

Proces vývoje programového systému

Objektově orientovaný přístup vznikl v oblasti programování, ale postupně se rozšířil i do dalších fází vývoje programového systému. Vznikaly různé metodiky objektové analýzy a návrhu programového systému.

Metodiky návrhu programového systému definují:

- **procesy** množinu předem stanovených kroků při návrhu SW,
- **notaci** jazyk - grafické symboly a textové značky pro reprezentaci prvků softwarových systémů,
- **heuristiky** množinu pravidel uplatňovaných při návrhu SW.

Podrobné zvládnutí jedné či několika metodik přesahuje rámec těchto skript, podrobnější popis můžeme nalézt například v [OOMT]. Metodiky se většinou zaměřují na vývoj velkých programových systémů a podrobně rozpracovávají jednotlivé fáze tohoto vývoje. Naším cílem je v tomto textu poskytnout studentům jen základní představu o objektově orientované analýze, návrhu a implementaci programového systému (v prostředí Delphi nebo Javy).

Notace UML

Jednotlivé metodiky objektově orientované analýzy a návrhu (Coad-Yourdon, OMT - Rumbaugh, Booch, OOSE- Jacobson, OOMT) používaly různou notaci pro vyjádření prvků návrhu. Pro výrobce CASE nástrojů bylo problematické zvolit notaci, kterou budou podporovat. Proto firma Rational Software Corporation povolala 3 nejvýznamnější představitele objektově orientovaných metodik (Grady Booch, Jim Rumbaugh, Ivar Jacobson), kteří se spojili a společně vypracovali jednotnou notaci nazvanou **UML (Unified Modelling Language)**, která byla poprvé publikována v roce 1995. Notace se neustále rozvíjí a je podporována celou řadou CASE nástrojů. V současné době zahrnuje 10 druhů diagramů, které reprezentují různé stránky návrhu programového systému. Důležité je si uvědomit, že UML je jen jazyk pro záznam analýzy a návrhu, ne metodika. Existují různé metodiky používající jako notaci UML. Sama firma Rational Software Corporation vypracovala i metodiku pro vývoj SW, kterou distribuuje pod názvem Rational Unified Process (RUP).

UML zahrnuje následující typy diagramů:

- diagram užití (Use Case diagram, model jednání),
- diagram tříd (Class diagram),
- diagram objektů (object diagram),
- sekvenční diagram (sequence diagram),
- diagram spolupráce (collaboration diagram),
- diagram aktivit (activity diagram),
- stavový diagram (state diagram),
- diagram komponent (component diagram),
- diagram balíčků (package diagram),
- diagram nasazení (deployment diagram).

Jednotlivé typy diagramů reprezentují různé dimenze pohledu na navrhovaný systém. Některé diagramy jsou statické (diagram tříd), jiné zachycují dynamiku v systému (stavový diagram). Některé diagramy se dívají na systém z lidského hlediska (diagram užití), jiné jsou spíše technologické (diagram nasazení, diagram komponent). V tomto textu se zaměříme zejména na diagram užití, diagram tříd a sekvenční diagram.

Poznámka:

České termíny pro jednotlivé typy diagramů UML nejsou ještě ustálené. V různých zdrojích se můžete setkat s různými názvy, zejména pokud jde o diagram užití (Use Case diagram).

Diagram užití

Diagram užití popisuje chování systému z hlediska uživatele. Modeluje úlohy, které vykonávají uživatelé, reprezentovaní aktory, s využitím navrhovaného SW. Každá úloha je reprezentována jako **typ užití** (Use case) a celý systém je souhrnem takových typů užití - modeluje se pomocí **diagramu užití**. Tento diagram popisuje i okolí navrhovaného systému a tím vymezuje jeho hranice.

V diagramu užití se specifikuje:

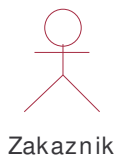
- jaké typy uživatelů (lidé i jiné systémy) používají systém,
- jaké činnosti vykonávají jednotlivé typy uživatelů,
- jaké činnosti nemohou jednotlivé typy uživatelů vykonávat.

Diagram užití obsahuje tyto prvky:

- **aktor,**
- **typ užití,**
- **vztah** (relace).

Aktor (Actor)

Aktor reprezentuje prvek okolí systému, který buď informace dostává nebo je přijímá.



Obrázek 3.1 : Vyjádření aktora v UML

Pokud vyjmenujeme všechny aktory, specifikujeme okolí systému a vymezíme tak jeho hranice. Aktorem nemusí být pouze živá osoba (obsluha, manažer, ředitel, aj.), ale také jiné systémy (externí), komponenty pro přístup k datům apod. Aktor se v UML označuje panáčkem, u kterého se uvádí stručný a výstižný popis (například recepční, manažer, vedoucí, externí systém, bankovní systém atd.).

Typ užití

Typ užití specifikuje jeden prvek funkcionality systému, kterou využívá aktor (charakterizuje určitý způsob použití systému uživatelem).



ProhlizeniZajezdu

Obrázek 3.2 : Vyjádření typu užití v UML

Typ užití je chápán jako množina následných transakcí v systému, které v konečném důsledku vedou ke splnění požadované funkcionality. Znázornění typu užití v notaci UML je uvedeno na obrázku 3.2. Typ užití se označí jménem, je

vhodné k němu připojit také slovní popis - scénář typu užití. Ve slovním popise se nepoužívají pojmy jako objekty, třídy, ale pouze pojmy z dané problémové oblasti (zákazník, rodné číslo, adresa atd.).

Vztah (relace)

Vztahy v diagramu užití mohou být dvojího typu:

- vztah mezi aktorem a typem užití
- vztah mezi typy užití

Vztah mezi aktorem a typem užití

Vztah mezi aktorem a typem užití vyjadřuje tok informace mezi vnějším prvkem (aktorem) a typem užití aplikace. Někdy se tento vztah označuje jako **komunikační asociace**

a znázorňuje se šipkou mezi aktorem a typem užití. Definování těchto toků je velmi důležité, protože se tím vymezí hranice aplikace.



Obrázek 3.3: Vztah mezi aktorem a typem užití v UML

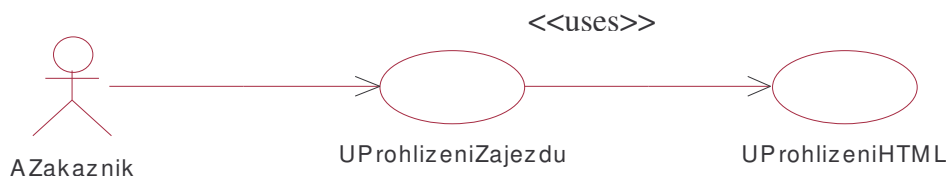
Vztah mezi typy užití

Vztah mezi typy užití dává do určitého vztahu dva typy užití. Existují dva typy vztahů mezi typy užití:

- **Uses**
- **Extends**

Vztah uses

Při tvorbě diagramu užití se mohou některé typy užití v různých částech navrhovaného systému opakovat - například přihlášení do systému nebo výběr prvku ze seznamu. V těchto případech je lepší opakující se činnosti vyjmout do samostatného typu užití a odkázat se na ně v jiných typech užití, které je používají, pomocí relace uses.



Obrázek 3.4: Vztah uses v UML

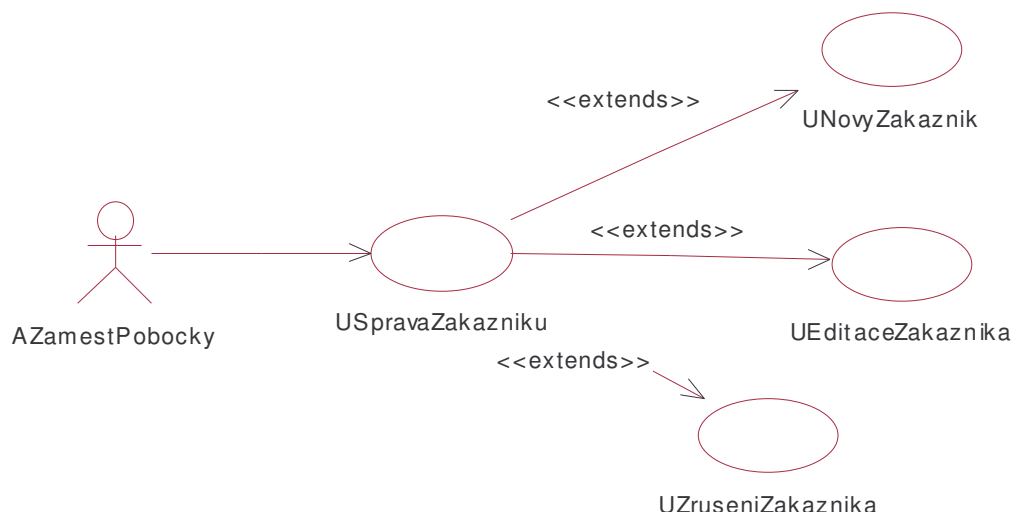
Vztah extends

Vztah extends představuje vztah generalizace - specializace v diagramech užití. Umožňuje zjednodušit hlavní diagram užití, který se vyjádří pomocí obecných typů užití. Ty je možné v dalším kroku zpodrobnit a specifikovat u nich podmíněné větve, chybové stavy apod. Vztah extends mezi typy užití se používá v případě:

- volitelného chování aplikace ,
- chování za specifických podmínek,
- chování různých toků aplikace podle volby aktora.

Slovní popis typů užití (slovní scénáře)

Diagram užití dává přehled o jednotlivých způsobech použití navrhovaného systému. Každý typ užití je však třeba doplnit slovním popisem. V popise se používají slova problémové oblasti nikoli programátorský žargon. Je vhodné popsat i kontroly formulářových položek, unikátnosti položek v systému a také chybové stavy a reakce systému na ně.



Obrázek 3.5: Vztah extends v UML

Diagram tříd (Class diagram)

Diagram tříd reprezentuje strukturu tříd v rámci systému, u každé třídy zachycuje atributy a metody a vyjadřuje vztahy mezi třídami. Představuje statický pohled na modelovaný systém. Nelze jím vyjádřit interakce mezi třídami, ke kterým dochází v čase. Tento diagram se vytváří v etapě analýzy a postupně se doplňuje a zpřesňuje v dalších fázích tvorby systému (viz 3.2). Je také základem pro implementaci a nástrojem dokumentace.

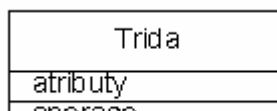
Můžeme tedy mluvit o různých úrovních diagramu tříd:

- **konceptuální** (celková hierarchie tříd),
- **specifikační** (třídy a rozhraní),
- **implementační** (včetně skrytých atributů a všech metod).

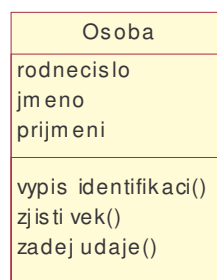
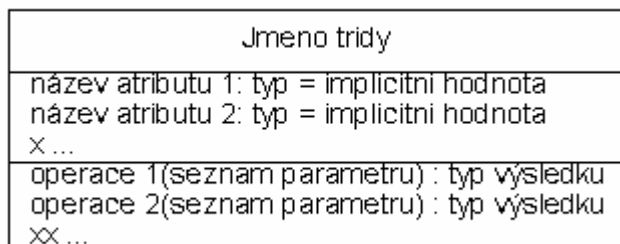
Základním prvkem diagramu tříd je třída. Třída se znázorňuje obdélníkem, který se skládá ze tří částí:

- **jméno třídy**,
- **datové položky** třídy - atributy,
- **operace** - metody třídy.

Symbol pro třídu v notaci UML je uveden na obrázku 3.6, úplná notace zápisu třídy je uvedena na obrázku 3.7. Příklad třídy *Osoba* je zachycen na obrázku 3.8.



Obrázek 3.6: Zápis třídy v UML



Obrázek 3.7: Úplný zápis třídy

v UML

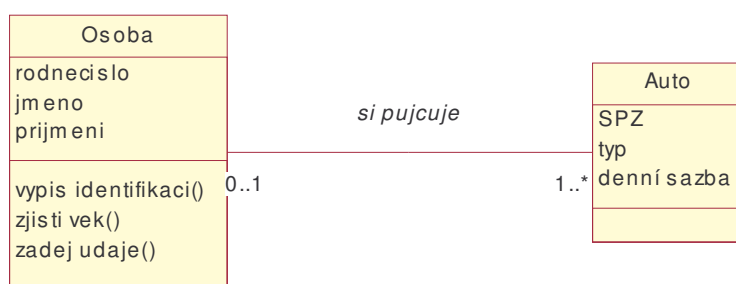
Obrázek 3.8: Příklad třídy *Osoba*

Asociace

Když modelujeme nějaký systém, identifikujeme v něm objekty a seskupíme je do tříd. Objekty však nejsou izolované. Každý objekt má v sobě kromě dat i operace s těmito daty, ale to pro celý systém nestačí. Objekt ke svému chování zpravidla potřebuje využít schopností jiného objektu. Objekty jsou ve vzájemném vztahu (komunikují spolu, posílají si zprávy). Tyto vztahy modelujeme jako asociace. Asociace může být analyticky chápána dvojím způsobem - jako agregace nebo běžná asociace.

Běžná asociace

Při běžné asociaci získává vnější objekt referenci na vnitřní objekt zvenku a dočasně. Tento vnitřní objekt tedy není vlastnictvím vnějšího objektu, ale je mu jen dočasně zapůjčen - předává se formou parametru zaslané zprávy (metody). Proto vnější objekt nemůže vnitřní objekt vytvářet ani rušit. Příklad běžné asociace mezi třídou *Osoba* a *Auto* je znázorněn na obrázku 3.9. Asociace je dobré pojmenovat, přičemž jméno vyjadřuje význam daného vztahu (v našem případě *si půjčuje*)



Obrázek 3.9: Běžná asociace

Násobnost (Multiplicity)

Násobnost určuje, kolik instancí jedné třídy má vztah k jedné instanci asociované třídy. Násobnost se často popisuje jako „jeden“ nebo „mnoho“. UML umožňuje určit násobnost přesněji. Násobnost se značí speciálními symboly na konci čáry asociace. V tabulce 3.1 jsou uvedeny možnosti vyjádření násobnosti v UML.

Symbol	význam	anglický výraz
1	právě jeden	Exactly one
1..*	jeden a více	One or more
0..*	nula a více	Zero or more
0..1	nula nebo jeden	Zero or one
m..n	určený interval (např. 4..7)	Specified range

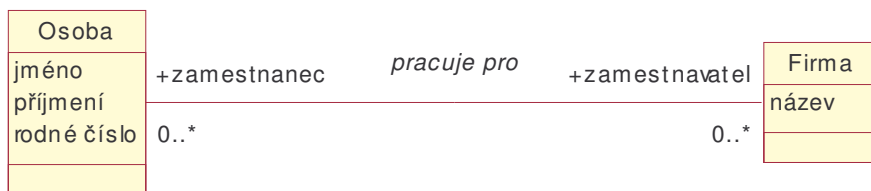
Tabulka 3.1: Možnosti vyjádření násobnosti v UML

Na obrázku 3.9 je znázorněna násobnost u asociace mezi třídou *Osoba* a *Auto*. Tento vztah lze interpretovat: „Osoba si může půjčit jedno a více aut, jedno auto má půjčené žádná nebo jedna osoba“.

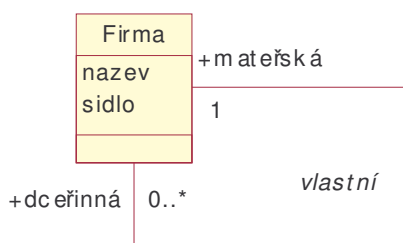
Role

Role představuje jeden konec asociace. Binární asociace má dvě role. Pokud není explicitně uveden název role, chápe se jako role název třídy. Definování role představuje pojmenování průchodu asociací od jednoho objektu k druhému. Na obrázku 3.10 jsou uvedeny role u asociace *Osoba*- *Firma*. *Osoba* vystupuje v roli zaměstnance a firma vystupuje v roli zaměstnavatele. Použití rolí není povinné, ale je mnohdy lepší a srozumitelnější uvést

v modelu role než pojmenovávat asociaci. Na druhé straně je použití rolí téměř nezbytné u asociace mezi objekty téže třídy viz obrázek 3.11.



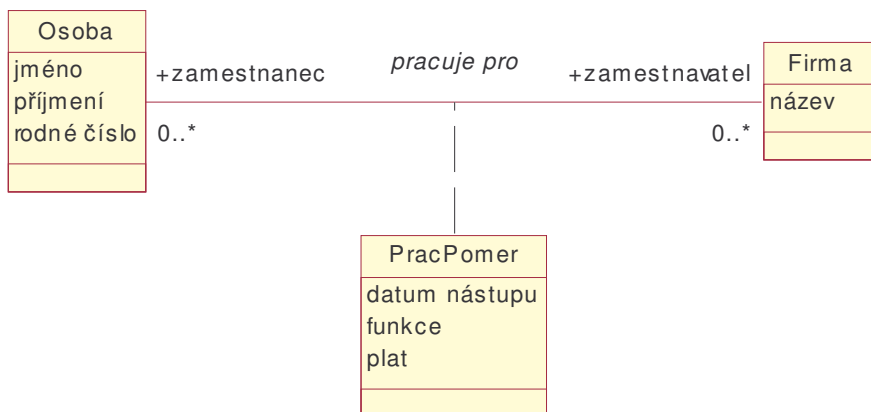
Obrázek 3.10: Role v asociaci "pracuje pro"



Obrázek 3.11: Role u asociace na sebe sama

Asociativní třída

V určitých případech nastanou v analýze situace, kdy sama vazba mezi objekty by měla nést nějakou informaci. Vezměme asociaci uvedenou na obrázku 3.10. Pokud osoba pracuje pro více firem (má více pracovních poměrů), potom je třeba u každého pracovního poměru sledovat od kdy probíhá, jakou funkci v tomto pracovním poměru osoba vykonává, jaký plat za to pobírá. Tyto informace nejsou ani atributem třídy *Osoba* ani atributem třídy *Firma*, ale vztahují se právě k asociaci mezi osobou a firmou, která představuje daný pracovní poměr. Můžeme proto navrhnout asociativní třídu *PracPomer*, která tyto informace ponese. V diagramu tříd ji můžeme znázornit tak, jak je uvedeno na obrázku 3.12.

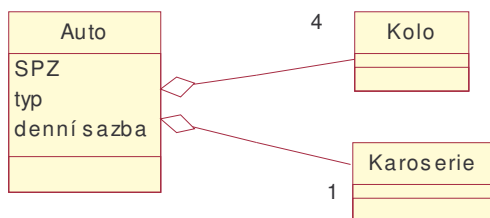


Obrázek 3.12: Asociativní třída

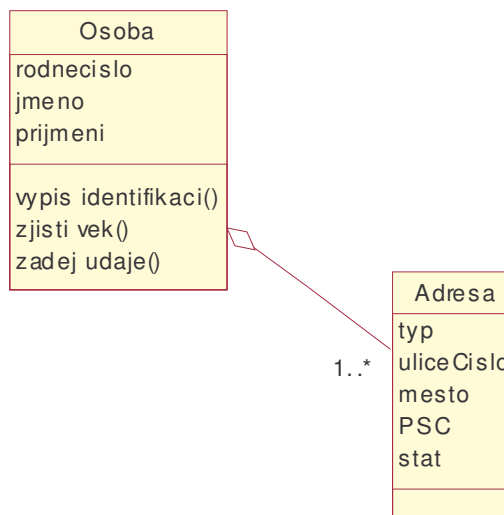
Agregace

Pokud vnější objekt obsahuje vnitřní objekt jako svou část, tj. vnitřní objekty skládají vnější objekt, potom hovoříme o agregaci. Jedná se tedy o vztah skládání celku z částí. Celek přitom odpovídá za vytvoření a zrušení svých částí - objektů. Části mohou nebo nemohou existovat

i mimo celek. Agregaci definujeme jako vztah celku k jedné části. Celku s více částmi odpovídá více agregací. Agregace je vlastně druhem asociace, můžeme tedy definovat násobnost části v celku. Na obrázku 3.13 je znázorněna agregace vyjadřující, že objekt *Auto* obsahuje 4 objekty *Kolo* a 1 objekt *Karoserie*. Na obrázku 3.14 je příklad vyjadřující, že objekt *Osoba* obsahuje jeden nebo více objektů *Adresa* (trvalé bydliště, přechodné bydliště apod.).

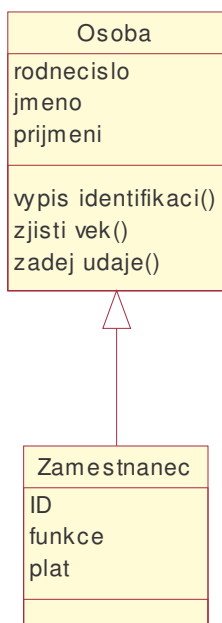


Obrázek 3.13: Agregace ve třídě Auto



Obrázek 3.14: Agregace ve třídě Osoba

Generalizace - specializace



Generalizace-specializace je druh abstrakce, který umožňuje sdílet stejné vlastnosti a chování mezi třídami a přitom umožňuje zachovat rozdíl mezi nimi. Vztah generalizace - specializace:

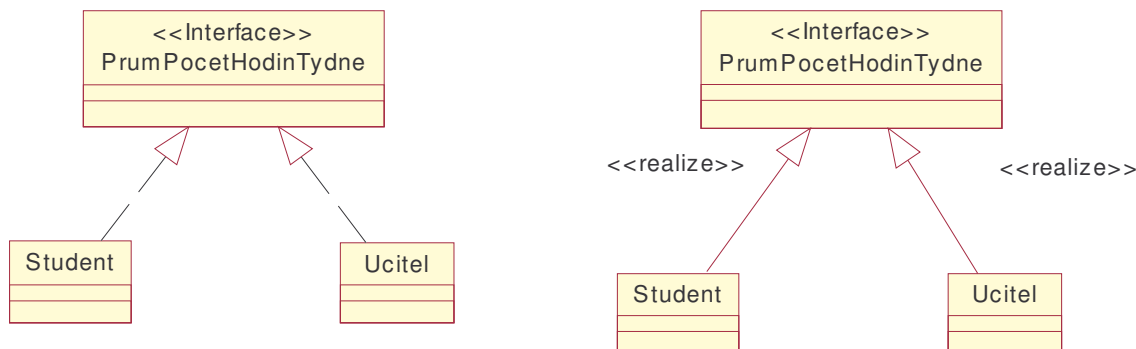
- je vztah mezi třídou a speciálním případem této třídy,
- je koncept užitečný jak pro konceptuální modelování, tak pro implementaci,
- usnadňuje modelování tím, že rozlišuje, co je stejné a čím se třídy liší,
- implementuje se pomocí dědičnosti, případně delegace (viz 3.2.4.2)

Obrázek 3.15: Příklad vztahu generalizace - specializace

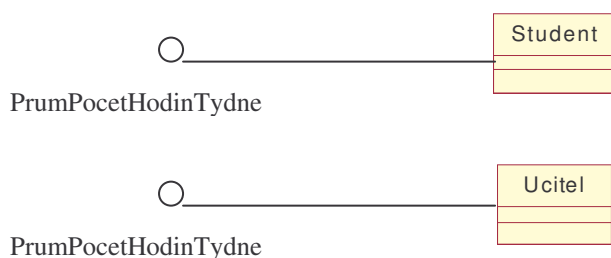
Rozhraní (interface)

Rozhraní představuje způsob specifikace používání třídy. Přínosem objektově orientovaného přístupu je oddělení rozhraní od implementace. Některé objektově orientované jazyky umožňují explicitně definovat rozhraní jako samostatnou entitu. Platí to pro jazyk Java (viz

6.6) i Object Pascal. V kontextu programovacích jazyků říkáme, že třída implementuje rozhraní (*implements*). V UML říkáme, že třída realizuje (*realize*) rozhraní. V diagramu tříd můžeme znázornit rozhraní několika způsoby, které jsou uvedeny na obrázku 3.16 a 3.17. Na nich jsou zachyceny třídy *Student* a *Ucitel*. Každá z těchto tříd realizuje rozhraní *Rozvrh*, které poskytuje rozvrh aktuálních kurzů, které student studuje (učitel vyučuje).



Obrázek 3.16: Příklady zachycení rozhraní v UML



Obrázek 3.17: Jiná možnost zachycení rozhraní v UML

Sekvenční diagram

Sekvenční diagram reprezentuje zasílání zpráv mezi objekty v rámci systému. Ve 2. kapitole jsme uvedli, že základní charakteristikou objektu je jeho schopnost přijmout zprávu a reagovat na ni. V rámci této reakce může proběhnout nejen určité zpracování, ale objekt může poslat zprávu dalšímu objektu. Dalším posíláním zpráv objektům vzniká **sekvence** posílání zpráv. Zápis takovéto sekvence je právě předmětem sekvenčního diagramu. Při popisu chování aplikace pomocí sekvenčního diagramu se vychází z diagramu užití (Use Case Diagram), kdy každý typ užití je popsán nějakou sekvencí zasílání zpráv.

Sekvenční diagram obsahuje následující prvky:

- **objekt** (může být popsáno z jaké je třídy),
- **zpráva** mezi objekty,
- **činnost** objektu a její popis.

Objekt

Objekt se v sekvenčním diagramu označuje pomocí svislé čáry. Na vrcholu této čáry se označí název objektu.

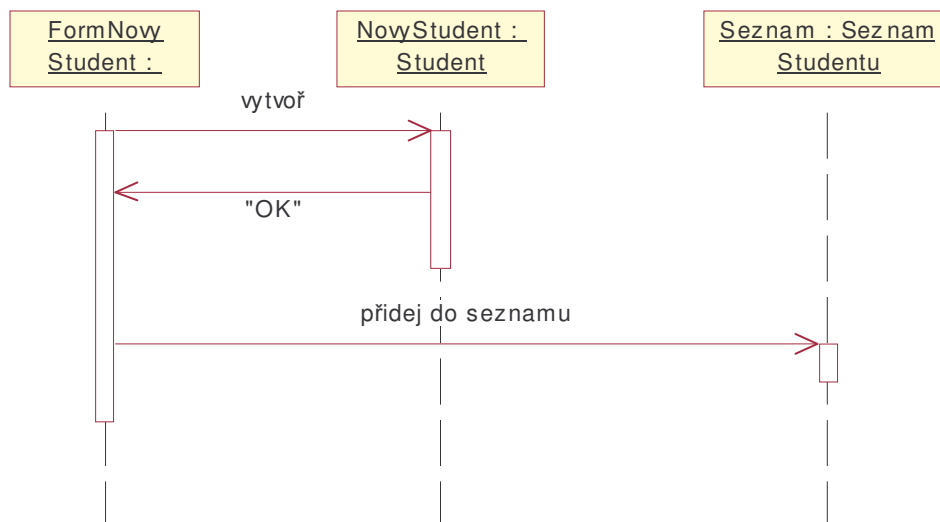
Zpráva

Zpráva se v sekvenčním diagramu znázorňuje pomocí šipky od objektu, který zprávu posílá, k objektu, který zprávu přijímá. Zprávy můžeme dělit na synchronní a asynchronní.

Synchronní zpráva je taková, kdy vysílající objekt vyčkává na zpracování této zprávy a pokračuje v další činnosti až po ukončení zpracování přijímajícího objektu. Pokud vysílající objekt pošle druhému objektu zprávu a nečeká na její zpracování, potom se jedná o

asynchronní zprávu. V tom případě přijímající objekt zpracovává zprávu a souběžně běží proces v objektu, který zprávu vyslal. Objekt, který zprávu přijal, může například výsledek své činnosti poslat zpět objektu jako zprávu.

V sekvenčním diagramu se znázorňuje **sekvence** zpráv daného typu užití. Vzniká tak jeden scénář zpracování.



Obrázek 3.18: Sekvenční diagram

Na obrázku 3.18 je zachycen scénář typu užití "Přidání nového studenta do evidence studentů". Interakci zahajuje objekt *FormNovyStudent*, který posílá požadavek na vytvoření objektu *NovyStudent* třídy *Student*. Parametrem této zprávy jsou atributy objektu třídy *Student*, které byly zadány v polích formuláře. Formulář potom zasílá zprávu do objektu *Seznam* třídy *SeznamStudentu* a požaduje přidání nového studenta, přičemž parametrem je instance právě naplněného studenta (na obrázku není vstupní parametr objekt znázorněn). Pokud má objekt schopnost přijímat určitou zprávu, znamená to, že na tuto zprávu objekt spustí některou z metod. Pokud u objektů sekvenčního diagramu máme určeny třídy, ke kterým objekty patří, potom můžeme na základě zpráv, které objekt přijímá, určovat jaké metody musí objekt mít. Znamená to, že u objektů, u kterých končí zpráva, musí existovat metoda, která je přiřazena k této zprávě. Většinou je dobré používat stejný název jak pro metodu, tak pro zprávu. Zpráva zasílaná objektu má své parametry jak vstupní, tak výstupní. Tyto parametry se poté stávají vstupními a výstupními parametry dané metody. Mohou se vyznačit do diagramu, nebo se popisují zvlášť. Sekvenční diagramy pomáhají odhalovat chování aplikace (metody objektů), ale také kompetence objektů za jednotlivé činnosti (kdo co bude konat sám a k čemu použije jiný objekt).

Diagram spolupráce (Collaboration diagram)

Diagram spolupráce zachycuje interakce mezi objekty v systému. V tomto ohledu je obdobný sekvenčnímu diagramu.

Diagram objektů (Object diagram)

Diagram objektů znázorňuje vztahy objektů v systému. Je odvozen z diagramu tříd a představuje jeho instancionalizaci. Používá se spíše jako příklad a ověření správnosti diagramu tříd.

Stavový diagram (State Transition diagram)

Stavový diagram znázorňuje jednotlivé stavy, kterými prochází určitý objekt v závislosti na událostech.

Diagram aktivit (Activity diagram)

Diagram aktivit umožňuje modelovat obchodní procesy.

Diagram komponent (Component diagram)

Diagram komponent ukazuje vzájemné závislosti softwarových komponent.

Diagram balíčků (Package diagram)

Diagram balíčků ukazuje rozdělení systému do modulů.

Diagram nasazení (Deployment diagram)

Diagram nasazení znázorňuje konfiguraci jednotlivých prvků systému při instalaci a běhu.

Podrobnější popis jednotlivých diagramů naleznete v [WWWDrbal], [OOMT], [MyslUML], [ZaklUML] a dalších zdrojích.

Fáze vývoje programového systému

V tomto odstavci je uveden postup vývoje programového systému s následujícími charakteristikami:

- programový systém menšího rozsahu,
- důraz kladen na objektově orientovaný vývoj,
- nezávislost na konkrétní metodice, snaha o zobecnění.

Postup můžeme rozložit do následujících fází:

1. Příprava plánu projektu
2. Specifikace požadavků
3. Analýza
4. Návrh
5. Implementace
6. Testování
7. Nasazení
8. Údržba

Příprava plánu projektu

V této fázi je třeba připravit plán, podle kterého bude projekt probíhat. Problematika řízení projektů je zpracována v řadě publikací a na VŠE je náplní samostatného předmětu "Projektování a řízení IS" a předmětu "Řízení projektů". Pro naše účely postačí, když si uvědomíme nutnost této fáze a základní činnosti, které je třeba v této fázi provést:

- určení vedoucího projektu,
- určení členů týmu,
- definování termínů dokončení jednotlivých fází a výstupů z těchto fází,
- Přechody z jedné fáze do druhé jsou charakterizovány tzv. milníky (milestone), které specifikují podmínky, za kterých lze fázi ukončit.
- definování metrik kvality,
- definování ekonomických charakteristik (nákladů, přínosů).

Závěrem této fáze je rozhodnutí, zda je projekt za daných požadavků, dostupných technologií, zdrojů a rozpočtu možné realizovat.

Specifikace požadavků

Dříve než zahájíme analýzu, návrh a kódování, je třeba identifikovat požadavky, které má program splňovat. Je třeba zjistit, jaké funkce bude zákazník, respektive uživatel, s programem realizovat, za jakých podmínek, v jakém prostředí apod. Je třeba sepsat požadavky systematicky a přehledně. Požadavkem se rozumí požadavek na funkcionalitu budoucí aplikace. Musí být vyjádřen pouze v pojmech problémové oblasti, aby jim uživatel dobře rozuměl. V mnoha případech je sám uživatel spoluautorem tohoto dokumentu. Výsledkem by měla být specifikace požadavků, která může mít například strukturu uvedenou v tabulce 3.2. Rozsah, podrobnost a forma takového dokumentu bývá zpravidla předmětem diskusí. Některé firmy dávají přednost textovým dokumentům, jiné diagramům.

Dokument Specifikace požadavků	
Identifikace aplikace	charakteristika aplikace a pro koho je určena
Funkce aplikace	hlavní funkce, které má aplikace mít a jejich stručný popis
Popis uživatelů	hlavní skupiny uživatelů, kteří budou program používat, popisuje, jak jej budou používat včetně jakých funkcí
Prostředí	specifikace operačního systému, hardware apod.
Požadavky na UI	specifický vzhled a ovládání, hardwarová omezení
SW interface	definuje interface na jiné systémy a vlastní interface
Omezení	veškerá omezení pokud jde o technologie, programovací jazyky apod.
Předpoklady a závislosti	například komunikace s externím SW
Výkonnost	požadavky na dobu odezvy, počet uživatelů apod.
Obchodní procesy	definuje vztah funkcí programu k procesům v organizaci

Tabulka 3.2: Specifikace požadavků

Analýza

Abychom mohli navrhnout programový systém, je třeba pochopit danou problémovou oblast. To je předmětem fáze analýzy. Této fáze se aktivně účastní uživatelé systému. Základem této fáze při objektově orientované analýze je tvorba diagramů užití (Use Case Diagram dle UML). Tyto diagramy jsou podrobně popsány v 3.1.1. Na základě analýzy jednotlivých typů užití se vytváří analytický návrh struktury tříd (diagram tříd viz. 3.1.2). Tento pohled na problémovou oblast je statický, a proto bývá doplňován scénáři pro jednotlivé typy užití, které se vyjadřují pomocí sekvenčních diagramů (viz. 3.1.8)

Fáze analýzy zahrnuje zejména následující kroky:

- definování typů užití a vytvoření diagramu užití,
- vytvoření analytického diagramu tříd,
- vytvoření sekvenčních diagramů pro každý typ užití,
- definování testů.

Již v této fázi je dobré navrhnout testy, na kterých má být vyvíjený SW testován. Návrh testů v ranných fázích vývoje má několik předností. Jednak svědčí o snaze vyvinout kvalitní SW, jednak vede vývojáře k vytvoření spustitelné verze. Pokud je realizován iterativní vývoj, potom výsledkem každé iterace by měla být spustitelná verze programu, která musí být otestována. Testy mají prověřit jak vlastní funkčnost aplikace, tak i uživatelské rozhraní. Včasným testováním uživatelského rozhraní samotnými uživateli, lze předejít mnohým zklamáním na konci projektu.

Návrh struktury tříd

Cílem fáze analýzy je definovat strukturu tříd objektů a u každé třídy definovat atributy a metody. Nalezení správných tříd, určení relevantních atributů a metod, definování vztahů mezi třídami, to vše je dosti náročný proces, který je do značné míry ovlivněn zkušenostmi analytika. Přesné postupy, jak se dobrat výsledné struktury tříd, metodiky většinou neuvádějí. Poskytují jen návody či doporučení. Většinou se doporučuje provádět analýzu a návrh iterativně, načrtnout základní strukturu tříd, tu upřesňovat a rozvíjet. Vhodná je pro tuto fázi práce v týmu, kdy členové týmu společně analyzují danou oblast a navrhnou třídy. Strukturu tříd je třeba zdokumentovat, nejlépe pomocí CASE nástroje, který podporuje standardní notaci například UML.

Výhoda použití CASE nástroje spočívá v možnosti snadných úprav modelu:

- v jednotlivých iteracích fáze analýzy a návrhu,
- při přechodu ke kódování, kdy přistupuje řada implementačních charakteristik,
- po nasazení systému při jeho údržbě a rozvoji.

Východiskem pro návrh tříd a vztahů mezi nimi je specifikace požadavků a diagramy užití. Rozborem těchto dokumentů získáme kandidáty na třídy. Můžeme postupovat následujícím způsobem, který je po zjednodušení převzat z [ZakUML]:

1. Výběr kandidátů

Pročítáme specifikaci požadavků a typy užití a vybíráme všechna podstatná jména, která si vypíšeme.

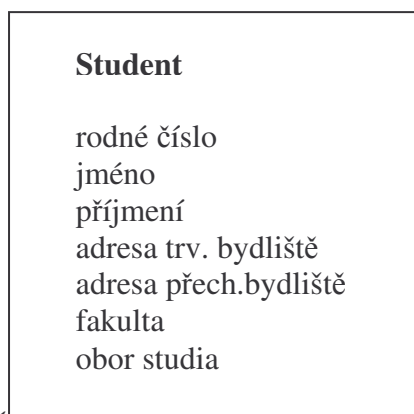
2. Rozlišení kandidátů na třídy a kandidátů na atributy

Mezi vybranými podstatnými jmény se snažíme odlišit kandidáty na třídy (entita, která může sama o sobě existovat - například *Student*) a kandidáty na atributy určité třídy - podstatné jméno, které samostatně neexistuje, ale je vlastností nějaké jiné entity - například *příjmení studenta*.

3. Záznam tříd

Výsledek tohoto procesu můžeme zaznamenat například na karty. Pro každou třídu připravíme kartu, na níž uvedeme jméno třídy a atributy nebo můžeme zapsat kandidáta na třídu následujícím způsobem:

Student = (rodné číslo, jméno, příjmení, adresa trvalého bydliště,
adresa přechodného bydliště, fakulta, obor studia..atd.)



4. Prověření, zda nemá být atribut třídou

Volba, zda má být určitý prvek třídou a nebo atributem může být někdy obtížná, v každém případě nemusí být konečná. Prvek navrhne například jako atribut, ale ukáže-li se později, že je to vhodnější, můžeme z něj udělat třídu. Příkladem může být třeba *rodné číslo*. Pokud budeme chtít provádět speciální kontroly rodného čísla,

poskytovat jej v různých formátech (s lomítkem, bez lomítka apod.), vypočítávat věk, bude možná vhodnější udělat pro něj třídu a do objektu třídy *Student* vložit instanci této třídy.

5. Dvě třídy nebo dvě instance téže třídy

Je třeba také odlišit, kdy dva prvky představují jen dvě instance téže třídy a kdy je třeba skutečně vytvořit dvě třídy. Můžeme si to ukázat na příkladě adresy. V našem výčtu atributů třídy *Student* se objevily atributy:

adresa trvalého bydliště,

adresa přechodného bydliště

Po hlubší analýze dojdeme k závěru, že pro ně bude vhodné vytvořit třídu. Tato třída bude mít několik atributů (ulice, číslo, město, psč, stát). Všechny tyto atributy jsou jak u trvalého bydliště, tak u přechodného bydliště. Také metody, které třída *Adresa* bude poskytovat, se nijak neliší pro adresu trvalého a adresu přechodného bydliště. Proto vytvoříme třídu jednu. Do třídy *Student* potom vložíme dvě instance téže třídy *Adresa*.

6. Kandidáti na vztah generalizace-specializace

Pokud se ve více třídách opakují tytéž informace, můžeme uvažovat o vytvoření obecné třídy. Tuto třídu vybavíme atributy a metodami, které jsou společné a od ní odvodíme speciální třídy. Pokud v našem případě kromě studentů budeme evidovat i učitele, ukáže se, že i ve třídě *Učitel* budou atributy jako *rodné číslo*, *jméno*, *příjmení*, *adresa trvalého bydliště*, *adresa přechodného bydliště*. Potom bude možná vhodné vytvořit třídu *Osoba* s atributy *rodné číslo*, *jméno*, *příjmení*, *adresa trvalého bydliště*, *adresa přechodného bydliště*. Od této třídy potom odvodíme speciální třídy *Student* a *Učitel*.

7. Vyhledání odvozených tříd typu seznam

Zároveň hledáme i odvozené třídy (kolekce) - seznam studentů, seznam předmětů.

U každé třídy je třeba si všimnout charakteristik uvedených v tabulce 3.3

Charakteristika	Poznámka
Jméno třídy	Je třeba dát třídě smysluplné jméno, které jasně vyjadřuje informace, které třída představuje Pokud změníte obsah třídy, možná bude třeba změnit i její jméno, aby bylo konzistentní s obsahem
Vztah k jiným třídám	Je třeba zkoumat, zda navrhovaná třída není speciálním případem nějaké již definované třídy a nebo naopak, jestli několik tříd neobsahuje společné vlastnosti, které by bylo možné přiřadit nějaké obecnější třídě. Podtřídy nemá smysl vytvářet za každou cenu, ale jen v těch případech, kdy je to smysluplné.
Funkcionalita třídy	Jde o to určit co navrhovaná třída: 1. zná (jaké má vlastnosti) 2. dělá (jaké má metody)
Spolupráce	Zajímáme se o interakci s ostatními třídami. Pokud dvě třídy příliš mnoho spolupracují, bude možná vhodnější vytvořit z nich jedinou třídu.

Tabulka 3.3: Charakteristiky uvažované při návrhu tříd

Navržená struktura tříd musí být porovnána s diagramy užití. Nastupují otázky typu:

Pokryjí navržené třídy všechny definované typy užití?

Poskytuje navržená struktura tříd prostor pro další rozšíření a přidání funkcí?

Při návrhu tříd se snažíme o jednoduchost. Přínos objektově orientovaného přístupu spočívá v možnosti rozdělit problém na malé části, které mohou stát samostatně a přitom reprezentují ucelenou funkčnost. Je lepší navrhnout více malých objektů, které reprezentují jasně a dobře

jednu věc, než jeden velký nepochopitelný a složitý objekt. Navíc malé, jasné objekty, lze snadno znovupoužít.

Návrh

Fáze návrhu bývá někdy rozdělována na dvě fáze :

- systémový návrh,
- objektový návrh.

V rámci systémového návrhu je třeba zvolit implementační prostředí a architekturu aplikace. Fáze objektového návrhu definuje třídy a vztahy mezi třídami, algoritmy metod a uživatelské rozhraní.

Volba implementačního prostředí, architektury a vytvoření či využití aplikačního rámce

V tomto kroku je třeba učinit tzv. implementační rozhodnutí, tj. definovat:

- platformu - operační systém,
- rozhodnout o technologické architektuře aplikace:
 - jednouživatelská (desktop) aplikace,
 - aplikace typu klient/server s tlustým klientem (např. pod Windows),
 - internetová aplikace,
 - vícevrstvá aplikace snadno přenositelná z tlustého na tenkého klienta
- vybrat programovací jazyk a vývojové prostředí,
- vytvořit a nebo použít aplikační rámec (application framework).

V závislosti na technologické architektuře aplikace je určeno uživatelské rozhraní:

- klasické aplikace - textové UI, grafické UI
- rozhraní akceptované prohlížečem.

Při vytváření uživatelského rozhraní aplikace se zpravidla využívá celá řada již hotových objektů, jak jednotlivých prvků jako jsou tlačítka, editační pole apod., tak i formulářů či celých šablon aplikací. Tyto prvky tvoří tzv. aplikační rámec. Pokud jsme vybrali programovací jazyk a vývojové prostředí, které již tyto objekty má vytvořeny, stačí je jen použít. Pokud ne, je třeba tyto prvky nejdříve naprogramovat. Takovým aplikačním rámcem je třeba :

- knihovna VCL v Delphi,
- knihovna Swing pro Javu,
- ActiveX prvky , CSS pro WWW stránky.

Aplikační rámec bývá dodáván jako knihovna tříd (MFC pro C++ nebo Swing pro Javu, kde je součástí základního SDK) a nebo je součástí vizuálních vývojových nástrojů jako jsou Delphi, JBuilder, NetBeans a další. S pomocí těchto nástrojů lze velmi rychle vytvořit prototypy aplikací a otestovat uživatelské rozhraní přímo koncovými uživateli.

Objektový návrh

Ve fázi objektového návrhu se přidávají další implementační třídy, optimalizují se algoritmy a datové struktury. V této fázi pracujeme s diagramy vytvořenými ve fázi analýzy a dále je doplňujeme o prvky, které jsou již implementačně závislé. Ve fázi návrhu přecházíme z aplikační oblasti do počítačové oblasti. Objekty identifikované při analýze tvoří kostru návrhu, ale návrhář musí volit mezi různými způsoby, jak je implementovat. Hlediskem přitom je minimalizace doby provádění, paměti a další metriky. Operace identifikované při

analýze je třeba vyjádřit ve formě algoritmů. Složité operace musí být dekomponovány na jednodušší. Je třeba zavést nové objekty pro uchování mezivýsledků.

Činnosti prováděné ve fázi objektového návrhu můžeme rozdělit do těchto kroků:

1. Práce s různými typy diagramů, kontroly konzistence a převod do algoritmů

Navzájem porovnáváme jednotlivé diagramy a snažíme se nalézt rozpory.

2. Návrh algoritmů pro implementaci operací

Každá operace musí být formulována jako algoritmus. Je třeba:

- vybrat nejvýhodnější algoritmus,
- zvolit pro daný algoritmus vyhovující datové struktury,
- definovat nové interní třídy a operace, je-li třeba,
- přiřadit odpovědnost za operace jednotlivým třídám.

3. Řešení přístupu k datům

Je třeba vyřešit uložení dat. Zvolit buď uložení v souborech a nebo použít databáze. V tom případě je třeba vytvořit datový model, definovat strukturu databáze a vytvořit vrstvu mapující data objektů do databáze.

4. Implementace řízení interakce s jinými objekty

Vztahy mezi třídami, které byly v diagramu tříd znázorněny jako asociace, je třeba implementovat. Musí pro ně existovat metody (je třeba rozhodnout v kompetenci které třídy budou), objekt se kterým se komunikuje musí být předán jako parametr dané metodě.

5. Prověření struktury objektů pro zavedení dědičnosti

Přeskupení tříd

V některých případech je stejná operace definována v různých třídách a mohla by být zděděna. Stačí například trochu pozměnit operaci nebo návrh tříd a je možné využít operaci z třídy předka bez jejího předefinování.

Abstraktní třídy

Je třeba projít třídy a hledat případné abstraktní třídy, které budou implementovat společnou základní funkčnost.

Rozhraní

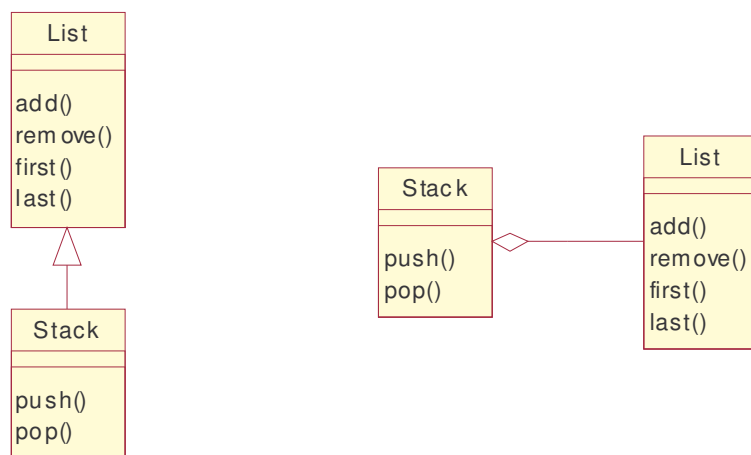
Pokud vývojové prostředí podporuje rozhraní, je třeba prověřit společné operace a případně je vyčlenit do rozhraní.

Využití delegace pro sdílení implementace

Dědičnost je mechanismus pro implementaci generalizace - specializace v případech, kdy chování nadtřídy je sdíleno ve všech podtřídách. Použití dědičnosti je oprávněné v těch případech kdy podtřída je specializací nadtřídy i v **analytickém smyslu**. Operace podtřídy přepisují operace nadtřídy a případně přidávají něco navíc. Pokud *třída B* dědí po *třídě A*, předpokládáme, že každá instance *třídy B* je i instancí *třídy A*. Někdy ovšem programátoři používají dědičnost jako implementační techniku bez záměru garantovat stejné chování tříd v dědičné hierarchii. Tak se stává, že nová třída dědí z třídy předka určité chování, ale jinak jsou třídy úplně rozdílné. To může vést k problémům v jiných operacích dané třídy, které mohou působit nechtěné chování. Jako příklad takové **dědičnosti implementace**, kterému je lépe se vyhnout, můžeme uvést implementaci tříd *List* a *Stack*. Třída *List* reprezentuje datovou strukturu typu spojového seznamu, do které je možné přidávat prvky v kterémkoli místě, mazat jakýkoli prvek. Naproti tomu *Stack* reprezentuje zásobník, u kterého je přesně stanoven způsob přidávání a odebírání prvků - prvky se přidávají a odebírají vždy na jednom konci - vrcholu zásobníku. Pokud máme implementovat třídu *Stack* a máme již třídu *List*, možná bychom ji chtěli využít a navrhli bychom třídu *Stack* jako potomka třídy *List*. V tom případě ovšem zdědíme do třídy *Stack* i operace pro přidání

prvku, vymazání prvku z kteréhokoli místa. Potom se třída *Stack* nebude chovat tak, jak bychom očekávali.

V případech, kdy chceme použít dědičnost jako implementační techniku, dosáhneme stejného cíle bezpečnějším způsobem pomocí vloženého objektu. Tento způsob se nazývá delegace, neboť objekt pro určitou činnost vyvolá metodu jiného objektu, který je jeho součástí a nebo s kterým je spojen. Toto řešení je ilustrováno na obrázku 3.18. [Merunka]



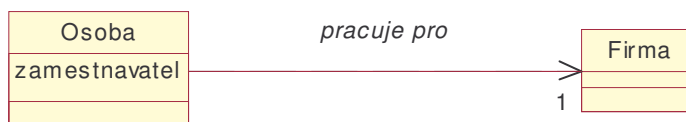
Obrázek 3.18: Nesprávné použití dědičnosti, řešení pomocí vloženého objektu

6. Návrh asociací

Asociace jsou konceptuální entity ve fázi analýzy. Při návrhu je třeba určit strategii, jak je implementovat. Tato strategie může být buď univerzální, použitá pro všechny asociace v modelu, a nebo se pro různé asociace použijí různé techniky. V analytické fázi většinou předpokládáme, že asociace jsou obousměrné. Pokud je však asociace jednosměrná, může být její implementace jednodušší. Buďte ovšem opatrní, protože se tato situace může změnit. Například přidání nové operace si vyžádá vytvořit asociaci obousměrnou. Při prototypování používáme vždy obousměrnou asociaci, protože tak můžeme snadno přidat nové chování.

Jednosměrné asociace

mohou být implementovány jako ukazatel - atribut, který obsahuje referenci na objekt. Pokud je násobnost asociace 1, jde o jednoduchý ukazatel (obrázek 3.19). Je-li násobnost N, jde o množinu ukazatelů. Jsou-li objekty v asociaci uspořádané, je lepší použít seznam.



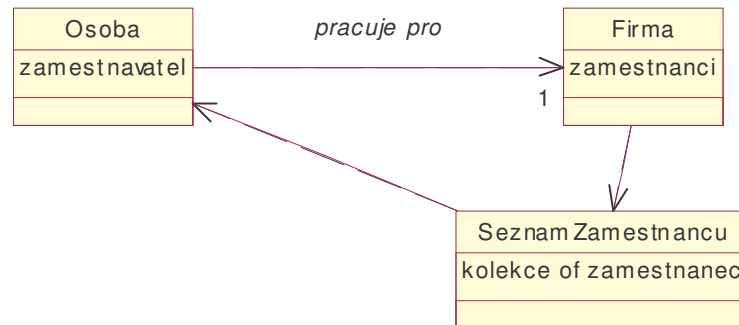
Obrázek 3.19: Implementace jednosměrné asociace pomocí ukazatele

Obousměrné asociace

Mnoho asociací se prochází oběma směry, i když ne se stejnou frekvencí.

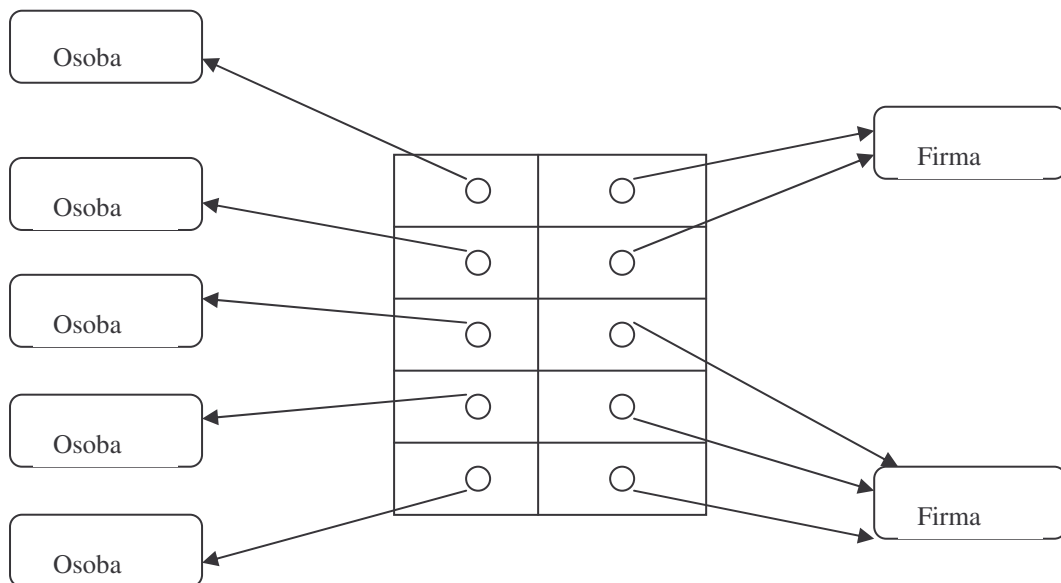
Obousměrnou asociaci můžeme realizovat jedním z následujících 3 způsobů:

- [1] asociaci implementujeme jako atribut (ukazatel) jen v jednom směru a při obráceném průchodu provádíme vyhledání - tento způsob je vhodný jen pro případy, kdy zpětný průchod není častý
- [2] implementujeme atributy (ukazatele) v obou směrech (viz obrázek 3.20), tento způsob dovoluje rychlý přístup, ale při změně jednoho ukazatele je třeba změnit i ukazatel druhý



Obrázek 3.20: Implementace obousměrné asociace

- [3] implementujeme asociaci jako zvláštní objekt - asociační třídy (obrázek 3.21)

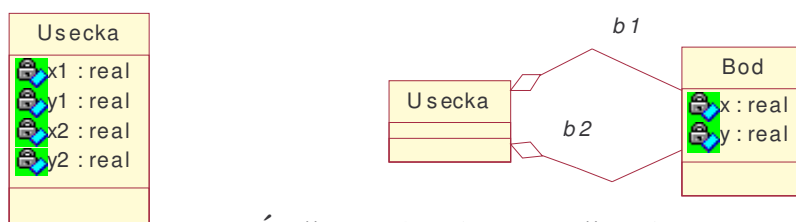


Obrázek 3.21: Implementace obousměrné asociace asociační třídou

7. rozhodnutí o reprezentaci objektů

Implementace objektů je někdy jasná, někdy je třeba se rozhodnout pro některou z alternativ. Atribut můžeme reprezentovat různými datovými typy (například číslo pracovníka jako *integer* nebo *string*). Pro určitý atribut můžeme definovat také samostatnou třídu. Na obrázku 3.22 je znázorněna třída *Usecka*, definovaná 4 souřadnicemi. Jinou možností určení úsečky je definovat třídu *Bod* a úsečku potom složit ze dvou instancí třídy *Bod* tak, jak ukazuje obrázek 3.23.

Obrázek 3.23: Úsečka složená ze dvou bodů



Obrázek

3.22: Úsečka se 4 atributy souřadnic

8. Doplnění vrstvy uživatelského rozhraní

Diagramy vytvořené ve fázi analýzy je třeba doplnit o objekty uživatelského rozhraní, zejména o třídy formulářů.

9. Seskupení tříd do modulů

Ve fázi návrhu bychom měli rozhodnout o rozdělení aplikace do komponent a vytvořit diagram komponent.

Implementace

Na začátku implementace je třeba určit pořadí implementace jednotlivých tříd. Probíhá-li fáze implementace v iteracích, je třeba určit, které třídy implementovat v jednotlivých iteracích. K tomu přistupuje ještě problém týmové práce. Jak rozdělit třídy mezi jednotlivé členy týmu? Jak se vypořádat se vzájemnými závislostmi tříd? Je dobré dodržovat pravidlo, že každá iterace reprezentuje verzi programu, kterou lze testovat vůči typům užití.

Doporučení pro programování

Jak si mohl vyzkoušet každý lyžař, šachista či kuchař, existuje velký rozdíl mezi tím, že někdo něco zná, a někdo to umí. S psaním objektově orientovaných programů je to podobné. Nestačí znát pouze principy, ale je třeba mít dovednosti, jak navrhnout třídy, jejich vazby, jak je spojit do programu, který dělá to, co po něm požadujeme. Zkušený programátor dodržuje principy, které umožňují psát čitelné a rozšiřitelné programy. Dobrý programovací styl je důležitý ve všech metodách programování, ale v objektově orientovaném návrhu nabývá na větší důležitosti, protože cílem objektově orientovaného přístupu je produkovat **znovupoužitelné a rozšiřitelné programy**. V objektově orientovaném programování platí obecné programovací zásady a doporučení, ale přidávají se i nové zásady, které jsou spojeny s novými rysy OOP jako je dědičnost apod. Uvedeme nejdůležitější zásady v rámci následujících kategorií:

- znovupoužitelnost,
- rozšiřitelnost,
- robustnost,
- programování ve velkém.

Pravidla pro zajištění znovupoužitelnosti:

1. Realizujte koherentní metody, tj. metody, které provádějí jednu funkci a nebo skupinu těsně svázaných funkcí.
2. Realizujte malé metody. Je-li metoda velká, rozdělte ji na menší, které budou lépe znovupoužitelné.
3. Realizujte konzistentní metody, to znamená, že stejné metody by měly mít stejná jména, podmínky, pořadí argumentů, návratové hodnoty, chybové stavy.
4. Oddělte řídicí a implementační metody. Řídicí metody přijímají rozhodnutí, drží celkový kontext, volají implementační metody, kontrolují stav včetně chybového. Implementační metody provádějí specifické operace bez rozhodování proč. Provádějí výpočet pro plně definované parametry.

5. Realizujte metody plně pokrývající danou oblast. Pokud se vstupní podmínky mohou vyskytovat v různých kombinacích, obslužte všechny možnosti.
6. Rozšiřte metodu, jak je to jen možné. Snažte se zobecnit parametry, vstupní podmínky, omezení, předpoklady, za jakých metoda pracuje, a kontext, v jakém pracuje.
7. Realizujte smysluplné akce pro nulové a extrémní hodnoty. Často je možné udělat obecnější metodu jen s malým navýšením kódu.
8. Vyhněte se globálním informacím. Minimalizujte vnější vazby, místo toho použijte parametry.
9. Vyhněte se módům. Funkce, jejichž chování velmi závisí na kontextu, jsou těžko znovupoužitelné.
10. Využívejte dědičnost. Společný kód implementujte jako metodu předka a u potomků ji předefinujte.
11. Využívejte vkládání objektů.

Pravidla pro zajištění rozšiřitelnosti:

1. Dodržujte zapouzdření tříd.
2. Ukryjte datové struktury.
3. Vyhněte se procházení více linků. Metoda musí mít jen omezenou znalost objektového modelu.
4. Rozlišujte **public** a **private** operace.

Pravidla pro zajištění robustnosti:

1. Program se musí bránit chybám - kontrolujte vstup.
2. Optimalizujte program až po té, co běhá - optimalizujte jen často volané funkce.
3. Kontrolujte parametry metod
4. Pokud je to možné, používejte pro uchování dat dynamické proměnné, které nevyžadují předem definované meze.
5. Vybavte program proměnnými pro ladění a monitorování běhu.

Pravidla při programování ve velkém - na velkých projektech, v týmu:

1. Nepředbíhejte s programováním - nejprve je třeba kompletně dokončit návrh.
2. Realizujte pochopitelné metody.
3. Realizujte čitelné metody.
4. Používejte stejná jména v kódu jako v objektovém modelu.
5. Volte správná jména - ujistěte se, že přesně vyjadřují operace, třídy, atributy, používejte dané konvence.
6. Seskupte třídy do modulů.
7. Dokumentujte třídy a metody.

Závěrečné testování

I když testy provádíme po každé iteraci fáze implementace, je třeba provést i závěrečné testování, které má odhalit chyby před nasazením systému.

Nasazení a údržba

Problematika zavádění programových systémů a jejich údržby přesahuje rámec těchto skript.

Příklad Cestovní kancelář

Jednotlivé fáze vývoje programového systému, které byly uvedeny v 3.2 budeme demonstrovat na příkladě řešení aplikace pro cestovní kancelář. Zároveň na dané problémové oblasti ukážeme i příklady jednotlivých diagramů UML. Fázi plánu projektu popisovat nebudeme.

Příklad Cestovní kancelář - specifikace požadavků

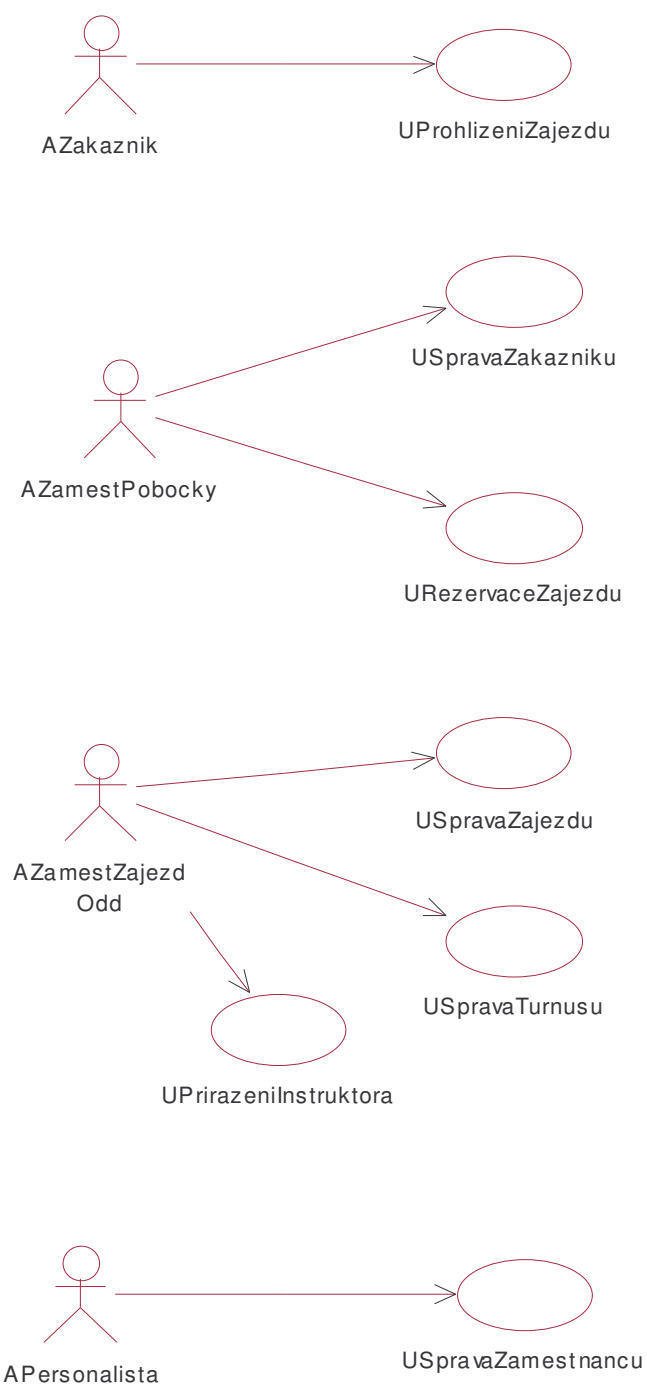
Výsledkem fáze specifikace požadavků je dokument **Specifikace požadavků**, který může mít strukturu uvedenou v tabulce 3.4.

Specifikace požadavků	
Identifikace	Aplikace Cestovní kancelář je určena pro podporu činnosti menší cestovní kanceláře. Bude umožňovat evidovat a nabízet zájezdy, evidovat zákazníky a zaměstnance, realizovat objednávky zájezdů.
Funkce aplikace	Evidence zájezdů Cestovní kancelář poskytuje různé druhy zájezdů po celém světě. Jejich výběr je každoročně upravován podle analýz prodeje z minulého roku a prognóz do budoucna. Každý zájezd, je-li v daném roce aktivní, je složen z turnusů. Turnusy se liší termínem, cenou a kapacitou. K turnusům jsou přiřazeni instruktoři. Evidence zákazníků Cestovní kancelář eviduje své zákazníky, kterými jsou soukromé osoby. Eviduje u nich, zda v minulosti již nějaký zájezd zakoupili. Evidence zaměstnanců Cestovní kancelář eviduje také své zaměstnance (vedení, zaměstnanci na pobočkách, zaměstnanci správy zájezdů, instruktoři) Objednávky zájezdů Registrovaní zákazníci se přihlašují na zájezdy (turnusy)
Popis uživatelů	Uživatelé aplikace budou následující typy: pracovníci vedení, zaměstnanci na pobočkách, zaměstnanci správy zájezdů, instruktoři zájezdů, zákazníci. Vedení firmy přijímá strategická a operativní rozhodnutí, provádí analýzy prodeje. Zaměstnanci na pobočkách spravují data o zákaznících a zajišťují přihlašování na zájezdy. Zaměstnanci správy zájezdů vytvářejí a editují zájezdy, turnusy, přiřazují instruktory k turnusům zájezdů. Zákazníci si mohou prohlížet nabídku zájezdů.
Prostředí	operační systém Windows, jednouživatelská aplikace
Požadavky na UI	klasická Windows aplikace, použití menu, panelu nástrojů, jednotný vzhled formulářů aplikace, ovládání pomocí myši i klávesnice
SW interface	generování HTML dokumentů
Omezení	programovací jazyk Object Pascal nebo Java
Předpoklady a závislosti	ukládání do souborů
Výkonnost	nepředpokládají se velké objemy dat

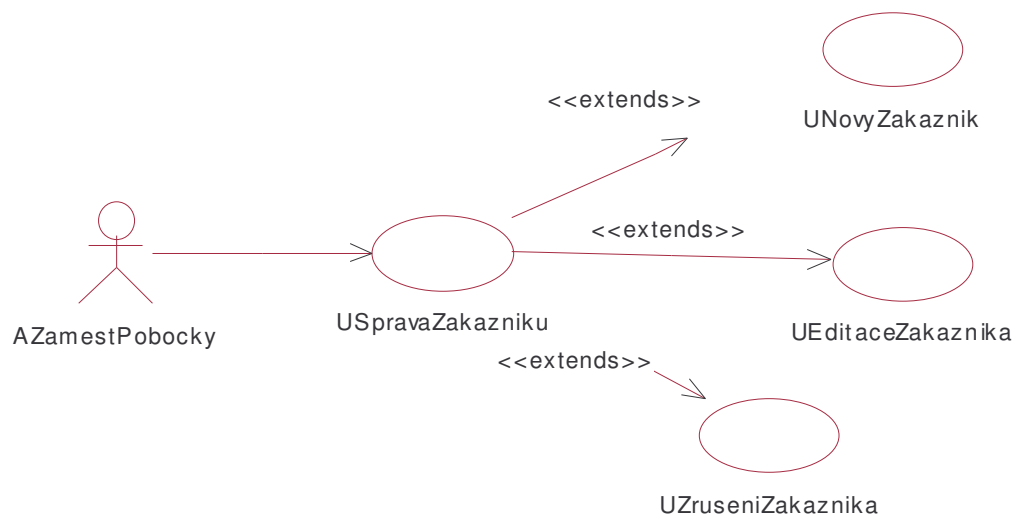
Tabulka 3.4: Specifikace požadavků aplikace Cestovní kancelář

Aplikace Cestovní kancelář - diagramy užití

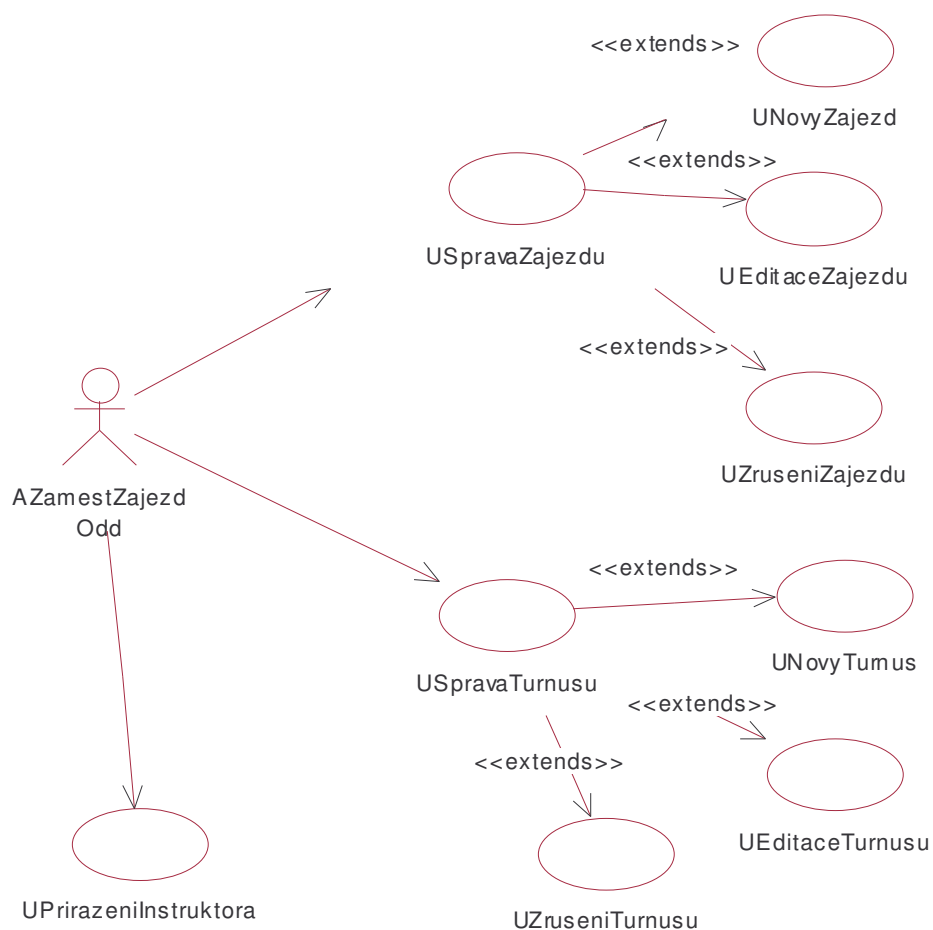
Po fázi specifikace požadavků následuje fáze analýzy, ve které se nejprve zaměříme na tvorbu diagramů užití. Na obrázku 3.24 je uveden diagram užití pro aplikaci Cestovní kancelář na nejvyšší úrovni. Hlavní diagram užití můžeme dále rozpracovat například tak, jak je uvedeno na obrázku 3.25 a 3.26.



Obrázek 3.24: Diagram užití pro aplikaci Cestovní kancelář- hlavní



Obrázek 3.25: Podrobnější rozvedení typu užití USpravaZakazniku



Obrázek 3.26: Podrobnější rozvedení typu užití USpravaZajeZdu

Identifikovali jsme 4 aktory - zákazník, zaměstnanec pobočky, zaměstnanec zájezdového oddělení a zaměstnanec personálního oddělení. Diagram užití je vytvořen v CASE nástroji Rational Rose, jehož ovládání je popsáno v příloze 1. Diagram užití by měl být doplněn

o slovní popis jednotlivých typů užití. Příklad slovního popisu typu užití je uveden na obrázku 3.27.

Případ užití UNovyZakaznik - slovní popis (scénář)

Obsluze se zobrazí formulář pro zadávání položek zákazníka: rodné číslo, jméno, příjmení, adresa - ulice, město, psč. Obsluha zadává údaje, po volbě založit zákazníka se provádějí kontroly na povinné položky - rodné číslo, jméno, příjmení - (chyba 1), rodné číslo se kontroluje podle pravidel kontroly rodného čísla (chyba 2). Program kontroluje, zda zákazník s daným rodným číslem v systému není (chyba 3). Zákazník se uloží.

chyba 1 Obsluze se oznámí "Je třeba vyplnit povinné položky" a systém se vrací do formuláře

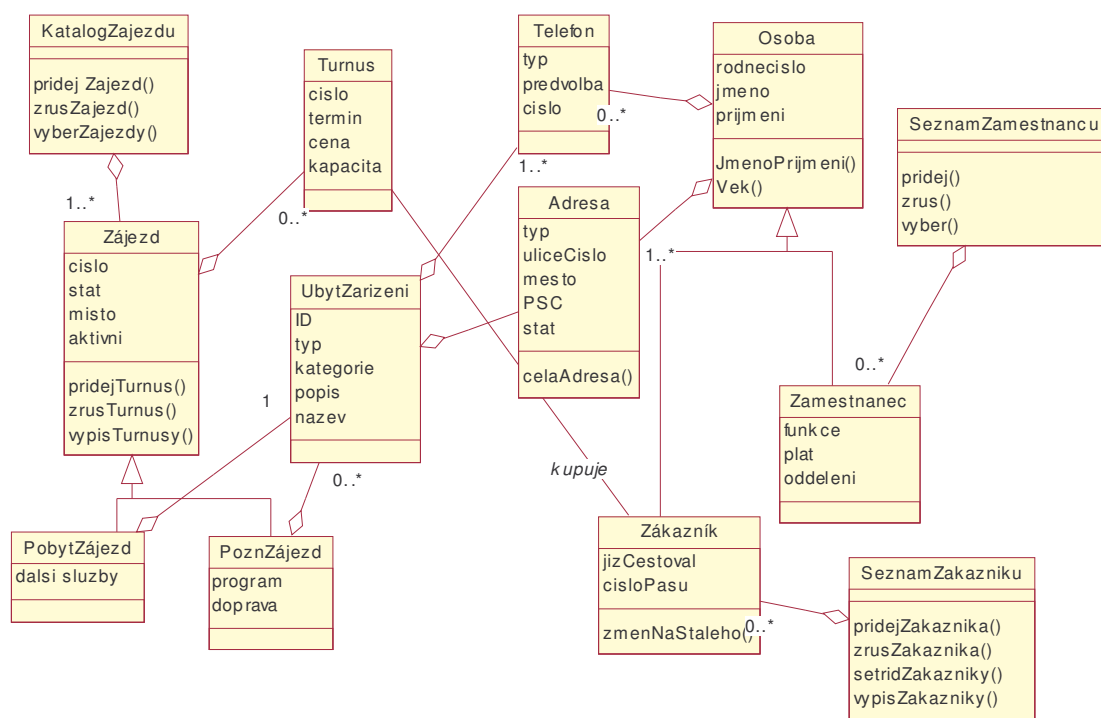
chyba 2: Obsluze se oznámí "Chybně vyplněná položka rodné číslo" a systém se vrací do formuláře

chyba 3: Obsluze se oznámí "Rodné číslo v systému již existuje" a systém se vrací do formuláře

Obrázek 3.27: Slovní popis případu užití UNovyZakaznik

Aplikace Cestovní kancelář - diagram tříd na konceptuální úrovni

Na základě diagramu užití, zejména slovního popisu typů užití a specifikace požadavků provedeme první návrh tříd. Budeme postupovat podle doporučení uvedených v 3.2.3.1 a tak můžeme dojít například k diagramu tříd uvedeném na obrázku 3.28.



Obrázek 3.28: Konceptuální diagram tříd aplikace Cestovní kancelář

Poznámka: Zastavme se u jedné otázky, se kterou se při tvorbě diagramu tříd často setkáváme. Jde o rozdíl mezi běžnou asociací a agregací. Pokusíme se objasnit daný problém na příkladě naší cestovní kanceláře. Základem aplikace je přiřazování zákazníků k jednotlivým zájezdovým turnusům. Ke každému turnusu bude přiřazeno "nula a více" zákazníků, kteří se na daný turnus přihlásili. Bude tento vztah asociací a nebo agregací?

Vztah Turnus - Zákazník jako asociace

Existuje globální seznam zákazníků, který obsahuje všechny zákazníky cestovní kanceláře (v našem případě třída *SeznamZakazniku*). V objektu třídy *Turnus* jsou potom jen ukazatele na objekty zákazníků z tohoto seznamu (viz obrázek 3.28).

Vztah Turnus - Zákazník jako agregace

Samostatný seznam zákazníků neexistuje. Objekty zákazníků jsou drženy vždy u příslušných turnusů. Přihlásí-li se zákazník na turnus, vytvoří si objekt turnus vložený vnitřní objekt

zákazníka (respektive si jej přidá do svého seznamu zákazníků přihlášených na daný turnus). Protože rozlišení, zda použít asociaci či agregaci bývá někdy obtížné, je v sporných případech lepší označit takové vztahy jako asociace a při dalším rozvoji modelu je případně změnit na agregace.

Aplikace Cestovní kancelář - specifikační diagram tříd

Ve fázi návrhu se rozhodneme o implementačním prostředí, uložení dat, vzhledu uživatelského rozhraní. V souvislosti s tím doplníme konceptuální diagram tříd o třídy uživatelského rozhraní, další třídy potřebné například pro realizaci vazeb, třídy pro přístup k datům. Protože aplikace Cestovní kancelář je poměrně rozsáhlá, provedeme návrh a implementaci jen pro malou část dané oblasti. Vezmeme pro jednoduchost jen část odpovídající typu užití *USpravaZakazniku*.

