VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
katedra informatiky

Úvod do softwarového inženýrství

verze 1.1

Prof. Ing. Ivo Vondrák, CSc.

Ostrava 2002

Motto

Lékař, stavební inženýr a softwarový inženýr se přeli o to, která z jejich profesí je tou nejstarší. Lékař poznamenal: „ V Bibli svaté se píše, že Bůh stvořil Evu z žebra Adamova. Jinými slovy řečeno, se jedná o chirurgický zákrok a tudíž mohu směle tvrdit, že má profese je tou nejstarší v širém světě“. Stavební inženýr ho však přerušil a oponuje: „ Mnohem dříve, v knize Genesis, se však tvrdí, že Bůh definoval řád nebe a země z chaosu a nepořádku. Co je to jiného než aplikace oboru stavebnictví. Čili, milý doktore, mé povolání je ještě starší“. Načež se softwarový inženýr znuděně protáhne ve svém křesle, pousměje, a s pevným sebevědomím prohlásí: „Pánové, a kdo si myslíte, že vytvořil ten chaos?!“

Cíle předmětu

Cílem předmětu je uvést studenty do disciplíny zabývající se problematikou vývoje rozsáhlých softwarových systémů. Předmět je postaven na metodice objektově orientovaného přístupu využívající jazyk UML (Unified Modeling Language).

Po prostudování textu budete znát:

- jaký je životní cyklus (proces) tvorby softwarového díla;
- jakým způsobem se postupuje při specifikaci, analýze, návrhu, implementaci a testování softwarového systému;
- jak lze formalizovat celý postup s pomocí jazyka UML.

Získáte:

- schopnost specifikovat softwarový produkt nezávisle na vybraném programovacím jazyce a implementačním prostředí;
- praktické zkušenosti jak používat jazyk UML.

Budete schopni:

- přistupovat k tvorbě softwaru inženýrským, tedy systematickým způsobem.

Čas potřebný k prostudování základního učiva předmětu

• Základní pojmy:	1,5 hod.
• Softwarový proces:	3,0 hod.
• Byznys modelování:	3,0 hod.
• Specifikace požadavků:	3,0 hod.
• Analýza a návrh:	6,0 hod.
• Implementace:	3,0 hod.
• Testování:	1,5 hod.
• **Celkem:**	**21 hod.**

Obsah předmětu

1. Základní pojmy
- 1.1. Definice softwarového inženýrství
- 1.2. Schéma procesu vývoje softwarového díla

2. Softwarový proces
- 2.1. Definice softwarového procesu
- 2.2. Základní typy softwarového procesu
- 2.3. Process RUP, jeho cykly, fáze a iterace
- 2.4. Základní a podpůrné toky činností
- 2.5. Jazyk UML

3. Byznys modelování
- 3.1. Diagramy aktivit a tříd jazyka UML
- 3.2. Metoda BPM

4. Specifikace požadavků
- 4.1. Aktéři a funkční specifikace pomocí případů použití
- 4.2. Diagramy případů užití
- 4.3. Definice pojmu objekt
- 4.4. Vztahy mezi objekty a jejich interakce

5. Analýza a návrh
- 5.1. Modely a jejich diagramy
- 5.2. Definice pojmu třída
- 5.3. Vztahy mezi třídami a objekty
- 5.4. Diagramy tříd jazyka UML
- 5.5. Specifikace dynamického chování
- 5.6. Strukturování softwarového systému
- 5.7. Návrh a jeho cíle
- 5.8. Architektura výsledného systému
- 5.9. Návrhové vzory a aplikační rámce
- 5.10. Model nasazení

6. Implementace
- 6.1. Mapování elementů logického modelu na komponenty
- 6.2. Zdrojové, binární a spustitelné komponenty
- 6.3. Diagramy nasazení

7. Testování a nasazení softwarového produktu
- 7.1. Cíle procesu verifikace a validace
- 7.2. Modely testování
- 7.3. Nasazení softwarového systému

Použitá literatura

Humphrey, W. Managing the Software Process, Addison-Wesley/SEI series in Software Engineering, Reading, Ma, 1989

Gamma, E., Helm,R., Johnson,R., Vlissides,J. Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994

Icon Computing, Inc. Object-Oriented Design, Idioms and Architectures, Austin, 1996

Rational Unified Process Whitepaper: Best Practices for Software Development Teams - Rational Software, 1998

Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process, Addison Wesley Longman, Inc., 1999

Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modeling Language User Guide, Addison Wesley Longman, Inc., 1999

Schmuller, J.: Teaching Yourself UML in 24 Hours, Sams, 1999

1. Úvodní pojmy

V této kapitole se dozvíte:

co je předmětem disciplíny softwarové inženýrství a s jakými základními pojmy budeme dále v tomto předmětu manipulovat.

Doba potřebná k prostudování této kapitoly je:

- 1,5 hodiny

1.1 Co je to softwarové inženýrství

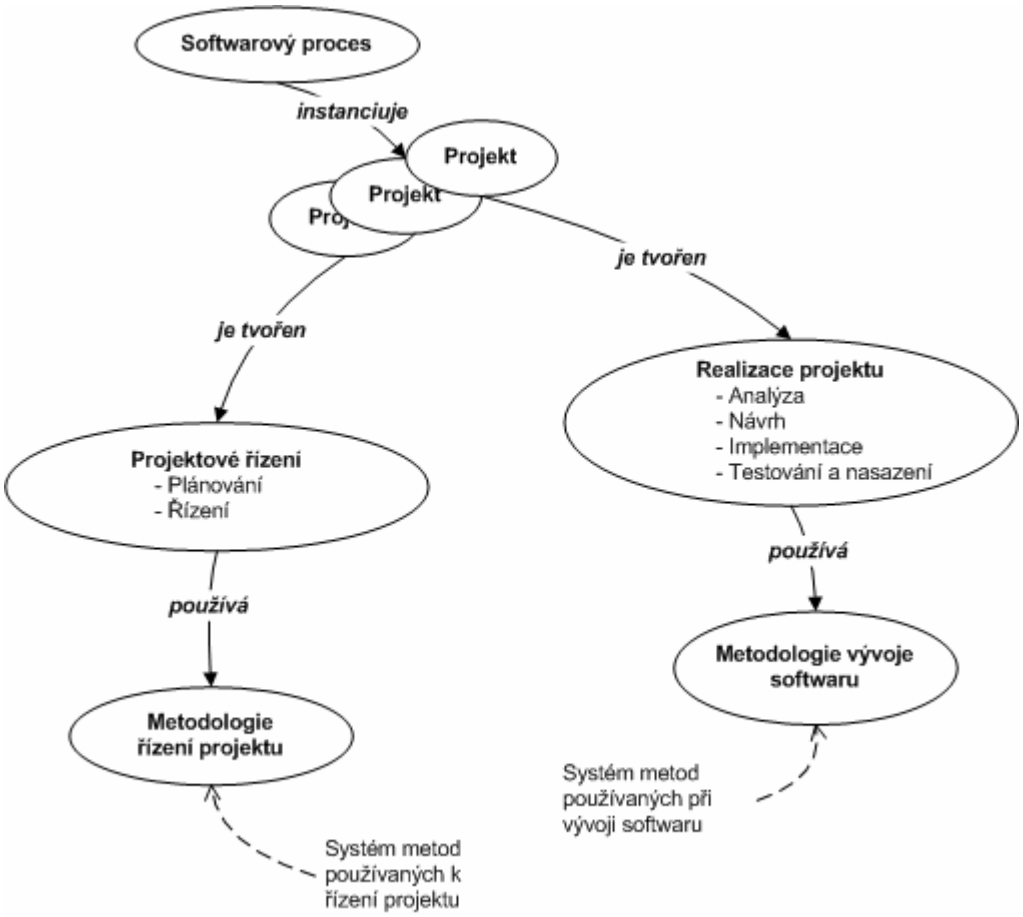
Softwarové inženýrství je inženýrská disciplína zabývající se praktickými problémy vývoje rozsáhlých softwarových systémů

Z pohledu dané definice vyplývá, že vývoj softwarového systému zahrnuje celou řadu faktorů nutných k úspěšnému vytvoření požadovaného produktu:

- technické aspekty zahrnující počítačovou infrastrukturu a softwarové vybavení;
- netechnické aspekty dané organizační strukturou organizace vyvíjející daný produkt a jejími ekonomickými možnostmi;
- znalostmi z oblasti specifikace požadavků na softwarový produkt, jeho analýzy, návrhu, implementace, testování a na konec také instalace u zákazníka;
- lidské zdroje schopné aplikovat výše uvedené znalosti a uplatnit je tak při realizaci softwarového systému;
- řízení spjaté s vývojem samotného produktu umožňující efektivní využití všech výše uvedených faktorů s cílem vytvořit produkt požadované kvality.

1.2 Schématické vyjadření vývoje softwarového produktu

Vývoj softwarového produktu, jak již bylo předesláno v minulé kapitole, zahrnuje celou řadu činností a metod. Základem je stanovený tzv. *softwarový proces*, tedy postup činností nutných k vytvoření softwarového produktu. Dle takto stanoveného předpisu jak vytvářet software se následně definují *projekty* (hovoří se o tzv. *instanciaci* procesu) vztažené k jednotlivým zakázkám. Každý projekt je dále tvořen svou *realizací*, činnostmi vázanými na vývoj vlastního produktu, a *řízením* celého projektu. Jak tato technická tak i netechnická část vyžaduje svou metodologii. V případě řízení se jedná o *metodologii projektového řízení* danou systémem metod používaných v projektovém řízení, zatímco *metodologie vývoje softwarového systému* je tvořena systémem metod používaných při vývoji softwaru. Schéma výše popsaného je vizualizováno pomocí sémantického grafu na obrázku č.1.



Obr. 1: Sémantický graf vývoje softwarového produktu

Záměrem tohoto kurzu je věnovat se části vývoje softwarových systémů spjatých s technickými činnostmi. Vlastní řízení softwarového projektu a s ním spjatá problematika je náplní specializovaného kurzu Procesní a projektové řízení.

Shrnutí

Obsahem této kapitoly bylo čtenáři přiblížit softwarové inženýrství jako vědní disciplínu věnovanou problematice vývoje rozsáhlých softwarových systémů a seznámit ho se základními pojmy a stavebními kameny procesu tvorby takových systémů.

Kontrolní otázky

1. Čím se zabývá softwarové inženýrství?
2. Jaký je vztah mezi softwarovým procesem a projektem?

2. Softwarový proces

V této kapitole se dozvíte:

- co je tzv. softwarový proces, jaké jsou základní modely softwarového procesu používané v praxi;
- jak je definován softwarový proces RUP a jaký je jeho vztah k jazyku UML.

Doba potřebná k prostudování této kapitoly jsou:

- 3 hodiny

2.1. Definice softwarového procesu

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

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

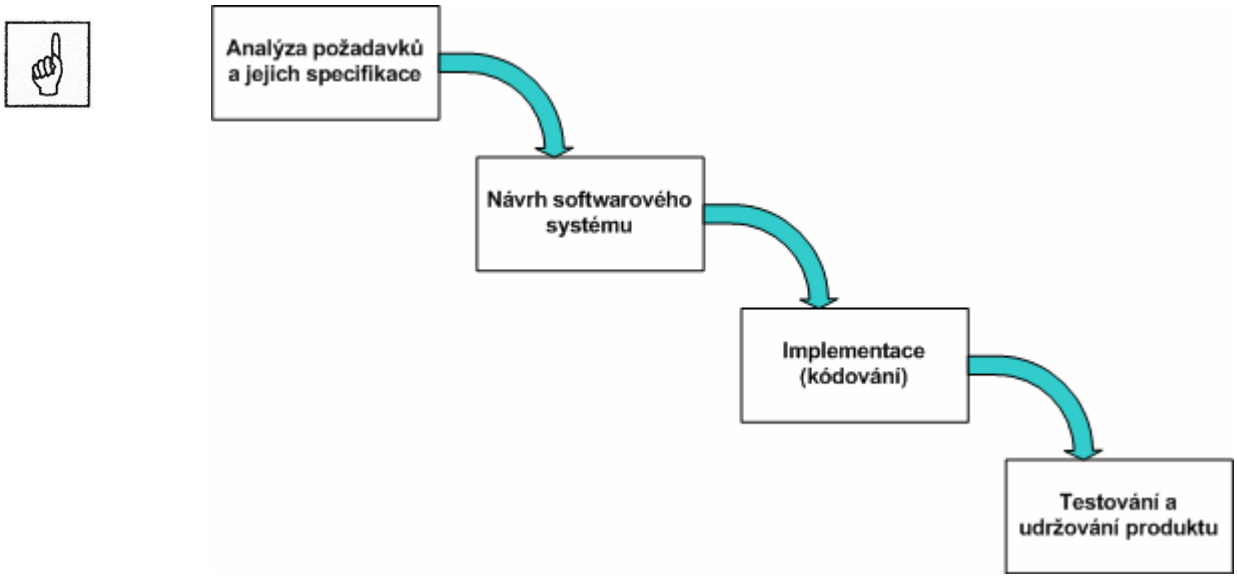
- Krokem může být aktivita nebo opět podproces (hierarchická dekompozice procesu).
- Aktivity a podprocesy mohou probíhat v čase souběžně, tudíž je vyžadována jejich koordinace.
- Je nutné zajistit opakovatelnost použití procesu ve vztahu k jednolivým softwarovým projektům, tedy zajistit jeho znovupoužitelnost.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

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

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

2.2. Základní typy softwarového procesu

Za desítky let vývoje softwarových systémů se objevila celá řada modelů jak by měl takový softwarový proces vypadat. Dodnes však neexistuje detailně a přesně definovaná podoba softwarového procesu, který by mohl přijat jako referenční. Nicméně lze říci, že základem téměř všech se stal tzv. vodopádový model (obr. 2.1), který je možno v různých modifikacích a rozšířeních nalézt ve většině současných přístupů.



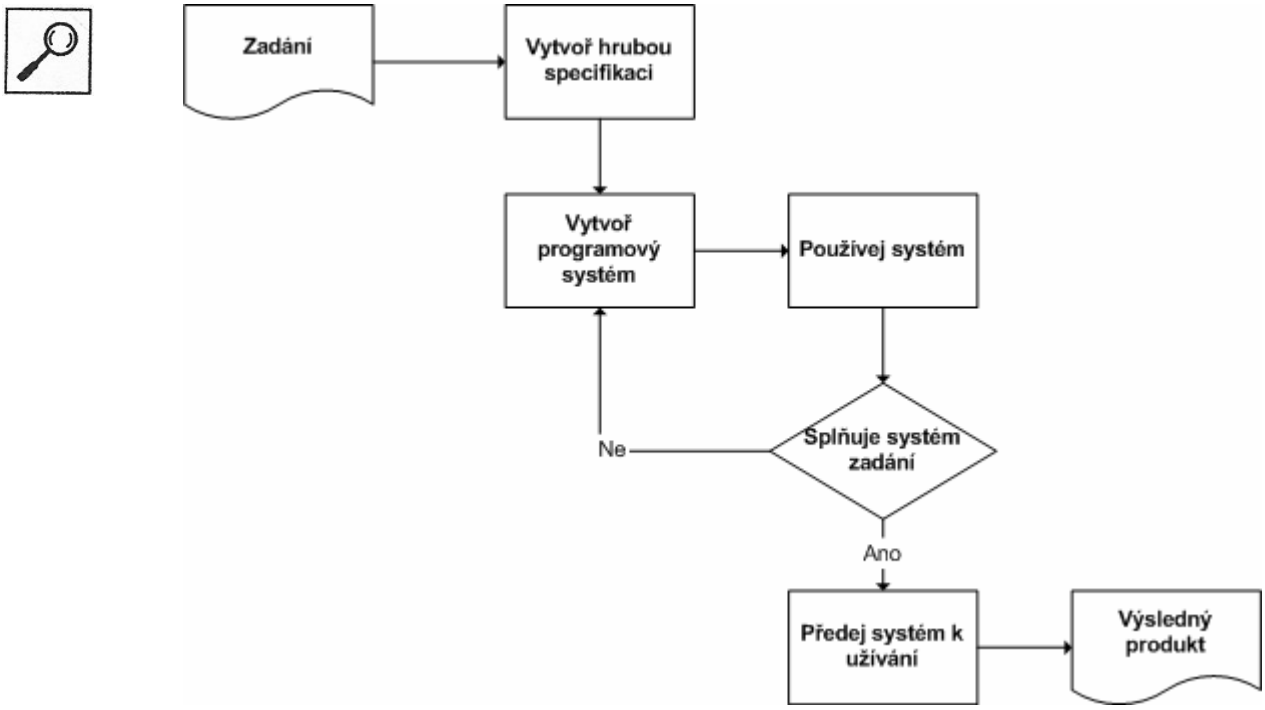
Obr. 2.1: Vodopádový model softwarového procesu

Tento vodopádový model vychází z rozdělení životního cyklu softwarového díla na čtyři základní fáze: *analýza požadavků a jejich specifikace*, *návrh softwarového systému*, *implementace (kódování)* a na konec *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 řečeno, výsledky předchozí fáze „vtékají“ jako vstupy do fáze následující. Nedostatky tohoto základního modelu softwarového procesu, které v průběhu let vedly k jeho modifikaci jsou následující:

- 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.

Snaha odstranit tyto zásadní nedostatky vedla k různým modifikacím základního vodopádového modelu. Jedná se např. o model *inkrementální* postavený na principu postupného vytváření verzí softwarového systému zahrnujících stále širší škálu funkcí definovaných postupně v průběhu jeho vytváření. V podstatě se jedná o celou řadu menších vodopádů s výrazně kratším životním cyklem, kde každý z nich odpovídá nové sadě doplňujících požadavků. *Spirálový* model zahrnuje do svého živostního cyklu další fáze jako je vytvoření a hodnocení *prototypu* ověřující funkcionalitu cílového systému, přičemž každý cyklus nabaluje další požadavky specifikované zadavatelem.
Dříve než se podíváme na model softwarového procesu RUP (Rational Unified Process), který budeme v dalším považovat za referenční, neboť respektuje současnou úroveň znalostí v oblasti softwarového inženýrství , si ještě úkažeme jiný přístup k vývoji softwaru

vycházející z tzv. „průzkumného“ programování (obr. 2.2). Nejedná se softwarový proces v pravém slova smyslu, bohužel je to však stále jedna z možností, jak se software vyvíjí. Považujte tento způsob vývoje spíše za odstrašující případ než za jednu z možných cest.



Obr. 2.2: Průzkumné programování

2.3. RUP (Rational Unified Process), jeho cykly, fáze a iterace

Proces RUP je produktem výzkumného úsilí řady velkých firem, zabývajících se vývojem softwarových systémů, koordinovaných firmou Rational (odtud také vlastní název procesu). Process RUP definuje disciplinovaný přístup k přiřazování úkolů a zodpovědností v rámci vývojové organizace. Jeho cílem je zajistit vytvoření produktu vysoké kvality požadované zákazníkem v rámci predikovatelného rozpočtu a časového rozvrhu.

V čem je tento proces odlišný oproti výše zmíněnému vodopádovému modelu? Jakým způsobem odtraňuje uvedené nedostatky? Odpověď spočívá v dodržení následujících několika principů:

- softwarový produkt je vyvíjen iteračním způsobem
- jsou spravovány požadavky na něj kladené
- využívá se již existujících softwarových komponent
- model softwarového systému je vizualizován
- průběžně je ověřována kvalita produktu
- změny systému jsou řízeny.

2.3.1 Vývoj softwaru iteračním způsobem

V současné období, 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.

2.3.2. 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 nedostatečné specifikaci zadání. Proces RUP popisuje jak požadavky na softwarový systém doslova vylákat od zadavatelů, jak je organizovat a následně i dokumentovat. Monitorování změn v požadavcích se stává nedílnou součásti vývoje stejně jako správně dokumentované požadavky sloužící ke komunikaci mezi zadavateli a týmem řešitelů.

2.3.3. Vývoj pomocí komponent

Proces RUP poskytuje systematický přístup k *definování architektury využívající nové či již existující komponenty*.

Tyto komponenty jsou vzájemně propojovány v ráci korektně definované architektuře buď 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 usílí dalšímu vývoji v této oblasti. Vývoj softwaru se tak přesouvá do oblasti skládání produktu z prefabrikovaných komponent.

2.3.4. Vizualizace modelování softwarového 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 považovaný jazyk UML (Unified Modeling Language) primárně určený pro účely modelování softwarových systémů.

2.3.5. 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.

2.3.6. Řízení změn

Co rozumíme pod pojmem řízení změn:

Řízení změn umožňuje zaručit, že každá změna je přijatelná a všechny změny systému jsou sledovatelné.

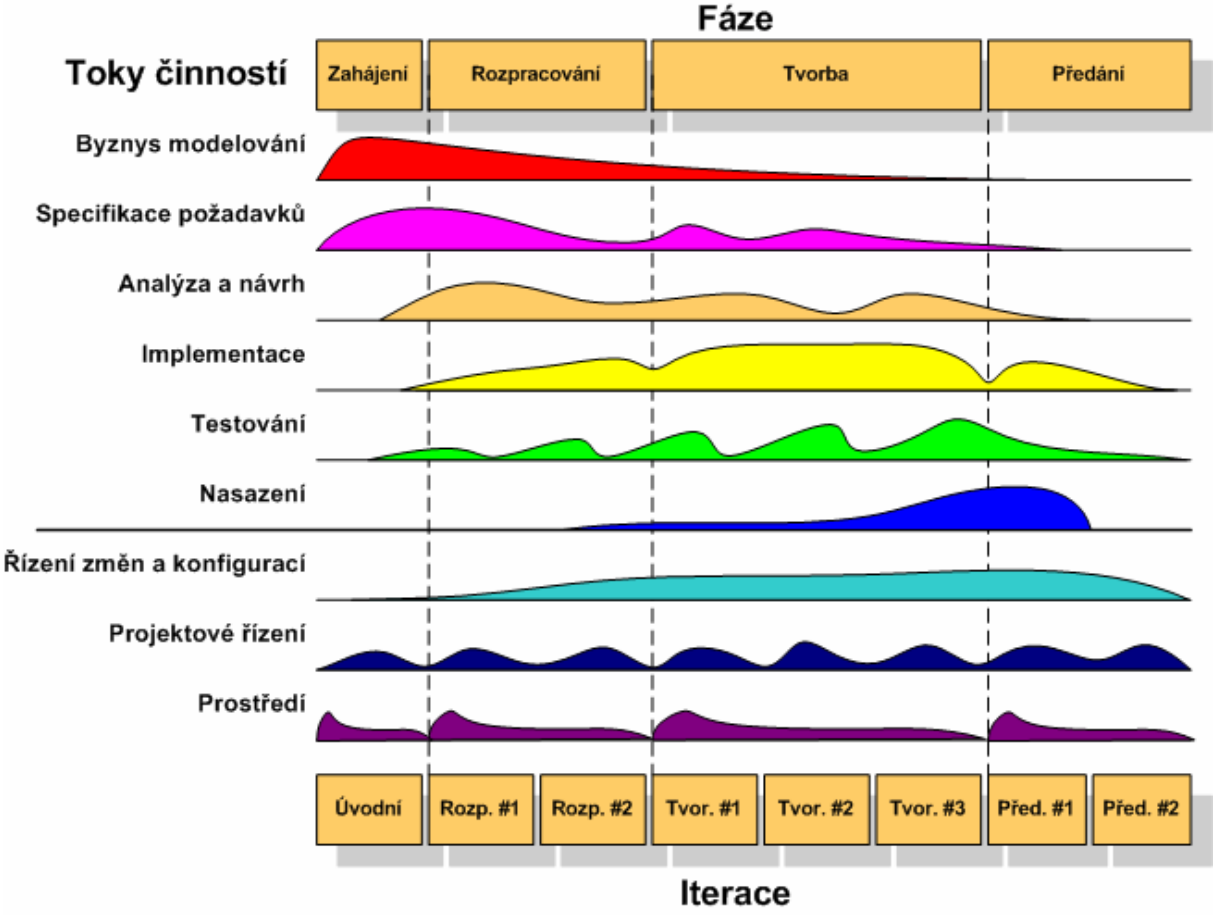
Důvod, proč je kladen na tuto problematiku takový důraz spočívá v tom, že prostředí ve kterém je softwarový systém vyvíjen podléhá častým a mnohdy i radikálním změnám, které je nutno integrovat do vlastního řešení.

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.

Příklad:
Předpokládejme, že firma vyvíjí informační systém podniku zahrnující mzdovou agendu zaměstnanců. V okamžiku, kdy dojde ke změně zákonů v této oblasti, musí být zřejmé, která část systému musí být změněna a jak se promítne i do ostatních substystémů. Správně navržený systém řízení pak dokáže lokalizovat kdo tyto změny zajistí, přičemž bude minimálně ovlivněno okolí těchto pracovníků. Např. tým softwarových inženýrů vyvíjející uživatelské rozhraní by takový zásah neměl vůbec pocítit.

2.3.7. Schématické vyjádření procesu RUP

Vlastní softwarový proces má svou statickou strukturu ve smyslu toho, z jakých se skládá toků činností (bude popsáno níže) a aktivit a na druhou stranu má také svou dynamickou stránku popisující jak je softwarový produkt vyvíjen v čase (obr. 2.3).



Obr. 2.3: 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.

2.3.8. Cykly, fáze a 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.

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

Jinými slovy řečeno v rámci každé iterace proběhnou činnosti vázané na byznys modelování, následují specifikace požadavků, analýza a návrh, implementace, testování a rozmístění (instalace). K tomu probíhá celá řada činností z podpůrných toků (viz níže) týkajících se správy konfigurací, řízení projektu a přípravy prostředí ve kterém je systém vyvíjen a nasazen (obr. 2.4).

Obr. 2.4: Iterace vývoje produktu

2.3.9. Statická struktura procesu

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:

  Model je 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.).

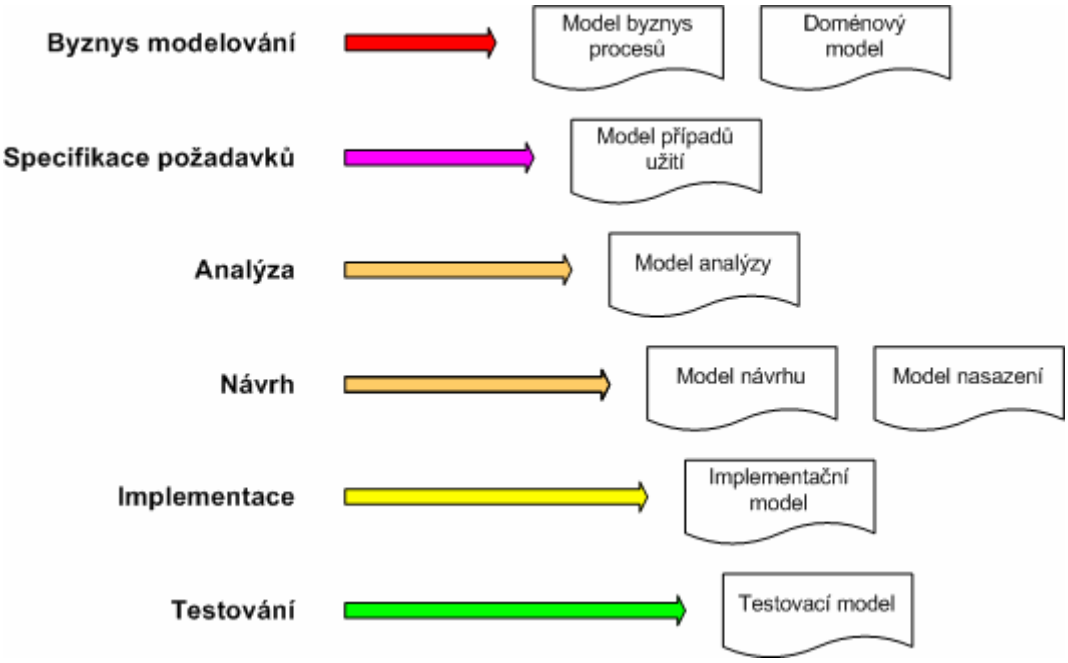
2.4. Základní a podpůrné toky činností

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.

Výstupem těchto základních toků činností jsou modely nahlížející na vytvářený systém z daného pohledu abstrakce (zjednodušení) (obr. 2.5).



Obr. 2.5: Toky činností a modely jimi vytvářené

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

- *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.

2.5 Jazyk UML

Poslední částí kapitoly věnované softwarovému procesu je úvod do problematiky jazyka UML (Unified Modeling Language), který slouží k vytváření výše uvedených modelů vznikajících v průběhu realizace požadovaného produktu. V průběhu let se UML stal standardizovaným jazykem určeným pro vytvoření výkresové dokumentace (softwarového) systému:

UML je jazyk umožňující specifikaci, vizualizaci, konstrukci a dokumentaci artefaktů softwarového systému.

Co rozumíme pod jednolivými charakteristikami jazyka UML:

- Specikace vyjadřuje zásadu vytvoření přesných, jednoznačných a úplných modelů softwarového procesu.
- Vizualizace znamená, že se jedná o grafický jazyk.
- Konstrukce odpovídá požadavku přímého napojení jazyka na širokou škálu programovacích jazyků.

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
2. *Logický náhled*
    a. Diagram tříd
    b. Objektový diagram
3. *Dynamický náhled popisující chování*
    a. Stavový diagram
    b. Diagram aktivit
    c. Interakční diagramy
        i. Sekvenční diagramy
        ii. Diagramy spolupráce
4. *Implementační náhled*
    a. Diagram komponent
    b. Diagram rozmístění

Všechny tyto diagramy budou představeny v následujících kapitolách, kde budou popsány jednolivé základní toky činností včetně artefaktů, které jsou jimi vytvářeny.

Shrnutí

V této kapitole byl čtenář obeznámen s problematikou softwarového procesu, jakožto po částech uspořádané posloupnosti aktivit vedoucí k vytvoření softwarového díla. Jako referenční model softwarového procesu byl použit proces RUP (Rational Unified Process), který reprezentuje v současné době jakýsi standard zahrnující to nejlepší, co se v dané oblasti používá u velkých softwarových firem. Proces RUP byl popsán ze dvou hledisek – dynamického a statického. První z obou rozděluje vývoj softwaru do cyklů, kde na konci každého z nich je úplná verze systému určeného k distribuci. Cykly samotné jsou tvořeny čtyřmi fázemi (zahájení, rozpracování, tvorba a předání), přičemž každá z nich obsahuje iterace postupným způsobem rozšiřující vytvářený produkt až do výsledné podoby. Statický aspekt je definován množinou základních a podpůrných toků činností zahrnující pracovníky a jimi vytvářené artefakty. Jako jazyk používaný pro potřeby specifikace, vizualizace a konstrukce softwaru byl vybrán jazyk UML (Unified Modeling Language). Tento jazyk slouží k popisu vyvíjeného systému ze čtyř základních pohledů – funkčního, statického, dynamického a implementačního.

Kontrolní otázky

1. Jak je definován softwarový proces?
2. Z kterých základních fází se skládá vodopádový model softwarového procesu?
3. K čemu slouží jazyk UML?

Úkoly

1. Nakreslete schéma procesu RUP definujte co je to cyklus, fáze a iterace!

K zamyšlení

Představte si, že jste majitelem softwarové firmy. Pokuste se sestavit tým z Vašich kolegů a slovně popsat, jak by jste řešil projekt informačního systému půjčovny videokazet a DVD.

3. Byznys modelování

V této kapitole se dozvíte:

- čím se zabývá tok činností nazývaný jako byznys modelování
- jaké diagramy jazyka UML se používají k specifikaci podnikových procesů a doménového modelu
- jaké jsou alternativní přístupy k byznys modelování

Doba potřebná k prostudování této kapitoly jsou:

- 3 hodiny

Smyslem byznys modelování je poskytnout společný jazyk pro komunity softwarových inženýrů a odborníků z oblasti podnikání.

Modely, které jsou v rámci daného toku činností vytvářeny jsou dvou typů:

- *Modely podnikových procesů* popisující množinu vzájemně propojených procedur a aktivit vedoucí ke splnění podnikatelského cíle.
- *Doménový model* postihující nejdůležitější objekty vyskytující se v kontextu daného systému. Doménovými objekty rozumíme entity existující v prostředí, ve kterém funguje daný systém.

Než se podíváme na problematiku vytváření těchto modelů, definujme se příklad na kterém si budeme demostrovat celý postup vývojem softwarového systému z pohledu jak procesního přístupu, tak z pohledu použití jazyka UML.

Příklad:
Předpokládejme, že majitel autosalónu má zájem na zvýšení efektivity fungování svého podniku a rozhodne se pro investici do vybudování informačního systému. Cílem bude evidovat zákazníky, automobily, jejich výbavení, dostupnost a vše ostatní, co s touto problematikou souvisí. Dále se předpokládá, že systém bude umět komunikovat s informačním systémem výrobce z důvodu zkrácení lhůt dodávek.

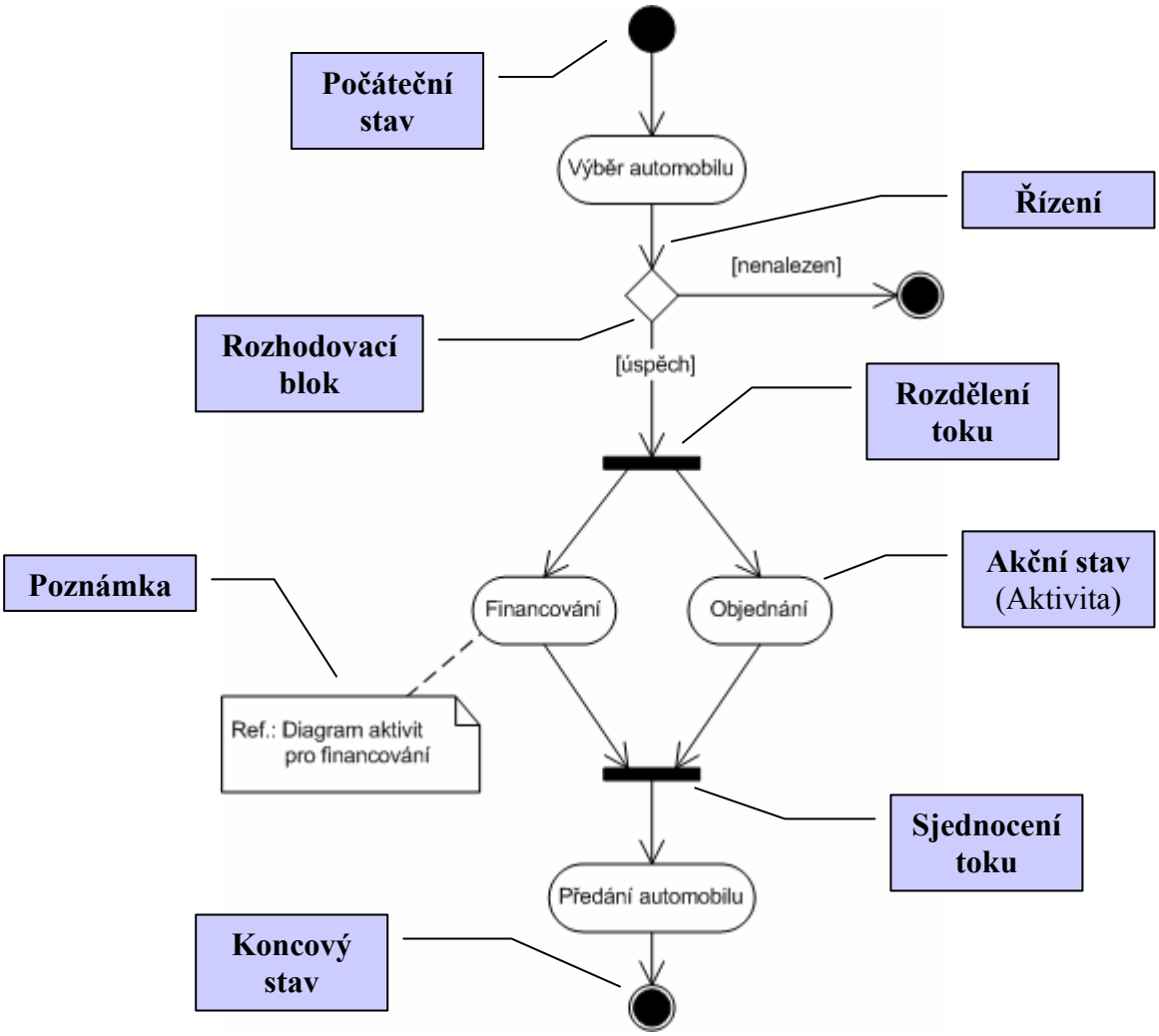
3.1. Diagramy aktivit a tříd jazyka UML

Jazyk UML pro potřeby specifikace byznys modelů využívá dvou typů diagramů:

- *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.
- *Diagram tříd* je graf tvořený pracovníky a věcmi vzájemně propojenými statickými (v čase se neměnnými) vazbami. Účelem je zachytit, na rozdíl od předchozího diagramu, statický aspekt modelované domény (oblasti řešení).

3.1.1. Diagram aktivit

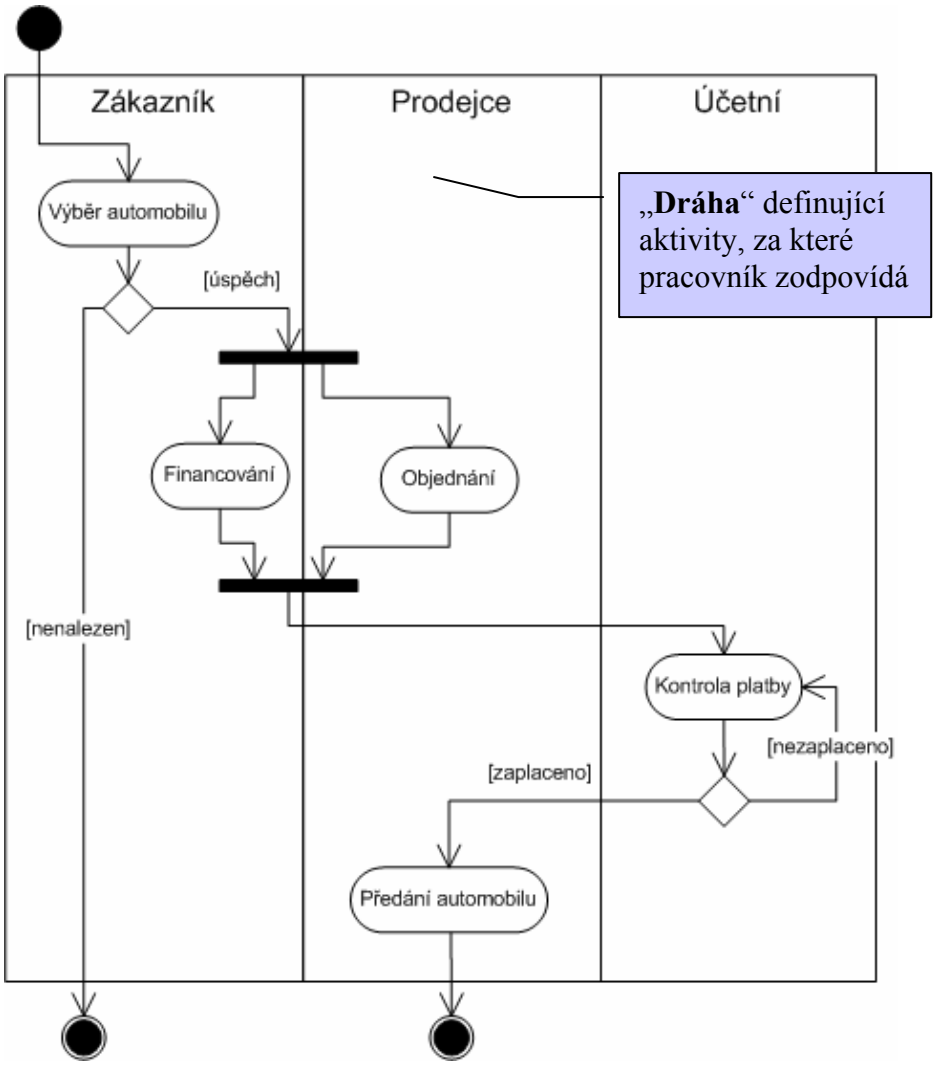
Diagram aktivit popisuje jednolivé procesy pomocí aktivit reprezentujících jeho (akční) stavy a přechody mezi nimi. Jaká je notace tohoto diagramu lze demonstrovat na příkladu procesu prodeje automobilu (obr. 3.1.1).



Obr. 3.1.1: Diagram aktivit popisující prodej automobilu

Příklad (pokr.):
Prodej automobilu je zahájen aktivitou jeho výběru. V případě, že požadovaný automobil je v nabídce autosalónu je tok činností rozdělen do dvou souběžných vláken týkajících se financování (diagram pro tento podproces může být řešen samostatným diagramem) a objednání automobilu ze skladu, případně u výrobce. Po ukončení těchto vláken pokračuje proces předáním automobilu zákazníkovi.

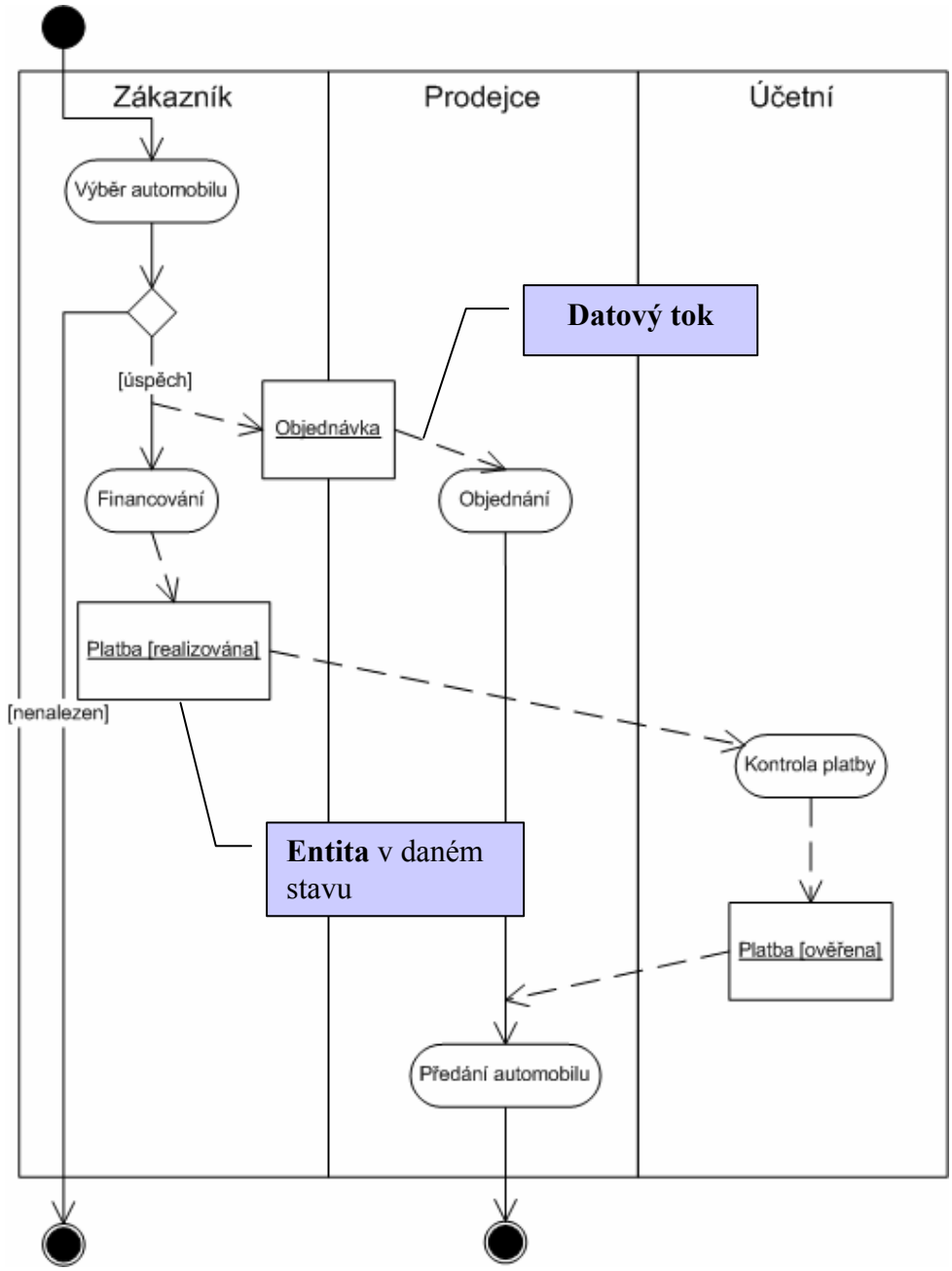
Takto definovaný diagram aktivit popisuje tok činností, nicméně není z něj zcela zřejmé, kdo tyto činnosti realizuje a kdo za ně zodpovídá. K tomuto účelu UML nabízí možnost zavedení tzv. drah vztažených k jednotlivým pracovníkům a obsahující pouze ty aktivity, za které tento pracovník zodpovídá. Náš původní diagram pak dostává novou strukturu odrážející výše uvedené (obr. 3.1.2).



Obr. 3.1.2: Zodpovědnosti pracovníků v diagramu aktivit

Oproti minulému diagramu byl tento doplněn o aktivitu kontroly platby a byly zřejmým způsobem definovány zodpovědnosti jednolivých pracovníků realizujících tento proces.

Definování drah umožňuje do diagramu aktivit zahrnout také pracovníky. Jediné, co v takovém diagramu chybí, jsou entity (v softwarovém procesu nazývané jako artefakty), které jsou při jeho realizaci vytvářeny, upravovány nebo spotřebovávány. I pro toto rozšíření diagram aktivit poskytuje nástroje v podobě notace modelující takové entity. Každá taková entita může mít kromě svého jména také název stavu, ve kterém se nachází (obr. 3.1.3).

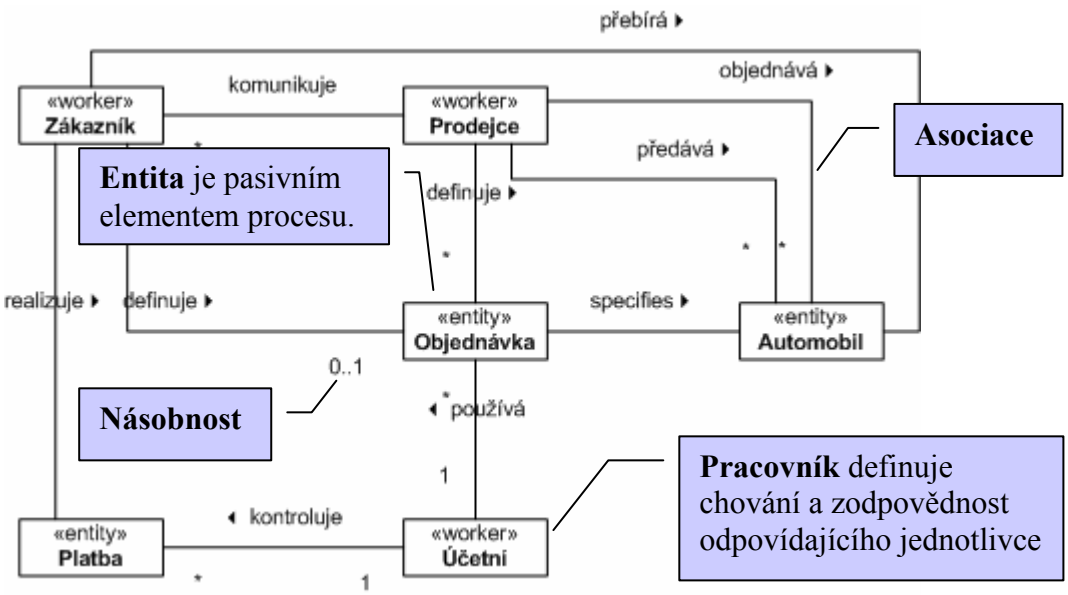


Obr. 3.1.3: Modelování entit a jejich stavů v diagramu aktivit

Příklad (pokr.):
Prodej automobilu je zahájen aktivitou jeho výběru. V případě, že požadovaný automobil je v nabídce autosalónu je tok činností rozdělen do dvou souběžných vláken týkajících se financování produkující entitu platby. Druhé vlákno je spuštěno entitou objednávky reprezentující vstup do aktivity objednání automobilu ze skladu, případně u výrobce. Spuštění aktivity předání automobilu následující za objednáním je umožněno v okamžiku, kdy aktivita ověření platby změní platbu z původního stavu realizována do stavu ověřena.

3.1.1. Diagram tříd

Předchozí diagramy byly zaměřeny především na popis dynamiky (chování) organizační jednotky. Prostřednictvím drah zodpovědnosti a entit byly zavedeny do těchto diagramů také strukturální elementy. Pro další porozumění problematice je tedy vhodné podchytit tuto statickou strukturu pomocí speciálního diagramu – diagramu tříd (obr. 3.4).



Obr. 3.1.4: Diagram tříd

Diagram tříd doménového modelu je tvořen dvěma typy elementů reprezentujících aktivně působící pracovníky označené klíčovým slovem <<worker>> a pasivními entitami označené jako <<entity>>. Tyto elementy jsou vzájemně propojeny vazbami (asociacemi) vyjadřující konceptuální nebo fyzické spojení těchto elementů. Násobnost těchto asociací vyjadřuje četnost výskytu elementu (hvězdička vyjadřuje obecnou četnost n).

Příklad (pokr.):
Prodej automobilu potřebuje tři aktivní aktéry procesu nazvané jako Zákazník, Prodejce a Účetní, které v diagramu reprezentuje element pracovník. Zákazník musí komunikovat s prodejcem, aby bylo možné definovat pasivní element (entitu) nazvaný jako Objednávka. Tato objednávka specifikuje jaký automobil má být předán zákazníkovi v případě, že tento realizuje svou platbu a tato platba je podle objednávky zkontrolována účetním. Protože může nastat i situace, že zákazník nenajde v nabídce autosónu svůj automobil, je u objednávky definována násobnost 0..1. Jinými slovy, produktem komunikace mezi zákazníkem a prodejce je maximálně jedna objednávka. Na druhou stranu účetní nekontroluje pouze jednu platbu, ale všechny platby, které jsou realizovány i jinými zákazníky. Odtud obecná násobnost u entity Platba v rámci asociace *kontroluje*.

3.2. Metoda BPM (Business Process Modeling)

Model procesu je pomocí metody BPM sestavován ze tří relativně nezávislých pohledů tvořených funkčním, objektovým a koordinačním modelem.

1. *Funkční model* slouží k identifikaci architektury procesu včetně jeho zákazníků a produktů. Cílem je nalezení odpovědi na otázku jaké procesy organizace potřebuje k zajištění své podnikatelské činnosti a jaká je jich struktura.
2. *Objektový model* identifikuje statickou strukturu obsahující všechny entity (objekty), které v rámci procesu vystupují. Z pohledu jejich aktivní účasti pak rozlišujeme mezi aktivními a pasivními objekty. Aktivní objekty jsou realizátoři procesu (lidé a zařízení jimi používané), zatímco pasivní objekty jsou procesem manipulovány, vytvářeny či spotřebovávány. Vzájemné relace dokumentují vztahy (interakce), do kterých objekty vstupují. Jestliže funkční model se snaží odpovědět na otázku jaké procesy v podniku existují, pak tento model se snaží odhalit kým a čím je proces realizován.

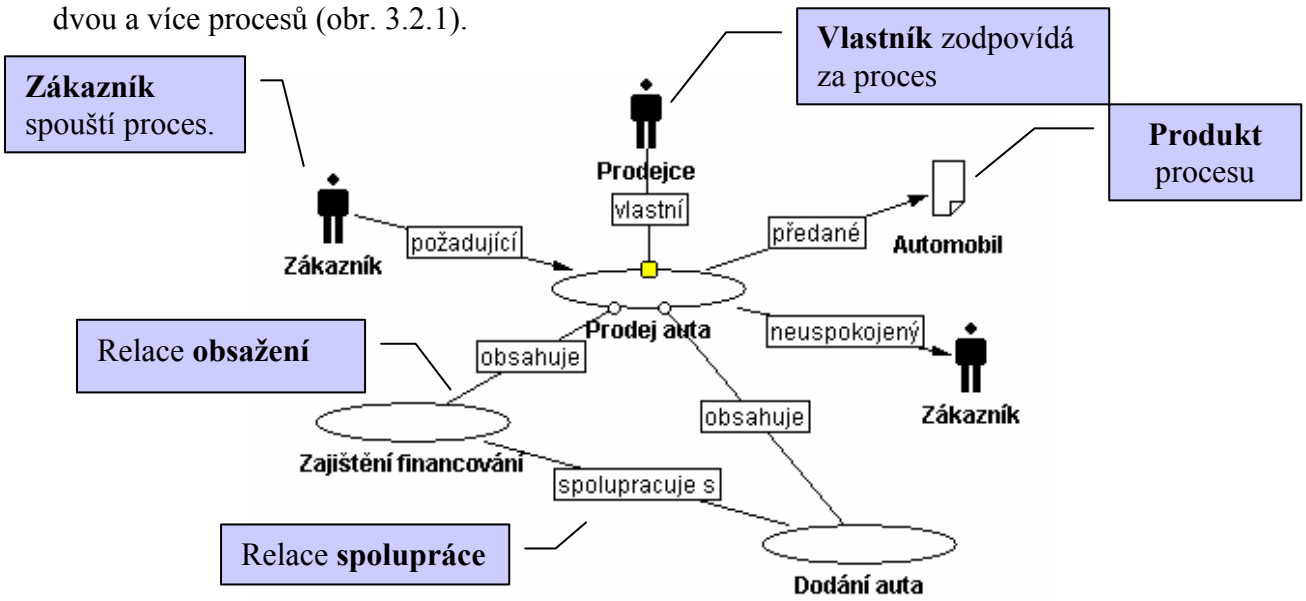
3. *Koordinační model* vychází z předchozích dvou a popisuje jak bude proces realizován cestou specifikace a koordinace aktivit reprezentujících interakce mezi objekty.

Podívejme se na tyto jednolivé pohledy podrobněji v následujících kapitolách.

3.2.1. Funkční model

Hlavním účelem funkčního modelu je identifikace architektury (kostry) podnikového procesu, jeho zákazníků a produktů.

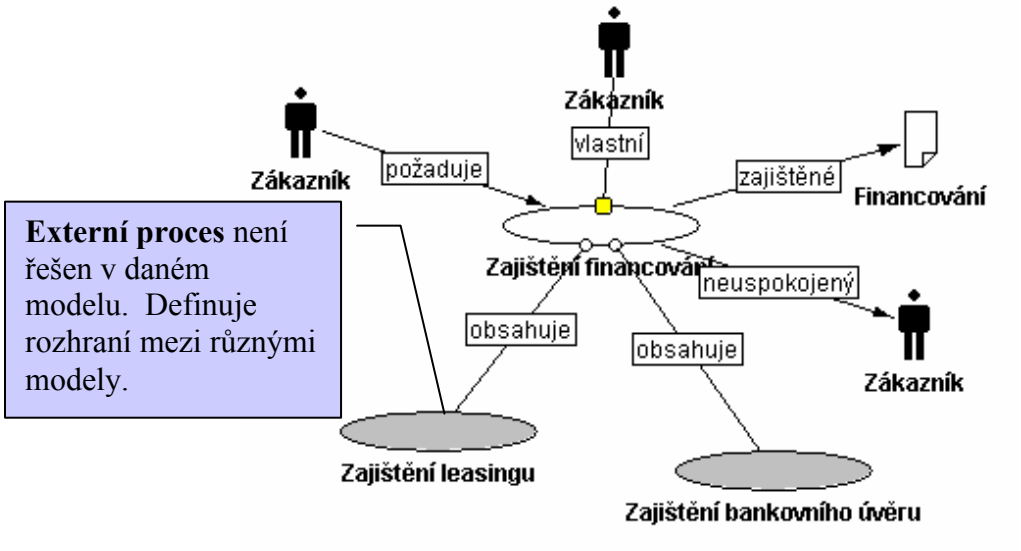
Z hlediska existence vztahů mezi procesy metoda BPM zavádí dvě základní relace - *obsažení* a *spolupráce*. Prvá z nich slouží k identifikací podprocesů, kterým je v určitém okamžiku předáváno řízení, zatímco druhá relace identifikuje možnost souběžné existence a realizace dvou a více procesů (obr. 3.2.1).



Obr. 3.2.1: Funkční model

Příklad (pokr.):
Zákazník požadující automobil je procesem uspokojen, tedy produktem procesu je předaný automobil, nebo odchází z autosalónu jako neuspokojený. Vlastní proces prodeje auta obsahuje dva podprocesy – zajištění financování a proces dodání auta. Oba podprocesy spolupracují, neboť produkt procesu zajištění financování je nutný jako vstup dodání auta.

Je vcelku zřejmé, že stejným způsobem jsou rozpracovány oba podprocesy, které mají své zákazníky, produkty, případně i podprocesy (obr. 3.2.2).

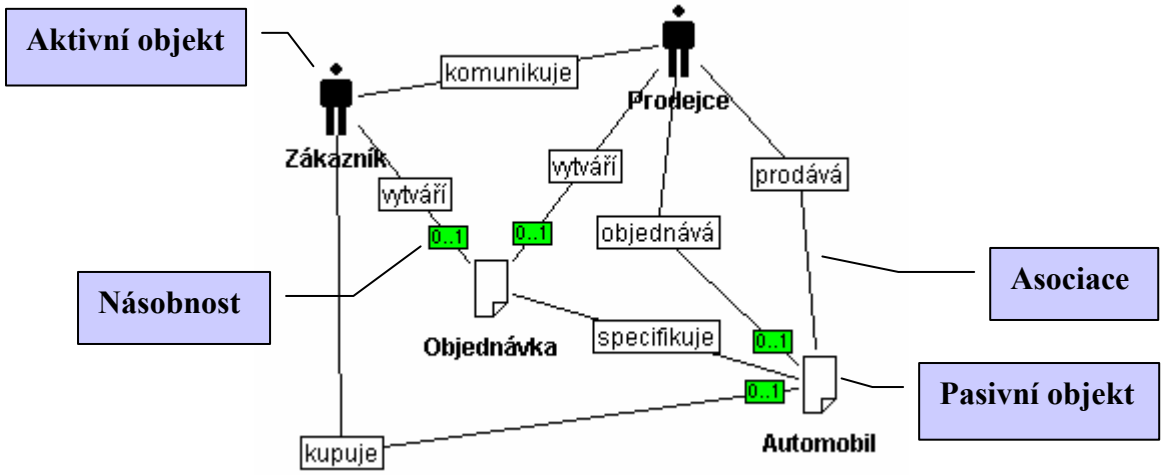


Obr. 3.2.2: Rozpracování podprocesu Zajištění financování

3.2.2. Objektový model

Objektový model identifikuje statickou strukturu všech objektů (aktivních a pasivních) nutných pro realizaci procesu.

Obektový model je sestavován pro každý proces definovaný ve funkčním modelu zvláště. V našem případě bychom tedy měli mít specifikovány objektové modely pro procesy Prodej auta, Zajištění financování a Dodání auta. Elementy, vztahy i násobnosti znázorňované těmito modely mají stejný význam jako v případě jazyka UML diagram tříd (obr. 3.2.3).

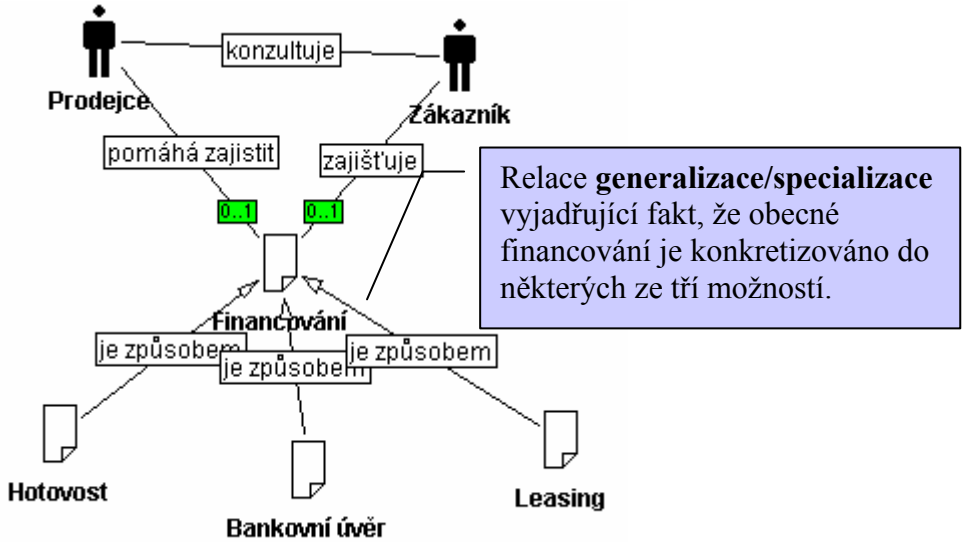


Obr. 3.2.3: Objektový model procesu Prodej auta

Příklad (pokr.):
Je vcelku zřejmé, že zákazník musí s prodejcem komunikovat. Výsledkem jejich společného úsilí může být objednávka obsahující veškerou požadovanou informaci definovanou atributy tohoto pasivního objektu. Tímto způsobem je specifikován automobil objednaný prodejcem a zakoupený zákazníkem. Samozřejmě, že tento model obsahuje i variantu, kdy požadovaný automobil není k dispozici. Tento alternativní scénář je podchycen v diagramu objektového

modelu kardinalitou 0..1, čili objednávka je buďto vytvořena nebo vůbec nebude existovat, pak ale také nedojde ani ke koupi automobilu.

Proces zajišťující financování bude mít objektový model definovaný z pohledu objektů a vazeb vztažených k tomuto procesu (obr. 3.2.4).



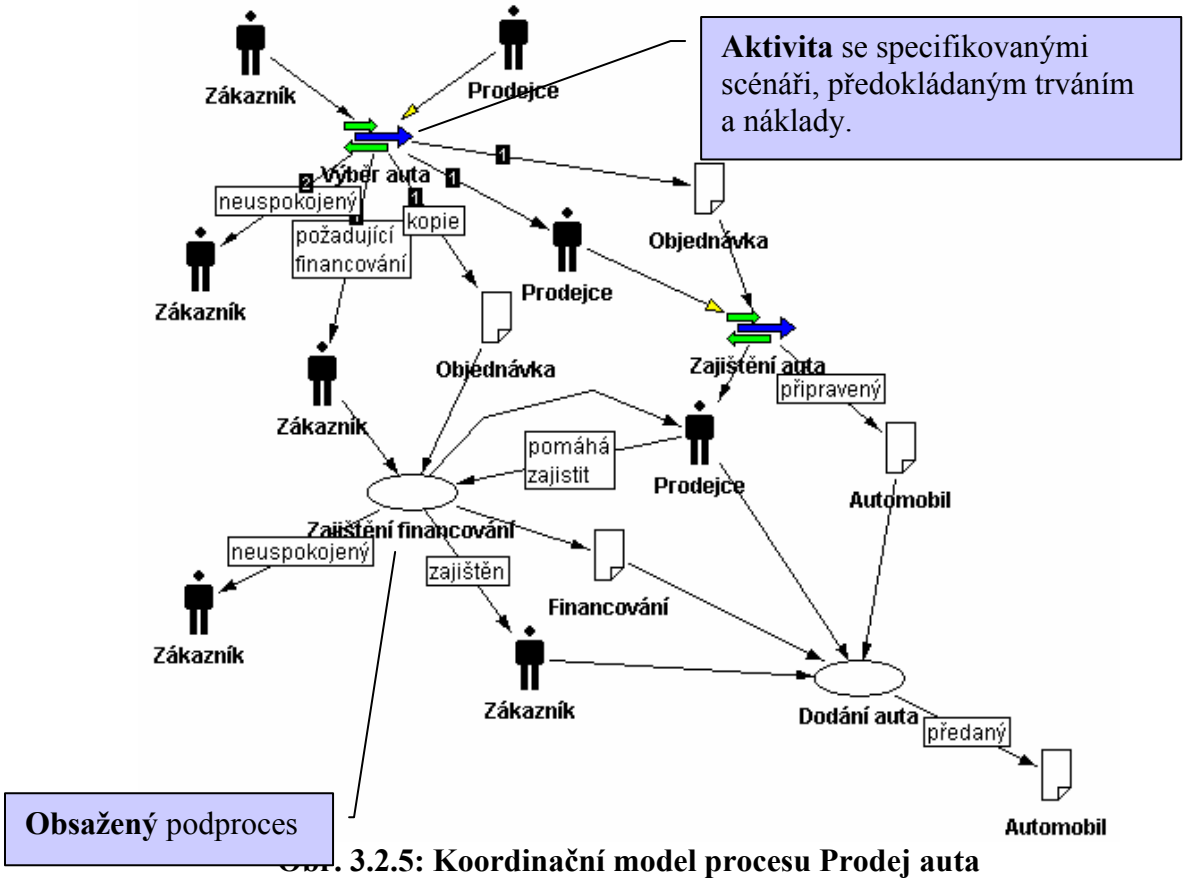
Obr. 3.2.4: Objektový model procesu Zajištění financování

3.2.3. Koordinační model

Koordinační model vyjadřuje jak bude proces realizován. Koordinační model specifikuje interakce mezi objekty a definuje způsob jejich synchronizace.

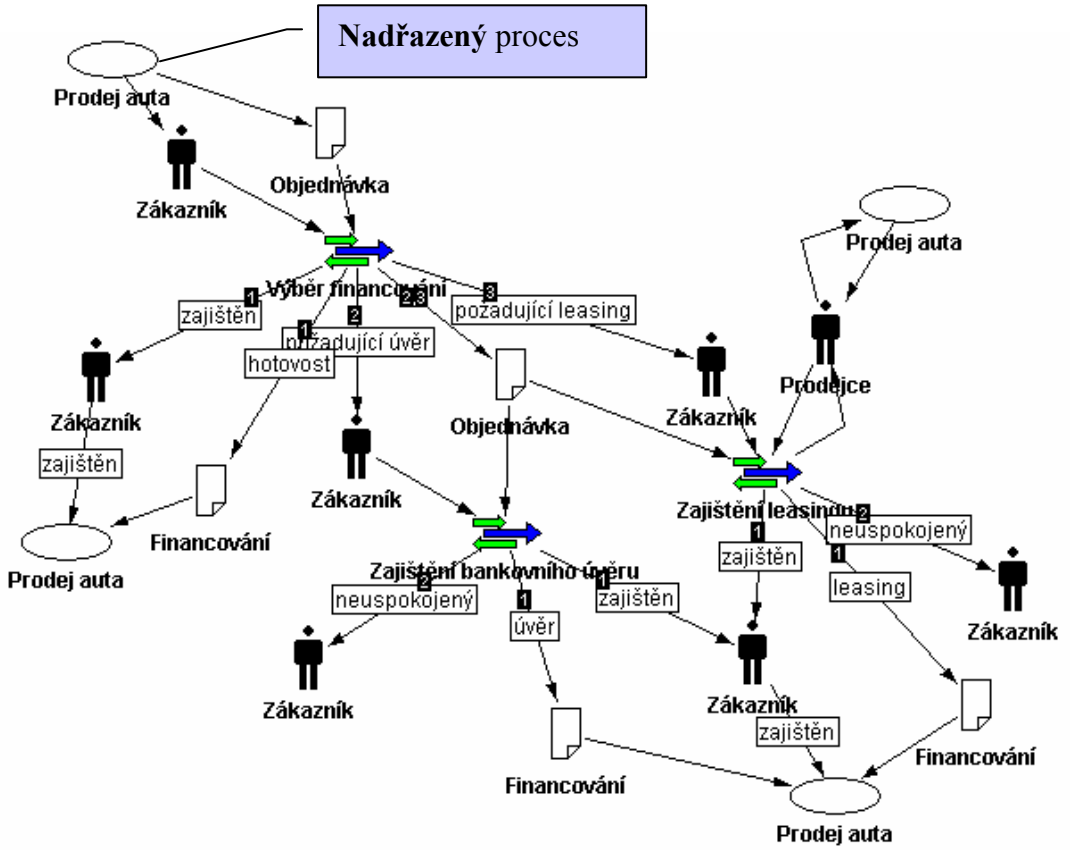
Podstatou mechanismu koordinace je splnění podmínky, že aktivita může být spuštěna pouze za podmínky připravenosti všech požadovaných objektů na vstupu do aktivity. Aktivita definuje alternativní scénáře svého průběhu, přičemž každý ze scénářů má určen své náklady a predikovaný čas realizace. Na základě architektury procesu identifikované funkčním modelem jsou atomické aktivity koordinačního modelu doprovázeny ikonami podprocesů umožňujících vnořování do dalších, podrobnějších koordinačních diagramů (obr. 3.2.5).

Příklad (pokr.):
Celý proces prodeje automobilu je zahájen aktivitou výběru auta, která ke svému spuštění vyžaduje přítomnost zákazníka i prodejce. Jak je patrné tato aktivita má dva možné scénaře svého průběhu. První (označený návěštím 1) odpovídá situaci, že zákazník najde požadovaný automobil a výsledkem je specifikovaná objednávka. Druhý scénář ukončuje celý proces a zákazník odchází jako neuspokojený. Pokračováním prvého scénáře je rozdělení toku do dvou souběžně prováděných větví. Prodejce se pokouší na základě specifikované objednávky zajistit automobil přímo ze skladu, případně musí zajistit dodávku přímo od výrobce. Souběžně s touto činností se zákazník pokouší o zajištění financování reprezentované odpovídajícím podprocesem. Zřejmě analogickým způsobem je rozpracován i podproces finacování, který je tvořen alternativami pro financování hotovostí, leasingem nebo bankovním úvěrem. Pouze v případě, že všechny požadované objekty (pasivní i aktivní) jsou připraveny na vstupech posledního z podprocesů dodání auta, lze tento realizovat a ukončit tak celý proces prodeje auta uspokojením zákazníka.



Obr. 3.2.5: Koordinační model procesu Prodej auta

Analogicky lze sestavit koordinační model i pro podproces financování (obr. 3.2.6).



Obr. 3.2.6: Koordinační model procesu Zajištění financování

Tímto způsobem definovaný proces využívající formálních, na matematické teorii Petriho sítí založených, přístupů umožňuje provádět analýzy a simulace. Cílem je ověřit korektnost takto specifikovaných procesů.

Kromě uvedených metod existuje celá řada jiných zaměřená na problematiku byznys modelování. Za všechny alespoň ty nejdůležitější, kterými jsou metoda IDEF (Integrated DEFinition vyvinutá pro U.S. Air Force a dostupná na adrese http://www.idef.com) a metoda profesora Scheera nazvaná ARIS (Architecture of Integrated Information Systems dostupná na adrese http://www.ids-scheer.com).

Shrnutí

V této kapitole byl čtenář obeznámen s problematikou byznys modelování. Hlavním cílem tohoto toku činností je pochopit doménovou oblast, porozumět zadavatelům projektu a nalézt v rámci specifikovaných procesů, které jejich části budou řešeny pomocí implementovaného softwarového systému. Jinými slovy řečeno, specifikace funkcionality budovaného systému by měla vycházet z modelů podnikových procesů. Čtenář byl dále seznámen s nástroji jazyka UML sloužícími těmto účelům (diagram aktivit a tříd). Dále byla popsána metoda BPM postavená na třech základních náhledech – funkčním, objektovém a koordinačním.

Úkoly

1. Popište vlastními slovy co vyjadřuje objektový model na obrázku 3.2.4.!
2. Popište vlastními slovy co vyjadřuje koordinační model na obrázku 3.2.6.!

3. Specifikujte pomocí UML hlavní procesy půjčovny videokazet a DVD!

4. Specifikace požadavků

V této kapitole se dozvíte:

- čím se zabývá tok činností nazývaný jako specifikace požadavků
- jaké diagramy jazyka UML se používají k specifikaci požadavků

Doba potřebná k prostudování této kapitoly jsou:

- 3 hodiny

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

4.1. Aktéři a funkční specifikace pomocí případů použití

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

- *Aktéry* definující uživatele či jiné systémy, kteří budou vstupovat do interakce s vyvíjeným softwarovým systémem.
- *Případy užití* specifikující vzory chování realizovaných softwarovým systémem. Každý případ užití lze chápat jako posloupnost vzájemně navazujících transakcí vykonaných v dialogu mezi aktérem a vlastním softwarovým systémem.

Jazyk UML pro potřeby sestavení modelů specifikace požadavků využívá dvou typů diagramů:

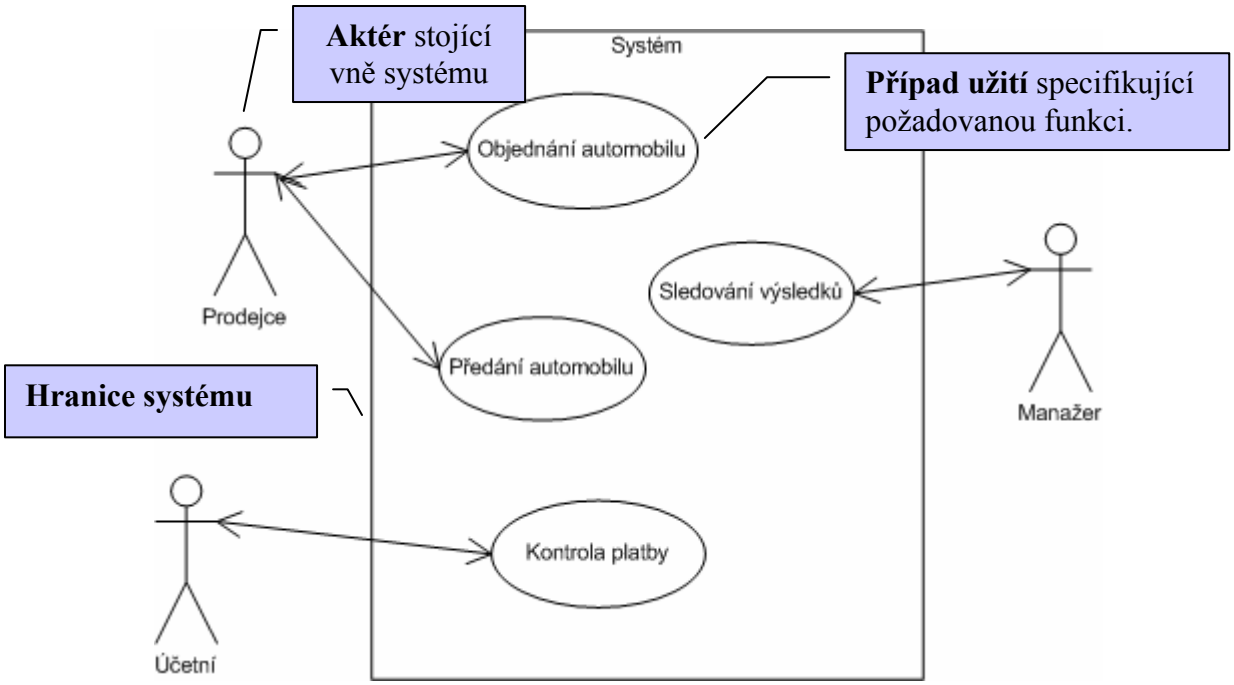
- *Diagram případů užití* popisující vztahy mezi aktéry a jednotlivými případy použití.
- *Sekvenční diagram* zobrazující vzájemnou interakci participujících objektů organizovanou podle časového hlediska.

4.2. Diagramy případů užití

Účelem diagramu případů úžití 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 cestou jejich softwarové implementace.

Notace používaná diagramy případů užití je tvořena grafickými symboly reprezentující aktéry a případy užití v jejich vzájemných vazbách (obr. 4.1).

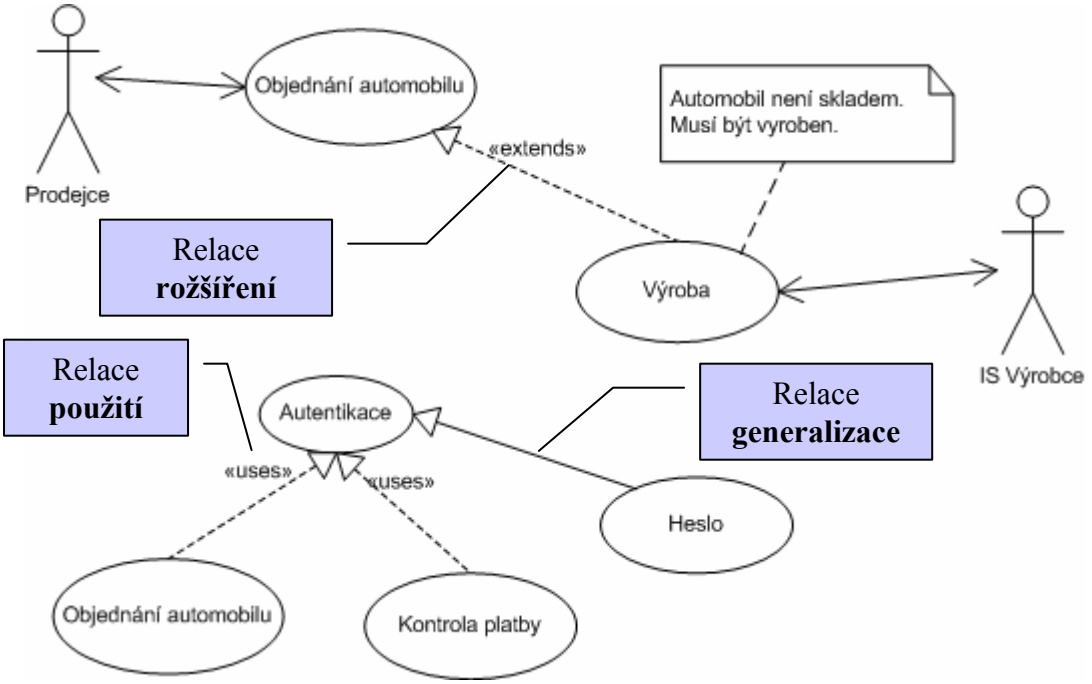


Obr. 4.1: Diagram případů užití

Příklad (pokr.):
Z procesu definovaného v minulé kapitole vyplývá, že aktivita objednání automobilu a jeho předání zákazníkovi jsou místa, která lze zajistit informačním systémem. V prvém případě prodejce bude moci ověřit, zda-li vybraný automobil je ve skladu, nebo zda-li je nutné jej objednat u výrobce. Stejně tak předání automobilu by mělo být evidováno s cílem zaznamenat jméno a adresu zákazníka, případně problémy, které s předání vznikly. Kontrola platby je aktivita zajišťovaná účetním. Ten by měl mít možnost nahlédnout do seznamu provedených plateb a ověřit jejich správnost. Vše by mělo být monitorováno manažerem autosalónu, který by měl mít k dispozici dosažené hospodářské výsledky. Z analýzy procesu prodeje auta tak vyplývají následující funkce – případy užití, které by měl softwarový systém implementovat: Objednání automobilu, Předání automobilu, Kontrola platby a Sledování výsledků.

Je vcelku zřejmé, že čím více a různorodějších podnikových procesů bude organizace realizovat, tím složitější a obsáhlejší budou požadované případy užití. Vzniká tak nutnost tyto případy užití strukturovat a zpřehlednit. Diagram případů užití k tomuto účelu zavadí tři typy relací mezi jednotlivými případy užití (obr. 4.2):

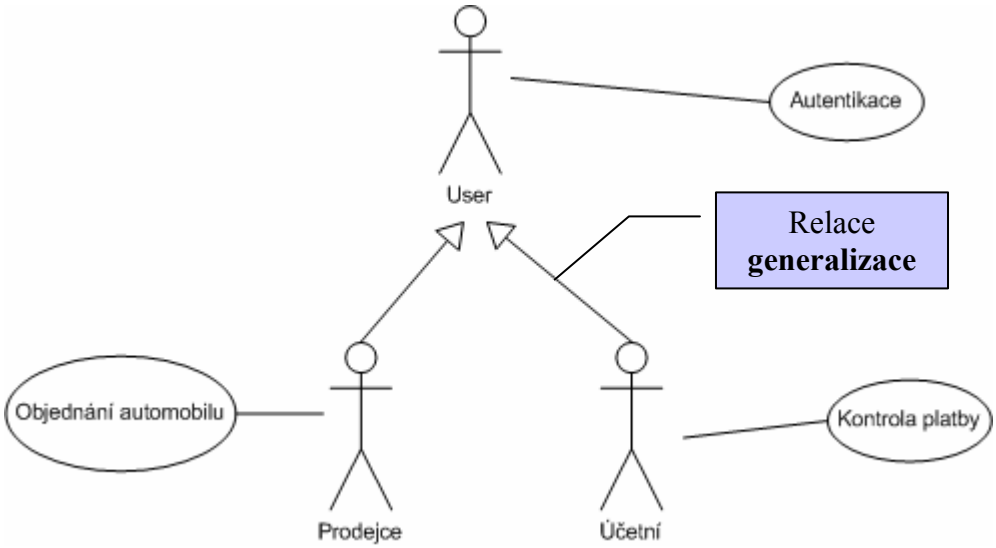
- Relace *používá* označovaná klíčovým slovem <<uses>> vyjadřuje situaci, kde určitý scénář popsaný jedním případem užití je využíván i jinými případy užití.
- Relace rozšiřuje označovaná klíčovým slovem <<extends>> vyjadřuje situaci, kde určitý případ užití rozšiřuje jiný či představuje variantní průchody jím popsaným scénářem.
- Relace zobecnění/specializace vyjadřuje vztah mezi obecnějším případem užití a jeho speciálním případem.



Obr. 4.2: Strukturace případů užití

Příklad (pokr.):
Případ užití objednání automobilu může být v určitý okamžik, kdy automobil není skladem, rozšířen o scénář objednání ve výrobě, jehož aktérem je informační systém výrobce automobilu. Případy užití objednání a kontroly platby vyžadují autentikaci popsanou samostatným případem užití a konkretizovanou do podoby autentikace pomocí hesla uživatele.

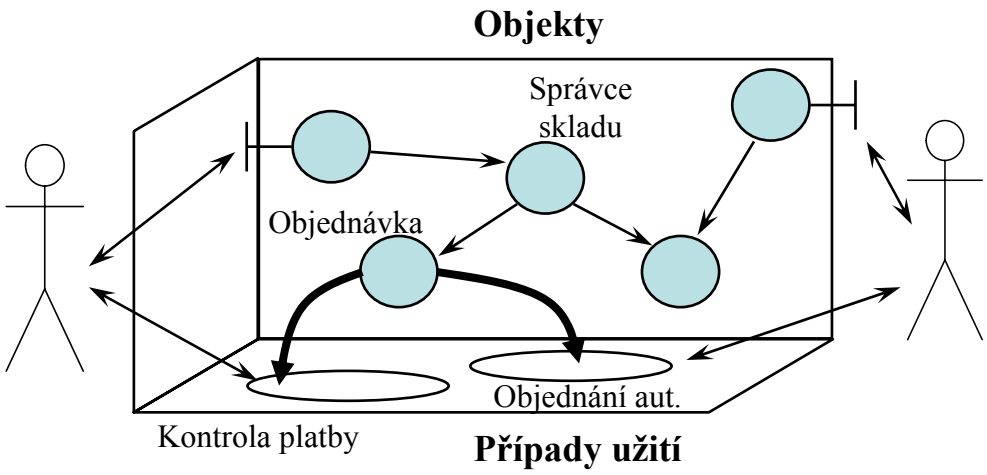
Kromě toho, že je možné strukturovat případy užití, lze obdobný přístup aplikovat i na samotné aktéry. Relace, která se k tomu používá je relace *generalizace*, jejímž smyslem je k jednotlivým případům užití přiřadit i soubor schopností, které daný aktér musí zvládat. Obecnější schopnosti jsou přiřazeny větší skupině aktérů, přičemž jejich specializací tyto obecné schopnosti doplňujeme o další konkrétnější (obr. 4.3).



Obr. 4.3: Strukturace aktérů

Příklad (pokr.):
Prodejce i účetní v našem příkladu mají společné schopnosti se přihlásit k informačnímu systému autosalónu pomocí jim přiřazeného hesla.

Pomocí uvedených diagramů jsme tedy dokázali specifikovat funkcionalitu reprezentovanou jednotlivými případy užití. Dalším krokem, který je nutné učinit je, jak tato funkcionalita bude řešena, kdo ji zajistí. V úvodu kapitoly jsme zmínili pojem objekt, o kterém jsme tvrdili, že právě v interakci s jinými objekty dokáže realizovat jednotlivé scénáře. Cílem tedy bude identifikovat v našem systému objekty, které budou zodpovídat za plnění požadované funkcionality poskytováním svých služeb v komunikaci s jinými objekty (obr. 4.4).



Obr. 4.4: Objekty a případy užití

Než si ukážeme jak probíhá výše zmíněná interakce mezi objekty realizující specifikované případy užití, musíme se zastavit u pojmu objekt a důsledně jej definovat.

4.3. Definice pojmu objekt

Objekt je identifikovatelná samostatná entita daná svou:

- ***identitou*** *- jedinečností umožňující ji odlišit od ostatních,*
- ***chováním*** *- službami poskytovanými v interakci s ostatními objekty.*

Kromě těchto primárních vlastností vyjádřených v definici má objekt také sekundární vlastnosti, kterými jsou:

- *atributy* – (v čase se měnící) datové hodnoty popisující objekt,
- *doba existence* – časový interval daný okamžikem vzniku a zániku objektu,
- *stavy* odrážející různé fáze doby existence objektu.

Náhled na objekt může být dvojí: vnější a vnitřní. Vnějším pohledem rozumíme množinu zodpovědností implementovaných pomocí služeb, které objekt poskytuje. Tyto služby jsou formálně specifikovány tzv. protokolem zpráv, které můžeme na objekt zaslat. Dále objekt může vnějšímu světu poskytnout své vztahy k ostatním objektům a také některé ze svých atributů. Vniřní pohled pak umožnuje nahlédnout na všechy atributy, které jsou před vnějškem objektu ukryty a na způsob jak jsou jednolivé služby implementovány.

Příklad:

Považujme automobil za objekt. Každý jednolivý automobil má svou jedinečnost, identitu, která jej odlišuje od ostatních i v případě, že má shodné hodnoty atributů jako jsou např. barva, šířka, výška, rozvor apod. Smyslem, zodpovědností, automobilu je dopravit pasažéry z místa A do místa B prostřednictví služeb jízda, rozjezd, zastavení apod. Tyto služby využívá řidič cestou „zasílání zpráv“ pomocí volantu, pedálů a řadící páky. Vnější náhled na automobil je stejný u všech typů (jsou řízeny stejným způsobem), různě jsou však tyto služby implementovány (vnitřní pohled). Otáčení volantem může být s posilovačem nebo bez něj, brždění může být monitorováno systémem ABS a podobně. Doba života automobilu začíná jeho vyrobením a končí jeho recyklací. Během svého života prochází různými stavy jako jsou jede, stojí, garážovaný, v servisu a tak dále.

4.4. Vztahy mezi objekty a jejich interakce

Softwarový systém není tvořen jediným objektem, ale celou jejich řadou. Tyto objekty mezi sebou komunikují s cílem splnit požadovanou funkcionalitu. Aby tyto objekty mohly mezi sebou komunikovat je nutné aby byly vzájemně mezi sebou propojeny:

Spojení je fyzická nebo konceptuální vazba mezi objekty.

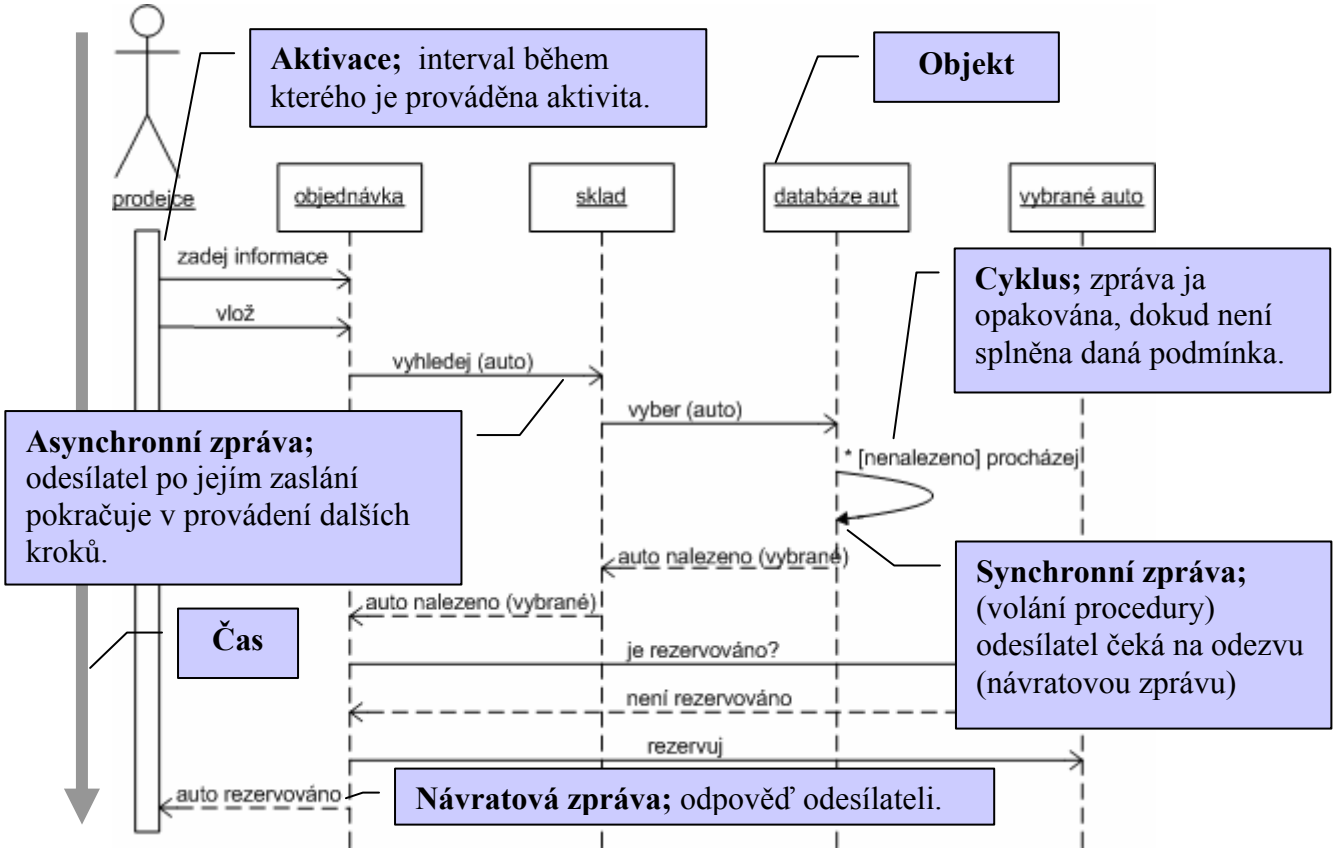
Podíváme-li se na knihu ležící na stole, pak se jedná o fyziké spojení mezi objekty kniha a stůl. Příkladem konceptuálního spojení může být obsah tvrzení, že student (objekt) Petr Nadaný studuje na VŠB – TU Ostrava.

Z hlediska objektově orientovaného přístupu je důležité si uvědomit, že právě množina vzájemně propojených objektů tvoří systém. Díky těmto spojením může dojít k vzájemné interakci mezi objekty vedoucích:

- k výslednému požadovanému chování systému jako celku,
- ke změnách v konfiguraci a stavech objektů a tím i celého systému.

Ve vztahu k definovaným případům užití je nutné definovat takové interakce mezi objekty, které povedou ke splnění jejich funkcionality, účelu ke kterému byly navrženy.

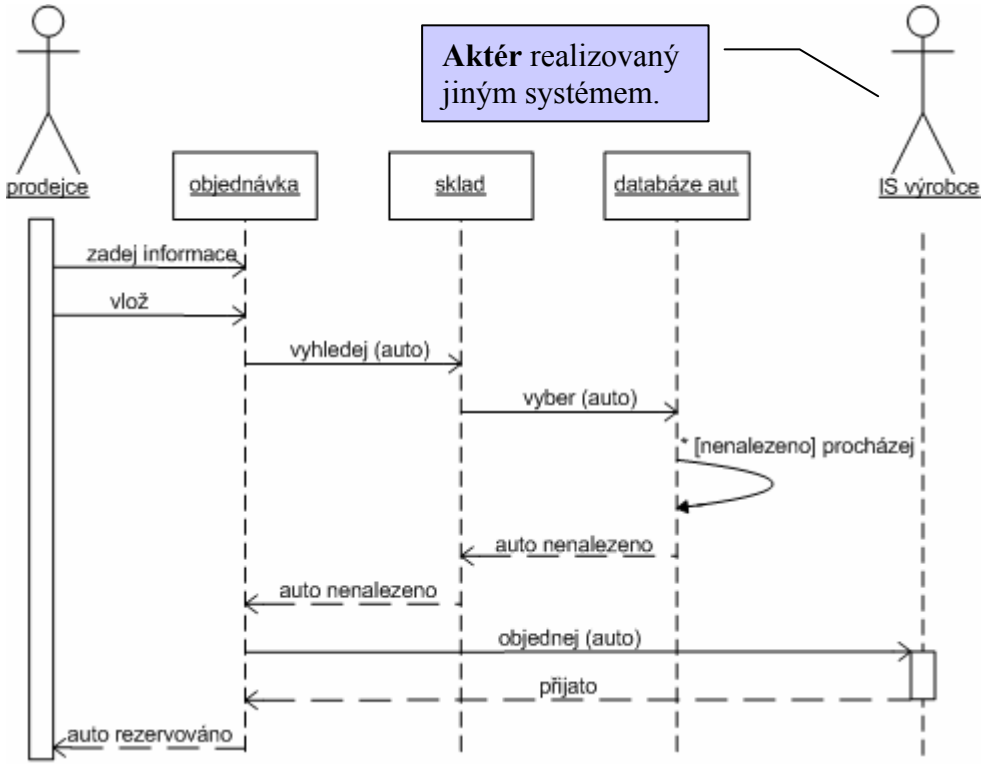
Jazyk UML poskytuje pro účely zaznamenání těchto vzájmených intrakcí tzv. sekvenční, někdy také nazývaný interakční, diagram. Tento diagram postihuje jaké zprávy (požadavky) jsou mezi objekty zasílány z pohledu času (obr. 4.5). Diagram je tvořen objekty uspořádanými do sloupců a šipky mezi nimi odpovídají 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 (příklad prodejce z obrázku 4.5). 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

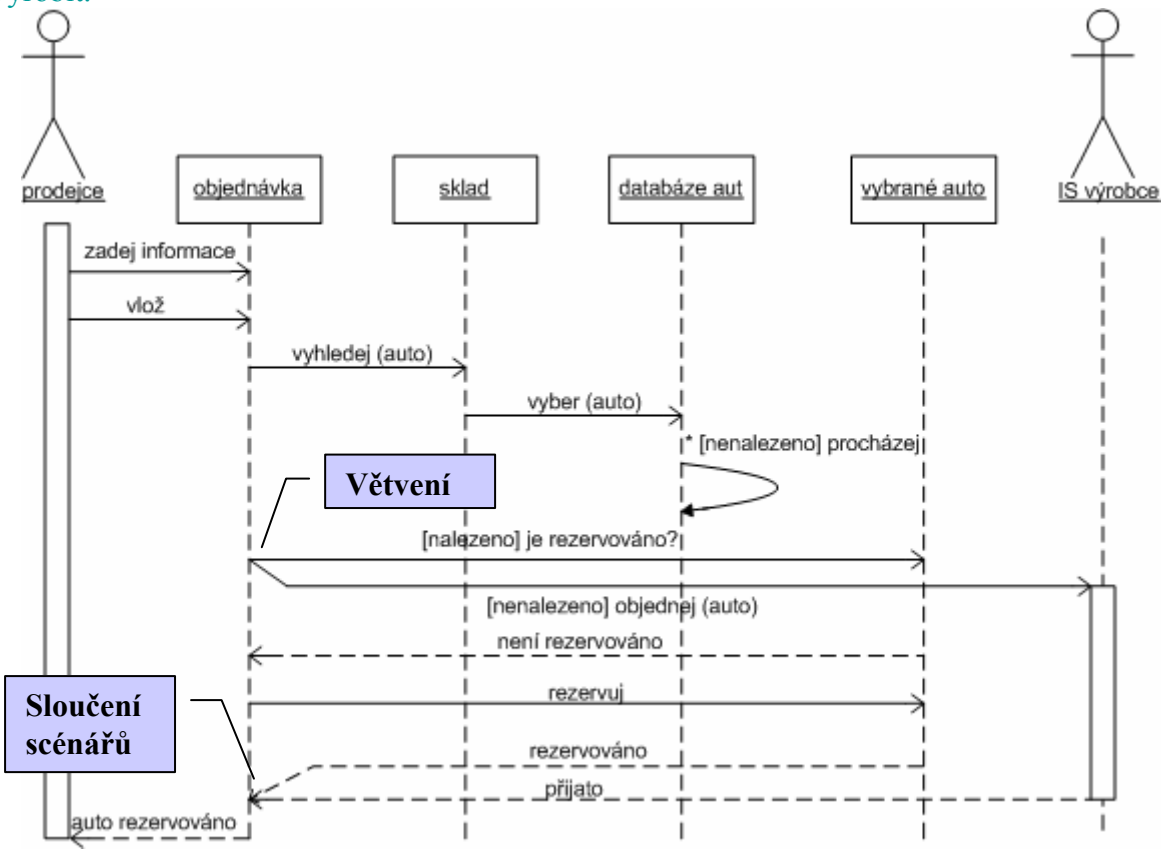
Příklad (pokr.):
V našem příkladu prodeje automobilu jeden z možných scénářů případu užití Objednání automobilu je realizován interakcí mezi prodejcem (aktér) a objekty objednávky, skladu, databáze aut a vybraným autem. Vzájemnou interakci zahajuje prodejce zadáním informace o požadovaném automobilu do objektu objednávka. Požadavkem (zasláním zprávy) vlož objednávka reaguje zprávou vyhledání daného automobilu (argument zprávy) ve skladu. Sklad tento požadavek předává databázi cestou zaslání zprávy vyber. Tato vybírá auto opakováním (vyjádřeno znakem hvězdičky *) operace procházej, dokud specifikované auto není nalezeno. Nalezené auto je označené jako vybrané a předáno pomocí návratových zpráv zpět k objektu objednávka. Objednávka nejprve požádá o ověření, zda auto není již rezervováno a pokud tomu tak skutečně není, provede jeho rezervaci zasláním zprávy rezervuj. Po ukončení této operace předá objednávka zpět k prodejci cestou návratové zprávy informující o tom, že auto bylo rezervováno

Z uvedeného textu vyplývá jeden zřejmý nedostatek. Tím je zachycení právě jediné možnosti z celé řady možných variant vzájemné interakce. Zjevně by sekvenční diagram vypadal jinak v případě, že požadovaný automobil se v databázi aut nenachází. Řešení této situace je možné dvěma způsoby. První spočívá ve vytvoření tolika diagramů, kolik je variant daného scénáře nebo spojit všechny scénáře do jediného diagramu. V případě druhého řešení se jedná určitě o úspornější řešení, nicméně obvykle za cenu snížené čitelnosti diagramu. Z tohoto důvodu je vhodné používat tento přístup u situací, kdy variant scénáře je jen velmi omezené množství. V našem příkladu obr. 4.6. popisuje variantní scénář samostaným diagramem, zatímco obr. 4.7 popisuje spojení dvou variant do jediného sekvenčního diagramu.



Obr. 4.6: Alternativní scénář

Příklad (pokr.):
Varianta scénáře objednání automobilu může vyžadovat spolupráci od jiného aktéra (Informační systém výrobce) v případě, že požadovaný automobil není na skladu a musí se vyrobit.



Obr. 4.7: Spojení scénářů

Shrnutí

V této kapitole byl čtenář obeznámen s problematikou specifikace požadavků. Hlavním cílem tohoto toku činností je pochopit jakým způsobem se požadavky modelují pomocí tzv. případů užití a jak lze tyto případy užití stejně jako jejich aktéry strukturovat. Neméně důležitým bodem této kapitoly bylo zavedení pojmu objekt, definovaný jako entita daná svou jedinečností a chováním. Interakce objektů realizující požadovanou funkcionalitu specifikovanou v případech užití byla hlavní náplní závěru kapitoly. Jazyk UML pro tyto účely poskytuje tzv. sekvenční diagramy znázorňující vzájemnou komunikaci objektů z časové hlediska.

Úkoly

1. Popište vlastními slovy co vyjadřuje sekvenční diagram na obrázku 4.7!
2. Na základě sestavených diagramů procesů půjčovny videokazet a DVD definujte jednotlivé případy užití!

3. Sestavte sekvenční diagramy pro případ užití Zapůjčení videokazety z příkladu půjčovny videokazet a DVD!

5. Analýza a návrh

V této kapitole se dozvíte:

- čím se zabývá tok činností nazývaný jako analýza a návrh
- jaké diagramy jazyka UML se používají k vytvoření modelů analýzy a návrhu

Doba potřebná k prostudování této kapitoly jsou:

- 6 hodin

Cílem toku činností zabývající se analýzou a návrhem softwarového produktu je ukázat jakým způsobem bude produkt realizován v implementační fázi.

5.1. Modely a jejich diagramy

Modely, které jsou v rámci analýzy a návrhu vytvářeny jsou následující:

- *Model analýzy*, který zkoumá specifikované požadavky z pohledu objektů, které lze nalézt v problémové doméně.
- *Model návrhu* dále upřesňuje model analýzy ve světle skutečného implementačního prostředí. Model návrhu tak představuje abstrakci zdrojového kódu, jinými slovy řečeno, reprezentuje „výkresovou“ dokumentaci určující jak bude zdrojový kód strukturován a napsán.
- *Model nasazení* specifikuje topologii technických prostředků určených ke spuštění výsledného softwarového produktu.

Jako v předchozích tocích činností, tak i v analýze a návrhu jsou výše uvedené modely vytvářeny pomocí k tomu určených diagramů. V tomto případě se jedná o následující diagramy, přičemž prvé dva z nich jsem již poznali v předcházejících kapitolách popisujících úvodní etapy vývoje softwarového systému zabývající se byznys modelováním (diagram tříd) a specifikací požadavků (sekvenční diagram):

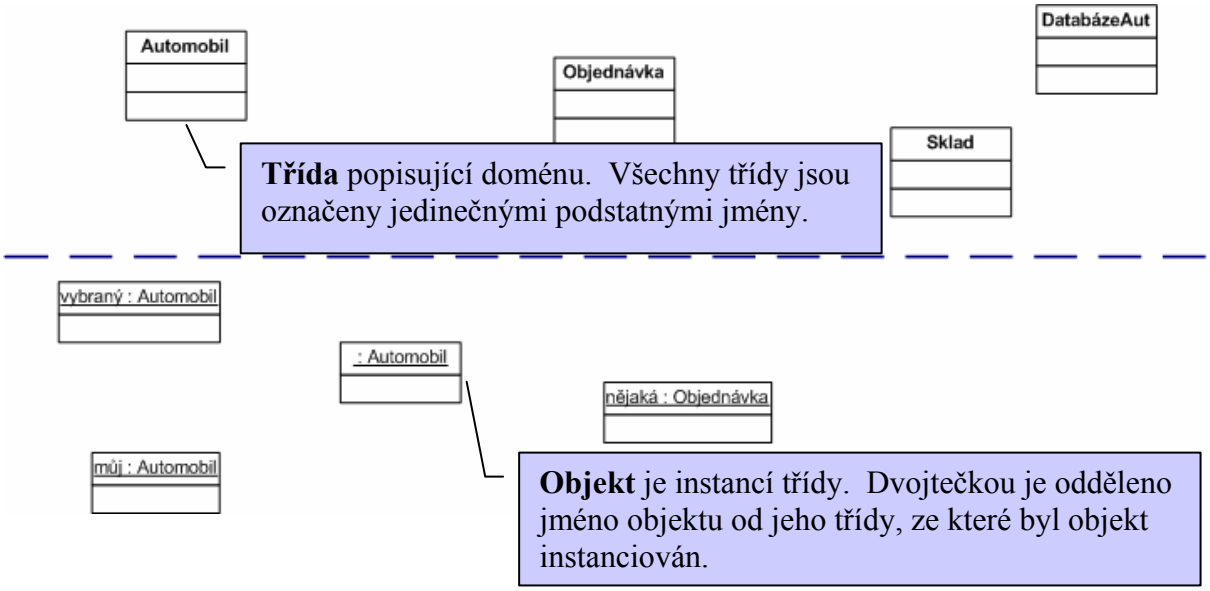
- *Diagram tříd* specifikující množinu tříd, rozhraní a jejich vzájemné vztahy. Tyto diagramy slouží k vyjádření statického pohledu na systém.
- *Sekvenční diagram* popisující interkace mezi objekty z hlediska jejich časového uspořádání.
- *Diagram spolupráce* je obdobně jako předchozí sekvenční diagram zaměřen na interkace, ale z pohledu strukturální organizace objektů. Jinými slovy není primárním aspektem časová posloupnost posílaných zpráv, ale topologie rozmístění objektů.
- *Stavový diagram* dokumentující životní cyklus objektu dané třídy z hlediska jeho stavů, přechodů mezi těmito stavy a událostmi, které tyto přechody uskutečňují.
- *Diagram nasazení* popisující konfiguraci (topologii) technických prostředků umožňujících běh vlastního softwarového systému.

5.2. Definice pojmu třída

V předcházejícím textu jsme se seznámili s pojmem třída. Nahraďme tedy zatím intuitivní chápání tohoto pojmu definicí následujícího znění:

Třída je popis množiny objektů mající společnou strukturu, chování, vztahy a sémantiku.

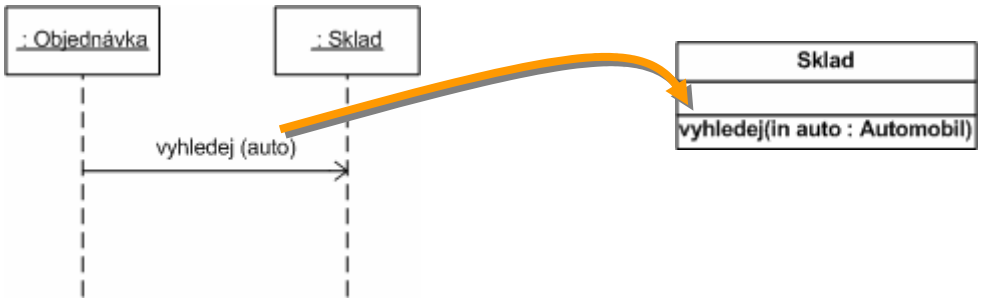
Třídy nalezneme v sekvenčím diagramu zkoumáním interakcí mezi objekty vystupujícími v sekvenčních diagram. Objekt může být instancí právě jedné třídy (stejně jako např. strojírenský výrobek může být vyroben pouze podle jediné výrobní dokumentace). Jména tříd volíme tak, aby byla součástí slovníku popisujících doménovou oblast (reinkarnace reálného světa do softwarového systému). Pojmy instance třídy a objekt můžeme zaměňovat, neboť vyjadřují v tomto kontextu totéž.



Obr. 5.2.1: Objekty a třídy

Příklad (pokr.):
V sekvenčním diagramu popisujícím proces prodeje auta vystupují objekty, které můžeme popsat třídami Automobil, Objednávka, Sklad, DatabázeAut apod. Tyto třídy (popisy) pak slouží jako předloha pro vytváření konkrétních objektů (instancí tříd). V našem příkladu tyto objekty můžeme pojmenovat a mít tak dvě instance třídy Automobil pojmenované jako *vybraný* a *můj* a jednu instanci třídy Automobil jako bezejmennou. Instance třídy Objednávka je jediná a pojmenovaná jako *nějaká*.

Popis chování objektů vytvářených podle daných tříd spočívá v definování jejich operací. Tyto operace, stejně jako třídy samotné, hledáme v sekvenčních diagramech. Zpráva zaslaná z jednoho objektu (instance třídy) na druhý znamená, že třídy tohoto druhého objektu musí definovat jak bude tento objekt reagovat na přijetí této zprávy.



Obr. 5.2.2: Identifikace operace

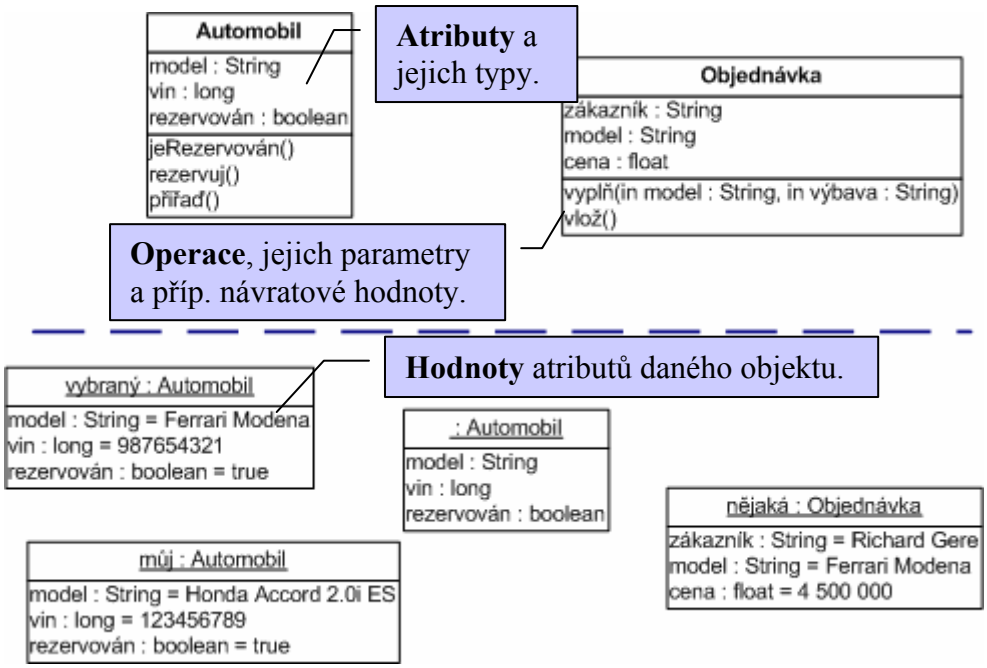
Příklad (pokr.):
V části sekvenčního diagramu popisující proces prodeje (nepojmenovaná) instance třídy Objednávka zasílá zprávu *vyhledej* s argumentem *auto* na instanci třídy Sklad. Odtud jednoznačně vyplývá, že třída Sklad popisující všechny objekty, které podle ní budou vytvořeny musí definovat operaci *vyhledej* se vstupním (*in*) argumentem *auto*, který je opět instancí, tentokráte třídy Automobil.

Struktura třídy je reprezentována svými atributy. Vyhledání atributů vychází ze zkoumání doménové oblasti a z její znalosti. Každý objekt, který je pak podle dané třídy vytvořen, tyto atributy naplňuje pro něj jedinečnými hodnotami. Výjimkou jsou tzv. atributy třídy, jejichž hodnota je společná všem instancím.

Obr. 5.2.3: Atributy třídy

Příklad (pokr.):
Z obecné znalosti prodeje automobilu vyplývá, že objednávka musí obsahovat jméno zákazníka, vybraný model a také cenu vybraného automobilu. Odtud vyplývají tři atributy *zákazník*, *model* a *cena*, které musí třída definovat a její instance naplnit konkrétními hodnotami.

Definováním operací a atributů vytváříme kompletní obraz tříd a jejich instancí (objektů), které budou tvořit základ diagramu tříd modelu analýzy a posléze i návrhu. Důležité je si uvědomit, že hodnoty atributů jsou vlastní vždy dané instaci, třída pouze definuje pojmenování těchto atributů a jejich typ. Třída také definuje operace, které jsou společné všem jejím objektům a tudíž se v notaci objektů již explicitně neuvádí (obr. 5.2.4).



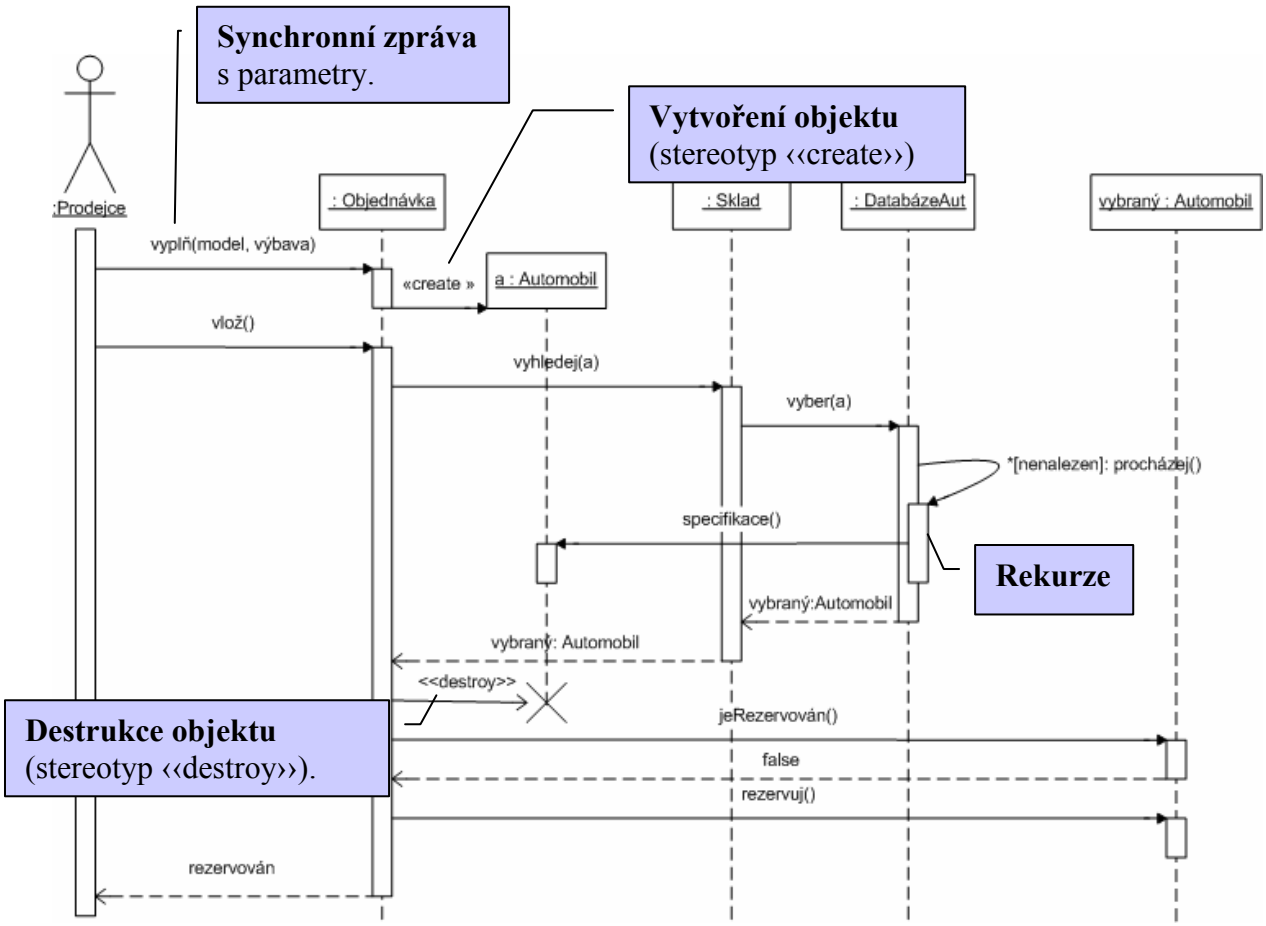
Obr. 5.2.4: Operace a atributy

5.3. Vztahy mezi třídami a objekty

Vztahy (relace) mezi třídami specifikují cestu, jak mohou objekty mezi sebou komunikovat.

Zdrojem pro definovaní těchto vztahů mezi třídami je opět sekvenční diagram, případně diagram spolupráce. Pokud totiž požadujeme aby objektu byla zaslána zpráva od jiného objektu, pak musí mezi těmito objekty existovat propojení, které je specifikováno v podobě vazby mezi třídami těchto objektů.

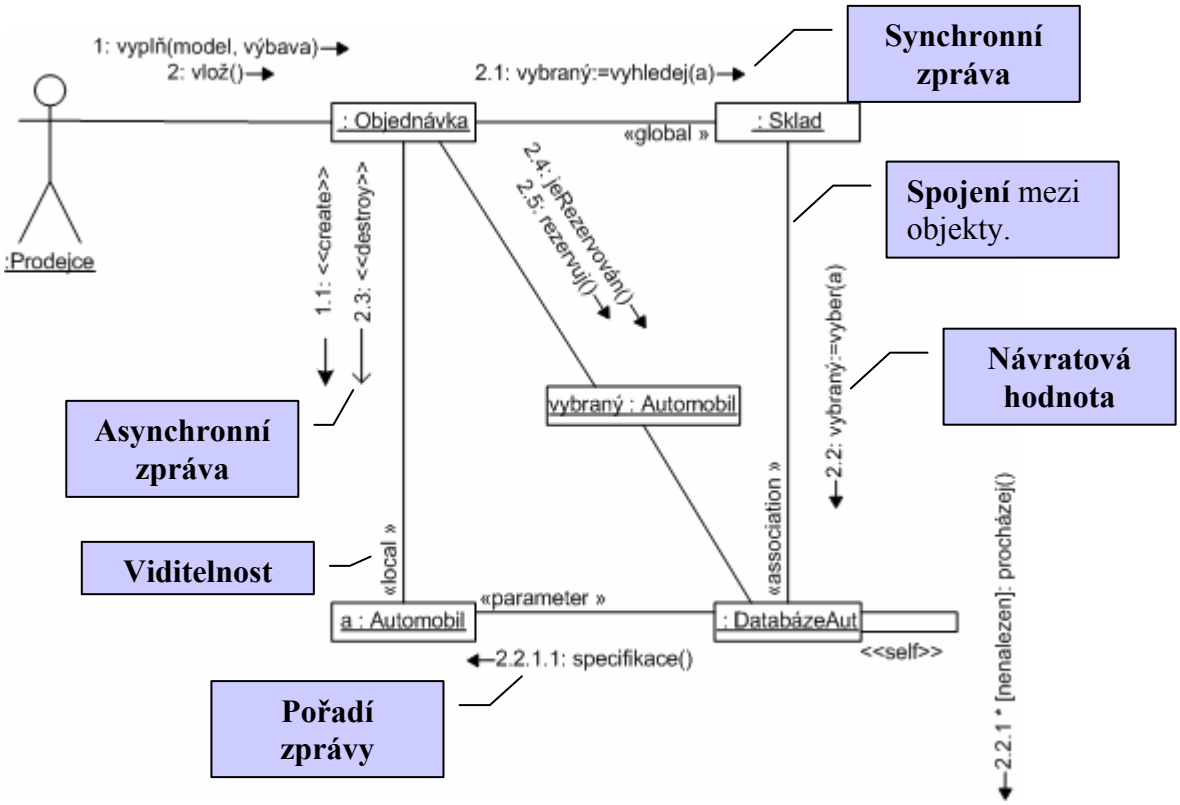
Než přistoupíme k další kategorizaci typů vztahů mezi třídami, ukažme si na upřesněném sekvenčním diagramu a diagramu spolupráce jak mezi sebou objekty komunikují. Jak již bylo dříve uvedeno, sekvenční diagram upřednostňuje časový aspekt zasílání zpráv.



Obr. 5.3.1: Upřesněný sekvenční diagram

Upřesněním sekvenčního diagramu rozumíme jeho další zpodrobnění ve smyslu zavedení trvání operací vyvolaných v reakci na přijetí zprávy (obdélníky na ose objektu), zavedení návratových hodnot (čárkované šipky), zavedení typu zpráv v podobě synchronních (čekajících na odpověď od adresáta zprávy) či asynchronních (pokračujících dále v provádění operace bez čekání na odpověď) a zavedení tak zvaných stereotypních (stále se opakujících) zpráv typu vytvoření instance třídy, případně její destrukce.

Diagram spolupráce upřednostňuje při popisu vzájemné komunikace mezi objekty jejich topologii, tedy jejich rozložení a vzájemné spojení.



Obr. 5.3.2: Diagram spolupráce

Diagram spolupráce příkladu prodeje automobilu popisuje tutéž situaci jako předcházející sekvenční diagram. Časová posloupnost zaslání zpráv je vyjádřena jejich pořadovým číslem. Návratová hodnota je vyjádřena operátorem přiřazení :=. Opakované zaslání zprávy je dáno symbolem * a v hranatých závorkách uvedením podmínky opakování cyklu. Navíc tento diagram zavádí i následující typy viditelnosti vzájemně spojených objektů:

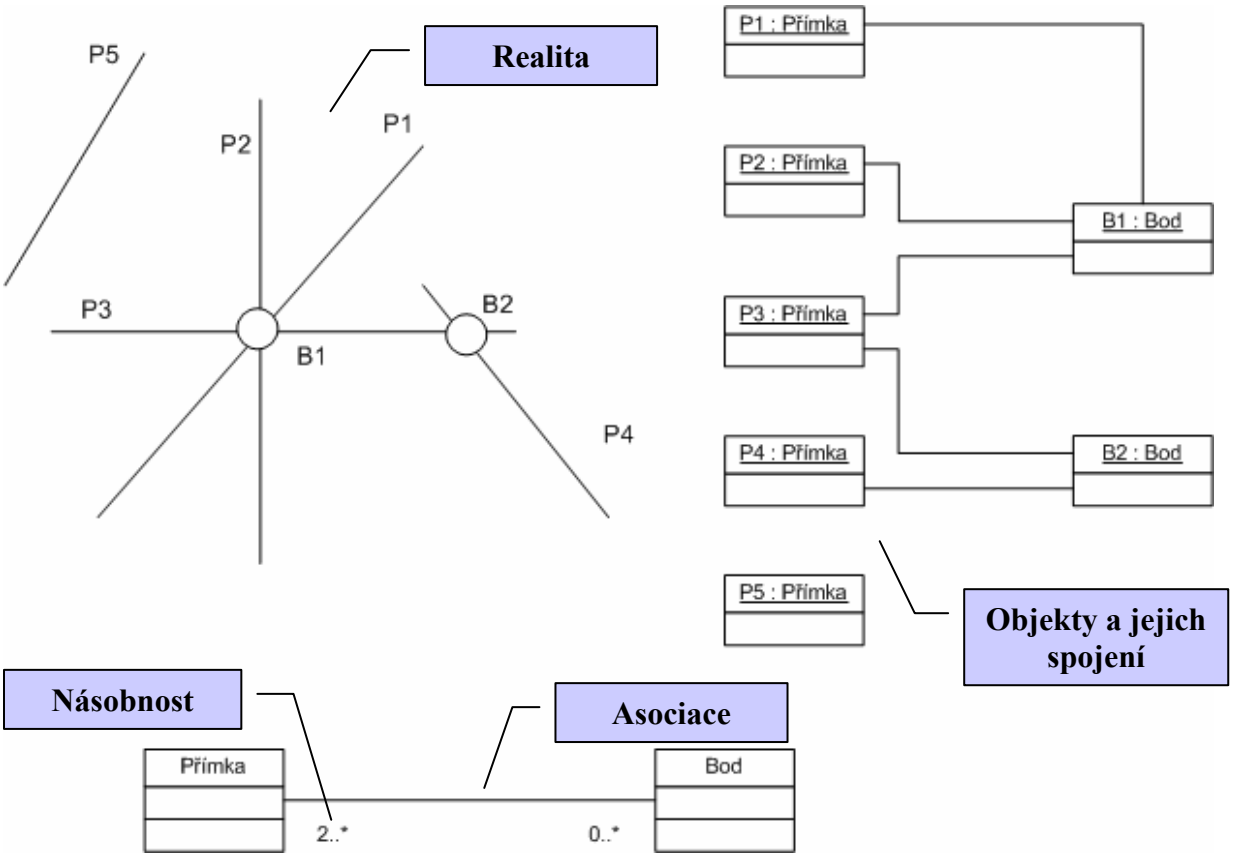
- <<local>> vyjadřuje situaci, kdy objekt je vytvořen v těle operace a po jejím vykonání je zrušen;
- <<global>> specifikuje globálně viditelný objekt;
- <<parameter>> vyjadřuje fakt, že objekt je předán druhému jako argument na něj zaslané zprávy;
- <<association>> specifikuje trvalou vazbu mezi objekty (někdy se také hovoří o tzv. známostním spojení).

Oba diagramy tak předurčují jaké musíme mít spojení mezi objekty, aby bylo možné zajistit vzájemné předávání zpráv mezi nimi. Zobecněním všech možných konfigurací těchto spojení mezi objekty pak určíme vazby mezi jejich třídami.

Příklad:

Pokusme se celou situaci osvětlit na příkladu přímek a bodů daných průsečíky těchto přímek (obr. 5.3.3). K dané reálné úloze vytvořme nejprve diagram spolupráce bez specifikace zpráv (tyto v daný okamžik pro nás nejsou důležité) postihující danou konfiguraci. Zřejmě posunutí úsečky P5 směrem k bodu B1 povedene k vytvoření nových dvou průsečíků s přímkami P2 a P3. To bude vyžadovat sestavení nového diagramu spolupráce. Takových možných konfigurací máme příliš mnoho, než abychome je mohli všechny popsat zvláštním diagramem spolupráce. Proto musíme přistoupit k zobecnění a vytvoření takového diagramu, který

postihne všechny varianty. Tím je diagram tříd, který mezi třídy Přímka a Bod zavádí vazbu (asociaci), která postihuje všechna možná propojení, neboť platí, že každá instance třídy Přímka může být spojena s 0 až n instancemi třídy Bod podle toho, zda li se protíná s jinou přímkou či nikoliv. Stejně tak platí, že libovolná instance třídy Bod je asociována minimálně s dvěmi instancemi třídy Přímka. Z těchto formulací tedy skutečně vyplývá, že tento diagram pomocí dvou tříd a jediné vazby mezi nimi dokáže postinout nekonečné množství konfigurací naší geometrické úlohy.

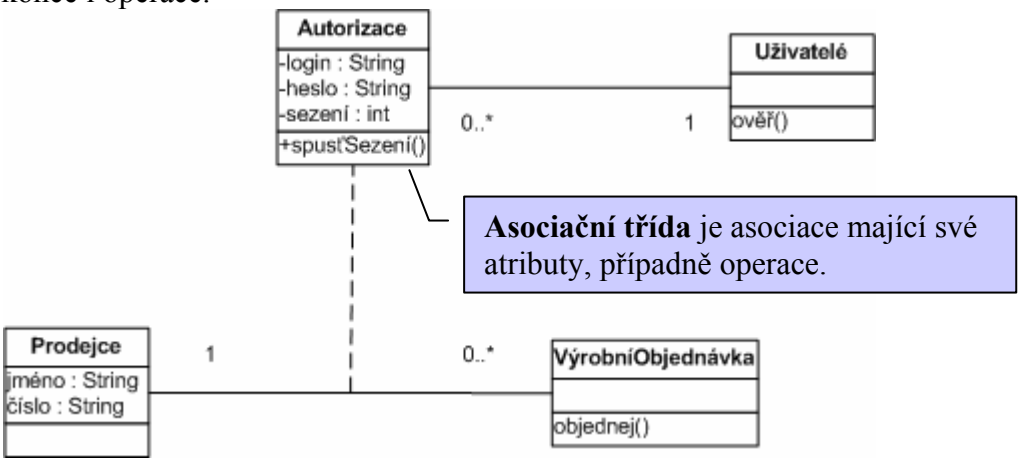


Obr. 5.3.3: Spojení a asociace

Diagram tříd zobecňující všechny možné konfigurace systému popsané diagramy spolupráce dále řadí vazby mezi třídami do následujících čtyř skupin:

- *Asociace* popisující 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* popisující vztah mezi celkem a jeho částmi, kde některé objekty definují komponenty jejichž složením vzniká celek reprezentovaný jiným objektem.
- *Závislost* reprezentující 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.

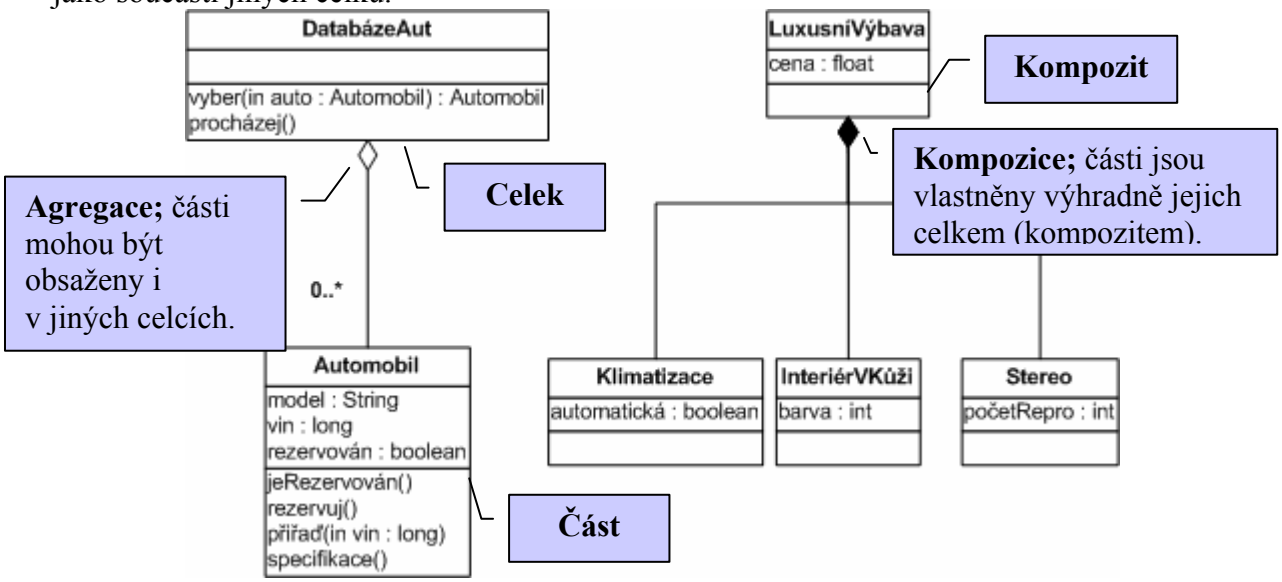
Asociace může být jednoduchá jako jsme viděli v případě geoemetrické úlohy, nebo může být reprezentována opět třídou. V tomto druhém složitějším případě můžeme asociaci přiřadit atributy a dokonce i operace.



Obr. 5.3.4: Asociační třídy

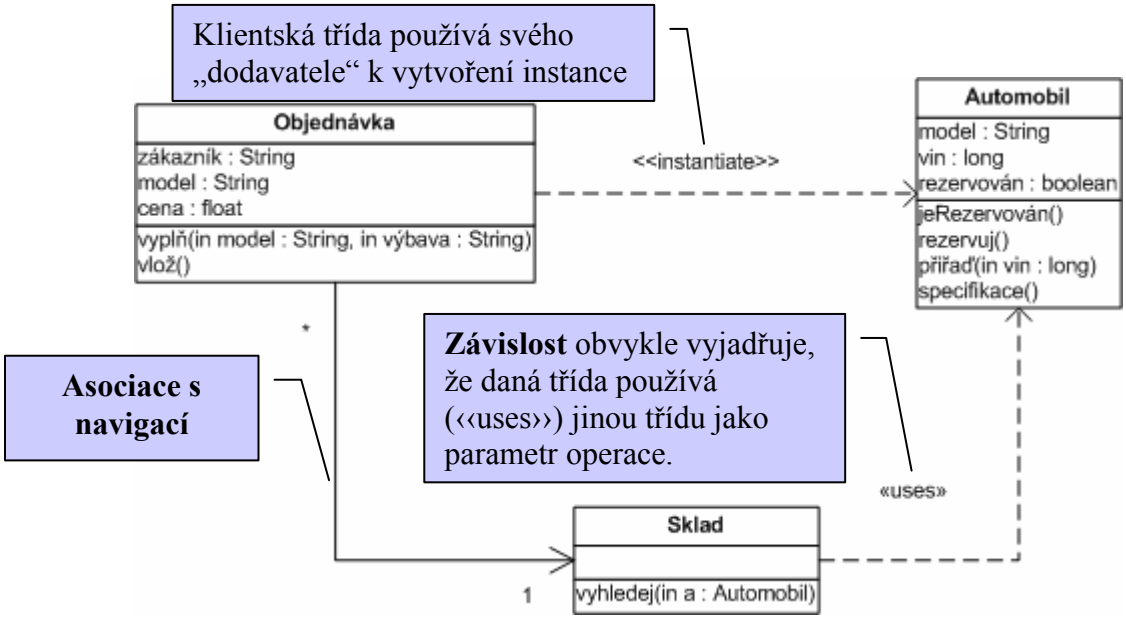
Příklad (pokr.):
Každý prodejce musí dříve nebo později vystavit objednávku (instance třídy VýrobníObjednávka) do výroby. Obecně jich může být až n. Předpokládáme-li, že objednávka je řešena pomocí on-line přístupu do informačního systému výrobce automobilu, pak každé vystavení objednávky (vytvoření asociace mezi prodejcem a objednávkou) probíhá prostřednictvím autorizace. Každá z těchto autorizací má atributy jako *login* uživatele, jeho *heslo* a také pořadové číslo *sezení*. Operace *spusťSezení* uskutečňuje samotné vložení objednávky do informačního systému výrobce. Kontrola správnosti je zajištěna instancí třídy Uživatelé, která ověří, zda-li prodejce s daným login a heslem má oprávnění k této operaci.

Relace složení částí do jednoho celku, má v podstatě dvě možné podoby (obr. 5.3.5). 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.



Obr. 5.3.5: Složení

Závislost je typ relace (vazby), která vzniká pouze dočasně pro potřeby poskytnutí služby klientskému objektu s tím, že poté tato vazba zaniká.



Obr. 5.3.6: Závislost

Příklad (pokr.):

Instace třídy objednávky v okamžiku specifikace požadovaného automobilu vytváří instanci třídy Automobil pomocí dočasné relace závislosti (<<instantiate>>). Obdobně instance třídy Sklad v určitý okamžik používá intanci třídy Automobil k jeho výběru z databáze.

Než si ukážeme způsob použití poslední z uvedených relací nazvané jako generalizace, zavedeme si tři další pojmy *role*, *typ* objektu a *rozhraní*, které úzce souvisí s zatím probranými relacemi asociace a závislosti.

Role definuje specifické chování objektu v daném kontextu jeho použití.

Stejně jako v reálném světě člověk vystupuje v různých rolích podle toho v jakém kontextu se právě nachází (v zaměstnání se nachází v roli pracovníka, doma je manželkou či manželem, v automobilu řidičem či pasažérem apod.) stejně vytupuje i objekt vzhledem k ostatním objektům se kterými je v relaci.

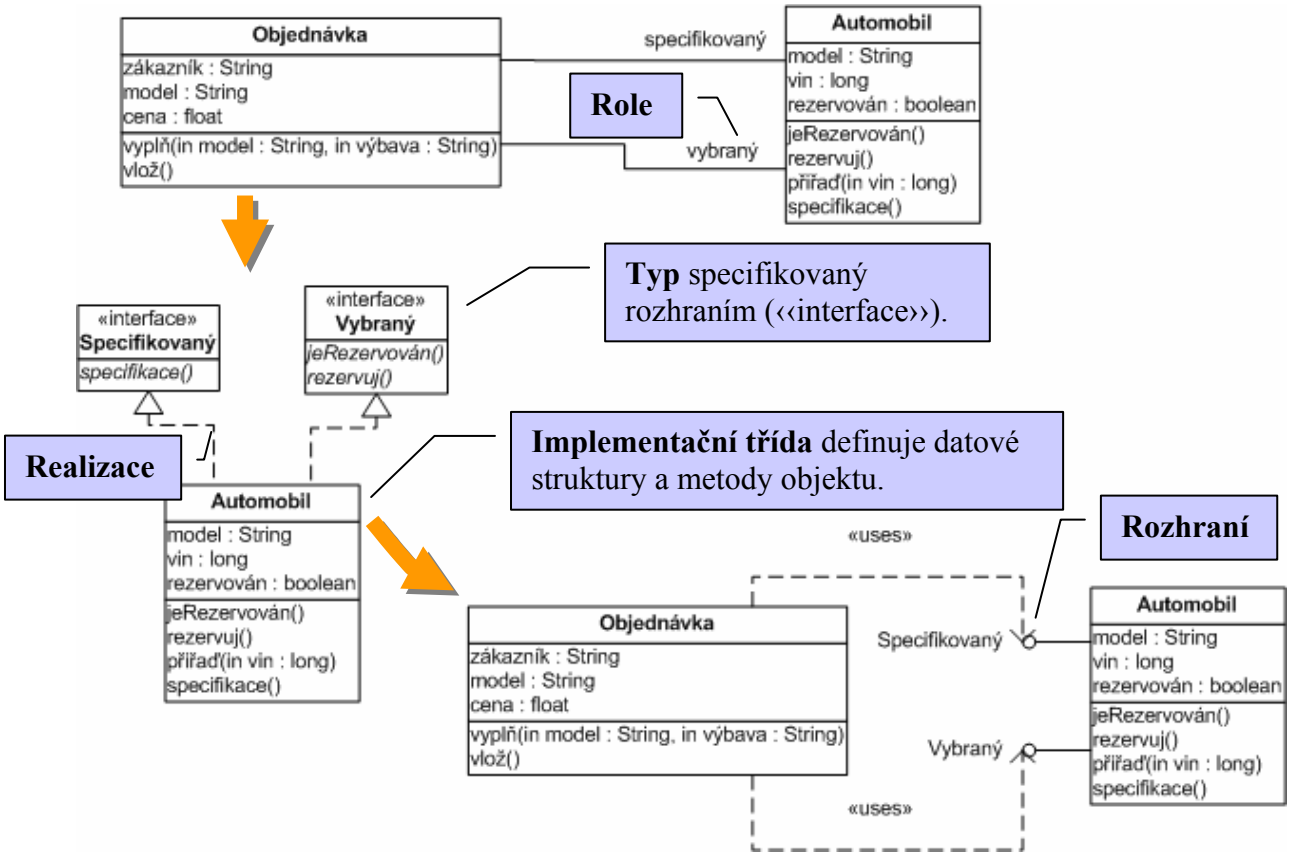
Typ objektu specifikuje skupinu operací, které mohou být objektem prováděny.

Každá role, ve které se objekt nachází pak odpovídá danému typu a naopak. Jinými slovy řečeno, podle aktuální role, ve které se objekt nachází, může tento provádět pouze operace dané odpovídajícím typem objektu.

Rozhraní je pojmenování skupiny externě viditelných operací.

Z uvedených definic vyplývá, že typ objektu a jeho rozhraní jsou de facto totéž a lze je tedy libovolně zaměňovat. Obvykle typ je pojem se kterým manipuluje analytik, zatímco pojem rozhraní je spíše vlastní programátorské obci. Důležité je si uvědomit, že rohraní pouze

definuje skupinu operací, ale nedefinuje jejich implementaci. To je úkolem až třídy, která dané rozhraní implementuje.
Z hlediska grafického jazyka UML je typ, nebo-li rozhraní, odlišeno od vlastních tříd pomocí klíčového slova <<interface>>.

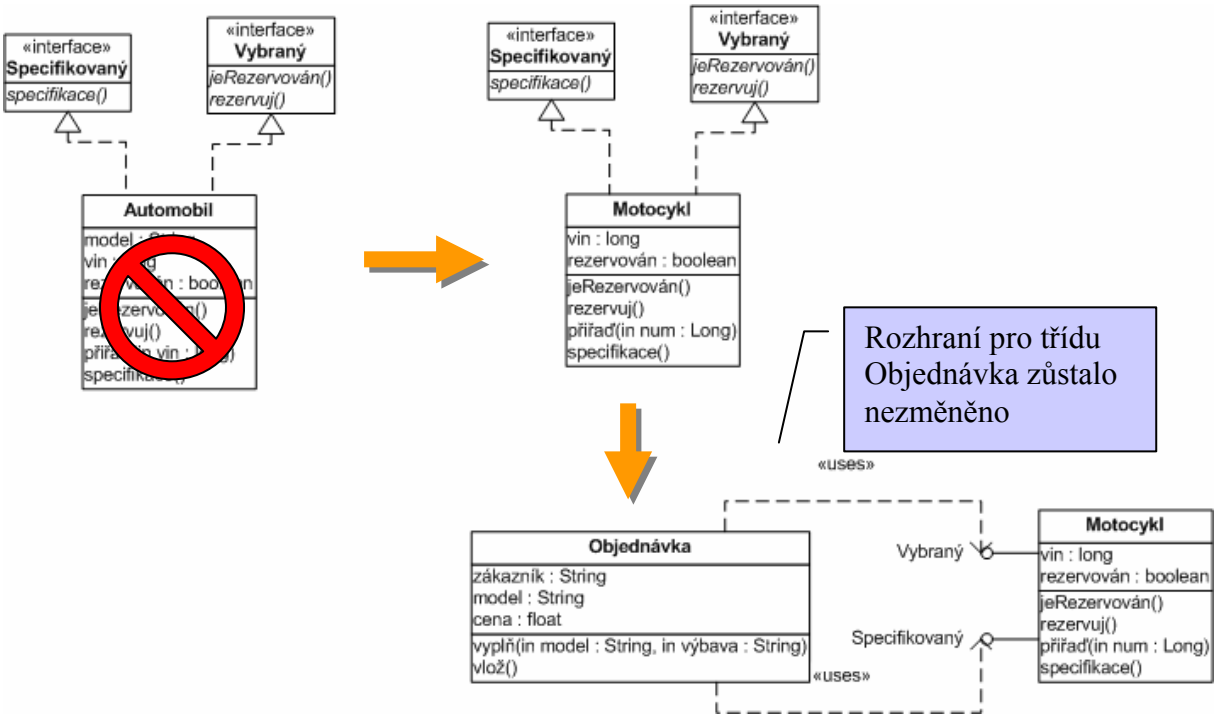


Obr. 5.3.7: Typy a implementační třídy

Příklad (pokr.):

Mějme instance třídy Objednávka a Automobil. Automobil vystupuje vůči objednávce ve dvou rolích. Jednou jako *specifikovaný* v případě, že objednávka definuje zákazníkem požadovaný automobil. V druhém případě pak automobil vystupuje v roli *vybraný*, pokud byl nalezen v databázi skladu automobilů. Dalším krokem je definování, které skupiny operací lze požadovat na instanci třídy Automobil podle toho, v jaké roli se nachází. Třída automobil tak musí realizovat dvě rozhraní Vybraný a Specifikovaný, které definují i odpovídající skupiny operací. Například zprávu *rezervuj* nemůžeme zaslat na objekt Automobil, pokud se nachází v roli *specifikovaný*. Nejdříve musí být nalezen v databázi a vybrán, aby mohl být následně rezervován. Grafický jazyk pak zavádí „konektory“ rozhraní, které explicitně definují, které z operací může asociovaná třída (v našem příkladu Objednávka) využívat.

Otázka, která nám vyvstane v daný okamžik na mysli, zní: „K čemu nám toto rozdělení specifikace operací od jejich implementace může sloužit?“. Odpověď je jednoduchá. Zajistíme flexibilitu a znovupoužitelnost našeho kódu tím, že implementační třídy můžeme kdykoliv zaměnit, aniž bychom museli měnit kontext (ostatní třídy), ve kterém tyto třídy existují.

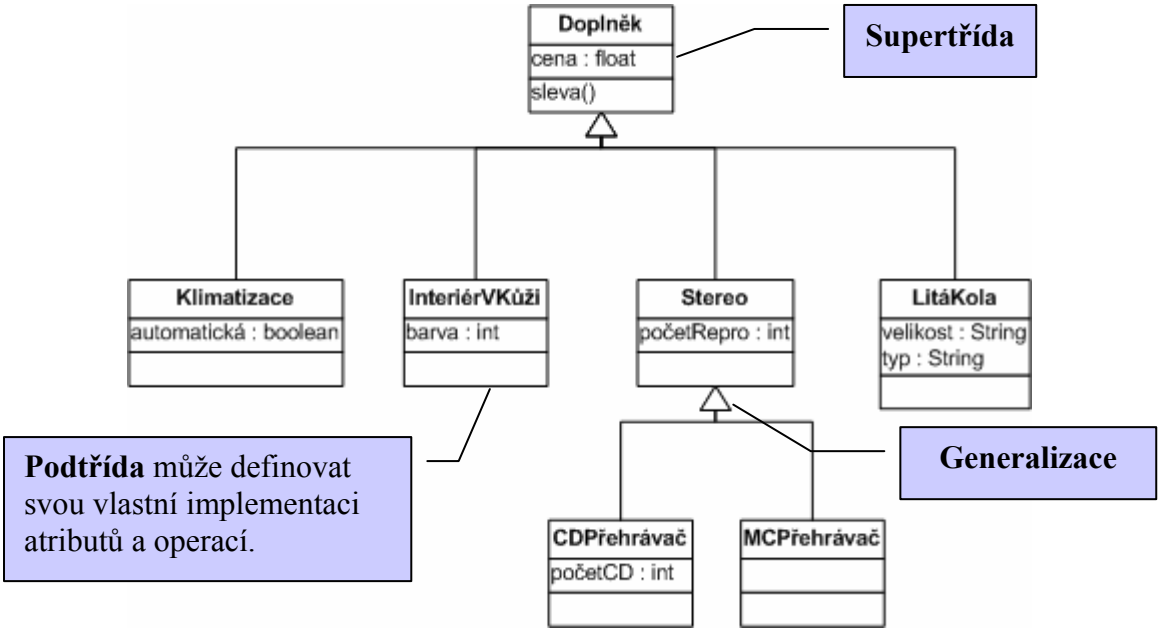


Obr. 5.3.8: Implementační nezávislost

Příklad (pokr.):

Pokud budeme chtít náš softwarový systém určený pro prodej automobilůpoužít také pro prodej motocyklů, pak v příapdě přísného dodržení oddělení specifikace operací od jejich implementace znamená, že stačí zaměnit třídu Automobil za třídu Motocykl bez toho, že bychom museli měnit třídu Objednávka. Ta totiž vyžaduje pouze definovaný protokol, množinu operací, které po objektu v roli specifikovaného či vybraného bude požadovat.

Relace generalizace (či specializace podle směru jejího vnímání) definuje konkretizace tříd do tzv. svých podtříd, přičemž tyto dědí (znovupoužívají) to, co pro ně jejich supertřída definuje.



Obr. 5.3.9: Generalizace a specializace

Příklad (pokr.):

Doplněk v našem příkladu prodeje automobilu je třída velmi obecná, která zavádí pouze základní vlastnosti (*cena*) a jedinou operaci aplikaci slevy. Tento obecný doplněk je dále konkretizován do třídy Klimatizace, LitáKola, InteriérVKůži a Stereo. Každá z těchto tříd dědí všechny vlastnosti svého předchůdce třídy Doplněk, kterými jsou již zmíněná *cena* a operace *sleva* a navíc přidávají své nové vlastnosti. U třídy Klimatizace je to atribut *automatická*, který je nastaven na hodnotu *true* nebo *false* podle toho, zda-li se jedná o klimatizaci automoatickou či manuální. Třída Stereo se sama navíc stává supertřídou pro své další možné konkretizace reprezentované třídami CDPřehrávač a MCPřehrávač.

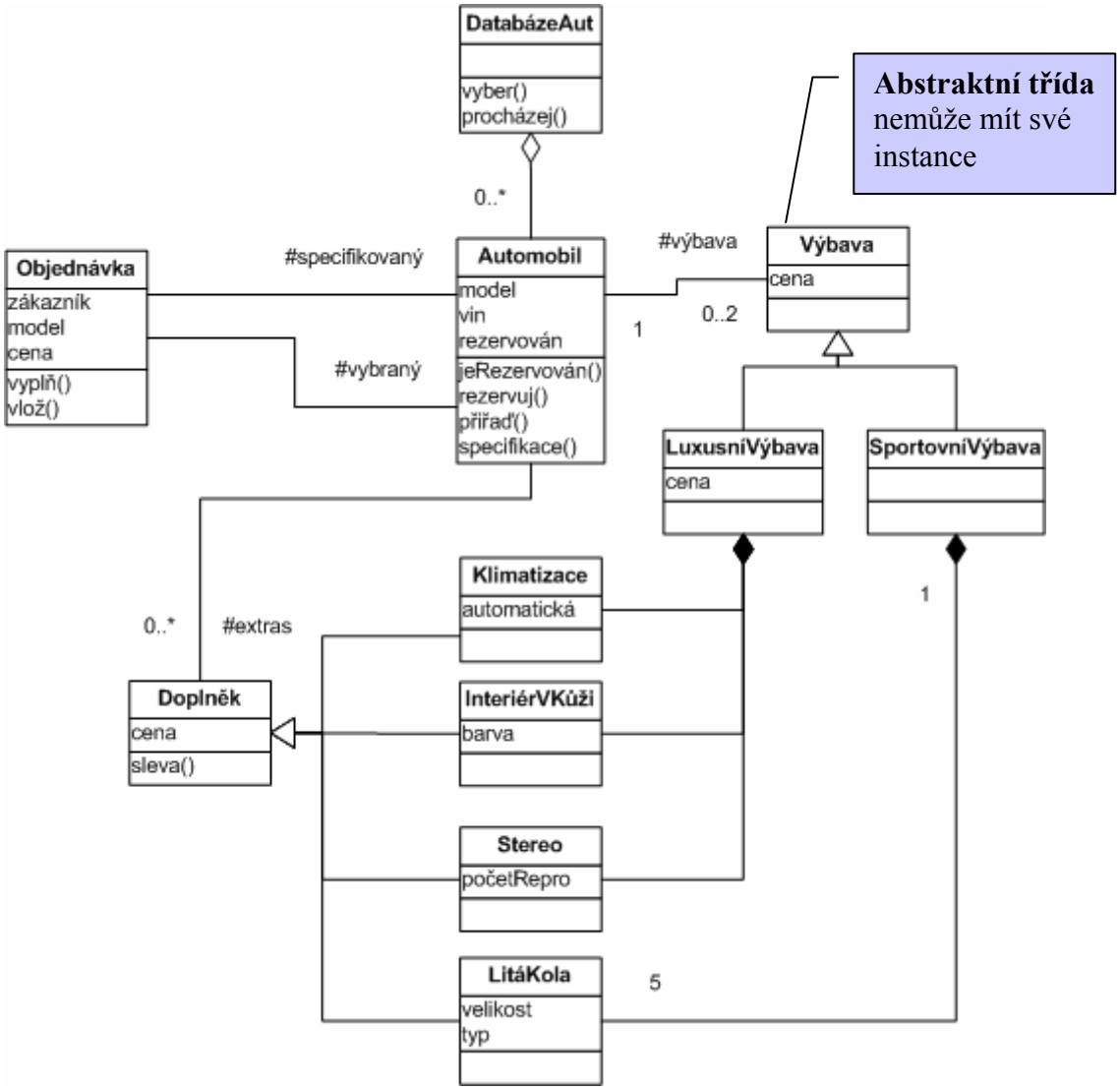
Se zavedením principu generalizace vedoucí k vytváření hierarchií tříd můžeme konečně zavést modifikátory viditelnosti (ve smyslu dostupnosti) jejich atributů, operací a vztahů k jiným třídám. Tyto jsou v zásadě tři (obr. 5.3.10):

- *Privátní* modifikátor umožňuje přistupovat k hodnotám takto označeného atributu, operacím či spojeným objektů pouze instanci dané třídy.
- *Chráněný* modifikátor rozšiřuje tento přístup i instancím podtříd, které jsou specializací dané třídy.
- *Veřejný* modifikátor odkrývá hodnoty atributů, operace a propojené objekty všem objektům bez omezení.

Obr. 5.3.10: Viditelnost

5.4. Diagramy tříd jazyka UML

V předcházející kapitole jsme se seznámili se všemi základními pojmy a jim odpovídajícími notacemi grafického jazyka, který umožňuje třídy, jejich rozhraní a vztahy mezi nimi strukturovat do podoby diagramu tříd. Tento diagram, jak již bylo zmíněno dříve, definuje základ statického popisu softwarového systému, neboť dokáže postihnout všechny dostupné konfigurace objektů. Ještě jednou si zopakujme, že zdrojem informací pro to, jak má diagram tříd vypadat, jsou diagramy sekvenční a spolupráce. Ty nám pomohou odhalit jaké objekty a tím i třídy potřebuje k realizaci požadované funkcionality, v jakých jsou tyto třídy relacích, které zprávy jsou mezi instancemi těchto tříd předávány. Bohužel tyto diagramy vždy postihují určitou konfiguraci systému a nepopisují strukturu tříd obecně nezávisle na časovém okamžiku, ve kterém se softwarový systém právě nachází (viz. geometrická úloha z předchozí kapitoly). K tomuto účelu sestavujeme diagram tříd, který je univerzální a na čase nezávislý. Právě z tohoto důvodu je diagram tříd stavebním kamenem modelu analýzy i návrhu a nemůžeme se bez něj obejít, poněvadž jeho sestavení je nutnou podmínkou následné implementace softwarové systému pomocí programovacího jazyka. Diagramů tříd postihujících různé aspekty výsledného systému sestavujeme celou řadu, i když teoreticky by pro úplné postižení měl stačit jediný. Důvodem je však omezená srozumitelnost takového diagramu v případě, že softwarový systém je tvořen stovkami tříd.



Obr. 5.4.1: Diagram tříd

Příklad (pokr.):

V našem příkladu prodeje automobilu je jádrem navrhovaného softwarového systému diagram tříd z obr. 5.4.1. Databáze aut se skládá obecně z n dostupných aut, která jsou nejprve specifikována a následně vybrána dle objednávky. Každý automobil má přiřazeny standardizované výbavy (luxusní a/nebo sportovní) nebo si zákazník vybírá doplňky podle svého. Výbavy se skládají z předdefinované skupiny doplňků, kde např. SportovníVýbava je tvořena pouze koly z lehké slitiny. Sestavený diagram tříd představuje jakousi první verzi. Jak bylo uvedenou v kapitole zabývající se softwarovým procesem, v čase tento diagram prochází celou řadou úprav a inkrementálně je rozšiřován a modifikován až do takové podoby, která splňuje podmínku funkcionality definovanou zadavatelem projektu.

5.5 Specifikace dynamického chování

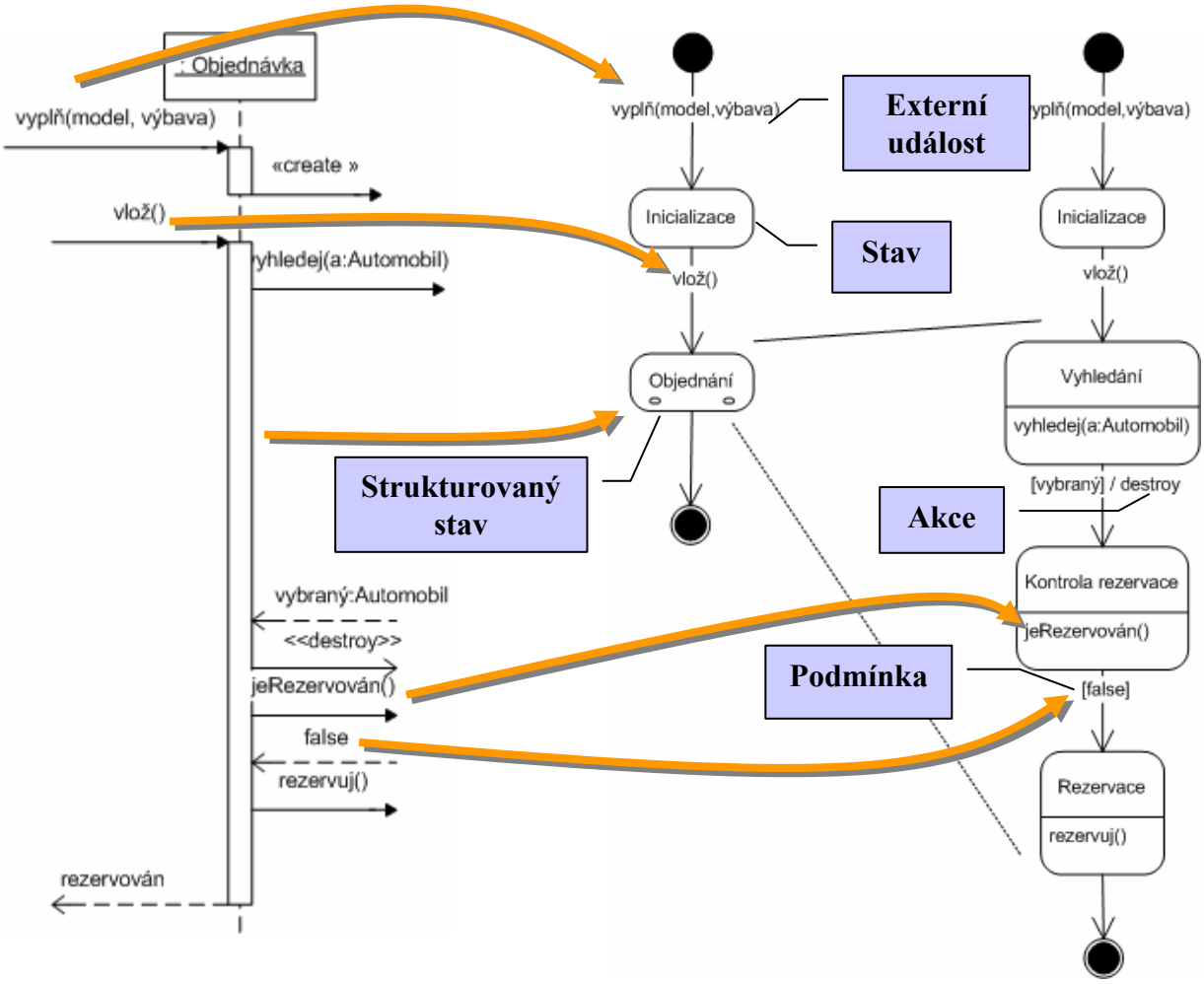
Jestliže jsme si v předcházející kapitole ukázali jak pomocí třídního diagramu postihnout statický aspekt softwarového systému, tak si nyní ukážeme jak zachytit i jeho dynamické chování. Diagramy sekvenční a spolupráce, jak již bylo uvedeno, modelují pouze určité časové snímky a lze z nich tedy jen obtížne popsat celý životní cyklus objektů z pohledu jejich dynamického chování. To je účelem použití tzv. *stavového diagramu*, který stejně jako

třídní diagram za svůj informační zdroj opět využívá především sekvenční diagram případně diagram spolupráce.

Stavový diagram zobrazuje životní cyklus objektu, události způsobující přechody z jednoho stavu do jiného a akce, které vyplývají z této změny stavu.

Další základní rozdíl vůči sekvenčnímu diagramu a diagramu spolupráce popisující interakce mezi několika objekty je, že tento diagram je sestavován pro každý objekt (přesněji řečeno pro jeho třídu) zvlášť, přičemž vybíráme pouze ty objekty, které mají skutečně významné dynamického chování.

Z definice stavového diagramu vyplývá, že musíme nalézt stavy objektu, události, které způsobují změny těchto stavů a nakonec akce, které z těchto změn vyplývají. Stavy nalezneme v sekvenčním diagramu v podobě operací, události jsou zprávy, které vedou ke spuštění operace, akce jsou pak zprávy, které jsou odesílany směrem k ostaním objektům při vykonávaní operace (obr. 5.5.1).



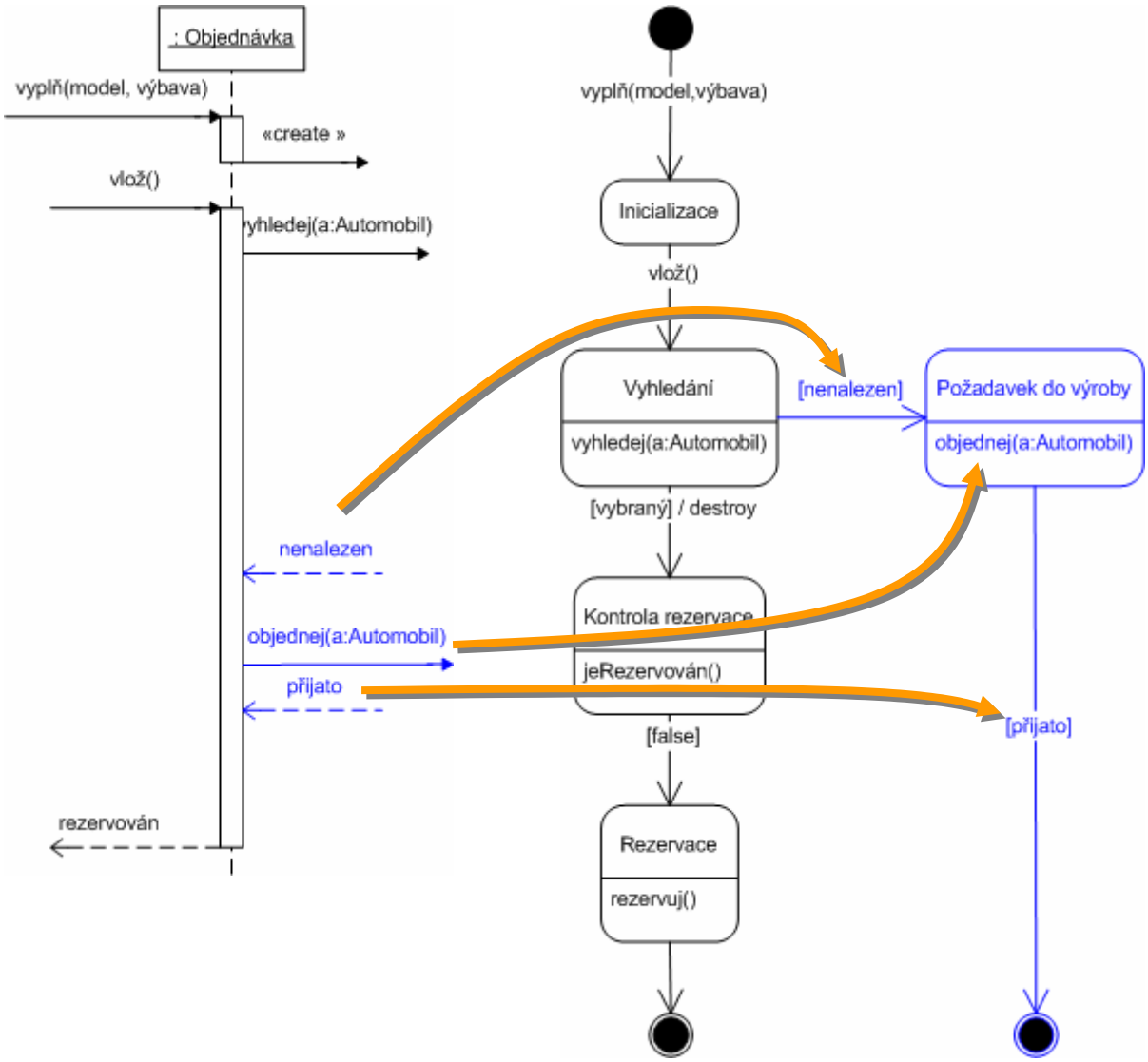
Obr. 5.5.1: Stavový diagram

Příklad (pokr.):

Sestavení stavového diagramu pro objekt Objednávka vychází z analýzy sekvenčních diagramů, ve kterých tento objekt vystupuje. Z vybrané části konkrétního sekvenčního diagramu vyplývají dva stavy (Inicializace a Objednání), do kterých se objekt dostává na

základě událostí *vyplň* a *vlož*. Druhý z uvedených stavů můžeme dále zpodrobnit a vyjádřit tak lépe jaké akce jsou objektem provedeny a jaké jsou také podmínky přechodů do nových stavů. Např. ve stavu Kontrola rezervace je provedena akce (zaslání zprávy) j*eRezervován*, která v případě negativní odpovědi *false* umožní přechod do nového stavu Rezervace. Tento je spojen s provedením akce *rezervuj*.

Protože objekt vystupuje obyvkle více než v jednom sekvenčním diagramu, je nutné stavový diagram doplnit o stavy a události, které tyto další sekvenční diagramy specifikují. Díky tomu se tak, analogicky s diagramem tříd, stává stavový diagram jakýmsi zobecněním (obr. 5.5.2) všech možných scénářů popsaných celou řadou sekvenčních diagramů.



Obr. 5.5.2: Spojení scénářů

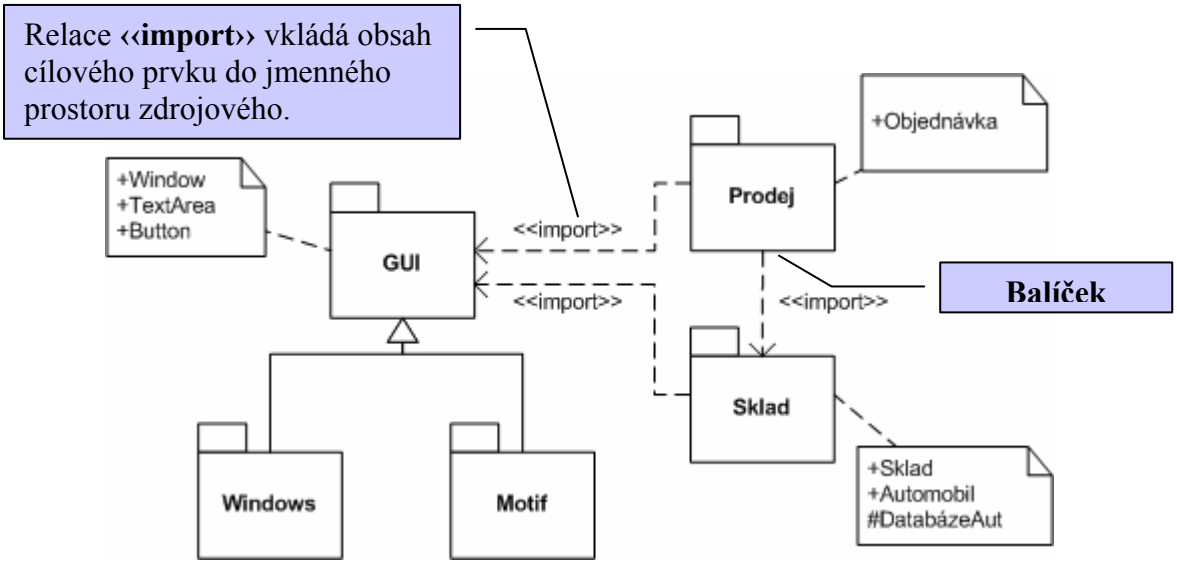
5.6 Strukturování softwarového systému

Už v kapitole popisující diagramy tříd jsme uvedli, že složitý systém je nutno posat velkým počtem diagramů. Zřejmě je tedy nutné respektovat při modelování softwarového systému principy, umožňující vhodným způsobem vytvářené modely, obvykle reprezentované celou řadou diagramů, strukturovat a lépe tak spravovat celou dokumentaci vytvářeného produktu.

Pro tyto účely zavádí UML pojem *balíček* který je definován následujícím způsobem:

Balíček je mechanismum obecného použití umožňující organizovat elementy do skupin s cílem redukovat složitost modelovaného systému.

Tyto balíčky mohou sdružovat libovolné části modelovaného systému. Pokud se bude jednat o části, které spolu souvisí z hlediska zajištění nějaké společné funkčnosti či účelu pak hovoříme o vytváření tzv. *subsystémů*. Samotné *modely*, které vytváříme jsou de facto také speciálním případem balíčku, který vyjadřuje zjednodušení reality, abstrakci systému vytvářeného za účelem jeho snadnějšího pochopení.



Obr. 5.6.1: Balíčky

Příklad (pokr.):
V případě systému prodeje auta můžeme z hlediska poskytované funkcionality vytvořit několi samostatných balíčků (subsystémů) zabývajících se vlastním prodejem a správou skladu. Protože výsledná aplikace musí poskytnout i služby grafického uživatelského rozhraní můžeme doplnit další subsystém GUI, který může být konkretizován do podoby rozhraní operačního systému Windows nebo UNIX.

Zavedením subsystému grafického uživatelského rozhraní se však již dostáváme do další etapy modelování softwarového systému, kterým je tzv. *návrh* zahrnující implementační prostředí využívané k vytvoření výsledné aplikace.

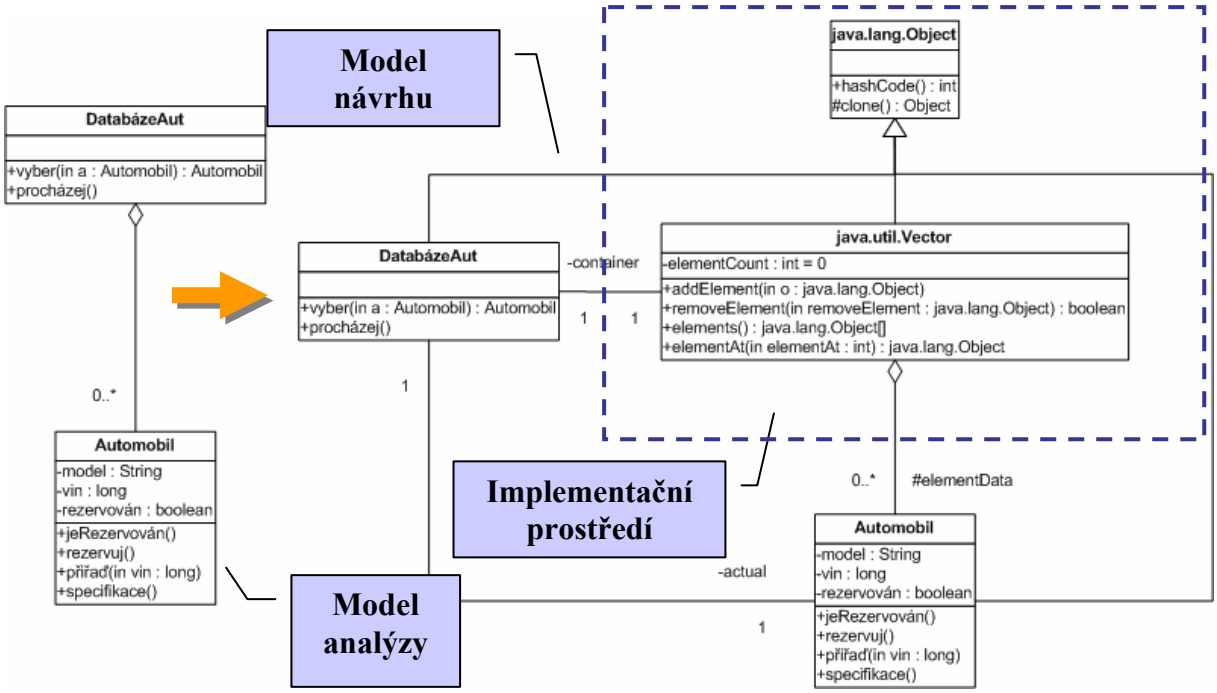
5.7 Návrh a jeho cíle

Cílem etapy návrhu v rámci toku činností zabývajícího se analýzou a návrhem je vytvoření modelu návrhu. Co je model návrhu můžeme definovat následujícím způsobem:

Model návrhu dále upřesňuje model analýzy ve světle skutečného implementačního prostředí.

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:

- definice systémové architektury,
- identifikace návrhových vzorů a možnosti znovupoužití tzv. rámcových řešení,
- definice softwarových komponent a jejich znovupoužití.



Obr. 5.7.1: Mapování na softwarové komponenty

Příklad (pokr.):

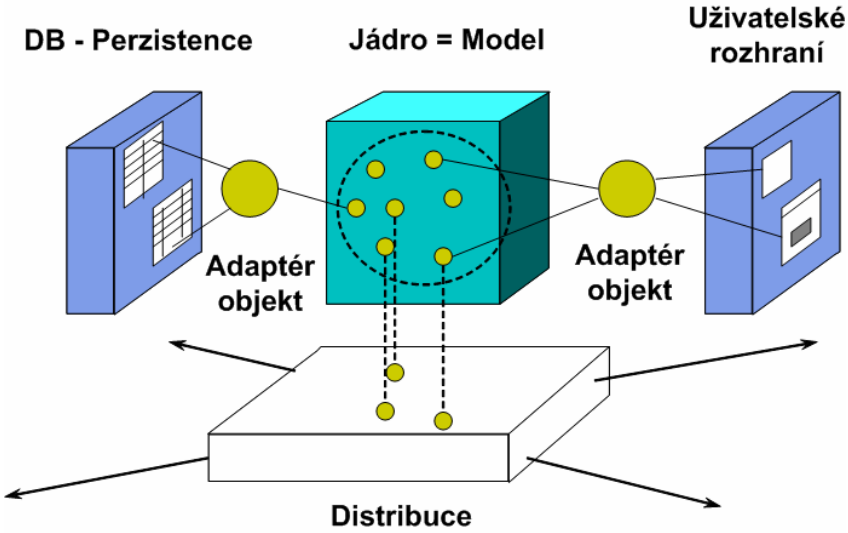
V modelu analýzy jeden z navržených diagramů tříd specifikuje třídu DatabázeAut jako celek skládající se z 0 až n instancí třídy Automobil. Mapování na již existující softwarové komponenty znamená, že k realizaci tohoto složení využijeme třídu java.util.Vector obsaženou v API (Application Programming Interface), nebo-li také v knihovně, platformy jazyka Java. Třída DatabázeAut bude asociována právě s jednou instancí třídy java.util.Vector, která bude obsahovat elementy reprezentované jednotlivými instancemi třídy Automobil. Tímto způsobem ušetříme práci programátorů, kteří by jinak museli navrhnout a implementovat dynamické pole umožňující vkládat, vyhledávat a odebírat požadované automobily. Protože v Javě všechny třídy mají jedinou společnou supertřídu java.lang.Object, je tento fakt v diagramu tříd návrhu vyjádřen pomocí relace generalizace.

5.8 Architektura výsledného systému

Implementační prostředí je tvořeno softwarovými komponentami spadajícími do následujících tří vrstev doplňujících objekty analýzy vztažené výhradně k doménové oblasti:

- *Uživatelské rozhraní* umožňující komunikaci vyvíjeného systému s uživateli nebo jinými systémy.
- *Distribuce* sloužící k rozložení rozsáhlých aplikací na více navzájem propojených výpočetních uzlů.
- *Persistence* umožňující datový obsah reprezentovaný hodnotami atributů udržovaných objekty trvale ukládat tak, aby bylo umožněno jeho přetrvání i mimo dobu běhu programu.

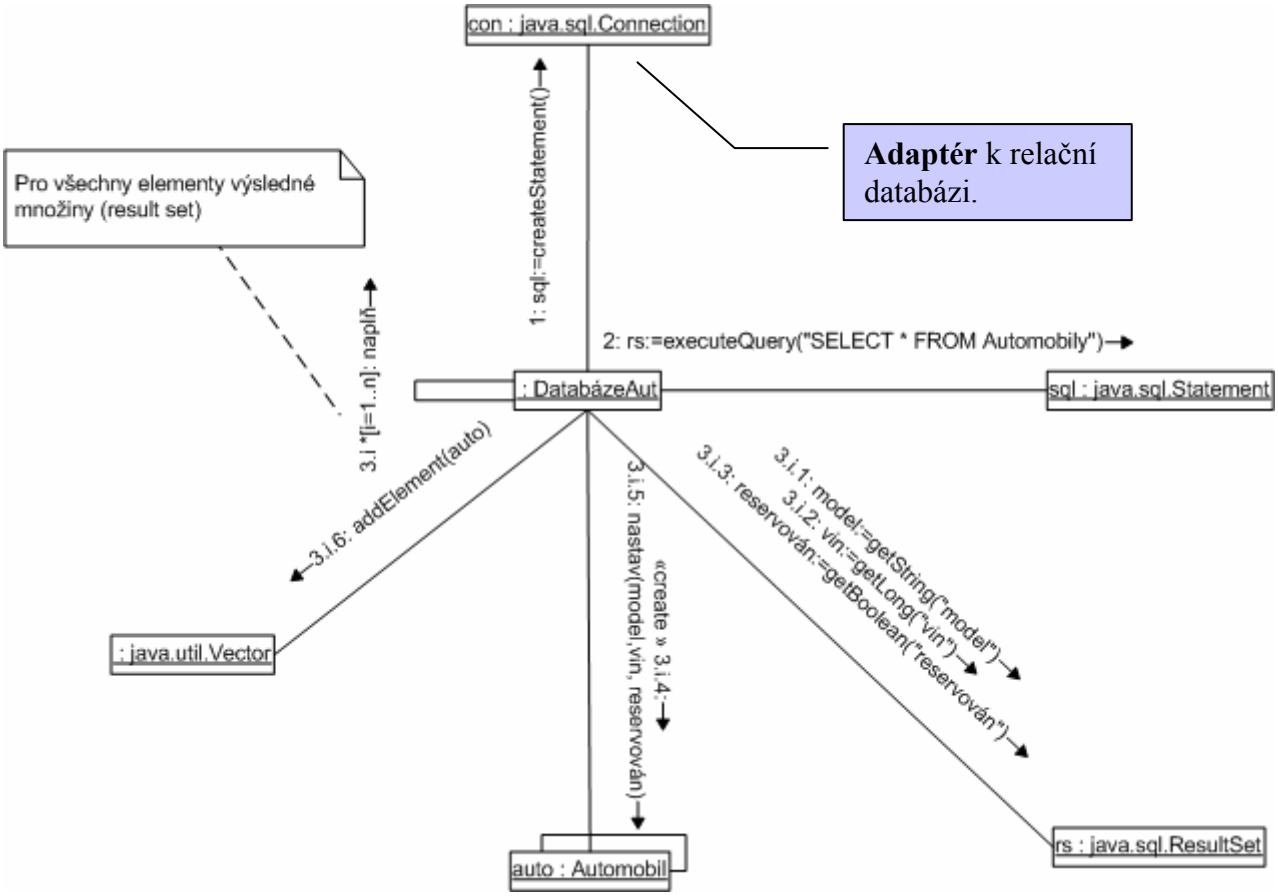
Celou situaci dokresluje obr. 5.8.1 znázorňující jak je jádro systému vytvořeného v etapě analýzy doplněno o výše uvedené tři vrstvy.



Obr. 5.8.1: Architektura softwarového systému

Za krátkou zmínku stojí tzv. adaptér objekty, které mají jediný úkol, kterým je zprostředkovat styk objektů analýzy se službami operačního systému a relačních databází. Obvykle tyto služby nejsou implementovány pomocí objektově orientovaných principů a je nutné je tzv. zapouzdřit do k tomu speciálně určených objektů nazvané adaptéry.

Celou tuto situaci dokumentuje příklad (obr. 5.8.2) diagramu spolupráce, kde adaptér objekt (instance třídy java.sql.Connection) umožňuje aplikaci připojit se na relační databázi, provést dotazovací příkaz jazyka SQL a na základě výsledku tohoto dotazu naplnit v našem případě instanci výše zmíněné třídy java.util.Vector elementy perzistentně uloženými v této databázi.



Obr. 5.8.2: Příklad použití adaptéru relační databáze

5.9 Návrhové vzory a aplikační rámce

Návrhové vzory můžeme chápat jako abstrakci imitování užitečných části jiných softwarových produktů.

Volně intepretováno, pokud zjistíme že používáme k řešení určitého problému úspěšné řešení, které se opakuje v různých produktech z různých doménových oblastí, pak zobecnění tohoto řešení se stává návrhovým vzorem.

Každý takový návrhový vzor je popsán množinou komunikujících objektů a jejich tříd, které jsou přizpůsobeny řešení obecného problému návrhu v daném konkrétním kontextu, tedy již existujícímu okolí.

Klasifikovat můžeme návrhové vzory podle způsobu jejich použití do těchto základních tří skupin:

- Návrhové vzory *tvořící* určené k řešení problému vytváření instancí tříd cestou delegace této funkce na speciálně k tomuto účelu navržené třídy.
- Návrhové vzory *strukturální* řešící problémy způsobu strukturování objektů a jejich tříd.
- Návrhové vzory *chování* popisující algoritmy a spolupráci objektů.

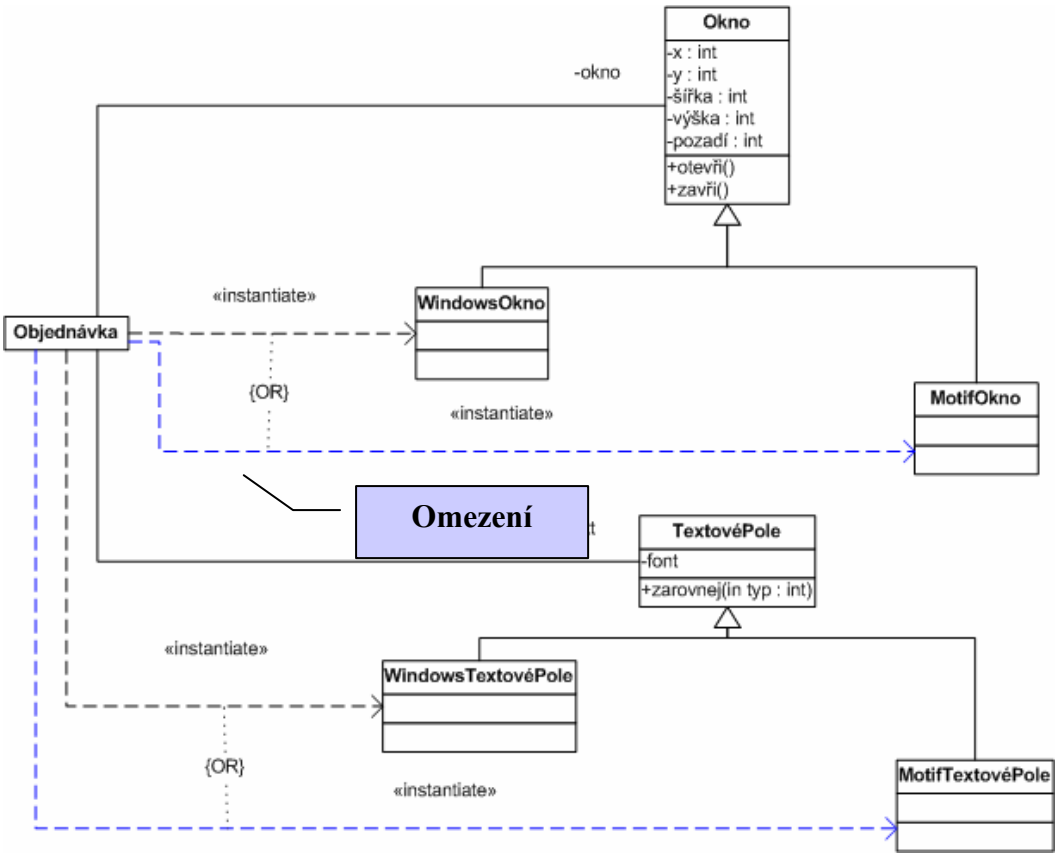
V dnešní době existuje velké množství návrhových vzorů, které svým počtem převyšující možnosti této publikace. V dalším tedy budou prezentovány pouze tři návrhové vzory jako

příklad, po jednom z každé z uvedených skupin. Samotné návrhové vzory budou uvedeny podle následujícího schématu. Nejprve bude definován *záměr*, účel ke kterému daný návrhový vzor slouží. Následovat bude *motivační* příklad, jinak také kontext, do kterého bude demonstrovaný návrhový vzor včleněn. Poté bude posáno *řešení* problému a nakonec bude toto řešení *zobecněno* do vlastního návrhového vzoru.

První z uvedených vzorů *Továrna* patří do skupiny návrhových vzorů tvořících a je definován dle uvedeného schématu následovně.

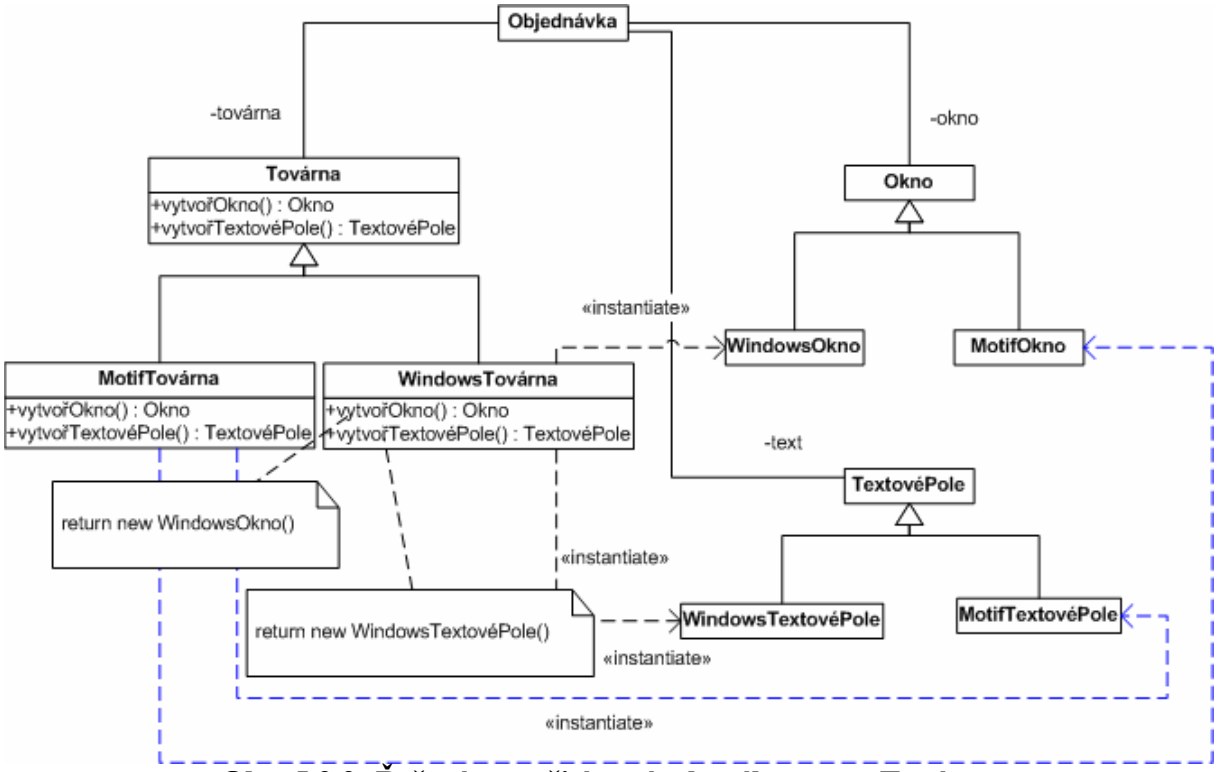
Záměr: Továrna poskytuje rozhraní pro vytváření rodin příbuzných objektů bez nutnosti specifikovat jejich konkrétní třídy.

Motivace: Uvažujme příklad z obr. 5.9.1. Mějme instanci třídy Objednávka, která z důvodu svého postupného vyplnění bude vázána na instanci tříd Okno a TextovéPole. Pokud bude naše aplikace bude provozována na operačním systému MS Windows, pak tyto instance (rodina příbuzných objektů) budou vytvářeny z konkretizovaných tříd WindowsOkno a WindowsTextovéPole. Analogicky pro operační systém UNIX budou tyto třídy MotifOkno a MotifTextovéPole. Problémem je, že při přechodu z jednoho operačního systému na druhý, budeme muset ve zdrojových kódech nalézt místa, kde jsou tyto instance vytvářeny a nahradit je danému operačnímu systému odpovídajícím třídám. To se sice na první pohled nemusí zdát nikterak složité, ale v případě, že rodina příbuzných objektů má desítky takových tříd, pak se tento problém stává nezanedbatelným. Cílem je tedy navrhnout takové řešení, aby objednávka nemusela vědět o konkrétních třídách, které budou vytvářet objekty okna a textového pole a využila k tomuto účelu prostředníka, který se postará o korektní vytvoření správných instancí.



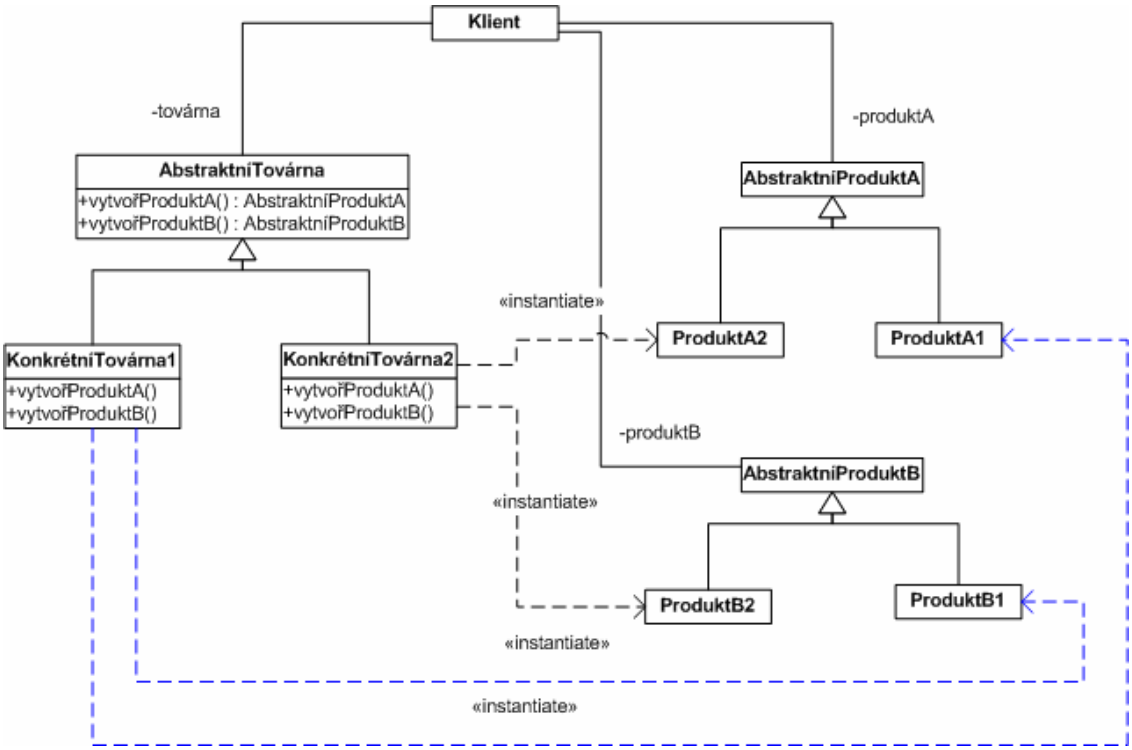
Obr. 5.9.1: Motivační příklad návrhového vzoru Továrna

Řešení: Problém nalezení prostředníka spočívá v navržení třídy Továrna s operacemi určenými k vytváření konkrétních instancí požadovaných tříd (obr. 5.9.2). Továrna je tzv. abstraktní třída, čili nemá svou instanci, ale slouží jako supertřída pro vytváření konkrétních továren na vytváření objektů. V našem případě se jedná o třídy WindowsTovárna a MotifTovárna. Vlastní objednávka je asociována s odpovídající továrnou, dle operačního systému, a pouze na tuto továrnu zasílá zprávy *vytvořOkno* nebo *vytvořTextovéPole*. Tyto operace dle dané továrny pak vytváří požadované instance tříd Okno a TextovéPole. Flexibilita, kterou toto řešení spočívá v tom, že pokud budeme chtít zavést okna a textová pole i pro jiné operační systémy, stačí pouze vytvořit podtřídu třídy Továrna, např. MacOSTovárna, a není nutné vůbec nic dalšího měnit na zdrojových kódech naší aplikace.



Obr. 5.9.2: Řešení s použitím návrhového vzoru Továrna

Zobecnění: Vzhledem k tomu, že takovýchto situací, kdy vytváříme instance tříd v aplikaci je celá řada, pokusíme se o zobecnění, tedy o abstraktní popis továrny tak, aby se vhodným zaměněním jmen tříd dala konkretizovat pro řešení obdobných problému. Tím je beze zbytku dán návrhový vzor Továrna (obr. 5.9.3).

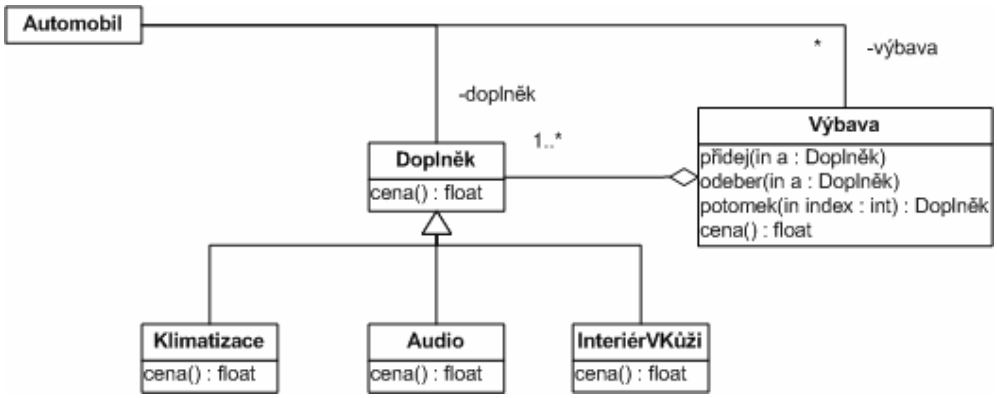


Obr. 5.9.3: Návrhový vzor Továrna

Dalším z uvedných návrhových vzorů je *Kompozit*, který patří do druhé skupiny tzv. strukturálních návrhových vzorů.

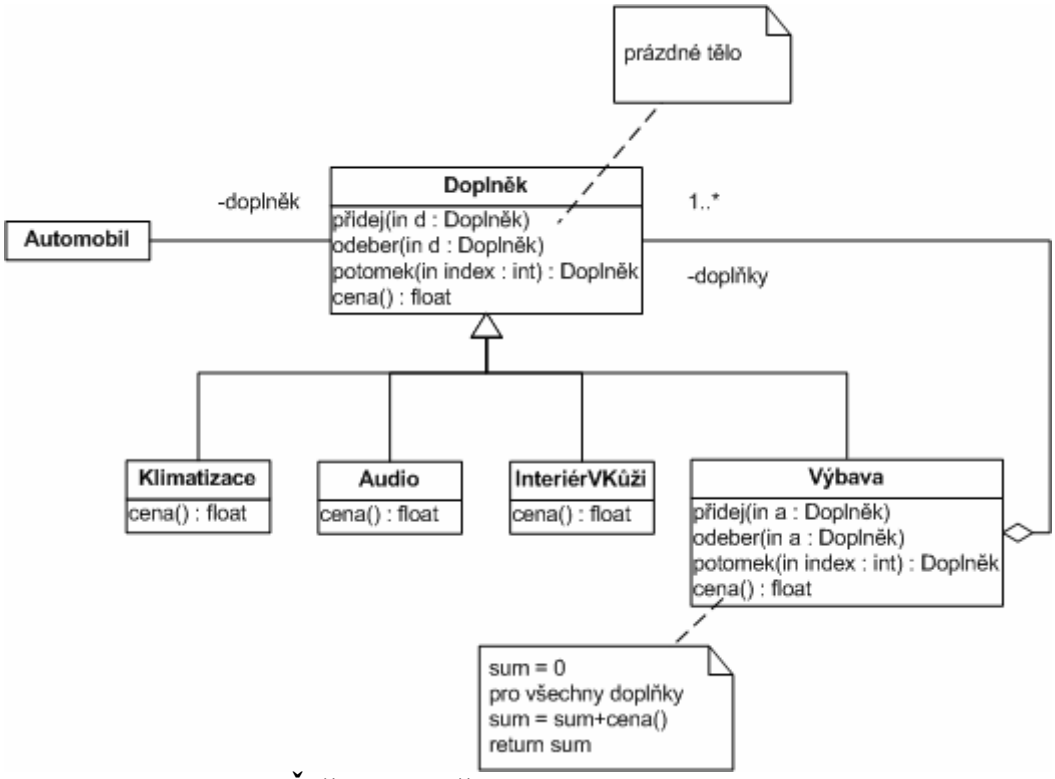
Záměr: Návrhový vzor Kompozit umožňuje složení objektů do stromové struktury reprezentující hierarchii celků a jejich částí. Navíc tento návrhový vzor umožňuje klientům pracovat s celky i jejich částmi jednotným způsobem. Klient tedy nemusí rozlišovat, zda-li se jedná o celek nebo část.

Motivace: Uvažujme příklad z obr. 5.9.4. Objekt třídy Automobil je na požadavek zákazníka rozšiřován o jednotlivé doplňky, nebo o celou výbavu zahrnující vybrané skupiny doplňků (vzpomeňme si na sportovní či luxusní výbavu z obr. 5.4.1). Problém je, že instace třídy automobil musí udržovat vazbu na n výbav a současně na n doplňků. Žádoucí je tedy zjednodušení pomocí návrhového vzoru, které smaže rozdíl mezi výbavou a doplňkem, tedy rozdíl mezi celkem a jeho částmi. Navíc nemáme možnost vytvářet nové výbavy tvořené z jiných výbav. Např. zákazník bude požadovat špičkovou výbavu, která bude zahrnovat jak luxusní výbavu se vším všudy tak i výbavu sportovní.



Obr. 5.9.4: Motivační příklad návrhového vzoru Kompozit

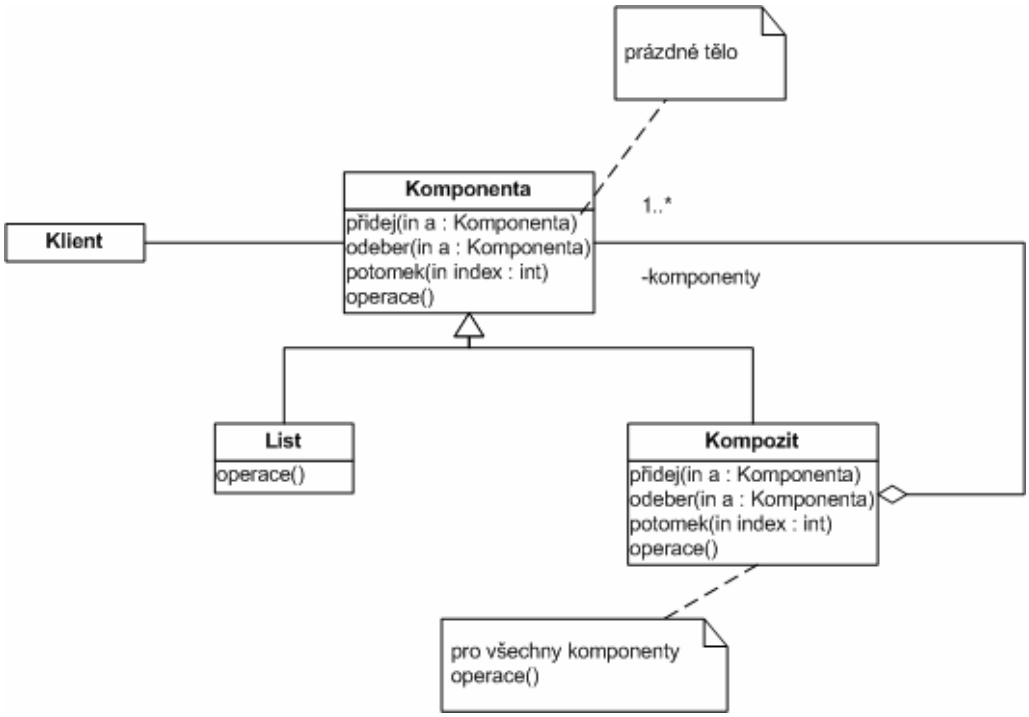
Řešení: Problém lze řešit pomocí návrhové vzoru Kompozit tím způsobem, že zavedeme abstraktní třídu Doplňek, která může být konkretizována buď do jednotlivých prvků výbavy jako je Klimatizace, LitáKola apod., nebo do třídy Výbava, které tvoří v podstatě celek pro vkládání doplňků a to jak jednotlivých prvků, tak opět nových výbav (obr. 5.9.5). Z pohledu stromové struktury instance třídy Výbava tvoří uzly stromu, zatímco prvky výbavy listy stromu. Díky této organizaci tak můžeme vytvářet celou hierarchií celků a jejich částí bez omezení hloubky této hierarchie. Navíc třída Automobil si udržuje jednou n-násonou asociaci na třídu Doplněk a nemusí tak explicitně odlišovat, jedná-li se pouze o prvek výbavy nebo celou výbavu.



Obr. 5.9.5: Řešení s použitím návrhového vzoru Kompozit

Zobecnění: Strukturování do stromu je natolik často používanou metodou, že je nutné provést zobecnění a definovat vlastní návrhový vzor Kompozit (obr. 5.9.6). Jádrem je třída Komponenta, která je konkretizována do nějaké dále nedělitelné třídy List nebo do třídy Kompozit sloužící jako nástroj pro skladání dalších komponent. Rekurzivně tak můžeme vytvořit hierarchii teoreticky neomezené hloubky. Za povšimnutí stojí i společná *operace* proveditelná jak listem tak kompozitem. Její provedení v případě instance třídy Kompozit spočívá v tom, že tato instance přepošle tuto zprávu na všechny své části, ze kterých je složena.

Tento návrhový vzor je používán zejména v návrhu grafického uživatelského rozhraní, kde kompozitem jsou plochy (např. tomůže být samotné okno či dialog), do kterých jsou vkládány další plochy nebo atomické prvky jako např. textová pole, tlačítka či posuvníky. Vytváří se tak hierarchie objektů uživatelské prostředí, které mají např. společnou operaci zobrazení vykreslující jejich obsah na obrazovku. Plochy přepošlou tento požadavek na všechny své komponenty, zatímco atomické prvky se vykreslí podle jim daného algoritmu.

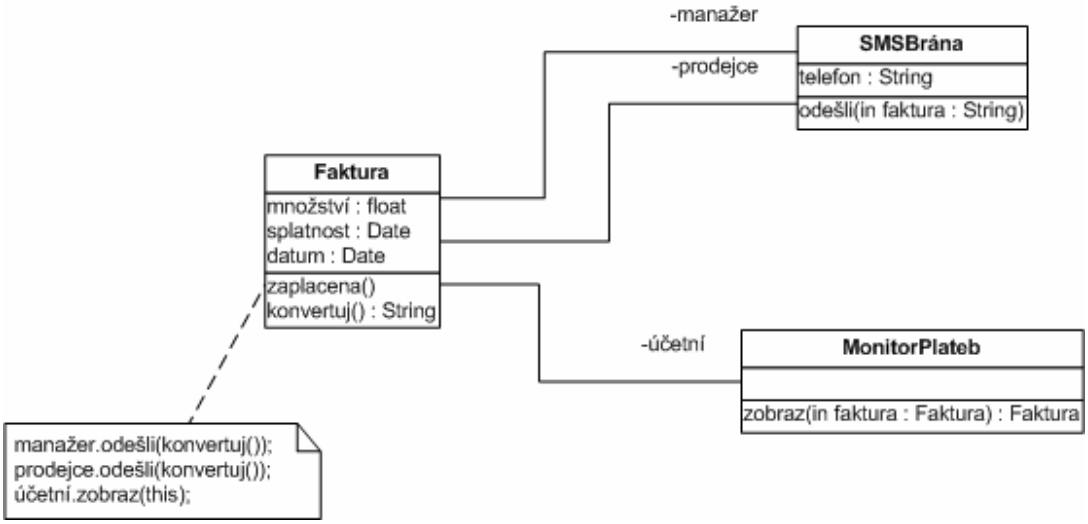


Obr. 5.9.6: Návrhový vzor Kompozit

Posledním z návrhových vzorů, který si uvedeme je návrhový vzor *Pozorovatel,* který je jedním z nejdůležitějších ze skupiny návrhových vzorů chování.

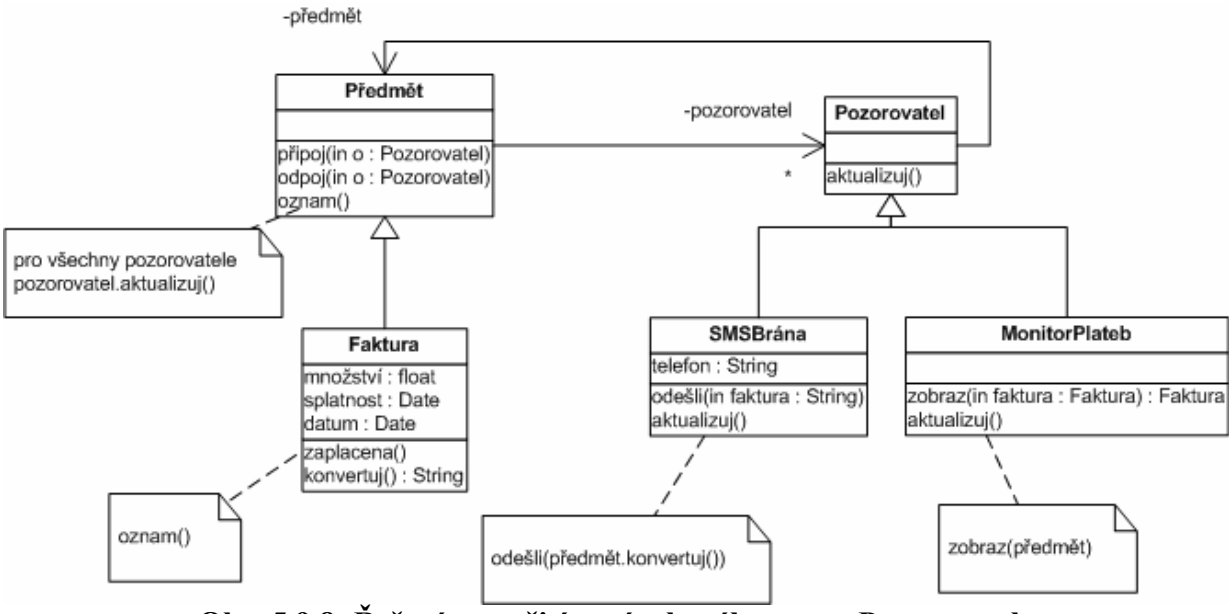
Záměr: Návrhový vzor Pozorovatel definuje závislost obecně n objektů na jediném takovým způsobem, že v případě změny stavu tohoto pozorovaného objektu dojde k oznámení této změny u všech závislých objektů a tyto jsou automaticky na základě této měny aktualizovány.

Motivace: Uvažujme příklad z obr. 5.9.7. Objekt třídy Faktura implementuje operaci *zaplacena* takovým způsobem, že v okamžiku, kdy je tato zpráva obdržena objekt informuje manažera a prodejce zasláním zprávy odešli na instance třídy SMSBrána a stejně tak zasílá zprávu zobraz na instanci třídy MonitorPlateb. Nevýhodou je, že každá faktura musí vědět o těchto svých pozorovatelích, pamatovat si je a zaslat jim odpovídající zprávy. Zavedení dalšího nového pozorovatele by také znamenalo upravit zdrojový kód Faktury.

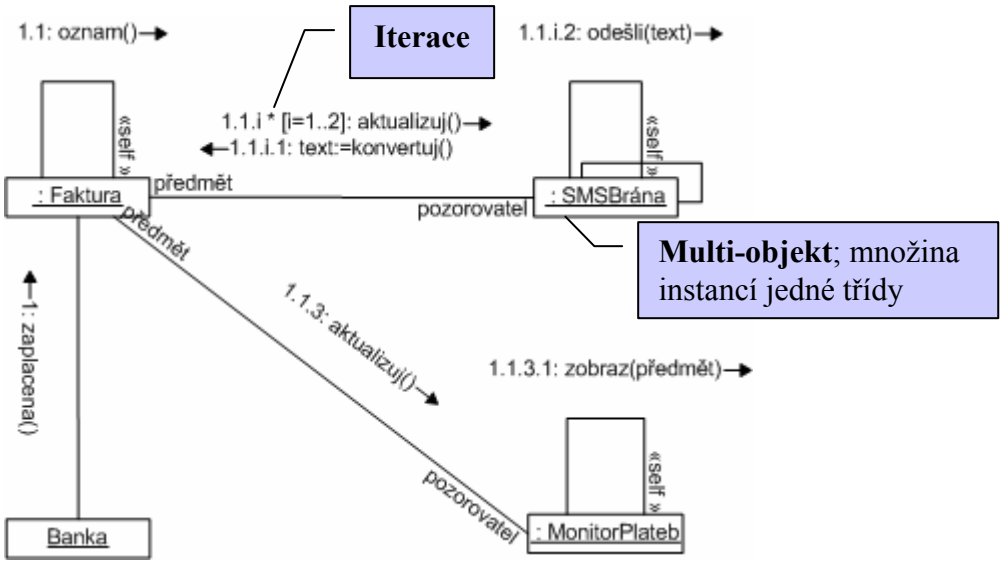


Obr. 5.9.7: Motivační příklad návrhového vzoru Pozorovatel

Řešení: Problém stanovení n pozorovatelů na jediném pozorovaném lze řešit pomocí návrhového vzoru Pozorovatel tím způsobem, že zavedeme dvě nové třídy Předmět pozorování a Pozorovatel (obr. 5.9.8). První z nich zavádí operace přidání a odebrání instancí třídy Pozorovatel a operaci *oznam*, která v případě svého provedení odešle na všechny zaregistrované pozorovatele zprávu aktualizuj. Třída Pozorovatel tedy zavádí operaci *aktualizuj*, která je jednotlivými podtřídami předefinována tak, aby odpovídajícím způsobem dokázaly jednolivé instance správně reagovat – instance třídy SMSBrána odešle textovou zprávu, zatímco instance třídy MonitorPlateb zobrazí informaci o zaplacení faktury na obrazovku. Operace *oznam* je vyvolána podtřídou třídy Předmět, v našem případě se jedná o výše zmíněnou třídu Faktura. Postup celého systému předávání zpráv je dokumentován na diagramu spolupráce (obr. 5.9.9).



Obr. 5.9.8: Řešení s použitím návrhového vzoru Pozorovatel

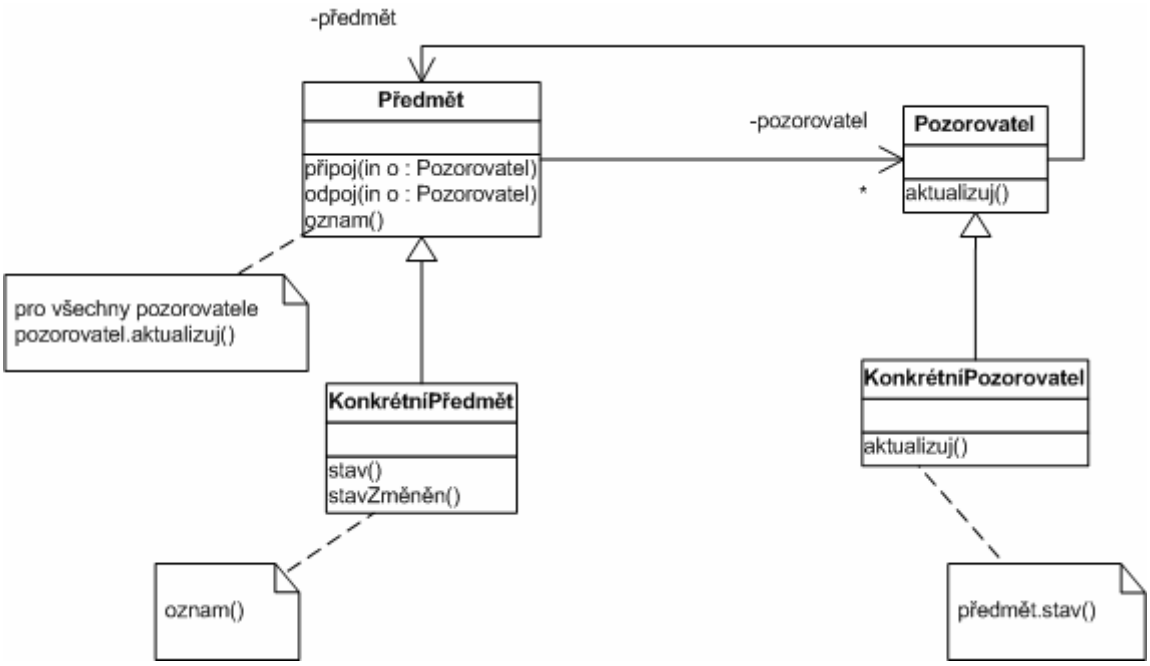


Obr. 5.9.9: Diagram spolupráce návrhového vzoru Pozorovatel

Zobecnění: Abstrakce definující návrhový vzor Pozorovatel (obr. 5.9.10) je opět velmi často používána v různých konkrétních situacích. Nasazuje se v situacích založených na tzv. nepřímé invokaci, kde objekt, který mění své chování a jeho pozorovatelé neudržují navzájem

žádné vazby a jejich propojení je zprostředkována jejich supertřídami. Smysl této nepřímé invokace spočívá v tom, že o přidání nového pozorovatele není nutné informovat (a tím měnit i zdrojový kód) pozorovaný objekt, ale stačí pouze vytvořit novou třídu, která je specializací třídy Pozorovatel a její instance přidat do množiny pozorujících objektů prostřednictvím operace *připoj*.

Tento návrhový vzor tvoří základ všech moderních grafických uživatelských rozhraní, kde změna stavu pozorovaných objektů vede k událostem (obvykle objekt předávaný jako parametr operace *oznam*), které jsou zpracovávány pozorujícími objekty pomocí operací typu *aktualizuj*.



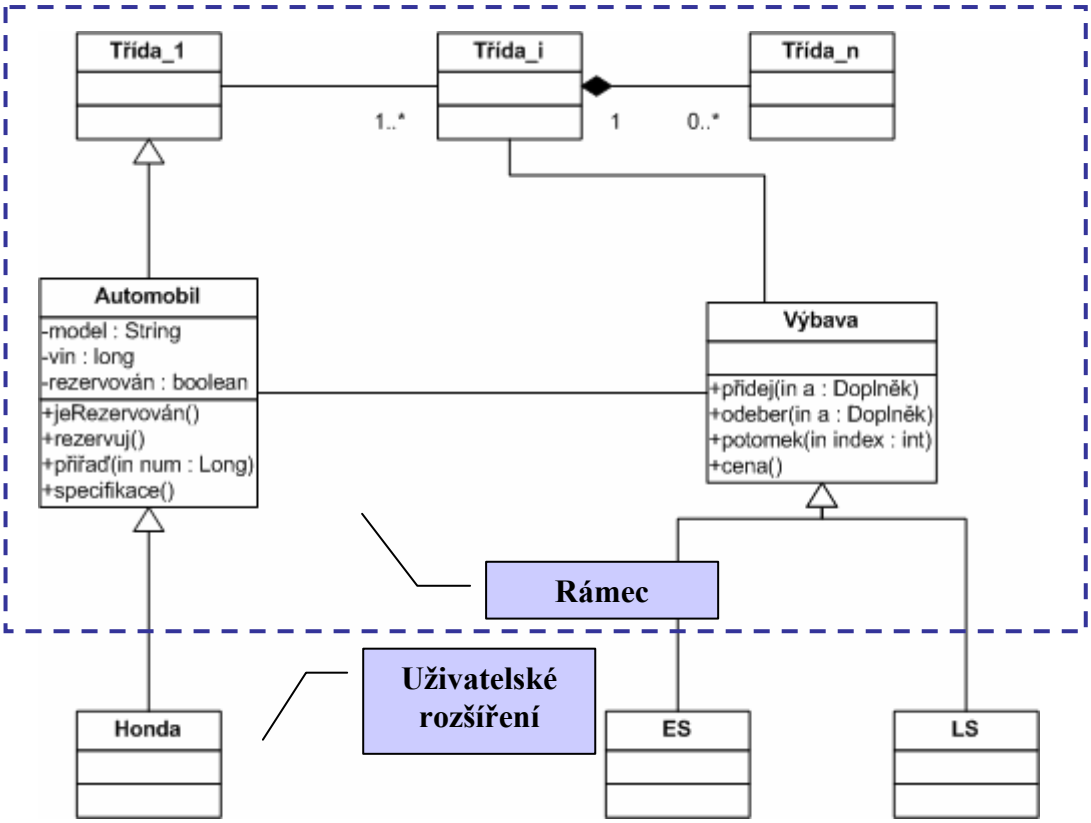
Obr. 5.9.10: Návrhový vzor Pozorovatel

Zatím jsme se naučili v části návrhových vzorů důsledně dodržovat princip znovupoužití užitečných částí jiných softwarových systémů. V podstatě tento princip znovupoužitelnosti můžeme posunout ještě dále v tom smyslu, že bychom opětovně použili téměř celý dříve vytvořený produkt a tento bychom pouze upravili. Pokud tento princip dodržíme a budeme jej systematizovat obdobně jako u návrhových vzorů, vytvoříme tzv. rámec.

Sofwarový rámec je množina abstraktních a konkrétních tříd tvořících dohromady generický softwarový systém.

Rozdíl oproti množině tříd obsažených v knihovnách má rámec své řízení, které se stará o spouštění operací námi dodaných tříd konkretizujících rámec pro danou konkrétní doménovou oblast (obr. 5.9.11).

Z uvedeného vyplývá nutnost návrhu aplikace na velmi vysoké úrovni obecnosti tak, aby ji bylo možné elegantně použít i v jiných úlohách. Jedná se o velmi náročný úkol, který však v případě úspěšného řešení znamená významnou výhodu v soutěži s ostatními firmami zabývajícími se vývojem softwaru dle principu ad hoc, tedy případ od případu.



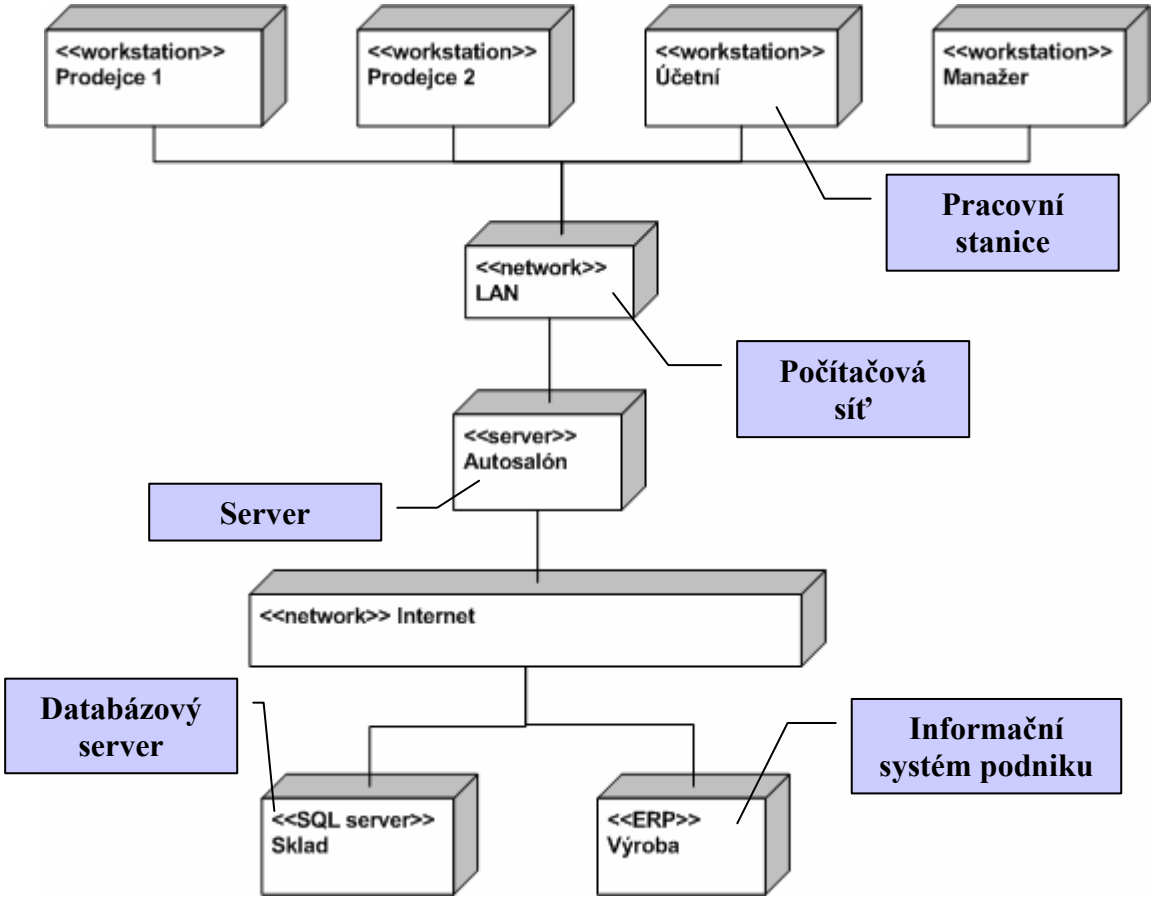
Obr. 5.9.11: Příklad použití rámce

5.10. Diagramy nasazení

Diagramy nasazení definují třetí z modelů vytvářených v rámci toku činností zabývajících se analýzou a návrhem. Tento model nese stejné jméno jako diagramy, které jej definují, tedy model nasazení. Smyslem tohoto modelu je specifikovat fyzickou strukturu výpočetních prostředků, které budou výsledný softwarový produkt provozovat. Odtud tedy lze definovat diagram nasazení následujícím způsobem:

Diagram nasazení popisuje konfiguraci technických prostředků umožňujících běh softwarového systému.

V této etapě analýzy a návrhu mají tyto diagramy pouze hrubý charakter, nicméně jejich sestavení je důležité pro další etapu vývoje reprezentovanou tokem činností nazývaným implementace. Právě v této implementační etapě je nutné logické modely z analýzy a návrhu konkretizovat do podoby softwarových komponent, fyzicky existujících částí softwaru, a tyto umístit na technické prostředky specifikované v diagramech nasazení.



Obr. 5.10.1: Diagram nasazení

Příklad (pokr.):
Informační systém zajišťující prodej automobilů bude využívat celou řadu počítačů navzájem propjených počítačovou sítí. Zaměstanci budou mít každý svou pracovní stanici připojenou pomocí lokální sítě k serveru autosalónu. Tento server bude také vstupní branou do sítě Internet, která umožní napojení jednak na databázový server skladu, případně na služby podnikového informačního systému (ERP – Entreprise Resource Planning) výrobce automobilu.

Shrnutí

V této kapitole byl čtenář obeznámen s problematikou analýzy a následného návrhu softwarového systému. Hlavním cílem tohoto toku činností je sestavit modely reprezentující kompletní dokumentaci implementovaného systému. Ukázali jsme si, jak ze sekvenčních diagramů lze sestavit statický model reprezentovaný diagramem tříd, který tvoří základ této dokumentace. Kromě statického náhledu byl dále představen stavový diagram popisující životní cyklus svým chováním významných objektů. Následný návrh doplňuje třídy a objekty typické pro doménovou oblast o tzv. implementační aspekt, čili o softwarové komponenty, které jsou nám dány k dispozici volbou konkrétního vývojového prostředí. Dále jsme si ukázali jaká je obecná architektura softwarových systémů tvořená svým jádrem (objekty doménové oblasti) a dale třemi vrstvami reprezentovanými uživatelským rozhraním, perzistencí a distribucí. Významná část této kapitoly byla věnována návrhovým vzorům (abstrakcím používaným k řešení typických situací) a také rámcům (tzv. generickým aplikacím). Závěr kapitoly byl věnován specifikaci konfigurace technických prostředků nutných k provozu výsledného softwarového systému.

Kontrolní otázky

1. Proč je diagram tříd základem dokumentace softwarového systému?
2. Jaký je rozdíl mezi modelem analýzy a návrhu?
3. Co jsou to návrhové vzory a jak jsou členěny?

Úkoly

1. Popište v čem spočívá rozdíl mezi agregací a kompozicí z obr. 5.3.5!
2. Sestavte stavový diagram třídy Automobil!
3. Doplňte diagram tříd z obr. 5.4.1. o rozhraní popisující role *vybraný* a *specifikovaný*!

4. Na základě sekvenčních diagramů z příkladu půjčovny videokazet a DVD sestavte diagram tříd popisující model analýzy této úlohy!
5. Vyberte ze sekvenčních diagramů alespoň dvě třídy s významným dynamickým chováním a sestavte pro ně stavové diagramy!

6. Implementace

V této kapitole se dozvíte:

- čím se zabývá tok činností nazývaný implementace
- jaké diagramy jazyka UML se používají ke specifikaci implementačního modelu

Doba potřebná k prostudování této kapitoly jsou:

- 3 hodiny

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

Implementační model specifikuje jak jsou jednotlivé elementy (objekty a třídy) vytvořené v etapě návrhu implementovány ve smyslu softwarových komponent, kterými jsou zdrojové kódy, spustitelné kódy, data a podobně.

Softwarová komponenta je definována jako fyzicky existující a zaměnitelná část systému vyhovující požadované množině rozhraní a poskytující jejich realizaci.

Podle typu softwarových komponent hovoříme o

- zdrojovém kódu, částech systému zapsaném v programovacím jazyce,
- binárním (přeloženém do strojové kódu procesoru) a spustitelném kódu,
- ostatních částech reprezentovaných databázovými tabulkami, dokumenty apod.

Jetliže 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 a programovacího jazyka poskytujícího konkrétní mechanizmus strukturování a modularizace.

Ke splnění těchto cílů, stejně jako v předchozích případech, nabízí jazyk UML prostředky, kterými jsou v tomto případě tyto dva následující diagramy:

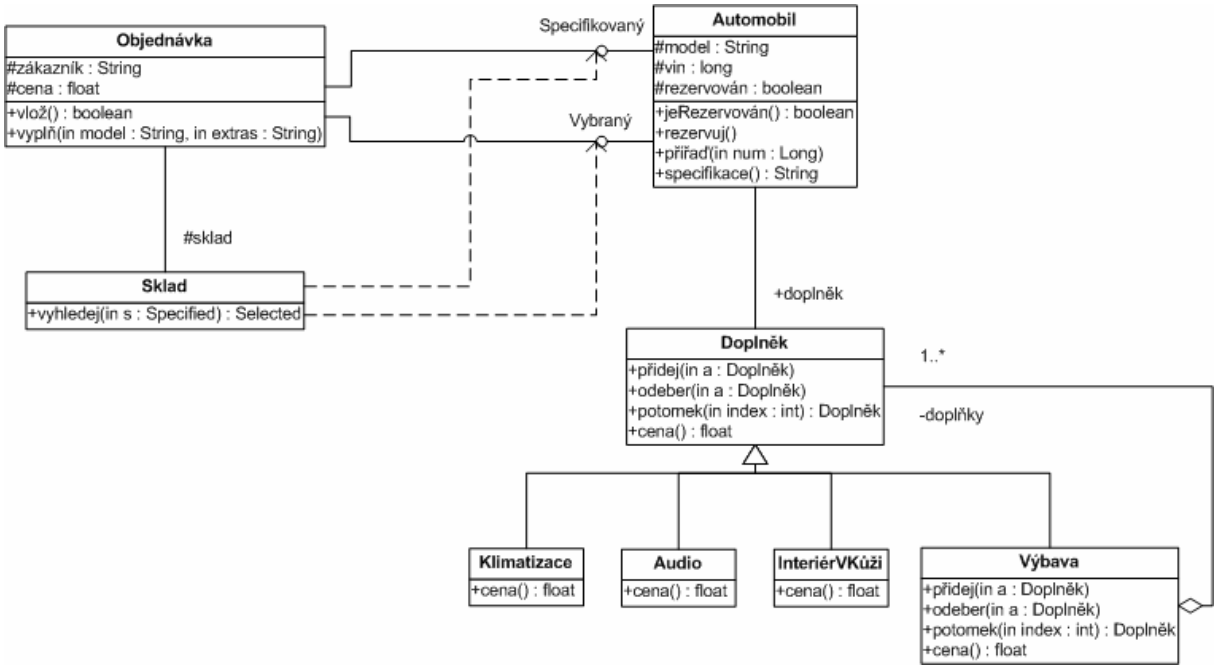
- *Diagram komponent* ilustrující organizaci a závislosti mezi softwarovými komponentami.
- *Diagram nasazení* upřesněný nejen ve smyslu konfigurace technických prostředků, ale především z hlediska rozmístění implementovaných softwarových komponent na těchto prostředcích.

6.1. Mapování elementů logického modelu na komponenty

Důsledná a přesna specifikace objektů a jejich tříd v etapě návrhu umožňuje automatické generování zdrojových kódů dle následující tabulky. Tabulka má dva sloupce, první z nich odpovídá elementům jazyka UML, zatímco druhá z nich popisuje jejich zobrazení v progamovacím jazyce, v našem případě se jedná o jazyk Java.

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

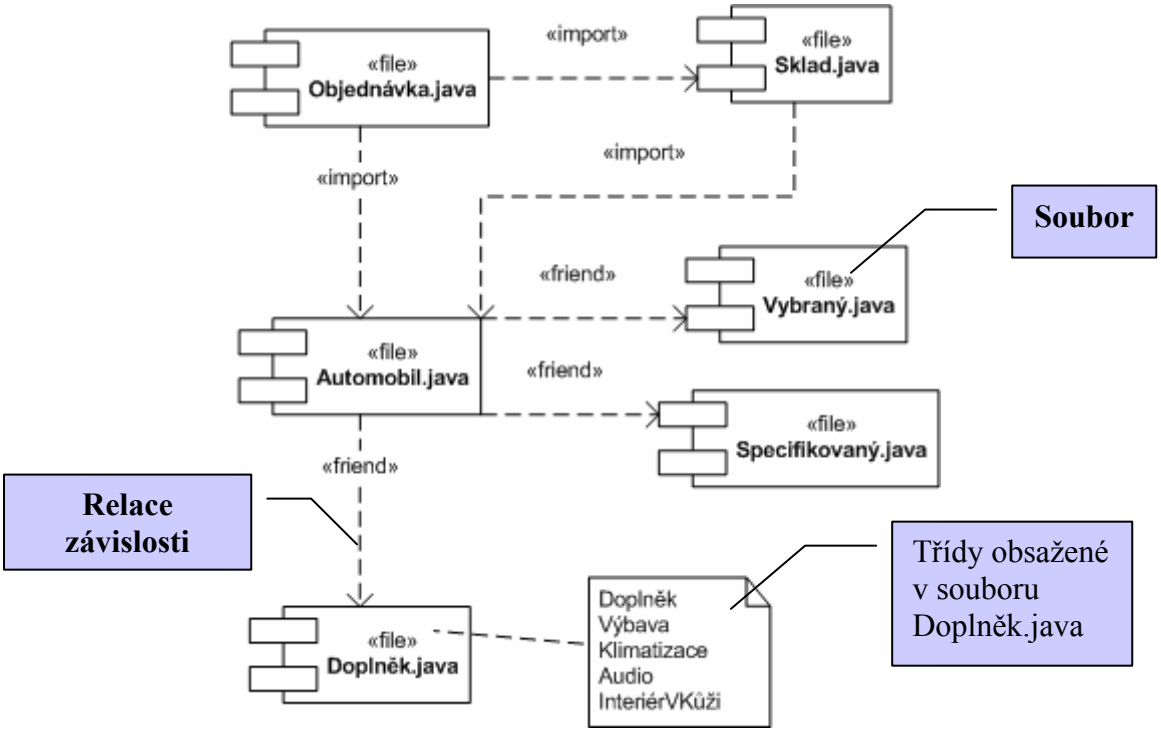
Z uvedené tabulky vyplývá, že jediné co nelze přímo odvodit z diagramů UML jsou těla metod, kód který je proveden v odezvě na přijatou zprávu a také jaká je fyzická struktura vytvářených souborů reprezentujících softwarové komponenty. Vše ostatní lze automaticky vygenerovat doslova bez účasti týmu programátorů. Abychom si mohli výše uvedené dokumentovat, vezměme si diagram tříd (obr. 6.1.1) reprezentující výsledek etapy návrhu a pokusme se z něj vytvořit požadované softwarové komponenty, v našem případě implementované pomocí jazyka Java.



Obr. 6.1.1: Výsledek návrhu

6.2. Zdrojové, binární a spustitelné komponenty

Ukázku vytvoření komponent zahájíme částí týkající se zdrojových komponent, tedy souborů vytvořených pomocí použitého programovacího jazyka. První krok tedy bude rozhodnout o fyzické organizaci těchto zdrojových souborů definovaných pomocí diagramu komponent.



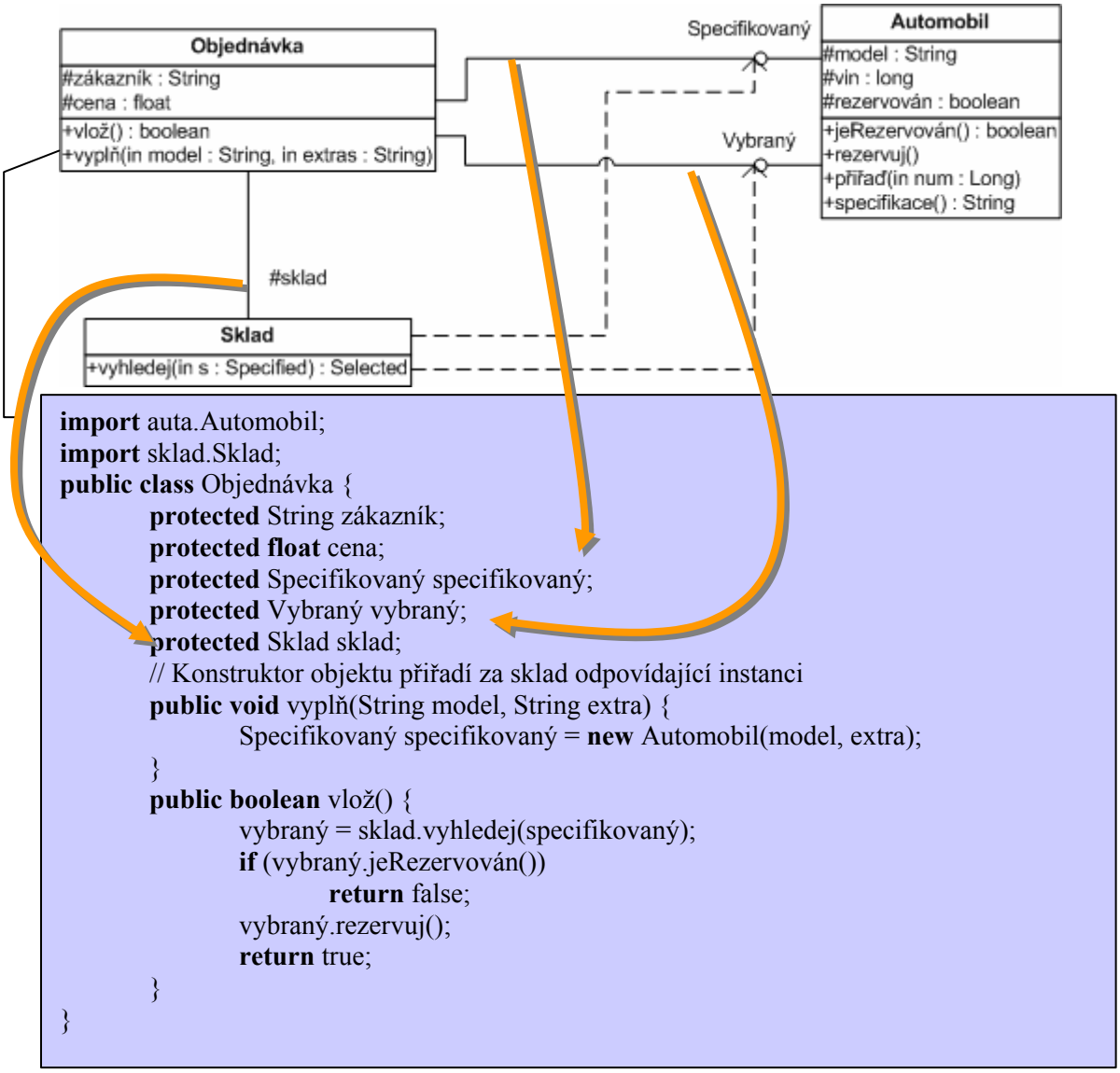
Obr. 6.2.1: Diagram komponent: zdrojový kód

Příklad (pokr.):

Z výsledneho diagramu tříd specifikující jádro aplikace prodeje automobilu vytvořme diagram komponent (obr. 6.2.1). Třídy Automobil, Objednávka, Sklad, Vybraný a Specifikovaný budou každá obsažena v samostatném souboru totožného názvu s koncovkou .java označující, že se jedná o kód zapsaný v programovacím jazyce Java. Hierarchii tříd popisující část doplňků (s využitím návrhového vzoru Kompozit) umístíme do jediného souboru odpovídající svým názvem supertřídě Doplněk. Relace závislosti nazvané jako <<import>> označují použití komponent z jiných subsystémů (balíčků), zatímco relace nazvané jako <<friend>> označují, že dané komponenty spadají do společného balíčku. V jazyce Java to znamená, že se nacházejí jejich soubory ve společném adresáři.

Máme-li takto dánu fyzickou strukturu, pak tým programátorů může započít s vytvářením zdrojových kódů. Pokud máme k dispozici vývojové nástroje typu CASE (Computer Aided Software Engineering), pak lze nechat tyto zdrojové kódy takovým nástrojem vygenerovat.

Postup vytváření zdrojových kódů vychází z tabulky mapování elementů jazyka UML na zdrojový kód jazyka Java. Jediné co je nutné doplnit, co nelze automaticky či jednoduše vytvořit, jsou těla metod reprezentované algoritmy operací. Tyto lze alespoň hrubým způsobem specifikovat jednak ze sekvenčních diagramů definujících operace, ale především pak ze stavových diagramů popisujících detailně chování objektů.



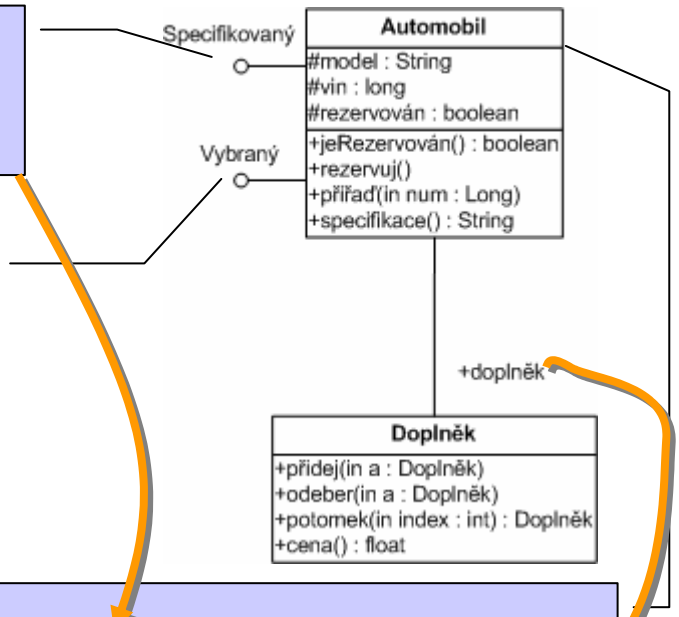
Obr. 6.2.2: Zdrojový kód: Objednávka.java

Příklad (pokr.):

Třída Objednávka využívá tříd Automobil a Sklad umístěných v balíčcích *auta* a *sklad*. Proto dle specifikace v diagramu komponent je nutné tyto třídy importovat. Atributy *zákazník* a *cena* jsou implementovány v podobě instančních proměnných stejně jako asociace na instance jiných tříd pojmenované jako *specifikovaný*, *vybraný* (jména rolí instance třídy Automobil) a *sklad*. Metody, jak vyplývá z diagramu tříd, jsou dvě *vyplň* a *vlož*.

```
package auta;
public interface Specifikovaný {
        String specifikace();
}
```

```
package auta;
public interface Vybraný {
        boolean jeRezervován();
        void rezervuj();
}
```

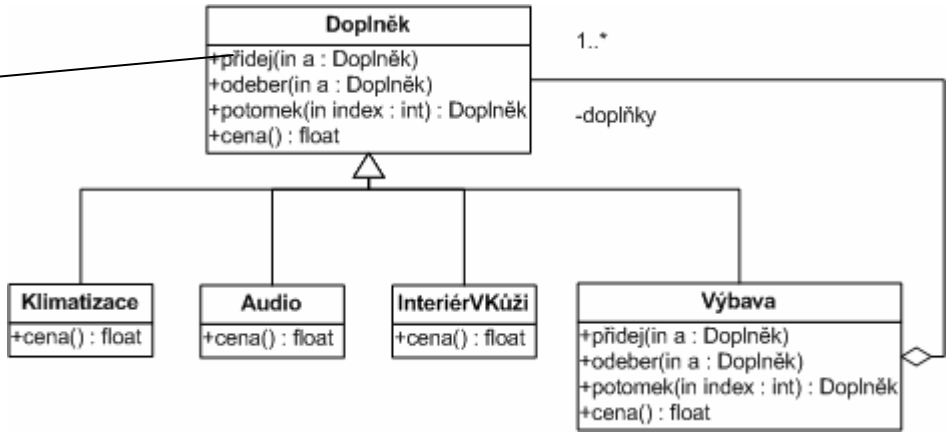



```
package auta;
public class Automobil implements Vybraný, Specifikovaný {
        protected String model;
        protected long vin;
        protected boolean rezervován;
        public Doplněk doplněk;

        public boolean jeRezervován() {
                return rezervován;
        }
        public void rezervuj() {
                rezervován = true;
        }
        public void přiřaď(long num) {
                vin = num;
        }
        public String specifikace() {
                return model;
        }
}
```

Obr. 6.2.3: Specifikovaný.java, Vybraný.java a Automobil.java

Příklad (pokr.):

Za pozornost v této části implementace stojí způsob, jakým jazyk Java realizuje rozhraní. K tomuto účelu slouží klíčové slovo *implements*, které v definici třídy specifikuje, která rozhraní (v našem případě Vybraný s Specifikovaný) jsou v rámci dané třídy implementována. Z hlediska samotné třídy platí, že musí definovat metody deklarované v rámci rozhraní, tedy *specifikace* z rozhraní Specifikovaný a *jeRezervován* a *rezervuj* z rozhraní Vybraný. V případě chybějících implementací těchto metod překladač jazyka Java bude hlásit chybu

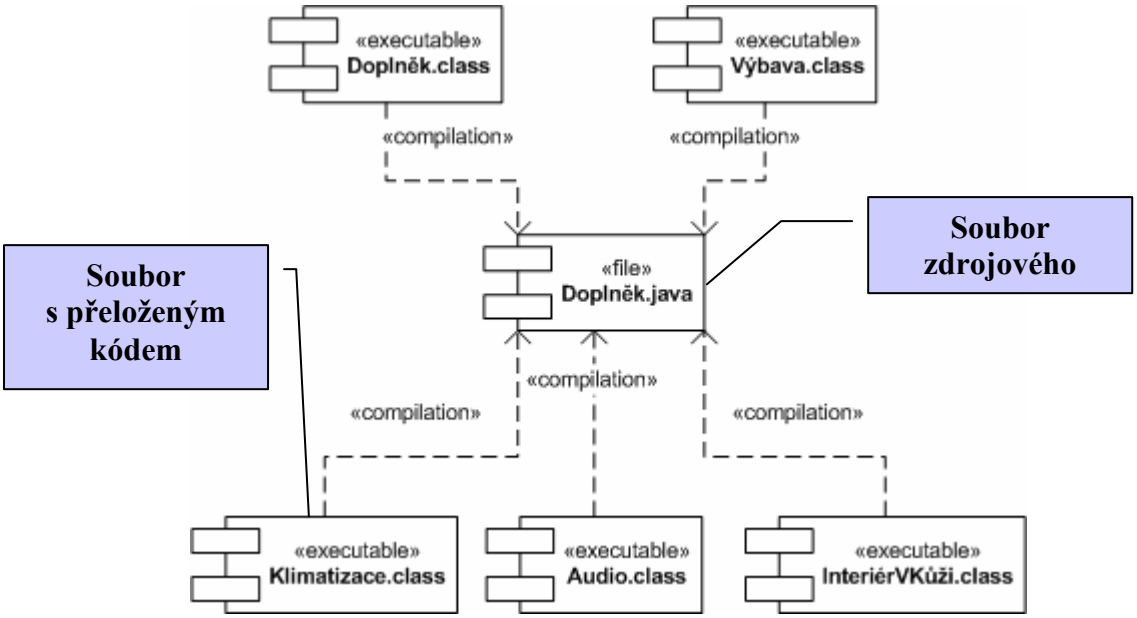



```
package auta;
public abstract class Doplněk {
        public void přidej(Doplněk d) {}
        public void odeber(Doplněk d) {}
        public Doplněk potomek(int index) {
                return null;
        }
        public abstract float cena();
}

class Klimatizace extends Doplněk {
        public float cena() {
                return 2000.0f;
        }
}
// ostatní třídy
class Výbava extends Doplněk {
        private Doplněk[] doplňky;

        public void přidej(Doplněk d) {/*  kód  */}
        public void odeber(Doplněk d) {/*  kód  */}
        public Doplněk potomek(int index) {
                return doplněk[index];
        }
        public float cena() {
                float sum=0;
                for (int i=0; i < doplňky.length; i++) {
                        sum = sum + doplňky[i].cena();
                }
                return sum;
        }
}
```

Obr. 6.2.4: Doplněk.java

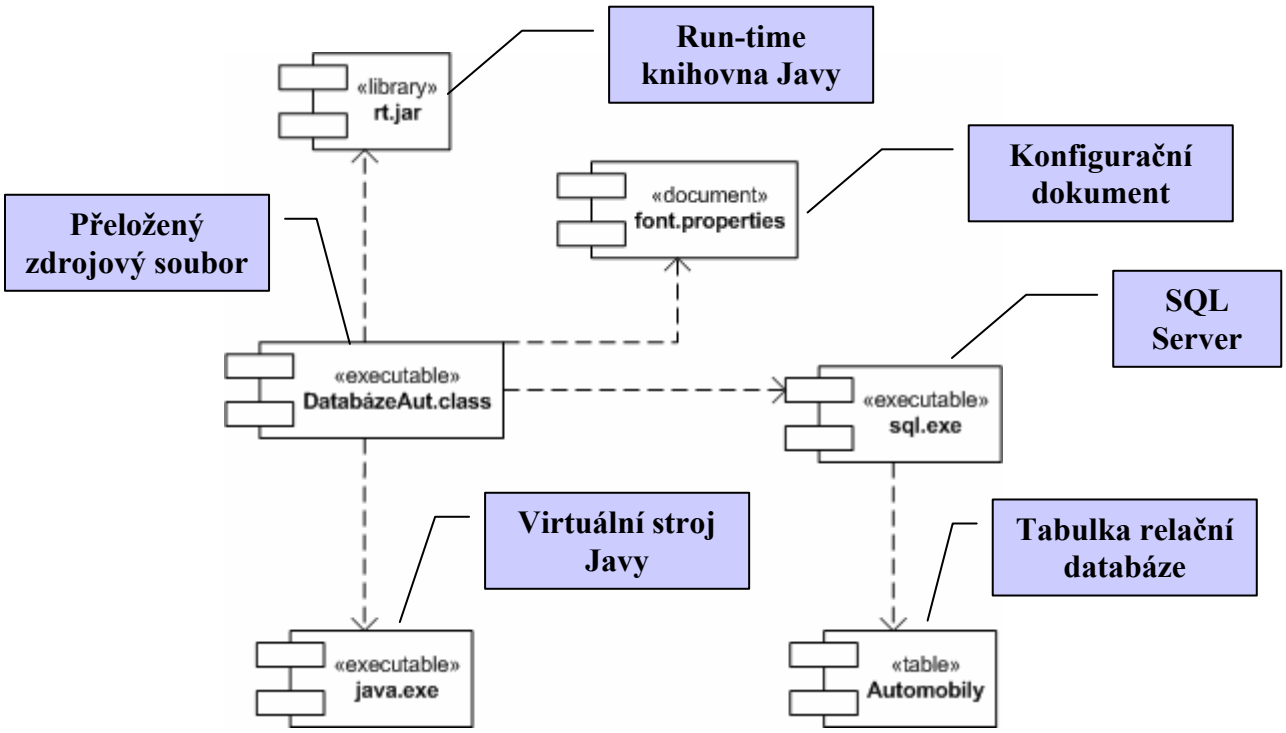
Příklad (pokr.):
Poslední z příkladů implementace zdrojových komponent dokumentuje způsob jakým je v jazyce Java řešena relace generalizace (dědičnost). Jedná se použití klíčového slova *extends*, které uvozuje supertřídu dané třídy. Tedy např. třída Výbava je podtřídou třídy Doplněk.

Binární komponenty, tedy komponenty přeložené do strojového kódu daného procesoru, mají také svůj diagram komponent obvykle vyjadřující z jakých zdrojových souborů jsou požadované binární vytvořeny.



Obr. 6.2.5: Diagram komponent: binární kód

Diagram spustitelných komponent specifikuje všechny komponenty vytvořené námi i ty, které nám dáva k dispozici implementační prostředí.

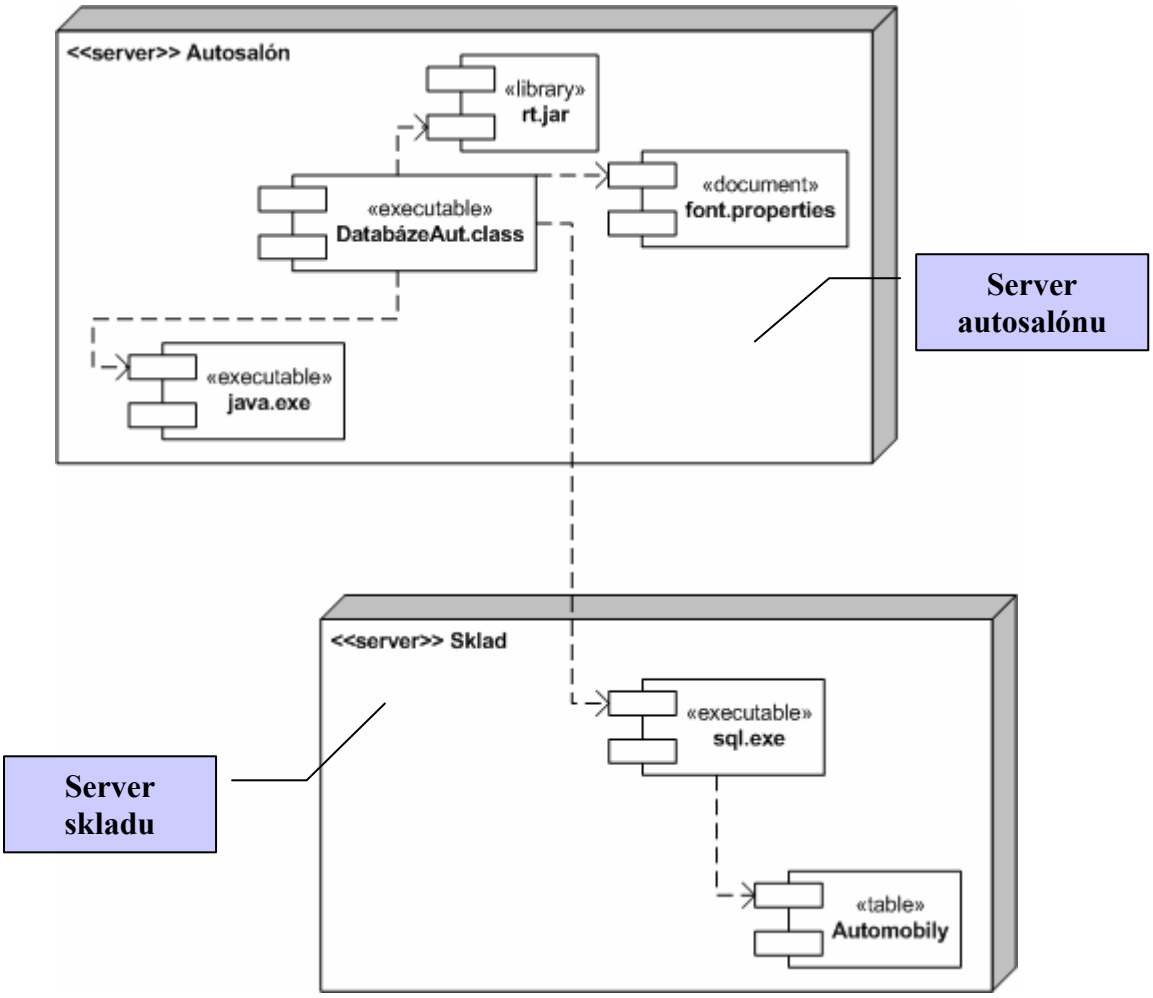


Obr. 6.2.6: Diagram komponent: běh programu

Příklad (pokr.):
V našem příkladu ke spuštění třídy Databáze aut potřebujeme jednak virtuální stroj jazyka Java (java.exe) a jeho knihovny (rt.jar). Konfigurační parametry fontů jsou uloženy v textovém souboru (font.properties). Samotná aplikace využívá služeb databázového serveru (sql.exe) pracujícího s databází aut strukturovanou v relačních tabulkách (Automobily).

6.3. Diagramy nasazení

Posledním úkolem, který činí etapu implementace kompletní je zpřesnění diagramů nasazení. Vlastní zpřesnění spočívá v umístění spustitelných a datových komponent na jednotlivé technické prostředky reprezentované počítači.



Obr. 6.3.1: Finální diagram nasazení

Shrnutí

V této kapitole byl čtenář obeznámen s problematikou implementace softwarového systému. Hlavním cílem tohoto toku činností je sestavit modely specifikující jak jsou fyzicky organizovány softwarové komponenty a hlavně, jak se tyto komponenty vytváří na základě diagramů sestavených v etapě návrhu. Ukázali jsme si mechanismus mapování specifikace tříd z UML diagramů na zdrojový kód programovacího jazyka. Závěr kapitoly byl věnován

diagramu nasazení, který dokumentuje jak jsou jednolivé komponenty rozmístěny na počítačích.

Kontrolní otázky

1. Jaké diagramy využívá impementace ke specifikaci fyzické organizace kódu?
2. Jak jsou implementovány relace generalizace v programovacím jazyce Java?

Úkoly

1. Definujte pojem komponenta a jaké typy komponent obsahuje model implementace!
2. Jakým způsobem je v programovacím jazyce vyjádřeno, že třída Výbava se skládá z obecně n instancí třídy Doplněk!

3. Na základě diagramů tříd z příkladu půjčovny videokazet a DVD sestavte diagram zdrojových komponent!
4. Implementujte kód tříd tvořících jádro aplikace půjčovny videokazet a DVD!

7. Testování a nasazení softwarového produktu

V této kapitole se dozvíte:

- čím se zabývá tok činností nazývaný jako testování a nasazení výsledného produktu
- jak se postupuje při testování, co je to verifikace a validace
- co je obsahem pojmu nasazení systému.

Doba potřebná k prostudování této kapitoly jsou:

- 1,5 hodiny

Testování se provádí z pohledu tří základních dimenzí reprezentovaných kvalitou, funkcionalitou a výkonností systému. Testování se týká všech vytvářených modelů a jejich diagramů.

Principy používané při testování jsou následující:

- Testy se plánují pro každou iteraci a zahrnují integrační testy a testy systémové. Integrační testy se provádí pro každý vytvořený produkt během iterace, zatímco systémový test se provádí pouze na konci iterace, kdy vzniká spustitelná verze vyvíjeného produktu.
- Testy se navrhují a následně implementují v podobě testovacích úloh, které jednoznačně definují co se má ověřit. Z tohoto pohledu hovoříme o testovacích procedurách, které specifikují jak se má test provádět, nebo se vytváří spustitelné testovací komponenty umožňující automatizaci procesu ověřování.
- Výsledky provedených testů jsou systematicky zprácovávány a vadné části jsou opakovaně testovány a případně zaslány zpět do toků činností jako je analýza, návrh nebo implementace s cílem nedostatky odstranit.

7.1. Cíle procesu verifikace a validace

Testování softwarových systémů probíhá dvěmi způsoby nazývanými verifikace a validace:

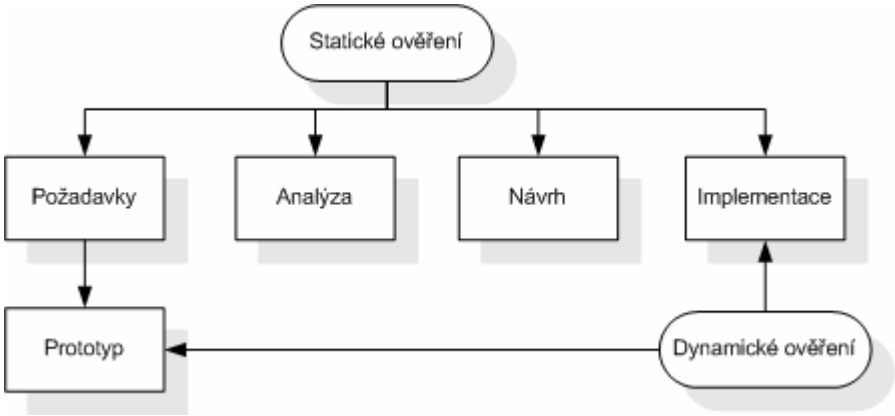
- *Verifikace* je proces testování hledající odpověď na otázku, zda-li softwarový produkt je vytvářen správně. Jinými slovy řečeno, hledáme nedostatky v samotném softwarovém systému.
- *Validace* je proces testování hledající odpověď na otázku, zda-li vytvářený software je správný. Jinými slovy, zda-li implementuje požadovanou funkcionalitu.

Z uvedeného vyplývá, že může nastat situace, kde systém je perfektně verifikován, ale je k ničemu, neboť nesplňuje zadavatelem specifikovanou funkcionalitu. Na druhou stranu systém splňuje všechny funkce, ale je nestabilní nebo pomalý.

Kromě verifikace a validace také rozlišujeme mezi statickým a dynamickým ověřováním (obr. 7.1.1):

- Statické techniky se zabývající testováním vytvořených modelů a jejich diagramů, čili umožňují pouze ověřit korespondenci mezi vytvořeným systémem a jeho specifikací.

- Dynamické techniky zkouší již implementovaný produkt cestou jeho spuštění a tím ověřují i jeho užitečnost a kvalitu.



Obr. 7.1.1: Statické a dynamické ověřování aplikace

7.2. Modely testování

Na rozdíl od předchozích modelů, které byly definovány různými typy diagramů, pro testování nemá jazyk UML k dispozici žádný diagram. Model testování je dán souborem:

- *testovacích úloh* definujících co se má u systému testovat,
- *testovacích procedur* specifikujících postupy pro provedení testovacích úloh,
- *testovacích komponent*, které automatizují testovací procedury.

Testovací úloha specifikuje způsob testování systému zahrnující informace o tom, co se má testovat s jakými vstupy a za jakých podmínek.

V podstatě existují dva typy nejběžněji používaných testovacích úloh:

- Testovací úlohy určené k ověření případů užití nebo jejich specifických scénářů. Takové úlohy ověřují výsledky interakce aktéra se systémem. Tento tzv. *black-box* test ověřuje z vnějšku pozorovatelné chování systému.
- Testovací úlohy specifikující jak ověřovat realizaci případů užití. Tyto úlohy ověřují interakce mezi komponentami implementující případ užití. Tento tzv. *white-box test* je určen k ověření vnitřní intrakcí daného systému.

Testovací procedura specifikuje jak provést jednu nebo více testovacích úloh nebo jejich částí.

Testovací procedury lze specifikovat pomocí:

- instrukcí popisujícími jak ručně provést danou testovací úlohu,
- popisu jak vytvořit spustitelnou testovací komponentu,
- specifikace jak použít nástroj pro automatické testování.

Testovací komponenty automatizují jednu nebo více testovacích procedur nebo jejich části.

Testovací komponenty jsou používány k ověření komponent tvořících implementační model cestou poskytnutí požadovaných vstupů, řízení a sledování běhu ověřované komponenty a vytvoření výsledné zprávy o průběhu testu. K vytvoření testovacích komponent se mohou používat různé skriptovací jazyky nebo k tomuto účelu vytvořené systémy automatizované verifikace.

7.3. Nasazení softwarového systému

Účelem toku činností nazvaného jako nasazení je úspěšně vytvořit výsledný produkt a dodat jej koncovým uživatelům.

Nasazení softwarového systému pokrývá celou řadu činností jako:

- vytvoření výsledného produktu či jeho verzí
- kompletace softwarového systému
- distribuce softwarového systému
- instalace softwarového systému u uživatele
- poskytnutí asistenční služby uživatelům
- plánování a řízení beta testování
- migrace již existující dat a softwarových produktů.

Shrnutí

V této kapitole byl čtenář obeznámen s problematikou testování softwarového systému. Byly zavedeny pojmy verifikace a validace jako procesy ověření správnosti vytvářeného systému a jeho funkcionality. Byly zavedeny pojmy testovací úloha, procedura a komponenta jako tři základní složky modelu testování. Závěr kapitoly byl věnován problematice činností spojených s vytvořením výsledného produktu a jeho nasazení u koncového uživatele.

Kontrolní otázky

1. Jaký je rozdíl mezi procesem verifikace a validace?
2. Co je to testovací úloha, procedura a komponenta?

Úkoly

1. Definujte pro případ užití popisující prodej automobilu testovací úlohu typu black-box testu a proceduru založenou na instrukcích pro její ruční provedení!
2. Definujte pro stejný případ užití white-box test včetně instrukcí jak jej realizovat!