II. Softwarové inženýrství

Update: 3. května 2018

Obsah

1 Softwarový proces. Jeho definice, modely a úrovně vyspělosti.

Softwarové inženýrství je inženýrská disciplína zabývající se praktickými problémy vývoje rozsáhlých softwarových systémů.

1.1 Softwarový proces

Softwarový proces je po částech uspořádaná množina kroků směřujících k vytvoření nebo úpravě softwarového díla.

- 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 jednotlivým 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.

1.2 Modely softwarového procesu

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.

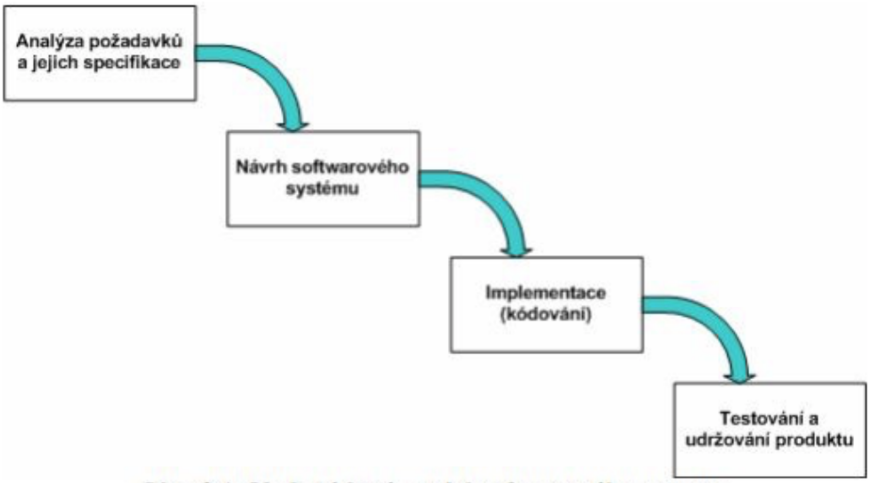
1.2.1 Vodopádový model

Stal se základem téměř všech modelů softwarového procesu. Vychází z **rozdělení životního cyklu softwarového díla** na **čtyři** základní fáze:

1. **analýza požadavků** a jejich **specifikace**,
2. **návrh** softwarového systému,
3. **implementace**,
4. **testování** a udržování vytvořeného produktu.

Princip vodopádu spočívá v tom, že **následující množina činností spjatá s danou fází** nemůže započít dříve, než skončí předchozí. Jinými slovy, výsledky předchozí fáze „vtékají“

jako vstupy do fáze následující.



Obr. 2.1: Vodopádový model softwarového procesu

Model je možno v **různých modifikacích** a **rozšířeních** nalézt ve většině současných přístupů. Tyto modifikace vznikly především kvůli odstranění některých jeho **nedostatků**, mezi které patří:

- **Prodleva** mezi zadáním projektu a vytvořením spustitelného systému je příliš dlouhá.
- **Výsledek závisí** na **úplném a korektním zadaní požadavků** kladených na výsledný produkt.
- **Nelze odhalit výslednou kvalitu produktu** danou splněním všech požadavků, dokud není výsledný softwarový systém hotov.

1.2.2 Modifikace Vodopádového modelu

- **Inkrementální**: **postupné vytváření verzí** softwaru zahrnující postupně širší spektrum funkcí definovaných postupně v průběhu jeho vytváření. V podstatě se jedná o **více menších vodopádů provedených za sebou** tak, aby každý z nich odpovídal nové sadě doplněných požadavků.
- **Spirálový**: 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.

1.3 RUP (Rational Unified Process)

Rational Unified Process (RUP) je **metodika vývoje softwaru** vytvořená společností Rational Software Corporation. Je použitelná pro **jakýkoliv rozsah projektu**, ale díky vysoké rozsáhlosti RUP je vhodné přizpůsobit metodiku specifickým potřebám. RUP je vhodnější spíš pro **rozsáhlejší projekty** a **větší vývojové týmy**, neboť klade důraz na **analýzu** a **design**, **plánování**, **řízení zdrojů** a **dokumentaci**. **Hlavní znaky:**

- softwarový produkt je vyvíjen **iteračním způsobem**,
- jsou **spravovány požadavky** na něj kladené,
- využívá se **již existujících softwarových komponent**,
- model softwarového systému je **vizualizován** (**UML**),
- průběžně je **ověřována kvalita** produktu,
- **změny** systému **jsou řízeny** (každá změna je přijatelná a změny jsou sledovatelné).

1.3.1 Vývoj software iteračním způsobem

V současné době, kdy se předmětem vývoje staly softwarové systémy vysoké úrovně sofistikace, je **nemožné nejprve specifikovat celé zadání**, následně navrhnout jeho řešení, vytvořit softwarový produkt implementující toto zadání, vše otestovat a předat zadavateli k užívání. Jediné možné řešení takovému problému je přístup postavený na **iteracích**, umožňující **postupně upřesňovat cílový produkt** cestou jeho **inkrementálního rozšiřovaní** z původní hrubé formy do výsledné podoby. Softwarový systém je tak **vyvíjen ve verzích**, které lze průběžně ověřovat se zadavatelem a případně jej pozměnit pro následující iteraci. **Každá iterace končí vytvořením spustitelného kódu**.

1.3.2 Správa požadavků

Kvalita výsledného produktu je dána mírou uspokojení požadavků zadavatele. Právě otázka korektní specifikace všech požadavků bývá problémem všech softwarových systémů. Velmi často výsledek i mnohaletého úsilí týmu softwarových inženýrů propadne díky neodstatečné specifikaci zadání. Proces RUP popisuje jak požadavky na softwarový systém doslova vykládat od zadavatelů, jak je organizovat a následně i dokumentovat. Monitorování změn v požadavcích se stává nedílnou součástí vývoje stejně jako správně dokumentované požadavky sloužící ke komunikaci mezi zadavateli a týmem řešitelů.

1.3.3 Vývoj pomocí komponent

Proces RUP poskytuje systematický přístup k definování architektury využívající nové či již existující komponenty. Tyto komponenty jsou vzájemně propojovány v rámci korektně definované architektury buď pro případ od případu (ad-hoc), nebo prostřednictvím komponentní architektury využívající Internet, infrastrukturu CORBA (Common Object Request Broker Architecture), nebo COM (Component Object Model). Již dnes existuje celá řada znovupoužitelných komponent a je zřejmé, že softwarový průmysl věnuje velké úsilí dalšímu vývoji v této oblasti. Vývoj software se tak přesouvá do oblasti skládání produktu z prefabrikovaných komponent.

1.3.4 Vizualizace modelování SW systému

Softwarový proces RUP popisuje jak vizualizovat model softwarového systému s cílem uchopit strukturu a chování výsledné architektury produktu a jeho komponent. Smyslem této vizualizace je skrýt detaily a vytvořit kód užitím jazyka postaveného na grafickém vyjádření stavebních bloků výsledného produktu. Základem pro úspěšné použití principů vizualizace je za průmyslový standard považován jazyk **UML** primárně určený pro účely modelování softwarových systémů.

1.3.5 Ověřování kvality softwarového produktu

Princip ověřování kvality vytvářeného produktu je součástí softwarového procesu, je obsažen ve všech jeho aktivitách, **týká se všech účastníků vývoje** softwarového řešení. Využívají se objektivní měření a kriteria (metriky) kvantifikující kvalitu výsledného produktu. Zajištění kvality tak není považováno za něco co stojí mimo hlavní linii vývoje produktu a není to záležitost zvláštní aktivity realizované speciální skupinou.

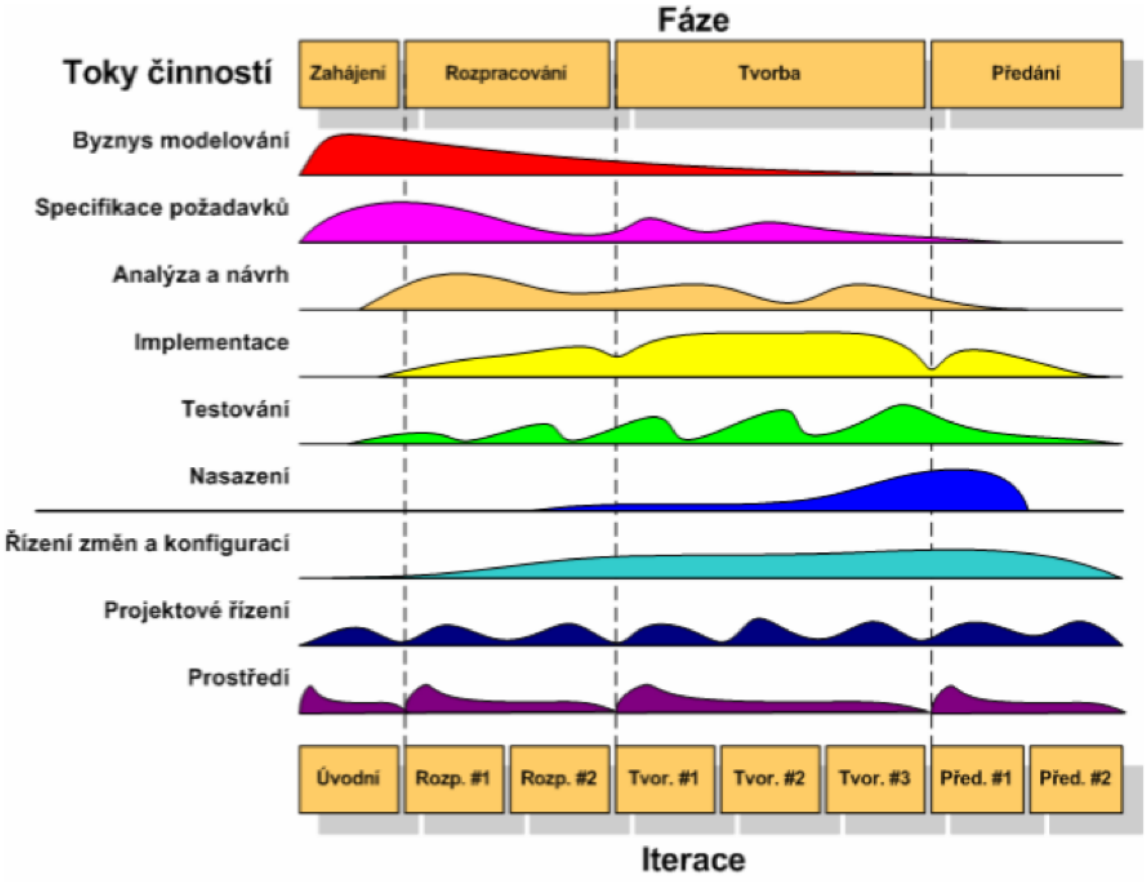
1.3.6 Řízení změn

Řízení změn umožňuje zaručit, že každá změna je přijatelná a všechny změny systému jsou sledovatelné. Důvod, proč je kladen na tuto problematiku takový důraz spočívá v tom, že prostředí ve kterém je softwarový systém vyvíjen podléhá častým a mnohdy i radikálním změnám, které je nutno integrovat do vlastního řešení.

Proces RUP popisuje jak **řídit**, **sledovat** a **monitorovat změny** a umožňuje tak úspěšný iterativní vývoj. Nedílnou součástí této problematiky je vytvoření bezpečného pracovního prostředí poskytující maximalní možnou ochranu před změnami vznikajících v jiném pracovním prostředí.

1.3.7 Schématické vyjádření RUP

Vlastní softwarový proces má svou statickou strukturu ve smyslu toho, z jakých se skladá **toků činností** a **aktivit**. Na druhou stranu má také svou dynamickou stránku popisující jak je softwarový produkt vyvíjen v čase (níže).



Obr.: Schématické vyjádření procesu RUP

V čem se tento přístup liší o dříve zmíněného vodopádového modelu? Základní rozdíl spočívá v tom, že **toky činností probíhají souběžně**, i když z obrázku jasně vyplývá, že objem prací se liší podle fáze rozpracování softwarového systému. Zřejmě, těžiště činností spjatých s byznys modelováním a specifikací požadavků bude v úvodních fázích, zatímco problematika rozmístění softwarových balíků na počítačích propojených sítí (počítačové infrastruktuře) bude záležitostí fází závěrečných. Celý životní cyklus je pak rozložen do čtyř základních fází (zahájení, rozpracování, tvorba a předání), přičemž pro každou z nich je typická realizace několika iterací umožňující postupné detailnější rozpracování produktu.

1.3.8 Cykly, fáze, iterace

Každý **cyklus vede k vytvoření takové verze systému**, kterou lze **předat uživatelům a implementuje jimi specifikované požadavky**. Jak již bylo uvedeno v předchozí kapitole, každý takový vývojový cyklus lze rozdělit do **čtyř** po sobě jdoucích **fází**:

1. **Zahájení**, kde je původní myšlenka rozpracována do **vize koncového produktu** a je definován rámec toho, jak celý systém bude vyvíjen a implementován.

2. **Rozpracování** je fáze věnovaná **podrobné specifikaci požadavků** a **rozpracování architektury** výsledného produktu.

3. **Tvorba** je zaměřena na **kompletní vyhotovení požadovaného díla**. Výsledné programové vybavení je vytvořeno kolem navržené kostry (architektury) softwarového systému.

4. **Předání** je závěrečnou fází, kdy **vytvořený produkt je předán do užívání**. Tato fáze zahrnuje i další aktivity jako je beta **testování**, **zaškolení** apod.

Každá fáze může být dále **rozložena do několika iterací**.

1.3.9 Iterace

Iterace je **úplná vývojová smyčka vedoucí k vytvoření spustitelné verze kódu** reprezentující **podmnožinu** vyvíjeného cílového produktu, a která je **postupně rozšiřována každou iterací** až do výsledné podoby.

1.3.10 Statická struktura procesu

Definuje **KDO** (role), **CO** (artefaty), **JAK** (aktivity a jejich toky) a **KDY** to má vytvořit. Smyslem každého procesu a tedy i softwarového je specifikovat kdo v něm vystupuje, co má vytvořit, jak to má vytvořit a kdy to má vytvořit. Z tohoto pohledu hovoříme o následujících elementech tvořích strukturu každého softwarového procesu:

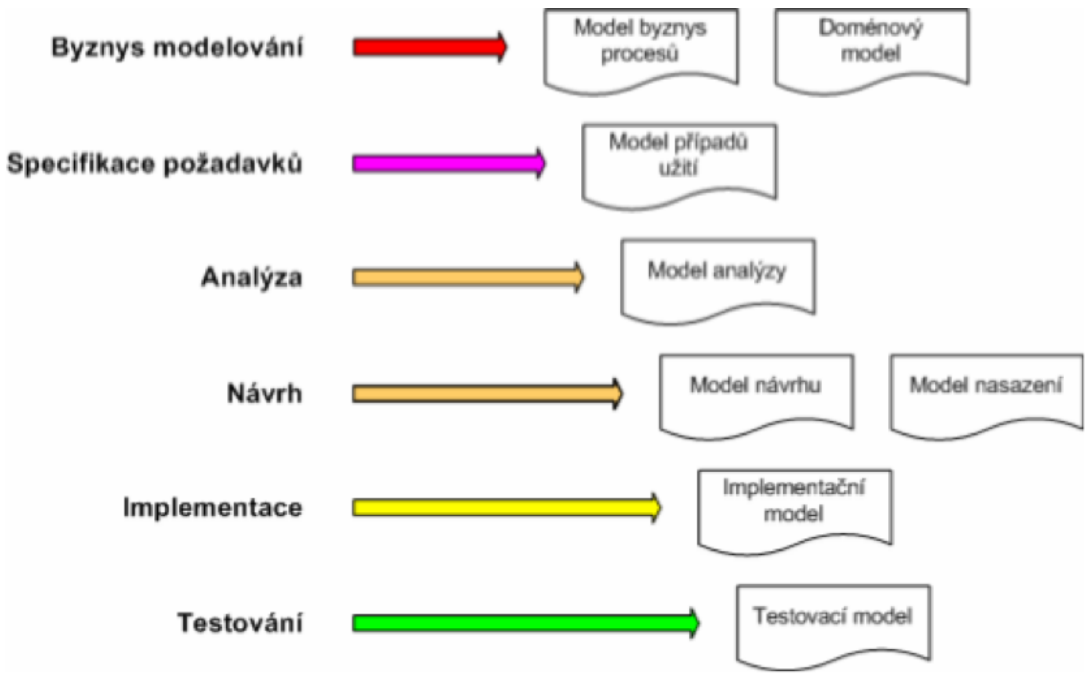
- **Role (pracovníci)** definující chování, kompetence a zodpovědnosti jednotlivých osob (analytik, programátor, projektový manažer apod.) nebo skupiny osob spolupracujících v týmech. Jednotlivé osoby (lidské zdroje) jsou mapovány na požadované role dle toho, jak jsou požadované kompetence slučitelné se schopnostmi těchto osob.
- **Artefakty** reprezentující entity, které jsou v procesu vytvářeny, modifikovány nebo zužitkovány (modely, dokumentace, zdrojové kódy apod.). Základním artefaktem, který je nosným pro vývoj softwarového systému je **model - zjednodušení reality umožňující lépe pochopit vyvíjený systém**.
- **Aktivity (činnosti)** prováděné pracovníky s cílem vytvořit nebo upravit artefakty (kompilace zdrojových kódů, vytvoření návrhu apod.).

- **Toky (workflow)** činností reprezentující posloupnosti aktivit vytvářející požadované produkty (byznys modelování, specifikace požadavků apod.).

1.3.11 Základní a podpůrné toky činností

Vývoj softwarového systému je dán celou řadou aktivit a činností, které jsou uspořádány do toků charakteristických svým účelem. Z tohoto pohledu můžeme hovořit o tzv. základních tocích, jejichž výsledkem je část softwarového produktu (artefakt) a o tocích podpůrných, které nevytváří hodnotu, ale jsou nutné pro realizaci základních toků. Základní toky vytvářející vlastní softwarový produkt jsou následující:

- **Byznys modelování** popisující strukturu a dynamiku podniku či organizace.
- **Specifikace požadavků** definující prostřednictvím specifikace tzv. případů použití softwarového systému jeho funkcionalitu.
- **Analýza a návrh** zaměřené na specifikaci architektury softwarového produktu.
- **Implementace** reprezentující vlastní tvorbu softwaru, testování komponent a jejich integraci.
- **Testování** zaměřené na činnosti spjaté s ověřením správnosti řešení softwaru v celé jeho složitosti.
- **Rozmístění** zabývající se problematikou **konfigurace** výsledného produktu na cílové počítačové infrastruktuře.



Obr. 2.5: Toky činností a modely jimi vytvářené

Podpůrné toky, samozřejmě ty nejdůležitější a spjaté s vlastním vývojem, jsou:

- **Řízení změn a konfigurací** zabývající se problematikou správy jednotlivých verzí vytvářených artefaktů odrážejících vývoj změn požadavků kladených na softwarový systém.
- **Projektové řízení** zahrnující problematiku koordinace pracovníků, zajištění a dodržení rozpočtu, aktivity plánování a kontroly dosažených výsledků. Nedílnou součástí je tzv. **řízení rizik**, tedy identifikace problematických situací a jejich řešení.
- **Prostředí a jeho správa** je tok činností poskytující vývojové organizaci metodiku, nástroje a infrastrukturu podporující vývojový tým.

1.4 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 softwarový proces** a každý projekt je řešen **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.

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

2.1 Specifikace požadavků

Specifikace a analýza požadavků je první fáze vývoje softwaru. **Cílem je definovat požadavky na software a popsat jeho funkcionalitu**. Výsledkem této fáze by měly být dokumenty, které se stanou součásti smlouvy mezi zadavatelem a vývojovým týmem. **Kvalita** výsledného produktu je pak dána **mírou uspokojení požadavků zadavatele**.

V rámci specifikace požadavků je třeba analyzovat procesy u zadavatele, které bude software řešit, nebo s ním nějak jinak souvisí. K popisu těchto procesů dobře poslouží **Diagram případu užití**, **Sekvenční diagram** a **Diagram aktivity**.

2.1.1 Cíle požadavků

- chceme vytvořit a udržovat dohody se zákazníky a dalšími zainteresovanými stranami o tom, co by **systém měl dělat a proč**,
- aby vývojáři lépe pochopili **požadavky na systém**,
- definování **hranic systému**,
- vytvořit základ pro plánování **technického obsahu** interakcí,
- poskytnout základ pro **odhad nákladů** a **času** na **vytvoření systému**,
- definovat **uživatelské rozhraní** pro systém, se zaměřením na potřeby uživatelů.

2.1.2 Aktivity spojené s analýzou požadavků

- **Sběr požadavků** - komunikace se zákazníky a uživateli za účelem získání jejich požadavků na systém.
- **Analýza požadavků** - identifikování nejasných požadavků, nekompletních, nebo protichůdných a následné řešení těchto nesrovnalostí.
- **Zaznamenání požadavků** - dokumentování požadavků v různých formách, jako běžný textový dokument, případy užití (use case), nebo specifikace procesů.

Požadavky je také nutno: **organizovat**, **dokumentovat**, **priorizovat**, **filtrovat** a **sledovat**.

2.1.3 Typy požadavků

- **Funkční požadavky (chování)** - se používají k vyjádření chování systému zadáním jak vstupních tak i výstupních podmínek.
- **Doplňující požadavky (nefunkční)**

- **Použitelnost (Usability)** - se zabývá lidskými faktory, jako je vzhled, snadné používání, atd.
- **Spolehlivost (Reliability)** - četnost a závažnost selhání, obnovitelnost a přesnost.
- **Výkon (Performance)** - se zabývá množstvím transakcí, jako je rychlost, doba odezvy, atd.
- **Podporovatelnost (Supportability)** - řeší, jak těžké je udržet systém a další vlastnosti potřebné k udržení systému po jeho vydání.

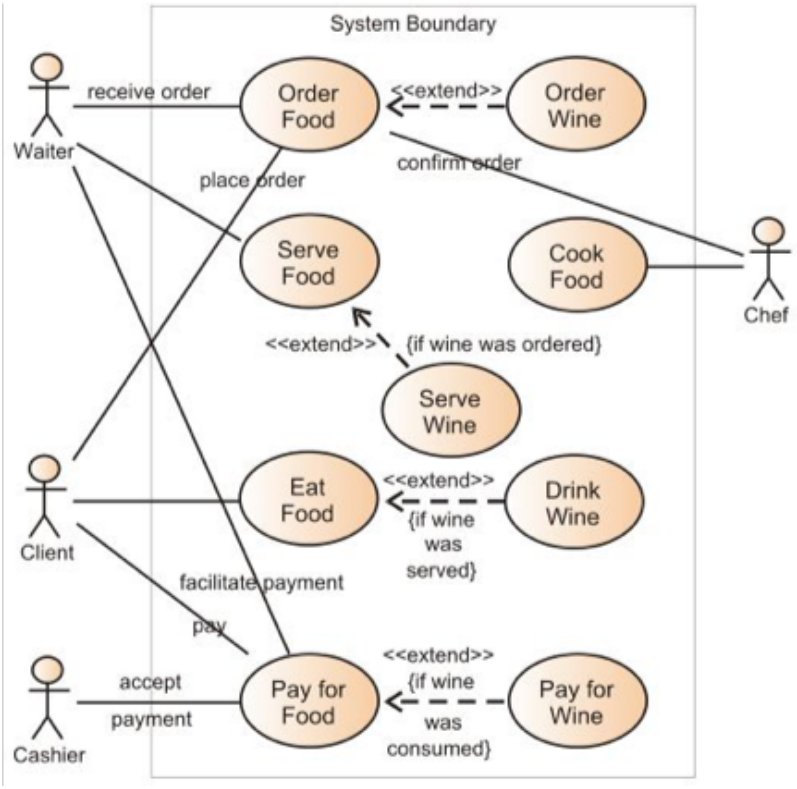
2.2 Diagram případů užití

Případ užití (use case) - je technika pro zdokumentování případného požadavku na nový systém, nebo změny na stávající systém. Každý use case poskytuje jeden nebo více scénářů, které zaznamenávají, jak by systém měl spolupracovat s koncovým uživatelem nebo jiným systémem.

Diagram případu užití popisuje **vztahy** mezi **aktéry** a jednotlivými **případy užití**. Součástí diagramu jsou:

- **Aktéři** - definují **uživatele** či **jiné systémy**, kteří budou vstupovat do interakce s vyvíjeným softwarovým systémem.
- **Relace (vztahy)** - **vazby** a **vztahy** mezi aktéry a případy užití. V diagramu jsou znázorněny **šipkami případně čárami**.
- **Případy užití** - specifikující vzory chování softwarovým systémem. Každý případ užití lze chápat jako **posloupnost vzájemně navazujících transakcí** vykonaných **v dialogu mezi aktérem a vlastním softwarovým systémem**.
- **Scénář** - unikátní sekvence akcí skrz use-case (jedna cesta/instance provedení).

Účelem diagramu případů užití **je definovat co existuje vně vyvíjeného systému** (**aktéři**) a **co má být systémem prováděno (případy užití)**. Vstupem pro sestavení diagramu případů užití je **byznys model**, konkrétně modely podnikových procesů. **Výsledkem** analýzy těchto procesů **je seznam požadovaných funkcí softwarového systému**, které podpoří nebo dokonce nahradí některé z uvedených aktivity jejich softwarovou implementací.



Pro složitější a obsáhlejší diagramy případů užití se zavadí **tři typy vztahů**:

- **Include** – případ užití může **obsahovat** jiný (UC s include se vždy provede před samotným UC).
- **Extend** – případ užití může **rozšiřovat** jiný (UC s extend se může nebo nemusí provést).
- **Generalizace** – případ užití může být **speciálním případem** jiného (dědičnost).

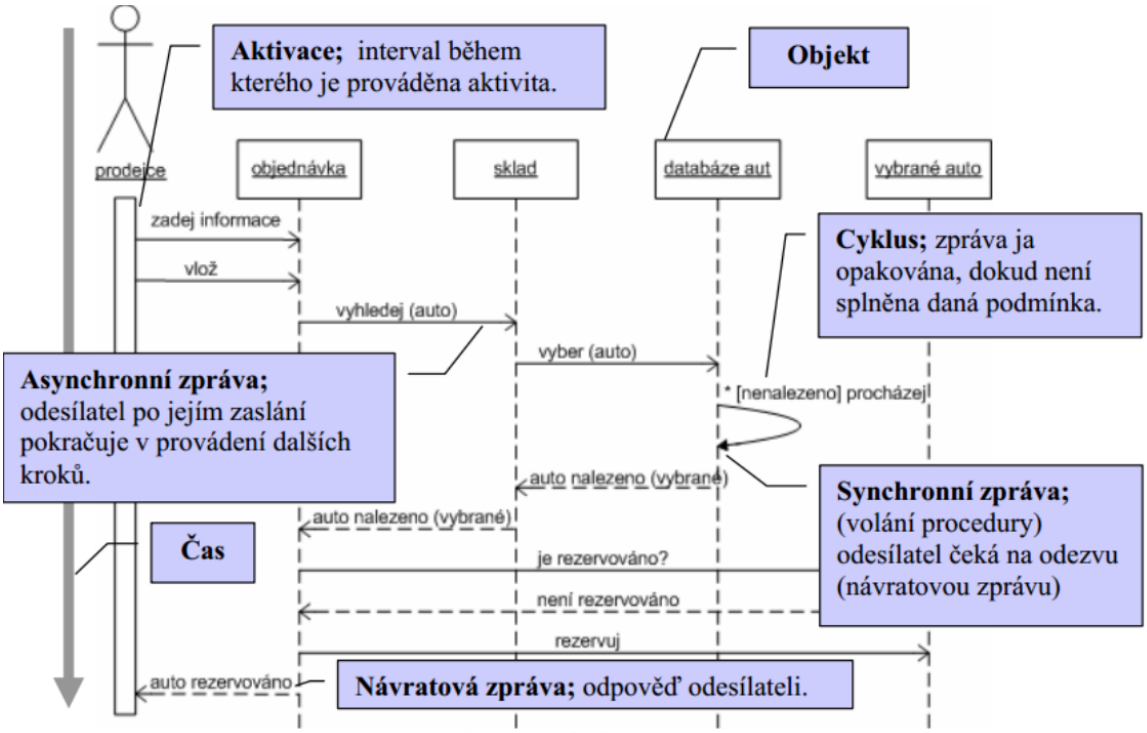
2.2.1 Doporučená forma vytváření use-case

- 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.

2.3 Sekvenční diagram

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

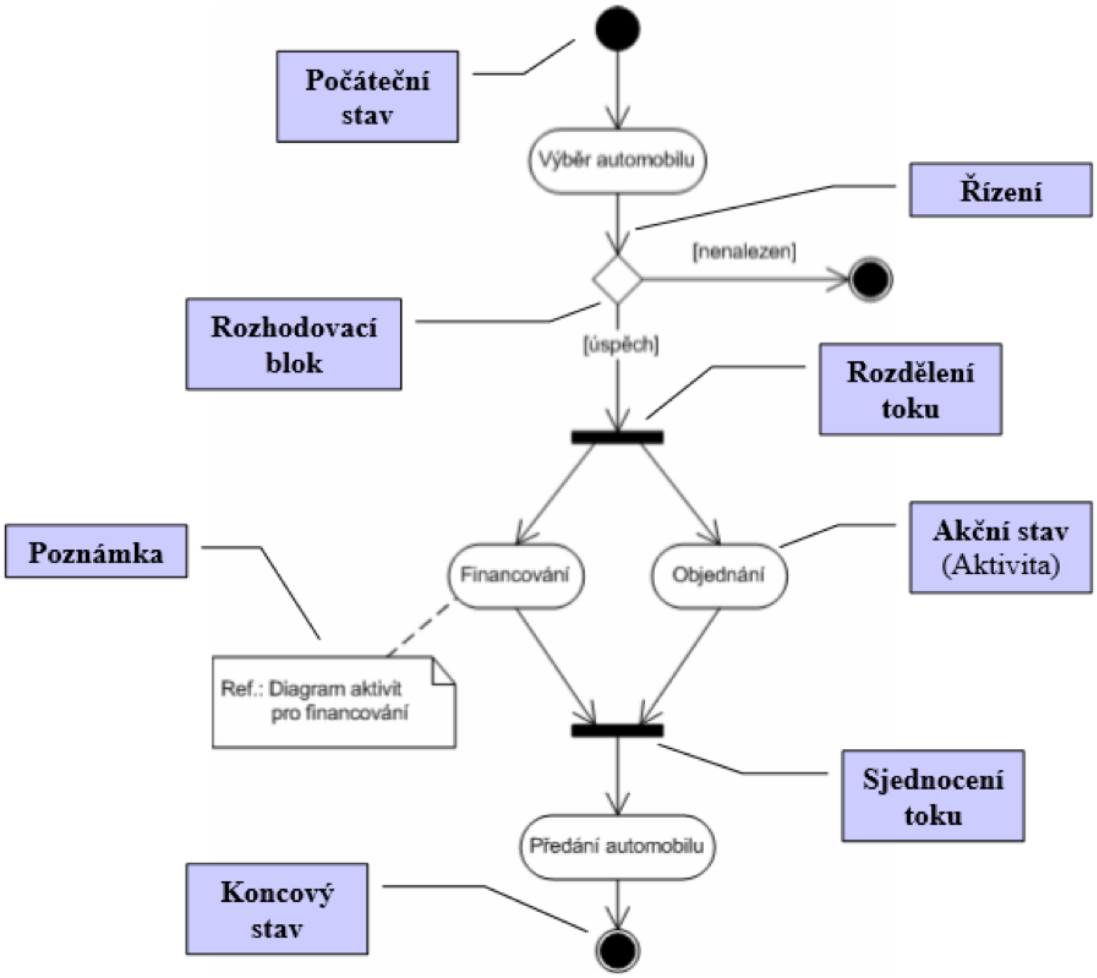
Tento diagram popisuje jaké **zprávy** (**požadavky**) **jsou mezi objekty zasílány z pohledu času**. Diagram je tvořen **objekty uspořádanými do sloupců** a šipky mezi nimi odpovídají vzájemně si zasílaným zprávám. Zprávy mohou být synchronní nebo asynchronní. V případě **synchronních zpráv odesílatel čeká na odpověď** (odezvu) adresáta, v případě **asynchronní zprávy odesílatel nečeká** na odpověď a pokračuje ve vykonávání své činnosti. Souvislé provádění nějaké činnosti se v sekvenčním diagram vyjadřuje svisle orientovaným obdélníkem. Odezvu adresáta lze opět modelovat, v tomto případě tzv. **návratovou zprávou (přerušovaná čára)**. Tok času probíhá ve směru od shora dolů.



Obr. 4.5: Sekvenční diagram

2.4 Diagram aktivit

Diagram aktivit popisuje jednotlivé procesy pomocí aktivit reprezentující jeho (akční) **stavy** a **přechody mezi nimi**.



Obr. 3.1.1: Diagram aktivit popisující prodej automobilu

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

Model návrhu dále **upřesňuje model analýzy ve světle skutečného implementačního prostředí**. Model návrhu tak představuje **druhou fázi** vývoje softwaru. Jde o abstrakci zdrojového kódu, která bude sloužit jako hlavní dokument programátorům v další implementační fázi.

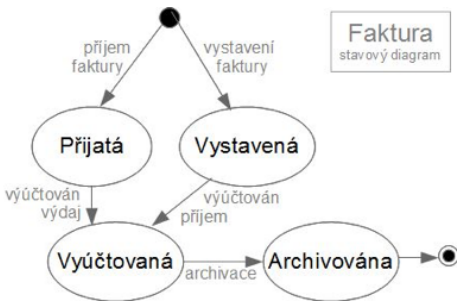
3.1 Návrh a jeho cíle

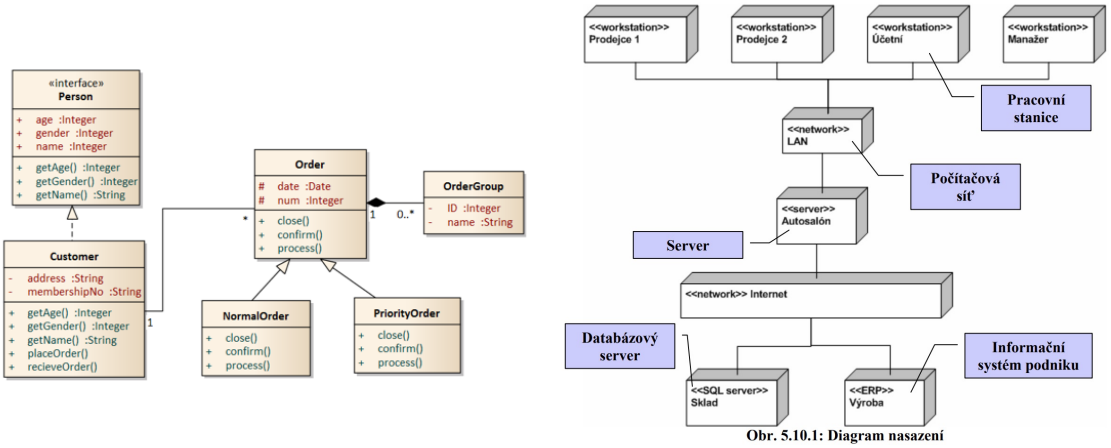
Pojem implementační prostředí v podstatě vyjadřuje **možnost namapovat navržené softwarové komponenty** obsažené v modelu analýzy na architekturu systému určeného k provozu vyvíjené aplikace **s maximálním možným využitím služeb již existujících softwarových komponent**. Postup včlenění implementačního prostředí do vyvíjené aplikace je dán následující posloupností činností:

1. Definice **systémové architektury**.
2. Identifikace **návrhových vzorů** a možnosti znovupoužití tzv. rámcových řešení.
3. Definice softwarových komponent a jejich **znovupoužití**.

V této fázi se uplatňují následující diagramy:

- **Diagram tříd** - specifikuje **množinu tříd, rozhraní a jejich vzájemné vztahy**. Tyto diagramy slouží k vyjádření **statického** pohledu na systém.
- **Sekvenční diagram** - popisuje **interakce mezi objekty** z hlediska jejich **časového** uspořádaní.
- **Diagram spolupráce** - je obdobně jako sekvenční diagram zaměřen na **interakce**, ale z pohledu strukturální organizace objektů. Jinými slovy není primárním aspektem časová posloupnost posílaných zpráv, ale **topologie rozmístění objektů**.
- **Stavový diagram** - dokumentuje **životní cyklus objektu** dané třídy a stavy, ve kterých se může nacházet.
- **Diagram nasazení** - popisuje **konfiguraci** (topologii) **technických prostředků** umožňující běh vlastního softwarového systému (rozmístění HW a SW).





Obr. 5.10.1: Diagram nasazení

3.2 Návrhové vzory - členění

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

- **Creational Patterns (vytvářející)** - určené k řešení problému vytváření instancí tříd cestou delegace této funkce na speciálně k tomuto účelu navržené třídy. Většinou se jedná o dynamická rozhodnutí učiněná za běhu programu.
- **Structural Patterns (strukturální)** - představují skupinu návrhových vzorů zaměřujících se na možnosti uspořádání jednotlivých objektů a jejich 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.

3.2.1 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)** - Zajišťuje, že daná třída má pouze jednu instanci (využívá se privátních konstruktorů a samotná třída ve statické proměnné uchovává danou

instanci).

- **Prototype (Prototyp, Klon)** - Specifikuje druh objektů, které se mají vytvořit použitím prototypového objektu. Nové objekty se vytváří kopírováním tohoto prototypového objektu.

3.2.2 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 (Kompozit, Strom, Složenina)** - Komponuje objekty do stromové struktury a umožňuje klientovi pracovat s jednotlivými i se složenými objekty stejným způsobem.
- **Decorator (Dekorátor)** - Dekorátor se vytváří za účelem změny instancí tříd bez nutnosti vytvoření nových odvozených tříd, jelikož pouze dynamicky připojuje další funkčnosti k objektu. 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.

3.2.3 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)** - Řeší problém, jak se pohybovat mezi prvky, které jsou sekvenčně uspořádány, bez znalosti implementace jednotlivých prvků posloupnosti.
- **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.

4.1 OOP

Objektově-orientované programování (OOP) je metodika vývoje softwaru, která jej odlišuje od procedurálního programování. Paradigma OOP popisují způsob vývoje a zápisu programu a způsob uvažování o problému.

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.

4.1.1 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 jejich nepodstatné vlastnosti.
- Postup řešení je v řadě případů efektivnější než při procedurálním přístupu (ne vždy), kdy se úlohy řeší jako posloupnost příkazů.

4.1.2 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.

4.2 Třída

Třída představuje základní konstrukční prvek OOP. 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.

4.3 Objekt, jeho vlastnosti a vztahy

Jednotlivé prvky modelované reality (jak data, tak související funkčnost) jsou v programu seskupeny do entit, nazývaných objekty. Objekt je **identifikovatelná samostatná entita**

(dána jedinečností a chováním). Objekty si pamatují svůj stav a navenek poskytují operace přístupné jako metody pro volání. **Objekt** je také **instancí třídy**.

Ve vztahu k definovaným případům užití je nutné definovat takové interakce mezi objekty, které povedou ke splnění jejich funkcionality, účelu ke kterému byly navrženy. Jazyk UML poskytuje pro účely zaznamenání těchto vzájemných interakcí tzv. **sekvenční diagram**.

4.4 Rozhraní

Rozhraní entity je **souhrn informací, kterým entita specifikuje, co o ní okolí ví** a **jakým způsobem je možné s ní komunikovat**. Je to množina metod, která může být implementována třídou. Interface **pouze popisuje metody**, jejich vlastní implementace však neobsahuje. Je **pojmenování skupiny externě viditelných operací**. Každá třída však může implementovat libovolný počet rozhraní, do jisté míry tedy rozhraní vícenásobnou dědičnost nahrazují. Implementace rozhraní není na hierarchii tříd nijak vázána a nevzniká z ní vztah dědičnosti.

4.5 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).

4.6 Základní vztahy

Vztahy (relace) mezi třídami **specifikují cestu, jak mohou objekty mezi sebou komunikovat.** Relace složení částí do jednoho celku, má v podstatě dvě možné podoby. Jedná se o tzv. **agregaci**, pro kterou platí, že části mohou být obsaženy i v jiných celcích, jinými slovy řečeno, jsou sdíleny. Nebo se jedná o výhradní vlastnictví částí celkem, pak hovoříme o složení typu **kompozice**. Druhá z uvedených typu složení má jednu důležitou vlastnost z hlediska životního cyklu celku a jeho částí. Existence obou je totiž totožná. **Zánik** celku (**kompozitu**) **vede i k zániku jeho částí** na rozdíl od agregace, kde části mohou přežívat dále jako součástí jiných celků.

- **Asociace** popisuje **skupinu spojení (mezi objekty)** mající společnou strukturu a sémantiku. Vztah mezi asociací a spojením je analogický vztahu mezi třídou a objektem. Jinými slovy řečeno, jedná se tedy o dvousměrné propojení mezi třídami popisující množinu potenciálních spojení mezi instancemi asociovaných tříd stejně jako třída popisuje množinu svých potenciálních objektů.

- **Složení** popisuje **vztah mezi celkem a jeho částmi**, kde některé objekty definují komponenty jejichž složením vzniká celek reprezentovaný jiným objektem.
- **Závislost** reprezentuje slabší formu **vztahu mezi klientem a poskytovatelem služby**.
- **Zobecnění (generalizace)** je taxonomický **vztah mezi obecnějším elementem a jeho více specikovaným elementem**, který je plně konzistentní s prvním z uvedených pouze k jeho specifikaci **přidává další konkretizující informaci**.

4.7 Rysy OOP

- **Skládání** - Objekt může obsahovat jiné objekty.
- **Delegování** - Objekt může využívat služeb jiných objektů tak, že je požádá o provedení operace.
- **Dědičnost** - 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**. Pozná se tak, že několik objektů poskytuje stejné rozhraní, pracuje se s nimi navenek 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, neboť rozhraní třídy je podmnožinou rozhraní podtřídy. U **polymorfismu nepodmíněného dědičností** je dostačující, jestliže se rozhraní (nebo jejich požadované části) u různých tříd shodují, pak jsou vzájemně polymorfní.
- **Zapouzdření** - zaručuje, že objekt **nemůže přímo přistupovat k „vnitřnostem“** jiných objektů, což by mohlo vést k nekonzistenci. Každý objekt navenek poskytuje rozhraní, pomocí kterého (a nijak jinak) se s objektem pracuje.

5 Mapování UML diagramů na zdrojový kód.

Důsledná a přesná specifikace objektů a jejich tříd v etapě návrhu umožňuje **automatické generování zdrojových kódů**. K mapování UML diagramů na kód dochází v první části implementační fáze vývoje.

Z **diagramu tříd** lze vygenerovat jednotlivá rozhraní, třídy s proměnnými a metodami (bez implementace) - `http://www.milosnemec.cz/clanek.php?id=199`. Úkolem samotné implementace je pak dopsat těla metod, jejichž chování může být popsáno v **diagramu aktivit**.

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 adresáři specifikovaném pomocí package

Cílem implementace je **doplnit** navrženou architekturu (kostru) aplikace **o programový kód** a vytvořit tak kompletní systém.

Implementační model specifikuje, jak jsou jednotlivé elementy (objekty a třídy) implementovány ve smyslu softwarových komponent.

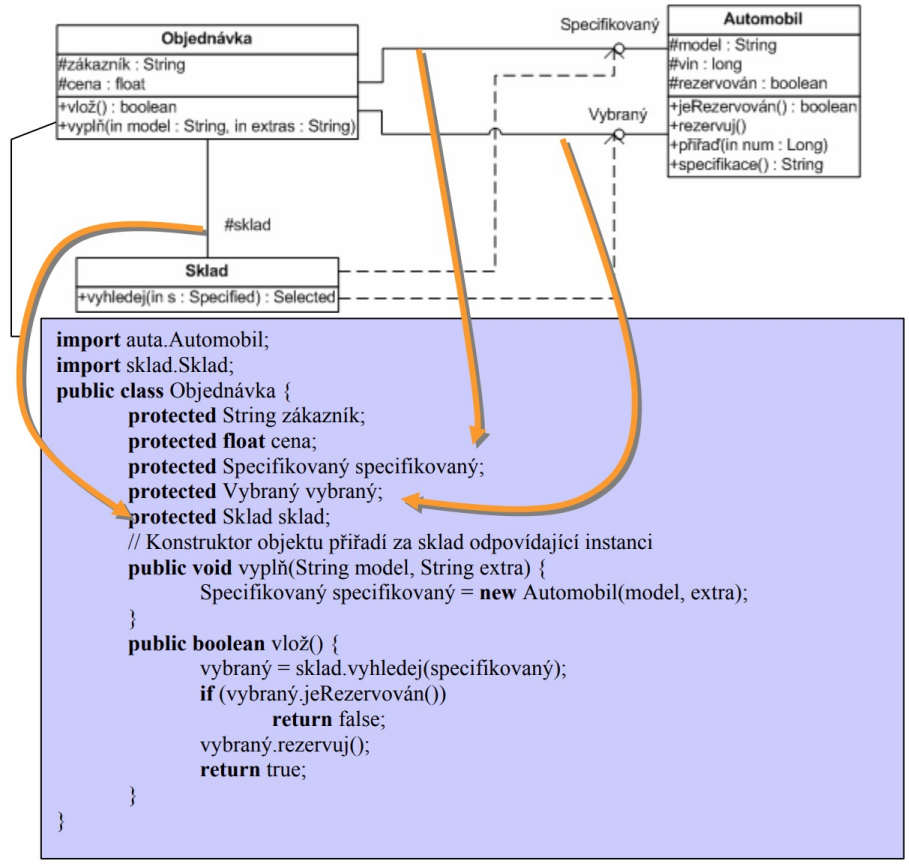
Softwarová komponenta je definována jako **fyzicky existující** a **zaměnitelná část** systému, která vyhovuje požadované množině rozhraní a poskytuje jejich realizaci. Typy softwarových komponent dělíme na:

- **Zdrojové kódy** - části systému zapsané v programovacím jazyce.
- **Binární a spustitelné kódy** - přeložené do strojové kódu procesoru.
- **Ostatní části** - databázové tabulky, dokumenty apod.

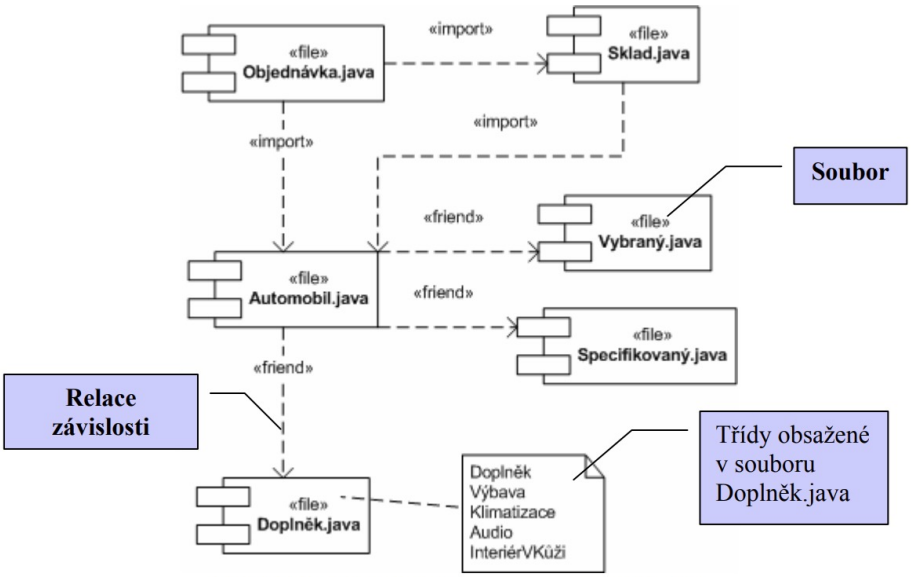
Jestliže jsme ve fázi analýzy a návrhu pracovali pouze s abstrakcemi dokumentovanými v podobě jednotlivých diagramů, pak v **průběhu implementace dochází k jejich fyzické realizaci**. Implementační model se tedy také zaměřuje na specifikaci toho, jak budou tyto komponenty fyzicky organizovány podle implementačního prostředí, pro splnění těchto cílů lze využít následující diagramy:

- **Diagram komponent** - ilustruje organizaci a závislosti mezi softwarovými komponentami.

- **Diagram nasazení** - upřesňuje nejen konfiguraci technických prostředků, ale především rozmístění implementovaných softwarových komponent na těchto prostředcích.



Obr. 6.2.2: Zdrojový kód: Objednávka.java



Obr. 6.2.1: Diagram komponent: zdrojový kód

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

Většina moderních jazyků používá zejména **automatické správy paměti**: BASIC, C#, Dylan, Erlang, Haskell, Java, JavaScript, Lisp, ML, Modula-3, Perl, PHP, Prolog, Python, Ruby, Scheme, Smalltalk, atd.

Výhody Automatické správy paměti

- 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**.

Nevýhody Automatické správy paměti

- 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**.

6.1 C

Správa paměti je v **rukou programátora**. V jazyce C se setkáme se správou paměti pouze pro **dynamicky alokovanou paměť**, tu spravujeme pomocí **4 funkcí** (malloc, realloc, calloc, free). **Staticky alokovaná** paměť (pole, datové typy) je známá v době kompilace a je tedy v paměti pořád. Dynamicky alokovanou paměť musíme spravovat sami.

6.2 C++

C++ obsahuje volitelnou podporu pro Garbage Collectory třetích stran, nicméně standard jako takový nepředepisuje jejich použití. Správa paměti je tedy víceméně v **rukou programátora**. K správě paměti tedy využíváme operátorů **new** (přiděluje) a **delete** (ruší). Zároveň specifikace třídy implementuje **destruktor**, kde je programátor povinen uvolnit veškerou alokovanou paměť v rámci dané třídy (objektu). Tento destruktor se volá při zavolání operátoru **delete** na daný objekt.

6.3 C#

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

6.4 Java

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**. **Java** GC používá **5 algoritmů** (JDK 7, Serial GC, Paralel GC, Paralel Old GC, Garbage First GC, Mark & Sweep GC).

6.5 Garbage collector

Garbage collecting je **způsob automatické správy paměti**. Uvolňování paměti šetří čas při vývoji a osvobozuje programátora od uvolňování objektů, které již dále nejsou zapotřebí, což ho většinou stojí značné úsilí. Je to vlastně **pomůcka pro stabilnější program**, protože zabraňuje některým třídám provozních chyb. Například zabraňuje **chybám ukazatelů**, které ukazují na již nepoužívaný objekt, nebo který je již zrušen a tato paměť se dále k ničemu nevyužívá. **Základní princip garbage collectingu:**

1. Vyhledají se v programu takové datové objekty, které nebudou v budoucnu použity.
2. Vrácení zdrojů, kde se vyskytovaly nalezené objekty.

6.6 Mark & Sweep (Typ GC)

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.

Tato metoda má několik nevýhod. Největší je, že při garbage collectionu je **přerušen běh programu**. To znamená,že programy **pravidelně zamrznou**, takže je nemožné pracovat s aplikacemi používající reálný čas.

6.7 Reference counting (Typ GC)

Ke každému objektu je přiřazen **čítač referencí**. Když je objekt vytvořen, jeho čítači je nastavena **hodnota 1**. V okamžiku, kdy si nějaký jiný objekt nebo kořen programu (kořeny jsou hledány v programových registrech, v lokálních proměnných uložených v zásobnících jednotlivých vláken a ve statických proměnných) uloží referenci na tento objekt, hodnota

čítače je **zvětšena o 1**. Ve chvíli, kdy je reference mimo rozsah platnosti (např. po opuštění funkce, která si referenci uložila), nebo když je referenci přiřazena nová hodnota, čítač je **snížen o 1**. Jestliže je hodnota čítače některého objektu nulová, může být tento objekt uvolněn z paměti.

6.8 Generační algoritmus (Typ GC)

Staví na **dvou základních principech**:

- Mnoho objektů se stane **odpadem** krátce **po** svém **vzniku**.
- Jen malé procento **referencí** ve „starších“ objektech **ukazuje na objekty mladší**.

Rozděluje si paměť programu do několika částí, tzv. „**generací**“. Objekty jsou vytvářeny ve spodní (nejmladší) generaci a po splnění určité podmínky, obvykle stáří), jsou přeřazeny do starší generace. Pro každou generaci může být **úklid** prováděn v rozdílných **časových intervalech** (nejkratší intervaly obvykle budou platit pro nejmladší generaci) a dokonce pro rozdílné generace mohou být použity **rozdílné algoritmy úklidu**. V okamžiku, kdy se prostor pro spodní generaci zaplní, všechny dosažitelné objekty v nejmladší generaci jsou zkopírovány do starší generace. I tak bude množství kopírovaných objektů pouze zlomkem z celkového množství mladších objektů, jelikož většina z nich se již stala odpadem.

6.9 Nevýhody GC

- Garbage collector potřebuje ke své práci **procesorový čas**, aby mohl rozhodovat o tom, jestli je objekt v paměti „mrtvý“, nebo „živý“.
- Některé garbage collectory mohou způsobit i dosti znatelné **pauzy**, což je vážný problém pro systémy běžící v reálném čase.
- O stavu objektů musí mít garbage collector uloženou informaci. Tyto informace vyžadují určitou **paměť navíc**.
- Některé jazyky s garbage collectorem neumožňují programátorovi **znovupoužití paměti**, i když ví, že paměť už nebude použita. To vede k **nárůstu alokace paměti**.

6.10 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.

6.10.1 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.

Podobný software je často označován termíny jako **virtualizace** a **virtuální servery**. Hostitelský software, který poskytuje tuto schopnost je často označovaný jako **hypervisor** nebo **virtuální strojový monitor** (virtual machine monitor). Softwarové virtualizace mohou být prováděny **ve třech hlavních úrovních**:

- **Emulace**, plná systémová simulace nebo „plná virtualizace s dynamickým přestavením (recompilation)“ — virtuální stroj simuluje kompletní hardware, dovolující provoz nemodifikovaného OS na úplně jiném procesoru.
- **Paravirtualizace** — virtuální stroj nesimuluje hardware, ale místo toho nabídne **speciální rozhraní API**, které vyžaduje modifikace OS.
- **Nativní virtualizace** a „**plná virtualizace**“ — virtuální stroj jen částečně simuluje dost hardwaru, aby mohl nemodifikovaný OS běžet samostatně, ale hostitelský OS musí být určený pro stejný druh procesoru. Pojem nativní virtualizace se někdy používá ke zdůraznění, že je **využita hardwarová podpora pro virtualizaci** (tzv. virtualizační technologie).

6.10.2 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** (**JVM**) od firmy Sun Microsystem.

6.10.3 Virtuální prostředí

Virtuální prostředí je jiný **druh virtuálního stroje**. Ve skutečnosti, to je **virtualizované prostředí pro běh programů na úrovni uživatele** (tj. ne jádra operačního systému a ovladače, ale aplikace). Virtuální prostředí jsou vytvořena použitím softwaru zavádějícího virtualizaci na úrovni operačního systému.

6.11 Podpora paralelního zpracování

Paralelně programovaný software využívá možnost rozdělení jednoho velkého výpočetního problému na několik menších problémů, které jsou řešeny „**současně**". Prvky sloužící

k paralelnímu zpracování výpočtu mohou být různé. Jedná se například o jeden **počítač s více procesory**, **několik počítačů** v síti, **specializovaný hardware** nebo **kombinaci** těchto prvků.

Vlákno označuje v informatice **odlehčený proces**, pomocí něhož se **snižuje režie** operačního systému při změně kontextu, které je nutné pro zajištění **multitaskingu** nebo při masivně **paralelních výpočtech**. Zatímco běžné procesy jsou navzájem striktně odděleny, **sdílí** vlákna nejen **společný paměťový prostor**, ale i **další struktury**.

Operační systém, který vlákna nepodporuje, má technicky jedno vlákno na každý proces, zatímco při podpoře vláken je možné v rámci jediného procesu vytvořit mnoho vláken. Vlákna usnadňují díky sdílené paměti vzájemnou komunikaci, což však přináší další komplikace v podobě **souběhu** (race condition).

6.11.1 Shrnutí

- Vlákno - **Thread**.
- Vlákno je **samostatně prováděný výpočetní tok**.
- Vlákna běží v **rámci procesu**.
- Vlákna jednoho procesu běží v rámci stejného prostoru paměti. **Sdílí jeho prostředky**.
- Každé vlákno má vyhrazený prostor pro specifické proměnné (runtime prostředí).

6.12 Semafor

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.

6.13 Vlákna v operačním systému

Vlákna **běží v rámci výpočetního toku procesu.** S ohledem na realizaci se mohou nacházet:

- V **uživatelském prostoru procesu** - realizace vláken je na úrovni knihovních funkcí. Vlákna nevyžadují zvláštní podporu OS, jsou rozvrhována uživatelským knihovním rozvrhovačem. **Nevyužívají více procesorů.**

- V **prostoru jádra OS** - tvoří entitu OS a jsou také rozvrhována systémovým rozvrhovačem. Mohou **paralelně běžet na více procesorech**.

6.14 Kdy použít vlákna?

Vlákna je výhodné použít, pokud aplikace splňuje některé následující kritérium:

- Je **složena z nezávislých úloh**.
- Může být **blokována** po dlouho dobu.
- Obsahuje **výpočetně náročnou** část.
- Musí reagovat na asynchronní události.
- Obsahuje úlohy s nižší nebo vyšší prioritou než zbytek aplikace.

6.15 Typické aplikace

- **Servery** - obsluhují více klientů najednou. Obsluha typicky znamená **přístup k několika sdíleným zdrojům** a hodně vstupně výstupních operací (I/O).
- **Výpočetní aplikace** - na **víceprocesorovém systému** lze výpočet urychlit rozdělením úlohu na více procesorů.
- **Aplikace reálného času** - lze využít specifických rozvrhovačů. Více vláknová aplikace je výkonnější než složité asynchronní programování, neboť vlákno čeká na příslušnou událost namísto přerušování vykonávání kódu a přepínání kontextu.

6.16 Typické aplikace

Modely řeší způsob **vytváření** a **rozdělování práce** mezi vlákna:

- **Boss/Worker** - hlavní vlákno, řídí rozdělení úlohy jiným vláknům.
- **Peer** - vlákna běží paralelně bez specifického vedoucího.
- **Pipeline** - zpracování dat sekvencí operací. Předpokládá dlouhý vstupní proud dat.

6.17 Mechanismy synchronizace

Stejné principy jako synchronizace procesů. Základními primitivy jsou:

- **Mutex** - zámek kritické sekce.
- **Podmíněná proměnná** (condition variable) synchronizace hodnotou proměnné. Čekání vlákna na probuzení od jiného vlákna.

6.17.1 Mutex

Mutex = mutual exclusion, neboli vzájemné vyloučení je algoritmus používaný v programování jako **synchronizační prostředek**. Zabraňuje tomu, aby byly současně vykonávány dva (nebo více) kódy nad stejným sdíleným prostředkem. Základní operace:

- **Lock** - uzamknutí mutexu (přiřazení mutexu vláknu). Pokud nelze mutex získat, vlákno přechází do blokovaného režimu a čeká na uvolnění zámku.
- **Unlock** - uvolnění zámku. Pokud jiná vlákna čekají na uvolnění, je vybráno jedno vlákno, které mutex získá.
- **Rozšířené metody:**
  - **Rekursivní** - vícenásobné zamykání stejným vláknem.
  - **Try** - okamžitý návrat pokud není možné mutex získat.
  - **Timed** - pokus o získání zámku s omezenou dobou čekání.

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

Správné programování je situace, kdy vše půjde podle plánu a nikdy nedojde k nějaké nepředvídané situaci. Pod takovouto situací si můžeme představit **chyby** programu **způsobené programátorem** – **dělení nulou**, sáhnutí **mimo rozsah pole**, pokus o v**olání metody na nullové referenci** a podobně. Druhou skupinou pak mohou být neplatné **uživatelské vstupy** – pokud uživatel předá do kolonky věk řetězec, pokusí se uložit soubor někam, kam nemá přístup a jiné. Poslední skupinou jsou chyby mimo kontrolu a moc programátora – **vyčerpání paměti**, **zásah operačního systému** atd.

Při všech těchto situacích (a mnohých dalších) dojde k vyvolání výjimky. **Výjimky** (exceptions) dovolují tvorbu programů, které mohou reagovat na různé **chybové stavy vzniklé za běhu** Cílem výjimek je učinit programy **robustnější** tím, že jsou rozšířeny o schopnosti správy chyb vzniklých na úrovni aplikace nebo operačního systému.

7.1 C#

Výjimky v jazyce C#:

- Jsou **objektově orientované**.
- Oddělují zpracování chyb od výkonného kódu.
- Nejsou hlídané (jsou **unchecked**).
- Měly by být dokumentovány.

Každá výjimka v jazyku C# je:

- **Objekt**.
- Odvozena z třídy **Exception** nebo z některé z je-jich následníků.
- Obsahuje informace o:
  - Svém původu.
  - Důvodu vzniku.

Pomocí zachycení výjimky je možné vzniklý (chybový) stav **zpracovat (obsloužit)**. Jestliže dojde ke vzniku výjimky, která není nijak obsloužena, pak dochází k **ukončení programu** s odpovídající **běhovou chybou**. Mechanismus výjimek je v jazyce C# založen na klíčových slovech: `try`, `catch`, `finally`. Vlastní vyhození výjimky lze pomocí klíčového slova `throw`.

7.1.1 try

- Vymezuje z**ačátek chráněného bloku** kódu.
- Jestliže nějaká operace v tomto bloku způsobí výjimku, pak je okamžitě **řízení předáno do bloku**, který je umístěný za klíčovým slovem **catch**.

7.1.2 catch

- Slouží k zápisu bloku, který se provede, jestliže ve sdruženém chráněném bloku dojde ke vzniku výjimky.
- Po bloku try může následovat **více bloků catch**, z nichž každý slouží pro **ošetření specifického typu výjimky**.
- **Zápis:** `catch (typ_výjimky err){ ... }`
- Proměnná `err` obsahuje **objekt** vyjadřující podrobné údaje o vzniklé výjimce (např. vlastnost **Message** obsahuje textový popis chyby, která způsobila výjimku).

7.1.3 finally

- Udává **volitelný blok** používaný většinou **k uvolnění zdrojů** alokovaných v bloku try.
- Blok `finally` se vykoná vždy po opuštění try/catch.

7.1.4 Třída Exception

- Definovaná ve **jmenném prostoru** `System`.
- Reprezentuje chyby, ke kterým dochází v **době běhu programu**.
- Slouží jako **výchozí třída** pro všechny další výjimky.

7.2 Java

Výjimky samotné jsou objekty, které dědí ze speciální hierarchie, jejímž kořenem je třída **Throwable**. Throwable dále rozšiřují třídy **Error** a **Exception**.

7.2.1 Error

Výjimky typu Error bychom **neměli v žádném případě ošetřovat** - značí totiž **kritickou chybu** (nedostatech zdrojů pro práci virtuálního stroje, přetečení zásobníku, nenalezení potřebné třídy při classloadingu atp.) – a většině případů **je ani ošetřit nemůžeme**.

7.2.2 Runtime exception

Mezi výjimky dědící z Exception patří také podtřída RuntimeException (dokumentace). 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 **obvyklé chyby**, které **způsobil sám programátor** (neplatný index pole, volání nad nullovým ukazatelem...).

Zvláštní vlastností těchto výjimek je, že **nemusíme deklarovat v hlavičce metody možnost jejich vyvolání** (klíčové slovo throws). Tím je usnadněno vybublání chyby skrz program a jeho případné ukončení. Tyto výjimky označujeme jako **nekontrolované (unchecked exceptions)**.

7.2.3 Exception

Zbylé výjimky dědící přímo z Exception značí ty situace, které by se sice neměly stávat, ale na které **jsme schopni adekvátně zareagovat**. Pokud uživatel zadá neplatný adresář, tak mu vyhubujeme a řekneme, že to má zkusit ještě jednou. Obdobně jsme schopni vyřešit situaci, kdy uživatelské rozhraní nenalezne knihovny potřebné pro uživatelský motiv (look and feel) na daném systému. V tomto případě můžeme přejít k výchozímu nastavení uživatelského rozhraní. (Klasické chyby které normálně ošetřujeme).

Protože se jedná, jak jsme si již řekli, o chyby, ze kterých se program může snadno zotavit, tak je **vždy musíme uvést v hlavičce metody, jež je může vyvolat**. To učiníme pomocí klíčového slova **throws** a seznamu jmen tříd vyvolávaných výjimek. Ve volající metodě se pak musíme rozhodnout, zda-li výjimky ošetříme, nebo necháme vybublat dál (opět je uvedeme v hlavičce). Tyto výjimky označujeme jako **kontrolované (checked exceptions)**.

7.2.4 Vyvolávání výjimek

Výjimku můžeme vyvolat v našem kódu prostřenictvím příkazu **throw** následovaným objektem výjimky (`throw new NumberFormatException()`).

7.2.5 Try-catch-finally

Každou výjimku bez ohledu na její typ můžeme zpracovat. K tomu používáme bloky try, catch a finally. V bloku try uvedeme sekci, která je krirická a vyvolává výjimku. Až v **mnoha blocích catch** můžeme postupně zpracovávat případné výjimky dle jejich typu. Volitelný blok **finally** obsahuje kód, který **se vyvolá vždy**, bez ohledu na to, k čemu dojde v bloku try (dokonce se vyvolá i případě, že dojde k opuštění metody prostřednictvím příkazu return).

7.3 Datové proudy

Proudy **streamy** můžeme dělit dle jejich směru na **vstupní** (přenášejí data do aplikace) a na **výstupní** (přenášejí data z aplikace). Proudy také rozlišujeme na **binární** a **znakové**. Jak již názvy napovídají, tak zatímco binární proudy využijeme pro libovolná binární data (tj. data vkládáme je po bajtech), tak znakové proudy jsou určeny pouze pro text (znaky).

Proudy jsou základní cestou jak pracovat s daty, náhodným i sekvenčním přístupem. Mezi nejjednodušší stream patří výpis na obrazovku `System.out.print();` `console.Writeln();`.

7.3.1 Streamy v .NET

- **Textové**: `StreamWritterSreamReader`
- **Binární**: `BinaryWriter`, `BinaryReader`
- **Další**: `MemoryStream`, `GZipStream`, `Bufferedstream`

7.3.2 Streamy v Javě

V Javě všechny vstupní streamy dědí z abstraktní třídy **InputStream** a všechny výstupní z **OutputStream** a obě tyto abstraktní třídy ze třídy `Object`.

- **Textové**: `FileWriter`, `FileReader` (Tyto streamy jsou obaly pro bytové streamy, tzn. přijímají jako argument objekt `FileInputStream` nebo `FileOutputStream`)
- **Binární**: `FileInputStream`, `FileOutputStream`
- **Další**: `ObjectInputStream`, `ObjectOutputStream`

7.3.3 Binární

Binární proudy umožňují přenést **libovolná data**. Základní operací definovanou v InputStreamu je metoda **read**, pomocí které můžeme z proudu přečíst jeden byte. Analogicky výstupní proud definuje metodu **write**. **Čtení a zápis po jednotlivých bytech 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.

Třídy **BufferedInputStream** a **BufferedOutputStream** (**v Javě**) za nás tento nedostatek řeší. Tyto proudy obsahují **pole, které slouží jako vyrovnávací paměť**. V případě čtení z disku bufferovaný proud načte celý blok dat a uloží jej do pole, ze kterého data dále posílá našemu programu. V okamžiku, kdy se pole vyprázdní, učiní dotaz na další blok. Tímto způsobem dochází k eliminaci velkého množství zbytečných a drahých volání. (V .NET existuje třída **BufferedStream**, nicméně .NET implementuje buffering do určité míry i u jiných streamů, nesetkáme se tedy s jeho použitím tak často.)

7.3.4 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).

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

8.1 Jazyk UML

UML je jazyk umožňující specifikaci, vizualizaci, konstrukci a dokumentaci artefaktů softwarového systému. V průběhu let se UML stal **standardizovaným jazykem** určeným pro vytvoření výkresové dokumentace (softwarového) systému. K vytváření jednotlivých modelů systému jazyk UML poskytuje **celou řadu diagramů** umožňujících **postihnout různé aspekty systému**. Jedná se celkem o čtyři základní náhledy a k nim přiřazené diagramy:

1. **Funkční náhled**
   (a) **Diagram případů užití** - popisující vztahy mezi aktéry a jednotlivými případy použití.
2. **Logický náhled**
   (a) **Diagram tříd** - specifikující množinu tříd, rozhraní a jejich vzájemné vztahy. Tyto diagramy slouží k vyjádření statického pohledu na systém.
   (b) **Objektový diagram**
3. **Dynamický náhled popisující chování**
   (a) **Stavový diagram** - dokumentující životní cyklus objektu dané třídy z hlediska jeho stavů, přechodů mezi těmito stavy a událostmi, které tyto přechody uskutečňují.
   (b) **Diagram aktivit** - popisující podnikový proces pomocí jeho stavů reprezentovaných vykonáváním aktivit a pomocí přechodů mezi těmito stavy způsobených ukončením těchto aktivit. Účelem diagramu aktivit je blíže popsat tok činností daný vnitřním mechanismem jejich provádění.
   (c) **Interakční diagramy**
      i. **Sekvenční diagramy** - popisující interkace mezi objekty z hlediska jejich časového uspořádaní.
      ii. **Diagramy spolupráce** - je obdobně jako předchozí sekvenční diagram zaměřen na interkace, ale z pohledu strukturální organizace objektů. Jinými slovy není primárním aspektem časová posloupnost posílaných zpráv, ale **topologie rozmístění objektů**.
4. **Implementační náhled**
   (a) **Diagram komponent** - ilustrující organizaci a závislosti mezi softwarovými komponentami.

(b) **Diagram nasazení** - upřesněný nejen ve smyslu konfigurace technických prostředků, ale především z hlediska rozmístění implementovaných softwarových komponent na těchto prostředcích.

8.2 Diagramy a jejich použití v rámci fází vývoje

- **Specifikace požadavků**: Diagram případů užití, Sekvenční diagramy, Diagram aktivit.
- **Návrh**: Diagram tříd, Objektový diagram, Stavový diagram, Sekvenční diagramy.
- **Implementace**: Diagramy spolupráce, Diagram komponent, Diagram nasazení.