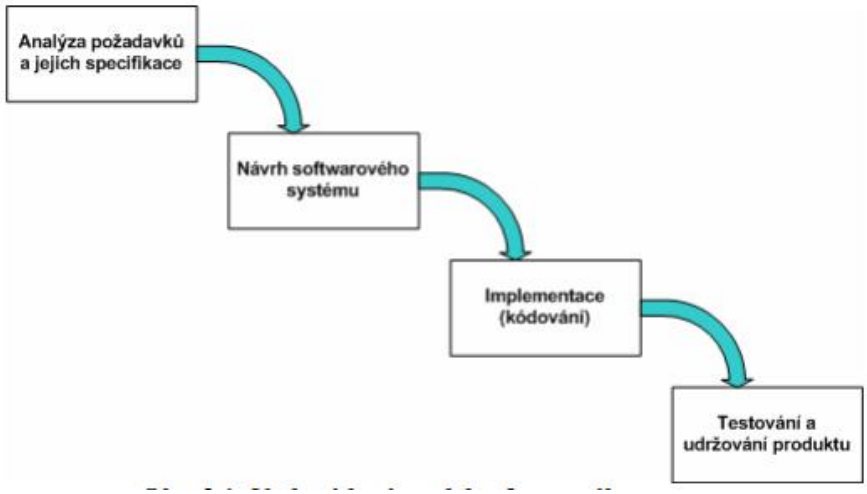
Definice softwarového procesu

Softwarový proces je po částech uspořádaná množina kroků směřujících k vytvoření nebo úpravě softwarového díla. Je-li cíl vytvoření softwaru, pak mluvíme o **softwarovém procesu**. Disciplína zabývající se problémy softwarového procesu se nazývá **softwarové inženýrství**. Je to postup činností nutných k vytvoření nebo úpravě softwarového produktu.

Modely softwarového procesu

Vodopádový model



- Základní, nejstarší model
- Princip modelu je v tom, že následující množina činností nemůže započít dříve, než skončí předešlé činnosti. Výsledky předchozí fáze jsou na vstupu následující fáze.
- Vychází z předpokladu, že lze dopředu přesně definovat všechny požadavky, které se navíc dále nebudou měnit.
- Příliš velká prodleva mezi zadáním a vytvořením spustitelného systému.
- Kvalita výsledku závisí na úplnosti a korektnosti zadání, kvalitu nelze odhalit před dokončením.
- **Inkrementální model –** postupné vytváření SW v iteracích, kdy jedna verze obsahuje pouze nějakou funkcionalitu

Spirálový model

- zahrnuje do svého životního cyklu další fáze jako je vytvoření a hodnocení **prototypu ověřující funkcionalitu cílového systému**, přičemž každý cyklus nabaluje další požadavky specifikované zadavatelem
- postup do další fáze závisí na důsledně provedené analýze všech rizik a možných problémů
- Model je založen na iterativním přístupu a především zavádí opakovanou analýzu všech rizik. Lépe se tak vyrovnává s pozdější úpravou požadavků
- Hlavní myšlenkou je zde navazování nových částí na již důkladně prověřený základ

RUP model (Rational Unified Process)

Process RUP navrhla firma Rational. Definuje disciplinovaný přístup k přiřazování úkolů a zodpovědností v rámci vývojové organizace. Jeho cílem je zajistit vytvoření produktu vysoké kvality požadované zákazníkem v rámci predikovatelného rozpočtu a časového rozvrhu.

Základní principy RUP:

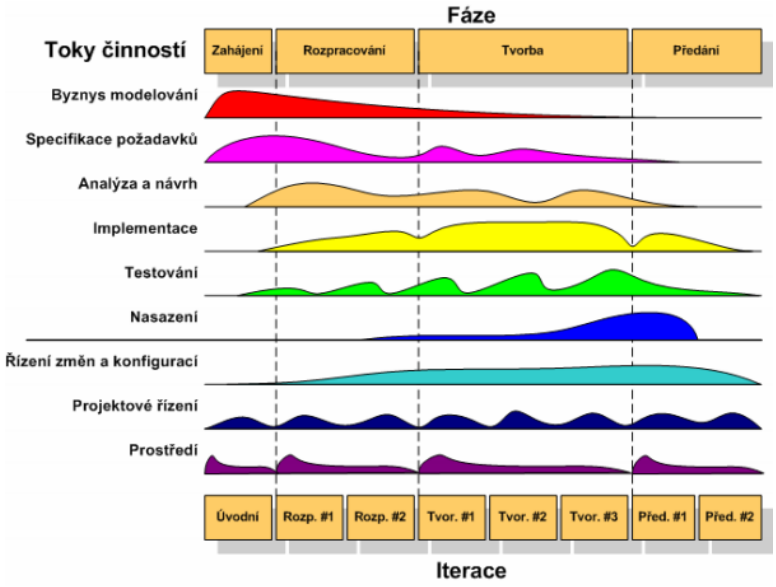
- **Iterační vývoj** – Na konci každé iterace je spustitelný kód (verze). Softwarový systém je tak vyvíjen ve verzích, které lze průběžně ověřovat se zadavatelem a případně jej pozměnit pro následující iteraci.
- **Správa požadavků** - Proces RUP popisuje jak požadavky na softwarový systém doslova vylákat od zadavatelů, jak je organizovat a následně i dokumentovat
- **Užívání existujících SW komponent** - systematický přístup k definování architektury využívající nové či již existující komponenty.
- **Vizualizace modelu SW systému** - vizualizace struktury a chování výsledné architektury produktu a jeho komponent s využitím UML.
- **Průběžné ověřování kvality** - je součástí všech aktivit a týká se všech účastníků vývoje. Kvalita se ověřuje pomocí objektivních měření a kritérií.
- **Řízení změn** - umožňuje zaručit, že každá změna je přijatelná a všechny změny systému jsou sledovatelné.

Cyklus

- Vede k vytvoření verze systému, kterou lze předat uživatelům
- Dělí se na následující **fáze**:
    1. **Zahájení** – původní myšlenka je rozpracována do vize koncového produktu a je definován rámce toho, jak bude systém vyvíjen a implementován
    2. **Rozpracování –** podrobná specifikace požadavků
    3. **Tvorba –** implementace
    4. **Předání –** zahrnuje i testování, školení atd.

Každá fáze je složena z několika iterací: Iterace - úplná vývojová smyčka vedoucí k vytvoření spustitelné verze systému reprezentující podmnožinu vyvíjeného cílového produktu a která je postupně rozšiřována každou iterací až do výsledné podoby

Obr. 2.4: Iterace vývoje produktu



Obr. 2.3: Schématické vyjádření procesu RUP

Úrovně vyspělosti

Úroveň definice a využití softwarového procesu je hodnocena dle stupnice SEI (Software Engineering Institute) 1 - 5 vyjadřující vyspělost firmy či organizace z daného hlediska.
Tento model hodnocení vyspělosti a schopností dodavatele softwarového produktu se nazývá **CMM (Capability Maturity Model)** a jeho jednotlivé úrovně lze stručně charakterizovat asi takto:

1. **Počáteční (Initial)** – firma nemá definován SW proces a každý projekt je řešen případ od případu
2. **Opakovatelná (Reteatable)** – firma identifikovala v jednotlivých projektech opakovatelné postupy a tyto je schopna reprodukovat v každém novém projektu
3. **Definovaná (Defined)** – SW proces je definován (dokumentován) na základě integrace dříve identifikovaných opakovatelných kroků
4. **Řízená (Managed)** – na základě definovaného SW procesu je firma schopna jeho řízení a monitorování
5. **Optimalizovaná (Optimized)** – zpětnovazební informace získána dlouhodobým procesem monitorování SW procesu je využita ve prospěch jeho optimalizace

Sběr a analýza požadavků

Cílem specifikace požadavků je popsat co má softwarový systém dělat prostřednictvím specifikace jeho funkcionality. Modely specifikace požadavků slouží k odsouhlasení zadání mezi vývojovým týmem a dodavatelem.

Sběr a analýza požadavků je první fáze vývoje softwaru. Cílem je definovat požadavky na software a popsat jeho funkčnost.

Modely, které jsou v rámci specifikace činností vytvářeny, vychází z tak zvaných případů užití (Use Case), které jsou tvořeny:

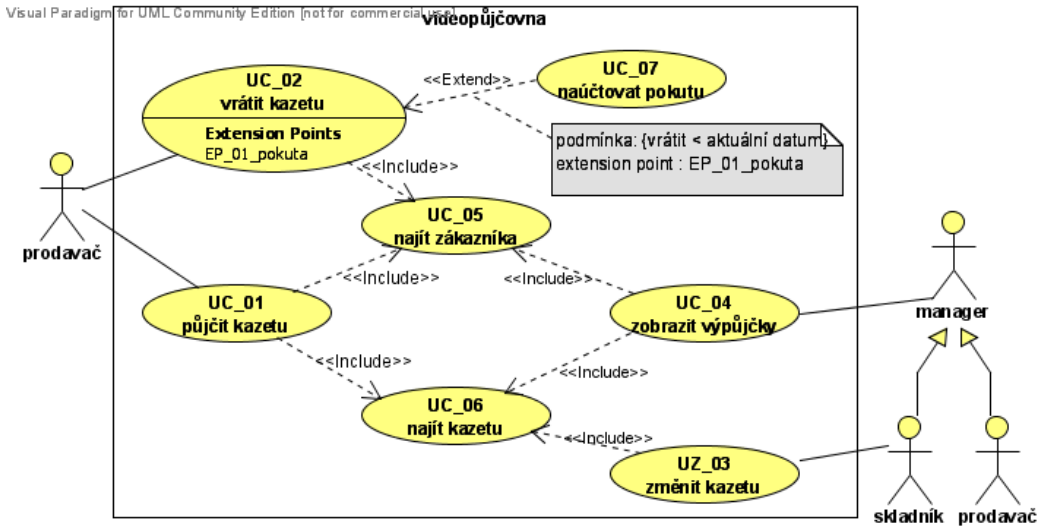
- **Aktéry –** definují uživatelé či jiné systémy, kteří budou vstupovat do interakce s vyvíjeným softwarovým systémem.
- **Případy užití –** specifikují vzory chování – posloupnost interakcí mezi aktérem a systémem

Pro potřeby sestavení modelů specifikace požadavků využívá dvou typů diagramů: **Diagram případů užití (Use Case diagram)** a **Sekvenční diagram.**

Diagramy UML

Use-Case diagram

Účelem use-case je definovat co existuje vše systému (aktéři) a co má být systémem prováděno (případy užití) a přístup aktérů k systému. Vstupem je **byznys model** (modely podnikových procesů). Výsledkem analýzy těchto procesů je **seznam požadovaných funkcí** softwarového systému.



Součástí diagramu je:

- **Aktér** – ten co systém používá, je znázorněn panáčkem vně systému, co přistupuje k různým případům užití
- **Případ užití** – znázorňuje funkci systému. Znázorněn je oválem uvnitř systému
- **Relace –** vazby a vztahy mezi aktéry a případy užití (šipky, čáry). Rozlišujeme několik typů relací:
    - Relace **používá** označována klíčovým slovem <<uses>> vyjadřuje situaci, kdy určitý scénář popsaný jedním případem užití je využíván i jiným případem užití
    - Relace **rozšiřuje** (<<extends>>) vyjadřuje situaci, kde určitý případ užití rozšiřuje jiný či představuje variantní průchody jim popsaným scénářem. Jeden proces může být rozšířen o jiný. (např. Otevřít dokument → Import z jiného formátu)
    - Relace **zahrnuje** (<<include>>) – jeden proces se využívá i v rámci jiného. Případ užití napojený pomocí vazby <<include>> se spustí vždy, když je spuštěn případ, na který je napojen
    - Relace **přístupu k systému** – aktér přistupuje k systému. Je znázorněn plnou čarou se šipkou.
    - Relace **specializace/generalizace** vyjadřuje dědičnost mezi objekty.

Jednotlivé případy užití se skládají z textové specifikace, ve kterých by mělo být popsáno:

- Co systém dělá (ale ne jak)

- Jak a kdy činnost začíná a končí
- Kdy má systém interakci s aktérem
- Které údaje jsou měněny
- Jaké kontroly vstupních údajů jsou prováděny
- Základní, alternativní a chybové průběhy

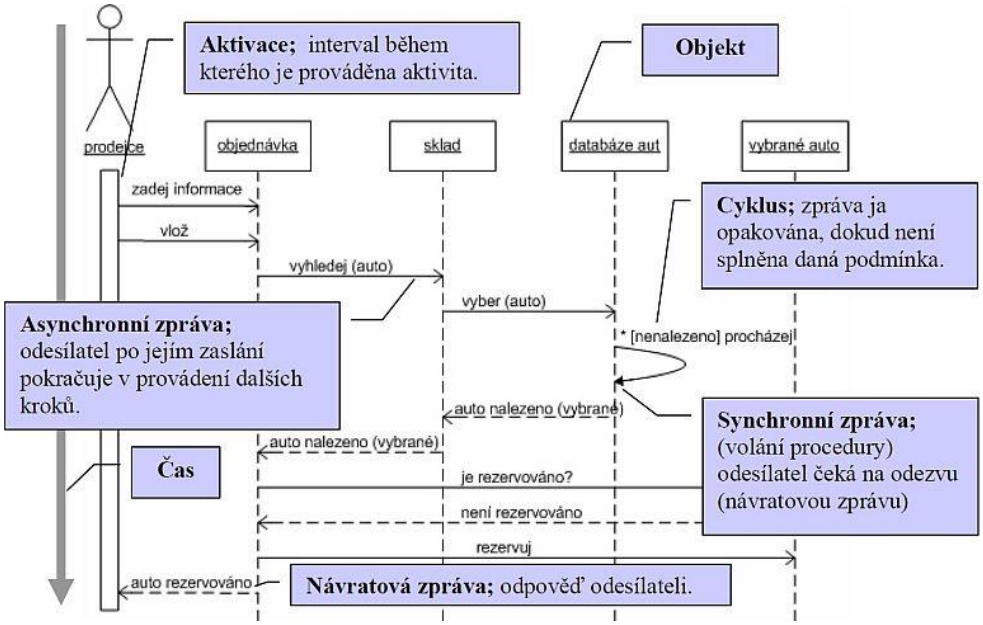
Sekvenční diagram

Objekty v systému zodpovídají za plnění požadované funkcionality poskytnutím svých služeb v komunikaci s jinými objekty, dokáže tedy realizovat jednotlivé scénáře. **Sekvenční (interakční) diagram** definuje interakce mezi objekty, které vedou ke splnění funkcionality.

Komunikace objektů je znázorněna v čase, takže z diagramu lze vyčíst i životní cyklus objektu. Diagram popisuje, jaké zprávy jsou mezi objekty zasílány z pohledu času. Tok času je od shora dolů. Je tvořen objekty uspořádanými do sloupců a šipky mezi nimi odpovídají vzájemně si zasíláním zprávám. Zprávy mohou být následující:

- **Synchronní** – odesílatel čeká na odpověď
- **Asynchronní** – odesílatel nečeká na odpověď a pokračuje ve vykonávání své činnosti.

Souvislé provádění nějaké činnosti objektu vyjadřuje svisle orientovaný obdelník.



Návrh

Cílem analýzy a návrhu softwarového produktu je ukázat, jakým způsobem bude produkt realizován v implementační fázi. Cílem fáze návrh je vytvoření **modelu návrhu** – upřesnění modelu analýzy ve světle skutečného implementačního prostředí. Model návrhu tak představuje abstrakci zdrojového kódu. **Určuje, jak bude zdrojový kód strukturován a napsán.** Vzniká abstrakce zdrojového kódu výsledného softwaru

Implementační prostředí představuje architekturu systémů určeného pro provoz vyvíjeného softwaru, přičemž je vhodné maximálně využít služeb, které v tomto prostředí existují.

Úkoly návrhu:

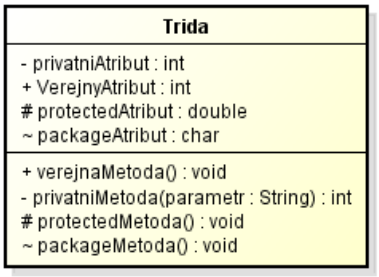
- Definice systémové architektury
- Identifikace návrhových vzorů ve vyvíjeném SW
- Definice softwarových komponent a jejich znovupoužití

UML Diagramy

Diagram tříd

Specifikuje množinu tříd, rozhraní a jejich vzájemné vztahy. Tyto diagramy slouží k vyjádření statického pohledu na systém. Používá se při návrhu aplikace. Diagram tříd je návod pro programátora, takže jeho práce je jen přepsat diagram do kódu. Zdrojem informací pro diagram tříd jsou diagramy sekvenční a spolupráce.

Základem je **třída,** která obsahuje:

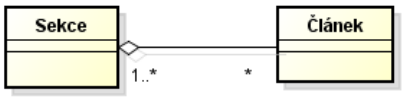


- **Název**
- V prostřední části jsou **atributy** s datovými typy
- Před každým atributem je modifikátor přístupu:
  - – privátní atribut
  - + veřejný atribut
  - # protected atribut
  - ~ atribut viditelný v rámci balíku
- V poslední části jsou **metody** a jejich návratový typ

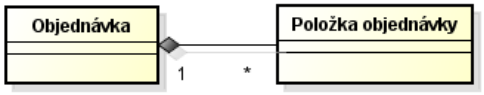
Vztahy mezi třídami

- **Asociace** – určuje základní vztah mezi dvěma entitami (posílají si zprávy). Ty mohou existovat nezávisle na sobě. U čáry lze napsat i **roli –** definuje specifické chování objektu v daném kontextu jeho použiti
  - Definováno plnou čárou
  - Šipka určuje, jaká třída si uchovává odkaz na druhou třídu

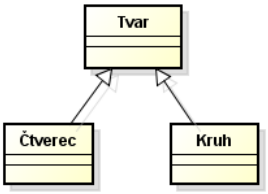
- **Agregace** – reprezentuje vztah typu celek – část. Prázdný kosočtverec je umístěn u té entity, která reprezentuje celek. Z hlediska implementace je to entita, která drží kolekci. Entita, která tvoří část, může existovat sama o sobě a být součásti i jiných kolekcí.



- **Kompozice** – podobná agregaci, avšak reprezentuje silnější vztah. Část nemá bez celku smysl. Pokud zanikne celek, zanikají automaticky i jeho části.



- **Generalizace** – dědičnost, kdy jedna entita dědí od druhé



- **Realizace** – přerušovaná čára s prázdnou šipkou; když nějaká třída implementuje rozhraní

Multiplicita

- Můžeme uvést u asociace, agregace a kompozice, udává, kolik může mít třída jiných tříd ve vztahu
    - 1 – označuje konkrétní hodnotu – právě 1
    - * - označuje libovolný počet (i 0)
    - 1..* - interval

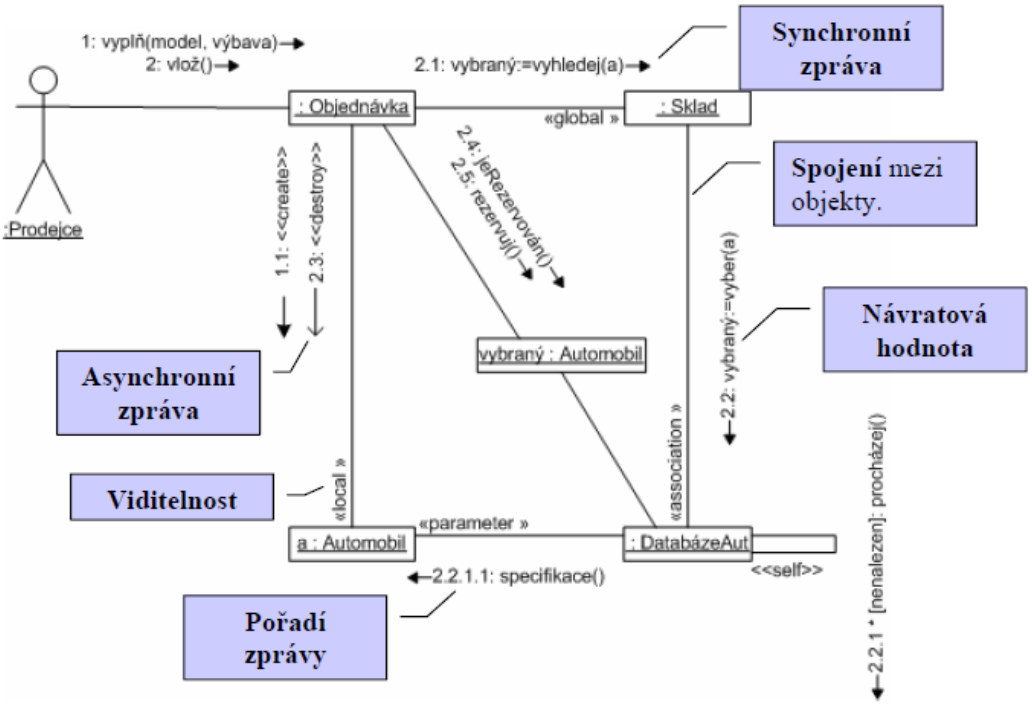
Sekvenční diagram

Viz předešlá otázka 2

Diagram spolupráce

Spolu se sekvencím diagramem patří do skupiny **interakčních diagramů**. Použití vhodné tam, kde chceme zdůraznit strukturální aspekty spolupráce – **kdo s kým komunikuje**. Nezobrazují časové souvislosti. Objekty si mohou posílat zprávy. Diagram spolupráce ukazuje objekty a spojení a zprávy, které si objekty posílají. Skládá se z:

- **Objekt**
- Mezi objekty jsou **spojení**
- **Zpráva** je zakreslená jako šipka (plná = synchronní, normální šipka = asynchronní)
- U šipky pak je text **casovaPosloupnost návratováHodnota := fce(argument1, argument2, ...)**
- Časová posloupnost zaslání zpráv je vyjádřena pořadovým číslem.
- Návratová hodnota je vyjádřena operátorem přiřazení :=



Obr. 5.3.2: Diagram spolupráce

Stavový diagram

Zobrazuje životní cyklus objektu, události způsobující přechody z jednoho stavu do jiného a akce, které vyplývají z této změny stavu. Popisuje celý životní cyklus objektů z pohledu dynamického chování. Sekvenční diagram a diagram spolupráce dokážou modelovat pouze určité časové snímky. Diagram je sestavován pro každá objekt (pro jeho třídu).

Vyjadřuje stavy určitého objektu a přechody mezi těmito stavy. Obsahuje:

- **Stav** – situace, kdy objekt splňuje nějakou podmínku; počáteční stav je označen plným kolečkem a koncový stav kolečkem s puntíkem.
- **Průchod** – spojení mezi dvěma stavy; objekt přejde z jednoho stavu do druhého za splnění určitých podmínek
- **Volba** – rozděluje přechod na dva segmenty. Každá větev má své podmínky (prostě if)
- **Rozvětvení** – Použije se v situaci, kdy jeden zdrojový stav je rozdělen na dva čí více cílových stavů
- **Spojení** – použije se, když více zdrojových stavů přechází do jednoho společného cílového stavu.

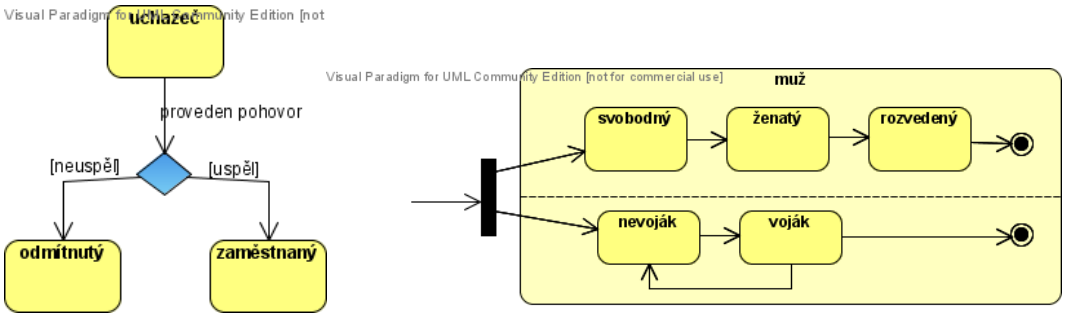


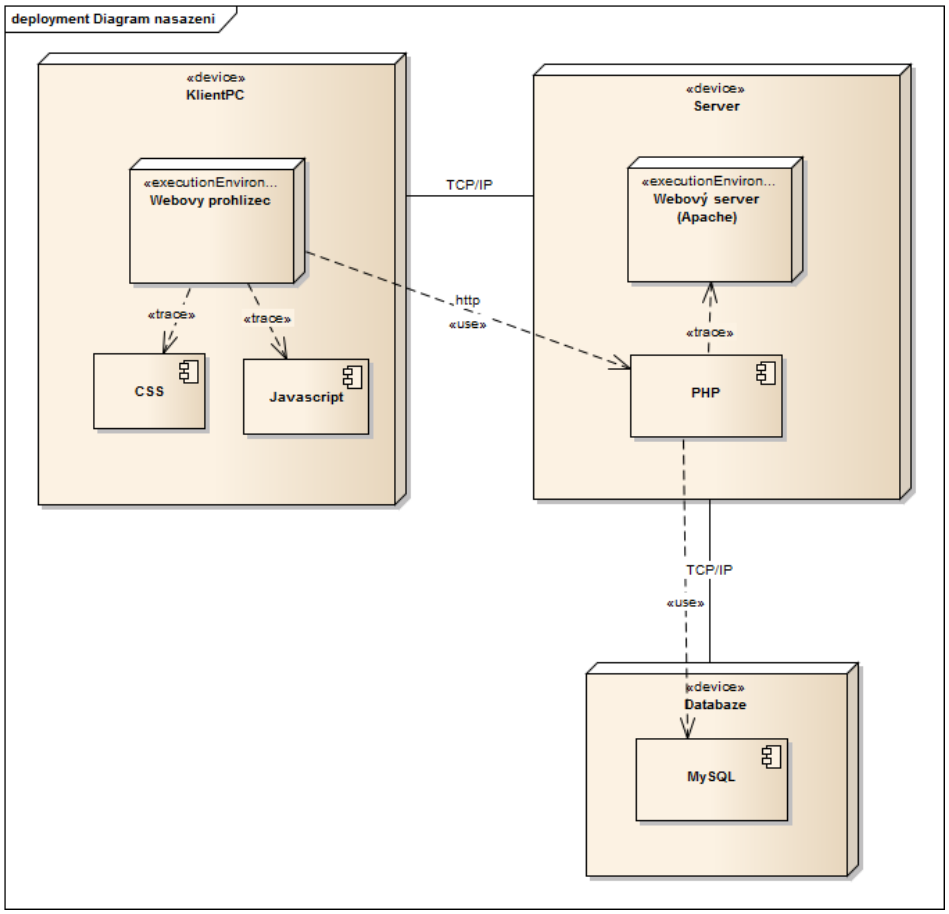
Diagram nasazení

Smyslem tohoto modelu je specifikovat fyzickou strukturu výpočetních prostředků, které budou použity pro provoz SW. Popisuje konfiguraci technických prostředků umožňujících běh SW systému. Zachycuje rovněž rozmístění implementovaných SW komponent na jednotlivé technické prostředky (server, počítač) - ukazuje rozložení jednotlivých softwarových komponent na hardwarových zdrojích (ale to bych spíš řekl, že to je záležitost implementační části, ne návrhové)

Základním prvkem diagramu nasazení jsou tzv. **uzly** (nodes), které jsou vzájemně propojeny **komunikačními cestami**.

Uzly mohou být typu:

- **<<device>>** je uzel, který představuje typ fyzického zařízení (hardware).
- **<<execution environment>>** reprezentuje typ prostředí zpracování softwaru (např. webový server).



Návrhové vzory

Návrhové vzory představují úspěšné řešení určitých problémů při vývoji SW, které se často opakují. Jsou to šablony pro řešení různých problémů. Každý vzor je popsán množinou komunikujících tříd, které jsou přizpůsobeny řešení obecných problémů v daném kontextu.

Pokud pro určitý problém, který se opakuje, používám úspěšně řešení, pak zobecnění toho řešení se stává návrhovým vzorem.

Klasifikace návrhových vzorů:

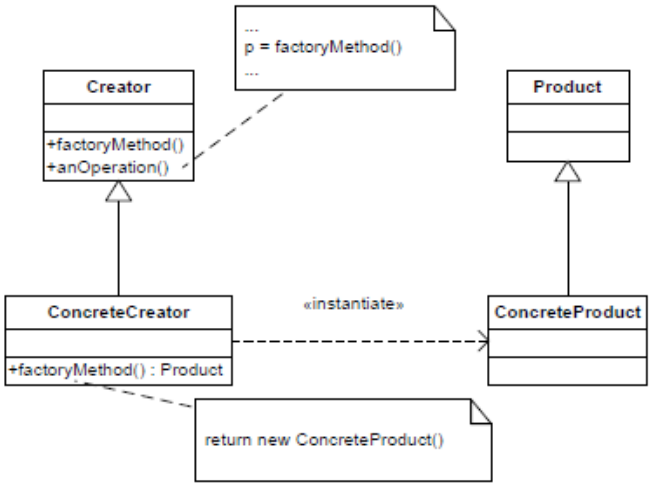
- **Návrhové vzory tvořící** – k řešení problémů vytváření instancí tříd cestou delegace této funkce na speciálně k tomu určené třídy
- **Návrhové vzory strukturální** – řešící problémy způsobu strukturování objektů a jejich tříd. Snaží se zpřehlednit architekturu SW
- **Návrhové vzory chování** – řeší chování systému – spolupráci mezi objekty.

Návrhové vzory tvořící

Návrhový vzor Tovární metoda

Účelem Tovární metody je definovat rozhraní pro vytvoření objektu, přičemž o kterou instancující třídu se jedná, rozhoduje odpovídající podtřída.

Je tvořen třídou **Product**, která definuje rozhraní objektů vytvářených tovární metodou, a která je konkretizována do třídy **ConcreteProduct** implementující požadované rozhraní. Třída **Creator** deklaruje tovární metodu **factoryMethod**, která je implementována v její konkretizaci **ConcreteCreator** odpovídající za vytváření instancí třídy ConcreteProduct.

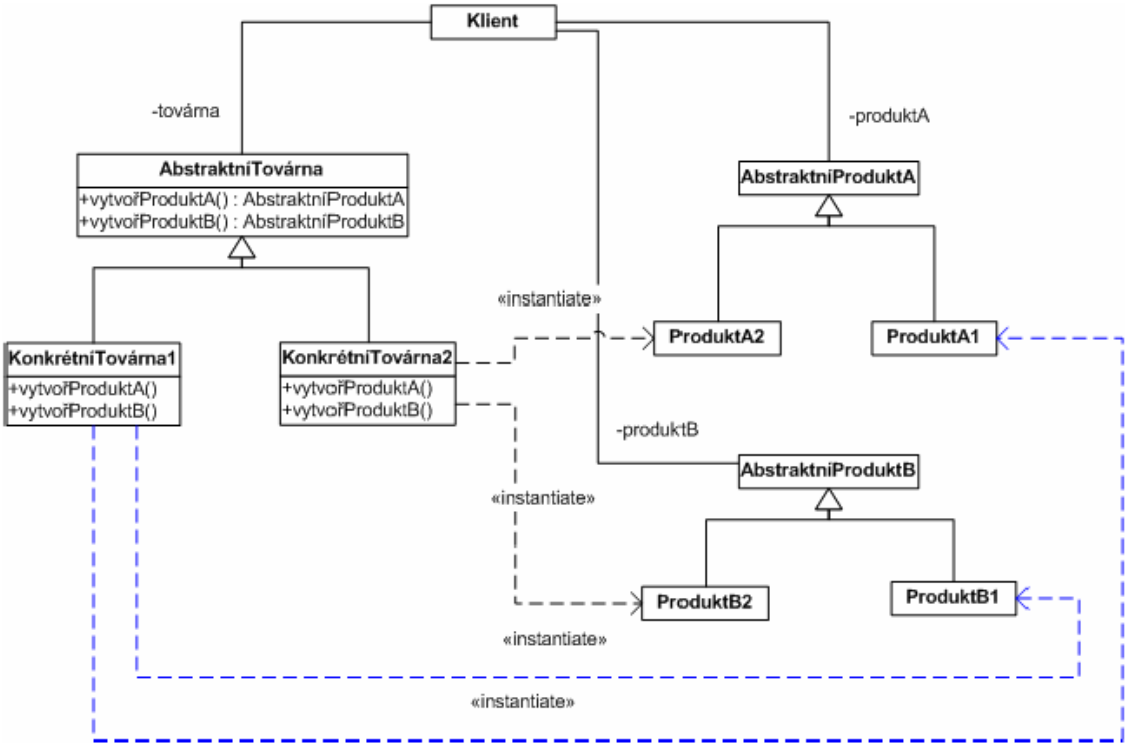


Obr. 4.2: Návrhový vzor Tovární metoda

Návrhový vzor Továrna

Poskytuje rozhraní pro vytváření rodin příbuzných objektů bez nutnosti specifikovat jejich konkrétní třídy. Nahrazuje přímé volaní konstruktoru voláním tvořící metody továrního objektu. Když se změní typ vytvářeného objektu, musíme v kódu nalézt všechna místa, kde se vytváří daný objekt. K eliminaci slouží tento návrhový vzor. Továrna je zobecněním návrhového vzoru Tovární metoda, protože se nevytváří jeden objekt, ale skupina příbuzných objektů.

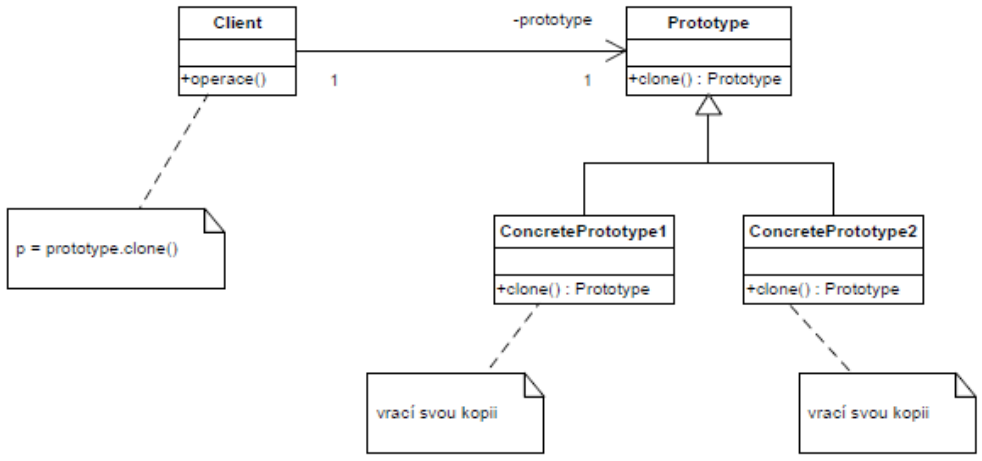
Továrna je tvořená třídou **AbstractFactory** deklarující rozhraní pro vytváření produktů s rozhraním deklarovaným ve třídě **AbstractProduct**. Třída **ConcreteFactory** pak implementuje odpovídající tovární metody pro vytváření konkrétních produktů.



Obr. 5.9.3: Návrhový vzor Továrna

Návrhový vzor Prototyp

Účelem Prototypu je poskytnout prototypové instance pro vytvoření specifikovaných objektů cestou vytvoření kopie tohoto prototypu. Používá se, když je vytváření instance třídy velmi časově náročné nebo nějak výrazně složité. Potom je výhodnější než náročně vytvářet mnoho instancí, vytvořit jednu a ostatní objekty z ní naklonovat. Prototyp je specifikován třídou **Prototype**, která deklaruje rozhraní pro vlastní klonování – operaci **clone**. Třída **ConcretePrototype** pak implementuje tuto operaci. **Client**, který požaduje vytvoření odpovídající instance třídy **ConcretePrototype**, udržuje asociaci na odpovídající prototyp.

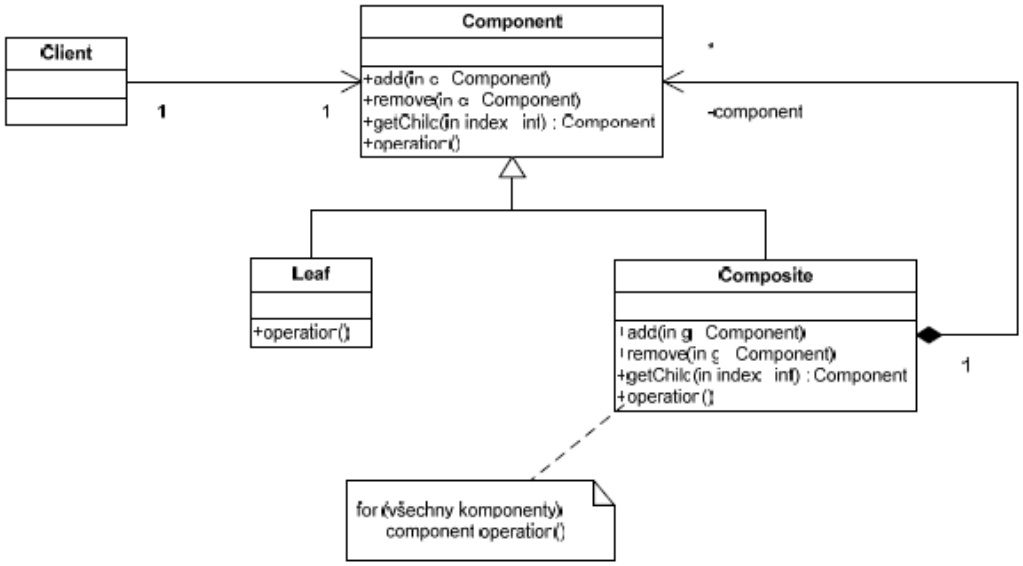


Obr. 4.6: Návrhový vzor Prototyp

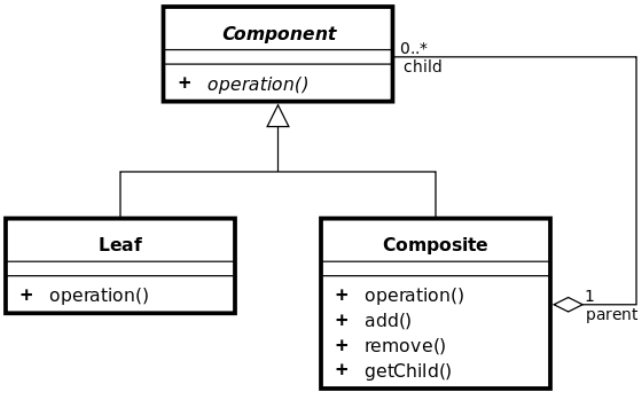
Návrhové vzory strukturální

Návrhový vzor kompozit

Účelem kompozitu je skládat objekty do stromové struktury reprezentující hierarchie typu celek a jeho části. Kompozit umožňuje manipulovat s celkem stejně jako s jeho části. Kompozit je specifikován pomocí třídy **Component**, která deklaruje rozhraní pro manipulaci s objekty v kompozici (add, remove, getChild) a požadované operace pro všechny třídy. Třída **Leaf** vyjadřuje primitivní elementy (listy) ve stromu s definovaným chováním. Třída **Composite** implementuje rozhraní pro manipulaci s objekty v kompozici a požadovanou operaci realizuje průchodem všemi obsaženými komponentami a zaslání zprávy na každou z nich.



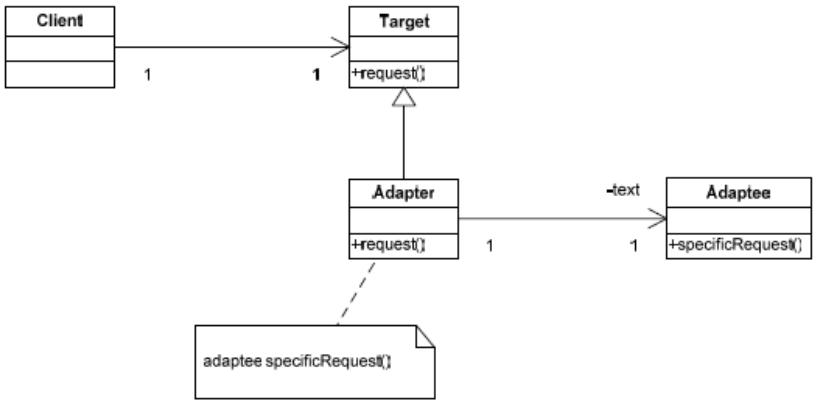
Obr. 4.8: Návrhový vzor Kompozit



Příklady použití: uživatelské rozhraní – vnější okno se skládá z několika vnořených prvků.

Adaptér

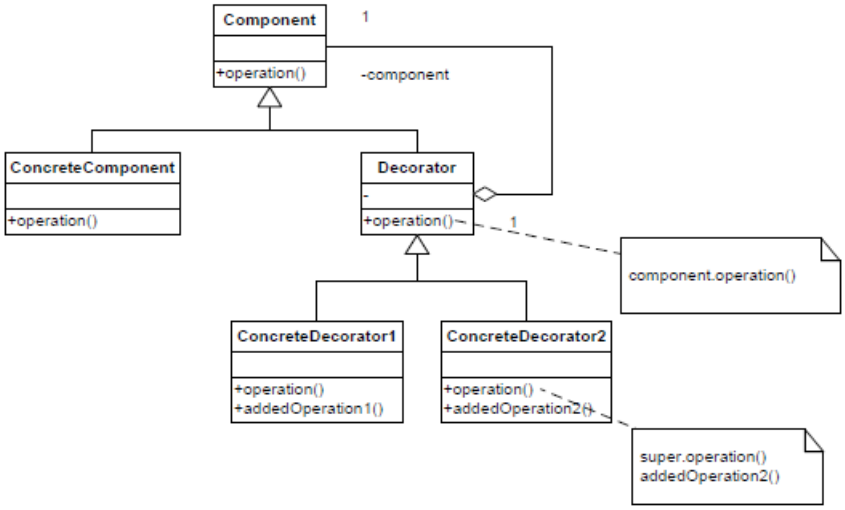
Účelem návrhového vzoru adaptér je převést rozhraní objektu na jiné rozhraní požadované jeho klientem. Umožňuje komunikaci mezi objekty, které by jinak díky rozdílnosti svých rozhraní této komunikace nebyly schopny. Návrhový vzor Adapter (nebo také Wrapper) se používá při práci s komponentou, která má nestabilní nebo s naší aplikací nekompatibilní rozhraní. Umožňuje komponentu obalit naším rozhraním a tak aplikaci úplně odstínit od rozhraní původního. Součástí vzoru Adapter je třída **Target** definující rozhraní specifické pro využití třídou **Client**. Třída **Adaptee** definuje funkcionalitu a rozhraní, které je třeba přizpůsobit. Třída **Adapter** pak přizpůsobuje rozhraní třídy **Adaptee** na cílové rozhraní definované třídou **Target.**



Obr. 4.10: Návrhový vzor Adaptér

Dekoratér

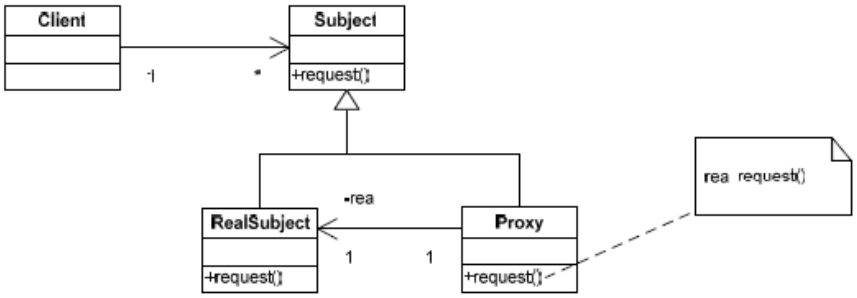
Účelem návrhového vzoru Dekorater je dynamicky přidávat k objektu další funkcionalitu. Dekorátor se vytváří za účelem změny instancí tříd bez nutnosti vytvoření nových odvozených tříd, jelikož pouze dynamicky připojuje další funkčnosti k objektu. Každá dodaná funkčnost je u dekorátoru implementována jako „ozdobení“ (dekorování) jiného objektu. Dekorátor tedy rozšiřuje objekt, ne třídu, jak je tomu u statické dědičnosti. Je specifikován třídou **Component** definující rozhraní pro objekty, ke kterým se budou dynamicky přidávat další služby. **ConcreteComponent** je objekt, jehož služby budou rozšiřovány. Třída **Decorator** realizuje vazbu na objekt **Component** a implementuje rozhraní definované touto třídou. **ConcreteDecorator** přidává další službu k dané komponentě.



Obr. 4.12: Návrhový vzor Dekoratér

Proxy

Poskytuje zástupce za jiný objekt a umožňuje tak řízený přístup k tomuto objektu. Jeho asi nejčastější použití je zapouzdření instance jiného objektu nebo přidání pomocné funkčnosti. Proxy nám tedy umožňuje řídit přístup ať už k celému či částečnému rozhraní objektu přes nějaký jiný zastupující objekt. Součástí je třída **Proxy** umožňující přístup instance třídy **Client** k objektu **RealSubject** výhradně přes své rozhraní. Třída **Subject** definuje společné rozhraní pro dva z uvedených objektů, tedy **RealSubject** i jeho zástupce **Proxy.**

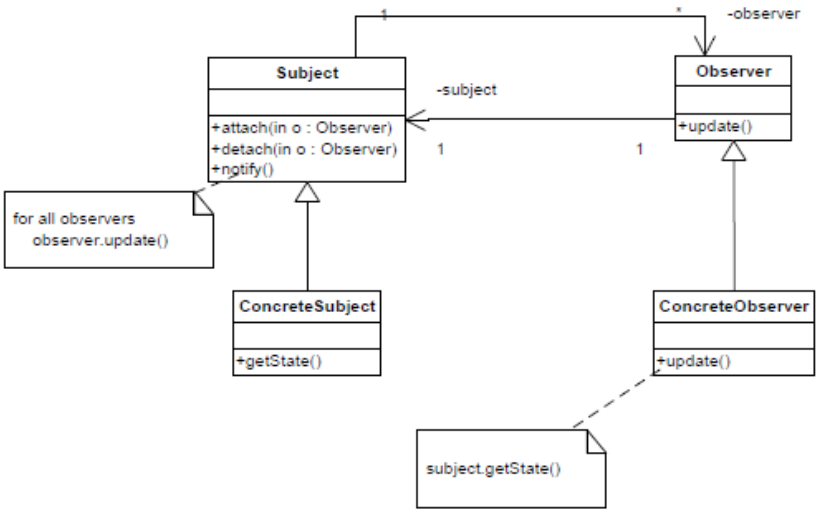


Obr. 4.14: Návrhový vzor Proxy

Návrhové vzory chování

Observer

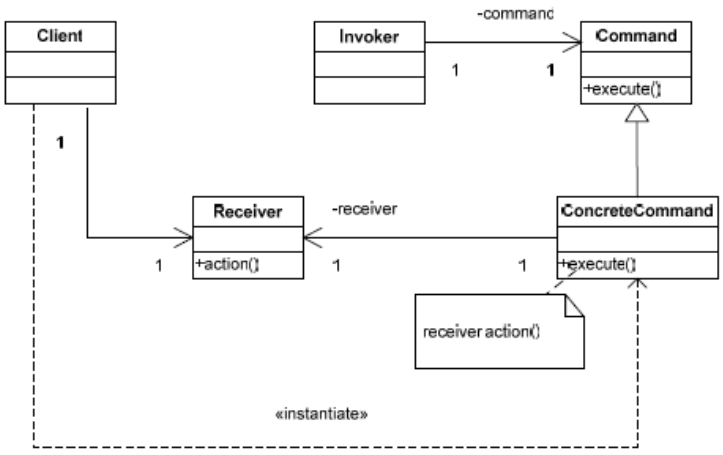
Účelem návrhového vzoru Pozorovatel je definovat závislost řady objektů na jediném takovým způsobem, že změna stavu tohoto objektu vede k automatické aktualizaci všech závislých objektů. Návrhový vzor Observer umožňuje objektu spravovat řadu pozorovatelů, kteří reagují na změnu jeho stavu voláním svých metod. Tvořen je třídou **Subject**, která definuje mechanismus udržování seznamu zaregistrovaných pozorovatelů a pomocí implementované operace **notify** na každý z nich odesílá zprávu **update**. Třída **Subject** je supertřídou pro všechny konkrétní předměty pozorování (**ConcreteSubject**), jejichž jediným úkolem je v případě změny stavu poslat na sebe sama zprávu notify. Třída **Observer** deklaruje operaci update, která je konkrétními pozorovateli **ConcreteObserver** implementována tak, aby zajistila zpracování změny stavu u pozorovaného objektu.



Obr. 4.22: Návrhový vzor Pozorovatel

Příkaz

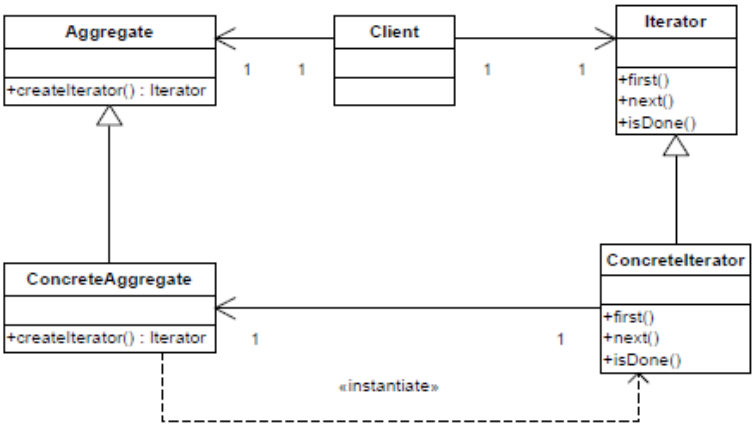
Účelem je zapouzdření požadavky do objektů a tyto ukládat a předávat dál jako jiné objekty. Návrhový vzor Command se využívá v situacích, kdy je třeba vykonání nějakého příkazu zapouzdřit do třídy a tu jako proměnnou předávat v aplikaci, například z důvodu jejího zařazení do fronty k pozdějšímu vykonání nebo rekonstrukci historie příkazů. Důležité je, aby zapouzdřený příkaz obsahoval všechny potřebné informace nutné ke svému vykonání. Vzor je specifikován třídou **Command** deklarující rozhraní pro vykonání příkazu. **ConcreteCommand** je podtřídou, která definuje vazbu na příjemce příkazu (třída **Reciever**) a implementuje operaci **execute** vrstvou volání odpovídajících operací asociovaného příjemce. **Reciever** tak implementuje tyto operace, zatímco **Invoker** dává pokyn k provedení příkazu.



Obr. 4.18: Návrhový vzor Příkaz

Iterátor

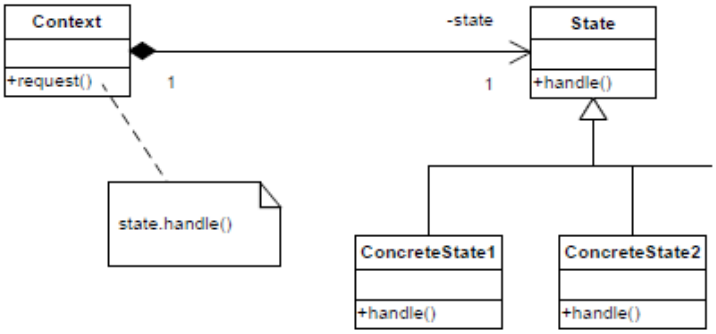
Poskytuje rozhraní pro přístup k elementům agregujících objektů sekvenčním způsobem bez nutnosti znát jejich vnitřní implementaci. Iterator zajišťuje možnost procházení prvků bez znalosti jejich implementace. Třída **Iterator** deklaruje rozhraní pro přístup k prvkům kolekcí sekvenčním způsobem. **ConcreteIterator** implementuje převzaté rozhraní pro konkrétní typ kolekce se kterou je asociován. Třída **Aggregate** je supertřída všech konkrétních kolekcí (**ConcreteAggregate**). **Aggregate** deklaruje operace **createIterator**, která je pak předefinována pro konkrétní kolekce tak, že k nim vytváří jim odpovídající iterátory.



Obr. 4.20: Návrhový vzor Iterátor

Stav

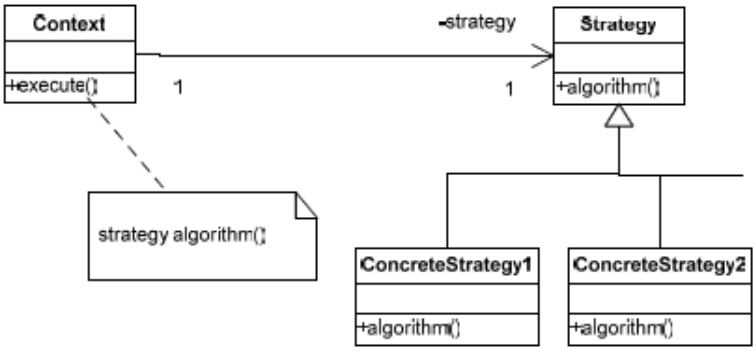
Umožnuje objektu změnit své chování, jakmile se změní jeho vnitřní stav. Tímto se objekt projevuje jako objekt jiné třídy. Návrhový vzor Stav je vhodným řešením v případě, kdy máme objekt, jež během své existence mění své vnitřní chování tak, že nabývá různých stavů. Přičemž chování objektu v různých stavech se výrazně liší. Při změně vnitřního stavu objektu se pak objekt reprezentující původní stav zamění za objekt jiný, jež odpovídá stavu novému. Třída **Context** definuje rozhraní vůči svým klientům, a která modeluje svůj aktuální stav pomocí vazby na konkrétní instanci podtřídy **State**. Konkrétní stav je pak definován v třídách **ConcreteState**, které dědí operaci **handle** popisující konkrétní algoritmus zpracování požadavku **request** podle aktuálního stavu objektu **Context**.



Obr. 4.24: Návrhový vzor Stav

Strategie

Definuje rodinu algoritmů zapouzdřených do objektů a zajišťuje tak jejich zaměnitelnost. Umožňuje zaměňovat algoritmy nezávisle na objektu, který je využívá. V aplikacích často používáme naše algoritmy, může se jednat o způsob výpočtu slevy v internetovém obchodě, způsob výpočtu daně, způsob uložení dat, způsob validace dat a podobně. Algoritmů pro stejnou věc může být více a to z důvodu kompatibility se staršími daty nebo jednoduše proto, že algoritmus závisí na konkrétní situaci. Pokud bychom chtěli za běhu vyměnit algoritmus, budeme muset bez znalosti Strategy nebo podobného vzoru implementovat switch či podobné větvení. To může být s narůstajícím počtem algoritmů nepřehledné a nový algoritmus vyžaduje zásah do zdrojového kódu. Je tvořen třídou **Strategy** deklarující rozhraní pro použití konkrétních algoritmů využívaných třídou **Context**. **ConcreteStrategy** implementuje tyto algoritmy. Třída **Context** má pak jediný úkol a to udržovat vazbu na aktuálně vybraný algoritmus.



Obr. 4.26: Návrhový vzor Strategie

Objektově orientované paradigma

Objektově orientované paradigma je programovací styl a způsob, jak přistupovat k programování. OOP podněcuje k rozkladu problému na elementární prvky. Každá komponenta se stává samostatným a nezávislým objektem, který obsahuje své vlastní instrukce a data, vztahující se k tomuto objektu.

- Při řešení úlohy vytváříme model popisované reality – popisujeme entity a interakce mezi entitami
- Abstrahujeme od nepodstatných detailů – při popisu entity vynecháváme nepodstatné vlastnosti entit
- Postup řešení je v řade případů efektivnější než při procedurálním přístupu (tam se úlohy řeší jako posloupnost příkazů)

Cíle OOP

- Snaha o znovupoužitelnost komponent
- Rozkládá složitou úlohu na dílčí části, které jdou pokud možno řešit nezávisle
- Řešení v počítači odpovídá reálnému světu
- Skrytí detailů implementace před uživatelem

Vlastnosti OOP

1. **Zapouzdření**

Mechanismus, který sdružuje dohromady kód a data a zabezpečuje je před vnějšími zásahy či zneužitím. **Spojením kódu s daty vzniká objekt**. Zapouzdření zaručuje, že objekt nemůže přímo přistupovat k datům jiných objektů, což by mohlo vést k nekonzistenci. Každý objekt navenek zpřístupňuje jen své **rozhraní**, pomocí kterého se s objektem pracuje. Obvykle jsou přístupné pouze veřejně dostupné metody dané třídy (tzv. „getry“ a „setry“), které umožňují přístup k vnitřním hodnotám a které ostatní třídy využívají (volají), aby získaly či modifikovaly hodnoty v daném objektu.

Pro schování dat uvnitř objektu slouží klíčové slovo **private**. Privátní kód nebo data jsou známá a dostupná pouze pro jinou část stejného objektu, ne z jiných objektů.

2. **Polymorfismus**

Polymorfismus umožňuje, aby bylo jedno jméno použito pro dva nebo více souvisejících, ale technicky různých účelů. Například odkazovaný objekt se chová podle toto, jaké třídy je instancí. Několik objektů poskytuje stejné rozhraní, pracuje se s ním stejným způsobem, ale jejich konkrétní chování se liší podle implementace. Souvisí to s dědičností, kdy na místo, kde je očekávána instance nějaké třídy, můžeme dosadit i instanci libovolné její podtřídy.

Dalším příkladem polymorfismu může být funkce abs() v C++, která vrací absolutní hodnotu z čísel int, nebo float. A podle toho, jaký typ dáme do parametru metody při volání, podle toho se zavolá příslušná funkce. Volba konkrétní implementace je tak dána typem dat. Jedna metoda lze zavolat s různým počtem parametrů a s různými typy parametrů.

3. **Dědičnost**

Jeden objekt může získat vlastnosti jiného objektu pomocí dědičnosti. Objekt může zdědit sadu vlastností a metod a do ní může vložit své vlastnosti a metody nebo přepsat stávající metody. Ve většině případů je povolena jednoduchá dědičnost – třída může dědit jen od jedné třídy. V C++ je možná i vícenásobná dědičnost.

Třída, objekt, rozhraní

Třída – mechanismus pro vytváření objektů. Je to popis množiny objektů mající společnou strukturu, chování. Třída je abstraktní, není nositelem dat, je to jen předpis, ze kterého se vytváří reálné objekty.

Objekt – OOP definuje program jako soubor spolupracujících objektů s přesně stanoveným chováním a stavem. Objekty napodobují objekty z reálného světa. Objekty si pamatují svůj stav a navenek poskytují operace – metody. Objekt je instancí třídy – je to konkrétní entita reálného světa vytvořená z předpisu (třídy).

Rozhraní (interface) – předepisuje třída, která od ní bude odvozena, jaké metody musí implementovat. Odvozený objekt může implementovat i další metody, ale musí implementovat všechny metody uvedené v interface. Interface nedefinuje žádné proměnné, pouze konstanty a neobsahuje těla metod, pouze hlavičky. Implementace je pak na odvozené třídě. Každá třída může implementovat několik rozhraní současně.

Vlastnosti objektu vs. Třída

Objekt je identifikovatelná samostatná entita, která je instancí třídy a je dána svou:

- **Identitou** – jedinečností umožňující ji odlišit od ostatních
- **Chováním** – službami, které poskytuje jiným objektům

Objekt má také sekundární vlastnosti:

- **Atributy** – v čase se měnící datové hodnoty popisující objekt
- **Doba existence** – časový interval daný okamžikem vzniku a zániku objektu
- **Stavy** – odrážení různé fáze doby existence objektu

Třída je jen předloha toho, jak bude objekt dané třídy vypadat. Definuje, jak budou vypadat **operace (metody)** objektu a jaké **atributy** bude mít. Každý objekt si následně tyto atributy naplní svými daty. Třída nemá hodnoty, pouze definuje pojmenování těchto atributů a jejich typ.

Třída vs. Rozhraní

Rozhraní určuje metody, které mají být implementovány ve třídě, která toto rozhraní implementuje. Třída musí definovat všechny metody uvedené v interface a v interface mohou být pouze hlavičky metod.

Abstraktní třída

Existuje i hybrid mezi klasickou třídou a interface. Je to obyčejná třída, která nemá definované některé metody, obsahuje pouze hlavičku těchto metod. Předpokládá se, že nějaká jiná třída bude z abstraktní třídy dědit a doplní chybějící definice. Na rozdíl od interface má schopnost definice metod a může obsahovat i atributy.

Třídní vs. Instanční vlastnosti

Vlastnosti se mohou dělit na statické a instanční. **Statické vlastnosti** se mohou používat i bez vytvoření instance dané třídy, jsou vázané jen na třídu samotou. Vhodné, kdy potřebujeme volat nějakou podpůrnou funkci dané třídy nebo nějaké konstanty související s třídou. Přístup je pomocí názvu třídy a tečky. Hodnoty statických atributů musí být definovány přímo ve třídě a existují jen jednou.

Instanční proměnné udávají vlastnosti konkrétního objektu třídy. Vznikají s objektem a mají pro každý objekt své vlastní hodnoty. Bez vytvořeného objektu se nedají volat.

Mapování UML diagramů na zdrojový kód

Cílem implementační fáze vývoje softwaru je doplnit navrženou architekturu aplikace o programový kód. Implementační model specifikuje, jak jsou jednotlivé elementy (objekty a třídy) vytvořené. Ve fázi návrhu jsme pracovali pouze s abstrakcemi, které byly dokumentovány pomocí různých diagramů. V průběhu implementace dochází k jejich **fyzické realizaci.**

Implementační model se zaměřuje také na specifikaci toho, jak budou jednotlivé komponenty fyzicky organizovány. K tomu slouží v UML:

- **Diagram komponent** – ilustrující organizaci a závislosti mezi SW komponentami
- **Diagram nasazení** – poskytuje upřesnění o konfiguraci technických prostředků a znázorňuje umístění implementovaných SW komponent na různých prostředcích – na server atd.

Mapování elementů logického modelu na komponenty

Pokud je specifikace objektů a jejich tříd v diagramu tříd důsledná a úplná, je možné z tohoto diagramu tříd automaticky vygenerovat zdrojové kódy dle následující tabulky:

Analýza a návrh (UML)	**Zdrojový kód (Java)**
Třída	Struktura typu *class*
Role, Typ a Rozhraní	Struktura typu *interface*
Operace	Metoda
Atribut třídy	Statická proměnná označená *static*
Atribut	Instanční proměnná
Asociace	Instanční proměnná
Závislost	Lokální proměnná, argument nebo návratová hodnota zprávy
Interakce mezi objekty	Volání metod
Případ užití	Sekvence volání metod
Balíček, Subsystém	Kód nacházející se v daném adresáři specifikovaným ve zdrojovém souboru pomocí *package*

Pak se jen do vygenerovaného kódu doplní těla metod a vytvoří se fyzická struktura vytvářených souboru reprezentující SW komponenty.

Komponenta může představovat soubor napsaný ve zdrojovém kódu, zkompilovanou komponentu v binárním kódu nebo další komponenty reprezentované databázovými tabulkami, dokumenty atd.

Postup mapování:

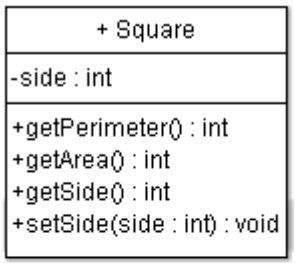
1. První krok bude rozhodnout o fyzické organizaci zdrojových souborů a to podle **diagramu komponent**.
2. Z třídního diagramu se vytvoří zdrojové kódy (pomocí nástrojů je lze i automaticky vygenerovat) do souborů určených v předešlé části
3. Poté je nutné doplnit těla metod – tyto lze specifikovat pomocí **sekvenčních diagramů** a ze **stavových diagramů**, které detailně popisují chování objektů.
4. Umístění spustitelných komponent na jednotlivé technické prostředky pomocí **diagramu nasazení**

Implementace tříd

- Pokud nějaká třída využívá jinou třídu, je třeba tuto třídu **importovat**

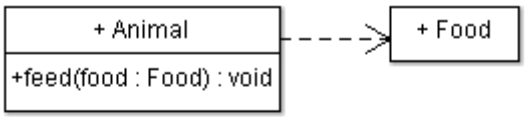
- **Asociace** na jiné třídy jsou převedeny na instanční proměnné daných tříd
- **Diagram komponent** také obsahuje binární komponenty, které jsou nutné pro implementaci (může se jednat o virtuální stroj Javy nebo nějaké knihovny)

Mapování diagramu tříd na kód




```
public class Square {
  private int side;
  public int getSide() {return side;}
  public void setSide(int side) {this.side = side;}
  public int getPerimeter() {return 4*side;}
  public int getArea() {return side*side;}
}
```

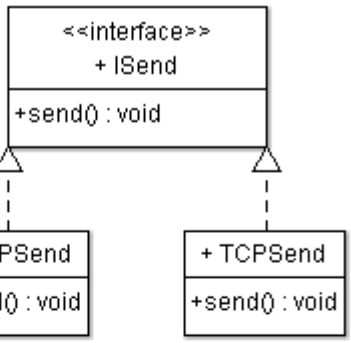
- Přepisujeme název třídy a její přístupnost
- Názvy atributů a jejich datové typy + přístupnost (- privátní; + vveřejný; # protected)
- Hlavičky metod, jejich přístupnost, návratová hodnota a parametry s jejich datovými typy




```
public class Food {
  // ...
}

public class Animal {
  void Feed(Food food) {
    // ...
  }
}
```

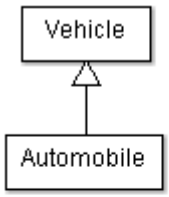
- **Závislost –** Animal používá ve své metodě Feed třídu Food. Jedná se nejslabší vztah indikující závislost jedné třídy na druhé.




```java
public interface ISend {
  public void send();
}

public class UDPSend implements ISend {
  @Override
  public void send() {
    // ...
  }
}

public class TCPSend implements ISend {
  @Override
  public void send() {
    // ...
  }
}
```

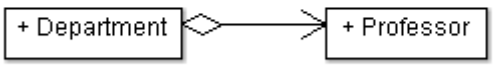



```java
public class Vehicle {
  // ...
}

public class Automobile extends Vehicle {
  // ...
}
```

- **Asociace** – vztah mezi instancemi. Často popisuje schopnost jedné instance poslat zprávu druhé instanci typicky prostřednictvím referenční proměnné.
- **Agregace, Kompozice**

Agregace




```java
public class Professor {
  // ...
}

public class Department {
  private List<Professor> professorList;
  // ..

}
```

Správa paměti

Většina současných aplikací ke své činnosti využívá dynamického přidělování paměti, které umožňuje efektivně využívat paměť i v případě předem neznámých požadavků na její velikost nutnou pro konkrétní běh aplikace.

Typy přidělování paměti

- **Statické přidělování paměti** – již při překladu lze určit, kde v paměti budou data umístěna a jakou velikost budou mít
- **Dynamické přidělování paměti** – paměť je vyhrazena až při běhu programu

Správa paměti je rozdělena do tří úrovní:

- Na úrovni **technického vybavení** – zabývá se elektronickými prvky, v nichž jsou data skutečně uložena (RAM, Cache)
- Na úrovni **operačního systému** – paměť se přiděluje uživatelským programům; OS může mít více paměti, než je ve skutečnosti k dispozici – virtuální paměť
- Na úrovni **aplikačních programů** – přidělování úseků omezené dostupné paměti pro objekty a datové struktury jednoho programu. Řeší dva hlavní úkoly
    - **Přidělování paměti** – přidělí úsek paměti odpovídající délky z většího bloku paměti získané od OS
    - **Regenerace paměti** – Není-li úsek dále využíván, může být uvolněn a dán pro opakované použití. Dělí se to na *manuální správu paměti* a *automatická správa paměti*

Problémy správy paměti

- **Předčasné uvolnění paměti** – uvolní se paměť, avšak pokouší se k ní program přistupovat později, nastává při manuální správě paměti
- **Únik paměti** – program neustále přiděluje novou paměť, aniž by ji zase uvolňoval
- **Externí fragmentace** – rozdělení na mnoho malých bloků a při požadavku na větší blok sice je místo, ale tak velký blok se nedá nalézt

Přístupy ke správě paměti

- **Manuální správa paměti**
    - programátor má plnou kontrolu nad tím, zda a ve kterém okamžiku bude paměť uvolněna
    - Vede k výše uvedeným chybám, moderní programovací jazyky se orientují na automatickou správu paměti
    - Funkce pro přidělování a uvolňování paměti z **hromady**– malloc/free v jazyce C
    - Paměť ze **zásobníku** se alokuje sama, použití pro lokální proměnné
- **Automatická správa paměti**
    - Služba, která je součásti jazyka, automaticky regeneruje paměť, kterou by program již znovu nevyužil
    - Nazývá se **Garbage Collector** – opakovaně používá bloky paměti, které již jsou nedosažitelné z jisté sady programových proměnných – na které se není možné dostat pomocí ukazatelů
    - Metoda vynulování odkazu na objekt (obj = null) – je-li tento odkaz jediným odkazem na objekt, stane se objekt nedostupných a při nejbližší příležitosti se paměť uvolní
    - **Výhody**

        - Programátor se může věnovat řešení skutečného problému
        - Nastává menší množství chyb spojených s přístupem do paměti
        - Správa paměti je často mnohem efektivnější
    - **Nevýhody**
        - Paměť může být zachována jen proto, že je dostupná, i když není dále využita

Metody přidělování paměti

Volná paměť je tvořena obvykle seznamem souvislých paměťových bloků, jejichž adresu a délku známe. Úkolem přidělování paměti je pro zadanou velikost požadované paměti vyhledat vhodný úsek volné paměti, tento úsek označit za obsazený a vrátit jeho počáteční adresu.

- **Předělování ze zásobníku** – přidělování z jediného souvislého bloku paměti, organizovaného jako zásobník
- **Výběr prvního vhodného bloku** - Máme-li volné bloky paměti různé délky seřazené do seznamu, je třeba při požadavku na přidělení paměti určité velikosti vyhledat v seznamu dostatečně velký blok. Metoda výběru prvního vhodného bloku jednoduše prochází seznam volných bloků a vybere z něj první blok, jehož velikost je větší nebo rovna požadované velikosti. Je-li blok delší, rozdělí se a zbývající část je vložena zpět do seznamu.
- **Výběr nejlepšího vhodného bloku** - vyhledání nejmenšího volného bloku paměti, jehož velikost je větší nebo rovna požadované. To vede k minimalizaci ztrát zajištěním menší fragmentace paměti.
- **Přidělování s omezenou velikostí bloku** - Tyto metody přidělování paměti jsou založeny na hierarchickém dělení volného paměťového prostoru na části. V nejjednodušším případě je paměť rozdělena na dvě velké části, ty se dále dělí na menší části. Binární: Bloky mají velikost 2^n; Fibonaciho dělení: bloky velikosti Fib(n). Spojovat lze pouze sousední bloky.

Metody regenerace paměti

Regenerace paměti se používá v případech, kdy chceme již nepoužívané bloky paměti dát k dispozici pro další přidělování. V zásadě existují dvě skupiny metod - metody založené na **sledování odkazů** a metody založené na **čítačích odkazů**.

- **Dvoufázové značkování** – Jme schopni najít všechny kořenové ukazatele a víme, kde jsou v datech umístěny další ukazatele. Rekurzivním průchodem jsme pak schopni označit postupně všechny bloky paměti, dostupné z kořenových ukazatelů, a zbývající neoznačené bloky uvolnit. Vyžaduje znalost struktury objektů za běhu aplikace.
- **Regenerace kopírováním** - Všechny bloky jsou zkopírovány do jedné souvislé oblasti paměti. Následně jsou aktualizovány odkazy na kopírované objekty.
- **Inkrementální regenerace** - Inkrementální regenerace paměti probíhá po částech a střídá se s prováděním samotné aplikace.
- **Generace s počítáním odkazů** - Pro každý přidělený blok paměti můžeme udržovat informaci o počtu odkazů, které na tento blok ukazují. Paměť pak může být uvolněna v případě, že se čítač odkazů sníží na nulu a na blok paměti tedy neukazuje žádný ukazatel.

C

- Řeší přidělování paměti pomocí standardních knihovních funkcí – **malloc()** pro přidělení prostoru určité velikosti a **free()** pro uvolnění přidělené paměti. Vrácený ukazatel funkce malloc() musí být přetypován na požadovaný typ.
- Implementace je obvykle seznamem volných bloků

C++

- Přidělování paměti je již součástí syntaxe jazyka. Pro přidělení paměti je operátor **new** a pro uvolnění operátor **delete**.

C#

- Správa paměti je v C# plně automatizovaná, paměťový prostor se přiděluje operátorem **new**, jeho uvolnění zajistí systém řízení běhu programu.
- Používá algoritmus next fit (vyhledávání začíná na pozici, kde předchozí vyhledávání skončilo z zabrání hromadění menších bloků na začátku seznamu), regenerace s kopírováním, využívá generací
- Udržuje pointer na další volné místo NextObjPointer
- Při regeneraci paměti jsou objekty na hromadě seřízeny dle jejich vzdálenosti od kořenů
- Udržuje 3 generace: vytvořené objekty, objekty, které přežily jeden průchod GC, objekty, které přežily více průchodů GC
- Inkrementální regenerace – dva vlákna běžící na pozadí;používá metodu dvoufázového značkování a identifikuje „garbage“; volá finalize a uvolňuje paměť
- Lze explicitně uvolnit paměť pomocí Systém.GC.Collect

Java

- Správa je rovněž plně automatizovaná
- O uvolňování paměti se stará separátní vlákno, které běží s nízkou prioritou a zajišťuje kontinuální sledování nepoužitých bloků paměti. Před uvolněním paměti se volá metoda **finalize()**
- Přidělování paměti se provádí operátorem **new**
- Používá inkrementální regeneraci, pro vyhledávání referencí se využívají metadata.
- GC lze přímo volat Systém.gc()

Python

- Správa paměti je automatizovaná
- Proměnná je v pythonu řešená jako odkaz do paměti, při změně hodnoty se změní odkaz na jinou část paměti.
- Používá počítaní referencí, počet referencí lze zjistit pomocí sys.getrefcount(objekt)
- Python si některé často používané hodnoty automaticky udržuje v paměti – pamatuje si je a nevytváří je znovu (kvůli výkonu) – např. čísla - 5 až 256, jednotlivé znaky

Virtuální stroj

Virtuální stroj je software, který vytváří virtualizované prostředí mezi platformou počítače a operačním systémem, ve kterém koncový uživatel může provozovat software na abstraktním stroji.

Může to být **hardwarový virtuální stroj.** Na jednom počítači může běžet více virtuálních strojů s různými OS. Příklad: virtuální servery.

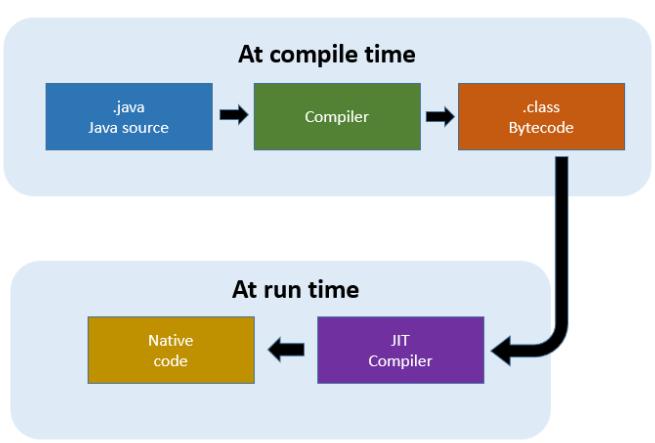
Aplikační virtuální stroj

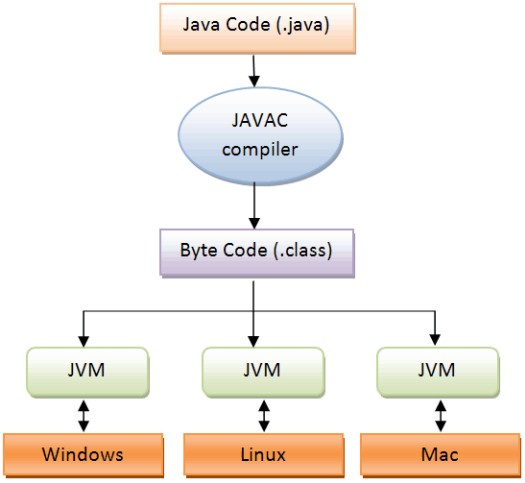
Počítačový software, který izoluje aplikace používané uživatelem na počítači. Virtuální stroj je psán pro různé platformy a tak jakákoliv aplikace psaná pro virtuální stroj může běžet na jakékoliv platformě místo toho, aby se musely vytvářet různé verze pro různé platformy – vytvoří se jen jedna pro daný virtuální stroj.

Příklad aplikace virtuálního stroje – **Java Virtual Machine.**

JVM je sada počítačových programů a datových struktur, který využívá virtuální stroj pro spouštění programů vytvořených v Javě. Tento virtuální stroj zpracovává pouze mezikód (bytekod). Ten je vytvořen ze zdrojových kódu Javy. JVM je k dispozici pro mnoho platforem a proto je možné aplikace v Javě vytvořit pouze jednou a spustit na kterémkoliv z platforem.

JVM je **interpret** umožňující vykonávat zdrojový kód programu přeložený do mezikódu (.class, .jar). Tento kód musí být před spuštěním aplikace přeložen do strojového kódu pomocí tzv. **just-in-time kompilace.**





Podpora paralelního zpracování

Paralelní programování je paradigma, které umožňuje naprogramovat úlohy tak, aby byly schopny paralelního běhu. Standardní struktura počítačového softwaru je založena na sekvenčním výpočtu. Při řešení problému je algoritmus určený k řešení tohoto problému realizován jako série za sebou následujících instrukcí. Tyto instrukce jsou prováděny pomocí CPU jednoho počítače. Z toho plyne, že současně může být vykonávána pouze jedna instrukce. Teprve po vykonání této instrukce následuje vykonávání další instrukce. Paralelně programovaný software využívá možnost rozdělení jednoho velkého výpočetního problému na několik menších problémů, které jsou řešeny „současně“.

Pro zrychlení výpočtu se používá buď **zdánlivý** paralelní běh úloh pomocí multitaskingu nebo víceprocesorové systémy nebo počítačové clustery – **skutečný** paralelismus. Operační systémy s preemptivním multitaskingem vytvářejí dojem souběžného provádění více vláken ve více procesech. To je zajištěno rozdělením času procesoru mezi jednotlivá vlákna po malých časových intervalech. Pokud časový interval vyprší, je běžící vlákno pozastaveno, uloží se jeho kontext a obnoví se kontext dalšího vlákna ve frontě, jemuž je pak předáno řízení. V případě, že máme k dispozici více procesorů, jsou mezi ně vlákna přidělována ke zpracování a k současnému běhu pak skutečně dochází.

Využití:

- Servery, které obsluhují více požadavků zároveň (např. WWW server)
- GUI – pro náročné operace se provede samostatné vlákno a GUI tak nezamrzne, ale dá se s programem dál pracovat
- Výpočetně složité algoritmy nad velkými daty, které vyžadují velké množství času
- Rozlišení mezi úlohami s různou prioritou.

Software

- Programy využívající sdílenou paměť komunikují pomocí manipulace s proměnnými ve sdílené paměti
- Programy využívající distribuovanou paměť používají metodu zasílání zpráv

Synchronizace

Komunikace mezi vlákny v rámci jednoho procesu je jednodušší než komunikace mezi různými procesy, a to právě díky sdílené paměti, pomocí které mohou vlákna komunikovat. Na druhé straně je třeba zajistit, aby vlákna k této sdílené paměti přistupovala synchronizovaně, aby nedocházelo např. k přepisu jedné informace několika vlákny současně.

Pokud nezajistíme synchronizovaný přístup ke sdíleným zdrojům (v rámci téže aplikace nebo i mezi více aplikacemi současně), může to vést k situacím jako je uváznutí nebo časový souběh. Při **uváznutí** (deadlock) přestanou dvě vlákna reagovat, neboť na sebe vzájemně čekají. **Časový souběh** nastává tehdy, pokud může k nějaké zvláštní situaci dojít v závislosti na kritickém načasování dvou událostí. Abychom se tomuto problému vyhnuli, musíme zajistit, aby v době, kdy jedno vlákno čte nebo zapisuje sdílená data, k nim nemohlo přistupovat žádné jiné vlákno.

TODO Flynnova klasifikace paralelních systémů

Vlákna

Zatímco běžné procesy jsou navzájem striktně odděleny, sdílí vlákna jednoho procesu nejen společný paměťový prostor, ale i další datové struktury. Operační systémy používají pro oddělení různých běžících aplikací **procesy**. Proces je tvořen paměťovým prostorem a jedním nebo více **vlákny**. Tento paměťový prostor jednotlivá vlákna sdílejí. Vlákno je samostatně plánovatelný tok řízení programu. Představuje tedy jistou posloupnost instrukcí, jejíž provádění může být přerušeno např. po vypršení určitého časového limitu nebo čekáním na nějakou událost. Po ukončení důvodu přerušení může vlákno dále pokračovat v činnosti.

Vlákna v Javě

Každé vlákno v Javě je instancí třídy java.lang.Thread. Tato třída zajišťuje spuštění, zastavení a ukončení vlákna. Vlákno musí implementovat metodu run, která definuje činnost vlákna. Této metodě je předáno řízení po spuštění vlákna metodou start.

Vlákno v průběhu svého života prochází posloupností následujících stavů:

- **New** - bezprostředně po vytvoření ještě nejsou vláknu přiděleny žádné systémové prostředky, vlákno neběží.
- **Runnable** - po provedené metody start je vlákno připraveno k běhu. V tomto stavu se může nacházet více vláken, ovšem jen jedno z nich (na počítači s jedním procesorem) je ve stavu „běžící“.
- **Not runnable** - do tohoto stavu se vlákno dostane, je-li pozastaveno voláním jedné s metod sleep, wait nebo suspend, případně čekáním na dokončení operace vstupu/výstupu.
- **Dead** - do tohoto stavu se vlákno dostane ukončením metody run nebo voláním metody stop.

Sync vláken v Jave

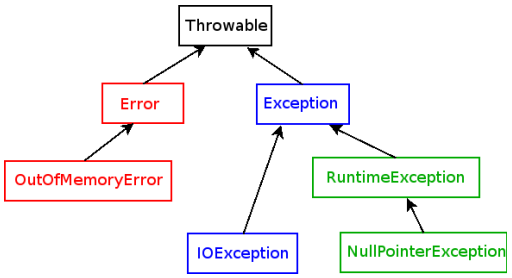
Sekce kódu, které nesmějí být vykonávány paralelně, se nazývají **kritické sekce**. Tyto kritické sekce například modifikují stejná data. Kritická sekce je realizována příkazem **synhronized (odkaz na objekt) {}.**

Zpracování chyb v moderních programovacích jazycích

Výjimky představují určité situace, ve kterých musí být výpočet přerušen a řízení předáno na místo, kde bude provedeno ošetření výjimky a rozhodnutí o dalším pokračování výpočtu. Starší programovací jazyky pro ošetření žádnou ošetření výjimek obvykle neměly. U všech funkcí, které mohly objevit chybu, bylo třeba stále testovat různé příznaky a speciální návratové hodnoty, kterými se chyba oznamovala, a v případě, že jsme na testování chyby zapomněli, program běžel dál a na chybu v nejlepším případě nereagoval nebo se zhroutil na zcela jiném místě, kde již bylo obtížné zdroj chyby dohledat. V lepším případě bylo možné nastavit adresu, na kterou program v případě výskytu chyby skočil a kde se pak provedlo její ošetření (např. příkaz ON GOTO v jazyce Basic nebo funkce signal() v jazyce C).

V moderních programovacích jazycích, jako je C++ nebo Java se pro ošetření výjimek používá metoda strukturované obsluhy. To znamená, že programátor může pro konkrétní úsek programu specifikovat, jakým způsobem se má konkrétní typ výjimky ošetřit. V případě, že výjimka nastane, vyhledá se vždy nejbližší nadřazený blok, ve kterém je výjimka ošetřena.

Výjimky jsou v jazyce Java reprezentovány jako objekty; tyto objekty jsou instancemi tříd odvozených obecně od třídy **Throwable** se dvěma podtřídami **Error** a **Exception**.



Každá z podtříd odpovídá jedné skupině výjimek. Třída **Error** představuje vážný problém v činnosti aplikace, tyto výjimky by se neměly zachycovat. Značí totiž kritickou chybu (nedostatek zdrojů pro práci virtuálního stroje, přetečení zásobníku atp.) – a ve většině případů je ani ošetřit nemůžeme.

Třída **Exception** zahrnuje výjimky, které má smysl ošetřovat – např. otevření neexistujícího souboru, chybný formát čísla atd. Do této skupiny by měly patřit také veškeré výjimky definované uživatelem.

Mezi výjimky dědící z Exception patří také podtřída RuntimeException. Runtime exception jsou výjimky, které sice nejsou kritické z hlediska samotné možnosti pokračování aplikace (na rozdíl od Error), ale přesto se velmi často neošetřují. Značí totiž obvykle chyby, které způsobil sám programátor (neplatný index pole, volání nad nullovým ukazatelem...).

Příkaz try-catch

Pro zpracování výjimky slouží blok try, který je následován jedním nebo více bloky catch, které popisují způsob ošetření jednotlivých výjimek, které nastanou v bloku try.

```
InputStream fs = null;
try {
   fs = new FileInputStream("data.txt");
} catch( FileNotFoundException e ) {
   System.err.println("Soubor data.txt nenalezen");
   System.exit(2);
}
```

Nastane-li v těle try výjimka typu FileNotFoundException, výpočet těla se ukončí a přejde se do bloku catch.

Blok catch jako parametr přijímá typ výjimky. V bloku catch může být místo konkrétní třídy výjimky rovněž uvedena některá nadtřída zahrnující celou skupinu výjimek. Například pro ošetření libovolné chyby v operacích vstupu a výstupu můžeme použít třídy IOException. Celý blok zpracování výjimky je ukončen sekcí „finally“ nebo „ensure“, která slouží pro uvolnění systémových zdrojů použitých při zpracování výjimky.

Vytvoření vlastní výjimky

Vlastní výjimka se vytvoří pomocí vytvoření třídy, která dědí od třídy Exception nebo některé její podtřídy. Vyhození výjimky je způsob signalizace, že probíhající část zpracovávaného kódu nemohla být provedena normálně.

```
class MojeVyjimka extends Exception {
   MojeVyjimka(String msg) { super(msg); }
}
```

V místě, kde chceme výjimku vyvolat, pak pouze vytvoříme instanci třídy MojeVyjimka a použijeme ji jako argument příkazu throw. Ten zajistí vyvolání výjimky a její případné ošetření.

Princip datových proudu (I/O)

Technika datových proudů nám umožňuje pracovat se soubory. I/O systém nejen jazyka C poskytuje programátorovi stále stejné rozhraní, bez ohledu na skutečně použité I/O rozhraní. Aby se toho dalo dosáhnout, zavádí se mezi programátorem a zařízením (hardwarem) určitá míra abstrakce, která se nazývá **datový proud.** Skutečné zařízení provádějící vstupně výstupní operace se nazývá **soubor**. Za soubor můžeme považovat diskový soubor, obrazovku, klávesnici, port, tiskárnu atd. právě z důvodu, že se všemi těmito I/O zařízeními pracujeme jednotně – jako se souborem. Výhodou tohoto přístupu je, že pro programátora vypadá jedno hardwarové zařízení stejně jako druhé. Datový proud automaticky ošetřuje rozdíly.

Datový proud je připojen k souboru pomocí **operace otevření (open)** a odpojen od souboru pomocí **operace uzavření** (close). **Aktuální pozice** nám určuje místo v souboru, kde se bude provádět další operace se souborem.

Typy proudů se dělí podle typu přenášených dat:

- **Binární data –** obrázky, zvuk atd.
- **Textová data –** textové soubory s ASCII nebo UNICODE znaky

Typ proudu si volíme při otevírání proudů zvolením vhodné třídy.

Rozdíl mezi znakově a bytově orientovanými datovými proudy

Příklad v C#: My pro práci se soubory budeme využívat především třídy System.IO.StreamReader a System.IO.StreamWriter. Tyto třídy implementují třídy System.IO.TextReader a System.IO.TextWriter pro práci s bytovými proudy v patřičném kódování.

K tomu, abychom s těmito třídami mohli pracovat, budeme potřebovat i nějaký proud, odkud (kam) budeme číst (zapisovat). Pro práci se soubory se používá proud System.IO.FileStream, pro paměť jako úložiště dat se používá System.IO.MemoryStream

Znakový datový proud

Rozlišuje jednotlivé znaky nebo řádky. Čtení probíhá sekvenčně. Při zápisu do textového souboru v C# si vytvoříme soubor pomocí třídy **FileStream** s módem Create, takže pokud soubor s daným názvem existuje, bude přepsán. Pro tento soubor pak vytvoříme **StreamWriter**, který slouží k ukládání hodnot, například pomocí metody **WriteLine,** která zapíše do textového souboru jeden řádek.

Zápis probíhá podobně. Vytvoříme si proud pomocí **FIleStream** a **StreamReader**, který poskytuje metodu ReadLine() pro čtení jednotlivých řádků.

Binární datový proud

Při práci s binárním souborem je nezbytný blokový přenos dat. Do binárního proudu zapisujeme v podobě binárního tvaru. U binárního souboru můžeme libovolně přistupovat ke zvolené část dat – náhodný přístup. U textového souboru čteme sekvenčně znak po znaku.

Do binárních souborů můžeme ukládat binární data – např. při čtení můžeme načíst celou hodnotu typu decimal apod.

Zápis do binárního souboru probíhá tak, že se vytvoří proud pomocí **FileStream**. Ten ale můžeme vytvořit i voláním metody System.IO.File.Create(). Tento proud pak dáme jako parametr konstruktoru **BinaryWriter**, který poskytuje metodu Write pro zápis hodnoty.

Čtení probíhá obdobně. Při čtení z binárního souboru musíme mít na paměti, který datový typ se právě má číst. Pokud jsme do našeho binárního souboru ukládali například informace o zaměstnancích včetně dat narození, platu apod., kde se vyskytuje více typů (string, decimal a jiné), musíme si na pořadí typů dát pozor. Znovu se vytvoří proud v podobě souboru, který se předá konstruktoru třídy **BinaryReader**, která má metodu např. ReadInt32(), která čte jednotlivé int čísla.

Je možné ukládat do binárního souboru i celou instanci třídy, nejen jednoduché typy jako v případě BinaryWriter. Tento proces se nazývá **serializace**.

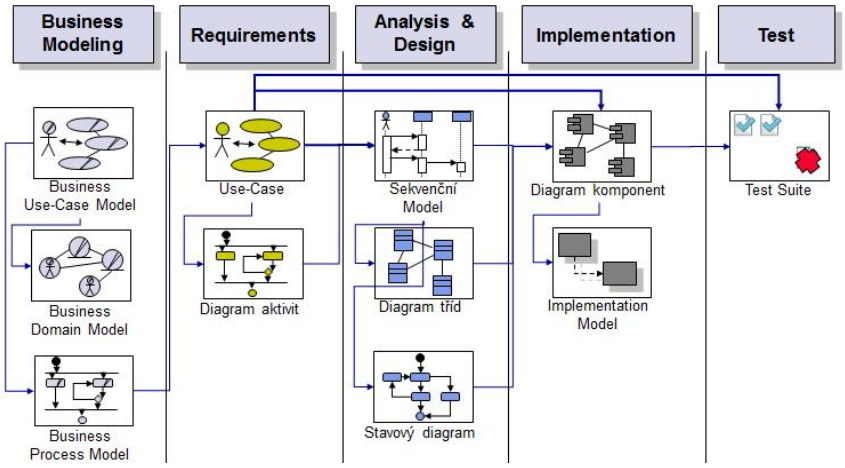
Jazyk UML – typy diagramů a jejich využití v rámci vývoje

UML (Unified Modeling Language) je unifikovaný modelovací grafický jazyk pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů. Pomocí něj můžeme realizovat nejrůznější schémata napříč procesem vývoje softwaru. V průběhu let se UML stal standardizovaným jazykem určeným pro vytvoření výkresové dokumentace systému. UML je jazyk pro **specifikaci, visualizaci, konstrukci a dokumentaci** artefaktů SW systémů.

Každý UML diagram se používá v jiné části vývoje SW:

1. Byznys modelování
    a. Diagram aktivit
    b. Diagram tříd
2. Specifikace požadavků
    a. Diagramy případů užití
    b. Sekvenční diagramy
3. Analýza a návrh
    a. Diagramy tříd
    b. Sekvenční diagramy
    c. Diagramy spolupráce
    d. Stavové diagramy

    e. Diagramy nasazení
4. Implementace
    a. Diagramy komponent
    b. Diagramy nasazení



Diagramy se dělí na 3 základní skupiny

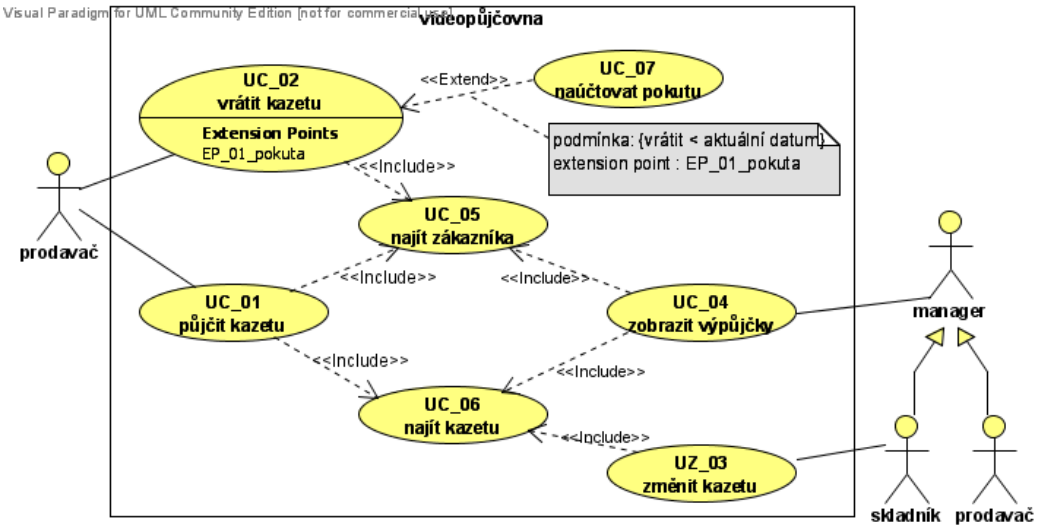
- **Diagramy struktury –** popisují strukturu systému – z čeho je složený
- **Diagramy chování –** Popisuje chování systému – jak funguje
- **Diagramy interakce –** patří pod diagramy chování; popisují interakci mezi jednotlivými částmi systému

Dělení diagramů podle toho, jak postihují různé aspekty systému:

- **Funkční náhled**
    - Diagram případů užití
- **Logický náhled**
    - Diagram tříd
    - Objektový diagram
- **Dynamický náhled popisující chování**
    - Stavový diagram
    - Diagram aktivit
    - Interakční diagramy
        - Sekvenční diagramy
        - Diagramy spolupráce
- **Implementační náhled**
    - Digram komponent
    - Diagram rozmístění

Use case diagram

Účelem use-case je definovat co existuje vně systému (aktéři) a co má být systémem prováděno (případy užití) a přístup aktérů k systému. Vstupem je **byznys model** (modely podnikových procesů). Výsledkem analýzy těchto procesů je **seznam požadovaných funkcí** softwarového systému.



Součástí diagramu je:

- **Aktér** – ten co systém používá, je znázorněn panáčkem vně systému, co přistupuje k různým případům užití
- **Případ užití** – znázorňuje funkci systému. Znázorněn je oválem uvnitř systému
- **Relace –** vazby a vztahy mezi aktéry a případy užití (šipky, čáry). Rozlišujeme několik typů relací:
    - Relace **používá** označována klíčovým slovem <<uses>> vyjadřuje situaci, kdy určitý scénář popsaný jedním případem užití je využíván i jiným případem užití
    - Relace **rozšiřuje** (<<extends>>) vyjadřuje situaci, kde určitý případ užití rozšiřuje jiný či představuje variantní průchody jim popsaným scénářem. Jeden proces může být rozšířen o jiný. (např. Otevřít dokument → Import z jiného formátu)
    - Relace **zahrnuje** (<<include>>) – jeden proces se využívá i v rámci jiného. Případ užití napojený pomocí vazby <<include>> se spustí vždy, když je spuštěn případ, na který je napojen
    - Relace **přístupu k systému** – aktér přistupuje k systému. Je znázorněn plnou čarou se šipkou.
    - Relace **specializace/generalizace** vyjadřuje dědičnost mezi objekty.

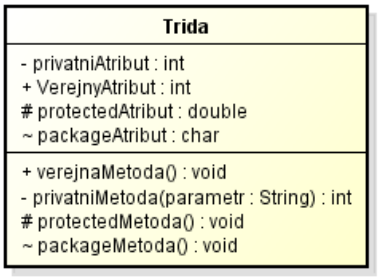
Jednotlivé případy užití se skládají z textové specifikace, ve kterých by mělo být popsáno:

- Co systém děla (ale ne jak)
- Jak a kdy činnost začíná a končí
- Kdy má systém interakci s aktérem
- Které údaje jsou měněny
- Jaké kontroly vstupních údajů jsou prováděny
- Základní, alternativní a chybové průběhy

Diagram tříd

Specifikuje množinu tříd, rozhraní a jejich vzájemné vztahy. Tyto diagramy slouží k vyjádření statického pohledu na systém. Používá se při návrhu aplikace. Diagram tříd je návod pro programátora, takže jeho práce je jen přepsat diagram do kódu. Zdrojem informací pro diagram tříd jsou diagramy sekvenční a spolupráce.

Základem je **třída,** která obsahuje:



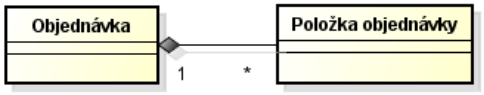
- **Název**
- Ve prostřední části jsou **atributy** s datovými typy
- Před každým atributem je modifikátor přístupu:
    - – privátní atribut
    - + veřejný atribut
    - # protected atribut
    - ~ atribut viditelný v rámci balíku
- V poslední části jsou **metody** a jejich návratový typ

Vztahy mezi třídami

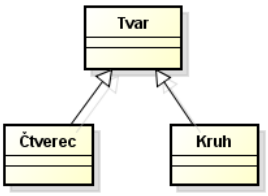
- **Asociace** – určuje základní vztah mezi dvěma entitami (posílají si zprávy). Ty mohou existovat nezávisle na sobě. U čáry lze napsat i **roli –** definuje specifické chování objektu v daném kontextu jeho použiti
    - Definováno plnou čárou
    - Šipka určuje, jaká třída si uchovává odkaz na druhou třídu

- **Agregace** – reprezentuje vztah typu celek – část. Prázdný kosočtverec je umístěn u té entity, která reprezentuje celek. Z hlediska implementace je to entita, která drží kolekci. Entita, která tvoří část může existovat sama o sobě a být součásti i jiných kolekcí.

- **Kompozice** – podobná agregaci, avšak reprezentuje silnější vztah. Část nemá bez celku smysl. Pokud zanikne celek, zanikají automaticky i jeho části.



- **Generalizace** – dědičnost, kdy jedna entita dědí od druhé



- **Realizace** – přerušovaná čára s prázdnou šipkou; když nějaká třída implementuje rozhraní

Multiplicita

- Můžeme uvést u asociace, agregace a kompozice, udává, kolik může mít třída jiných tříd ve vztahu
    - 1 – označuje konkrétní hodnotu – právě 1
    - * - označuje libovolný počet (i 0)

- 1..* - interval

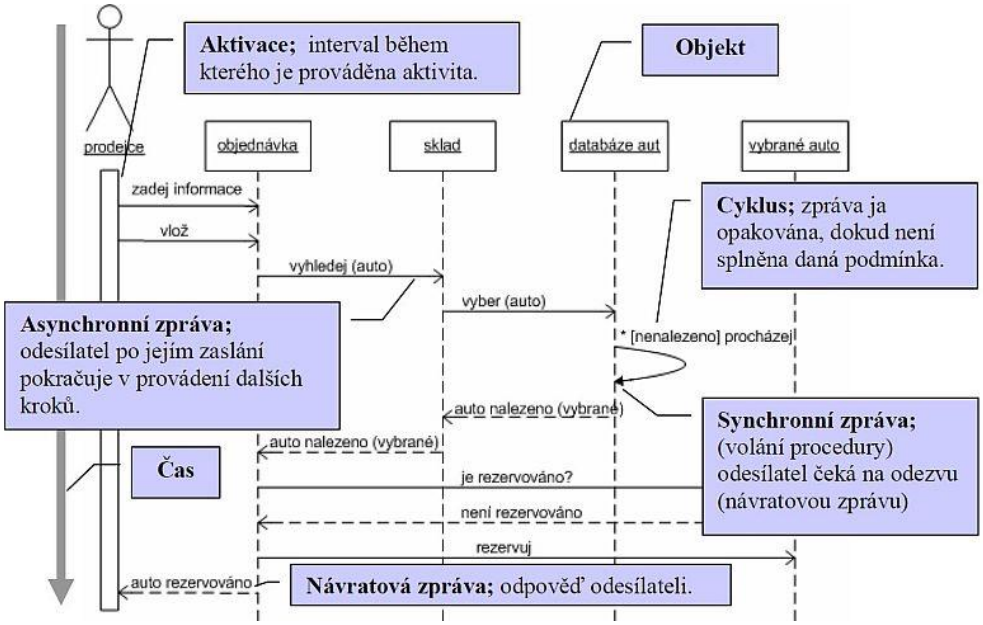
Sekvenční diagram

Objekty v systému zodpovídají za plnění požadované funkcionality poskytnutím svých služeb v komunikaci s jinými objekty, dokáže tedy realizovat jednotlivé scénáře. **Sekvenční (interakční) diagram** definuje interakce mezi objekty, které vedou ke splnění funkcionality.

Komunikace objektů je znázorněna v čase, takže z diagramu lze vyčíst i životní cyklus objektu. Diagram popisuje, jaké zprávy jsou mezi objekty zasílány z pohledu času. Tok času je od shora dolů. Je tvořen objekty uspořádanými do sloupců a šipky mezi nimi odpovídají vzájemně si zasíláním zprávám. Zprávy mohou být následující:

- **Synchronní** – odesílatel čeká na odpověď
- **Asynchronní** – odesílatel nečeká na odpověď a pokračuje ve vykonávání své činnosti.

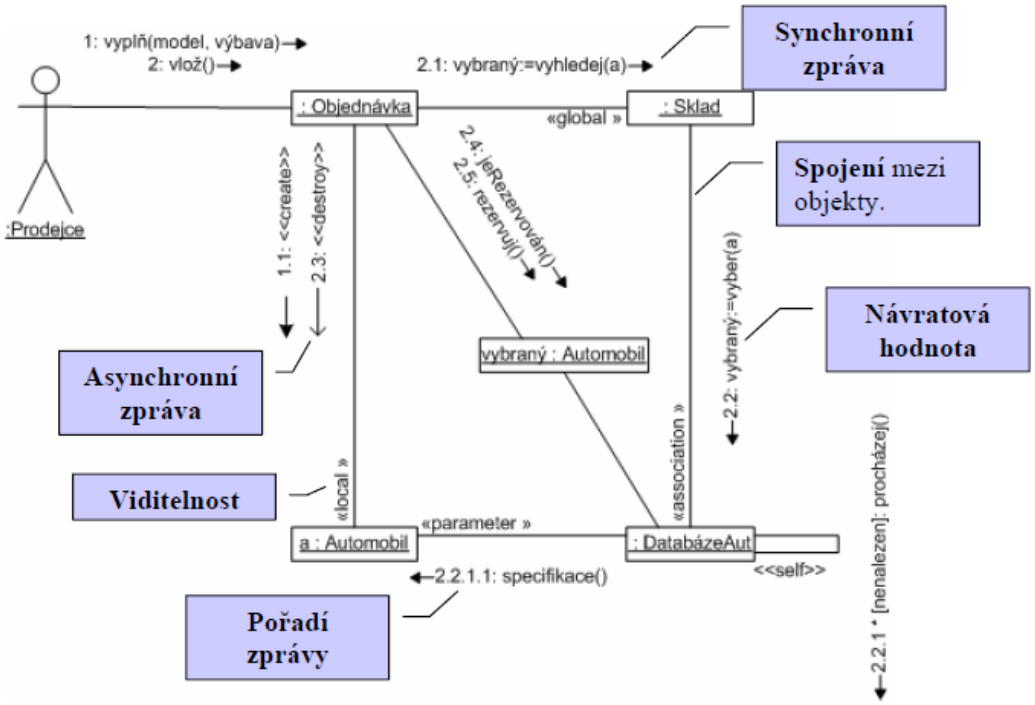
Souvislé provádění nějaké činnosti objektu vyjadřuje svisle orientovaný obdelník.



Diagramy spolupráce

Spolu se sekvencím diagramem patří do skupiny **interakčních diagramů**. Použití vhodné tam, kde chceme zdůraznit strukturální aspekty spolupráce – **kdo s kým komunikuje**. Nezobrazují časové souvislosti. Objekty si mohou posílat zprávy. Diagram spolupráce ukazuje objekty a spojení a zprávy, které si objekty posílají. Skládá se z:

- **Objekt**
- Mezi objekty jsou **spojení**
- **Zpráva** je zakreslená jako šipka (plná = synchronní, normální šipka = asynchronní)
- U šipky pak je text **casovaPosloupnost návratováHodnota := jmeno(argument1, argument2, ...)**
- Časová posloupnost zaslání zpráv je vyjádřena pořadovým číslem.
- Návratová hodnota je vyjádřena operátorem přiřazení :=



Obr. 5.3.2: Diagram spolupráce

Popisuje vzájemnou komunikaci mezi objekty s důrazem na vyjádření jejich topologie – rozložení a vzájemné propojení. Diagram zavádí i následující typy viditelnosti vzájemně spojených objektů:

- <<local>> - vyjadřuje situaci, kdy objekt je vytvořen v těle operace a po jejím vykonání je zrušen
- <<global>> - specifikuje globálně viditelný objekt
- <<parameter>> - vyjadřuje fakt, že objekt je předán druhému jako argument na něj zaslané zprávy
- <<association>> - specifikuje trvalou vazbu mezi objekty
- <<self>> - definuje speciální auto asociaci, tedy odkaz na sebe sama

Stavové diagramy

Zobrazuje životní cyklus objektu, události způsobující přechody z jednoho stavu do jiného a akce, které vyplývají z této změny stavu. Popisuje celý životní cyklus objektů z pohledu dynamického chování. Sekvenční diagram a diagram spolupráce dokážou modelovat pouze určité časové snímky. Diagram je sestavován pro každá objekt (pro jeho třídu).

Vyjadřuje stavy určitého objektu a přechody mezi těmito stavy. Obsahuje:

- **Stav** – situace, kdy objekt splňuje nějakou podmínku; počáteční stav je označen plným kolečkem a koncový stav kolečkem s puntíkem.
- **Průchod** – spojení mezi dvěma stavy; objekt přejde z jednoho stavu do druhého za splnění určitých podmínek
- **Volba** – rozděluje přechod na dva segmenty. Každá větev má své podmínky (prostě if)
- **Rozvětvení** – Použije se v situaci, kdy jeden zdrojový stav je rozdělen na dva čí více cílových stavů
- **Spojení** – použije se, když více zdrojových stavů přechází do jednoho společného cílového stavu.

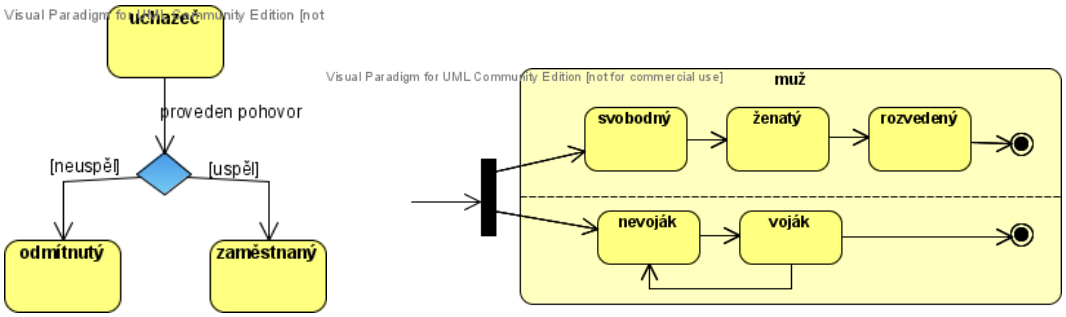


Diagram komponent

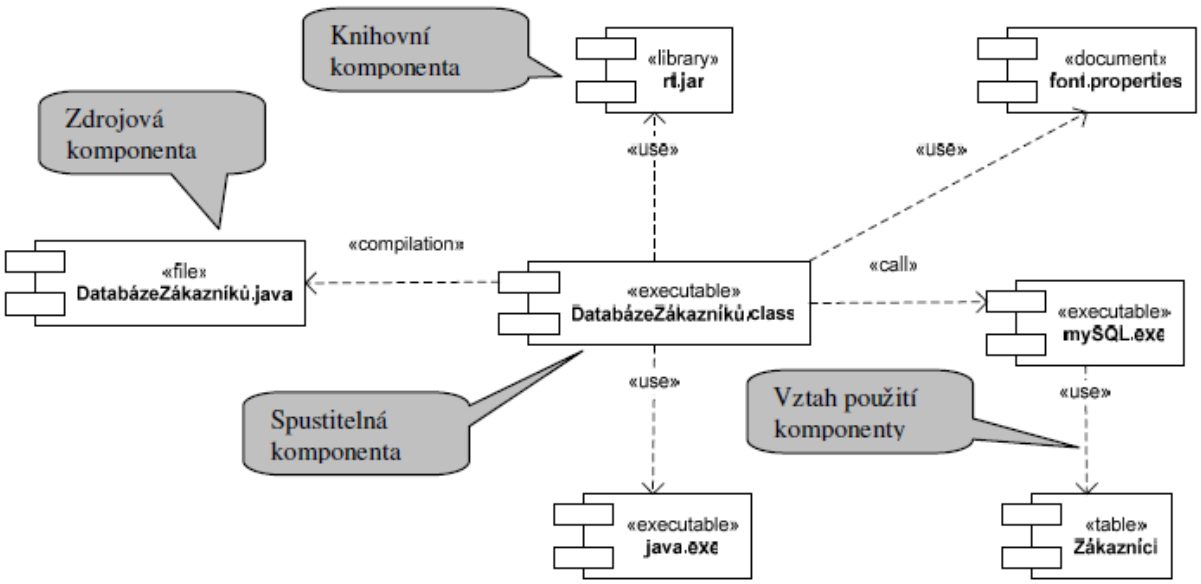
Smyslem tohoto modelu je specifikovat fyzickou strukturu výpočetních prostředků, které budou použity pro provoz SW. Popisuje konfiguraci technických prostředků umožňujících běh SW systému. Zachycuje rovněž rozmístění implementovaných SW komponent na jednotlivé technické prostředky (server, počítač) - ukazuje rozložení jednotlivých softwarových komponent na hardwarových zdrojích (ale to bych spíš řekl, že to je záležitost implementační části, ne návrhové a to je taky diagram nasazení)

Cílem specifikace diagramu komponent je rozhodnout a popsat fyzickou organizaci zdrojových, binárních či datových souborů

Základním prvkem diagramu nasazení jsou tzv. **uzly** (nodes), které jsou vzájemně propojeny **komunikačními cestami**.

Uzly mohou být typu:

- **<<device>>** je uzel, který představuje typ fyzického zařízení (hardware).
- **<<execution environment>>** reprezentuje typ prostředí zpracování softwaru (např. webový server).

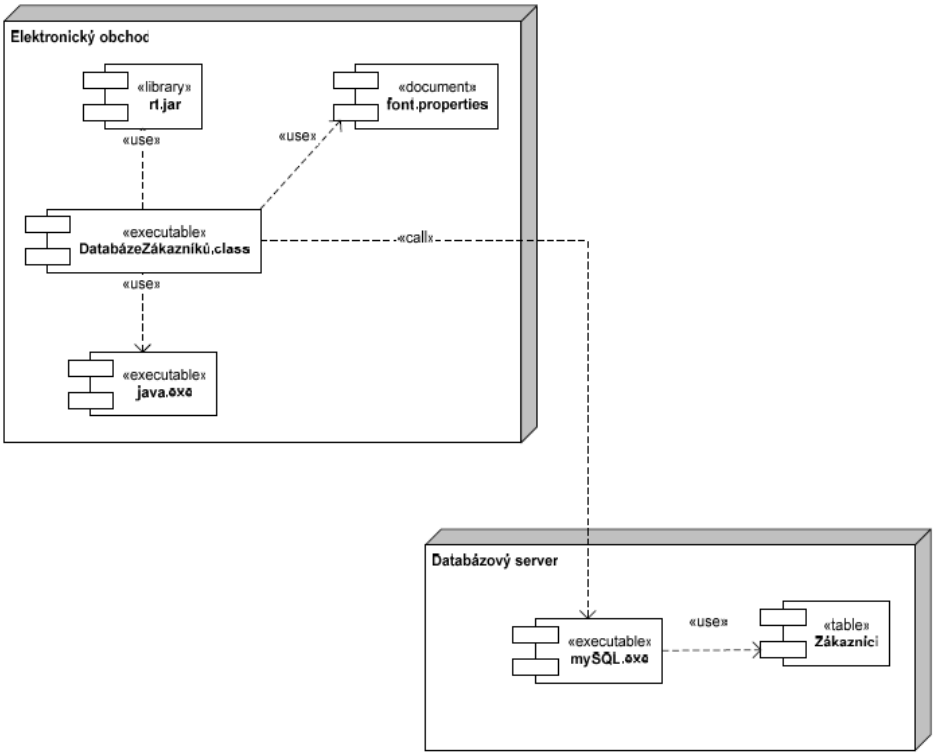


Obr. 2.21: Diagram komponent

Diagram nasazení

Účelem diagramu nasazení je specifikovat způsob jak budou jednotlivé komponenty umístěny na jednotlivých technických prostředcích reprezentovaných počítači. Po implementaci nových komponent a jejich propojení s již existujícími komponentami je nutné rozhodnout o tom, na kterých uzlech počítačové sítě budou tyto umístěny.

Příklad: Aplikace DatabázeZákazníků bude spolu s několika komponentami umístěna na serveru, zatímco databázový stroj bude spuštěn na svém databázovém serveru, kde budou umístěné i odpovídající tabulky.



Obr. 2.22: Diagram nasazení