Softwarové inženýrství

1) Softwarový proces. Jeho definice, modely a vyspělostní úrovně.

Softwarový proces

Proces obecně je po částech uspořádaná množina kroků směřujících k dosažení cíle. Je-li cíl vytvoření softwaru pak mluvíme o **softwarovém procesu**. Disciplína zabývající se problémy softwarového procesu se nazývá **softwarové inženýrství**.

Vzhledem k tomu, že vývoj softwaru je relativně nová problematika dodnes není jasně definováno jak by měl správný softwarový proces vypadat. Byla však vyvinuta řada různých přístupu k vývoji softwaru. Lze říct že základem všech je vodopádový model, který lze nalézt ve většině používaných přístupů.

Model je zjednodušená realita pro snadnější pochopení.

Statická struktura procesu definuje KDO (**role**), CO (**artefaty**), JAK (**aktivity** a jejich **toky**) a KDY to má vytvořit.

Definice softwarového procesu a vyspělostní úrovně

Ve vztahu k této definici je nutné si dále uvědomit následující:

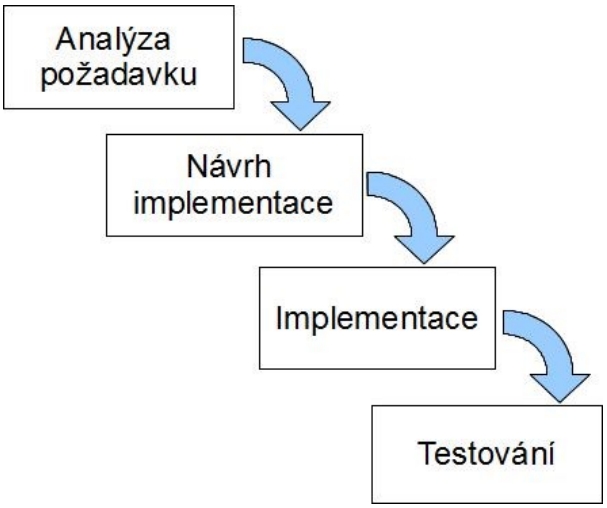
- Krokem může být aktivita nebo opět podproces (hierarchická dekompozice procesu).
- Aktivity a podprocesy mohou probíhat v čase souběžně, tudíž je vyžadována jejich koordinace..
- Je nutné zajistit opakovatelnost použití procesu ve vztahu k jednolivým softwarovým projektům, tedy zajistit jeho znovupoužitelnost. Cílem je dosáhnout stabilních výsledků vysoké úrovně kvality.
- Řada činností je zajišťována lidmi vybavenými určitými schopnostmi a znalostmi a majícím k dispozici technické prostředky nutné pro realizaci těchto činností.
- Softwarový produkt je realizován v kontextu organizace s danými ekonomickými možnostmi a organizační strukturou.

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 softwarový proces a každý projekt je rešen případ od případu (ad hoc).
2. ***Opakovatelná*** (Repeatable) - firma identifikovala v jednotlivých projektech opakovatelné postupy a tyto je schopna reprodukovat v každém novém projektu.
3. ***Definovaná*** (Defined) - softwarový proces je definován (a dokumentován) na základě integrace dříve identifikovaných opakovatelných kroků.
4. ***Řízená*** (Managed) - na základě definovaného softwarového procesu je firma schopna jeho řízení a monitorování.
5. ***Optimalizovaná*** (Optimized) - zpětnovazební informace získaná dlouhodobým procesem monitorování softwarového procesu je využita ve prospěch jeho optimalizace.

Vodopádový model

Vodopádový model je složen z několika kroků, které následují po sobě a nemůžou začít dříve, než skončí předchozí fáze. Začíná analýzou požadavků na software na základě které je zhotoven návrh implementace (programátorská analýza), podle které se systém naimplementuje. Finální software se otestuje a nasadí k zákazníkovi. Do vodopádového modelu mohou být zahrnuty ještě kroky Nasazení a Údržba systému, které na obrázku nejsou.

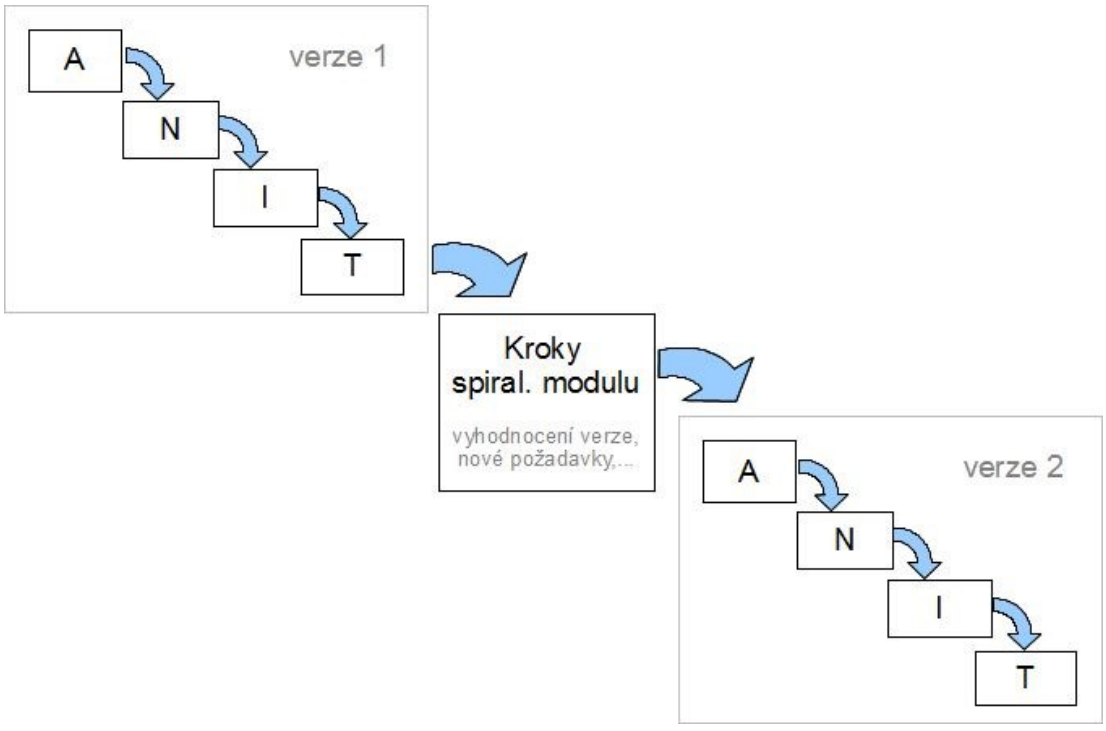


Nevýhody vodopádového modelu:

- **Dlouhý vývoj** - prodleva mezi zadáním a výsledným produktem. Během vývoje zákazník nic nevidí a ani my nedokážeme odhalit výslednou kvalitu produktu.
- **Přesné zadání** - na začátku je nutné vytvořit přesné a korektní zadání, aby byl výsledný produkt v pořádku.

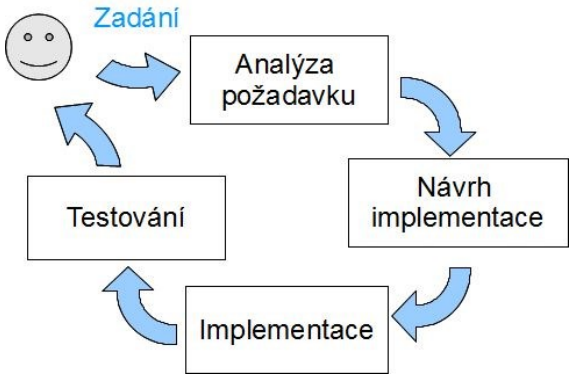
Z těchto důvodů byl vodopádový model dále modifikován a vznikly tak tyto modely:

- **Inkrementální model** založen na principu verzování. Finální projekt je rozdělen na dílčí verze. Do novějších verzí je postupně přidávána další funkčnost systému a až poslední verze obsahuje plnohodnotný systém. Jednotlivé verze se vyvíjejí vodopádovým modelem, ale čas na jejich realizaci je menší a můžeme sledovat vývoj systému. Viz. obrázek nahoře bez meziverzových kroků.



- **Spirálový model** je to samé co výše zmíněný inkrementální, s tím že mezi jednotlivé verze byly vloženy další procesy jako například zhodnocení verze z pohledu finálního systému, či přidání nových požadavků zákazníka. Viz. obrázek nahoře.
- **Průzkumné programování** - urputná snaha zjistit od zákazníka co vlastně chce, tím že mu předkládáme hotové systémy. Analýza požadavku je odbytá hrubou specifikací a vše směřuje k rychlé realizaci softwaru, ke které se zákazník vyjádří.

Model RUP (Rational Unified Proces) - iterační přístup



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.

Software je zde **vyvíjen iterativně** (**v cyklech**). 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.

Při vývoji se využívá **existujících komponent.** Vývoj softwaru se tak přesouvá do oblasti skládání produktu z prefabrikovaných komponent.

Model softwarového systému je **vizualizován pomocí UML,** který pomáhá uchopit strukturu a chování výsledné architektury produktu.

Součástí RUP jsou i metody pro **správu požadavků**, které obsahují postupy jak dostat ze zadavatele požadavky na systém, jak s nimi dále pracovat. Ve všech aktivitách se neustále **ověřuje kvalita** softwarového produktu. RUP zahrnuje i princip **řízení změn**, který se stará o monitorování změn, ke kterým v systému došlo, ať už vlivem doplnění požadavků, či oprav chyb apod. Řízení změn umožňuje zaručit, že každá změna je přijatelná a všechny změny systému jsou sledovatelné.

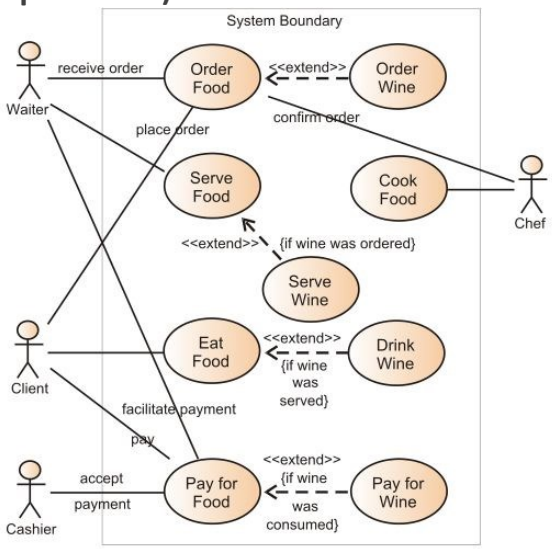
2) Vymezení fáze „sběr a analýza požadavků“. Diagramy UML využité v dané fázi

Specifikace požadavků

Specifikace a analýza požadavků je první fáze vývoje softwaru. Cílem je definovat požadavky na software a popsat jeho funkčnost. Výsledkem této fáze by měly být dokumenty, které se stanou součásti smlouvy mezi zadavatelem a vývojovým týmem.

V rámci specifikace požadavků je třeba analyzovat procesy u zadavatele, které bude software řešit, nebo s ním nějak jinak souvisí. K tomu jasné a srozumitelnému popisu těchto procesů dobře poslouží **Use-case diagram, Sekvenční diagram** popřípadě **Diagram aktivit.**

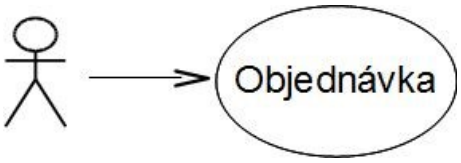
Use-case diagram (diagram případu užití)



Use-case diagram

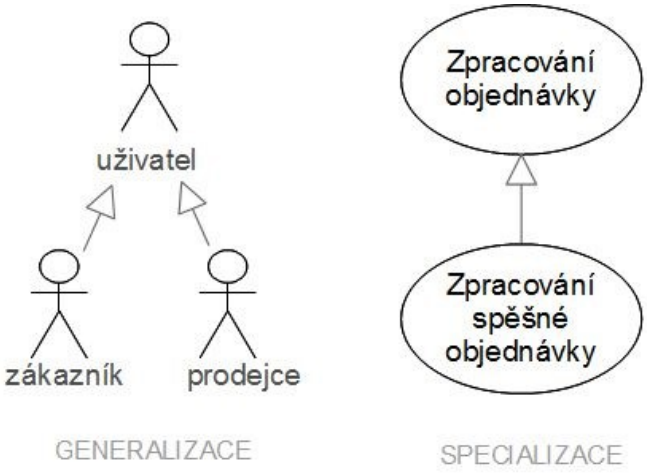
UML diagram který se zabývá aktéry, vztahy mezi nimi a přístupy aktérů k systému. Součástí diagramu je:

- **Aktér** - ten co systém používá. Znázorněn panáčkem co stojí mimo systém a přistupuje k různým případům užitím.
- **Případ užití (use-case)** - znázorňuje funkci systému. V diagramu je znázorněn oválkem uvnitř systému.
- **Relace** - vazby a vztahy mezi aktéry a případy užití. V diagramu jsou znázorněny šipkami případně čárami. Rozlišujeme několik typů relací:
    - Relace přístupu k systému (mezi aktérem a use-case). V diagramu znázorněná plnou čárou či šipkou.



    - Relace <<include>> ve smyslu zahrnuje. Používá se uvnitř systému, kdy jeden proces (use-case) se využívá i v rámci jiného.

    - Relace <<extend>> ve smyslu rozšiřuje. Používá se v případě že jeden proces může být rozšířen o jiný.
      Příklad z obrázku nahoře: proces Eat Foot může být rozšířen o proces Drink Wine, v případě že je naservírováno víno.
    - Relace specializace/generalizace vyjadřuje dědičnost mezi objekty.
- **Scénář** - unikátní sekvence akcí skrz use-case (jedna cesta/ instance provedení)

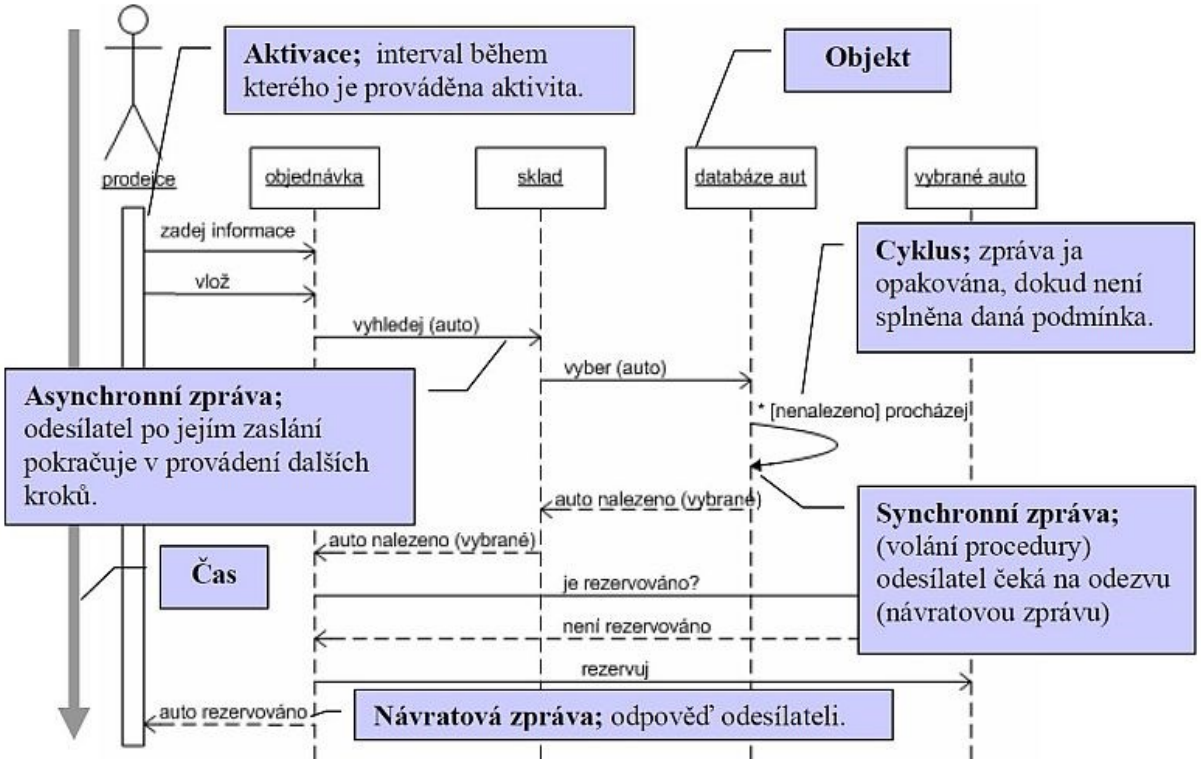


Doporučená forma use-case:

- má vyjadřovat co systém dělá (ne jak) a co od něj očekávají aktéři
- měly by být používány jen pojmy problémové domény - žádné neznámé termíny
- případy užití by měly být co nejjednodušší - ať jim rozumí i zadavatelé - nepoužívat příliš <include> a <extend>
- nepoužívat příliš funkční dekompozici (specializaci)
- primární aktéři umístěni vlevo a pojmenováni krátkým podstatným jménem
- každý aktér by měl být propojen s minimálně jedním use-case
- základní use case vlevo a další kreslit směrem doprava, rozšiřující use-case směrem dolů

Sekvenční diagram

Sekvenční (Iterační) diagram popisuje funkce systému z pohledu objektů a jejich iterací. Komunikace mezi objekty je znázorněná v čase, takže z diagramu můžeme vyčíst i životní cyklus objektu.



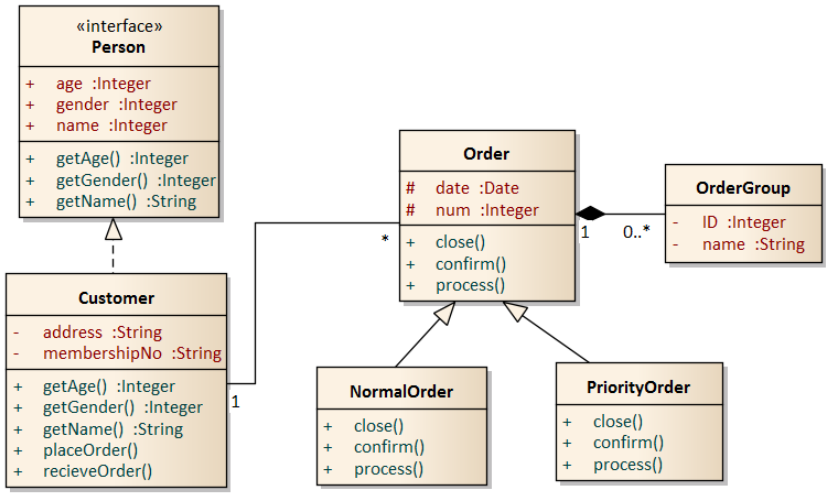
3) Vymezení fáze „Návrh“. Diagramy UML využité v dané fázi. Návrhové vzory – členění, popis a příklady.

Návrh implementace

Návrh implementace je druhá fáze vývoje softwaru. Jde o abstrakci zdrojového kódu, která bude sloužit jako hlavní dokument programátorům v další implementační fázi. Model návrhu dále upřesňuje model analýzy ve světle skutečného implementačního prostředí.

V fázi návrh implementace se uplatňují tyto UML diagramy:

- **Diagram tříd** -



Třídní diagram

statický pohled na systém zobrazuje třídy a vztahy mezi nimi.

- **Sekvenční diagram** - popisuje iterace mezi objekty v čase.
- **Diagram spolupráce** - zaměřuje se podobně jako sekvenční na iterace mezi objekty, ale bez časové posloupnosti.
- **Stavový diagram** - popisuje životní cyklus objektu a stavy ve kterých se může nacházet.

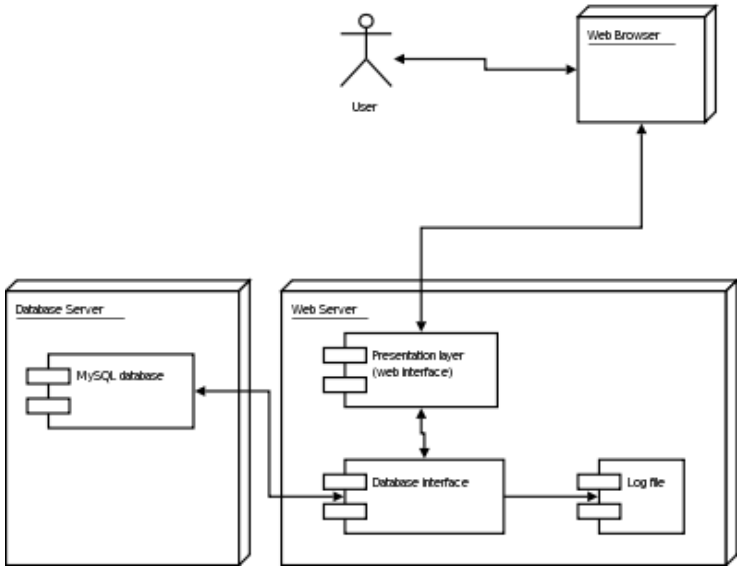


Diagram nasazení ukázka webového servru

Návrhové vzory

Návrhové vzory jsou metodiky (šablony) pro řešení různých problému, se kterými vývojář může setkat. Objektově orientované návrhové vzory typicky ukazují vztahy a interakce mezi třídami a objekty, aniž by určovaly implementaci konkrétní třídy.

Návrhové vzory se dělí do těchto 3 základních skupin:

- **Creational Patterns (vytvářející)** - řeší problémy související s vytvářením objektů v systému. Snahou těchto návrhových vzorů je popsat postup výběru třídy nového objektu a zajištění správného počtu těchto objektů. Většinou se jedná o dynamická rozhodnutí učiněná za běhu programu.
- **Structural Patterns (strukturální) -** představují skupinu návrhových vzorů zaměřujících se na možnosti uspořádání jednotlivých tříd nebo komponent v systému. Snahou je zpřehlednit systém a využít možností strukturalizace kódu.
- **Behavioral Patterns (chování) -** zajímají se o chování systému. Mohou být založeny na třídách nebo objektech. U tříd využívají při návrhu řešení především principu dědičnosti. V druhém přístupu je řešena spolupráce mezi objekty a skupinami objektů, která zajišťuje dosažení požadovaného výsledku.

Každá z těchto skupin obsahuje mnoho návrhových vzorů, které si nyní rozvedeme:

Creational patterns (Vzory týkající se tvorby objektů)

- **Abstract Factory** (Abstraktní továrna) – Definuje rozhraní pro vytváření rodin objektů, které jsou na sobě závislé nebo spolu nějak souvisí bez určení konkrétní třídy. Klient je odstíněn od vytváření konkrétních instancí objektů.
- **Factory Method** (Tovární metoda) – Definuje rozhraní pro vytváření objektu, které nechává potomky rozhodnout o tom, jaký objekt bude fakticky vytvořen. *Tovární metoda nechává třídy přenést vytváření na potomky.
- **Builder** (Stavitel) – Odděluje tvorbu komplexu objektů od jejich reprezentace tak, aby stejný proces tvorby mohl být použit i pro jiné reprezentace.
- **Singleton** (Jedináček) – Potřebujete-li, aby měla třída maximálně jednu instanci.
- **Prototype** (Prototyp, Klon) – Specifikuje druh objektů, které se mají vytvořit použitím prototypového objektu. Nové objekty se vytváří kopírováním tohoto prototypového objektu.

Structural Patterns (Vzory týkající se struktury programu)

- **Adapter** (Adaptér) – Potřebujete-li, aby spolu pracovaly dvě třídy, které nemají kompatibilní rozhraní. Adaptér převádí rozhraní jedné třídy na rozhraní druhé třídy.
- **Bridge** (Most) – Oddělí abstrakci od implementace, tak aby se tyto dvě mohly libovolně lišit.
- **Composite** (Strom, Složenina) – Komponuje objekty do stromové struktury a umožňuje klientovi pracovat s jednotlivými i se složenými objekty stejným způsobem.
- **Decorator** (Dekorátor) – Použijeme jej v případě, že máme nějaké objekty, kterým potřebujeme přidávat další funkce za běhu. Nový objekt si zachovává původní rozhraní.
- **Facade** (Fasáda) – Nabízí jednotné rozhraní k sadě rozhraní v podsystému. Definuje rozhraní vyšší úrovně, které zjednodušuje použití podsystému.
- **Flyweight** (Muší váha) – Je vhodná pro použití v případě, že máte příliš mnoho malých objektů, které jsou si velmi podobné.
- **Proxy** – Nabízí náhradu nebo zástupný objekt za nějaký jiný pro kontrolu přístupu k danému objektu.

Behavioral Patterns (Vzory týkající se chování)

- **Observer** (Pozorovatel) – V případě, kdy je na jednom objektu závislých mnoho dalších objektů, poskytne vám tento vzor způsob, jak všem dát vědět, když se něco změní. Observer je možné použít v situaci, kdy je

definována závislost jednoho objektu na druhém. Závislost ve smyslu propagace změny nezávislého objektu závislým objektům (pozorovatelům). Nezávislý objekt musí informovat závislé objekty o událostech, které je mohou ovlivnit.

- **Command** (Příkaz) – Zapouzdřete požadavek jako objekt a tím umožněte parametrizovat klienty s různými požadavky, frontami nebo požadavky na log a podporujte operace, které jdou vzít zpět.
- **Interpreter** (Interpret) - Vytváří jazyk, což znamená definování gramatických pravidel a určení způsobu, jak vzniklý jazyk interpretovat.
- **State** (Stav) – Umožňuje objektu měnit své chování, pokud se změní jeho vnitřní stav. Objekt se tváří, jako kdyby se stal instancí jiné třídy.
- **Strategy** (Strategie) – Zapouzdřuje nějaký druh algoritmů nebo objektů, které se mají měnit, tak aby byly pro klienta zaměnitelné.
- **Chain of responsibility** (Zřetězení zodpovědnosti) – Řeší jak zaslat požadavek bez přesného vymezení objektu, který jej zpracuje.
- **Visitor** (Návštěvník) – Reprezentuje operaci, která by měla být provedena na elementech objektové struktury. Visitor vám umožní definovat nové operace beze změny tříd elementů na kterých pracuje.
- **Iterator** (Iterátor) – Nabízí způsob, jak přistupovat k elementům skupinového objektu postupně bez toho, abyste vystavovali vnitřní reprezentaci tohoto objektu.
- **Mediator** (Prostředník) – Umožňuje zajistit komunikaci mezi dvěma komponentami programu, aniž by byly v přímé interakci a tím musely přesně znát poskytované metody.
- **Memento** (Memento) – Bez porušování zapouzdření zachyťte a uložte do externího objektu interní stav objektu tak, aby ten objekt mohl být do tohoto stavu kdykoliv později vrácen.
- **Template method** (Šablonová metoda) – Definuje kostru toho, jak nějaký algoritmus funguje, s tím, že některé kroky nechává na potomcích. Umožňuje tak potomkům upravit určité kroky algoritmu bez toho, aby mohli měnit strukturu algoritmu.

4) Objektově orientované paradigma. Pojmy třída, objekt, rozhraní. Základní vlastnosti objektu a vztah ke třídě. Základní vztahy mezi třídami a rozhraními. Třídní vs. instanční vlastnosti.

Objektově orientované paradigma.

Objektově-orientované programování (OOP) je metodika vývoje softwaru. Paradigma OOP popisuje způsob vývoje a zápisu programu a způsob uvažování o problému.

Základní paradigma OOP

- Při řešení úlohy vytváříme model popisované reality - popisujeme entity a interakci mezi entitami
- Abstrahujeme od nepodstatných detailů - při popisu/modelování entity vynecháváme nepodstatné vlastnosti entit
- Postup řešení je v řadě případů efektivnější než při procedurálním přístupu (ne vždy), kdy se úlohy řeší jako posloupnost příkazů

Cíle OOP

- Je vedeno snahou o znovupoužitelnost komponent.
- Rozkládá složitou úlohu na dílčí součásti, které jdou pokud možno řešit nezávisle.
- Přiblížení struktury řešení v počítači reálnému světu (komunikující objekty).
- Skrytí detailů implementace řešení před uživatelem.

OOP != třídy a objekty - představuje přístup, jak správně navrhnout strukturu programu

Model OOP

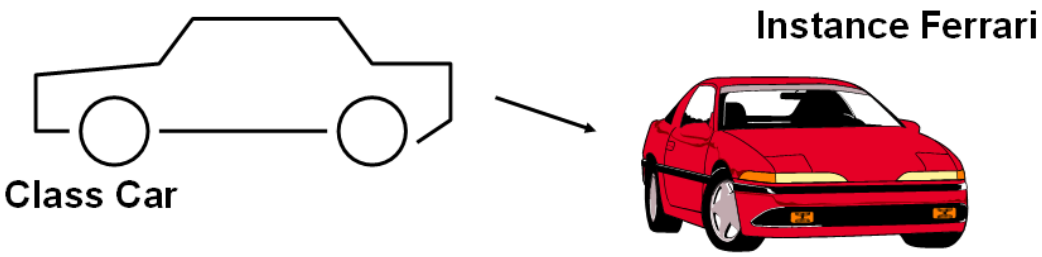
OOP definuje program jako soubor spolupracujících komponent (objektů) s přesně stanoveným chováním a stavem. Metody OOP napodobují vzhled a chování objektu z reálného světa s možností velké abstrakce. Model OOP se skládá z několika následujích konstrukcí:

Objekt

Jednotlivé prvky modelované reality (jak data, tak související funkčnost) jsou v programu seskupeny do entit, nazývaných objekty. Objekty si pamatují svůj stav a navenek poskytují operace přístupné jako metody pro volání. Objekt je také instancí třídy.

Třída (Class)

Třída slouží jako šablona pro vytváření instancí tříd - objektů. Seskupuje objekty stejného typu a podchycuje jejich podstatu na obecné úrovni. (Samotná třída tedy nepředstavuje vlastní informace, jedná se pouze o předlohu; data obsahují až objekty.) Třída definuje data a metody objektů.



Interface (rozhraní)

Interface předepisuje třídě, která od ní bude odvozena, jaké metody, případně properties musí implementovat. Odvozený objekt může implementovat i další metody. Interface nedefinuje žádné proměnné - pouze konstanty, ani neobsahuje naimplementované metody - obsahuje pouze abstraktní metody bez jejich implementace (jejich hlavičky). Třída může implementovat libovolný počet rozhraní (narozdíl od dědičnosti). Pokud vytvoříte rozhraní, které dědí od jiného rozhraní, automaticky tak přebírá všechny jeho metody a konstanty.

Abstraktní třída

Je to takový hybrid mezi rozhraním a klasickou třídou. Od klasické třídy má schopnost implementovat vlastnosti (proměnné) a metody, které se na všech odvozených třídách budou vykonávat stejně. Od rozhraní zase získala možnost obsahovat prázdné abstraktní metody, které si každá odvozená podtřída musí naimplementovat sama. S těmito výhodami má abstraktní třída i pár omezení, a to že jedna podtřída nemůže zdědit víc abstraktních tříd a od rozhraní přebírá omezení, že nemůže vytvořit samostatnou instanci (operátorem new).

1. Abstraktní třída	**2. Rozhraní**
• může obsahovat instanční proměnné • může obsahovat konkrétní metody i metody bez implementace (abstraktní) • lze rozšiřovat pouze jednu třídu	• nesmí obsahovat proměnné • všechny metody jsou bez implementace • do jedné třídy lze implementovat více rozhraní

Rysy OOP

- **Abstrakce** – programátor, potažmo program, který vytváří, může abstrahovat od některých detailů práce jednotlivých objektů. Každý objekt pracuje jako černá skříňka, která dokáže provádět určené činnosti a komunikovat s okolím, aniž by vyžadovala znalost způsobu, kterým vnitřně pracuje.

- **Zapouzdření** (Encapsulation)– zaručuje, že objekt nemůže přímo přistupovat k „vnitřnostem“ jiných objektů, což by mohlo vést k nekonzistenci. Každý objekt navenek zpřístupňuje rozhraní, pomocí kterého (a nijak jinak) se s objektem pracuje.

- **Skládání** – Objekt může obsahovat jiné objekty.

- **Delegování** – Objekt může využívat služeb jiných objektů tak, že je požádá o provedení operace.

- **Dědičnost** (Inheritance) – objekty jsou organizovány stromovým způsobem, kdy objekty nějakého druhu mohou dědit z jiného druhu objektů, čímž přebírají jejich schopnosti, ke kterým pouze přidávají svoje vlastní rozšíření. Tato myšlenka se obvykle implementuje pomocí rozdělení objektů do tříd, přičemž každý objekt je instancí nějaké třídy. Každá třída pak může dědit od jiné třídy (v některých programovacích jazycích i z několika jiných tříd).
- **Polymorfismus** – odkazovaný objekt se chová podle toho, jaké třídy je instancí. Pokud několik objektů poskytuje stejné rozhraní, pracuje se s nimi stejným způsobem, ale jejich konkrétní chování se liší podle implementace. U polymorfismu podmíněného dědičností to znamená, že na místo, kde je očekávána instance nějaké třídy, můžeme dosadit i instanci libovolné její podtřídy. U polymorfismu nepodmíněného dědičností je dostačující, jestliže se rozhraní (nebo jejich požadované části) u různých tříd shodují, pak jsou vzájemně polymorfní. Polymorfismus bývá často vysvětlován na obrázku se zvířaty, která mají všechna v rozhraní metodu Speak(), ale každé si ji vykonává po svém.

- **Genericita** - je možnost programovacího jazyka definovat místo typů jen „vzory typů“, kde typy proměnných, použité v definici (ADT typy) jsou vyvedeny vně definice jako parametry a jsou určeny později klientskou aplikací.

  Příklad:

  List<T> ...kde T je libovolný typ. Když List alokujeme, použijeme

  List<int> intList = new List<int>(), a tento list poté akceptuje jen čísla typu int.

  Kouzlo genericity vynikne pak v kombinací s dědičností, kdy do seznamu mohou být vloženy nejen objekty typu T, ale i objekty všech možných dědiců třídy (typu) T).

Třídní vs. instanční vlastnosti

Instanční proměnné

- udávají vlastnosti objektů
- přístup ze třídy nebo pomocí metod get() a set()
- vznikají s objektem

Statické proměnné

- nepopisují vlastnosti objektů
- existují pouze jednou pro třídu
- mohou existovat bez objektu
- přístup ze třídy nebo pomocí jména třídy

5) Mapování UML diagramů na kód

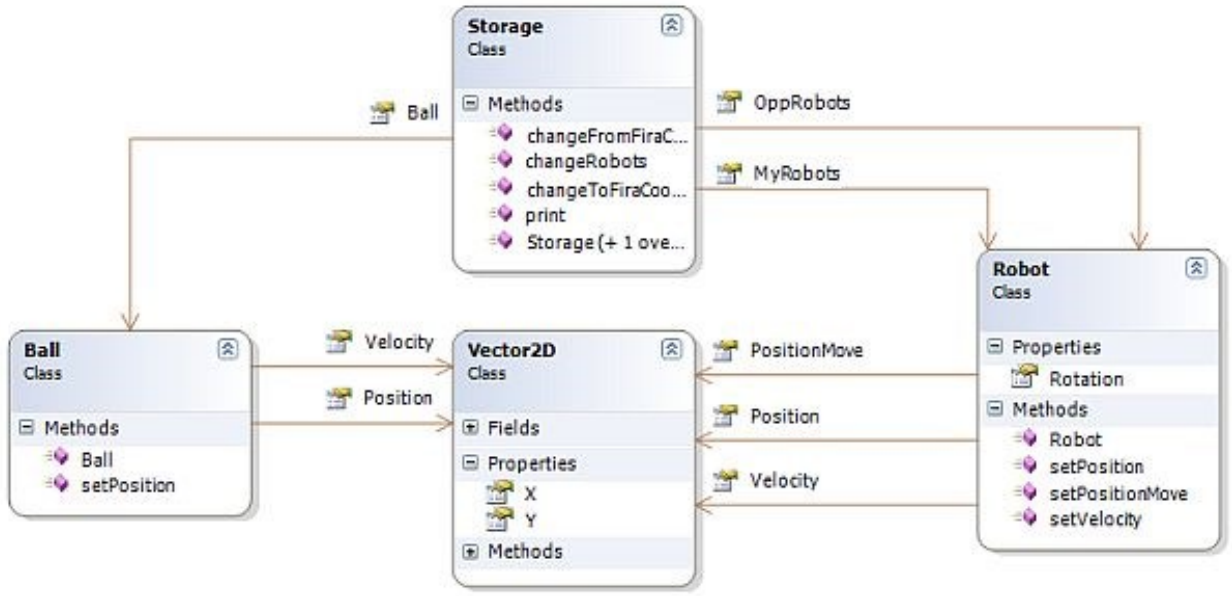
K mapování UML diagramů na kód dochází v první části implementační fáze vývoje. Jde o automatické vygenerování zdrojového kódu z UML diagramu.

Z ***diagramu tříd*** například vygeneruje jednotlivé rozhraní, třídy s proměnnými a prázdnými metodami. Velmi hezký tutoriál o tom jak se mapuje class diagram na kód najdete zde.

Z ***diagramů komponent*** pak vyčte a do generovaného kódu přidá požadované komponenty. **Komponenta** je už existující a vyměnitelná část systému s daným rozhraním, skrze které jí systém používá. Jde o komponentu napsanou v zdrojovém kódu, nebo už zkompilovanou komponentu v binárním kódu, či další komponenty reprezentovanými databázovými tabulkami, dokumenty, apod.

Úkolem samotné implementace je pak dopsat těla metod, jejichž chování může být popsáno v ***diagramu aktivit***.

Ukázka kódu z diagramu tříd




```
public class Storage
{
    public Ball Ball { get; set; }
    public Robot[] MyRobots { get; set; }
    public Robot[] OppRobots { get; set; }

    public Storage( ){}
}

public class Ball
{
    private Vector2D Position { get; set; }
    private Vector2D Velocity { get; set; }

    public Ball(){}
```

```
    public void setPosition(double x, double y){}
}…
```

6) Správa paměti (v jazycích C/C++, JAVA, C#, Python), virtuální stroj, podpora paralelního zpracování a vlákna.

Správa paměti

Správa paměti je v informatice soubor metod, které operační systém používá při přidělování operační paměti jednotlivým spuštěným procesům.

V jazyce Java i C# je správa paměti plně automatizovaná. O uvolňování paměti se stará separátní vlákno, které běží s nízkou prioritou a zajišťuje kontinuální sledování nepoužitých bloků paměti. Přidělování paměti se provádí operátorem new.

Výhody automatické správy paměti jsou následující:

- Programátor se může věnovat řešení skutečného problému;
- Rozhraní modulů jsou přehlednější - není třeba řešit problém zodpovědnosti za uvolnění paměti pro objekty vytvořené různými moduly;
- Nastává menší množství chyb spojených s přístupem do paměti;
- Správa paměti je často mnohem efektivnější.

Tento přístup má však i své nevýhody:

- Paměť může být zachována jen proto, že je dostupná, i když není dále využita.
- Automatická správa paměti není k dispozici ve starších, ale často používaných jazycích.

Pro zjištění toho, které úseky paměti se již nepoužívají, je k dispozici mnoho algoritmů. Většinou spoléhá automatická regenerace paměti na informace o tom, na které bloky paměti neukazuje žádná programová proměnná. V zásadě existují dvě skupiny metod - metody založené na sledování odkazů a metody založené na čítačích odkazů.

Správa paměti je v jazyce **C#** plně automatizovaná, paměťový prostor se přiděluje operátorem **new**, jeho uvolnění zajistí systém řízení běhu programu.

V jazyce **Java** je správa paměti rovněž plně automatizovaná, o uvolňování paměti se stará separátní vlákno, které běží s nízkou prioritou a zajišťuje kontinuální sledování nepoužitých bloků paměti. Přidělování paměti se provádí operátorem **new**.

Úrovně správy paměti

- **Technické vybavení**
  - registry, cache
- **Operační systém**
  - virtuální paměť
  - segmentace, stránkování
- **Aplikace**
  - Přidělování paměti
  - Regenerace paměti
    - Manuální – delete, dispose, free(), …
    - Automatická – *garbage collection*

Garbage collector (GC) - je obvykle část běhového prostředí (programovacího) jazyka, nebo přídavná knihovna, podporovaná kompilátorem, hardware, operačním systémem, nebo jakoukoli kombinací těchto tří. Má za úkol automaticky určit, která část paměti programu je už nepoužívaná, a připravit ji pro další znovupoužití.

Mark & Sweep

- Jeho varianta použitá v GC .NET.

Algoritmus nejdříve nastaví všem objektům, které jsou v paměti, speciální příznak navštíven na hodnotu ne. Poté projde všechny objekty, ke kterým se lze dostat. Těm, které takto navštívil, nastaví příznak na hodnotu ano. V okamžiku, kdy se už nemůže dostat k žádnému dalšímu objektu, znamená to, že všechny objekty s příznakem navštíven majícím hodnotu ne jsou odpad - a mohou být tedy uvolněny z paměti.

Virtuální stroj

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

Hardwarový virtuální stroj

Původní význam pro virtuální stroj, někdy nazývaný též hardwarový virtuální stroj, označuje několik jednotlivých totožných pracovních prostředí na jediném počítači, z nichž na každém běží operační systém. Díky tomu může být aplikace psaná pro jeden OS používána na stroji, na kterém běží jiný OS, nebo zajišťuje vykonání sandboxu, který poskytuje větší úroveň izolace mezi procesy, než je dosaženo při vykonávání několika procesů najednou (multitasking) na stejném OS.

Jedním využitím může být také poskytnutí iluze mnoha uživatelům, že používají celý počítač, který je jejich „soukromým“ strojem, izolovaným od ostatních uživatelů, přestože všichni používají jeden fyzický stroj. Další výhodou může být to, že start (bootování) a restart virtuálního počítače může být mnohem rychlejší, než u fyzického stroje, protože mohou být přeskočeny úkoly jako například inicializace hardwaru.

Podobný software je často označován termíny jako virtualizace a virtuální servery. Hostitelský software, který poskytuje tuto schopnost je často označovaný jako hypervisor nebovirtuální strojový monitor (virtual machine monitor).

Aplikační virtuální stroj

Dalším významem termínu virtuální stroj je počítačový software, který izoluje aplikace používané uživatelem na počítači. Protože verze virtuálního stroje jsou psány pro různé počítačové platformy, jakákoliv aplikace psaná pro virtuální stroj může být provozována na kterékoli z platforem, místo toho, aby se musely vytvářet oddělené verze aplikace pro každý počítač a operační systém. Aplikace běžící na počítači používá interpret nebo Just in time kompilaci.

Jeden z nejlepších známých příkladů aplikace virtuálního stroje je **Java Virtual Machine** od firmy Sun Microsystem. Java Virtual Machine umí zpracovat mezikód (Java bytecode), který je obvykle vytvořen ze zdrojových kódů programovacího jazyka Java. Díky tomu že je JVM k dispozici na mnoha platformách, je možné aplikaci v Javě vytvořit pouze jednou a spustit na kterékoliv z platforem, pro kterou je vyvinut JVM (např. Windows, Linux).

Paralelní zpracování

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.

Pro zrychlení výpočtu se používá buď zdánlivý paralelní běh úloh pomocí **multitaskingu** nebo víceprocesorové systémy nebo počítačové clustery. **Paralelizace** funguje na principu rozdělování složitějších úloh na jednodušší a je považována za obtížnější formu programování. Existuje několik rozdílných forem paralelních výpočtů: bitové, instrukční, datové a paralelní úlohy.

Paralelní počítačové programy jsou mnohem náročnější na vytvoření, než je tomu u sekvenčních programů, protože souběžnost zavádí několik nových tříd potencionálních programátorských chyb, z kterých je nejčastější **souběh**. Komunikace a synchronizace mezi jednotlivými úlohami je nejčastěji největší překážkou pro dobrou výkonnost paralelních programů.

Žádný program nemůže běžet rychleji než nejdelší řetězec na sobě závislých kalkulací (označuje se jako **kritická sekce**), protože výpočty závislé na předchozích výpočtech musejí být provedeny v určeném pořadí. Nicméně, mnoho algoritmů není jen jedním velkým řetězem závislých výpočtů, ale vyskytují se zde příležitosti k provádění paralelních výpočtů.

Vlákna (Thread)

Operační systémy používají pro oddělení různých běžících aplikací procesy. ***Proces*** je tvořen paměťovým prostorem a jedním nebo více vlákny. Vlákno je samostatně prováděný výpočetní tok (posloupnost instrukcí), který běží v rámci procesu. Každému vláknu přísluší vlastní priorita a řada systémových struktur.

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. Vzhledem k tomu, že tyto časové úseky jsou z pohledu uživatele velmi krátké, je výsledný dojem i na počítači s jediným procesorem takový, jako by pracovalo více vláken současně. V případě, že máme k dispozici více procesorů, jsou mezi ně vlákna přidělována ke zpracování a k současnému běhu pak skutečně dochází.

Přepnutí mezi vlákny bývá výrazně rychlejší než u procesovém multitaskingu, neboť vlákna sdílejí stejnou paměť a uživatelská práva svého mateřského procesu a není je třeba při přepínání měnit. Vlákno také spotřebuje méně paměti a je rychlejší na vytváření.

Vlákna je možné vytvořit i čistě na aplikační úrovni bez nativní podpory operačního systému (využitím sdílené paměti a dalších technik). Takto vzniklá vlákna je poté možné spouštět postupně v jednotlivých procesech operačního systému nebo takzvaně m:n, tedy v několika vláknech operačního systému současně spouštět větší počet aplikačních vláken.

Samotným zvyšováním počtu vláken však obvykle odpovídajícího zvýšení výkonu aplikace nedosáhneme. Naopak se doporučuje, abychom používali co nejméně vláken a tím omezili spotřebu systémových prostředků a nárůst režie. Typické problémy jsou následující:

- Pro ukládání kontextových informací se spotřebovává dodatečná paměť, a tedy celkový počet procesů a vláken, které mohou v systému současně existovat, je omezený.
- Obsluha velkého počtu vláken spotřebovává významnou část času procesoru. Existuje-li tedy příliš mnoho vláken, většina z nich příliš významně nepostupuje. Navíc pokud je většina vláken v jednom procesu, dostávají se vlákna jiných procesů na řadu méně často.
- Organizace programu s mnoha vlákny je složitá a může být zdrojem mnoha chyb. Zejména je obtížné zajistit jejich správnou synchronizaci.

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.

Životní cyklus

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 dokončeníoperace vstupu/výstupu.
- **Dead** - do tohoto stavu se vlákno dostane ukončením metody run nebo voláním metody stop.

Vícevláknové aplikace a ladění

Hlavní problémy vícevláknových aplikací souvisí se synchronizací

- **uváznutí** - **Deadlock** - je situace, kdy úspěšné dokončení první akce je podmíněno předchozím dokončením druhé akce, přičemž druhá akce může být dokončena až po dokončení první akce. Deadlocku můžeme zabránit například tím, že proces musí o všechny prostředky, které potřebuje, zažádat najednou. Buď je všechny dostane, nebo nedostane ani jeden.
- **souběh** (race conditions) - přístup více vláken ke sdíleným proměnným a alespoň jedno vlákno nevyužívá synchronizačních mechanismů. Vlákno čte hodnotu, zatímco jiné vlákno zapisuje. Zápis a čtení nejsou atomické a data mohou být neplatná. K zabránění souběhu se používají zámky (**lock**), jejichž použitím se zajistí, že jen jedno vlákno může v jeden okamžik přistoupit k označenému resource souboru nebo části kódu.
- **vyhladovění** - Vyhladovění - stav, kdy jsou vláknu neustále odepírány prostředky. Bez těchto prostředků program nikdy nedokončí svůj úkol.

Semafor je založen na atomických operacích V (*verhogen*, též označováno jako *up*) a P (*proberen*, též označováno jako *down*). Operace *down* otestuje stav čítače a v případě že je nulový, zahájí čekání. Je-li nenulový, je čítač snížen o jedničku a vstup do kritické sekce je povolen. Při výstupu z kritické sekce je vyvolána operace *up*, která odblokuje vstup do kritické sekce pro další (čekající) proces. Čítač je možné si představit jako omezení počtu procesů, které mohou zároveň vstoupit do kritické sekce nebo například jako počitadlo volných prostředků. Tato implementace neodstraňuje problém aktivního čekání.

7) Zpracování chyb v moderních programovacích jazycích. Princip datových proudů – pro vstup a výstup. Rozdíl mezi znakově a bytově orientovanými datovými proudy.

Zpracování chyb

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 chyb během výpočtu obvykle žádnou podporu 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.

Jazyk Java, podobně jako např. C++, využívá pro ošetření výjimek metodu 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 těchto podtříd odpovídá jedné skupině výjimek. V první skupině jsou výjimky, jejichž výskyt představuje vážný problém v činnosti aplikace a které by programátor neměl zachycovat. Druhou skupinu tvoří výjimky, jejichž ošetření má smysl - jde například o výjimky jako pokus o otevření neexistujícího souboru, chybný formát čísla apod. Do této skupiny by měly patřit také veškeré výjimky definované uživatelem.

Příkaz try-catch

Zpracování výjimky se vždy vztahuje k bloku programu omezenému příkazem **try** následovaném jedním nebo více bloky **catch** popisujícími způsob ošetření jednotlivých výjimek.

Příklad chceme-li ošetřit chyby při práci se souborem (v jazyce Java):

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

Příklad ošetření chyby při chybném parsování čísla (parsujeme například textový řetězec):

```
class TiskCisel {
  public static void main(String[] args) {
    try {
      int n = Integer.parseInt(args[0]);
      for(int i = 1; i <= n; i++)
        System.out.println(i);
    } catch( NumberFormatException e ) {
```

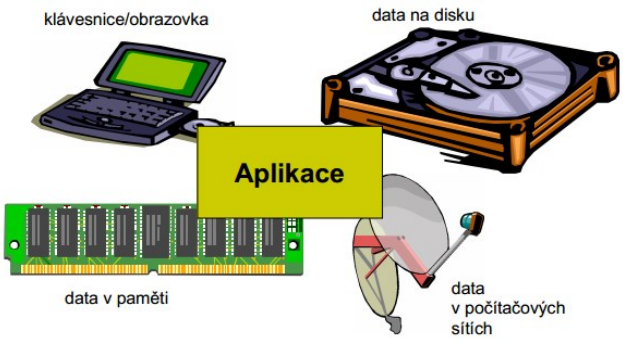
```
      System.err.println("Chyba: Nespravny format argumentu");
    }
  }
}
```

Potřebujeme-li vyvolat v programu **vlastní výjimku**, je třeba nejprve vytvořit vhodnou třídu, odvozenou od třídy Exception nebo některé její podtřídy.

```
class MojeVyjimka extends Exception {
  MojeVyjimka(String msg) { super(msg); }
}
class Vyjimky {
static void zpracuj(int x) throws MojeVyjimka
  {
    System.out.println("Volani zpracuj " + x);
    if( x < 0 )
      throw new MojeVyjimka("Parametr nesmi byt zaporny");
  }

  public static void main(String args[])
  {
    try {
      zpracuj(1);
      zpracuj(-1);
    } catch( MojeVyjimka e ) {
      e.printStackTrace();
    }
  }
}
```

Princip datových proudů



Datové proudy (dále jen proudy) jsou sekvence dat. Proud je definován svým vstupem a výstupem, těmi mohou být například soubor na disku, zařízení (vstupní klávesnice nebo myš, výstupní displej) nebo jiný program.

Vstupně–výstupní zařízení (I/O):

- ***znakové***: vstupní – klávesnice, výstupní – monitor, tiskárna.
- ***bloková***: pevný disk, flash disk, SD, microSD. . .

Z klávesnice čteme sekvenčně znak po znaku (***sekvenční přístup***), u binárního diskového souboru můžeme libovolně přistupovat ke zvolené části dat – ***náhodný přístup***.

Primitivní proudy pouze přenášejí binární data, specializované proudy umí s přenášenými daty manipulovat. Tyto manipulační schopnosti se liší podle typu přenášených dat.

Typy přenášených informací:

- ***binární data*** – obrázky, zvuk, objekty, ... - data uložené ve stejné formě v jaké jsou uloženy v počítači
- ***textová data*** – představují řádky textů v ASCII (abc) nebo UNICODE (مرحبا नमस्ते)

Typy proudů

Binární

Binární proudy umožňují přenést libovolná data. Základní operací definovanou v InputStreamu je metoda read, pomocí které můžeme z proudu přečíst jeden bajt. Analogicky výstupní proud definuje metodu write.

Čtení a zápis po jednotlivých bajtech je velmi pomalé, zvláště pokud uvážíme, že na druhé straně proudu může být disk – a každý dotaz může velmi snadno znamenat nutnost nového vystavení hlaviček.

Textový

Znakové proudy fungují stejným způsobem jako ty binární, pouze operují s textem Poměrně důležitou poznámkou je, že Java interně uchovává řetězce ve znakové sadě Unicode. Z toho plyne, že při každém zápisu a čtení ze znakového proudu dochází k překódování daného řetězce (znaků).

Proudy s vyrovnávací pamětí

- čtení/zápis s využitím vyrovnávací paměti
- rozhraní: BufferedReader/ BufferedWriter

Datové proudy

Java obsahuje třídy pro pohodlné čtení a zápis primitivních datových typů a typu String. Nejrozšířenější třídy jsou DataInputStream a DataOutputStream. String je ukládán v kódování UTF-8. Údaje takto uložené např. do souboru nejsou uživatelsky přívětivé a s výjimkou řetězců čitelné. Metody pro čtení jsou readDouble, readInt, readUTF. Obdobně metody pro zápis mají předponu write.

Objektové proudy

Většina standardních tříd implementuje rozhraní Serializable (serializovatelný), které je nezbytné pro jejich podporu objektovými proudy. Objektové proudy rozšiřují datové proudy, takže objektové proudy umí pracovat i s primitivními datovými typy. Nové metody jsou readObject a writeObject. Pokud metoda readObject vrátí jiný než očekávaný objekt, vyhodí výjimku typu ClassNotFoundException. Pokud se objekt neskládá jen z primitivních typů ale i z referencí na další objekty, je potřeba zachovat tyto reference. Proto je při zápisu objektu uložit i všechny objekty, na které má daný objekt odkaz. Podobně se bude chovat čtecí proud, který se bude snažit zrekonstruovat celou takovou síť objektů

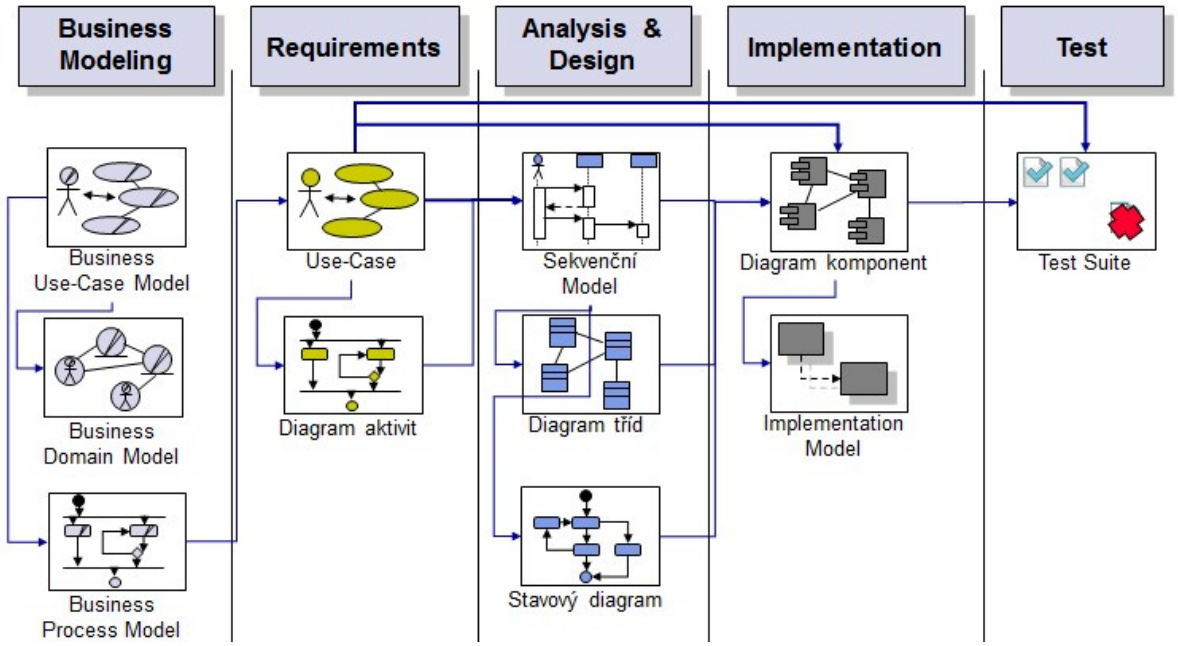
8) Jazyk UML – typy diagramů a jejich využití v rámci vývoje.

UML (Unified Modeling Language)

UML je jednotný jazyk pro tvorbu diagramů. Definuje standardy pro jednotnou strukturu diagramů. Pomocí něj můžeme realizovat nejrůznější schemata napříč procesem vývoje softwaru .

UML jazyk umožňuje **specifikaci** (struktura a model), **vizualizaci** (grafy), **konstrukci** (vygenerování kódu např. z class diagramu) a **dokumentaci artefaktů** softwarového systému.

Každý z UML diagramů se také používá v jiné fázi vývoje softwaru.



Dělení UML

Postihující různé aspekty systému:

1) **Funkční náhled**

- Diagram případů užití (Use case)

2) **Logický náhled**

- Diagram tříd
- Objektový diagram

3) **Dynamický náhled popisující chování**

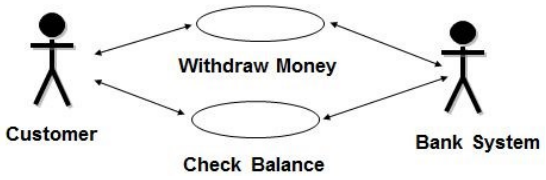
- Stavový diagram
- Diagram aktivit
- Interakční diagramy
- Sekvenční diagramy
- Diagramy spolupráce

4) **Implementační náhled**

- Diagram komponent
- Diagram nasazení

Funkční náhled

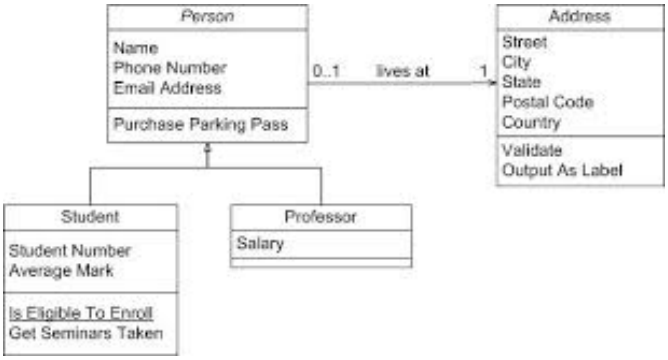
Diagram případu užití (***Use-case diagram***)



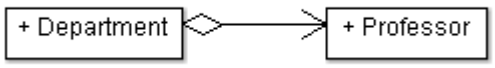
- poskytuje funkční náhled systému (kdo se systémem pracuje a jak).
- Uplatňuje se pro realizaci DFD, ve vývojové fázi specifikace požadavků.

Logický náhled

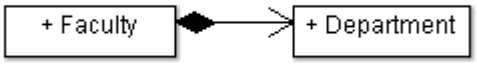
Třídní diagram (***Class diagram***) - diagram struktury tříd poskytuje logický náhled na systém. Znázorňuje datové struktury, operace u objektů a také jejich vazby. Třídní diagram se využívá pro tvorbu ERD a při návrhu implementace.



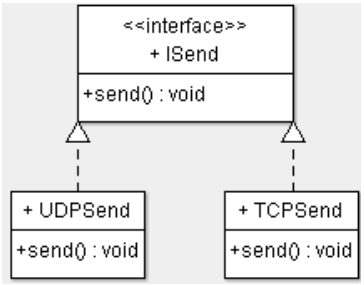
Typy vazeb v třídním diagramu



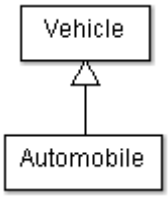
Agregace - vztah vyjdařující katedra má profesory, zánik department neznamená zánik professor



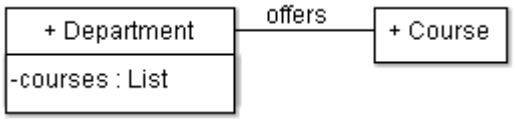
Kompozice - nejsilnější asociace, existence odkazovaného objektu (department) bez majitele (faculty) nemá smysl a zaniká i s ním



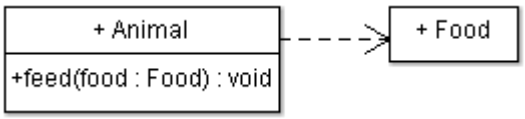
Realizace - implementace rozhraní



Generalizace (druhý obr.) - dědičnost - automobil dědí z vehicle

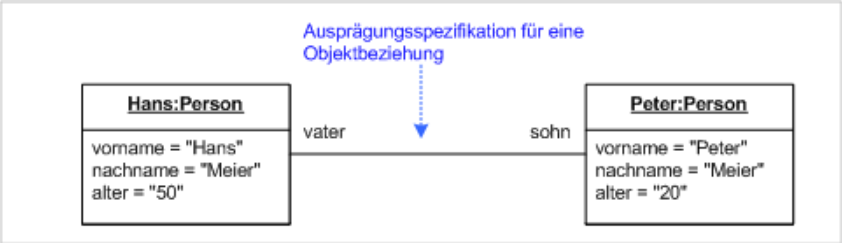


Asociace - vztah mezi instancemi - posílají si zprávy



Závislost - třída animal používá třídu food (je na ni závislá)

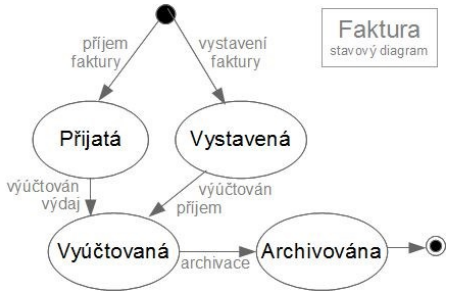
- **Objektový diagram** (***Object diagram***) zobrazuje instance tříd - objekty, někdy nazýván jako ***instanční diagram***.
  Je snímkem objektů a jejich vztahů v systému v určitém časovém okamžiku, který chceme z nějakého důvodu zdůraznit. Využívá se v datové analýze pro ERD.



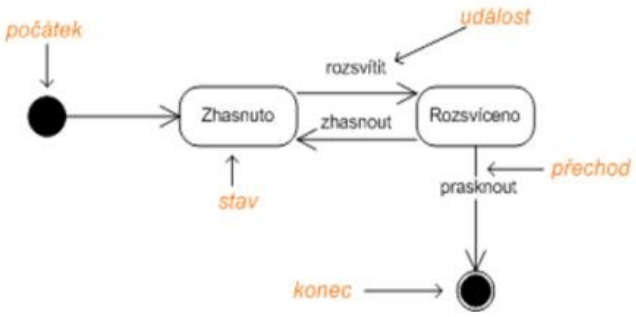
Objektový diagram

Dynamický náhled popisující chování

Stavový diagram (***State diagram***) zobrazuje životní cyklus objektů a stavy do kterých se během svého života dostávají. Obsahuje 3 základní prvky – stav, událost, přechod. Využívá se při návrhu implementace.



Stavový diagram – Faktura



Stavový diagram - životní cyklus žárovky

Diagram aktivit - (***Activity diagram***) grafické znázornění algoritmů. Tento diagram se používá pro modelování procedurální logiky, procesů a zachycení workflow. Každý proces v diagramu aktivit je reprezentován sekvencí jednotlivých kroků, které určují řídící tok. Tento diagram obsahuje i rozhodovací symbol (kosočtverec) s podmínkou, kdy tok procesu pokračuje na základě odpovědi jednou z větví. Diagram aktivit se využívá pro tvorbu minispecifikací a dalších podrobných analýz a při specifikaci požadavků.

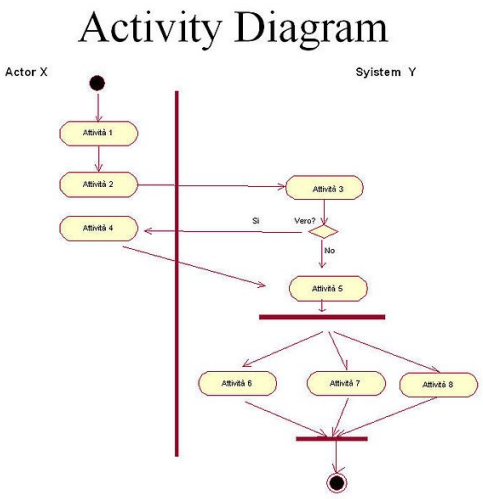


Diagram aktivit

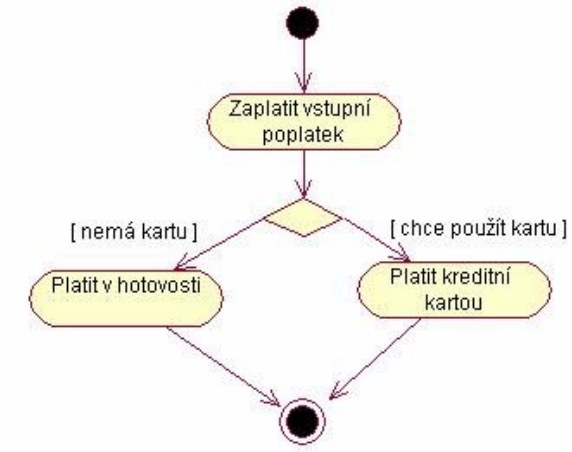
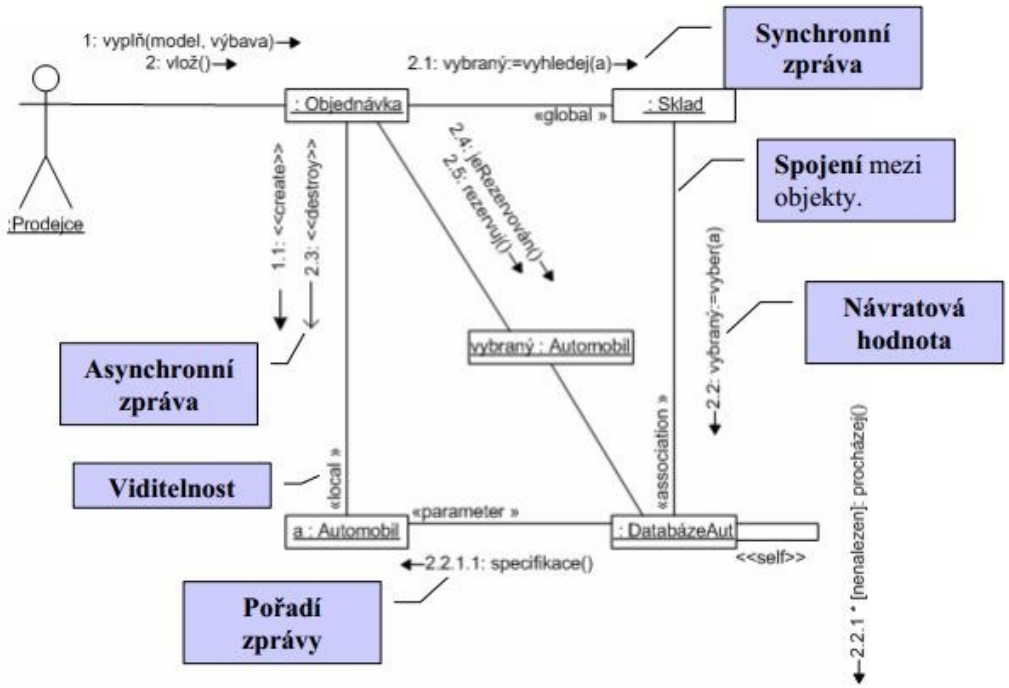


Diagram aktivit - platba

Sekvenční diagram (***Sequence diagram***) znázorňuje komunikaci mezi objekty v čase.
Používá se ve vývojové fázi specifikace požadavků a návrhu implementace. Viz okruh 2.

Diagram spolupráce (***Colaboration diagram***) - dynamický pohled na systém znázorňující chování objektů v systému. Diagram spolupráce upřednostňuje při popisu vzájemné komunikace mezi objekty jejich topologii, tedy jejich rozložení a vzájemné spojení. Podobný jako sekvenční ale bez času. Využití v implementační fázi.



Časová posloupnost zaslání zpráv je vyjádřena jejich pořadovým číslem. Návratová hodnota je vyjádřena operátorem přiřazení :=. Opakované zaslání zprávy je dáno symbolem * a v hranatých závorkách uvedením podmínky opakování cyklu. Navíc tento 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 (někdy se také hovoří o tzv. známostním spojení).

Implementační náhled

Diagram komponent - ilustruje organizaci a závislosti mezi softwarovými komponentami. Zdrojové komponenty tvoří soubory vytvořené použitým programovacím jazykem. Diagram spustitelných komponent specifikuje všechny komponenty vytvořené námi i ty, které nám dáva k dispozici implementační prostředí.

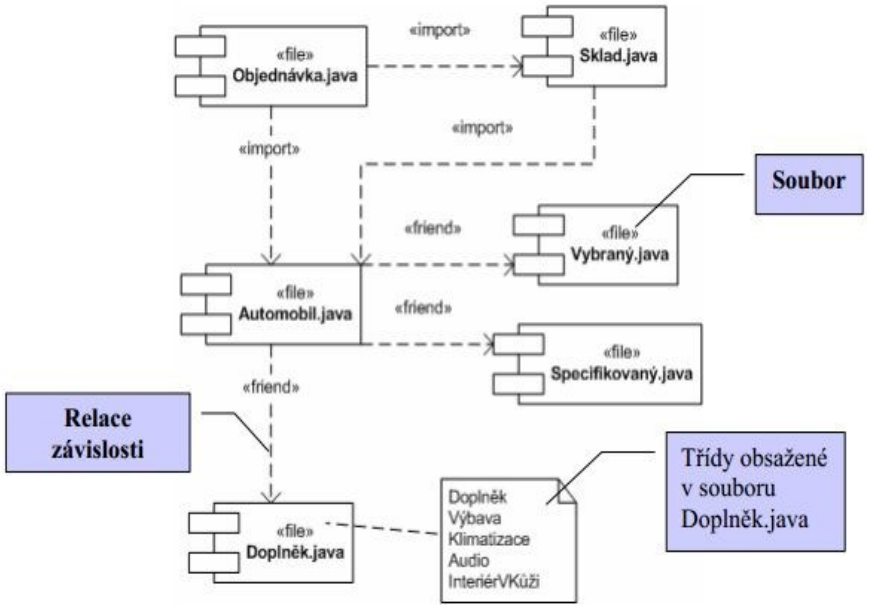


Diagram komponent - binární kód

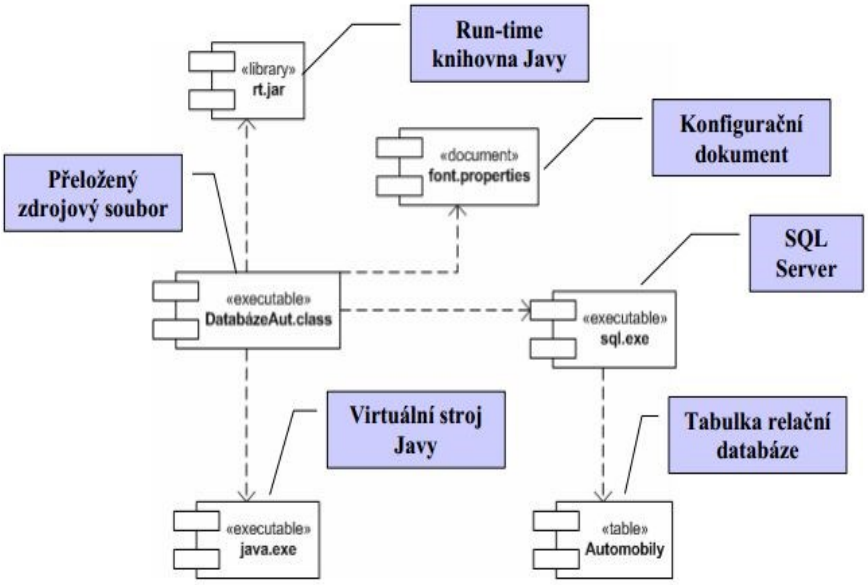


Diagram nasazení - upřesněný nejen ve smyslu konfigurace technických prostředků, ale především z hlediska rozmístění implementovaných softwarových komponent na jednotlivé technické prostředky reprezentované v počítači.

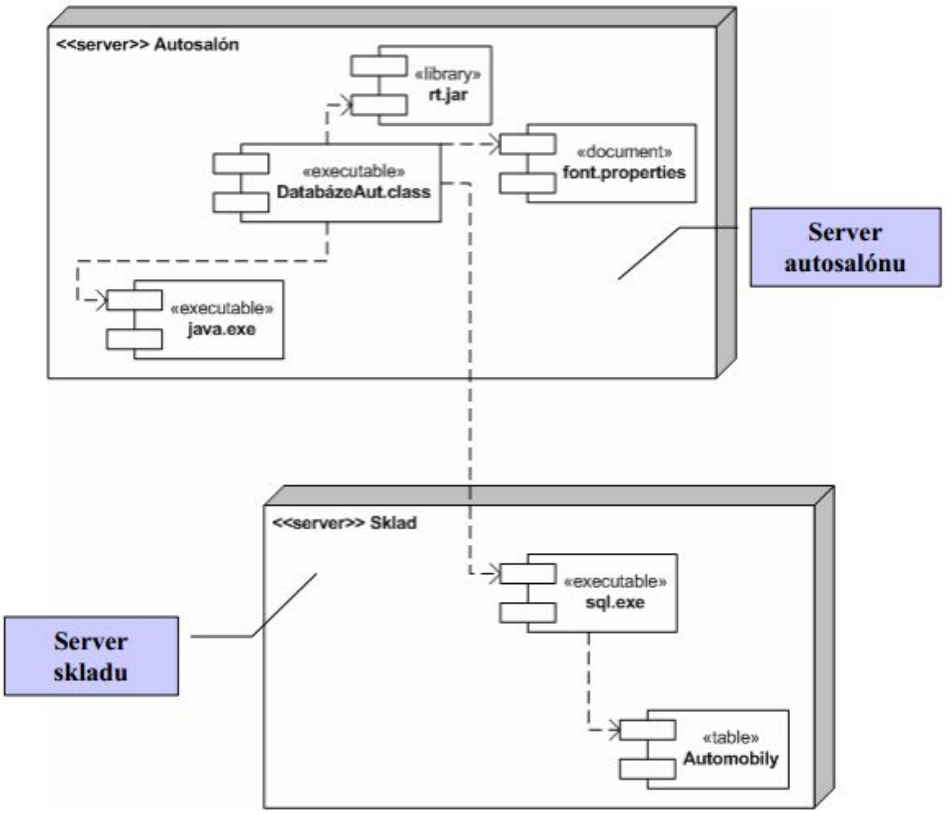


Diagram nasazení