Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta, katedra informatiky

Obor: Výpočetní technika a informatika

UML A VÝVOJ APLIKACÍ S VYUŽITÍM MODELOVÁNÍ

Bakalářská práce

Autor: Jiří Langer

Vedoucí práce: RNDr. Jaroslav Icha

České Budějovice, 2007

Anotace

V současné době představuje jazyk UML praktický standard, který se využívá při vývoji aplikací ve fázi analýzy a návrhu. Současná verze 2.0 nabízí celkem k využití 13 různých typů diagramů, které pokrývají různé fáze vývoje aplikací. Cílem práce je shrnout stručně vývoj jazyka UML a prezentovat stávající stav a ukázat na možnosti využití určité podmnožiny tohoto jazyka v úvodních kurzech programování pro studenty bakalářského i učitelského studia. V této souvislosti bude třeba se zaměřit na nalezení vhodného softwarového nástroje (pro potřeby vzdělávání), který by bylo možné užít v úvodním kurzu či případně ve výuce specielních předmětů zaměřených na výuku modelování.

Pro výuku v úvodním kurzu připraví autor bakalářské práce konkrétní návrhy projektů a jejich řešení s využitím UML. Inspirací k těmto projektům může být učebnice "*Objects First With Java*" a rovněž zdroje na webu.

Těžiště práce tedy spatřuji v aplikaci jazyka UML v oblasti přípravy studentů bakalářského i učitelského studia tak, aby absolventi byli připraveni na využívání aktuálních technologií jak v praxi vývojářské, tak i v oblasti vzdělávání.

Abstract

The UML language currently presents functional norm which is used during application development at analysis and design stage. Actual version 2.0 offers total of 13 different diagram types, which cover miscellaneous phases of application development. The goal of this dissertation work is to briefly summarize progress of the UML language, present current state and show the potential usage of certain subset of this language in opening courses of programming for both bachelor students and students of pedagogy. In this connection will be necessary to focus on finding suitable software tool (for educational needs), which would be possible to use in the opening course or eventually in tuition of special modelling focused courses.

The author of dissertation work will prepare concrete project propositions and resolutions for tuition in opening course with the usage of the UML. The inspiration for these projects may be the “Objects First with Java” book and also sources on the web.

I see the focal point of the work in use of the UML language to prepare bachelor students and students of pedagogy for derive benefits from actual technologies in developer and educational profession.

Poděkování

Děkuji RNDr. Jaroslavu Ichovi za odborné vedení a cenné připomínky během tvorby mé práce.

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně, a že jsem veškeré použité zdroje uvedl v Seznamu použitých zdrojů.

....

Jiří Langer

Obsah

1 ÚVOD 11

1.1 CHARAKTER PRÁCE 11
1.2 POUŽITÝ NÁSTROJ 11

2 VIZUÁLNÍ MODELOVÁNÍ A JAZYK UML 12

2.1 VIZUÁLNÍ MODELOVÁNÍ 12
2.2 VYJÁDŘENÍ MODELŮ 13
2.3 DIAGRAMY 13
2.4 ZROZENÍ UML 13
2.5 PŘEHLED HISTORIE VERZÍ UML 14
2.6 CO JE TO UML? 15
2.7 UML 2.0 16
2.8 MOŽNOSTI PŘIZPŮSOBENÍ UML 17
2.9 SPECIFIKACE UML 18
2.10 DIAGRAMY UML 18
2.11 ZPŮSOBY POUŽITÍ UML 19
2.11.1 DIAGRAMY 19
2.11.1.1 Kreslení konceptu 19
2.11.1.2 Kreslení detailních návrhů 19
2.11.1.3 UML jako programovací jazyk 20
2.11.2 METAMODEL 20
2.12 PROČ PRÁVĚ UML? 20
2.13 CASE NÁSTROJE 21
2.14 MODELEM ŘÍZENÝ VÝVOJ 22

3 DIAGRAMY TŘÍD *(CLASS DIAGRAMS)* 23

3.1 OZNAČENÍ TŘÍDY V UML 23

3.2 OPERACE **24**
3.3 STRUKTURÁLNÍ VLASTNOSTI **25**
3.3.1 ATRIBUTY 25
3.3.2 ASOCIACE 26
3.4 AGREGACE (AGGREGATION) A KOMPOZICE (COMPOSITION) **28**
3.5 GENERALIZACE (GENERALIZATION) **29**
3.6 POZNÁMKY (NOTES) **32**
3.7 ZÁVISLOST (DEPENDENCY) **32**
3.8 OMEZUJÍCÍ PRAVIDLA (CONSTRAINT RULES) **32**
3.9 ABSTRAKTNÍ TŘÍDY (ABSTRACT CLASSES) A ROZHRANÍ (INTERFACES) **33**
3.10 SHRNUTÍ DIAGRAMŮ TŘÍD **34**

4 DIAGRAMY OBJEKTŮ *(OBJECT DIAGRAMS)* 36

4.1 ZOBRAZENÍ OBJEKTU **36**
4.2 VZTAHY MEZI OBJEKTY **37**
4.3 SHRNUTÍ DIAGRAMŮ OBJEKTŮ **39**

5 SEKVENČNÍ DIAGRAMY *(SEQUENCE DIAGRAMS)* 40

5.1 VÝPOČET CENY OBJEDNÁVKY **40**
5.2 VYTVÁŘENÍ A ZÁNIK ÚČASTNÍKŮ **42**
5.3 SYNCHRONNÍ A ASYNCHRONNÍ ZPRÁVY **43**
5.4 CYKLY A PODMÍNKY **44**
5.5 SHRNUTÍ SEKVENČNÍCH DIAGRAMŮ **46**

6 PŘÍPADY UŽITÍ *(USE CASES)* 47

6.1 DIAGRAM PŘÍPADŮ UŽITÍ **47**
6.2 DVA POHLEDY NA PŘÍPAD UŽITÍ **47**
6.3 VNĚJŠÍ POHLED **48**
6.3.1 AKTÉŘI 48
6.3.2 JEDNODUCHÝ DIAGRAM 48

6.3.3 GENERALIZACE AKTÉRŮ 49
6.3.4 ZAHRNUTÍ A ROZŠÍŘENÍ PŘÍPADU UŽITÍ 50
6.3.4.1 Zahrnutí 51
6.3.4.2 Rozšíření 51
6.4 SLOVNÍ POPIS PŘÍPADU UŽITÍ 52
6.5 SHRNUTÍ PŘÍPADŮ UŽITÍ 54

7 STAVOVÉ DIAGRAMY *(STATE MACHINE DIAGRAMS)* 55

7.1 PŘÍKLAD STAVOVÉHO DIAGRAMU 55
7.2 STAV OBJEKTU V PROGRAMU 57
7.3 REALITA VERSUS INFORMAČNÍ SYSTÉM 57
7.4 IMPLEMENTACE STAVOVÉHO DIAGRAMU 60
7.4.1 IMPLEMENTACE POMOCÍ VNOŘENÝCH PŘEPÍNAČŮ 60
7.4.2 IMPLEMENTACE POMOCÍ NÁVRHOVÉHO VZORU STAV 61
7.4.3 IMPLEMENTACE POMOCÍ STAVOVÉ TABULKY 62
7.5 SHRNUTÍ STAVOVÝCH DIAGRAMŮ 62

8 DIAGRAMY AKTIVITY *(ACTIVITY DIAGRAMS)* 64

8.1 PŘÍPRAVA KÁVY 64
8.2 AUDIT 66
8.3 ZÓNY (PARTITIONS) 67
8.4 SHRNUTÍ DIAGRAMŮ AKTIVITY 69

9 DIAGRAMY BALÍČKŮ *(PACKAGE DIAGRAMS)* 70

9.1 BALÍČEK 70
9.2 ZOBRAZENÍ BALÍČKU 71
9.3 ZÁVISLOSTI 72
9.4 SHRNUTÍ DIAGRAMŮ BALÍČKŮ 73

10 DIAGRAMY KOMPONENT *(COMPONENT DIAGRAMS)* 74

10.1 KOMPONENTA 74
10.2 PROČ MODELOVAT KOMPONENTY 74
10.3 ZOBRAZENÍ KOMPONENTY 75
10.4 SYSTÉM ZPRACOVÁVAJÍCÍ OBCHODNÍ ZPRÁVY 76
10.5 SHRNUTÍ DIAGRAMŮ KOMPONENT 77

11 DIAGRAMY NASAZENÍ *(DEPLOYMENT DIAGRAMS)* 78

11.1 PRVKY DIAGRAMU NASAZENÍ 78
11.2 DIAGRAM NASAZENÍ UNIVERZITNÍHO SYSTÉMU 79
11.3 DIAGRAM NASAZENÍ SÍTĚ ČERPACÍCH STANIC 81
11.4 SHRNUTÍ DIAGRAMŮ NASAZENÍ 81

12 DIAGRAMY KOMUNIKACE *(COMMUNICATION DIAGRAMS)* 82

12.1 ZOBRAZENÍ DIAGRAMU KOMUNIKACE 82
12.2 VÝPOČET CENY OBJEDNÁVKY 82
12.3 SHRNUTÍ DIAGRAMŮ KOMUNIKACE 84

13 DIAGRAMY STRUKTURY *(COMPOSITE STRUCTURE DIAGRAMS)* 85

13.1 VNITŘNÍ STRUKTURA TŘÍDY 85
13.2 VNITŘNÍ STRUKTURA KOMPONENTY 85
13.3 SHRNUTÍ DIAGRAMŮ STRUKTURY 86

14 DIAGRAMY ČASOVÁNÍ *(TIMING DIAGRAMS)* 87

14.1 ÚVODNÍ PŘÍKLAD 87
14.2 DALŠÍ VARIANTA ZOBRAZENÍ DIAGRAMU ČASOVÁNÍ 88
14.3 PŘEDÁVÁNÍ ZPRÁV MEZI OBJEKTY 89
14.4 SHRNUTÍ DIAGRAMŮ ČASOVÁNÍ 90

15 PŘEHLEDOVÉ DIAGRAMY INTERAKCE *(INTERACTION OVERVIEW DIAGRAMS)* 91

15.1 VYTVOŘENÍ REPORTU PŘEHLEDU OBJEDNÁVKY 91
15.2 SHRNUTÍ PŘEHLEDOVÝCH DIAGRAMŮ INTERAKCE 92

16 ZÁVĚR 93

1 Úvod

1.1 Charakter práce

Práce je rozdělena do několika kapitol, z nichž první, poněkud delší, je teoretická a pojednává o vizuálním modelování a jazyku UML. Další kapitoly se postupně zabývají jednotlivými diagramy UML 2.0, tyto kapitoly vždy obsahují základní seznámení s daným diagramem a jeho hlavními prvky, jejichž význam je názorně demonstrován na příkladech. Závěr každé kapitoly tvoří stručné shrnutí daného diagramu. Cílem práce není podávat detailní popis syntaxe UML, ale spíše praktické pochopení jeho důležitých prvků. Tato forma se zdá být vhodná jednak pro první seznámení s UML v rámci samostudia (samozřejmě s dalším studijním materiálem pokrývajícím syntaxi), ale především jako materiál k seminářům, kdy mohou vypracované příklady sloužit jako vhodné ukázky, které je však možno lektorem, či studentem samotným v rámci domácí úlohy snadno modifikovat.

1.2 Použitý nástroj

Pro tvorbu projektů byl použit nástroj *Visual Paradigm for UML* verze 5.3, Standard Edition, na který vlastní fakulta licenci. S projekty však lze pracovat i v nižší edici tohoto programu, nejnižší edice tohoto produktu je k dispozici zdarma k nekomerčnímu použití a lze ji stáhnout, stejně jako ostatní edice, ze serveru http://www. www.visual-paradigm.com. Ostatní edice jsou zdarma k vyzkoušení po dobu jednoho měsíce.

2 Vizuální modelování a jazyk UML

2.1 Vizuální modelování

Pod pojmem *vizuální modelování* budeme rozumět vytváření standardizovaných diagramů, které se provádí obvykle před zahájením programování, tedy ve fázi upřesňování požadavků, analýzy a designu (návrhu).

Vizuální modelování představuje velice užitečnou pomůcku při vývoji softwarových systémů. Zejména u rozsáhlejších řešení si opomenutí modelovací fáze předtím, než se pustíme do psaní kódu, nelze představit. Modely mají při tvorbě softwaru stejný význam jako výkresy při realizaci stavby. Slouží k zachycení architektury a chování budoucího systému na různých úrovních a jako podklady pro práci programátorů, ale také usnadňují komunikaci se zákazníkem a v rámci vývojového týmu. Obecná poučka nám říká, že tvorbě modelu bychom měli věnovat nejméně tolik času jako programování samotnému.

Hlavní výhodou vizuálního modelování je, že velice usnadní orientaci ve vytvářeném systému. Člověk je schopen vnímat najednou daleko více informací v grafické formě než při čtení textu a hlavně, celou složitou strukturu si dokáže v hlavě vhodným způsobem uspořádat, takže když se pak věnuje studiu podrobností, umí si je správně zařadit do celkového obrázku. Tím se dostáváme k dalšímu aspektu modelování – model v žádném případě nemůže nahradit podrobnou dokumentaci, ve které je vše rozebráno a popsáno do detailů. V modelu nezobrazujeme všechny informace, protože představuje určitou abstrakci, zachycuje celkový pohled na modelovanou skutečnost – ty nejzákladnější elementy a jejich vztahy. Diagramy modelu musí být především přehledné.

2.2 Vyjádření modelů

S rozšiřováním objektového přístupu mimo akademickou půdu v 80. letech se zvyšuje i zájem o analýzu a design, na přelomu 80. a 90. let vzniká mnoho tzv. metodik (ve skutečnosti to však většinou nebyly metodiky v pravém slova smyslu, ale syntaxe pro vizuální modelování a sada pouček), které mají mnoho společného, obsahují však množství drobných odlišností a především často používají zcela odlišné notace pro vyjádření téhož.

2.3 Diagramy

Diagramy se v softwarovém vývoji používaly prakticky odjakživa, nejprve na intuitivním základě a teprve později byly přesně formalizovány. S diagramy se můžeme setkat u databázových schémat, kde je jejich úkolem přehledně zobrazit relační strukturu databáze, tedy vztahy mezi tabulkami, v objektových knihovnách, kde zobrazují hierarchickou strukturu a vazby mezi třídami či v podobě klasického vývojového diagramu, popisujícího dynamické chování programu.

2.4 Zrození UML

V roce 1994 odchází Jim Rumbaugh z *General Electric*, aby se spojil s Grady Boochem v *Rational Software* s cílem spojit své metodiky v jednu. Na mezinárodní konferenci OOPSLA (*Object-Oriented Programming Systems, Languages and Applications*) představují první veřejnou specifikaci své sloučené metodiky – „*Unified Method*“ verze 0.8 a oznamují, že *Rational Software* koupila metodiku „*Objectory*“ Ivara Jacobsona, který se posléze připojuje k „*Unified*“ týmu.

Během roku 1996, Booch, Rumbaugh a Jacobson (nyní často nazýváni „*Three amigos*“) pracují na své metodě pod novým názvem – „*Unified Modeling Language*“ (UML).

V témže roce navrhlo sdružení OMG (*Object Management Group*, spojení dodavatelů, které vzniklo za účelem definování a prosazování specifikací objektů CORBA) specifikaci RFP (*Request For Proposal*, postup vytváření standardu v síti Internet a zároveň požadavek na označení takto vzniklého standardu) pro objektově orientovaný jazyk pro vizuální modelování, v němž jako standard navrhlo jazyk UML.

V roce 1997 sdružení OMG jazyk UML přijalo jako průmyslový standard objektově orientovaného jazyka pro vizuální modelování. Od té doby všechny jiné soupeřící metody upadly v zapomnění a jazyk UML byl veřejností bez námitek přijat.

V době psaní této práce je poslední verzí UML 2.0, ve které jazyk UML doznal oproti předchozí verzi 1.5 podstatných změn, které rozšiřují jeho možnosti.

2.5 Přehled historie verzí UML

- 1994 – zahájení vývoje UML (Rumbaugh, Booch)
- 1995 – notace pro *Unified Method 0.8*, připojuje se Jacobson a model jednání, *Rational Unified Process*
- 1996 – UML 1.0 (zabudování připomínek, kooperace)
- 1997 – UML 1.1 (preciznější sémantika, přijat jako standard OMG)
- 1999 – UML 1.3 (upřesněná verze 1.1)

- 2001 – UML 1.4 (komponenty)
- 2003 – UML 1.5 (diagramy aktivit, sémantika akcí)
- 2005 – UML 2.0 (nové diagramy, změny) původně plánované ukončení květen 2002, v září 2003 přijata *OMG Final Adopted Specification*, slib ukončení specifikace v roce 2004, ale příliš mnoho hlav,obtížné

2.6 Co je to UML?

UML je zkratka pro „*Unified Modeling Language*“, tedy „*Sjednocený Modelovací Jazyk*“. Sjednocený proto, že jedním z jeho cílů je sjednocení používaných výrazových prostředků. Vznikl v polovině 90. let sloučením tří různých vizuálních jazyků a v roce 1997 byl přijat konsorciem OMG (*Object Management Group*) jako průmyslový standard pro záznam, vizualizaci a dokumentaci artefaktů systémů s převážně softwarovou charakteristikou.

UML je skupina grafických notací, podepřená jednotným metamodelem, která pomáhá při popisu a návrhu softwarových systémů, a to především systémů vytvořených objektově orientovaným stylem. Obsahuje bohatou sémantiku a syntaxi, který usnadňuje návrh a vizualizaci různých typů aplikací. Navíc tvůrci jazyka UML nebyli natolik naivní ani sebevědomí, aby věřili, že standardními elementy jazyka UML pokryli kompletně veškeré požadavky návrhářů a přímo do jazyka zabudovali mechanismy rozšíření, které usnadňují řízenou deklaraci a definici nových elementů jazyka.

UML nabízí standardní způsob zápisu jak návrhů systému včetně konceptuálních prvků jako jsou business procesy a systémové funkce, tak konkrétních prvků jako jsou příkazy programovacího jazyka, databázová schémata a programové komponenty. UML však neobsahuje způsob, jak se má

používat, ani neobsahuje metodiku(y) jak analyzovat, specifikovat či navrhovat programové systémy.

UML je jazyk, který umožňuje modelovat jednoduché i složité aplikace pomocí stejné formální syntaxe, a proto můžete výsledky své práce sdílet s ostatními návrháři. UML ale není všemocné. I když umíte UML, ovládáte pouze nástroj. Sice velmi výkonný, ale stále jen nástroj.

Při návrhu UML autoři postupovali tak, že z existujících konkurenčních jazyků vybrali nejlepší myšlenky a adoptovali je do jazyka UML, který podtrhl jejich význam integrací do nově promyšleného celku. Na druhou stranu se snažili vyvarovat slabin ostatních jazyků.

2.7 UML 2.0

Ve verzi 1.5 bylo do UML zahrnuto mnoho nových rysů, ale jejich sémantika nebyla zcela precizní. Hlavní důraz verze 2.0 je kladen na možnost precizního vyjádření sémantiky zápisů v UML. Motivací je fakt, že nově vznikající systémy jsou natolik složité, že není možné používat při návrhu stejné postupy jako dříve. Verze 2.0 byla navrhována tak, aby ten, kdo nemá o nové rysy zájem mohl používat UML stejně jako dříve. Nové rysy ovšem mohou přinést výhody a inspirovat nové podpůrné nástroje.

Standard ve verzi 2.0 se skládá ze čtyř částí:

- *UML 2.0 Infrastructure* – metamodel stojící v pozadí za UML, představující notaci UML (syntax), definuje základní elementy, jádro UML a související standardy, je specifikovaný pomocí *Meta-Object Facility* (MOF).

- *UML 2.0 SuperStructure* – sémantika definovaná pomocí metamodelu, popisuje prvky metamodelu, konstrukty používané uživateli UML – elementy diagramů a diagramy.
- *UML 2.0 Object Constraint Language* (OCL) – jazyk pro formálně přesnou specifikaci vstupních a výstupních podmínek a invariantů v jednotlivých diagramech.
- *UML 2.0 Diagram Interchange* – popis struktur pro výměnu konkrétních modelů mezi jednotlivými modelovacími nástroji, specifikace převodu do výměnných formátů (XML, XMI (*XML Metadata Interchange*), CORBA, IDL).

2.8 Možnosti přizpůsobení UML

Vedle vlastního standardu existují UML profily – přizpůsobení UML pro jednotlivé oblasti:

- *UML Profile for CORBA*
- *UML Profile for CORBA Component Model* (CCM)
- *UML Profile for Enterprise Application Integration* (EAI)
- *UML Profile for Enterprise Distributed Object Computing* (EDOC)
- *UML Profile for QoS and Fault Tolerance*
- *UML Profile for Schedulability, Performance, and Time*
- *UML Testing Profile*

Z UML též začínají vznikat různé dialekty – modelovací jazyky pro určité oblasti, které přebírají část UML, kterou modifikují a doplní o prvky specifické pro konkrétní oblast. Příkladem může být jazyk *Systems Modeling Language* (SysML), určený pro specifikaci, analýzu, návrh, verifikaci a validaci různých systémů (technické, programové, informační, procesní, zabezpečovací, ...). Též většina metodik pro analýzu a návrh systémů upřednostňuje části z UML a doplňuje je o další prvky.

2.9 Specifikace UML

Specifikaci UML, stejně jako další informace včetně seznamu *CASE nástrojů* podporujících UML lze nalézt na oficiálním webu projektu: http://www.uml.org.

2.10 Diagramy UML

Diagramy jsou nejznámější a nejpoužívanější částí standardu. Následuje přehled diagramů v UML 2.0 včetně jejich rozčlenění do skupin:

- **strukturní diagramy:**
  - diagram tříd (*class diagram)*
  - diagram komponent (*component diagram*)
  - diagram struktur (*composite structure diagram*)
  - diagram nasazení (*deployment diagram*)
  - diagram balíčků (*package diagram*)
  - diagram objektů (*object diagram*), též se nazývá diagram instancí
- **diagramy chování:**
  - diagram aktivit (*activity diagram*)
  - diagram případů užití (*use case diagram*)

- stavový diagram (*state machine diagram*)
- **diagramy interakce:**
    - sekvenční diagram (*sequence diagram*)
    - diagram komunikace (*communication diagram*, dříve *collaboration diagram*)
    - přehledový diagram interakce (*interaction overview diagram*)
    - diagram časování (*timing diagram*)

2.11 Způsoby použití UML

Různí lidé používají UML různými způsoby, tyto rozdíly vycházejí z používání modelovacích jazyků, které předcházely UML a také z architektury UML. Z hlediska architektury se používá UML na dvou úrovních:

2.11.1 Diagramy

2.11.1.1 Kreslení konceptu

Při tomto použití je UML podpůrným nástrojem pro komunikaci mezi vývojáři a pro zaznamenání myšlenek a návrhů. Do diagramů se kreslí pouze věci podstatné pro grafické vyjádření návrhu, části návrhu před tím, než se začne programovat. Důležitá je srozumitelnost, rychlost nakreslení a snadnost změny či navržení alternativ řešení.

2.11.1.2 Kreslení detailních návrhů

Cílem je zaznamenat kompletní návrh či kompletní realizaci. Při kreslení návrhu by měl analytik obsáhnout všechny prvky tak, aby programátor byl schopen vytvořit program bez velkého přemýšlení nad věcnou oblastí (pro programátora by neměla vzniknout potřeba konzultace s uživatelem). Při kreslení detailních návrhů se obvykle používají specializované programy,

CASE nástroje, které jsou schopny sdílet informace mezi jednotlivými modely a kontrolovat konzistenci návrhu. Při dokumentaci programu se často používání nástroje pro generování diagramů z vlastního kódu aplikace.

2.11.1.3 UML jako programovací jazyk

Při tomto použití vývojář nakreslí UML diagramy, ze kterých se vygeneruje přímo spustitelný kód. Toto vyžaduje specializované nástroje a velmi přesné vyjadřování v UML diagramech. V této souvislosti se velmi často používá pojem *Model Driven Architecture* (MDA), což je další standard skupiny OMG, který se snaží standardizovat použití UML jako programovacího jazyka.

2.11.2 Metamodel

Tento pohled používají autoři UML a autoři CASE nástrojů - nedívají se na UML jako na diagramy, pro ně je základem UML metamodel (diagramy jsou pouze grafickou reprezentací metamodelu). Při tomto přístupu se často používá pojem model místo pojmu diagram, např. místo diagramu tříd se používá pojem model tříd. Metamodel se popisuje pomocí *Meta-Object-Facility* (MOF) - abstraktního jazyka pro specifikaci, vytváření a správu metamodelů (další standard OMG). Pro výměnu metamodelů se používá XMI - na XML založený standard (součást standardu UML).

2.12 Proč právě UML?

Univerzální modelovací jazyk UML představuje prostředek pro komunikaci v komunitě softwarových inženýrů, vývojářů a databázových expertů. Pro jeho používání hovoří především tyto hlavní znaky a z nich plynoucí výhody:

- UML byl vyvinut na základě zkušeností s mnoha různými metodikami a notacemi.

- UML umožňuje vyjádření dokumentace analýzy, návrhu i implementace.
- UML je unifikovaný vyjadřovací prostředek pro dokumentaci obsahové části informatických projektů (dokumentace projektu obsahuje ještě tzv. projektovou dokumentaci, která popisuje projekt jako takový).
- UML umožňuje i vyjádření strukturovaného přístupu, ale byl určen zejména pro objektově-orientovaný přístup.
- další - univerzálnost, unifikovanost a z ní vyplývající rozšířenost a široká podpora v *case* nástrojích

2.13 CASE nástroje

CASE (*Computer Aided Software Engineering*) nástroje jsou nástroje pro podporu tvorby softwarových systémů (především ve fázi analýzy a návrhu). V současnosti všechny ve světě rozšířené objektově orientované CASE nástroje vycházejí z modelovacího jazyka UML. Při návrhu rozsáhlejších informačních systémů se bez použití těchto nástrojů neobejdeme. Diagramy UML je sice možné zachytit i obvyklými kreslícími programy (např. MS VISIO), pouze plnohodnotné CASE nástroje ale umožní propojení jednotlivých technik UML, sdílení modelů mezi členy týmu, a tedy týmovou práci.

Z běžně užívaných CASE nástrojů uveďme pro ilustraci *Rational Rose* firmy *Rational*, *Sparx Systems Enterprise Architect* či *Visual Paradigm for UML*, který byl použit pro tvorbu projektů této práce. Seznam ostatních nástrojů je možno nalézt na domovské stránce UML (http://www.uml.org), existuje však řada dalších, které zde nejsou zaregistrovány. Jejich cena je řádově desetinásobně vyšší než cena kreslících produktů (MS VISIO), kromě již zmíněných možností však zpravidla obsahují možnosti generování a

synchronizace kódu objektových prostředí (.NET, Java, C++ atd.), včetně generování databázových skriptů pro založení databáze, reverzního inženýrství (*reverse engineering* – zpětné vygenerování modelu z existujícího kódu) a správy datových modelů. Některé *CASE* nástroje v sobě dokonce obsahují metodiku (např. zmíněný *Rational Rose* a metodika RUP – *Rational Unified Process*) a umožňují podporu řízení softwarových projektů.

2.14 Modelem řízený vývoj

Současný způsob vytváření informačních systémů přešel postupně od kódem řízeného stylu ke specifikací řízenému stylu. Nejnovějším trendem konsorcia OMG je koncept „Modelem řízený vývoj“ (MDD - *Model Driven Development*) a jemu odpovídající architektura (MDA - *Model Driven Architecture*), který definuje přístup oddělující specifikaci funkcionality od specifikace implementace a tuto od specifikace implementace na konkrétní technologické platformě.

Specifikací řízený styl práce předpokládá, že co nejvíce kódu bude možno generovat ze specifikace, aby architekt a vývojář mohl pracovat s modelem, nikoliv s detaily implementace.

3 Diagramy tříd *(class diagrams)*

Diagram tříd je zřejmě nejdůležitějším a nejvíce používaným typem UML diagramu.

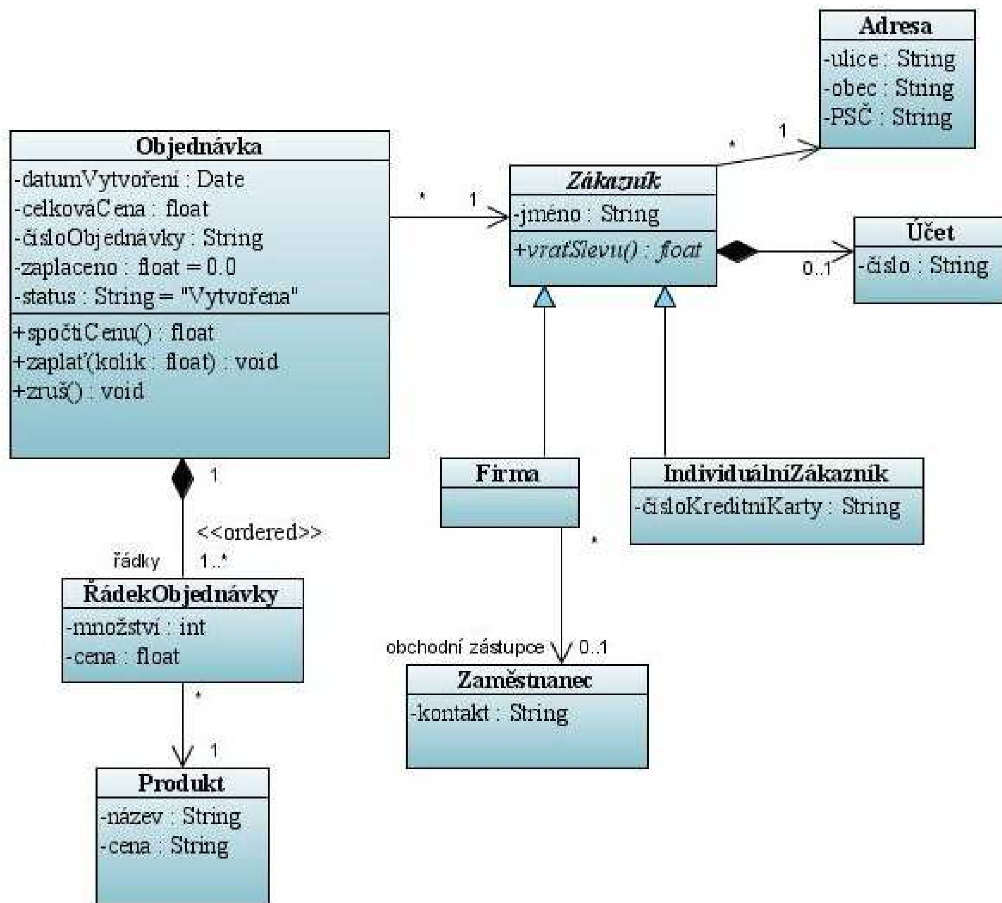
Diagram tříd popisuje typy objektů v systému a různé druhy statických vztahů, které mezi nimi existují. Také zobrazují strukturální vlastnosti a operace třídy a omezení *(constraints)* na vazby mezi objekty tříd. V UML se používá anglický termín *properties*, který v sobě zahrnuje jak strukturální vlastnosti tak operace třídy.

3.1 Označení třídy v UML

Ve všech diagramech UML můžeme zachycovat systém na různých úrovních abstrakce a detailu. Nejsou kladena žádná syntaktická omezení a volba stupně detailu závisí na momentální potřebě. U diagramu tříd je tento fakt asi nejzřejmější, což vychází z bohaté syntaxe tohoto diagramu (lze na něm použít zřejmě nejvíce prvků ve srovnání s ostatními diagramy UML2).

Třída, jako element diagramu tříd, představuje třídu v objektově orientovaném systému. Třída je v diagramu tříd znázorněna jako obdélník, který je rozdělen do několika částí. Povinný je však pouze název, který je uveden v horní části. V prostřední části jsou uvedeny atributy a v dolní operace.

Na následujícím obrázku je znázorněn diagram tříd představující část objednávkového systému. Postupně si vysvětlíme hlavní prvky diagramu tříd, na konci kapitoly by měl být význam obrázku zřejmý.



Obrázek ClD1: *Diagram tříd části objednávkového systému*

3.2 Operace

Operace jsou akce, které třída umí provádět. Operace tvoří základ budoucích metod. Plná forma zápisu operace v UML2 je ve tvaru:

viditelnost název (seznam parametrů) : návratový typ {vlastnost}

- *viditelnost* značí viditelnost tak, jak ji známe z programovacích jazyků – *private*(-), *public*(+),…
- *návratový typ* určuje typ návratové hodnoty

- *vlastnost* - zde můžeme uvést další vlastnosti, či omezení vztahující se k operaci.

- Parametry v *seznamu parametrů* jsou odděleny čárkou a jsou ve tvaru:

  směr název: typ = výchozí hodnota

  - *směr* může být vstupní *(in)*, výstupní*(out)* nebo vstupně-výstupní *(inout)*

  - *výchozí hodnota* je hodnota tohoto parametru pokud nebyla explicitně zadána *jiná* hodnota.

Například:

+*účetníZůstatek*(in datum:*Date*) : Částka

3.3 Strukturální vlastnosti

Zjednodušeně by se dalo říci, že strukturální vlastnosti (dále jen *vlastnosti*) tvoří základ instančních proměnných třídy na implementační úrovni. Vlastnosti mají dvě rozdílné formy notace: *atributy* a *asociace*.

3.3.1 Atributy

Plná forma zápisu atributu má tuto podobu:

viditelnost název: typ [násobnost] = výchozí hodnota {vlastnost}

Například:

+ jméno: *String* [1] = „Neuvedeno“ {*read only*}

U atributu je povinný pouze *název*, ostatní části jsou volitelné.

- *viditelnost* značí viditelnost tak, jak ji známe z programovacích jazyků: *private*(-), *public*(+),…
- *typ* určuje datový typ atributu
- *násobnost* představuje četnost hodnot nebo objektů v konkrétním čase představujících atribut

  1 - *objednávka* má VŽDY jedno *číslo*

  0..1 - *objednávka* může ale nemusí mít uvedeno *datumObdržení*

  * - *zákazník* může mít více *objednávek*, ale nemusí mít žádnou

- *výchozí hodnota* je hodnota tohoto atributu u nově vytvořené instance, pokud nebyla explicitně zadána (např. v konstruktoru) jiná hodnota.
- *vlastnost* - zde můžeme uvést další vlastnosti, či omezení vztahující se na atribut, vlastnost *{read only}* v našem příkladě sděluje, že atribut je pouze pro čtení.

3.3.2 Asociace

Sémantický význam asociace je stejný jako u atributu, avšak notace a možnosti použití jsou jiné. Asociaci pro znázornění vlastnosti třídy použijeme tehdy, jestliže tato vlastnost je objektového typu (je reprezentována objektem nějaké třídy) – tzn. neprimitivní datový typ, který má taktéž své vlastnosti a operace, jež si přejeme v našem diagramu znázornit. Asociace je znázorněna plnou čarou spojující dvě třídy, která je orientována od zdroje (třída vystupující ve vztahu jako nositel vlastnosti) k cíli (třída představující vlastnost). Název a viditelnost vlastnosti (a zřejmě budoucí instanční proměnné v implementaci třídy) píšeme na stranu cíle. Na rozdíl od atributu můžeme u asociací uvést násobnost na oba konce asociace. To znamená, že omezení týkající se počtu

objektů daných tříd musíme chápat obousměrně a v reálném čase může nastat libovolná kombinace přípustných počtů objektů daných tříd, které se podílejí na vztahu (asociaci).

Například:

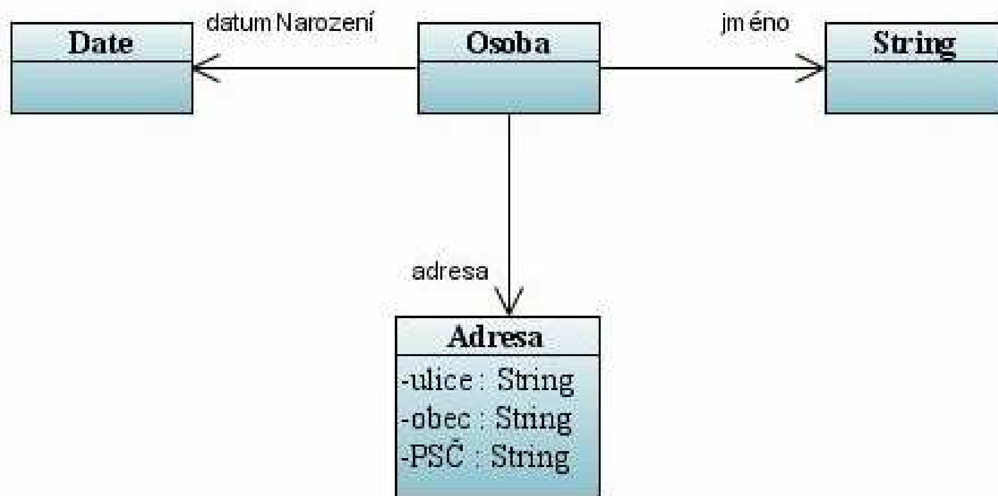
Libovolnou *objednávku* vlastní vždy 1 *zákazník* (povinně), avšak libovolný *zákazník* může mít 0 až neomezeně *objednávek* a jejich počet se může v čase měnit.

Firma může, ale nemusí mít *zaměstnance*, který vystupuje jako její *obchodní zástupce*, avšak libovolný *obchodní zástupce* může zastupovat najednou více *firem* (čímž se stává zaměstnancem všech těchto firem) a nebo může být „nezaměstnaný“ – nezastupuje žádnou firmu.

Atributy většinou použijeme pro hodnotové (ať už primitivní či obalené) jako *boolean* či *long* nebo jednoduché objektové (*Date*, *String*) typy, zatímco asociace použijeme pro významné třídy, které jsou v našem systému důležité samy o sobě, nejen pro to, že jsou vlastností jiné třídy.

Následující dva obrázky ukazují různý způsob zobrazení téhož. Jak je vidět, z prvního obrázku se nedozvíme nic o třídě *Adresa*, zatímco používat asociace pro typ *String* a *Date* na druhém obrázku se jeví jako zbytečné.

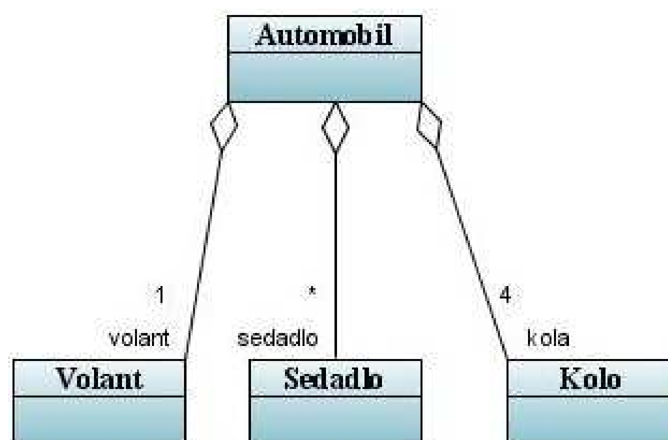
Obrázek ClD2a: *Vlastnosti třídy jako atributy*



Obrázek ClD2b: *Vlastnosti třídy jako asociace*

3.4 Agregace (Aggregation) a Kompozice (Composition)

Agregace je zvláštním druhem asociace mezi třídami. Udává vztah „býti částí“ a znamená, že budoucí instance jedné třídy (celek) obsahují jako své části jednu či několik instancí druhé třídy. Např. *automobil* je agregací čtyř *kol*, jednoho *volantu*, několika *sedadel*, atd. Agregace se značí plnou čarou s prázdným kosočtvercem na straně vlastníka (celku).



Obrázek ClD3: *Agregace*

Kompozice je silnějším typem agregace, kdy část (kompozit) nemůže existovat samostatně bez celku, protože její samostatná existence nemá význam. Například řádek objednávky musí být součástí objednávky a nesmí existovat samostatně. Pokud smažeme objednávku, zaniknou automaticky i všechny její řádky. Dalším omezením oproti agregaci je výlučné vlastnictví kompozitu pouze jedním vlastníkem (jeden konkrétní řádek objednávky se může vyskytovat pouze na jedné objednávce, na žádné jiné).

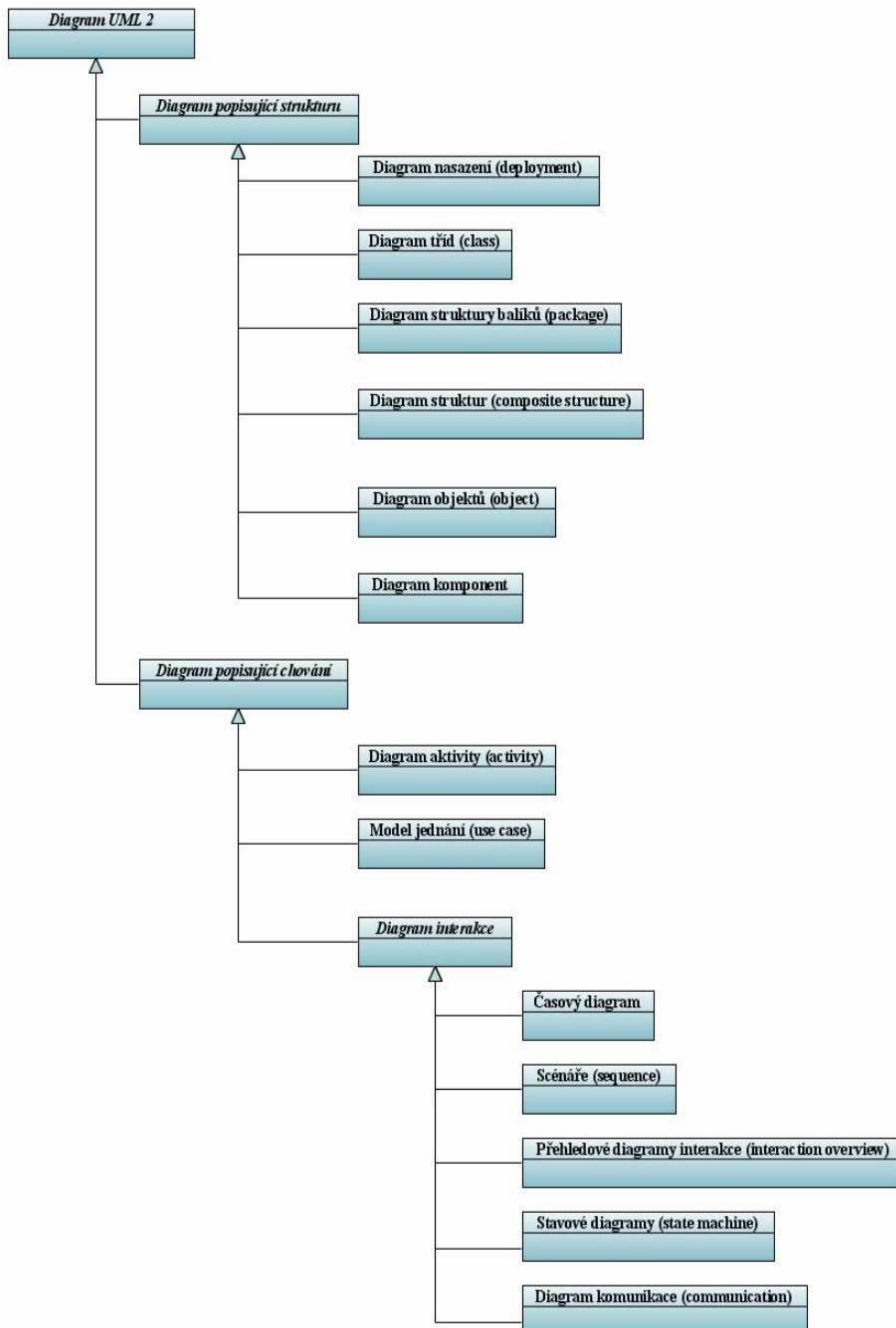
3.5 Generalizace (generalization)

Generalizace je koncept, který známe z OO jazyků jako dědičnost *(inheritance).*

V našem příkladě s objednávkou existuje abstraktní třída *zákazník* která má dva potomky – *IndividuálníZákazník* a *Firma*, vlastnosti z nadtřídy jsou pro obě podtřídy společné. Nadtřída je abstraktní proto, že obsahuje abstraktní metodu *vraťSlevu()*, která může být v každé podtřídě implementována jiným způsobem, což znamená, že jinak se určí výše slevy pro *individuálního zákazníka* a jinak pro *firmu*. To, že je nadtřída abstraktní také znamená, že z ní

nemůžeme tvořit instance, obecný *zákazník* v našem systému neexistuje, vždy to je buď individuální zákazník, nebo firma.

Jako jiný příklad si uvedeme uspořádání diagramů UML2. Abstraktní třída „*Diagram UML 2*“ představující nejobecnější UML 2 diagram má dva potomky – abstraktní třídy „*Diagram popisující chování*“ a „*Diagram popisující strukturu*“. Tyto třídy jsou abstraktní proto, že nemůžeme vytvořit jejich „instanci“. Abstraktní třídy na diagramu tříd poznáme tak, že jejich název je vypsán kurzívou. Od těchto abstraktních tříd jsou – přímo či nepřímo – pomocí generalizace odvozeny všechny diagramy UML 2.



Obrázek ClD4: *Diagram tříd hierarchie diagramů UML 2*

3.6 Poznámky (Notes)

Poznámka je důležitý a často používaný prvek UML. Můžeme ji použít na jakémkoliv diagramu. Symbol poznámky je obdélník s přehnutým pravým horním rohem, v obdélníku je vepsán text poznámky, čárkovanou čárkou můžeme poznámku připojit k prvku, kterého se týká.

3.7 Závislost (Dependency)

Mezi dvěma elementy existuje závislost, pokud změna definice jednoho elementu (dodavatele nebo zdroje) může způsobit změnu ve druhém elementu (klientu nebo cíli). Mezi třídami existují závislosti z různých důvodů: instance jedné třídy posílá zprávu instanci druhé třídy, instance jedné třídy obsahuje instanci druhé třídy jako svoji část, nebo ji používá jako parametr metody. Jestliže třída změní své rozhraní, zpráva poslaná instanci této třídy již nemusí být validní.

Závislost se v diagramu tříd značí přerušovanou čarou s šipkou označující směr závislosti.

Jak se náš systém rozrůstá, měli bychom se více starat o „správu“ závislostí. Jestliže nad vazbami ztratíme kontrolu, může mít jakákoliv změna v systému velmi nepříznivý dopad - musíme provádět další změny v závislých částech. Čím více vazeb, tím je těžší dělat v systému změny. O základech správy závislostí se zmíníme v kapitole pojednávající o diagramech balíčků.

3.8 Omezující pravidla (Constraint rules)

Podstatná část toho co zakreslíme do diagramu indikuje určitá omezení. Z obrázku je například zřejmé, že o*bjednávka* patří pouze jednomu

zákazníkovi, nebo, že pro každý typ *produktu* je na *objednávce* samostatný *řádek objednávky* (nemícháme jablka a hrušky), atd.

Základní konstrukty, jako asociace, atribut nebo zobecnění dobře vyjadřují důležitá omezení, ale nemůžeme pomocí nich vyjádřit každé omezení.

Pro vyjádření těchto omezení použijeme text, který musí být podle specifikace uzavřen ve složených závorkách ({}). Pro vyjádření omezení můžeme použít přirozený jazyk, programovací jazyk, nebo formální jazyk, který je součástí UML – *Object Constraint Language* (OCL), který je založen na predikátovém kalkulu. Použitím formálního jazyka se vyvarujeme nejednoznačnosti, která může vzniknout při popisu pomocí přirozeného jazyka, ovšem je potřeba mít jistotu, že všichni potenciální čtenáři našich diagramů znají dobře notaci OCL.

3.9 Abstraktní třídy (Abstract classes) a rozhraní (Interfaces)

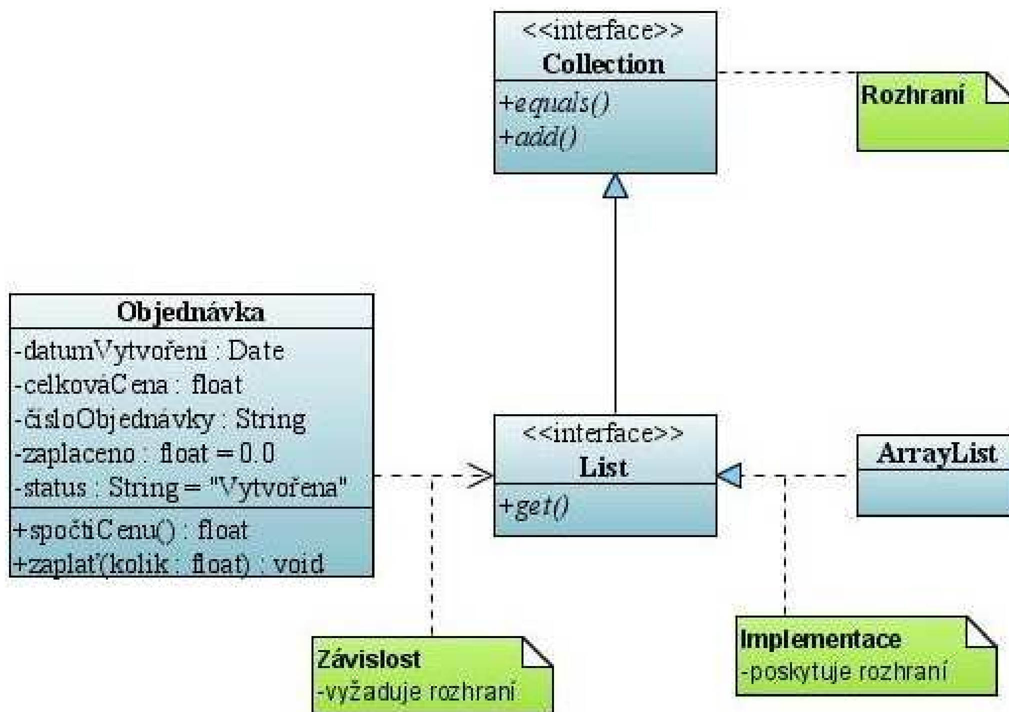
Abstraktní třída je třída, která nemůže být přímo instanciována. Instanciovat lze její (neabstraktní) podtřídy. Typicky má abstraktní třída jednu či více *abstraktních operací*. *Abstraktní operace* nemá implementaci, pouze tzv. signaturu (ta zahrnuje název operace, návratový typ a výčet typů parametrů v předem stanoveném pořadí, nepovinně může obsahovat i názvy parametrů) a její implementace je provedena v podtřídě od níž chceme tvořit instance. Abstraktní třídy a operace označujeme v UML kurzivou.

Rozhraní je třída, která nemá implementaci, tzn. že všechny její vlastnosti jsou abstraktní. Rozhraní v UML diagramu označujeme klíčovým slovem *<<interface>>*.

Třídy mají dva druhy vztahů s rozhraními: buď ho *poskytují* nebo *požadují*.

Třída *poskytuje rozhraní*, jestliže může být náhradou za toto rozhraní. V Javě docílíme poskytnutí rozhraní třídou, tím že ho (nebo jeho podtyp) v této třídě implementujeme.

Třída *vyžaduje rozhraní*, jestliže její instance potřebuje ke své funkčnosti instanci třídy, která toto rozhraní implementuje, existuje závislost třídy na tomto rozhraní.



***Obrázek ClD5:** Diagram tříd s rozhraními*

3.10 Shrnutí diagramů tříd

Diagramy tříd tvoří páteř UML, s výhodou je použijeme téměř všude. Problémem diagramů tříd se může stát jejich bohatá syntaxe, která může začátečníka snadno zahltit. Při používání diagramů tříd je proto užitečné držet se následujících doporučení:

- Nesnažte se používat veškeré dostupné prvky diagramů tříd. Začněte základními prvky: třídy, asociace, atributy, zobecnění, omezení, pokročilejší notace používejte pouze v případě, že je opravdu potřebujete.

- Diagramy tříd mohou být velmi užitečné při zkoumání obchodních domén. Při této činnosti je však potřeba striktně odhlédnout od souvislosti s budoucím softwarovým systémem a také používat velmi jednoduchou syntaxi.

- Nedělejte modely všeho, místo toho se soustřeďte na klíčové oblasti. Je lepší mít několik jednodušších diagramů, které používáte a udržujete aktuální, než mnoho složitých, zapadlých a neaktuálních modelů.

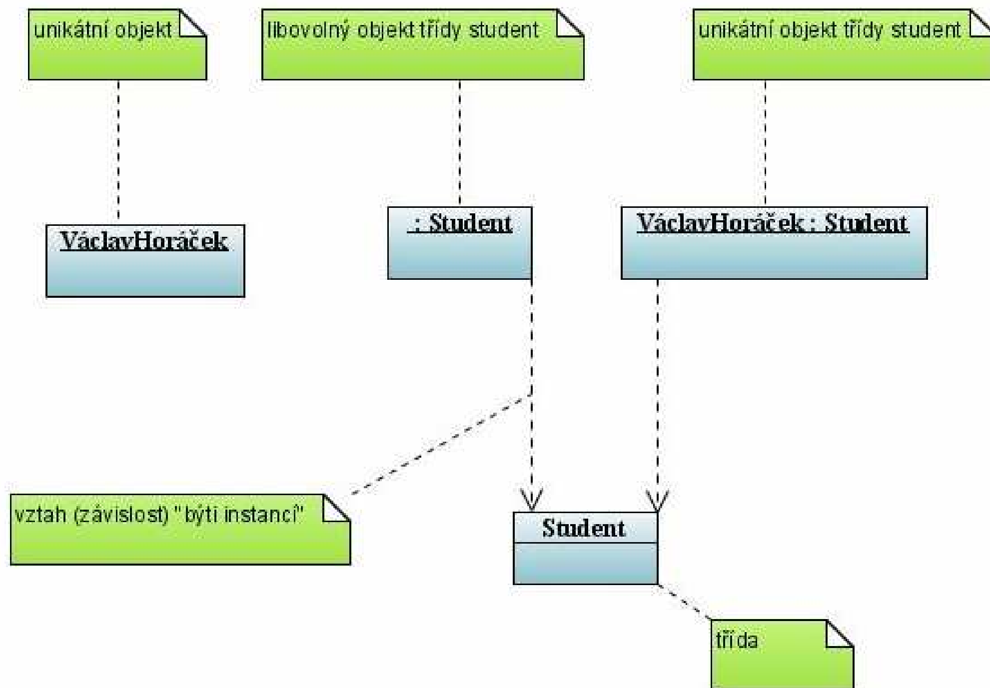
- Velké nebezpečí při používání diagramů tříd je přílišné zaměření na strukturu při současném přehlédnutí chování systému, proto je nutné kreslení diagramu tříd spojit s další technikou zobrazující chování systému. V ideálním případě se budete často přesouvat mezi oběma technikami.

4 Diagramy objektů *(object diagrams)*

Diagram objektů představuje jakýsi „snímek“ objektů v systému v daném časovém okamžiku. Protože zobrazuje instance tříd, je také někdy nazýván *„diagram instancí“ (instance diagram).*

4.1 Zobrazení objektu

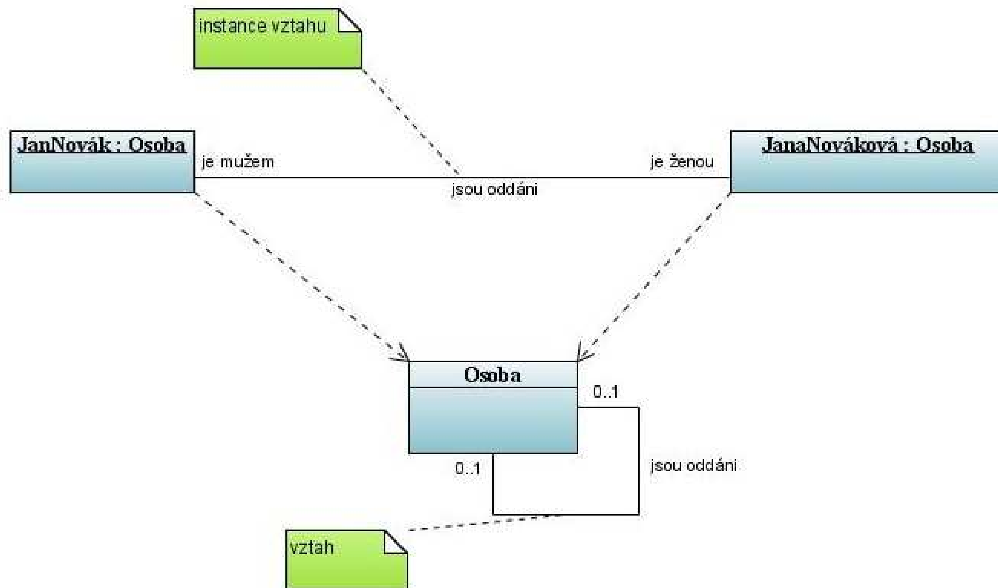
Pro zobrazení objektu se v UML používá stejný symbol jako pro třídu, tedy obdélník, název objektu je však podtržen. Objekt můžeme pojmenovat několika způsoby. *Plný název* objektu se skládá z názvu instance a názvu třídy, ze které byl vytvořen, oddělené dvojtečkou, můžeme však také uvést pouze *název objektu bez třídy*, či pouze *název třídy bez názvu instance* (libovolný objekt dané třídy), v tom případě však musíme na začátek názvu uvést dvojtečku. V diagramu můžeme též zobrazit třídu, ze které byl objekt vytvořen objekt, mezi objektem a třídou existuje automaticky jednosměrná závislost (tedy asymetrická relace – objekt je závislý na třídě, třída na objektu však ne) „býti instancí“, objekt „je instancí“ třídy. Uvedené skutečnosti jsou znázorněny na následujícím obrázku.



Obrázek OD1: *Možnosti zobrazení objektu*

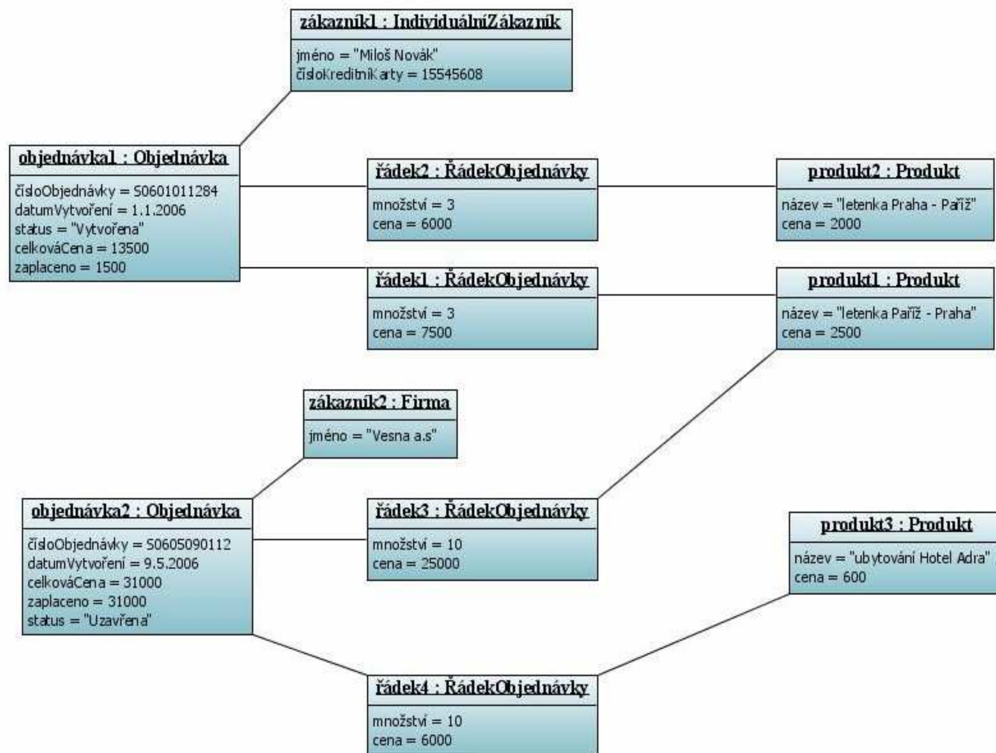
4.2 Vztahy mezi objekty

Stejně jako znázorňujeme určitý vztah mezi třídami, můžeme i mezi objekty znázornit *instanci tohoto vztahu*, tedy konkrétní realizaci vztahu, mezi konkrétními objekty.



Obrázek OD3: *Vztahy mezi objekty*

Protože objekty jsou vytvářeny ze tříd, mají všechny jejich vlastnosti (atributy a asociace), ty můžeme na diagramu objektů zobrazit včetně konkrétních hodnot. Na dalším obrázku vidíme možnou situaci *systému s objednávkami*. Je zřejmé, že několik podobných diagramů může snadno vyjasnit některé složitější části systému, i zde však platí, že méně je mnohdy více a proto i náš demonstrativní příklad může být v pro některé účely vyhovující, pro jiné však zbytečně podrobný.



Obrázek OD3: *Diagram objektů objednávkového systému*

4.3 Shrnutí diagramů objektů

Diagram objektů použijeme, chceme-li zobrazit stav části systému z hlediska objektů v daném čase. S výhodou ho použijeme pro zobrazení konkrétní situace, která v systému vzniká, což nám pomůže pochopit složitější strukturu definovanou diagramem tříd.

5 Sekvenční diagramy *(sequence diagrams)*

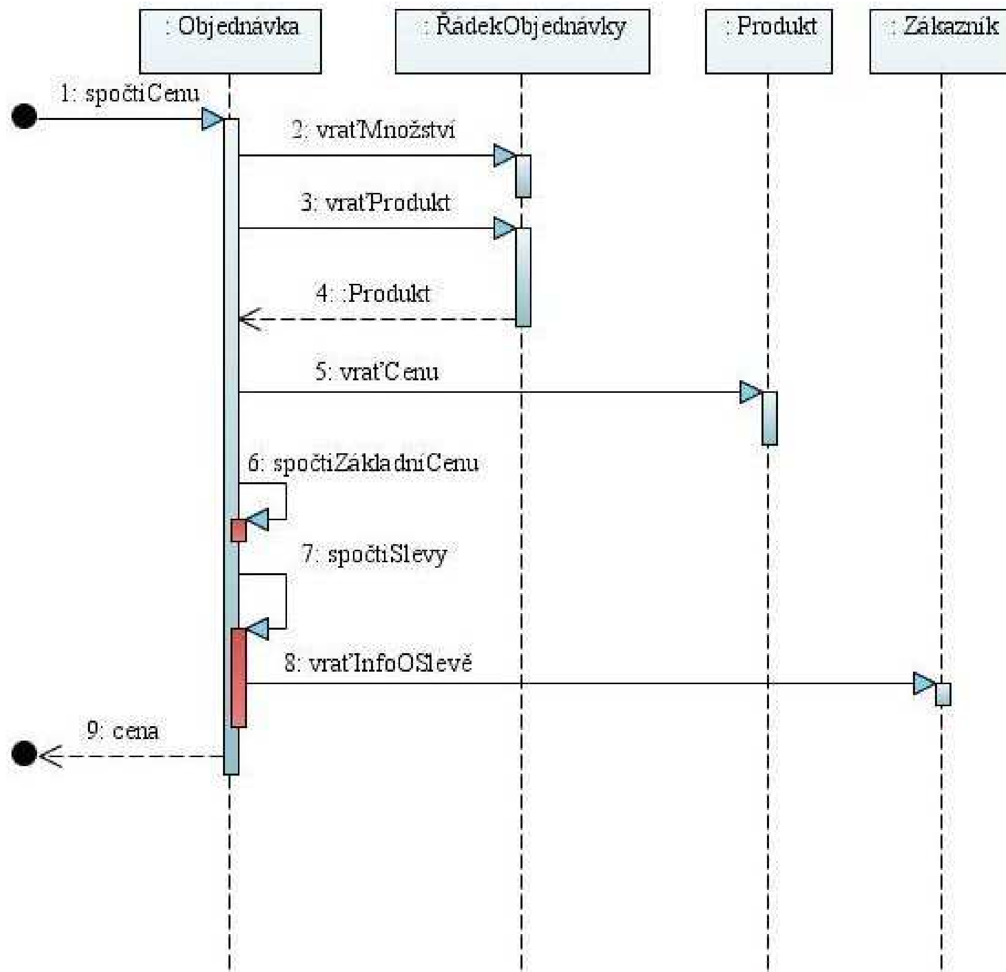
Sekvenční diagramy typicky zachycují chování v rámci jednoho *scénáře případu užití*. Diagram znázorňuje *skupinu objektů* a *zprávy*, které si posílají během tohoto scénáře.

5.1 Výpočet ceny objednávky

Začněme následujícím jednoduchým scénářem.

Máme *objednávku* a chceme spočítat její celkovou *cenu*, proto zavoláme metodu *spočtiCenu()* objektu *objednávka*. Tato metoda pracuje tak, že nejprve projde všechny *řádky*, které obsahuje (kolekci objektu *ŘádekObjednávky*), svou *cenu* zjistí *ŘádekObjednávky* tak, že s podívá na objekt *Produkt*, který obsahuje a nechá si od něj vrátit *cenu*, tu pak vynásobí *množstvím produktů*. Takto *objednávka* spočte *základní cenu*, poté si ještě musí nechat vrátit *informace o výši slevy pro zákazníka*.

Tento scénář lze znázornit následujícím sekvenčním diagramem:



Obrázek SD1: *Sekvenční diagram výpočtu ceny objednávky*

Sekvenční diagram znázorňuje přehledně interakci mezi *účastníky scénáře* (instancemi tříd). Každý účastník je znázorněn pomocí „*životní čáry*“ (*lifeline*) směřující vertikálně odshora dolů. Časová posloupnost jde tedy také odshora dolů a posloupnost volání zpráv je velmi zřetelná. *Zaslání zprávy* objektu je označeno plnou čarou s šipkou ukazující na objekt, kterému je zpráva zaslána. Čára je dále označena *názvem zprávy/metody*. Aktivitu účastníka a její trvání označuje *aktivační obdélník* (*activation bar*) na životní čáře účastníka. Pokud účastník volá metodu na sobě samém, vznikne další obdélník aktivace, to vidíme na životní čáře objednávky. Jako odpověď na zaslání zprávy může

účastník vracet *odpověď*, což označíme přerušovanou čarou s šipkou směřující k účastníkovi, kterému je odpověď předávána. Nad přerušovanou čarou můžeme napsat *název* nebo *typ odpovědi*. Vidíme, že odpovědí na zprávu *vraťProdukt* zaslanou *řádku objednávky* je *produkt*, tedy *účastník*, kterému je posléze zaslána zpráva *vraťCenu*.

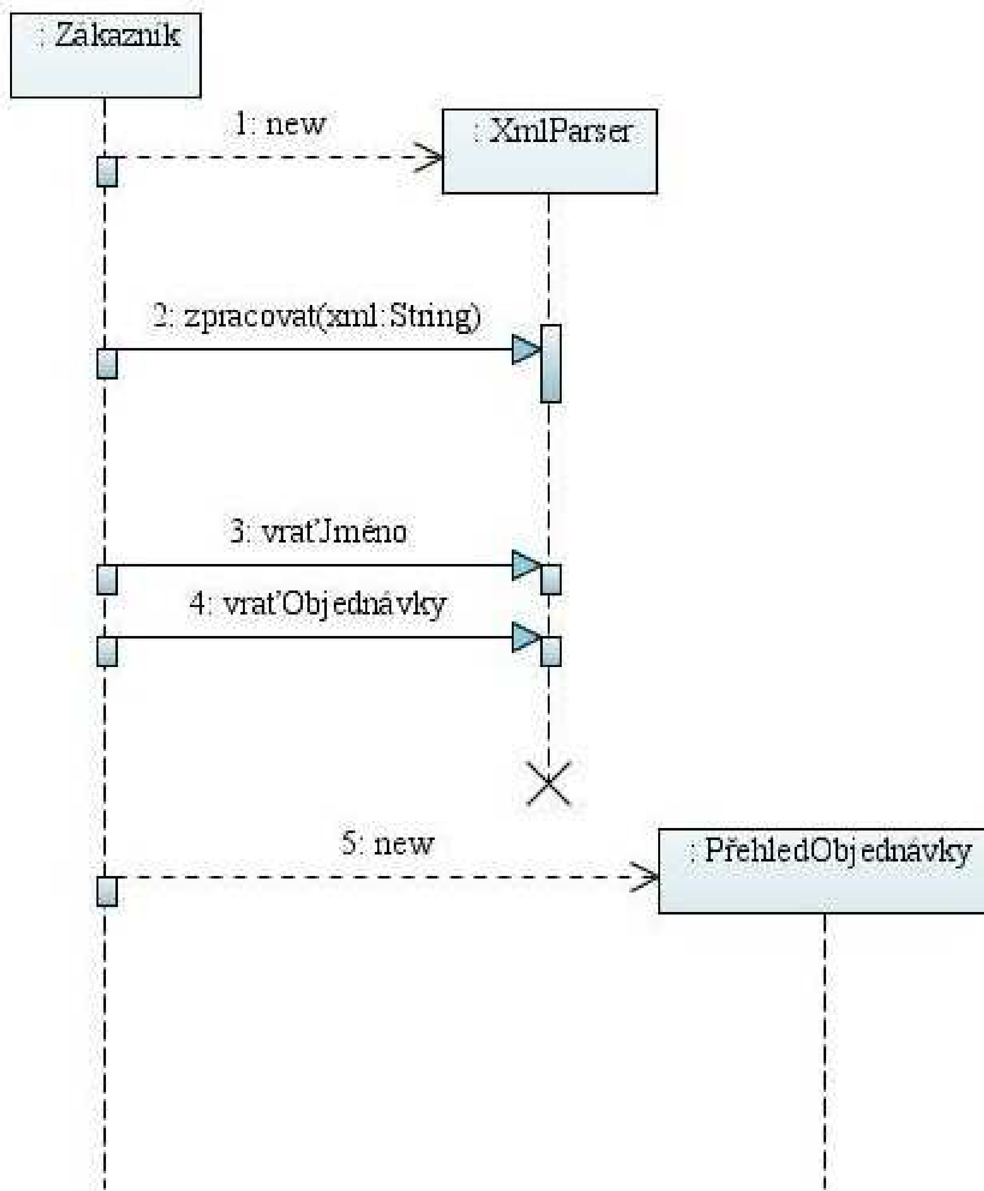
První zpráva *spočtiCenu* nemá (resp. má, ale z daného pohledu je anonymní) účastníka, který ji posílá, pochází z blíže neurčeného zdroje, je nazývána *found message*.

Opakem *found message* je tzv. *lost message*, tedy zpráva s anonymním cílem.

5.2 Vytváření a zánik účastníků

Účastník také může zaslat zprávu, která vyvolá vytvoření jiného účastníka. Příklad vidíme na následujícím obrázku. Účastník *Zákazník* má k dispozici data z externího systému ve formátu *xml*, k jeho zpracování potřebuje objekt třídy *XMLParser*, proto pošle zprávu *new*, která invokuje vytvoření nového objektu třídy *XmlParser* a poté mu zašle zprávu zpracuj a předá mu *xml* v řetězci. Zde si všimněte jak při zasílání zpráv mezi účastníky na sekvenčním diagramu předáváme parametry, v závorce za názvem zprávy uveden *název parametru* a *typ* oddělené dvojtečkou. Tento příklad je pouze demonstrativní, v praxi by byl *xml parser* zřejmě *singleton*, abychom ho nemuseli pokaždé znovu vytvářet.

Přestože ve většině moderních programovacích jazyků existují *garbage kolektory* a *finalizéry* zajištující zánik a uvolnění nepotřebných objektů z paměti, můžeme v sekvenčním diagramu označit zánik účastníka křížkem na životní čáře. Tak jsme to udělali i v našem příkladě u *xml parseru*, který v naší modelové situaci po zpracování *xml* již nebudeme potřebovat.



Obrázek SD2: *Sekvenční diagram znázorňující vytvoření a zánik objektu parseru*

5.3 Synchronní a asynchronní zprávy

Pokud zdroj posílá synchronní zprávu *(synchronous message)*, musí počkat, dokud nedoběhne metoda, kterou zasláním zprávy zavolal. Pokud posílá

asynchronní zprávu *(asynchronous message)*, čekat nemusí a případný výsledek obdrží někdy později. Asynchronní volání se často používají ve vícevláknových aplikacích a v middleware orientovaném na zasílání zpráv.

Synchronní volání je v UML 2 označeno plnou šipkou, zatímco asynchronní prázdnou.

5.4 Cykly a podmínky

Přestože sekvenční digramy se soustřeďují spíše na to, jak spolu objekty spolupracují, než na modelování řídící logiky obsahují notaci, která nám umožní řídící logiku zachytit. Touto notací jsou *interakční rámce (interaction frames).*

Interakční rámce vymezují určitou oblast sekvenčního diagramu, která může být rozdělena do jednoho či více *segmentů*. Každý interakční rámec má *operátor*, který určuje jeho význam a v každém segmentu může být *podmínka (guard)*, která určuje omezení pro interakci v tomto segmentu. Například pro cyklus použijeme interakční rámec s operátorem *loop* a v podmínce určíme množinu prvků, nad kterými budeme iterovat, pro podmínku typu *if – else* použijeme interakční rámec s operátorem *alt* a několika segmenty vyznačující větve s příslušnými podmínkami.

Dalšími často používanými operátory jsou:

- *opt* – segment je spuštěn, pokud je splněna podmínka, obdoba *alt* s jedním segmentem
- *par* – všechny segmenty jsou spouštěny paralelně
- *region* – kritická oblast; tento segment může být v danou chvíli spouštěn pouze jedním vláknem

- *ref* – reference; odkaz na interakci definovanou v jiném diagramu. V rámci jsou zahrnuty životní čáry, kterých se interakce týká
- *sd* – sekvenční diagram; takto můžeme označit celý sekvenční diagram.

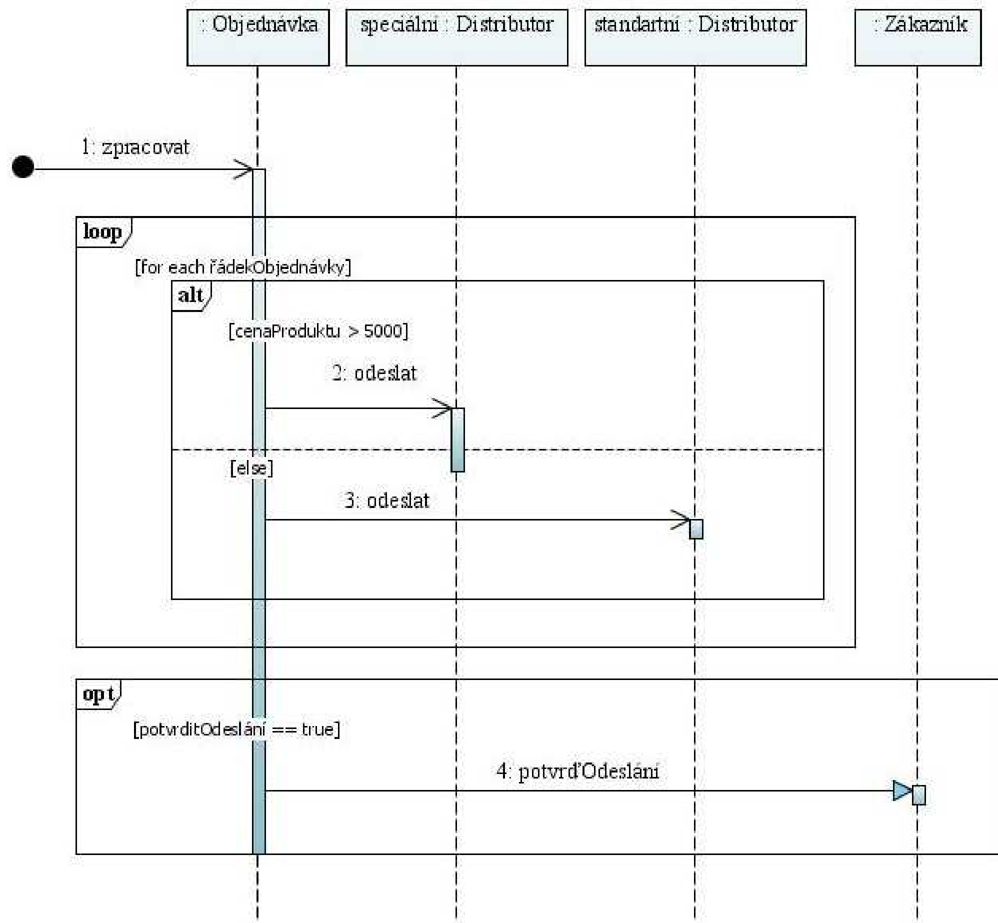
Sekvenční diagram na ***obrázku SD3*** představuje algoritmus pro odeslání objednaných produktů zákazníkovi, který můžeme vyjádřit pomocí pseudokódu takto:

```
zpracuj() {
  foreach(řádekObjednavky) {
    if(produkt.cena > 5000)
      speciální.odešli();
    else
      standartní.odešli();
  }
  if(potvrďitOdeslání == true)
   zákazník.potvrďOdeslání();
}
```

Slovně řečeno:

Po zavolání metody *zpracuj* nad objektem *objednávky* se projdou všechny *řádky objednávky*, pokud je *cena produktu* na daném řádku vyšší než 5000, bude *produkt* předán k doručení *specielnímu distributorovi* (který je např. sice dražší, ale zajistí nám bezpečné doručení), jinak bude produkt doručen *standardním distributorem*. Nakonec ještě ověříme, zda *zákazník* vyžaduje *potvrdit odeslání zboží* z objednávky, pokud ano, *potvrdíme odeslání*.

Všimněte si, že v tomto příkladu jsou zprávy zasílány asynchronně.



Obrázek SD3: *Sekvenční diagram s interakčními rámci*

5.5 Shrnutí sekvenčních diagramů

Sekvenční diagramy by měli být použity pro znázornění chování několika objektů v rámci jednoho případu užití. Jsou vhodné pro pohled na spolupráci mezi objekty, ale ne pro přesnou definici chování.

6 Případy užití *(use cases)*

Modelování případů užití představuje techniku získávání funkčních požadavků systému. Množina všech případů užití tvoří beze zbytku množinu všech funkčních požadavků na systém a tedy poskytuje ucelený pohled na funkčnost budoucího systému. Z důvodu přehlednosti a lepší spravovatelnosti se však případy užití nezachycují do jednoho diagramu, nýbrž se rozdělí do několika diagramů podle logického členění budoucího systému, například podle procesů probíhajících v systému. Takto členěný systém případů užití slouží později často jako základ pro napsání uživatelské příručky systému, kde každá část (diagram) je základem jedné kapitoly a také jako základ pro testovací scénáře.

6.1 Diagram případů užití

Diagram případů užití popisuje typické interakce mezi uživatelem a systémem nebo uvnitř systému a názorně tak ukazuje jak je systém používán.

Na diagram případů užití je možno pohlížet jako na grafický přehled případů užití systému. Diagram případů užití zobrazuje aktéry, případy užití vztahy mezi nimi:

- Jaký aktér spouští jaký případ užití
- Jaký případ užití obsahuje nebo rozšiřuje další případ užití

6.2 Dva pohledy na případ užití

Diagram případů užití obsahuje několik případů užití. Existují dva pohledy na případ užití:

- vnější pohled, který nazveme *prvek případu užití* chápeme jako jeden užitek systému
- vnitřní pohled, který nazveme *popis případu užití* slovně popisuje obsah případu užití

Aby byl případ užití jako prvek modelu kompletní, musí obsahovat oba pohledy.

6.3 Vnější pohled

Standard UML definuje pouze vnější pohled na případ užití.

Prvek případu užití se označuje pomocí elipsy, do níž je vepsán název prvku případu užití. Název se volí výstižný vzhledem k případu užití, zadává se jako podstatné jméno slovesné s několika přídavnými jmény a předměty resp. jako podstatné jméno vyjadřující nějakou činnost.

Na diagramu je systém nebo jeho část ohraničena pomocí obdélníku – *hranice systému (system boundary)*, abychom odlišili, co do systému patří a co se nachází mimo něj.

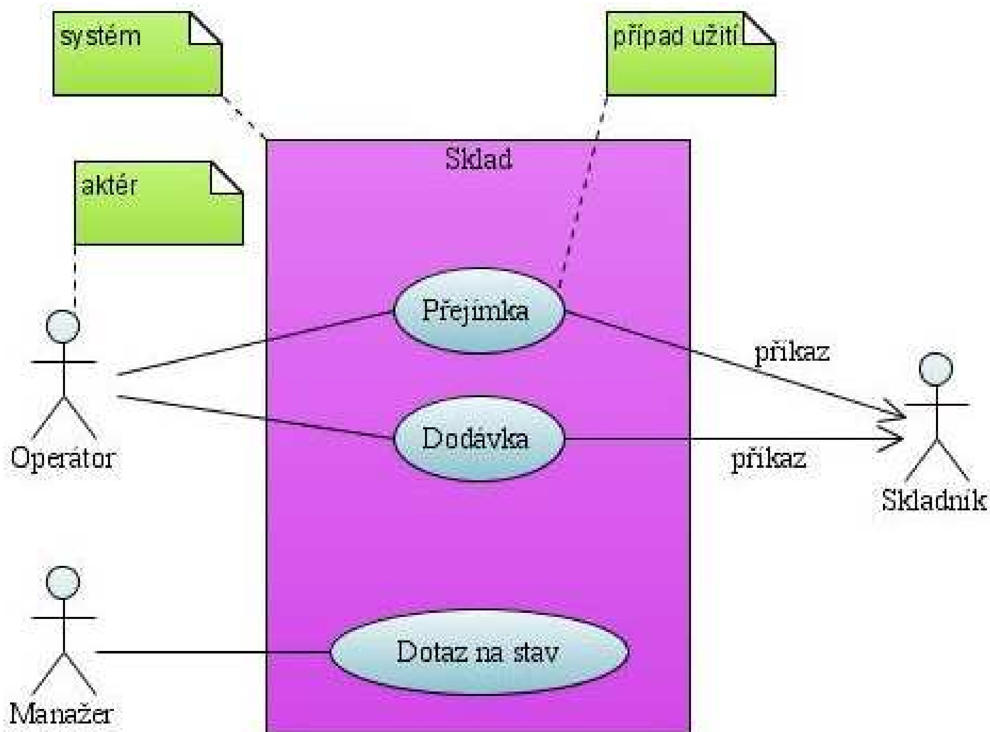
6.3.1 Aktéři

Vně systému se většinou nacházejí *aktéři (actors)*, kteří se systémem nějakým způsobem interagují – spouštějí jednotlivé prvky případu užití či reagují na jejích spuštění. Tato interakce je vyjádřena spojnicí mezi aktérem a prvkem případu užití.

6.3.2 Jednoduchý diagram

Na obrázku je znázorněn jednoduchý diagram případů užití, je zřejmé, že systém označený jako *sklad* plní tři funkce: *Přejímka* a *Dodávka* jsou

spouštěny aktérem *Operátor* a na jejich spuštění určitým způsobem reaguje aktér *Skladník*, *Dotaz na stav* je spouštěn aktérem *Manažer*.

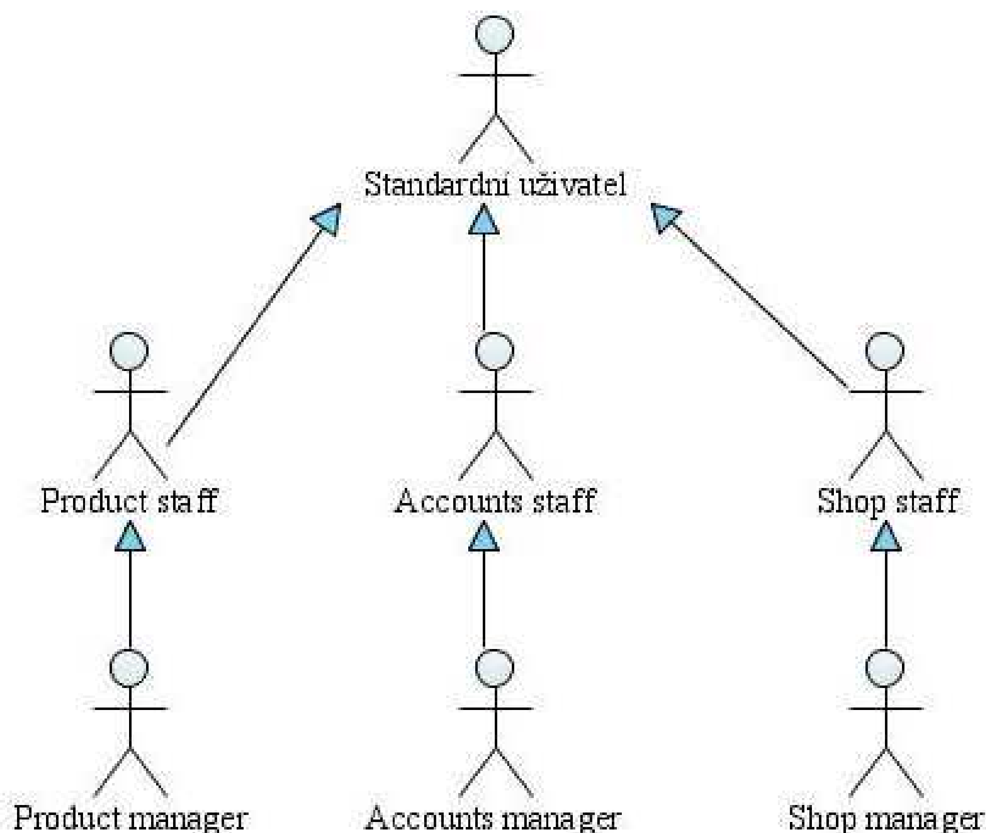


Obrázek UCD1: *Diagram případů užití znázorňující sklad*

6.3.3 Generalizace aktérů

Mezi aktéry může existovat hierarchická struktura realizovaná generalizací. Hierarchická struktura aktérů většinou odráží hierarchické uspořádání „rolí „ v podniku, který bude využívat budoucí systém. „Vyšší šarže“ má více pravomocí než šarže nižší, pokud je navíc aktér představující vyšší šarži potomkem aktéra s nižší šarží, zdědí všechna jeho práva, čímž získává přístup k případům užití – tedy k funkcionalitám systému, ke kterým mají přístup jeho předci.

Ve firmě prodávající letenky a mající několik poboček existuje následující hierarchie rolí zaměstnanců:



Obrázek UCD2: *Hierarchie aktérů*

Standardní uživatel je člověk na přepážce pobočky, který prodává letenky, *aktéři* na druhé úrovni dědičnosti, kromě toho, že obstarávají další činnosti (spravují produkty, účty a obchodní činnost v rámci své pobočky) mohou kdykoliv zastoupit standardního uživatele v prodeji letenek. V jejich činnosti je mohou nahradit jejich nadřízení ze třetí úrovně, avšak například *Product Manager* nemůže nahradit *Accounts Staff*, protože je z jiné větve a *Produkt Manager* nezdědil od *Accounts Staff* jeho práva.

6.3.4 Zahrnutí a rozšíření případu užití

Mezi případy užití mohou existovat závislosti, které se na diagramu případů užití označí přerušovanou čárou s šipkou označující směr závislosti.

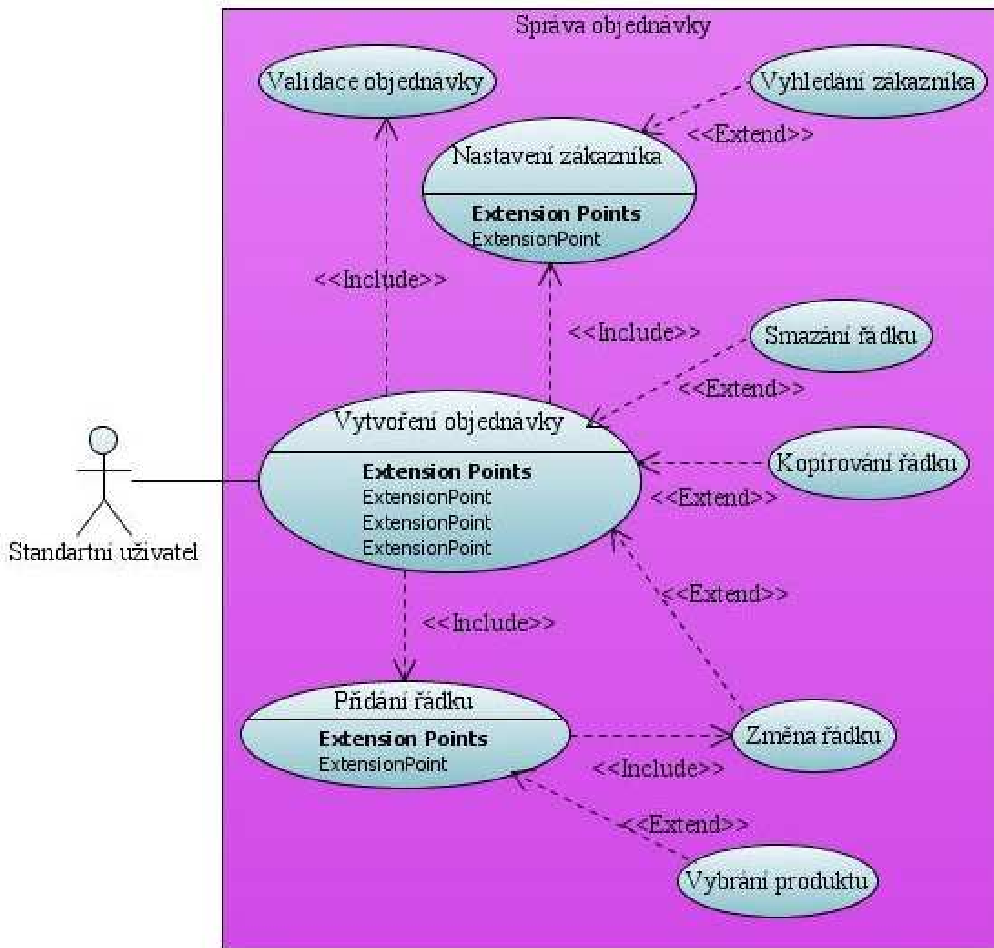
Existují dva druhy závislostí mezi případy užití:

6.3.4.1 Zahrnutí

Pokud jeden případ užití zahrnuje druhý, znamená to, že v rámci vykonávání prvního případu užití je v určitém okamžiku nutné vykonat druhý případ užití. Toto vykonání je povinné a nutné vykonávání prvního. Například případ užití „*Vytvoření objednávky*“ zahrnuje případ užití „*Nastavení zákazníka*“, protože nemůžeme vytvořit *objednávku*, která nikomu nepatří. Takovouto závislost označíme v diagramu stereotypem <<*include*>>.

6.3.4.2 Rozšíření

Jeden případ užití je rozšířen druhým, jestliže máme v rámci případu užití možnost spustit druhý případ užití. Například v případu užití „Vytvoření objednávky“ máme možnost kdykoliv smazat omylem přidaný řádek – tzn. spustí případ užití „Vymazání řádku objednávky“. Avšak spuštění rozšiřujícího případu užití není povinné. Takovouto závislost označíme v diagramu stereotypem <<*extend*>>. Všimněte si opačného směru závislosti. Bázový případ užití poskytuje určitou množinu bodů rozšíření *(extension points)*, k nimž lze připojit rozšíření v podobě nového chování.



Obrázek UCD3: *Diagram případů užití znázorňující vztahy mezi případy užití*

6.4 Slovní popis případu užití

Diagram případu užití může být užitečný, nedozvíme se z něj však důležité podrobnosti, ty jsou právě obsaženy v jeho popisu. Popis případu užití tvoří vnitřní pohled na případ užití, specifikace UML však neurčuje syntaxi popisu případu užití, proto neexistuje standardní způsob vyjádření popisu případu užití, v různých případech můžeme podle potřeby použít různé formáty zápisu..

Popis případu užití se skládá z jednoho či více *scénářů případu užití (use case scenario).* Jeden z těchto scénářů určíme jako hlavní. Hlavní scénář popisuje standardní průchod případem užití, který vede k úspěšnému splnění cíle případu užití. Mohou však existovat alternativní scénáře, které mohou nastat při průchodu hlavním scénářem, například jestliže není splněna požadovaná podmínka.

Objednání produktu

Hlavní scénář:

1. Zákazník prohlíží katalog a vybere si produkty
2. Zákazník zkontroluje a potvrdí výběr
3. Zákazník vyplní potřebné informace (adresu, způsob doručení)
4. Systém zobrazí podrobnosti o objednávce, včetně způsobu doručení a konečné ceny
5. Zákazník vyplní informace o své kreditní kartě
6. Systém autorizuje platbu
7. Systém potvrdí objednávku
8. Systém odešle zákazníkovi potvrzující e-mail

Rozšíření:

3a: Zákazník je stálým zákazníkem
.1: Systém zobrazí stávající nastavení zákazníka - adresu, platební informace a způsob doručení
.2: Zákazník může potvrdit nebo přepsat tyto hodnoty, vrací se do Hlavního scénáře, bod 6
6a: Systému se nepodaří autorizovat platbu
.1: Zákazník může znovu zadat informace o kreditní kartě nebo objednávku stornovat

6.5 Shrnutí případů užití

Případy užití přestavují hodnotný nástroj, který nám umožňuje pochopit funkční požadavky systému. Tvorba případů užití tedy většinou spadá již do raných fází vývoje systému – *sběru požadavků*. Kompletní množina případů užití představuje souhrn všech funkčních požadavků na budoucí systém a dá se proto s výhodou využít k odhadu časové náročnosti a pracnosti vývoje systému. Kromě funkčních požadavků je však nutné také analyzovat požadavky nefunkční, těmi mohou být např. požadavky na použitou architekturu, zabezpečení systému, nároky na výkonnost, či rychlost odezev a další.

Je důležité si uvědomit, že případy užití reprezentují externí pohled na systém, což mimo jiné znamená, že neexistují přímé souvislosti mezi diagramy případů užití a diagramy tříd.

Ačkoliv specifikace UML neříká nic o textovém popisu případů užití, tento popis obsahuje všechny důležité informace a proto je daleko důležitější než samotný diagram a je nutné věnovat mu patřičnou pozornost.

Případy užití by měli být především stručné a jasné. Složité popisy a diagramy se špatně studují a rapidně se zvyšuje riziko, že je čtenář nepochopí.

7 Stavové diagramy *(state machine diagrams)*

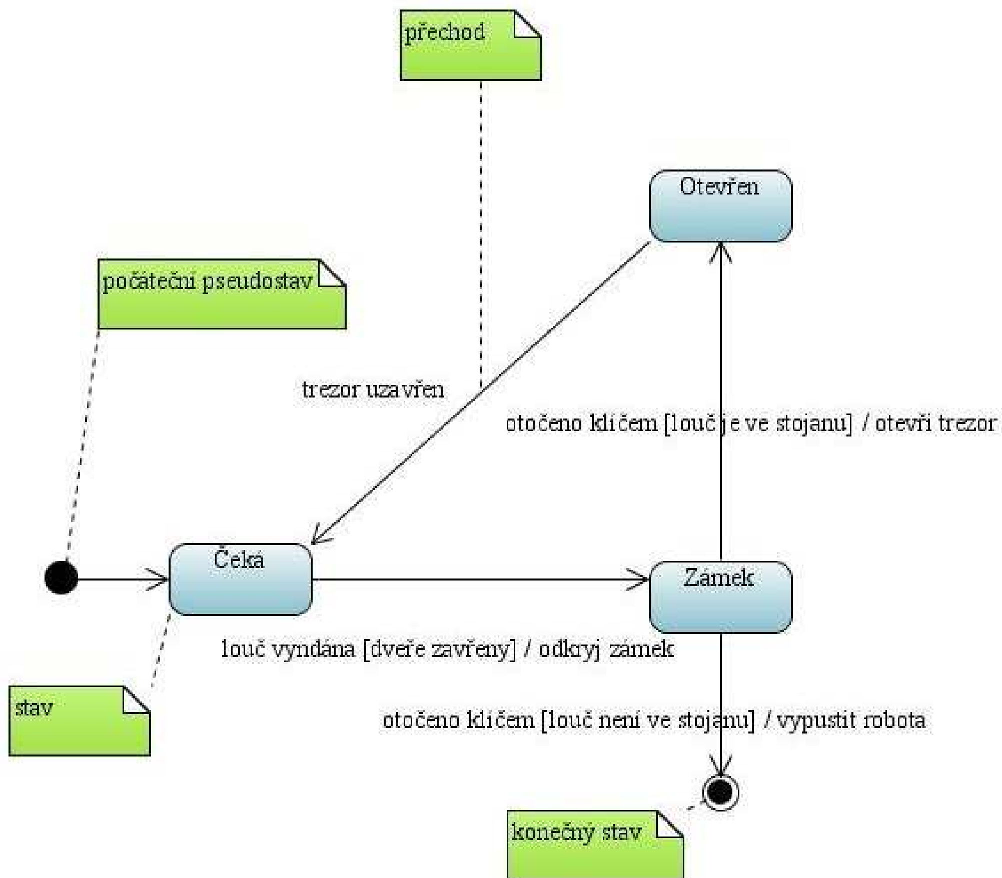
Stavové diagramy představují rozšířenou techniku popisu chování systému. Různé formy stavových diagramů se objevily již v 60. letech 20. století a až do vzniku UML byly přizpůsobovány potřebám daných objektově orientovaných technik, které je používaly. V objektově orientovaném přístupu jsou stavové diagramy používány k zobrazení chování objektu během jeho existence, napříč případy užití.

7.1 Příklad stavového diagramu

Pro seznámení se základy stavových diagramů uvažujme tento, trochu pohádkový příklad:

Ve starém hradě je místnost a v ní trezor s pokladem, který si jeho majitel chrání důmyslným mechanismem proti zlodějům. K otevření trezoru je potřeba vložit klíč do zámku a otočit jím, zámek je však ukryt ve stěně a odkryje se teprve když ze stojanu na zdi vyndáme louč a to ještě jen v případě, že jsou dveře do místnosti zavřené. Poté se objeví zámek, do kterého je možné vložit klíč a odemknout, ovšem pozor, ještě předtím je potřeba vložit louč zpátky do stojanu. Pokud se tak nestane a případný zloděj se pokusí odemknout, místo odemčení trezoru se uvolní tajné dveře a do místnosti vtrhne strážící robot a zloděje zneškodní.

Předpokládejme, že celý tento mechanismus je spravován jakýmsi „*kontrolerem*“, který monitoruje stav v místnosti a v závislosti na něm provádí popsané akce. Chování *kontroleru* je možné znázornit následujícím stavovým diagramem.



Obrázek SMD1: *Stavový diagram kontroler bezpečnostního mechanismu*

Náš *kontroler* se během své existence může nacházet v jednom ze tří stavů: *Čeká*, *Zámek* nebo *Otevřen*. Stavový diagram začíná stavem, ve kterém se nachází *kontroler*, poté co je vytvořen, což je v našem případě stav *Čeká*, tuto skutečnost vyjádříme v diagramu tzv. *počátečním pseudostavem* – černá tečka s šipkou ukazující do počátečního stavu. Dále jsou v diagramu znázorněny pravidla, podle kterých objekt *kontroleru* přechází mezi jednotlivými stavy formou *přechodů (transitions)* – spojnice stavů. Přechod indikuje změnu z jednoho stavu do druhého ve směru šipky. Každý přechod má popisek ve tvaru ***událost (parametry)[podmínka]/akce^zpráva***, přičemž všechny části jsou volitelné.

- *událost (trigger)* je většinou jednoduchá událost, která způsobí změnu stavu.
- *podmínka (guard)* je booleovská podmínka, která musí mít hodnotu *true*, aby mohla být změna stavu uskutečněna.
- *akce (activity)* je akce, která je přechodem mezi stavy spuštěna.

Konečný stav (černá tečka v kroužku) značí, že stavový diagram je „doběhl“, tzn. objekt zanikl. V našem příkladě tento stav nastává, když někdo odemkne zámek, aniž by předtím vrátil louč do stojanu – vyběhne robot a zloděje zneškodní, poté je třeba vše vrátit do původního stavu: odklidit tělo zloděje, připravit robota na své místo a znovu spustit systém.

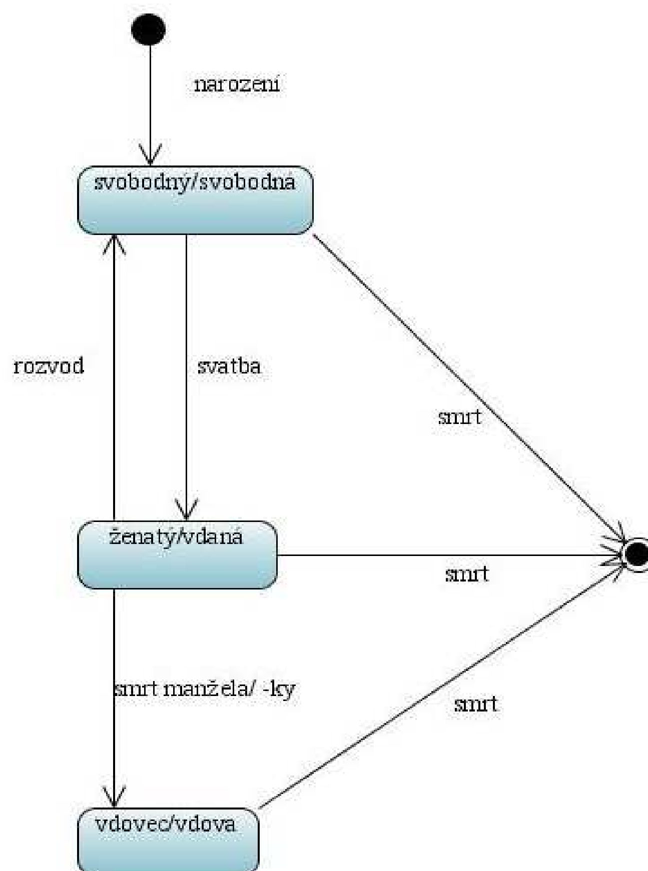
7.2 Stav objektu v programu

V programu je stav objektu dán kombinací hodnot jeho instančních proměnných, to mimo jiné znamená, že u objektu třídy, která nemá žádné instanční proměnné nemůžeme určit stav, tento objekt je „bezstavový“. Samozřejmě bychom mohli jakoukoliv kombinaci hodnot instančních proměnných „stavového“ objektu označit jako samostatný stav, to však nemá z pohledu systému žádný význam, proto je užitečné rozlišovat pouze stavy, v nichž objekt reaguje rozdílně na stejné události a které jsou zároveň užitečné pro náš systém. Náš *kontroler* například nesleduje, zda je v trezoru něco uloženo, nebo je prázdný.

7.3 Realita versus informační systém

Při modelování budoucího informačního systému vycházíme často z reálné skutečnosti, kterou má náš systém simulovat. Představme si, že máme vytvořit informační systém například pro fiktivní sociální instituci. Tento systém bude

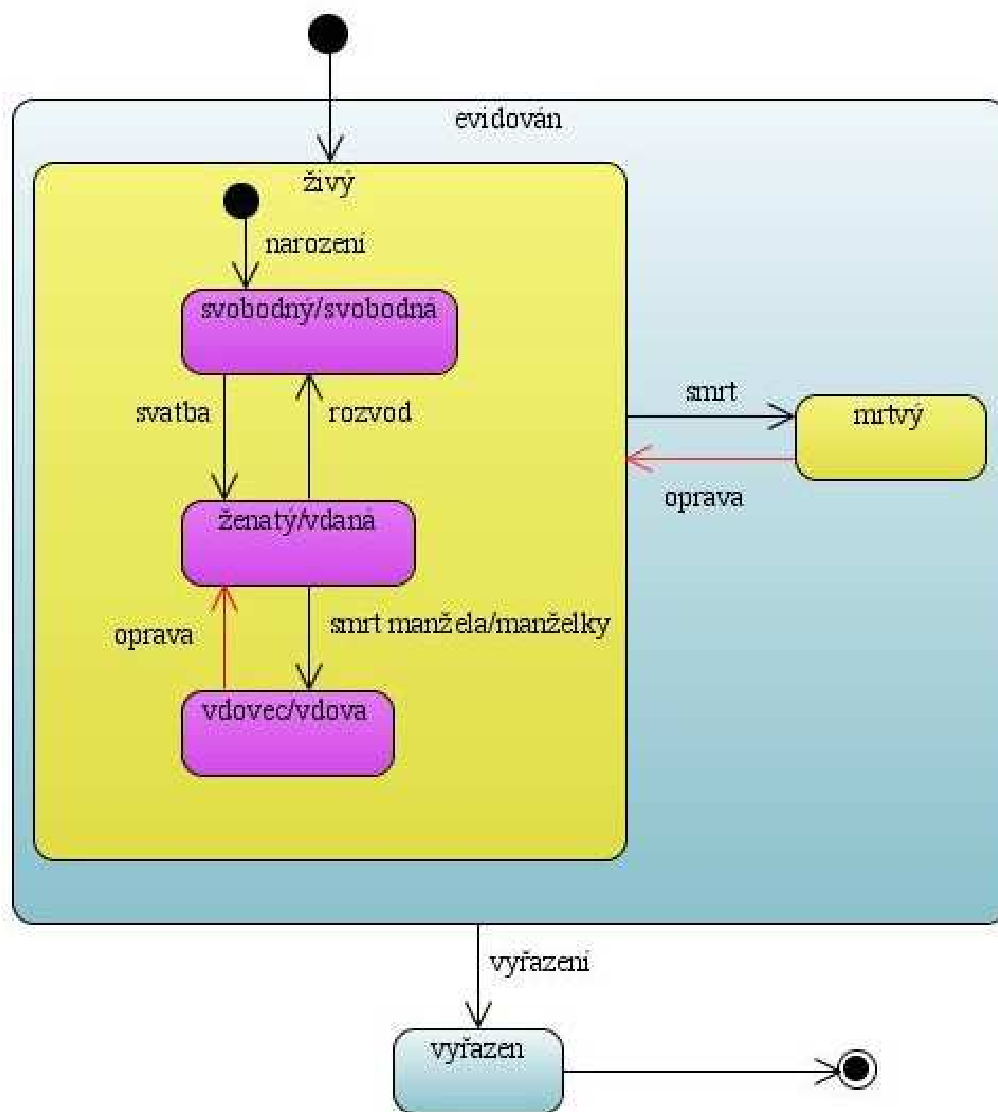
evidovat osoby, jimž budou vypláceny sociální dávky. Jejich výše bude záviset na tom v jakém „rodinném stavu“ se osoba nachází. Celá situace je znázorněna následujícím stavovým diagramem.



Obrázek SMD2: *Stavový diagram „rodinných stavů“*

Tento stavový diagram zachycuje životní cyklus osoby v reálné skutečnosti, představme si však situaci, kdy jistý pan Novák volá do instituce a sděluje:“Již třetí měsíc pobírám vdovecký důchod, přestože moje manželka je živa a zdráva“. K podobným chybám způsobených většinou lidským faktorem může dojít v každém systému. Když vyváříme model budoucího informačního systému, nesmíme zapomenout přemýšlet nad rámec modelované skutečnosti a postihnout skutečné rysy systému (to se týká všech pohledů na systém, ne

pouze stavového diagramu). V našem příkladě je tedy nezbytné doplnit do diagramu přechody, které v realitě neexistují, abychom zajistili potřebnou funkčnost.



Obrázek SMD3: *Stavový diagram sociálního systému*

7.4 Implementace stavového diagramu

Existují tři hlavní způsoby implementace stavového diagramu:

7.4.1 Implementace pomocí vnořených přepínačů

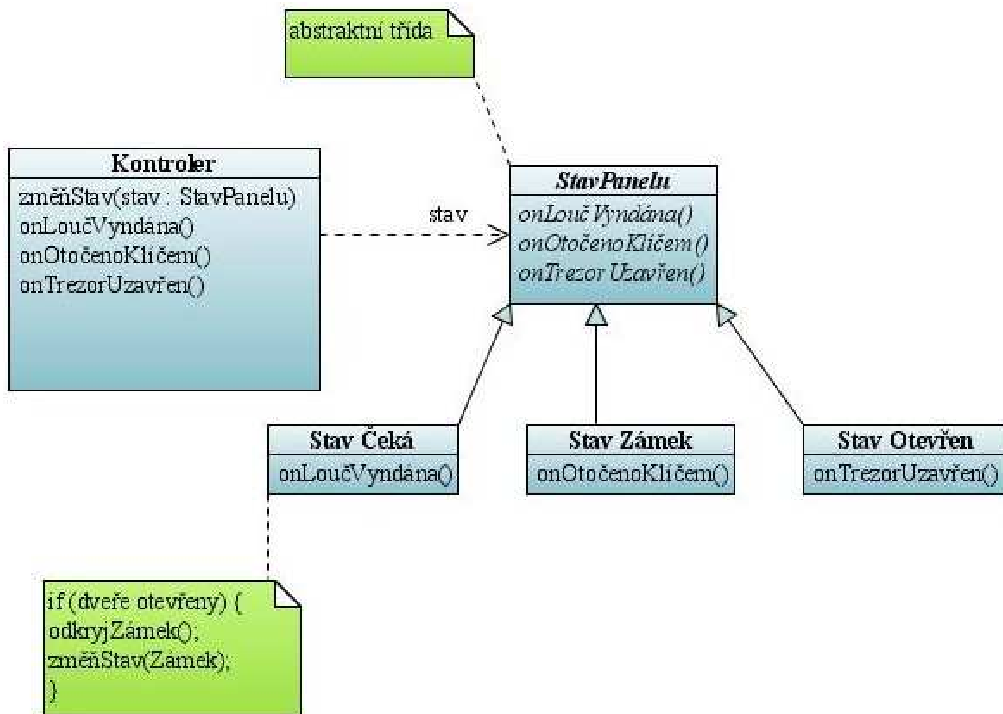
Tento způsob implementace stavového diagramu je nejpřímočařejší, v našem příkladě by mohla implementace vypadat například takto:

```
public void zpracujUdalost(UdalostPanelu udalost) {
    switch (soucasnyStav) {
      case StavPanelu.OTEVREN:
        switch (udalost) {
          case udalostPanelu.TREZOR_UZAVREN:
            soucasnyStav = StavPanelu.CEKA;
            break;
         }
         break;
      case StavPanelu.CEKA:
        switch (udalost) {
          case udalostPanelu.LOUC_VYNDANA:
            if (dvereOtevreny) {
              odkryjZamek();
              soucasnyStav = StavPanelu.ZAMEK;
            }
            break;
        }
        break;
      case StavPanelu.ZAMEK:
        switch (udalost) {
          case udalostPanelu.OTOCENO_KLICEM:
            if (loucVeStojanu) {
              otevriTrezor();
              soucasnyStav = stavPanelu.OTEVREN;
            }
            else {
              uvolniRobota();
              sucasnyStav = stavPanelu.KONEC;
            }
            break;
        }
        break;
    }
  }
```

Jak je vidět, má tento způsob implementace jednu velkou nevýhodu – i pro jednoduché stavové diagramy může být značně komplikovaný. Při jeho použití se nám řízení objektu může snadno vymknout kontrole, proto je lepší se mu ve většině případů vyhnout.

7.4.2 Implementace pomocí návrhového vzoru Stav

Návrhový vzor Stav (State) je jedním se základních návrhových vzorů. U třídy, jejíž objekty se mohou nacházet v různých stavech, které se liší svými reakcemi na zasílané zprávy (*kontroler* v našem příkladě), vytvoříme speciální atribut, který bude zastupovat část objektu, v níž jsou soustředěny stavové závislé vlastnosti objektů. Tento atribut, představující obecný stav, specifikujeme jako abstraktní třídu, či rozhraní, ve kterém deklarujeme metody odpovídající zprávám, na něž reagují instance vícestavové třídy rozdílně v závislosti na svém stavu. Pro každý stav definujeme speciální jednostavovou třídu jako potomka obecného stavu, který bude implementovat příslušné metody. Metody specifikující změnu stavu budou vracet odkaz na instanci jiné stavové třídy. Diagram tříd představující model *návrhového vzoru Stav* vidíme na následujícím obrázku.



Obrázek ClD6: *Diagram tříd znázorňující návrhový vzor Stav*

7.4.3 Implementace pomocí stavové tabulky

Při tomto přístupu vyjádříme informace stavového diagramu jako data, stejně jako v tabulce, která uložíme například v databázi či souboru. Poté je nutné vytvořit buď „interpret“, který pracuje s daty z tabulky za běhu, nebo generátor kódu, který vygeneruje na základě tabulky příslušné třídy.

Zdrojový stav	Cílový stav	Událost	Podmínka	Akce
Čeká	Zámek	Louč vyndána	Dveře zavřeny	Odkryj zámek
Zámek	Otevřen	Otočeno klíčem	Louč je ve stojanu	Otevřít trezor
Zámek	Konečný stav	Otočeno klíčem	Louč není ve stojanu	Vypustit robota
Otevřen	Čeká	Trezor uzavřen		

Tabulka SMD1: *Tabulka pro implementaci stavového diagramu*

Implementace pomocí stavové tabulky je sice pracnější, ale jakmile je jednou provedena, můžeme ji použít i v budoucnu. Implementace návrhového vzoru Stav je většinou snadnější, s malým množstvím kódu v každé třídě. V každém případě je ruční implementace stavového diagramu časově náročnější, proto je většinou nejvýhodnější využít schopnost generování kódu CASE nástroje.

7.5 Shrnutí stavových diagramů

Stavové diagramy jsou vhodné pro popis chování jednoho objektu napříč případy užití, příliš se však nehodí pro popis chování více spolupracujících

objektů. Pro tyto případy je vhodné kombinovat stavové diagramy s dalšími technikami.

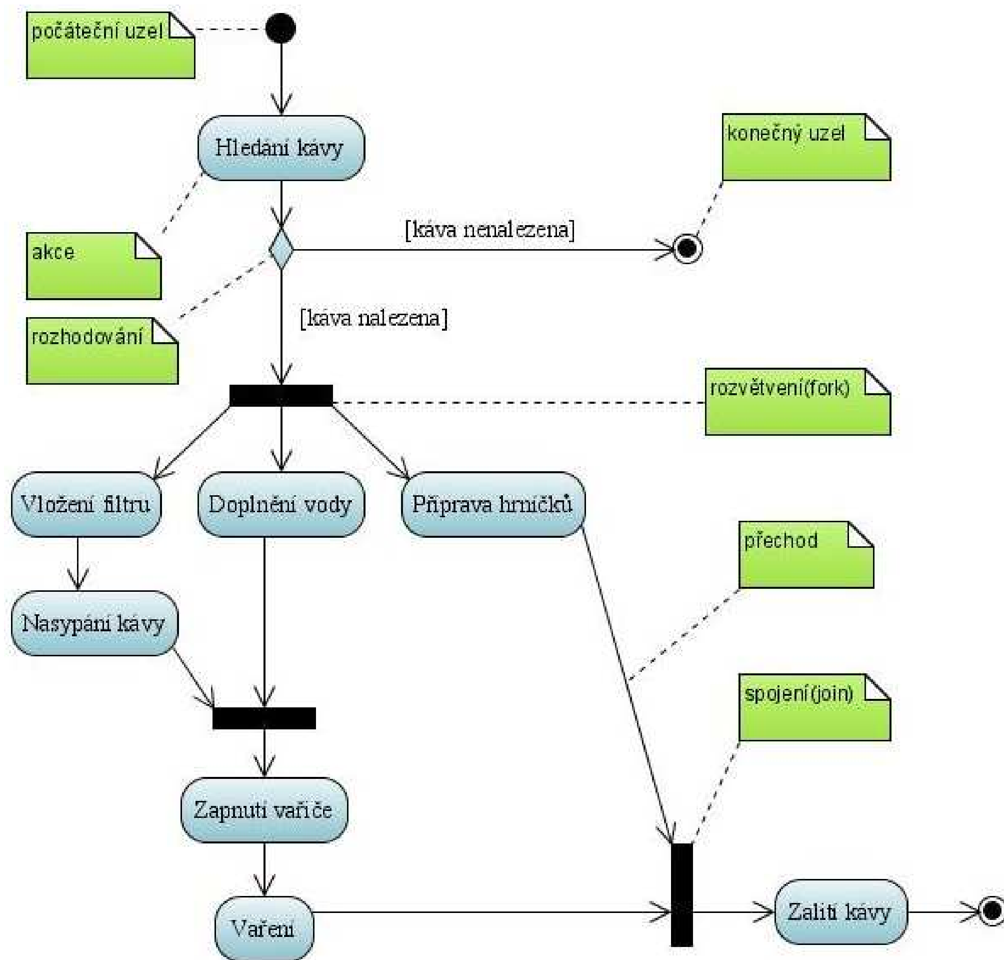
Při používání stavového diagramu se je nesnažte kreslit pro každou třídu v systému, většinou je to ztráta času. Vytvářejte stavové diagramy pouze pro ty třídy, které vykazují zajímavé chování, které může stavový diagram pomoci dobře vysvětlit.

8 Diagramy aktivity *(activity diagrams)*

Diagramy aktivity představují techniku k popsání procedurální logiky, business procesu nebo pracovního toku *(workflow)*. Tato technika se velmi podobá vývojovým diagramům *(flowcharts)*, které se používali před vznikem UML, hlavní rozdíl je v tom, že diagramy aktivity umožňují zobrazit paralelní chování.

8.1 Příprava kávy

Možnosti diagramu aktivity si ukážeme na postupu přípravy kávy. Diagram aktivity pro tuto činnost by mohl vypadat například takto:



Obrázek AD1: *Diagram aktivity znázorňující postup přípravy kávy*

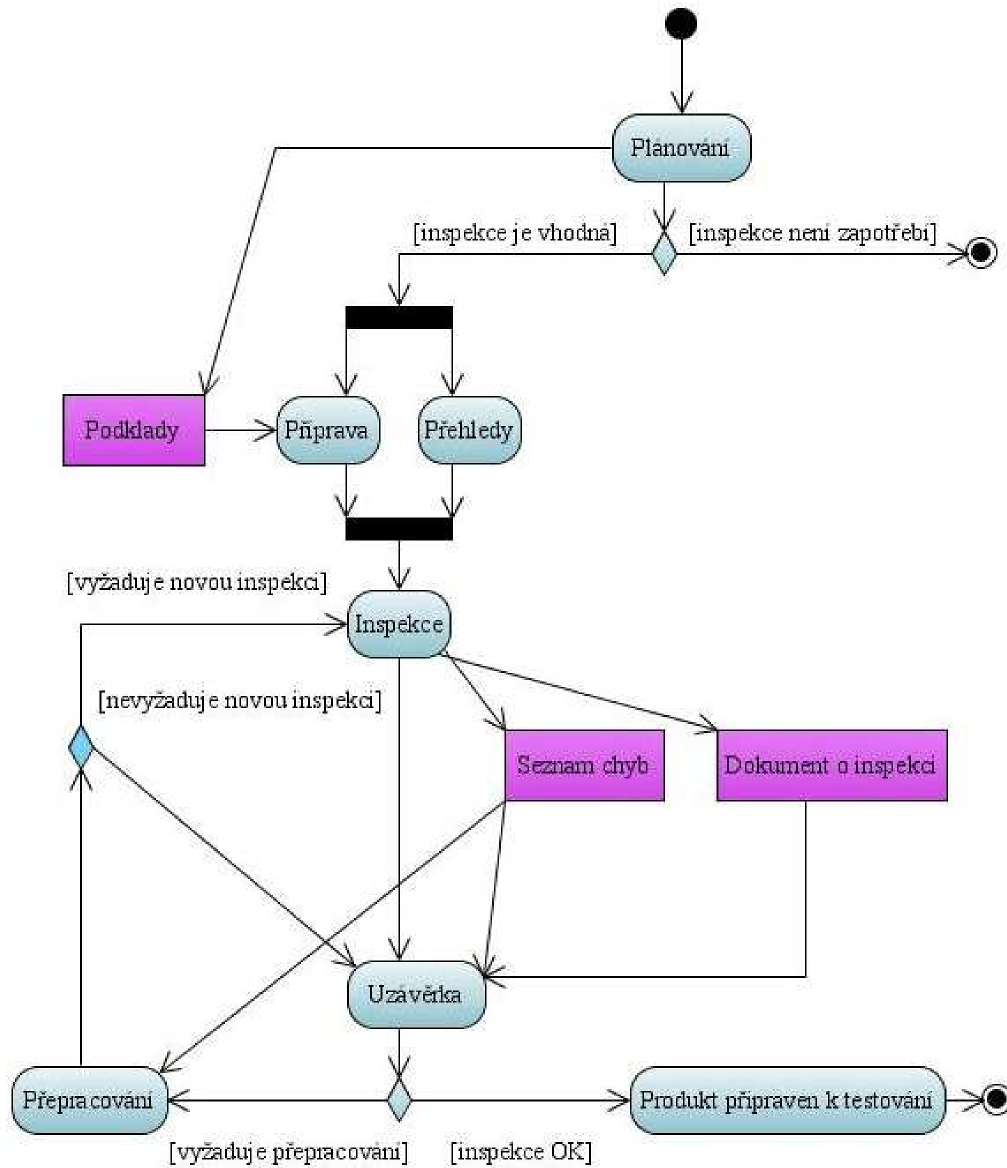
Začíná v *počátečním uzlu (initial node)* a poté následuje přechod na první *akci Hledání kávy*. Po této akci vstupujeme do tzv. *rozhodování (decision)*, z něhož vystupuje několik přechodů, následuje ten přechod, u něhož je splněna podmínka uvedená nad ním v hranatých závorkách, je zřejmé, že by vždy měla být splněna právě jedna podmínka, můžeme také použít klíčové slovo *[else]*, pro případ, že nebude splněna žádná z ostatních podmínek. Pokud je splněna podmínka *[káva nenalezena]*, jinými slovy, nepodařilo se nám během akce *Hledání kávy* kávu nalézt, vstupujeme do konečného uzlu a aktivita zde končí. Pokud kávu nalezneme vstupujeme do tzv. *rozvětvení (fork)*. Rozvětvení má

jeden vstupující tok a několik vystupujících toků, které probíhají paralelně, akce v souběžných tocích mohou tedy probíhat v navzájem libovolném pořadí či současně. Můžeme tak například nejprve připravit filtr s kávou a poté doplnit vodu či naopak, dokonce můžeme jednou rukou připravovat filtr s vodou, zatímco druhou rukou doplňujeme vodu. Diagram aktivity umožňuje tomu, kdo provádí proces, vybrat pořadí. To je velmi důležité pro procesní modelování, protože procesy často probíhají paralelně. Také je to důležité pro konkurenční algoritmy, ve kterých provádějí nezávislá vlákna úlohy paralelně.

Paralelní toky je nutno v určitém okamžiku synchronizovat, což zajistíme vložením prvku *spojení (join)*, do spojení vstupuje několik toků avšak vystupuje z něj pouze jeden. Synchronizace může nastat až v okamžiku splnění všech akcí ze vstupujících toků. Jak jste si jistě povšimli, uzly na diagramu aktivity jsou nazývány *„akce“*, *aktivita* je celý proces znázorněný na diagramu aktivity a skládá se z jednotlivých akcí.

8.2 Audit

Na dalším obrázku je znázorněn diagram aktivity představující proces auditu. Pokud během procesu vznikají či do něho vstupují instance nějakých tříd, je možné je znázornit symbolem obdélníku. Pokud vede šipka směrem z akce do objektu, znamená to, že objekt vznikne jako produkt dané akce, pokud vede šipka naopak, znamená to, že objekt vstupuje jako parametr do akce.

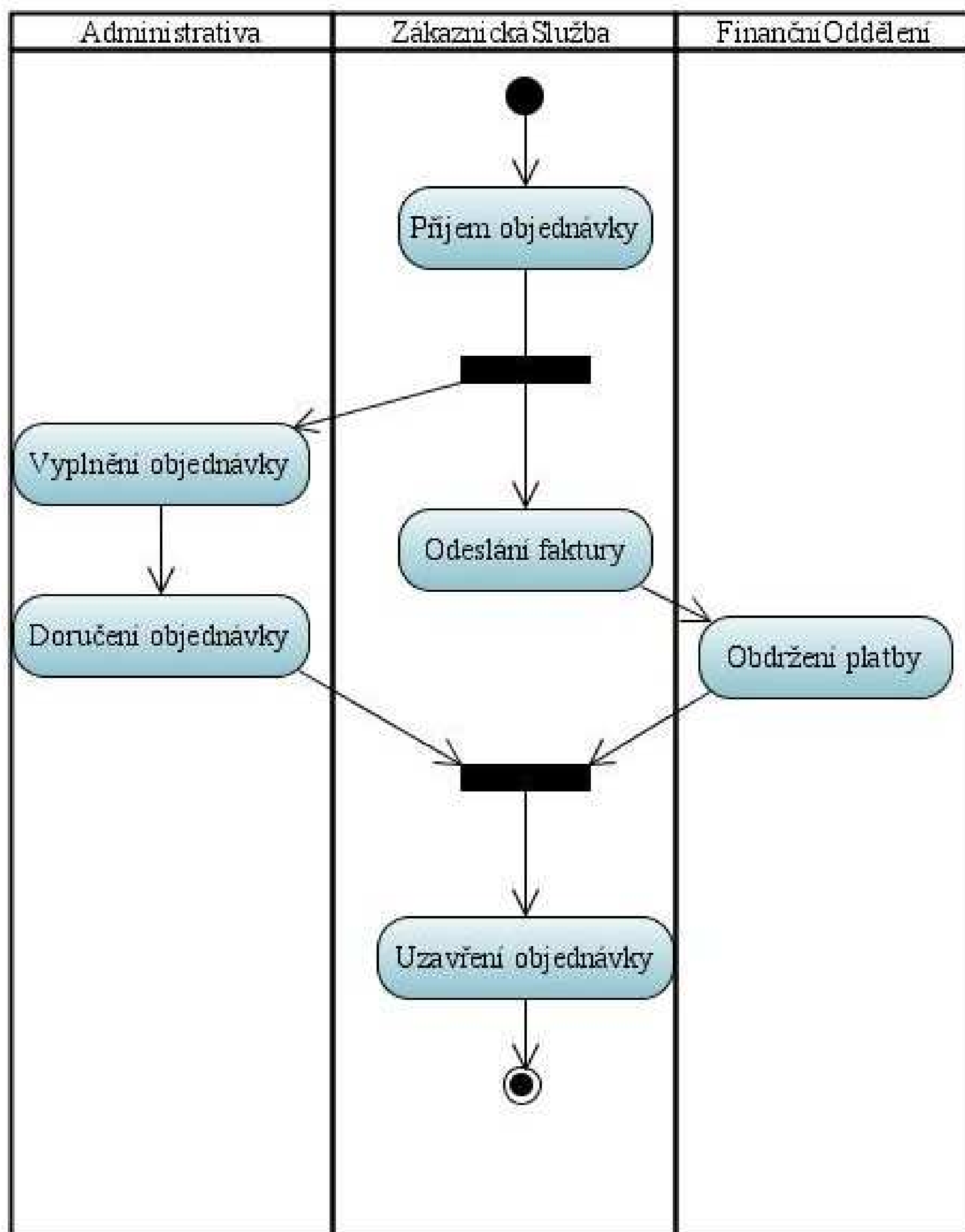


Obrázek AD2: *Diagram aktivity znázorňující postup auditu*

8.3 Zóny (partitions)

Diagram aktivity nám sice říká co se děje, avšak ne, kdo co dělá. Z programátorského hlediska tedy není jasné, která třída je zodpovědná za kterou akci. V modelování business procesů není jasné, která část organizace

se stará o kterou akci. Pro tyto účely UML nabízí tzv. *zóny (partitions)*, příslušné akce pak podle zodpovědnosti zakreslujeme do zóny dané třídy či organizační jednotky, jak vidíme na následujícím obrázku.



Obrázek AD3: *Diagram aktivity demonstrující použití zón*

8.4 Shrnutí diagramů aktivity

Síla diagramů aktivit spočívá v tom, že umožňují modelovat paralelní chování. To z nich dělá velice užitečný nástroj pro popis pracovních toků a procesního modelování. Můžeme je také použít jako vývojový diagram. Diagramy aktivit jsou také používány k popisu případu užití, potom je ale nutné dbát na to, aby byly jednoduché a srozumitelné pro doménového experta, lepším řešením popisu případu užití bývá textová forma.

9 Diagramy balíčků *(package diagrams)*

Třídy představují základní formu strukturování objektově orientovaného systému. Přestože jsou velice užitečné potřebujeme něco víc pro strukturování velkých systémů, které mohou mít i stovky tříd.

9.1 Balíček

Balíček (package) je sdružující prvek, který umožňuje sloučit elementy jakéhokoliv konstruktu UML do (z pohledu úrovně) vyšší logické jednotky. Tímto způsobem můžeme dekomponovat rozsáhlý systém na menší celky, touto dekompozicí můžeme docílit toho, že bude snadnější systému rozumět, udržovat jej a dělat v něm změny. Nejčastěji se balíčky používají ke sdružování tříd, ale můžeme je použít na jakékoliv elementy, například případy užití, které spolu logicky souvisí, protože se týkají určité stejné části systému můžeme sloučit do jednoho balíčku. Dále se budeme věnovat seskupení tříd do balíčků.

V UML modelu patří každá třída do nějakého balíčku, název třídy v rámci balíčku i její výskyt v daném balíčku musí být jedinečný, což znamená, že třída, která se vyskytuje v daném balíčku se nesmí vyskytovat v jiném balíčku, jiný balíček však může obsahovat jinou třídu se stejným názvem. Pokud chceme třídu jednoznačně identifikovat, musíme uvést její *plně kvalifikovaný název*, který se skládá z názvu balíčku (případně celé struktury balíčků),ve kterém je třída umístěna, a názvu třídy.

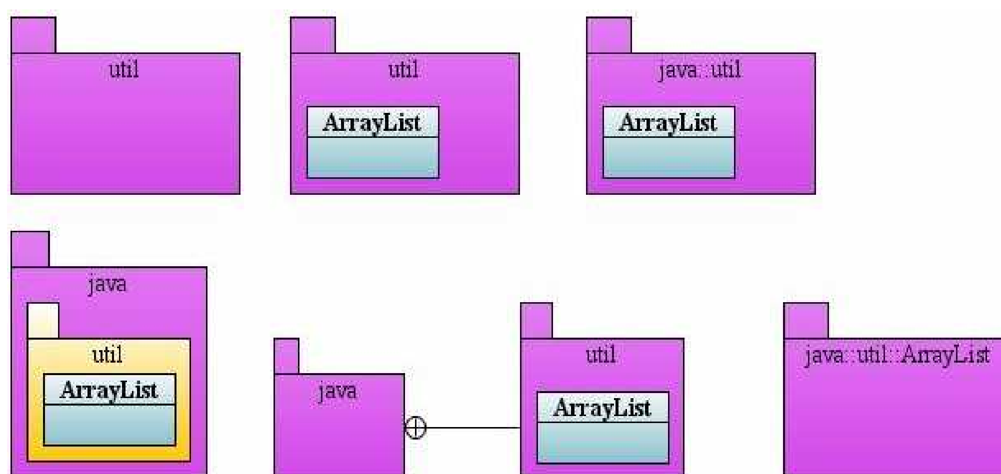
Balíčky obsahují třídy nebo další balíčky, čímž se vytváří hierarchická struktura, v níž se struktura balíčku na nejvyšší úrovni jakoby rozpadá do pod-balíčků.

Balíčky odpovídají programovým konstruktům „balíčky“ *(packages)* v Javě a „*jmenné prostory*“ *(namespaces)* v C++ a .NET.

9.2 Zobrazení balíčku

Pro zobrazení balíčku používá UML symbol obdélníku s malým obdélníčkem vlevo nahoře. Název balíčku je může být buď relativní, tzn., že zapíšeme pouze název balíčku, aniž bychom dále specifikovali, zda je tento balíček *podbalíčkem* jiného balíčku, či plně kvalifikovaný, potom udává jednoznačný název balíčku v hierarchii.

Třídy či podbalíčky obsažené v balíčku do něj můžeme zakreslit přímo, nebo připojit plnou čarou s přeškrtnutým kroužkem na straně vlastníka.



Obrázek PD1: *Možnosti zobrazení balíčku*

Do jednoho balíčku seskupujeme třídy, které spolu logicky souvisí a komunikují mezi sebou. Tyto třídy jsou pak schopny poskytovat ucelený soubor služeb.

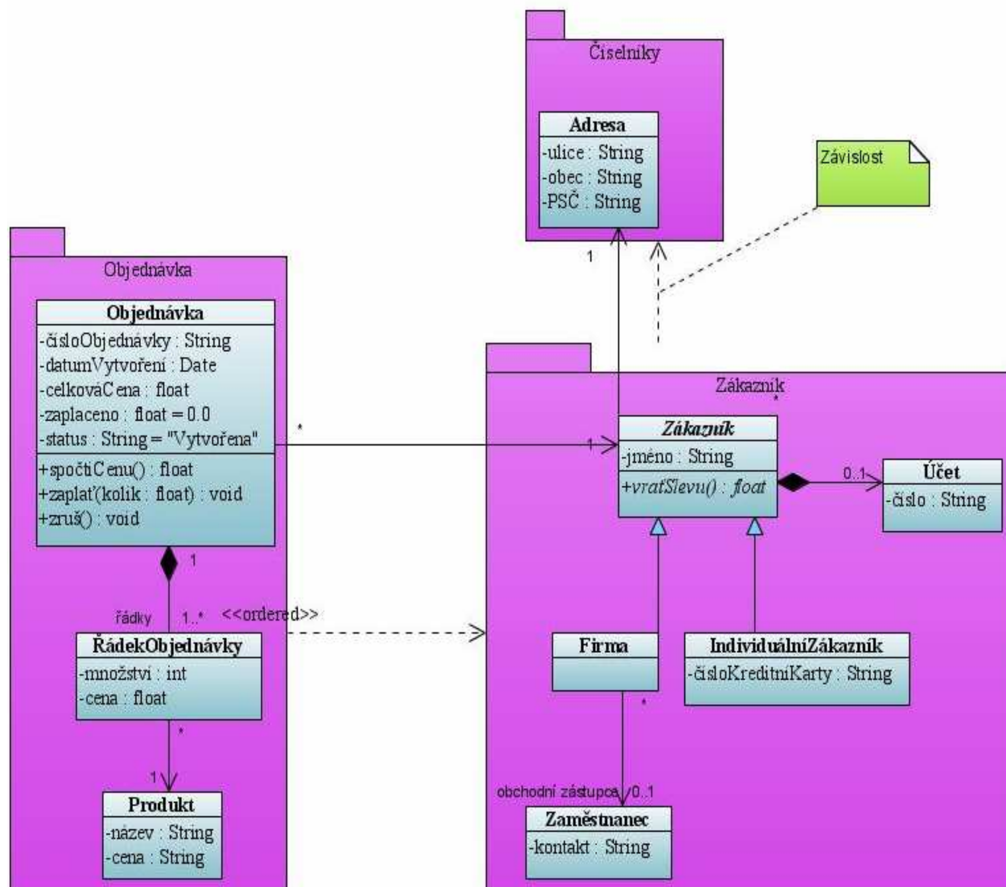
9.3 Závislosti

Mezi dvěma elementy existují závislosti, pokud změna v definici jednoho elementu vyvolá změnu v definici druhého elementu. Mezi třídami existují závislosti z různých důvodů: jedna třída posílá zprávu druhé třídě nebo jedna třída používá druhou třídu jako parametr při volání operace atd.

Dobrá analýza vyžaduje minimalizaci závislostí, které se projeví ve snadnější údržbě systému. Tento požadavek je většinou těžké dodržet pro závislosti mezi jednotlivými třídami, snadnější a v praxi uplatňovaná je minimalizace závislostí mezi balíčky.

Závislosti mezi balíčky vycházejí ze závislostí mezi třídami zkoumaných balíčků. Závislost mezi dvěma balíčky tedy existuje, existuje-li jakákoliv závislost mezi dvěma třídami z těchto balíčků. Z uvedeného vyplývá, že při seskupování tříd do balíčků by měla být dodržena určitá pravidla: třídy s těsnými vazbami – především dědičnost, kompozice a agregace by měly být umístěny ve stejném balíčku. Při návrhu diagramu balíčků je také nutné vyvarovat se cyklických závislostí mezi balíčky.

Na následujícím obrázku vidíme, jak bychom mohli do balíčků rozdělit náš objednávkový systém (viz. diagram tříd). Všimněte si závislostí mezi balíčky.



Obrázek PD2: *Závislosti mezi balíčky*

9.4 Shrnutí diagramů balíčků

Používání balíčků nám umožňuje rozčlenit systém do menších logických celků, což vede ke zpřehlednění struktury systému a získání lepší kontroly nad závislostmi jednotlivých částí systému.

10 Diagramy komponent *(component diagrams)*

Diagram komponent umožňuje modelovat fyzický aspekt objektově orientovaného softwarového systému. Ilustruje architekturu systému na úrovni tzv. komponent.

10.1 Komponenta

Komponenta je fyzickou součástí systému. Za komponentu můžeme považovat tabulku, datový i spustitelný soubor, dynamickou knihovnu, .. atd.

V objektově orientovaném prostředí je komponenta chápána jako fyzická nahraditelná část systému, která obaluje implementaci a poskytuje realizaci množiny specifikovaných rozhraní.

V objektově orientovaném systému je komponenta většinou implementací jedné, či několika tříd (např. *Enterprise Java Beans*).

10.2 Proč modelovat komponenty

Existuje několik důvodů, proč modelovat komponenty a vztahy mezi nimi:

- Abychom mohli zákazníkovi předvést strukturu hotového systému.
- Aby vývojáři dopředu věděli, co a s čím budou dělat.
- Aby osoby pověřené sepsáním dokumentace věděly, o čem psát.
- Abychom si sami usnadnili opakované použití komponenty.

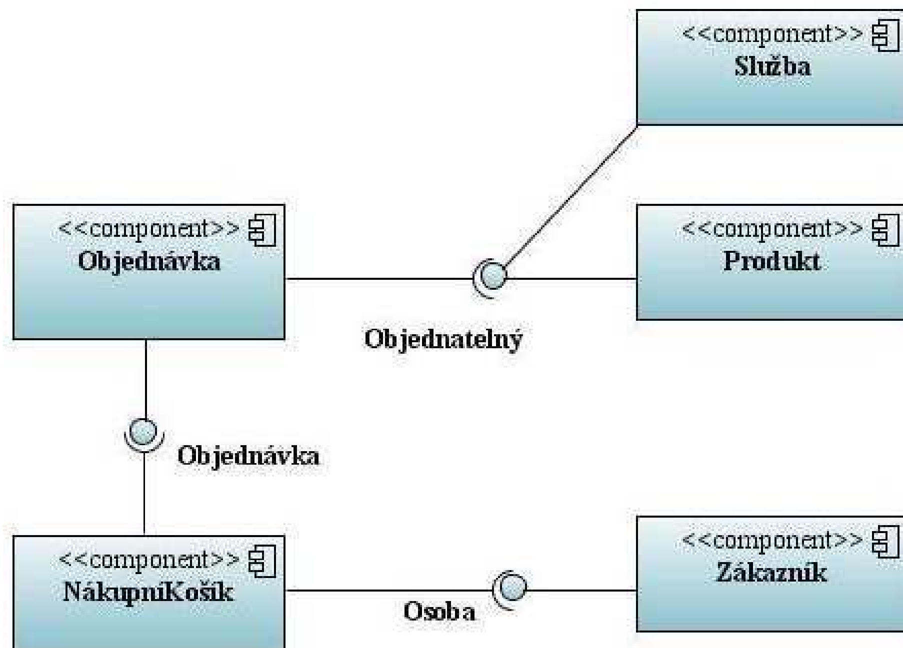
Všimněme si posledního bodu. Jednou z nejdůležitějších vlastností komponent je to, že umožňují opakované použití již jednou hotové věci. V dnešním světě velmi záleží na rychlosti, s jakou dokážeme připravovaný systém uvést do chodu. Když tedy pro jeden systém vyvineme komponentu, kterou budeme

schopni použít i v dalších projektech, bude to pro nás velká výhoda. Proto stojí za to věnovat modelování takových komponent čas i práci.

10.3 Zobrazení komponenty

Komponenta se v UML značí obdélníkem se symbolem komponenty v pravém horním rohu. Komponenty mohou být propojeny (komunikovat) přímo nebo prostřednictvím rozhraní. Komponenta může rozhraní poskytovat či vyžadovat. Rozhraní označíme kroužkem, a pokud ho komponenta poskytuje, je k ní tento kroužek připojen asociací. Pokud komponenta vyžaduje určité rozhraní, připojíme jí asociací k polokroužku (tzv. *socket*), k němuž připíšeme název požadovaného rozhraní.

Na následujícím obrázku vidíme diagram komponent znázorňující část fiktivního internetového obchodu.

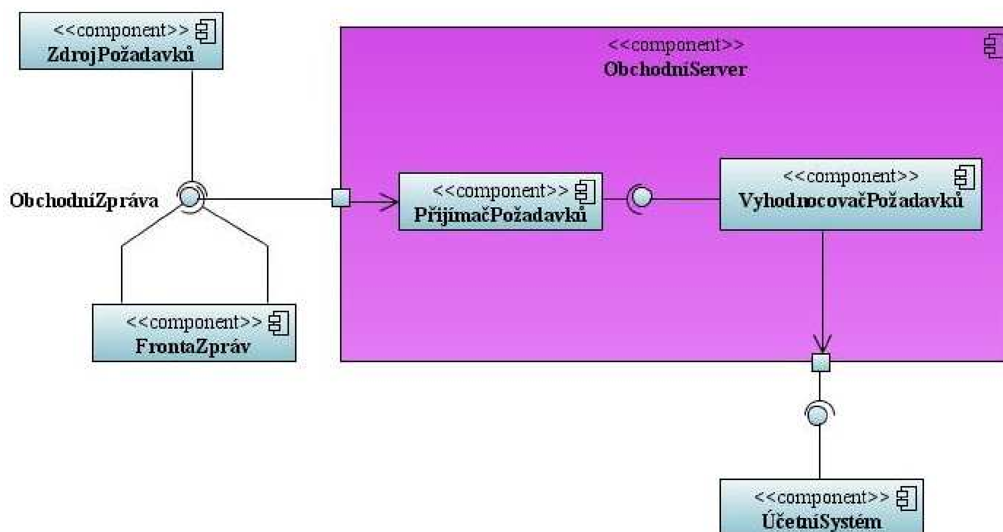


Obrázek DC1: *Diagram komponent části internetového obchodu*

10.4 Systém zpracovávající obchodní zprávy

Na dalším obrázku jsme znázornili poněkud složitější systém.

Komponenta *ZdrojPožadavků* zasílá požadavky komponentě *ObchodníServer*, k tomu používá rozhraní *ObchodníZpráva*, které obchodní server poskytuje, protože však existuje nebezpečí, že obchodní server nebude po síti vždy dostupný, existuje komponenta *FrontaZpráv*, které může poslat požadavek v tomto případě. Komponenta *FrontaZpráv* se stará o bezpečné doručení zpráv obchodnímu serveru, k tomu vyžaduje rozhraní *ObchodníZpráva*, které navíc sama poskytuje kvůli komunikaci se *ZdrojemPožadavků*.



Obrázek DC2: *Diagram komponent systému zpracovávajícího obchodní zprávy*

Vidíme, že komponenta *ObchodníServer* se skládá z dalších komponent, které komunikují s okolím prostřednictvím portů (malé čtverečky na hranici komponenty *ObchodníServer*), které slouží jako body interakce klasifikátoru s okolím (ať už komponenty *ObchodníServer*, či jejích subkomponent). Pokud chceme znázornit třídy, kterými je komponenta tvořena, znázorníme to

podobně – zakreslením tříd do komponenty. Po vyhodnocení požadavku obchodním serverem je výsledek zaslán přes blíže nespecifikované rozhraní komponentě *ÚčetníSystém*.

10.5 Shrnutí diagramů komponent

Diagram komponent použijeme v případě, že se náš systém skládá z komponent a chceme znázornit jejich vztahy, ať už přímé nebo prostřednictvím rozhraní. Diagram komponent nám též umožňuje znázornit vnitřní strukturu komponenty.

11 Diagramy nasazení *(deployment diagrams)*

Diagramy nasazení zobrazují fyzické uspořádání systému a znázorňují jak jsou jednotlivé části softwaru rozmístěny na části hardwaru.

11.1 Prvky diagramu nasazení

Hlavními prvky diagramu nasazení jsou *uzly (nodes)*. Uzel představuje jednotku, na které může být umístěn a spouštěn software. Rozlišujeme dvě základní formy uzlu:

- *zařízení (device)*, je hardware – počítač nebo jiná hardwarová jednotka
- *spouštěcí prostředí (execution environment)* je software, který hostí nebo obsahuje jiný software – např. operační systém, nebo kontejner (např. *EJB kontejner v J2EE*)

Uzel můžeme modelovat dvěma způsoby: buď jako *obecný uzel*, kdy nás zajímá pouze typ uzlu, nebo jako *konkrétní instanci*. Zde platí stejný vztah jako mezi třídou a jejími objekty (instancemi), název instance uzlu je v diagramu podtržen.

Uzly mohou být propojené *komunikačními cestami (communication path)*, což jsou asociace. Komunikační cesty můžeme označit například názvem či typem protokolu.

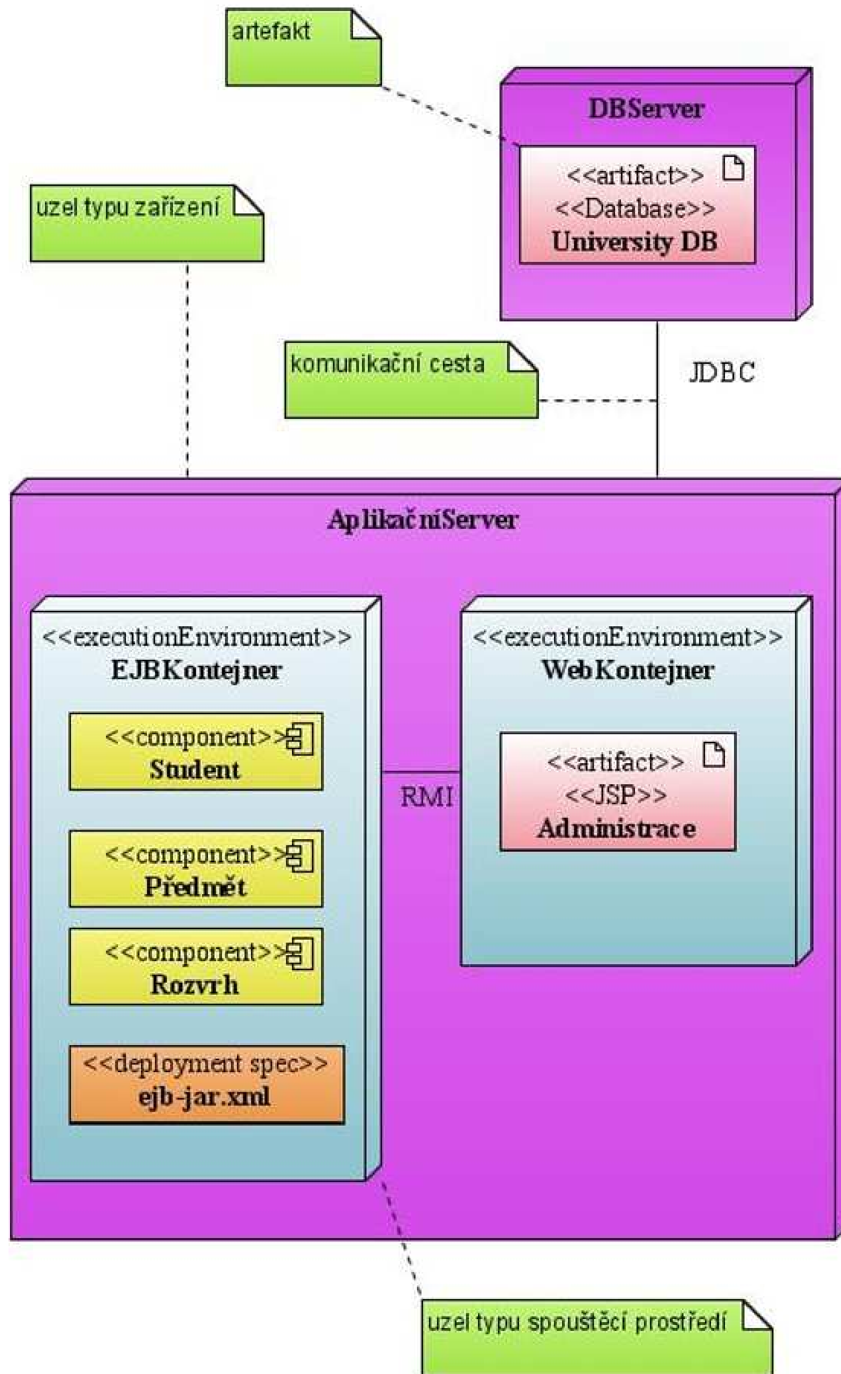
Uzly mohou obsahovat *artefakty (artifacts)*, což jsou fyzické manifestace softwaru, obvykle soubory. Ty mohou být spustitelné (.exe soubory, binární soubory, .DLL, .JAR, skripty, …atd.), datové, konfigurační, a další. Vepsání artefaktu do ikony uzlu znamená, že artefakt je *nasazen (deployed)* v tomto uzlu.

Dále mohou uzly obsahovat *komponenty* (viz. Diagram komponent) a *specifikace nasazení (deployment specifications)*, které představují konfigurační soubory.

11.2 Diagram nasazení univerzitního systému

V následujícím příkladě vidíme diagram nasazení fiktivního univerzitního systému.

Systém je nasazen na uzlu *AplikačníServer* typu zařízení, který obsahuje dva uzly typu běhové prostředí. V uzlu *EJBKontejner* jsou nasazeny tři komponenty, zajišťující business logiku aplikace a jedna specifikace nasazení, z názvu je zřejmé, že se jedná o xml soubor, ten obsahuje konfigurační informace pro vložené komponenty (tzv. *deployment descriptor*). Uzel *WebKontejner* obsahuje artefakt *Administrace*, který zajišťuje prezentační část aplikace a skládá se z *jsp komponent* (to jsme vyjádřili *stereotypem*). Uzly *EJBKontejner* a *WebKontejner* jsou propojeny pomocí RMI (*remote method invocation*). Celý aplikační server komunikuje pomocí JDBC s uzlem *DBServer*, který obsahuje artefakt představující databázi pro náš systém. Všimněte si, že jsme do diagramu nezakreslili, jak je systém přístupný zvnějšku. Vzhledem k tomu, že je prezentační část tvořena *jsp stránkami*, mohli bychom k němu přistupovat pomocí webovského prohlížeče. To bychom znázornili přidáním dalšího uzlu, představujícího tento prohlížeč a propojili bychom ho nejspíše protokolem http s *WebKontejnerem*.

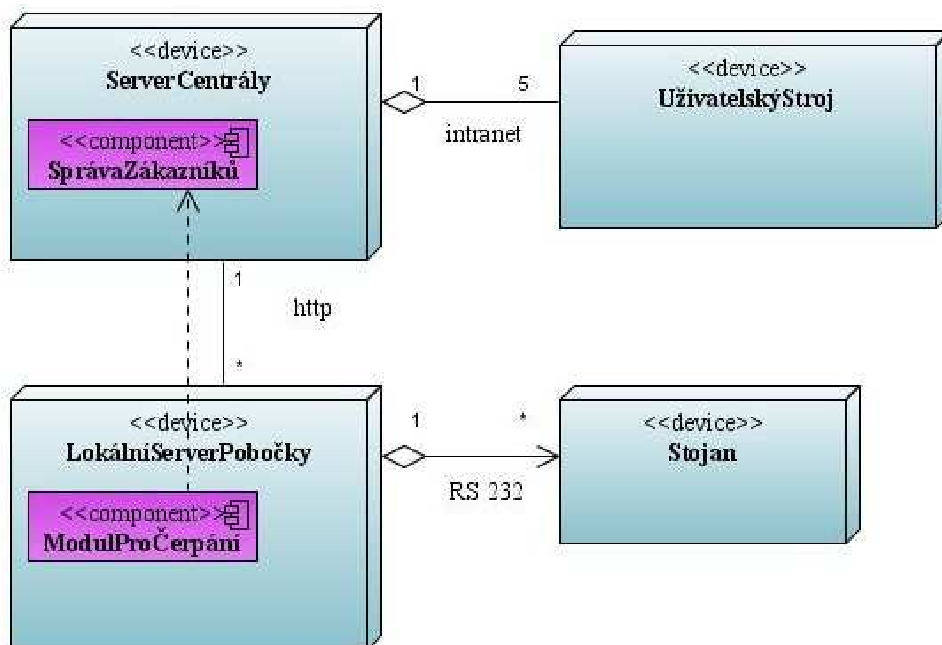


Obrázek DD1: *Diagram nasazení univerzitního systému*

11.3 Diagram nasazení sítě čerpacích stanic

Na dalším obrázku vidíme diagram nasazení sítě čerpacích stanic.

Systém je řízen centrálně pomocí pěti uživatelských strojů připojených na *ServerCentrály* pomocí intranetu. *ServerCentrály* komunikuje s jednotlivými *LokálnímiServeryPobočky*, každý tento server komunikuje s několika Stojany a obsahuje komponentu *ModulProČerpání*, která je závislá na komponentě *SprávaZákazníků* umístěné na *ServeruCentrály*.



Obrázek DD2: *Diagram nasazení sítě čerpacích stanic*

11.4 Shrnutí diagramů nasazení

Diagramy nasazení nám umožňují získat dobrý přehled o tom, jak jsou nasazeny jednotlivé části systému, což oceníme především u složitějších distribuovaných aplikací.

12 Diagramy komunikace *(communication diagrams)*

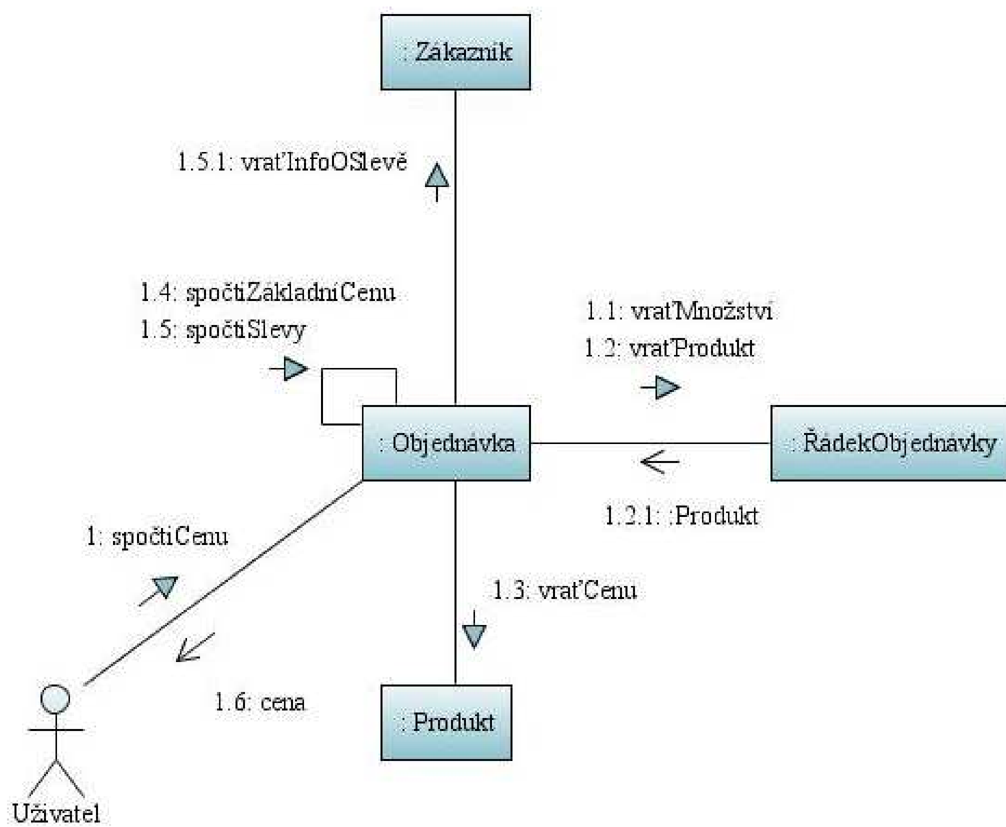
Diagramy komunikace, v UML 1.x nazývány diagramy kolaborace, jsou druhem diagramů interakce. Podobně jako sekvenční diagram, je diagram komunikace používán k modelování dynamického chování případu užití. Oproti sekvenčnímu diagramu se diagram komunikace více soustředí na zobrazení vazeb mezi interagujícími účastníky, než na časovou posloupnost.

12.1 Zobrazení diagramu komunikace

V diagramu komunikace jsou životní čáry účastníků zobrazeny jako obdélníky, které můžeme umístit na libovolnou pozici. Mezi účastníky můžeme zobrazit *spojnice (links)*, které znázorňují propojení účastníků. Zasílání zpráv se označuje jako v sekvenčním diagramu – šipkami. Důležité je číslování, které určuje posloupnost a vnořenost zpráv.

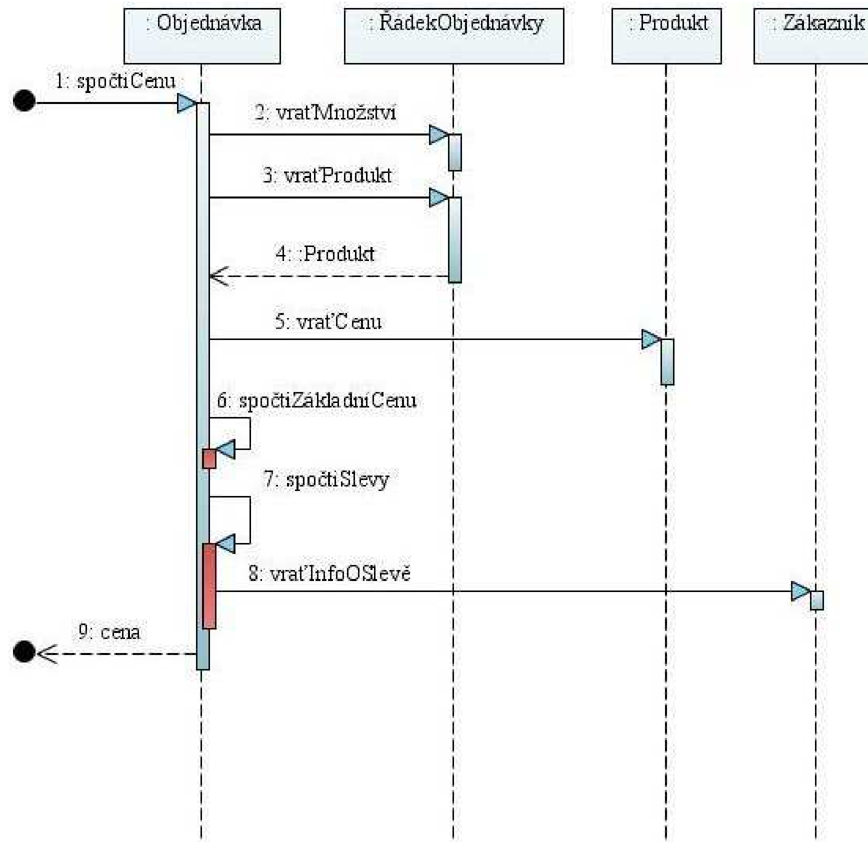
12.2 Výpočet ceny objednávky

Na následujícím obrázku vidíme diagram komunikace představující scénář pro výpočet ceny objednávky, tento scénář jsme již modelovali v kapitole o sekvenčních diagramech, oba diagramy zobrazují tutéž interakci, u diagramu komunikace jsme jako zdroj zprávy *vraťObjednávku* a příjemce jejího výsledku znázornili aktéra Uživatel, zatímco v sekvenčním diagramu byl zdroj i příjemce anonymní. Všimněte si také vnořeného číslování, ze kterého je zřejmá vnořenost volání zpráv (metod). Vidíme například, že všechny uvedené metody jsou prováděny v rámci metody *vraťObjednávku* a metoda *vraťInfoOSlevě* je volána v rámci metody *spočtiSlevu*.



Obrázek CD1: *Diagram komunikace výpočtu ceny objednávky*

Pro názornost zde ještě jednou uvádíme sekvenční diagram znázorňující tutéž interakci.



Obrázek SD1: *Sekvenční diagram výpočtu ceny objednávky*

12.3 Shrnutí diagramů komunikace

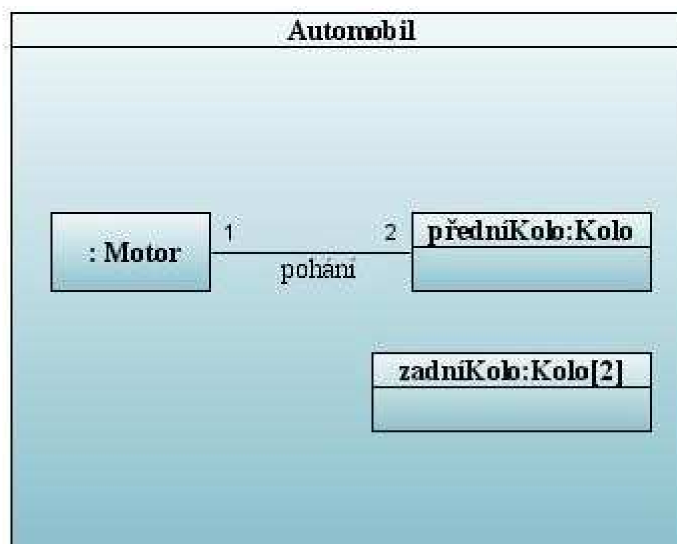
Diagramy komunikace dokáží (až na interakční rámce) zobrazit stejné informace jako sekvenční diagramy, ty jsou také proto v praxi používány častěji. Diagram komunikace může však být v některých situacích přehlednější a srozumitelnější. Diagram komunikace použijeme v případě, že se potřebujeme zaměřit na propojení účastníků během komunikace, pokud je však důležitější posloupnost volání, či potřebujeme znázornit interakční rámce, je na místě použití sekvenčního diagramu. Pokud chceme mít pro stejnou situaci oba typy diagramu, můžeme využít schopnosti nástroje VP-UML vygenerovat diagram komunikace ze sekvenčního diagramu.

13 Diagramy struktury *(composite structure diagrams)*

Jedním z nejvýznamnějších nových rysů UML 2 je možnost hierarchicky dekomponovat třídu do vnitřní struktury. To nám umožňuje rozdělit komplexní objekt na části. Přesně to nám umožňuje diagram struktury.

13.1 Vnitřní struktura třídy

Na následujícím obrázku jsme znázornili diagram struktury odkrývající vnitřní strukturu třídy *Automobil*.

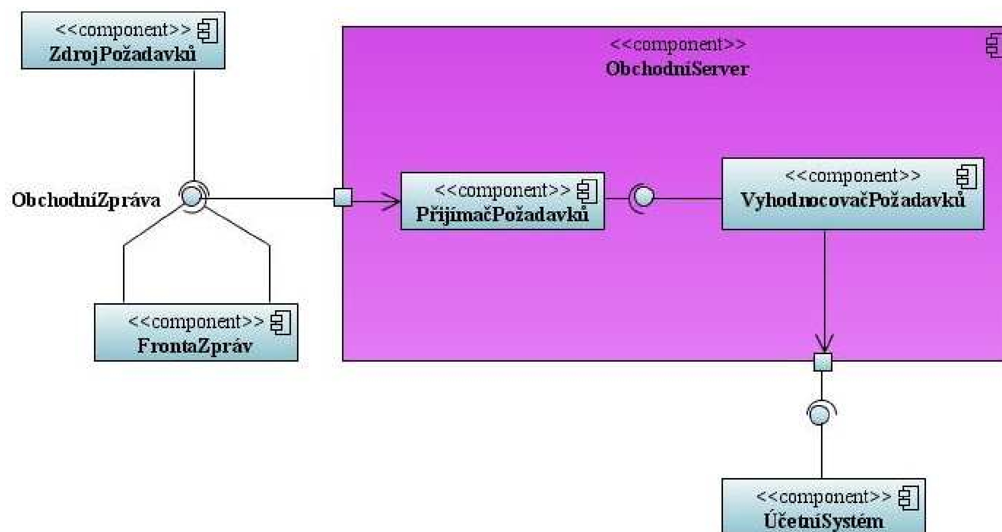


Obrázek CSD1: *Diagram struktury třídy Automobil*

13.2 Vnitřní struktura komponenty

Narozdíl od diagramu balíčků, který představuje sloučení prvků v *čase kompilace (compile-time)*, diagram struktury představuje sloučení prvků v *běhovém čase (runtime)*, proto je také část notace diagramů struktury

používána v digramech komponent. Tak jak jsme to učinili v diagramu komponent u komponenty obchodního serveru. Vidíme, že poskytovaná i požadovaná rozhraní můžeme delegovat pomocí portu.



Obrázek DC2: *Diagram komponent systému zpracovávajícího obchodní zprávy*

13.3 Shrnutí diagramů struktury

Diagramy struktury jsou novým typem diagramů UML 2, přestože již dříve existovali metody s podobnými myšlenkami. Sami tvůrci UML jsou přesvědčeni, že diagram struktury se stane užitečným a často využívaným typem diagramu.

14 Diagramy časování *(timing diagrams)*

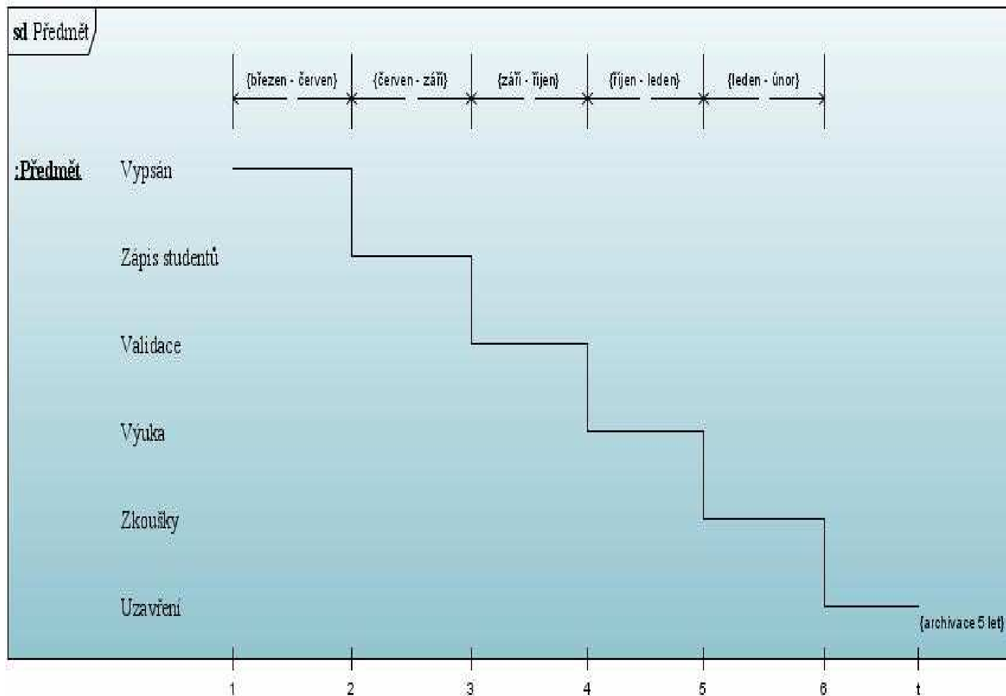
Diagramy časování jsou další formou diagramů interakce (jsou speciálním typem sekvenčního diagramu), důraz je zde kladen na časové hledisko. Můžeme pomocí nich znázornit chování jednoho či několika objektů.

14.1 Úvodní příklad

Uvažujme jednoduchý příklad diagramu časování popisující životní cyklus předmětu vyučovaného na univerzitě v zimním semestru.

Předpokládejme, že předmět je připraven v rozvrhu a vypsán již během letního semestru, studenti mají možnost si ho zapisovat od června do září, během září pak proběhne validace a pokud je předmět otevřen, je vyučován, během zkouškového období z něj studenti konají zkoušku a po skončení zkouškového období je předmět uzavřen. Informace o uzavřeném předmětu jsou uchovány v systému ještě nejméně 5 let, což vyjádříme pomocí *časového omezení (time constraint)*.

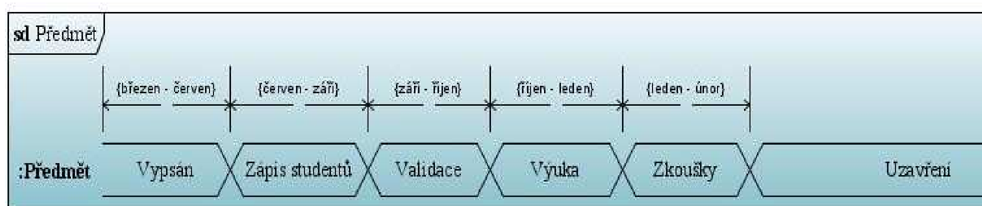
Jednotlivé objekty jsou vkládány do *časovacího rámce (timing frame)* jako *životní čáry (lifelines)*, které představují graf, zachycující chování objektu. Na vodorovné ose jsou znázorněny časové jednotky (čas plyne zleva doprava) a na svislé ose jednotlivé stavy objektů. V daném časovém úseku se objekt nachází v určitém stavu, což je přehledně znázorněno funkční křivkou. V našem příkladě jsme použili *podmínky trvání (duration constraints)*, které vymezují jednotlivé časové úseky.



Obrázek TD1a: *Diagram časování představující životní cyklus vyučovaného předmětu – robustní styl zobrazení*

14.2 Další varianta zobrazení diagramu časování

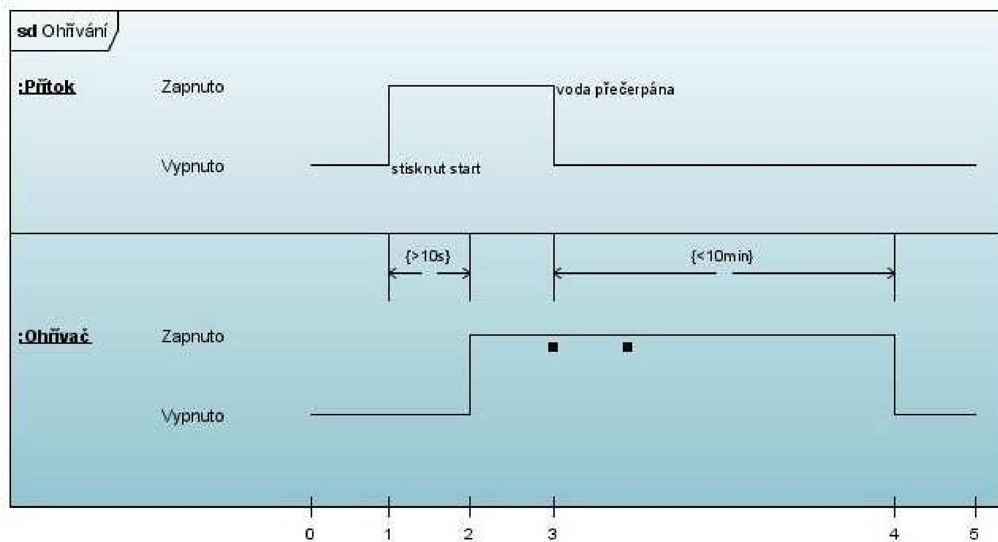
Existují dvě varianty, jak zobrazit diagram časování, ta, kterou vidíte na obrázku výše je označována jako *plný* nebo *robustní styl zobrazení*. Druhou variantou je tzv. *kompaktní styl zobrazení*. Kompaktní styl zobrazení je vhodnější, pokud má objekt více stavů, protože je v tomto případě přehlednější. VP-UML však nezobrazí označení časových jednotek ani časové omezení.



Obrázek TD1a: *Diagram časování představující životní cyklus vyučovaného předmětu - kompaktní styl zobrazení Diagram časování s více objekty*

Na dalším příkladě si ukážeme diagram časování se dvěma objekty. Tento diagram představuje interakci přítokové nádržky a ohřívací plošinky kávovaru.

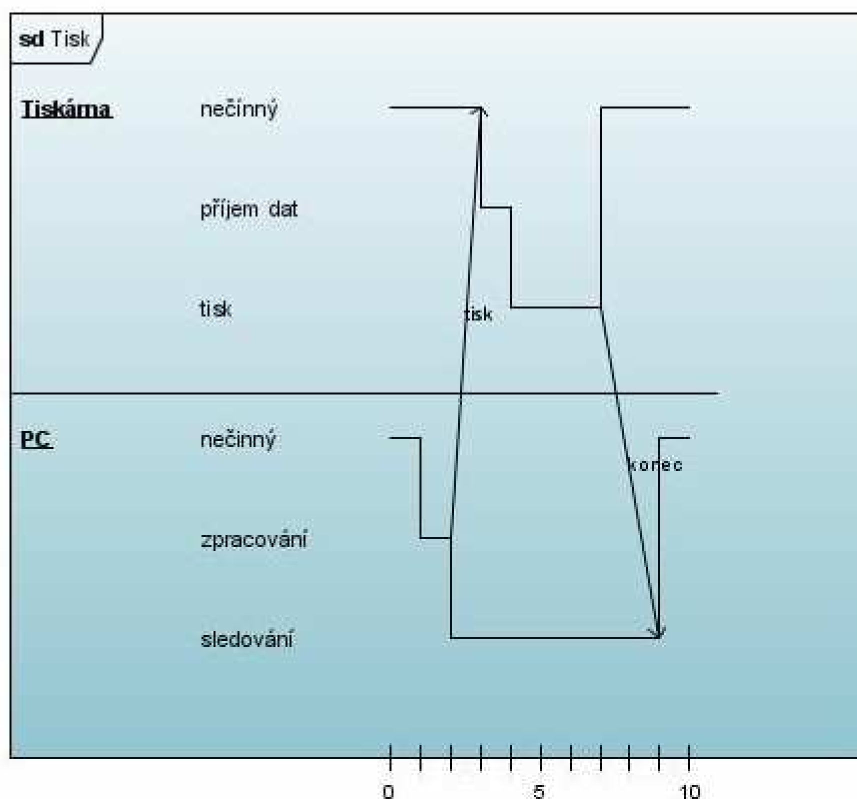
Zapnutí kávovaru je v diagramu znázorněno jako stimul, který uvádí do chodu přítokovou nádobku, která začne pomalu přelévat vodu do konvice umístěné na ohřívací plošině. Ohřívací plošina počká minimálně 10 sekund od zapnutí kávovaru a poté začne ohřívat. Po vyprázdnění přítokové nádržky se vypne přítok, ohřívání pokračuje dále, maximálně však 10 minut, poté se vypne.



Obrázek TD2: *Diagram časování představující interakci částí kávovaru*

14.3 Předávání zpráv mezi objekty

Poslední obrázek představuje interakci PC a tiskárny a předávání zpráv mezi objekty v diagramu časování.



Obrázek TD3: *Diagram časování znázorňující předávání zpráv mez PC a tiskárnou během tisku*

14.4 Shrnutí diagramů časování

Diagramy časování jsou užitečné pro znázornění časových omezení mezi změnami stavů na různých objektech. Diagramy časování se často používají k modelování vložených systémů *(embeded systems)*, např. systém kontroly vstřikování paliva v automobilech, používají se také v elektronických schématech a hardwarovém inženýrství. Příležitostně se mohou uplatnit i v podnikových systémech, ale to opravdu jen zřídka.

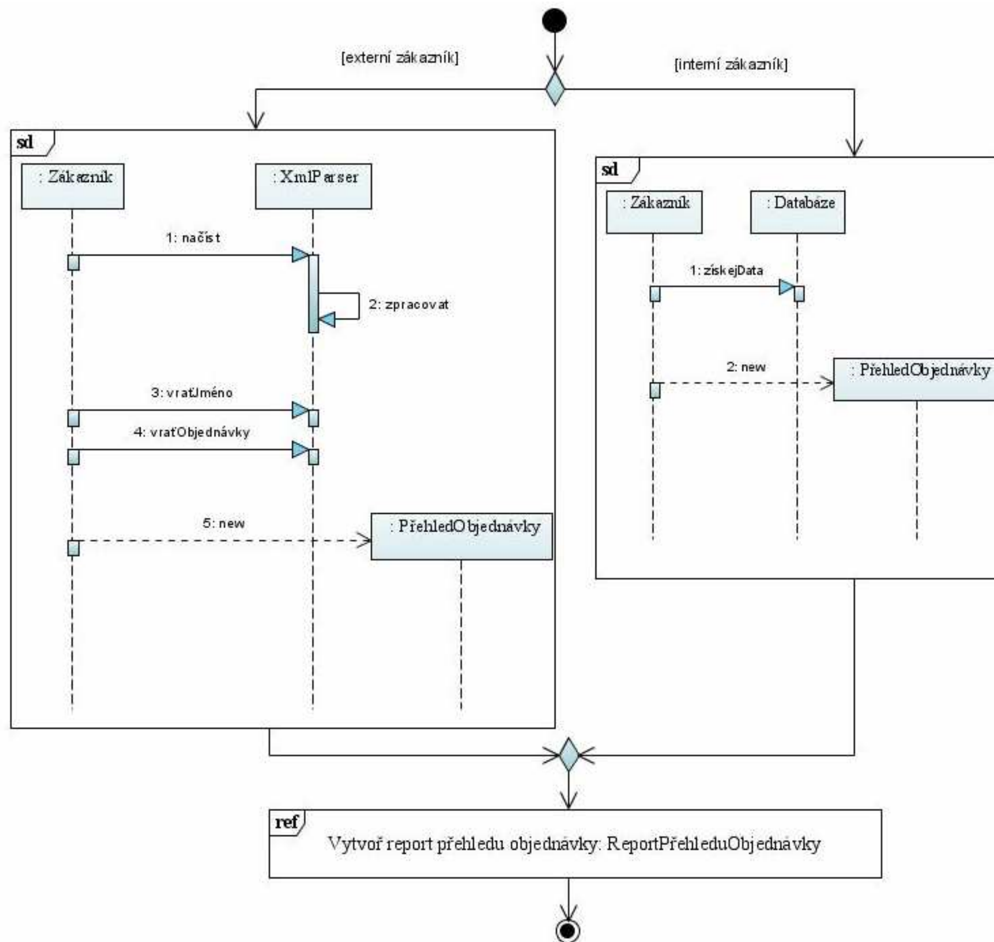
15 Přehledové diagramy interakce *(interaction overview diagrams)*

Přehledové diagramy interakce v sobě spojují diagramy aktivity a sekvenční diagramy. Na přehledový diagram interakce můžeme nahlížet buď jako na diagram aktivity, v němž jsou aktivity nahrazeny malými sekvenčními diagramy nebo jako na sekvenční diagram, jehož struktura je jakoby porušena notací diagramu aktivity, pomocí níž je znázorněn tok řízení.

15.1 Vytvoření reportu přehledu objednávky

Na následujícím obrázku vidíme přehledový diagram interakce, který znázorňuje postup vytvoření přehledu objednávky daného zákazníka.

Pokud se jedná o interního zákazníka, máme potřebná data o něm uložena v databázi, z nich poté vytvoříme nový objekt – Přehled objednávky (pravý sekvenční diagram), pokud je zákazník externí, dostaneme data o něm z nějakého externího systému, ve formátu xml, které musíme nejprve zpracovat *xml parserem*, poté získáme potřebná data a opět vytvoříme objekt Přehled objednávky (levý sekvenční diagram). Vytvořený přehled objednávky předáme jako vstupní parametr do sekvenčního diagramu pro vytvoření reportu (spodní sekvenční diagram, resp. pouze odkaz na něj).



Obrázek IOD1: *Přehledový diagram interakce vytvoření reportu přehledu objednávky*

15.2 Shrnutí přehledových diagramů interakce

Přehledový diagram interakce je novým typem diagramu UML od verze 2, nepřináší žádnou novou vlastní syntaxi a vlastně jen spojuje dohromady sekvenční diagram a diagramy aktivity, teprve čas ukáže, zda bylo takovéto spojení dvou syntaxí z praktického hlediska účelné.

16 Závěr

Modelovací jazyk UML nabízí ve verzi 2.0 velice bohatou syntaxi, některé prvky jsou však základní a používají se velmi často, tyto prvky jsem se pokusil vybrat a vysvětlit v kontextu příkladů, které prostupují celou prací a které by měli lépe osvětlit běžné způsoby používání daných prvků v praxi. Po prostudování celé práce by proto měl být čtenář seznámen se základy používání jazyka UML na praktické úrovni, ovšem takto získané znalosti je nutno doplnit studiem další literatury.

Seznam použité literatury:

[1] FOWLER, M. UML Distilled, Third Edition. : Addison-Wesley,

2005. ISBN 0321193687

[2] FOWLER, M. – SCOTT, K. UML Distilled, Second Edition. Addison-Wesley,

2002. ISBN 020165783X

[3] ARLOW, J. – NEUSTADT, I. UML a unifikovaný proces vývoje aplikací, Druhé vydání. Brno : Computer Press a.s., 2005. ISBN 807226947X

[4] BARNES, D. J. – KÖLLING, M. Objects First with Java . Prentice Hall,

2003. ISBN 0130449296

[5] SCHMULLER, J. Myslíme v jazyku UML. Praha : Grada Publishing, spol. s r.o., 2001. ISBN 8024700298

[6] KANISOVÁ, H. – MÜLLER, M. UML srozumitelně. Brno : Computer Press a.s., 2004. ISBN 8025102319

[7] PECINOVSKÝ, R. Myslíme objektově v jazyku Java 5.0. Praha : Grada Publishing a.s., 2004. ISBN 8024709414

Ostatní zdroje:

[8] http://www.visual-paradigm.com/VPGallery/diagrams

[9] http://www.sparxsystems.com/resources/uml2_tutorial

[10] http://www.agilemodeling.com

[11] http://www.uml.org