Návrhové vzory. Komponentový návrh, volná vazba versus těsná vazba. Oddělení politiky od mechanismu. Návrh distribuovaných systémů s použitím webových služeb a servisně orientované architektury (SOA). (Y36SI3)

Slovníček pojmů

- **Návrhový vzor** (design pattern) - představuje obecné řešení problému, které se využívá při návrhu programů. Jedná se o popis řešení problému nebo šablonu, která může být použita v různých situacích. Objektově orientované návrhové vzory typicky ukazují vztahy a interakce mezi třídami a objekty, aniž by určovaly implementaci konkrétní třídy. Algoritmy nejsou považovány za návrhové vzory, protože řeší konkrétní problémy a nikoliv problémy návrhu.

- **Komponenta** - komponenta je samostatná jednotka softwarového systému (programu) nebo také modulární část systému. Důležité je, že komponenta je samostatná, poskytuje a vyžaduje určitá rozhraní a jako taková je nahraditelná (jinou komponentou se stejnými rozhraními). Komponenta zapouzdřuje veškerou svoji funkcionalitu a data v sobě (využití v při komponentově orientovaném programování, kdy lze různé menší komponenty spojovat dohromady a vytvářet tak funkčnost programu).

- **Vazba** (volná, těsná) - míra znalosti jedné třídy*komponenty o druhé.

- **Webová služba** - jednoduše funkce*program*služba dostupná přes síť. Webová služba je technologie*standard, kdy klient (my) zavolá server (webová služba) přes protokol HTTP a zpět získá požadovaný výsledek. Technologie webových služeb silně využívá jazyka XML, pro jejich popis pomocí WSDL souboru (Web Service Description Language) nebo pro jednotlivá volání webové služby pomocí SOAP zpráv (Simple Object Access Protocol). Prakticky to vypadá následovně: klient někde najde popis webové služby (WSDL), ze kterého vyčte jak má službu zavolat (název metody, parametry, ...), vytvoří příslušnou SOAP zprávu, kterou pomocí protokolu HTTP odešle na server, webová služba požadavek zpracuje a zpět odešle SOAP zprávu s odpovědí.

- **SOA** (Service Oriented Architecture) - servisně orientovaná architektura. V současnosti asi nejpopulárnější způsob jak vytvářet distribuované a volně vázané systémy. Do systému jsou přidávány služby (programy, nebo například webové služby) a spojením funkcí různých služeb získáváme celkovou funkci systému. Výhodou je možnost znovu využít funkcí již vytvořených komponent (služeb) pro další a další části systému.

- **zapouzdření** (encapsulation) - třída, případně komponenta, je dobře zapouzdřená, pokud zvenčí není vidět*znát její konkrétní implementace. To znamená třídu*objekt využíváme pomocí jejího rozhraní (metody), ale jak konkrétně objekt funguje nevíme. To nám umožňuje daný objekt nahradit úplně jiným (jiná implementace se stejným rozhraním), my (klient) nic nepoznáme, ale můžeme tím například získat mnohem větší výkon.

Návrhové vzory

Historicky se vymyšlení návrhových vzorů připisuje architektovi Christopherovi Alexanderovi, který se snažil využít návrhových vzorů při návrhu domů. Jemu se to mnoho nepodařilo, nicméně jeho myšlenky byly využity v mnoha disciplínách včetně informatiky. V počítačích přišel rozmach návrhových vzorů s vydáním knihy *Design Patterns: Elements of Reusable Object-Oriented Software* (tzv. kniha GoF - Gang of Four), ve které bylo popsáno základních 23 vzorů a od té doby bylo vymyšleno mnoho dalších.

Přinos návrhových vzorů spočívá v **přenosu zkušeností**. Zkušení a chytří architekti vymyslí nějaký zajímavý a šikovný návrh a pokud se v něm podaří nalézt určité obecné vlastnosti, lze ho **znovupoužít** v dalších programech. Tedy mladí a nezkušení programátoři nemusí znovu a znovu (a hlavně blbě) vymýšlet vlastní návrhy, ale mohou využít již ty vymyšlené, mnohokrát osvědčené. To znamená nejenom **kvalitnější** a rychleji doručený software, ale také **úsporu** při vývoji.

Návrhový vzor je šablona, která má většinou následující strukturu: **definice vzoru**, **popis problému**, **návrh řešení** a **výhody*nevýhody**. Jako programátoři pak můžeme dle definice nebo popisu

problému najít vzor, který řeší náš problém, a dle návodu ho použít (samozřejmě musíme zvážit, zda-li výhody v našem případě převýší nad nevýhodami).

Návrhové vzory jsou tzv. *"best practices"* při návrhu software, tedy jedná se o obecně **doporučenou** metodu, která když se **dobře použije**, pravděpodobně povede k **úspěšnému konci**. Zde je důležité zmínit, že každá taková metoda je **omezena svými podmínkami** použití, čili je důležité dobře zhodnotit, zda-li je pro nás vhodná a pokud ano, tak máme ve svém návrhu určitou jistotu (**dlouholetě ověřenou**). Některé vzory se mohou **zdánlivě** do našeho návrhu hodit, ale při bližším prozkoumání jsou naprosto nevhodné nebo ho **špatně aplikujeme**! Často je to tak se vzorem *Singleton* či *MVC* (Model-View-Controller).

S návrhovými vzory částečně souvisí s metodika **GRASP** (General Responsibility Assignment Software Patterns). Nejedná se o pravé návrhové vzory, ale o určité **principy**, které **řeší nějaký běžný softwarový problém** a jejich použitím se nám výrazně zjednodušuje život. Tyto vzory se také zaobírají správným rozdělením odpovědnosti mezi třídy a objekty.

Mezi tyto vzory patří:

- **Information Expert** - řeší kam máme delegovat zodpovědnost (rozuměj funkci). Principem je podívat se na danou zodpovědnost, zjistit jaká vyžaduje data a kde tato data jsou a to nám říká, kde by tato zodpovědnost měla být.
- **Creator** - řeší kde máme vytvářet objekty dané třídy. Pokud máme dvě třídy A a B a třída B obsahuje nebo využívá většinu inicializačních dat pro tvorbu objektu třídy A, pak třída B je vhodným místem pro tvorbu objektů třídy A.
- **Controller** - řeší umístění aplikační*rozhodovací logiky. Controller je obecně "vrstva" hned za uživatelským rozhraním, která přijímá různé události a deleguje práci na další objekty, toto celé koordinuje a řídí.
- **Low coupling** - princip který hodnotí a doporučuje **malou závilost** mezi třídami, **malý dopad změn** v jedné třídě na další a **velký potenciál znovupoužití** kódu.
- **High cohesion** - princip, který hodně souvisí s principem *low coupling* a oba se doplňují. Cohesive (soudržný) objekt je taková, který obsahuje pouze silně související a podobně zaměřené zodpovědnosti. Příkladem techniky, která zvyšuje *soudržnost* je rozdělení systému na subsystémy a třídy. Tento princip napomáhá jednoduššímu porozumění a udržovatelnosti.
- **Polymorphism** - tento princip říká, že zodpovědnost za **rozdíl v chování závislý na typu** je umístěna přímo v daných typech. V praxi je to **polymorfismus**.
- **Pure fabrication** - tento princip o hovoří o třídách, které **nemají reálný protějšek** v **problémové doméně**, ale díky jejich použití **zvyšujeme** soudržnost tříd (high cohesion), **snižujeme** závislosti (low coupling) a **zvyšujeme** potenciáln znovupoužití. Jsou to například různé třídy typu *Service.
- **Indirection** - tento princip navrhuje zavést určitého prostředníka (např. controller mezi modelem a pohledem), který snižuje závislosti a zvyšuje možnost znovupoužití.
- **Protected variations**

Návrhové vzory se dělí na:

- **creational** (zabývající se tvorbou objektů) - Abstract Factory (abstraktní továrna), Builder (stavitel), Factory Method (tovární metoda), Prototype (prototyp), Singleton
- **structural** (strukturální) - Adapter (adaptér), Bridge (most), Composite ("strom"), Decorator (dekorátor), Facade (fasáda), Flyweight ("muší váha"), Proxy
- **behavioral** (zabývající se chováním objektů) - Chain of Responsibility (řetěz odpovědnosti), Command (příkaz), Interpreter (interpret), Iterator (iterátor), Mediator (prostředník), Memento (pamětník), Observer (pozorovatel), State (stav), Strategy (strategie), Template Method (šablonová metoda), Visitor (návštěvník)

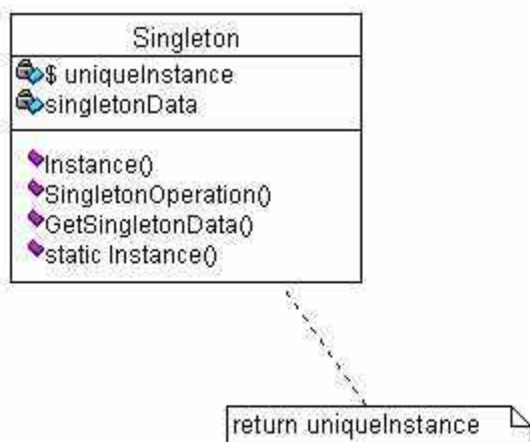
Popis některých vzorů

Toto jsou příklady některých návrhových vzorů, jejich mnohem více a pro jejich studium doporučuji buď knihy ve zdrojích, Wikipedii nebo Vico katalog (v odkazech). Nemá cenu zde opisovat to, co je napsáno jinde.

Singleton

- **Definice:** Zajišťuje, že třída má pouze jednu instanci a poskytuje k ní globální přístup.

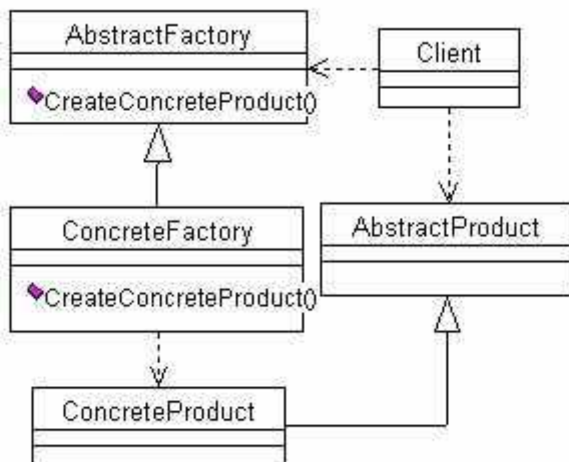
- **Problém:** Pro některé třídy je důležité mít pouze jednu instanci (například logovací služba, služba tisku, generátor ID, okenní systém - window manager). Pro zajištění jedné instance bychom mohli použít globální proměnnou, to nám umožní jednoduchý globální přístup, ale nezabrání nám to vytvořit více instancí. Lepší řešení je nechat třídu samotnou držet jedinou instanci sama sebe, tím zabráníme vytvoření jiné instance a zároveň třída může poskytnout globální přístup k této jediné instanci.
- **Řešení:** (Java) Například pro třídu Auto uděláme **private** konstruktor (aby nebylo možno vytvářet instance), vytvoříme **private static Auto auto** atribut, který bude držet naši jedinou instanci a **public static Auto getInstance()** metodu, která bude naši jedinou instanci vracet. Jediná instance se vytvoří při inicializaci třídy (ta má přístup k private konstruktoru), nikdo jiný ke konstruktoru nemá přístup, tudíž nemůže vytvořit jinou instanci a globální přístup k té jediné je zajištěn pomocí statické metody třídy (Auto.getInstance()) odkudkoliv v programu.



- **Výhody*nevýhody:** Některé služby musí být opravdu jedinečné a často k nim potřebujeme přistupovat, tudíž Singleton je pro takovou situaci vhodný. Bohužel, problémem může být právě globální dostupnost, která zbytečně svazuje program s danou implementací.

Abstract Factory

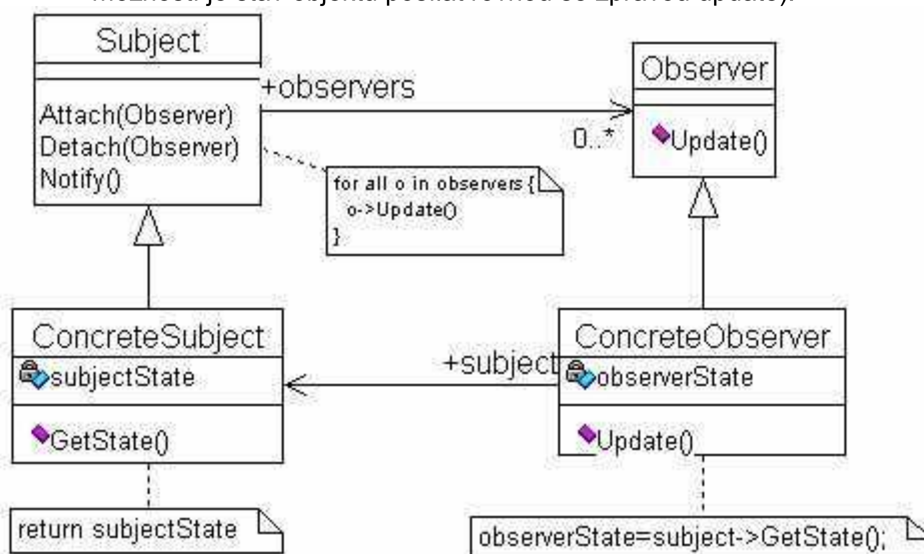
- **Definice:** Poskytuje rozhraní pro vytváření rodin souvisejících nebo závislých objektů aniž bychom specifikovali jejich konkrétní třídu.
- **Problém:** Grafické rozhraní má více tzv. look-and-feels (vzhled). Look-and-feel definuje vzhled a chování jednotlivých prvků rozhraní (tlačítko, scrollbar, ...). Aby aplikace byla přenositelná a bylo možno tyto look-and-feely jednoduše měnit, není možno natvrdo do kódu napsat třídy konkrétního look-and-feelu. Toto je možno vyřešit vytvořením abstraktní továrny, která definuje metodu pro vytvoření každého typu prvku rozhraní a konkrétní implementace továrny pro daný look-and-feel pak vytváří konkrétní instance těchto prvků. Klient není závislý na konkrétní implementaci, volá metody definované v abstratkní továrně pro vytvoření prvků rozhraní a pouhým nahrazením továrny za jinou lze změnit celý look-and-feel aplikace za běhu.
- **Řešení:** (Java) Vytvoříme abstraknít třídu AbstractFactory obsahující metody getButton, getScrollBar, poté vytvoříme konkrétní implementace pro Windows a Mac, třídu WindowsFactory extends AbstractFactory implementující metody v rozhraní pro platformu Windows, třídu MacFactory extends AbstractFactory implementující metody rozhraní pro platformu Mac. Při spuštění programu pak máme proměnnou AbstractFactory, do které vložíme instanci buď WindowsFactory nebo MacFactory a vytváříme rozhraní pomocí dostupných metod, záměnou instance v proměnné pak můžeme změnit vzhled, jednoduše a rychle.



- **Výhody*nevýhody:** Jasnou výhodou je oddělení rozhraní od implementace, získáme tak volnou vazbu komponent. Nevýhodou je možnost přidávat další metody do abstraktní továrny (když chceme přidat další prvek rozhraní), znamenalo by to opravit všechny existující impolementace!

Observer

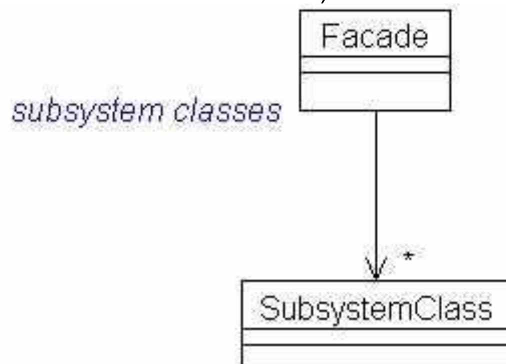
- **Definice:** Definuje one-to-many závislost mezi objekty tak, aby když objekt změní stav, všichni na něm závislí jsou o tom informování.
- **Problém:** Při návrhu se většinou snažíme systém postavit jako sadu oddělených tříd*komponent, které spolu interagují, nicméně tyto třídy spolu potřebují komunikovat co se týče jejich stavu, ale to nemůžeme*nechceme udělat tak, že bychom je natvrdo svázali (těsná vazba). Řešením je vzor pozorovatel, který definuje pozorovaný objekt, který je pozorován více pozorovateli, kteří jsou informování o každé změně v pozorovaném objektu.
- **Řešení:** (Java) Máme rozhraní Observer s metodou update, která je zavolána, když chceme pozorovatele informovat o nějaké změně. Dále máme rozhraní Subject, které má základní metody pro příhlášeník odběru zpráv, odhlášení z odběru a vyvolání notifikace (odeslání zprávy o změně). Konkrétní pozorovatel se pak přihlásí u konkrétního subjektu pro příjem zpráv o změně a pokud se daný objekt změní, zavolá metodu update pozorovatele, který si pak vyžádá změnu stavu metodou např. getState (jakou metodu zavoláme pro zjištění stavu objektu záleží na konkrétní instanci subjektu, toto není definováno v rozhraní Subject, další možností je stav objektu posílat rovnou se zprávou update).



- **Výhody*nevýhody:** Výhodou je volná vazba mezi komponentami, nevýhodou, že pozorovatelé o sobě navzájem nevědí.

Facade

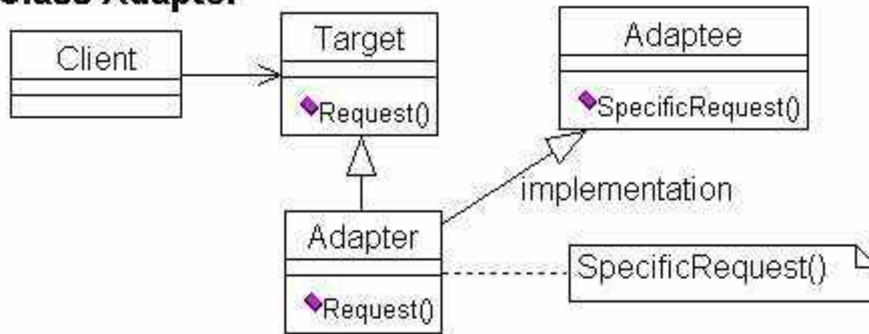
- **Definice:** Poskytuje sjednocené rozhraní k sadě různých rozhraní subsystému. Fasáda definuje rozhraní na vyšší úrovni, které tak čini subsystém jednodušeji použitelným.
- **Problém:** V programu máme určitý subsystém, u kterého bychom rádi měli dvě úrovně přístupu. Jednu pro klienty, kteří pouze vyžadují určitou funkci a druhou pro pokročilé klienty, kteří zároveň vyžadují i různé další informace a funkce. Aby se všichni nemuseli učit a porozumět složitému rozhraní pro pokročilé klienty, vytvoříme nad subsystémem fasádu, která poskytne zjednodušený pohled na subsystém a je mnohem jednoduše použitelná. Dalším příkladem může být subsystém, který se v průběhu let různými úpravami stal příliš složitý a rádi bychom jeho rozhraní zjednodušili*vylepšili.
- **Řešení:** (Java) Implementace je velmi jednoduchá. Klientům poskytneme třídu s nějakým rozhraním (které nějak vhodně abstrahuje a zjednodušuje subsystém) a v této třídě pak veškeré požadavky delegujeme na původní subsystém. Čili funkce zůstává stejná, ale použití je mnohem jednodušší (např. namísto volání pěti metod v určitém pořadí stačí zavolat jednu a ona to udělá za nás).

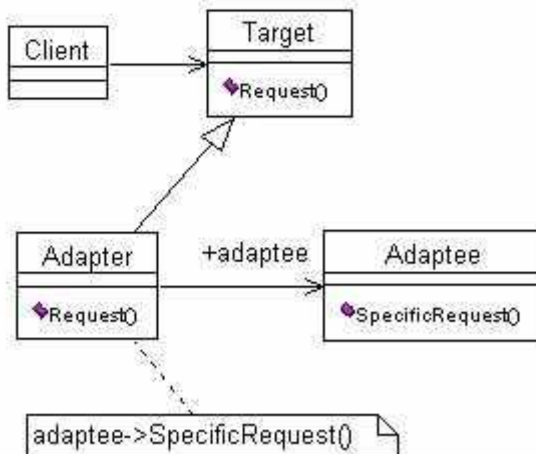


- **Výhody*nevýhody:** Výhodou je snížení vazeb mezi klientem a subsystémem, podporo vrstvení systému a možnost poskytnout určitým klientům vhodnější rozhraní.

Adapter

- **Definice:** Konvertuje rozhraní třídy na rozhraní jiné třídy, které je očekáváno klientem. Adaptér umožňuje spolupráci tříd, které by spolu jinak nemohli spolupracovat vzhledem k nekompatibilním rozhraním.
- **Problém:** Potřebujeme využít nějakou třídu, jejíž rozhraní s námi není kompatibilní. V takovém případě můžeme vytvořit adaptér, který upraví rozhraní dané třídy (poskytne nám vhodné rozhraní a volání deleguje na původní třídu) pro naše potřeby.
- **Řešení:** (Java) Implementačně existují dva přístupy (Adapter Class vs. Adapter Object). Adapter Class se implementuje tak, že máme nějaké požadované rozhraní a adaptované rozhraní a od obou podědíme náš adaptér. Adapter Object se implementuje pomocí **kompozice**, kdy dědíme pouze od požadovaného rozhraní a instanci adaptované třídy máme uloženou v atributech (delegujeme na ní volání).



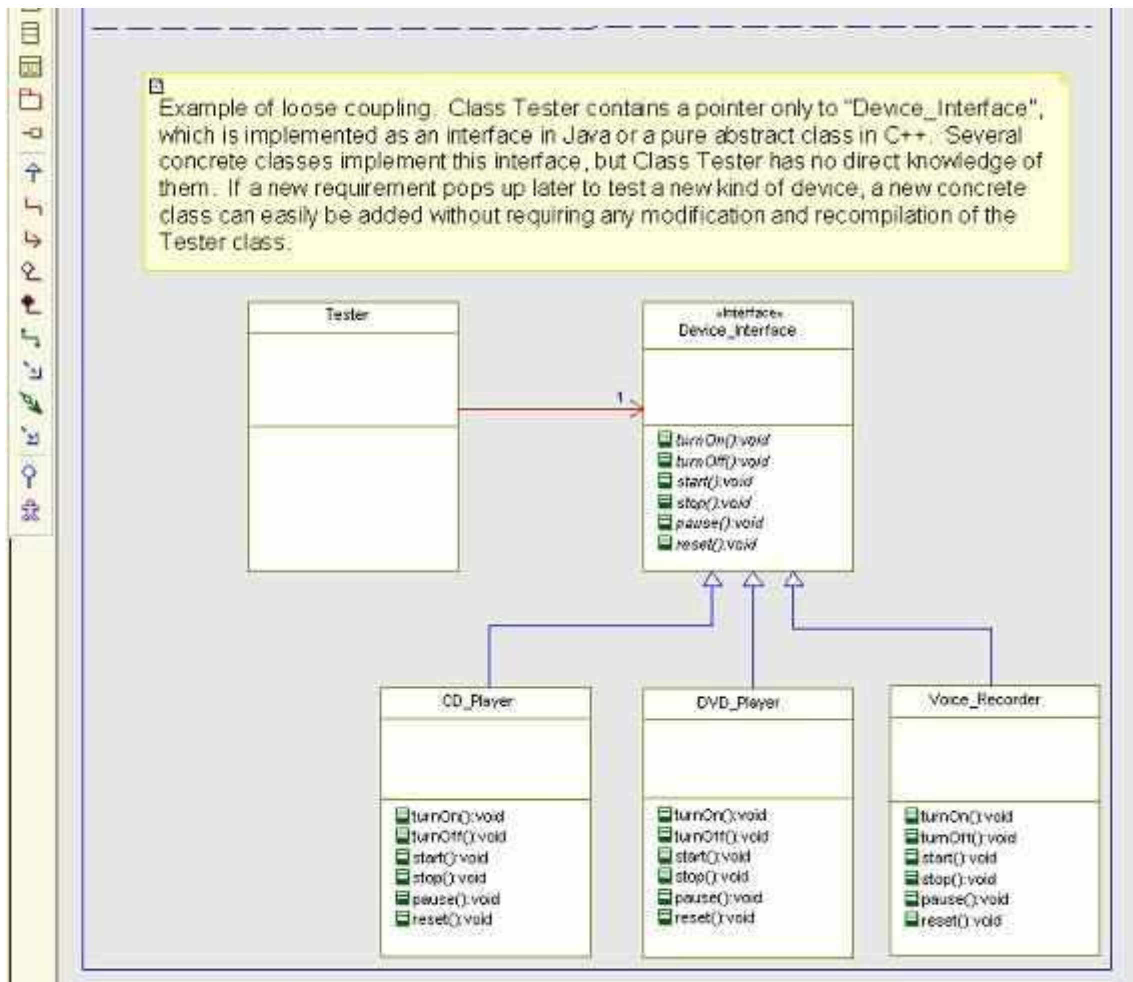


- **Výhody*nevýhody:** Zvyšuje znovupoužitelnost tříd díky oddělení rozhraní a implementace.

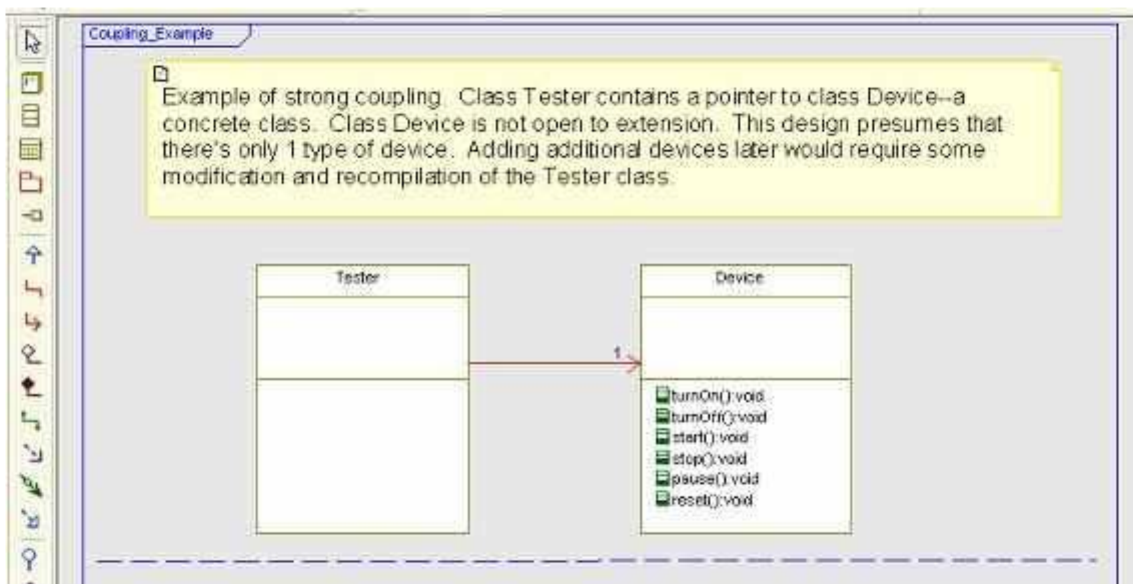
Komponentový návrh
Hlavní myšlenkou komponentového návrhu je tzv. **separation of concerns** ("oddělení starostí"), to znamená rozdělení celkové funkčnosti systému na jednotlivé **komponenty**, které budou **sdružovat spolu související funkce** (tzv. **high cohesion** - princip viz. GRASP). Tím vytvoříme sadu **volně vázaných** (tzv. **low coupling** - princip viz. GRASP) komponent a zároveň komponenty, které jsou dobře **zapouzdřené** (encapsulated) a **soudržné** (cohesive). Výhodou takového přístupu je možnost znovu využít již vytvořené komponenty, jasné oddělení povinností (zároveň se tím dobře oddělují chyby, chybu můžeme hledat v dané komponentě a nemusíme řešit, jaké vztahy a s jakými komponentami má) a celkově jednodušší správu systému. Nevýhodou je *vyšší režie*, namísto přímého volání voláme přes různé prostředníky zajišťující komunikaci komponent. Příkladem takového komponentového návrhu (architektury) je **SOA**.

Volná vs. těsná vazba
Vazba je **míra znalosti jedné třídy o druhé**. Tím není myšleno znalost metod a atributů třídy (tohoto se týka zapouzdření), ale spíše znalost třídy samotné (znalost toho, že vůbec existuje).

- **Volná vazba** (low coupling) - závisla třída **neobsahuje** ukazatel*referenci na konkrétní třídu s požadovaným chováním, ale obsahuje referenci pouze na **rozhraní** (interface), které může být implementováno mnoho různými třídami. To umožňuje **rozšiřitelnost**, nová konkrétní implementace může být jednoduše přidána aniž by bylo nutné znovu kompilovat nebo nějak měnit závislou třídu.



- **Těsná vazba** (high coupling) - závislá třída **obsahuje** ukazatel*referenci přímo na konkrétní třídu, která poskytuje požadované chování.



Míra vazby se dá "měřit" například tak, že si řekneme, jak moc by určité změny ovlivnila korektnost komunikace existujících tříd.

Oddělení politiky od mechanizmu
Způsob návrhu, který říká, že **mechanizmus** (implementace) **by neměl diktovat nebo příliš omezovat politiky** (pravidla). O tomto způsobu se většinou hovoří v souvislosti s bezpečností (autentizace a autorizace).

Příkladem budiž oddělení mechanizmu a politik u čipových karet. Mechanizmus (čtečka, zámek) nerozhoduje o tom, kdo má a nemá přístup. Toto rozhodnutí dělá centralizovaný server, který pravděpodobně rozhoduje na základě pravidel v databázi (kdo má kam přístup - politiky). Konkrétní rozhodnutí, kdo má kam přístup, mohou být **nezávislé** přidávána a odebírána pouze změnou databáze. A pokud by i schéma databáze bylo nevyhovující, lze ho nahradit zatímco mechanizmus (zámky, čtečky) funguje nezávislé dál.

Návrh distribuovaných systémů s pomocí webových služeb a SOA
Distribuovaný systém se skládá z **autonomních počítačů** propojených počítačovou **sítí**. Tyto počítače spolu **komunikují**, aby dosáhly požadovaného výsledku (své funkce). Na těchto počítačích běží speciální software, tzv. **distribuovaný program**. Takový systém lze využít pro výpočet náročných úloh rozdělením úlohy na menší části a rozdistribuováním těchto částí na jednotlivé počítače. Dnes má pojem distribuovaný systém mnohem širší význam, nejenom geograficky oddělené počítače, ale i oddělené procesy komunikující **posíláním zpráv** běžící na jednom stroji.

Příkladem distribuované systému je i architektura **SOA**, která se skládá z jednotlivých **služeb** (např. webových), které mezi sebou **komunikují** (posíláním zpráv) a doručují tak požadavanou funkcionalitu. Nové funkce lze jednoduše přidávat přidáním služby nebo novou interakcí již existujících služeb.

Webová služba
- **"Big web service"** - standardní webová služba postavená na **standardech WSDL a SOAP**
- **RESTful** - dnes velmi populární, využívá protokolu **REST** (REpresentational State Transfer), který je mnohem jednodušší než SOAP (především není tak ukecaný, protože nevyužívá XML) a nevyžaduje popis služby pomocí WSDL. REST je jednoduchý textový protokol využívající metod PUT, GET, POST a DELETE protokolu **HTTP**.

SOA
Mezi podniky dnes velmi populární přístup k tvorbě distribuovaných systémů. Je to velmi vágně definovaný pojem, pod kterým si každý představuje trochu něco jiného a konkrétní implementace SOA je asi pokaždé jiná. Jednou z možných implementací je pomocí webových služeb, které reprezentují **funkční stavební bloky** přístupné přes **standardní internetové protokoly** a **nezávislé na programovacím jazyku či operačním systému**. Služby buď představují úplně nové služby nebo mohou pouze obalovat již existující "legacy" systémy.

Důležité je, že služby jsou **samostatné**, **zapouzdřené**, **volně vázané**, mají **kontrakt** (poskytovaná a vyžadovaná rozhraní), dají se **skládat** a **znovupoužívat** v různých částech systému.

Každá služba má jednu ze dvou rolí:
- **poskytovatel** - poskytuje nějakou službu
- **spotřebitel** (klient) - využívá služeb jiné webové služby

Základní pricinpy SOA:
- znovupoužitelnost komponent - vytvářené komponenty by měly být co nejobecnější, abychom je mohli znova a znova používat v různých budoucích aplikacích.
- granularita - systém by měl být rozdělen na malé části, dalo by se říct, že čím menší, tím větší bude šance na jejich znovupoužití, na druhou stranu čím menší části, tím větší bude komunikační režie!
- modularita - systém by měl být modulární, tedy je možno různé komponenty připojovat a odpojovat a vše funguje bez problému dále.
- interoperabilita - jednotlivé moduly by měly využívat standardů a běžných rozhraní*formátů, aby mezi sebou mohli komunikovat a nebylo třeba vytvářet speciální adaptéry.
- dodržení standardů - souvisí s interoperabilitou a budoucí možností rozšíření architektury.
- identifikace a zařazení služeb - SOA systém musí zahrnovat nějaký systém pro zatřídění a možnost identifikace dostupných služeb (přihlášení služby, odhlášení služby, vyhledání služby).

- monitorování - je samozřejmě nutné celý systém monitorovat, jak z hlediska bezpečnosti, tak výkonu.

SOA lze implementovat i pomocí dalších technologií:
- SOAP, RPC (Simple Object Access Protocol, Remote Procedure Call)
- REST (Representational State Transfer)
- DCOM (Distributed Component Object Model)
- CORBA (Common Object Request Broker Architecture)
- DDS (Data Distribution Service)
- WCF (Windows Communication Foundation)