Dalo by se říci, že úplnost a dostatečnost se dají dohromady charakterizovat větou: „Třída by měla dělat to co od ní uživatelé očekávájí - nic víc, nic míň.“

Jednoduchá

Metody ve třídě by měli být navrženy tak, aby pokud možno atomické (čili aby se jejich funkčnost nedala dále rozdělit). Rovněž by třída neměla umožnovat provedení stejné operace různými způsoby, to by pak mohlo klienty mást.

Příklad

Mějme třídu, která má metody *vytvořUzivatele(nejake atributy pro uzivatele)*, *aktualizujProfil(Uzivatel)* a vy potřebujete zajistit, aby uživatel měl povinný a zároveň správně zadaný email, nick který bude max 5 písmen, vyplněnou kontrolní otázku pro případ ztráty hesla. Všechny tyto vyjmenované údaje půjdou upravovat v sekci „Uživatelský Profil". Realizaci můžete provést způsobem, že budete kontrolovat správnost uživatelových položek v obo metodách *vytvorUzivatele()* a i v *aktualizujProfil(Uzivatel)*, to ovšem vaše metody pak budou poskytovat dvě funkčnosti: kontrola uživatelových atributů a vytvoření/aktualizace uživatelova profilu. Aby třída byla **jednoduchá** měli byste kontrolou uživatelových profilů vyčlenit do speciální metody (bude volaná z *vytvořProfil()* a z *aktualizujProfil()*), která nebude delat nic jiného než kontrolovat emal, nick, kontrolní otázku a další co si navymýšlíte. Docílíte tím toho, že když budete chtít přidat uživatelovi povinně vyplněné pohlaví (Muž/Žena), tak kontrolu vyplnění přidáte **pouze** na jednom místě.

Vysoká soudržnost (Cohesion)

Třída by měla modelovat jednoduchý abstraktní pojem a měla by obsahovat množinu metod, které účelu třídy odpovídají. Pokud třída má mnoho různých odpovědností, tak z pravidla některé z nich jsou implementovány v pomocných třídách, jež naše třída volá. Soudržnost podporuje srozumitelnost a znovupoužitelnost třídy (dá se totiž vyjmout protože nevolá pro naplnění své zodpovědnosti milion dalších tříd). Soudržná třída obsahuje malý počet odpovědností, jež jsou dokonale propojeny a rovněž metody a atributy třídy jsou navrženy pouze za účelem splnění této množiny zodpovědností.

Minimalizace vazeb (Orthogonality)

Třída by měla být při plnění své odpovědnosti závislá na co nejmenším počtu okolních tříd. Častou chybou je domnění, že více je lépe. Ohledně počtu vazeb mezi třídami toto není úplně správný směr při navrhování systému. I pokud budete chtít z důvodu „znovupoužitelnosti" volat z jedné třídy kus kódu obsaženého v druhé třídě, tak toto sváží za uvážení, jestli výhoda v neduplikaci kódu nenaboura myšlenku minimalizace vazeb. Pokud se provazování tříd provádí v balíčku, tak toto není na škodu, protože zpravidla třídy v balíčku by si měli být příbuzné a všechny dohromady poskytovat nějakou sadu funkcí. Pokud ale začnete navazovat vazby mezi třídami z **různých** balíčku, tak je potřeba se nejdříve zamyslet, zda je to opravdu nezbytné.

Použití návrhových vzorů

Návrhový vzor je určitý postup jak řešit problém, na který programátoři narazili nesčetněkrát. Základní sbírkou návhových vzorů je kniha od autorů říkajících si „Gang of Four" (Odsud se taky často používá název „GoF vzory")

Návrhové vzory se dělí do 3 skupin:

Tvořivé (Creational)

Řeší problémy související s vytvářením objektů v systému. Snahou těchto návrhových vzorů je popsat postup výběru třídy nového objektu a zajištění správného počtu těchto objektů. Většinou se jedná o dynamická rozhodnutí učiněná za běhu programu. Mezi tyto návrhové vzory patří:

- Factory Method
- Abstract Factory Method
- Builder

- Prototype
- Singleton

Singleton

Všechny příklady návrhových vzorů jsou z odkazu na návrhové vzory ze **zdrojů na konci stránky**.

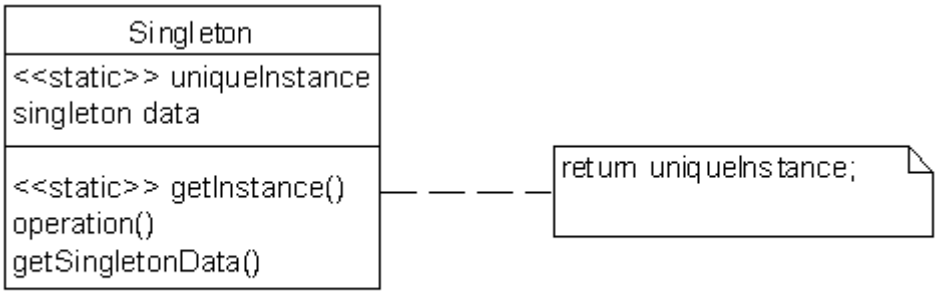
Problém

Cílem vzoru je zajištění existence pouze jedné instance dané třídy a poskytnutí globálního přístupu k ní.

Podmínky

Tento návrhový vzor je poměrně silně ovlivněn programovacím jazykem, který používáme. Při jeho návrhu a implementaci je nutné vycházet z možností a omezení tohoto jazyka.

Řešení:



Obr 6 Singleton schéma

Jak již bylo řečeno, je tato práce zaměřena na řešení vzorů s přihlédnutím k možnostem jazyka Java. Proto budou popisovány postupy odpovídající tomuto jazyku. Nejjednodušším principem, jak dosáhnout jedinečnosti instance, je zařazení proměnné, která bude obsahovat referenci na již vytvořený objekt Singletonu. Proměnná je označena v diagramu jako uniqueInstance a je definována jako statická. Označení metody za statickou zajistí její sdílení mezi různými instancemi dané třídy. Je dostupná i bez vytvoření instance třídy. Tím můžeme již v konstruktoru zjišťovat, jestli již byla vytvořena instance třídy. Pokud ano, je při vytváření objektu vrácena tato instance. Jestliže je proměnná uniqueInstance prázdná, je vytvořena nová instance a reference přiřazena do proměnné uniqueInstance. Je možné definování výjimky, která bude vyvolávána konstruktorem v momentě, když již byla vytvořena instance. Další možností, jak zajistit jedinečnost výskytu třídy, je definice třídy Singleton jako final a všechny její metody static, což nám zajistí, že třídu nebude možné dále využívat jako rodiče a k metodám se přistupuje přímo bez vytvoření instance. Nutné podotknout, že toto není úplně správné naplnění principu Singletonu, protože tento návrhový vzor vysvětluje zajištění existence pouze jedné instance. V popsaném postupu žádnou instanci třídy nevytváříme, a proto si tato nemůže zapamatovat svůj vnitřní stav a na jeho základě provádět rozhodnutí. Výhodou tohoto přístupu ovšem je, že se nemusíme starat o vyvolávání a odchytávání výjimek a ani nepotřebujeme prakticky „nefunkční“ Factory metody, které pouze zajišťují vytvoření konkrétní instance. Posledním přístupem, o kterém se zmíním, je využití statické metody pro vytvoření instance. Tento způsob je nejčistší a nejvíce používanou implementaci Singletonu. Konstruktor je deklarován jako private, což zajistí, že nemůže být volán přímo, ale pro vytvoření instance musí být použita statická metoda. Tato metoda je využití factory method ze stejného návrhového vzoru. Pokud uvažujeme o práci s více vlákny, je nutné tuto metodu synchronizovat, aby nedocházelo ke kolizím mezi jednotlivými vlákny a k vytvoření více instancí Singletonu.

Ukázka možného zajištění Singletonu:

```
public class Singleton {

 private static Singleton uniqueInstance;
  private Singleton() {
   }
   public static synchronized Singleton getInstance() {
      if (uniqueInstance == null) {
           uniqueInstance = new Singleton();
       }
       return uniqueInstance;
   }
   public void operation() {
     System.out.println("Singleton.operation() executing" );
   }

}
```

Pozn. (Miloš Klvan): Tento způsob vytvoření singletonu nezajišťuje, že bude v systému skutečně existovat pouze jedna instance této třídy. Je to kvůli možné souběžnosti vláken. Lepší je vytvořit instanci takto: private static Singleton uniqueInstance = new Singleton(); a v metodě getInstance() už jen vracet vytvořenou instanci. Na internetu je o tomto materiálů dost.

Strukturální (Structural)

Představují skupinu návrhových vzorů zaměřujících se na možnosti uspořádání jednotlivých tříd nebo komponent v systému. Snahou je zpřehlednit systém a využít možností strukturalizace kódu. Mezi tyto návrhové vzory patří:

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Adaptér

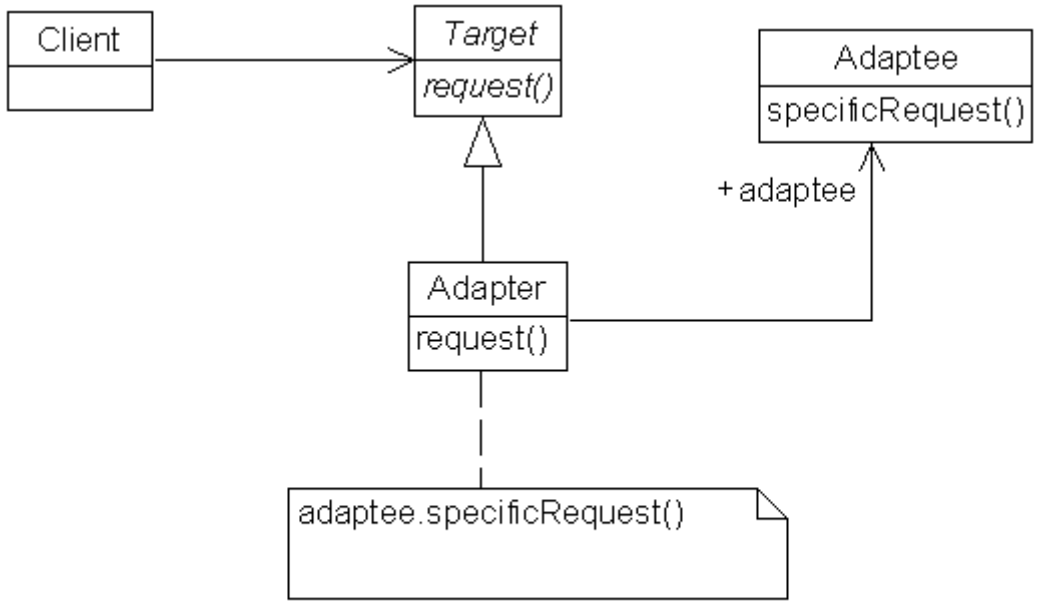
Problém:

Již název tohoto návrhového vzoru napovídá, že se jedná o přizpůsobení určité třídy, aby ji bylo možné využívat i jiným požadovaným způsobem. Problémem je zajištění konverze rozhraní jedné třídy na rozhraní jiné třídy.

Podmínky:

Základní situací, s kterou se setkáváme při přechodu na nové verze systémů, je zajištění jejich zpětné kompatibility. Například se rozhodneme využívat nové třídy, které poskytuje knihovna Swing a existuje mnoho stávajících klientů, kteří očekávají komunikaci podle standardů knihovny AWT (starší verze grafické knihovny). Abychom zajistili zpětnou kompatibilitu a zároveň mohli využívat nové technologie, musíme vytvořit mechanismus zpracování požadavků od stávajících klientů naší aplikaci. Tento problém lze řešit právě využitím tohoto návrhového vzoru.

Řešení:



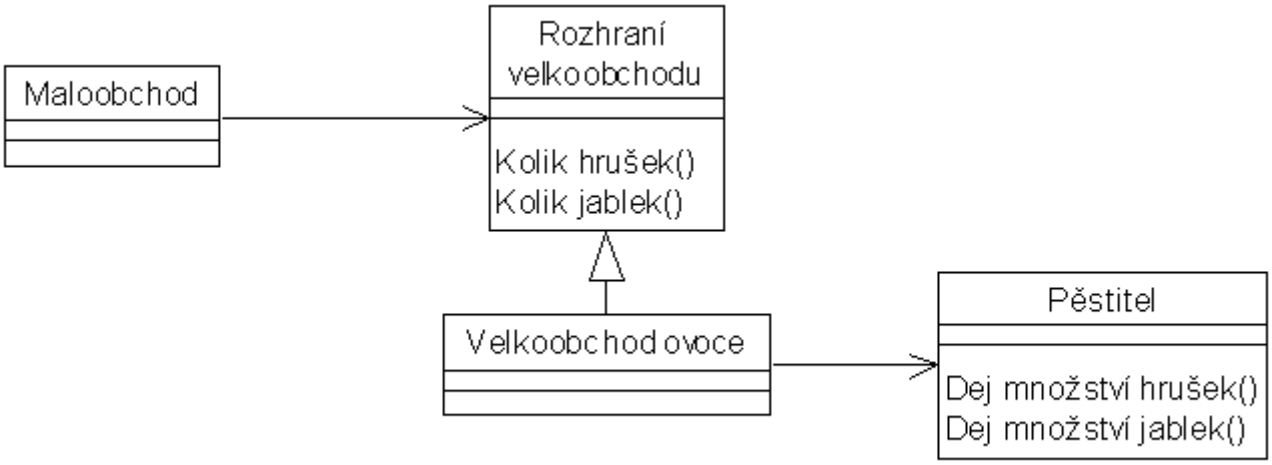
Obr 13 Objektový Adapter schéma

Adapter je třída, která zajišťuje funkcionalitu systému, ale neimplementuje požadované rozhraní. Rozhraní, které očekává třída Client, je definované v abstraktní třídě Target. Tato třída slouží jako předek pro konkrétní třídu Adapter, která má za úkol transformovat volání od klienta na příslušné metody tříd typu Adaptee.

Existují dvě možná řešení tohoto návrhového vzoru v jazyce Java. První je založen na odvození nové třídy od staré nevyhovující a přidání metod, které zajistí implementaci požadovaného rozhraní. Toto řešení je nazýváno třídní (class) adapter a využívá dědičnosti. Naproti tomu objektový (object) adapter využívá principu vložení reference na třídy Adaptee do Adapteru a delegování požadavků na metody Adaptee objektu. V takovém případě může jeden Adapter zajišťovat komunikaci s více objekty.

Příklad:

Maloobchod je zvyklý přijímat informace o stavu množství zásob ve skladech velkoobchodu. Ten mění logistickou strategii a přesouvá skladování ovoce na jednotlivé pěstitele. Z tohoto důvodu získává přehled o možných zásobách jednotlivých druhů ovoce a zeleniny přímo od pěstitele. Maloobchody by se mohly dotazovat také přímo pěstitelů, ale musely by se přizpůsobit jejich rozhraní. Druhou možností je využití rozhraní velkoobchodů. Velkoobchody potom slouží jako třídy Adapter, protože delegují požadavky od maloobchodů na jednotlivé pěstitele. Jedná se o objektový adaptér, protože si lze lehce představit situaci, kdy jeden obchod deleguje požadavek na více pěstitelů.



Obr 14 Object Adapter příklad

Výsledek:

Není nutné měnit klientskou část aplikace, která využívá adaptovanou třídu. Stávající klienti budou využívat existující rozhraní představované třídou Adapter. Noví klienti rozhraní, které je implementováno ve třídě Adaptee.

Odůvodnění a souvislosti:

Adapter slouží k zajištění propojenosti tříd, aby pracovaly v komplexním programu. Je využíván, jestliže je nutné přizpůsobit chování již vytvořených tříd a zajistit jejich vzájemnou komunikaci bez nutnosti měnit existujících rozhraní. Další možností je zajištění zpřehlednění složitého rozhraní, které je implementováno výchozí třídou. Přes třídu Adapter je potom odstíněna konkrétní implementace vytvořené třídy. Object adapter je využíván, jestliže chceme zajistit použití instancí několika existujících tříd. V takové situaci může být výhodné vytvořit třídu, která komunikuje se všemi adaptovanými a nevyužívat principu dědění nové třídy z výchozí.

Chování (Behavioral)

Tyto vzory se zajímají o chování systému. Mohou být založeny na třídách nebo objektech. U tříd využívají při návrhu řešení především principu dědičnosti. V druhém přístupu je řešena spolupráce mezi objekty a skupinami objektů, která zajišťuje dosažení požadovaného výsledku. Mezi tyto návrhové vzory patří:

- Chain Of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template
- Visitor

Observer

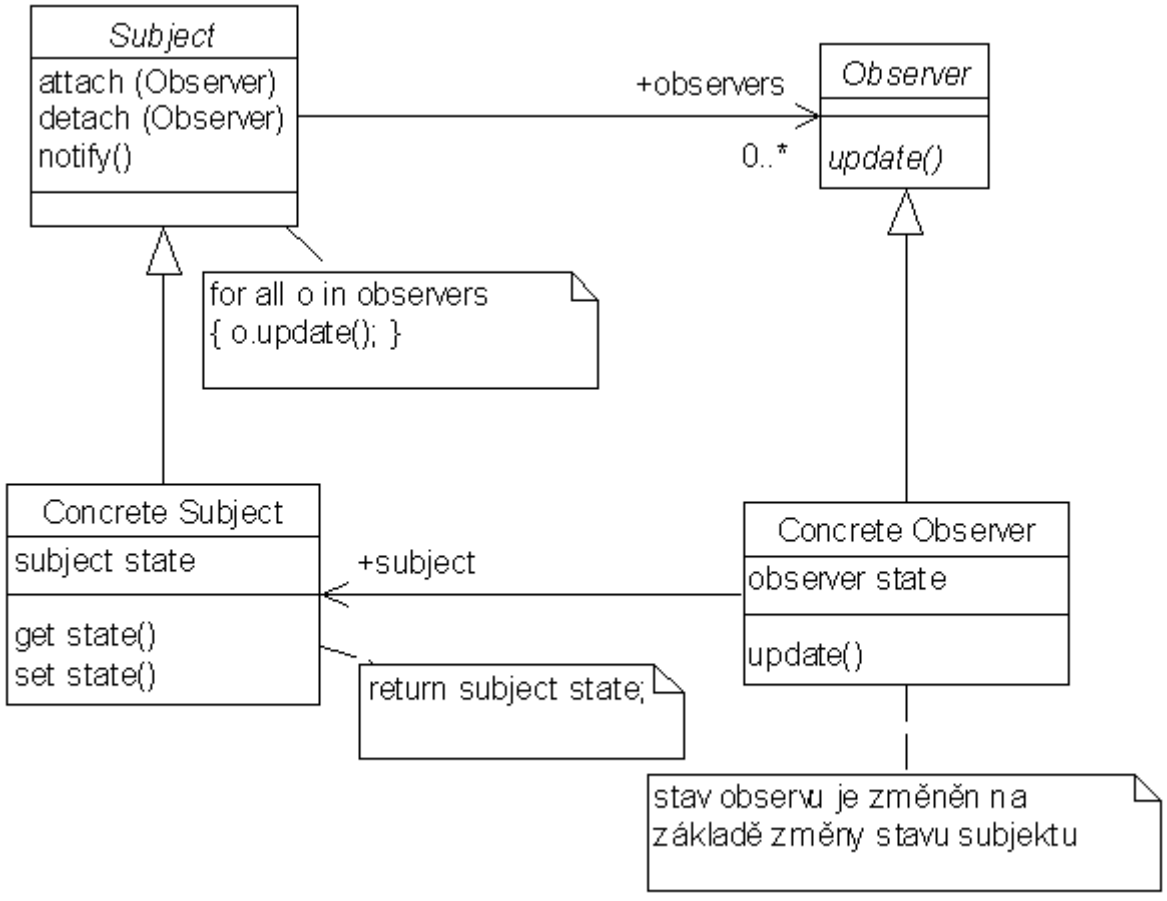
Problém:

Definování závislosti jednoho objektu k více objektům. Umožnění šíření události, která nastala v jednom objektu, ke všem závislým objektům.

Podmínky:

Observer je možné použít v situaci, kdy je definována závislost jednoho objektu na druhém. Závislost, ve smyslu tohoto návrhového vzoru, představuje propagaci změny nezávislého objektu závislým objektům (pozorovatelům). Nezávislý objekt musí informovat závislé objekty o událostech, které je mohou ovlivnit. Je nutné zajistit existenci způsobu dynamické modifikace seznamu podřízených objektů. Snahou je oddělit nezávislý nadřízený objekt od logiky informováni závislých objektů, aby tyto mohly být informovány bez znalosti jejich vnitřní struktury.

Řešení:



Obr 45 Observer schéma

Subject představuje rozhraní pro nezávislý (pozorovaný) objekt, na jehož změně závisí jiné objekty. Subject uchovává seznam podřízených objektů. Jsou definovány metody, které umožňují zaregistrovat se jako pozorovatel (metoda attach) a vymazat se ze seznamu pozorovatelů (detach). Subjekt může uchovávat odkazy na prakticky neomezené množství podřízených objektů. Metoda notify má za úkol informovat všechny pozorovatele, že nastala změna, která je může ovlivnit. Concrete Subject představuje konkrétní implementaci rozhraní Subject. Uchovává data (subject state), které poskytuje na požádání pozorovatelům (metoda get state). Je možné, že existuje více metod get state v závislosti na poskytovaných datech pozorovatelům. Observer je rozhraní pro podřízený objekt respektive pozorovatele. Objekt implementující toto rozhraní vlastní referenci na pozorovaný Concrete Subject a

pomocí metod attach a dettach si sám řídí, jestli má zájem získávat informace o změnách. Subject informuje závislé objekty pomocí volání metody update u všech podřízených objektů (Conrete Observer) a záleží na každé této třídě, jakým způsobem bude provedena implementace této metody a tím zjištěna reakce na změnu stavu pozorovaného objektu. Jestliže je Concrete Observer informován o změně, měl by voláním metody get state zjistit aktuální nastavení proměnných nezávislého Subjectu a provést synchronizaci s vlastními daty (observer state).

Příklad

Příkladem tohoto návrhového vzoru je MVC architektura popsaná v úvodu design návrhu aplikačního frameworku analyzovaného na konci práce. Dalším příkladem je informování o změně parametrů popisované v části Vybrané komponenty frameworku kapitola Aktualizace parametru pomocí Observeru.

Výsledek

Byla vytvořena vazba 1:N mezi objekty a je možné dynamicky měnit počet objektů na straně N. Implementací rozhraní pro nezávislý objekt a pozorovatele byly tyto objekty odděleny, a tím snížena míra vzájemné závislosti. Největší implementační propojení představuje asi metoda get state() poskytující data. V případě, že objem pozorovaných dat je velký a bylo by nerozumné vytvořit pouze jednu metodu poskytující toto množství dat, je možné implementovat více specializovaných metod na získání stavu pozorovaného objektu. Snížením závislosti mezi jednotlivými objekty je umožněno změnit pozorovatele bez ovlivnění nezávislého objektu Subject a opačně. Je umožněno přidávat a odebírat pozorovatele bez ovlivnění pozorovaného objektu a jiných pozorovatelů. Odůvodnění a souvislosti

Využití tohoto vzoru je především v návrhu a implementaci informování jednotlivých komponent o změnách ve zbývajících částech systému. Existuje několik otázek, které musí být brány v úvahu při využití tohoto vzoru:

- Jestliže frekvence změn v pozorovaném objektu je příliš velká, mohlo by informování o každé této změně vést k přílišnému zatížení systému. V takovém případě je vhodné uvažovat o informování pozorovatelů až po určité sadě změn.
- V návrhovém vzoru není uvažována situace, kdy jeden pozorovatel je závislý na více objektech. V takovém případě musí být uvažováno o správné implementaci metody update nebo o definici více takových metod. Použitím více modifikací této metody, lze zajistit i předávání různých druhů událostí.
- Měla by být také ošetřena situace, kdy Subject končí svoji činnost. Před vlastním uvolněním objektu Subject by mělo dojít k informování pozorovatelů.
- Musí být řízena vzájemná závislost jednotlivých pozorovaných objektů. Pozorovaný objekt neví nic o implementaci metody update u pozorovatelů, a proto se ani nezajímá o dopad volání této metody na systém. Jestliže informace o změně a následná synchronizace dat, vyvolá nekontrolovaný kaskádový efekt celým systémem (pozorovatel působí i jako pozorovaný objekt), mohou se vyskytnout výkonnostní i logické problémy.
- Problémem v implementaci může nastat, jestliže je vyvolána výjimka při informování pozorovatele (metoda update). Obecně je předpoklad, že pozorovatel nevyvolává výjimky. Není zde nikdo, kdo by je odchytil a zpracoval, protože Subject nezná implementaci pozorovatele, a proto se nestará ani o případné výjimky. Řešením této situace může být implementace dvou fázového protokolu komunikace. V první fázi Subject deklaruje, že se chystá změnit data. Pozorovatelé mohou tuto změnu odmítnout nebo potvrdit. Jestliže je udělen souhlas od všech pozorovatelů, je změna provedena a probíhá klasická komunikace dle návrhového vzoru.

Návrh rozhraní

Rozhraní : Rozhraní předepisuje pojmenovanou množinu veřejných funkcí.

Rozhraní je **předpis**, který říká, které funkce musí klasifikátor **(ten který rozhraní implementuje = třída, podsystém)** povinně implementovat. Způsob **předepsání** může být ve formě metod, jejich atributů a jejich návratových hodnot. Dále v **předepsání** musí být sémantika operace, slovně nebo jako pseudokód. Dále by mělo obsahovat množinu omezení, které zároveň musí klasifikátor splnit.

Rozhraní slouží hlavně k **oddělení implementace od specifikace**. Pokud totiž pracujete s rozhraním, tak je vám jedno co se konkrétně za tímto rozhraním skrývá, protože z podstaty rozhraní a faktu, že pracujete s klasifikátorem implementujícím toto rozhraní víte, že konkrétní implementace **musí** poskytovat funkce, které rozhraní předepisuje a je vám jedno jak této funkčnosti docílí (samozřejmě když budete hledět na rychlost implementace, tak vás bude zajímat i způsob implementace, ale to je okrajová záležitost).

Z odstavce výše vyplívá, že je velice užitečné používat rozhraní jak v rámci (uvnitř) vašeho programu, tak i rohraní vůči okolí. Pokud se totiž stane, že zjistíte, že implementace je špatná a budete ji chtít vymněnit za jinou, tak pokud pracujete s rozhraními, tak stačí aby tako nová implementace realizovala již staré rozhraní a budete ji moci v klidu začlenit do již existujícího systému.

Zároveň by se vám nemělo stávat, že budete měnit rozhraní (čili změníte předepisovanou funkčnost). Pokud se vám toto stane uvnitř vašeho systému, tak budete muset projít všechna místa, která s rozhraním pracovala a opravit je a zároveň také všechny klasifikátory. Zabereto sice čas, ale není to tak hrozné jako, když byste chtěli změnit rozhraní, které prezentuje vás program vůči okoli a toto okolí je zcela mimo vás a nemůžete ho ovlivnit. To pak hold si nemůžete dovolit změnit rozhraní a jediná možnost co vám zbývá je třeba danou metodu prohlasít jako „Deprecated“. „Deprecated“ je v Javě označení pro metody, které by se neměli používat, ale nelze je vyškrnout protože už dříve byly zveřejněny, než se přislo na to že jsou nevhodné.

V jazycích, kde není možnost použít rozhraní můžete využít abstraktních tříd, které v podstatě předepisují funkčnost stejným stylem. Rozhraní je ale lepší než dědění od abstraktní, protože v jazycích umožňujících pouze jednoho předka (Java) by mohla nastat situace, že byste musel děďit od dvou tříd, což nelze.

Na druhou stranu čeho je moc toho je příliš a toto platí i o rozhraních. Dělat rozhraní např. pro každou třídu udělá vás systém méně přehledný, protože nebudete vedět na kterou implementaci se teďka zrovna odkazujete. Pokud někde potřebujete použít třídy,o kterých 100% nepředpokládáte, že se budou v budoucnu měnit a nebo, že by se snad mohli měnit za běhu, tak si vystačíte i bez rozhraní.

Je zde taky jedna zásada KISS = „keep interfaces sweet and simple“

Komponenty a nasazení

Komponenta se chová jako černá skříňka, která ukrýva implementaci a zpřístupňuje rozhraní, jak s komponentou pracovat. Takovým příkladem by mohla být prodlužka, do vnitř nevidí (dráty jsou zakryté) a zároveň zpřistupňuje rozhraní (zastrčky na jednom a druhém konci) a jelikož má rozhraní a ukrývá implementaci, tak já mohu kdykoliv tuto prodlužku vyměnit třeba za prodlužku s přepětovou ochranou, protože rozhraní bude stejné, tak v klidu na ni pripojím oba konce, které byly připojeny i na starou prodlužku. Tuto analogii mohu provést i s komponentami také vymněnim jednu za druhou v případě, že obě poskytují stejné rozhraní.

Komponenta je ve skutečném světě vyjádření něčeho konkrétního, třeba zdrojového souboru, .jar (Java ARchive).

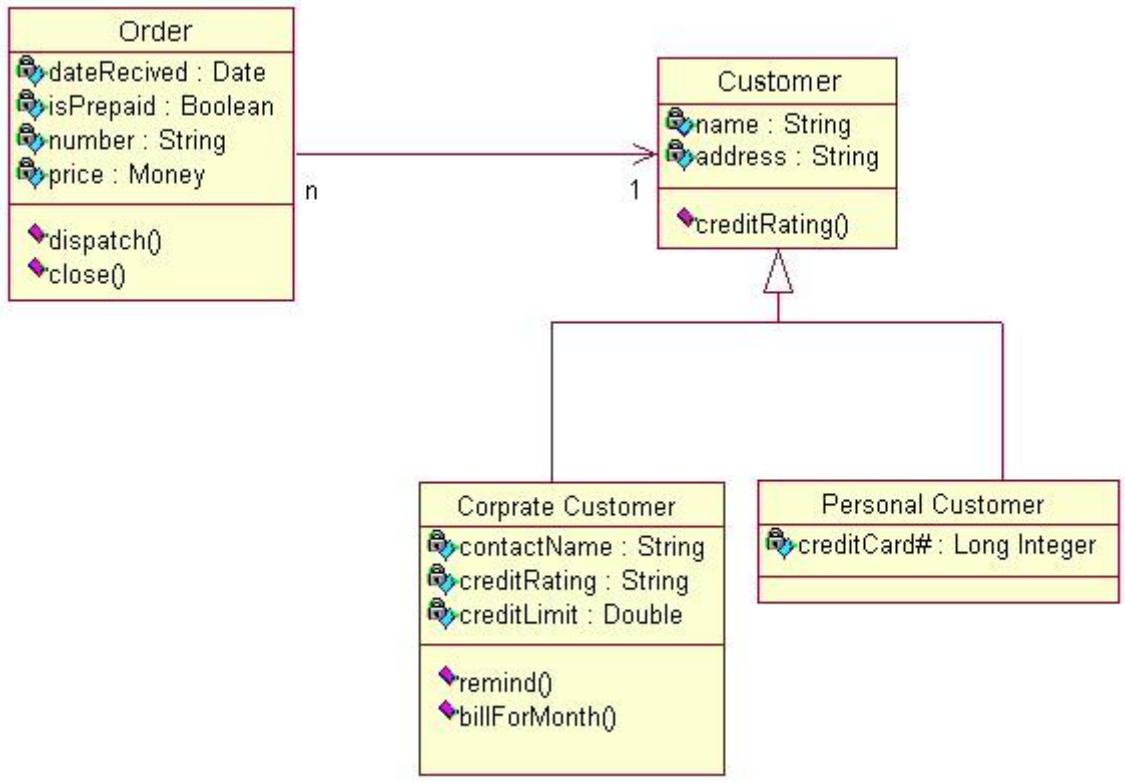
Speciálním případem komponenty by mohl být subsystém. Je to v podstatě menší část systému, která se chová jako komponenta, čili je to černá skříňka se zveřejněným rozhraním.

Odpovídající diagramy v UML

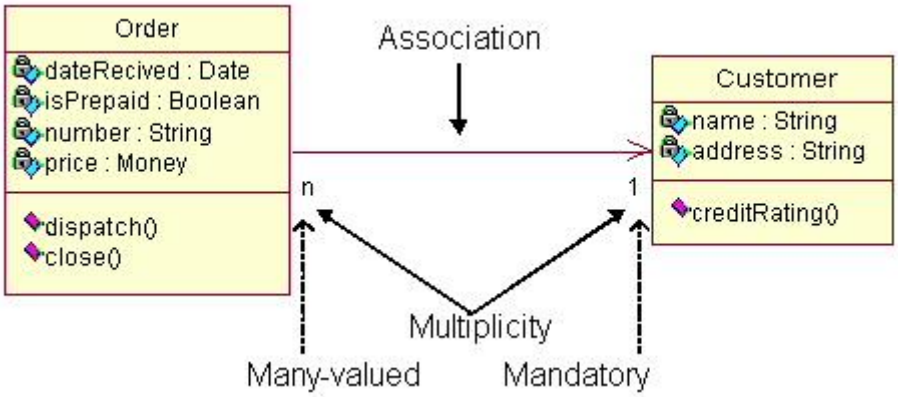
- strukturní diagramy:
  - diagram tříd
  - diagram komponent
  - composite structure diagram
  - diagram nasazení
  - diagram balíčků
  - diagram objektů, též se nazývá diagram instancí
- diagramy chování:
  - diagram aktivit
  - diagram užití
  - stavový diagram
- diagramy interakce:
  - sekvenční diagram
  - diagram komunikace
  - interaction overview diagram
  - diagram časování

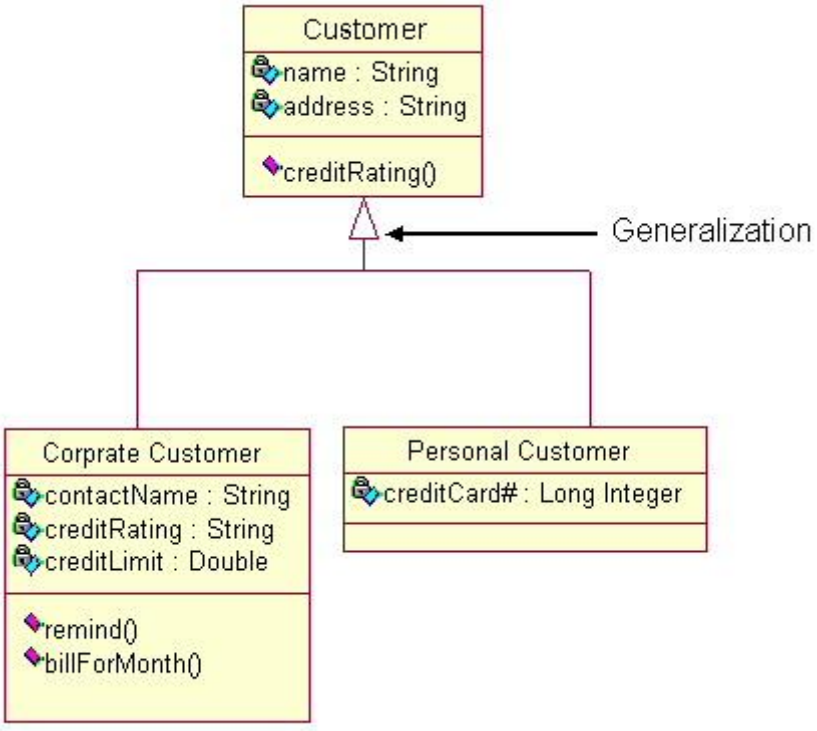
Class diagram

Tímto diagramem se dají modelovat třídy, ať už analytické či návrhové (záleží na míře specifikace).
Kompletní:



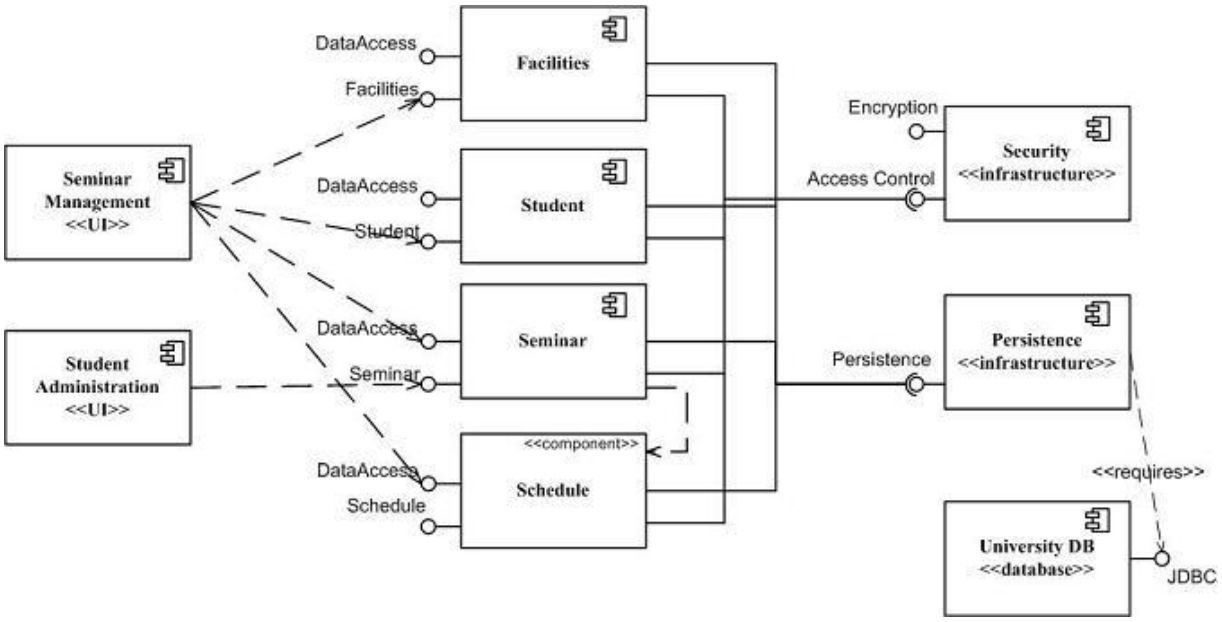
Vysvětlení jeho části:





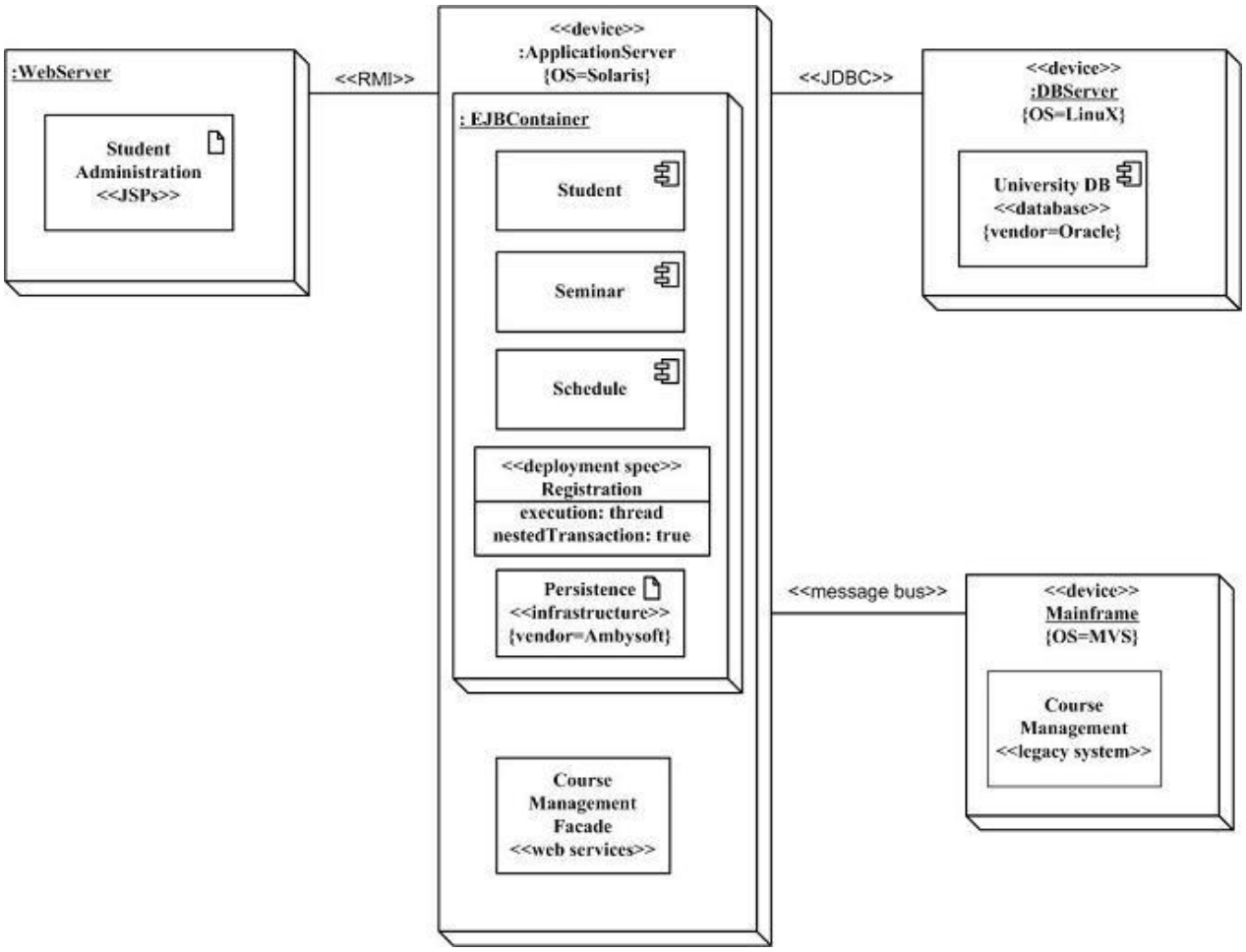
Component diagram (Diagram komponent)

Tímto diagramem se modelují komponenty a jejich nabízená či požadovaná rozhraní, které jsem zmíňil výše. Rozhraní které komponenta nabízí je na následujícím obrázku vyobrazeno jako „lízátko“ vyčnívající z komponenty. Rozhraní požadované je buď šipka k „lízátku“ a nebo prohloubený nástavec přiléhající na toto nabízené rozhraní.



Deployement diagram (Diagram nasazení)

Tento diagram se používá pro zobrazení, kde jsou které části systému nasazeny a zároveň jak se mezi jednotlivými častmi bude komunikovat (jakými protokoly)



Toto je diagram nasazení informačního systému školy.