Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Vzorový příklad analýzy informačního systému v CASE nástroji PowerDesigner

Plzeň 2013 Pavel Kraft

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 14. 5. 2013, Pavel Kraft.

Poděkování

Děkuji svým rodičům za jejich podporu při studiu. Dále děkuji
panu Ing. Josefu Weinrebovi, CSc. za jeho skvělé vedení při vzniku této práce.

Abstract

The complexity of information systems (IS) is constantly increasing. If we cannot handle with IS in its complexity during the development phase there is space for critically wrong decision. Such a decision can lead to project restart or cancellation. The role of CASE tools is to reduce these risks to minimum and to provide an option to develop an IS as a one whole which can be viewed from different points according to actual need. This approach helps to discover hidden conflicts or errors in IS architecture. The goal of this thesis is to demonstrate such an approach using Sybase PowerDesigner CASE modeling tool.

Obsah

1 Úvod

Složitost a rozsah informačních systémů (dále též IS) neustále vzrůstá. Udržet si přehled o celkovém stavu IS v různých fázích vývoje pouze z neúplných informací o té či oné komponentě nebo dílčího pohledu na systém není snadné, často nemožné. Pokud nelze s IS během vývoje zacházet v celém jeho rozsahu a komplexitě, vzrůstá pravděpodobnost kriticky chybného rozhodnutí, jelikož v mnoha případech ani nelze dohlédnout všech jeho důsledků. Takové rozhodnutí může vést až ke konci či restartu projektu, velmi nákladné noční můře každého, kdo se na vývoji IS podílí.

Úlohou CASE nástrojů je omezit tato rizika a poskytnout možnost zacházet s vyvíjeným systémem jako s jedním celkem, na který je možno nahlížet z mnoha pohledů dle potřeby. Takto lze snadno odhalit skryté konflikty či chyby v architektuře, jelikož IS vzniká od začátku jako celek.

Cílem této práce je demonstrovat na vzorovém příkladu, za použití CASE nástroje Sybase PowerDesigner výhody výše popsaného přístupu. Zvolená zájmová doména příkladu nesmí být příliš složitá, aby si zachoval svou názornost a mohl být využit jako výukový materiál. Fungování domény musí být zároveň všem důvěrně známo. Z těchto důvodů byl pro příklad analýzy zvolen hotelový IS. Při analýze bude postupováno od procesního prostředí, z tohoto prostředí budou dále odvozeny případy užití a další analytické modely UML. Na závěr bude vytvořen datový model pro konkrétní SŘBD.

2 Nástroje typu CASE

CASE je zkratkou pro Copmuter Aided Software Engineering, je to tedy softwarový nástroj pro podporu návrhu, vývoje, údržby a dalších činností během celého životního cyklu jiného softwarového produktu. Tato velmi obecná definice zahrnuje širokou škálu nástrojů od jednoduchých generátorů skriptů (jednoúčelové nástroje), až po komplexní nástroje typu Sybase PowerDesigner, dále též PD. (Více obecných informací o CASE nástrojích, jimiž se tato kapitola zabývá, ale i samotném PD, lze nalézt na [4], [6], [7] a [12]).

2.1 Historie a současnost

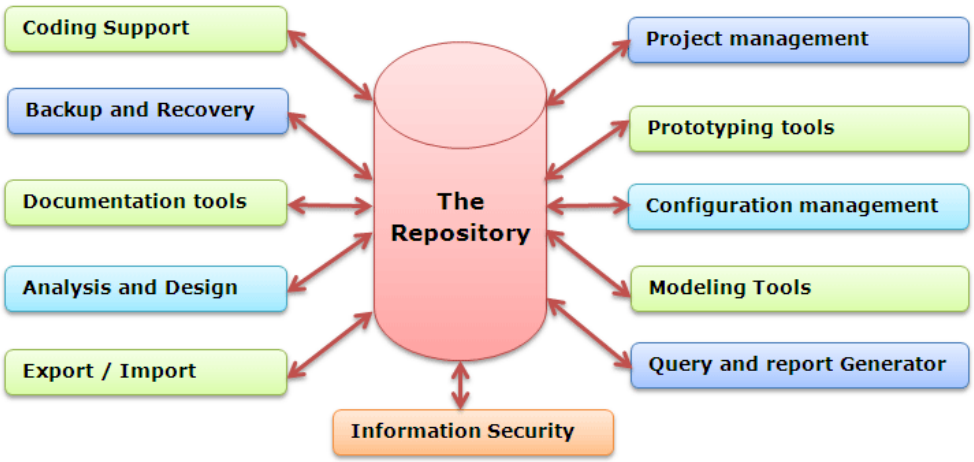
První CASE nástroje se začaly objevovat počátkem 80. let minulého století. Z počátku se jednalo o ryze textové nástroje, které sloužily např. k tvorbě a formátování dokumentace či jiným pomocným účelům. I to však (ve své době poměrně významně) ulehčilo práci programátorům, kteří se tak mohli více věnovat samotné tvorbě softwaru.

Později se, jakmile to výkon a možnosti výpočetní techniky umožnily, začaly objevovat plně grafické nástroje pro tvorbu diagramů, které vedly ke zjednodušení práce především během vývojové a analytické fáze životního cyklu software.

Další vývoj vedl ke vzniku nástrojů umožňujících ukládání informací o datových typech a použitých procesech v systému do tzv. slovníku (dále též repository). Jedná se v podstatě o základní vrstvu systému, která poskytuje data dalším vrstvám či modulům (viz Obr. 1). Tento přístup umožnil provádět automatickou kontrolu konzistence, úplnosti, či jiných logických souvislostí. V kombinaci s grafickými nástroji využívajícími slovník došlo ke vzniku již poměrně komplexních CASE nástrojů, které významně usnadňovaly návrh informačního systému, jelikož dokumenty týkající se návrhu byly mnohem lépe modifikovatelné a udržovatelné. Již nebylo nutné plýtvat časem na přímo nesouvisející rutinní činnosti. Brzy na to se objevily nástroje k testování a ověřování software, toto dále zefektivnilo vývojový proces IS.

Počátkem 90. let začaly vznikat komplexní CASE nástroje v podstatě dnešního typu, které v sobě integrovaly mnoho funkcí, včetně možnosti generovat jednodušší části kódu nebo databázové skripty pro různé SBŘD, reversního inženýrství apod. Postupně

byly uváděny též nástroje, které dokázaly obsáhnout celý životní proces IS včetně ekonomických hledisek.



Obr. 1: Slovník a moduly CASE nástroje. (zdroj: [12])

Další informace o historii CASE na [4].

2.2 Dvouvrstvá architektura

Současné CASE nástroje jsou založeny na **dvouvrstvé architektuře**. Základ každého z nich tvoří tzv. „repository“, kam se ukládají veškeré informace o navrhovaném systému (jedná se o databázi, která automaticky udržuje data v konzistentním stavu). Nad společným repository pracuje druhá modelová vrstva, která zpřístupňuje informace uložené v repository. Každá z modelových vrstev se opírá o jistou metodiku a reprezentuje jistý pohled na informace uložené ve společném repository. Jednotlivé modely jsou díky společnému repository na sebe **vzájemně převoditelné** (např. z diagramu tříd můžeme vygenerovat fyzický datový model).

2.3 Členění

Hlavní členění CASE nástrojů je dáno tím, v jaké fázi životního cyklu IS jsou využívány (viz Obr. 2). CASE nástroje se člení se do těchto kategorií:

- Pre CASE
- Upper CASE
- Middle CASE

- Lower CASE
- Post CASE

Pre CASE

Tyto nástroje slouží během přípravné fáze IS, která zahrnuje např. tvorbu globální strategie IS a dalších dokumentů nutných pro zahájení realizace IS.

Upper CASE

Slouží k mapování podnikových procesů, sběru požadavků a plánování. Pomocí těchto nástrojů vzniká globální návrh IS.

Middle CASE

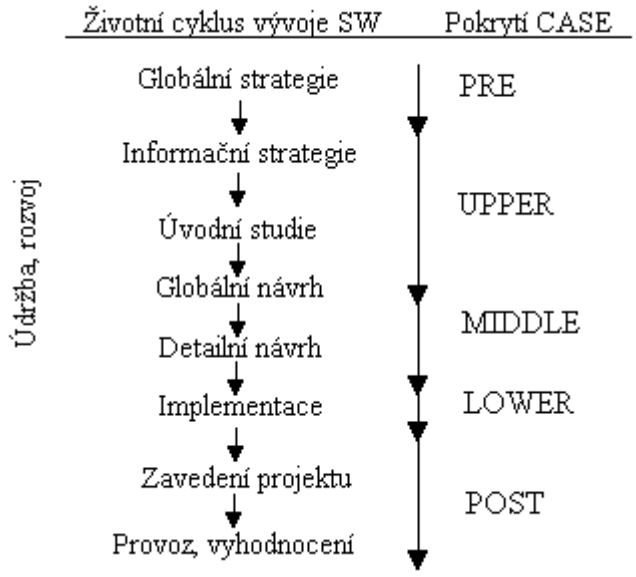
Tvoří jádro funkčnosti každého komplexního CASE prostředí. Podporuje detailní specifikaci, analýzu a návrh IS. Slouží též k vizualizaci IS.

Lower CASE

Tyto nástroje slouží během realizační fáze (implementace a testování) IS. Zajišťuje např. generování skriptů pro různé SBŘD, zpětné a dopředné inženýrství apod.

Post CASE

Tato kategorie slouží především ve fázi nasazení a údržby či při případném dalším rozvoji IS.



Obr. 2: Použití CASE nástrojů během životního cyklu IS.

2.4 Typy

Existuje mnoho typů CASE nástrojů, jenž slouží rozmanitým účelů během celého životního cyklu IS. Mezi hlavní typy patří:

- Nástroje pro modelování podnikových procesů
- Plánovací nástroje
- Nástroje pro analýzu rizika
- Nástroje pro řízení projektů
- Nástroje pro sběr požadavků
- Nástroje pro sledování metrik
- Dokumentační nástroje
- Nástroje pro kontrolu kvality
- Nástroje pro správu databází
- Nástroje pro konfiguraci software
- Nástroje pro analýzu a návrh
- Simulační nástroje
- Vývojové nástroje
- Integrační nástroje
- Testovací nástroje
- Nástroje pro reverzní inženýrství

V současné době je mnoho CASE nástrojů integrováno přímo do vývojových prostředí formou plug-inů. Rozdíl mezi komplexním CASE nástrojem a moderním vývojovým prostředím se tak v některých ohledech často stírá a není jasně definován (Obr. 3 a Obr. 4).

2.5 Co je a co není CASE nástroj

Možnosti využití moderních CASE nástrojů mohou být (a často jsou) velmi špatně pochopeny. Jejich účelem rozhodně není zastoupit člověka při analýze či návrhu IS, tedy tyto činnosti automatizovat. V obou případech je zapotřebí vyšších myšlenkových pochodů člověka, inženýra, kterých žádný automat schopen není.

Během fáze analýzy a návrhu je tento fakt poměrně zřejmý, nicméně v implementační fázi produktu tento rozdíl tak zřejmý být nemusí. Pro některé části systému je totiž možné přímo generovat implementační kódy, které vychází z analytického modelu. Nicméně během implementační fáze často vyvstávají nové okolnosti, které jsou pod rozlišovací schopností analytika či architekta, ale které mohou zásadně ovlivnit návrh systému. Celý proces převodu návrhu systému do použitelného produktu deterministicky popsat nedokážeme, proto je i zde lidská práce z velké části stále nenahraditelná. Na druhou stranu schopnosti CASE systému např. generovat skripty je velmi vhodné využít, ovšem člověk by vždy měl vědět, co dělá, právě kvůli výše popsaným možným důsledkům.

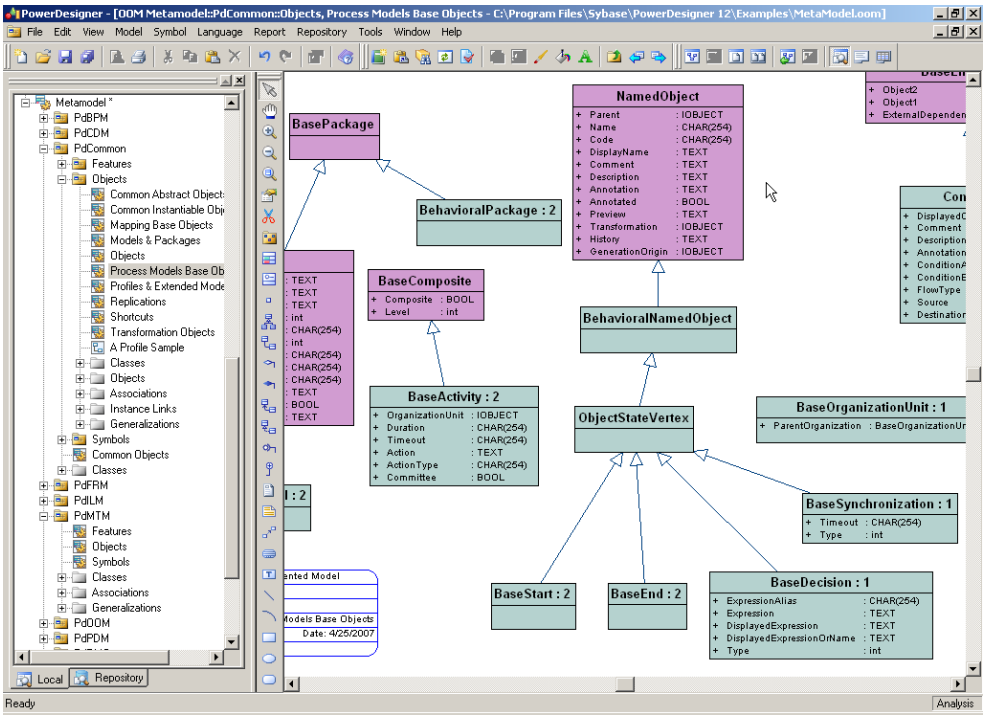
CASE nástroje nejsou samospasitelné, nelze očekávat, že se celý softwarový produkt s jejich pomocí zhotoví sám. Jsou to nástroje podpůrné, to ovšem nesnižuje jejich význam a užitečnost. Hlavními přínosy a negativy CASE nástrojů jsou:

Přínosy

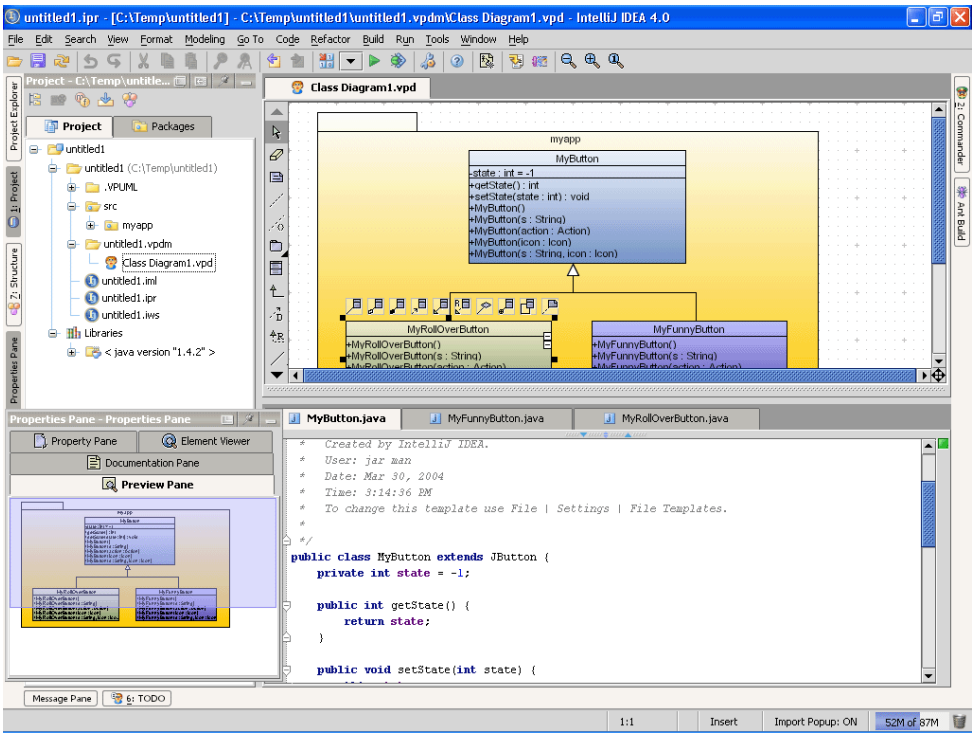
- dramaticky snižují náklady na vývoj
- podporují práci v týmu
- zvyšují produktivitu práce
- zrychlují vývojový proces
- umožňují snadnou modifikaci/údržbu produktu
- zkvalitňují návrh systému

Negativa

- vysoké počáteční náklady na zavedení
- výhody se neprojeví okamžitě
- vyžadují správné zacházení a školený personál



Obr. 3: CASE nástroj PowerDesigner od společnosti Sybase.



Obr. 4: Vývojové prostředí IntelliJ IDEA s integrovaným CASE nástrojem.

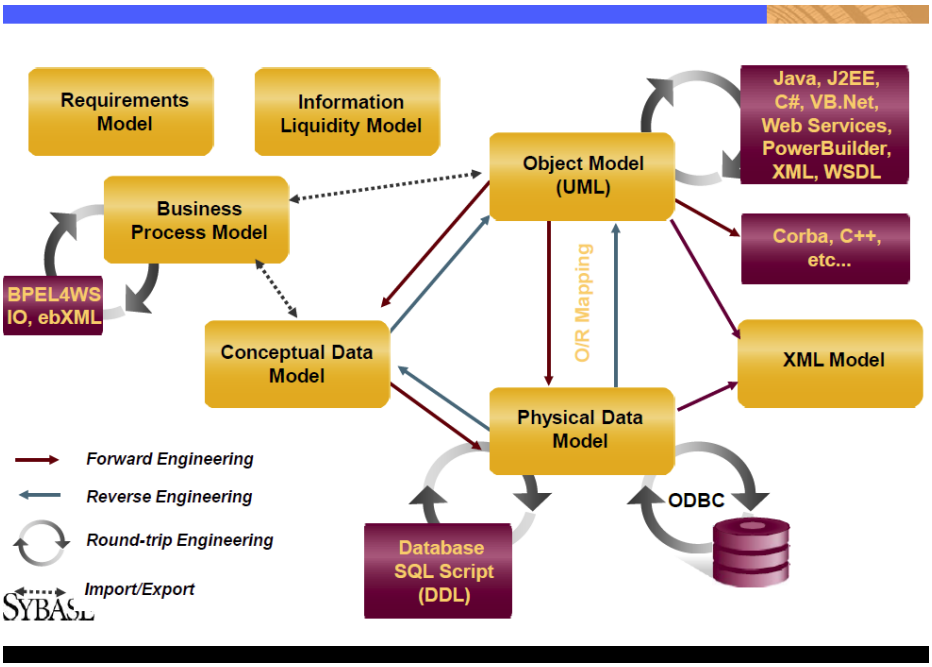
2.6 Sybase PowerDesigner

PowerDesigner patří mezi nejznámější CASE nástroje souččasnosti. Od roku 1995 je vyvíjen americkou společností Sybase (Sybase v roce 2010 zakoupil německý SAP). Jedná se o komplexní prostředí, které umožní pokrýt všechny aspekty podnikového rozvoje. Umožňuje analyzovat obchodní procesy v podniku, provádět datovou a objektovou analýzu informačních systémů. PowerDesigner nabízí integrované prostředí, které podporuje široce rozšířené přístupy a standardy jako je např. UML (Unified Modeling Language).

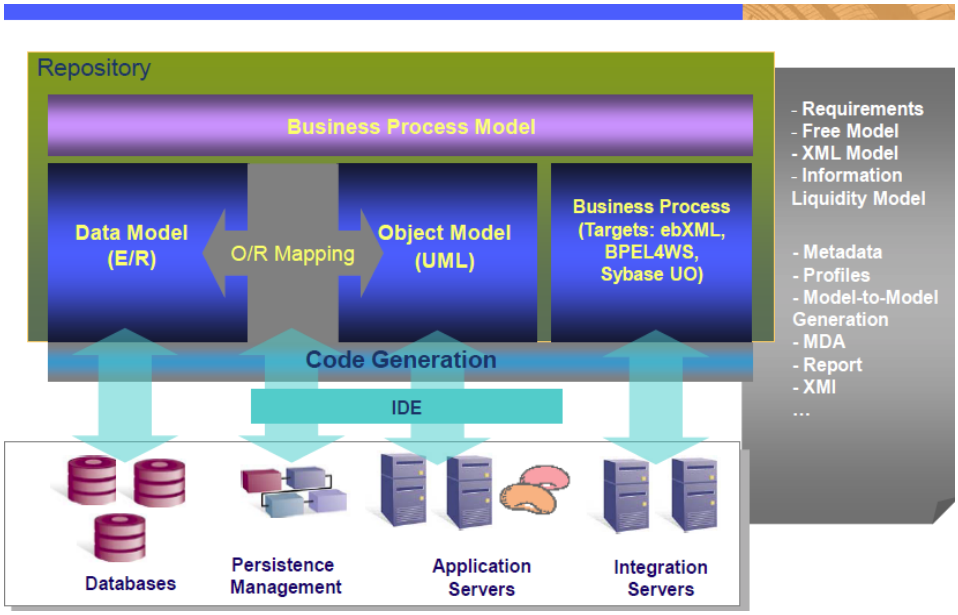
Tento CASE systém je vhodný pro manažery, analytiky, vývojáře nebo lidi zabývající se tvorbou dokumentace. Díky integraci všech modelů, dílčích pohledů na systém, dokáže každému pracovníkovi dle jeho role poskytnout odpovídající pohled na vyvíjený produkt a zajistit, aby jednotlivé modely nebyly v rozporu. Tímto přístupem se značně urychluje vývoj IS a omezuje se prostor pro vznik chyb vlivem nedokonalé komunikace mezi členy týmu během vývoje (pro detailní informace o možnostech PowerDesigneru viz [6]).

2.6.1 Vlastnosti

PD obsahuje moduly, které se zabývají procesním, objektovým a datovým modelováním. Tyto moduly jsou vzájemně provázány a jednotlivé modely lze mezi sebou převádět. Z modelů vytvořených v PD lze též generovat databázové skripty, nebo části implementačního kódu. Tyto vlastnosti PD je z hlediska vývoje IS jedny z nejpřínosnějších. Na Obr. 5 jsou znázorněny moduly PD a vztahy mezi nimi. Na Obr. 6 je zobrazena struktura PD.



Obr. 5: Moduly PD a vztahy mezi nimi. (zdroj: [6])



Obr. 6: Struktura PD. (zdroj: [6])

Následující stručný výčet dalších vlastností vychází z informací uvedených na stránkách výrobce PowerDesigneru (viz [6] a [7]).

Řízení požadavků

Sběr, provázání a reportování požadavků, jejich hierarchické zpracování a přiřazení jednotlivým uživatelům. Možnost synchronizovat požadavky s dokumenty ve formátu Microsoft Word.

Analýza dopadu změn

Přehledné zobrazení všech dopadů do modelu ještě před samotným provedením změny.

Generování dokumentace

Účinný drag-and-drop nástroj pro automatizovanou tvorbu dokumentace. Export do všech běžných formátů, RTF, HTML, Excel, atd.

Široké možnosti rozšíření

Customizovatelné GUI, tvorba vlastních rozšíření, uživatelské skripty.

Mapovací editor

Drag-and-drop nástroj umožňující mapovat na sebe jednotlivé objekty mezi **datovými modely**.

Podporované modelovací techniky

PowerDeisgner podporuje následující modelovací techniky:

- Modelování business procesů (BPM)
- Datové modelování
    - Modelování založené na principu tří architektur (konceptuální, logická, fyzická) a modelování datových skladů. Podpora Javy, XML a webových služeb v databázích.
- XML modelování
    - Podpora XML DTD a Schema elementů.
- Objektové modelování
    - Modely vycházející z UML 1.x a 2.0, široké možnosti úprav podle potřeb uživatelů.

Podporované platformy

Platformy podporované PowerDesignerem jsou následující:

- Procesy
    - BPMN, ebXML, BPEL4WS, podpora SOA.
- RDBMS
    - Obousměrný engineering pro téměř 60 relačních databází včetně nejnovějších verzí Oracle, IBM DB/2, MS SQL Server, Sybase, MySQL a mnoha dalších.
- Objektové jazyky
    - Obousměrný engineering pro jazyky Java, C#, C++, PowerBuilder, XML, VB.NET a další.
- Integrace při vývoji
    - Plug-iny pro synchronizaci kódu s modelem v nástrojích Eclipse, PowerBuilder a Visual Studio.

2.6.2 Dostupné edice PowerDesigneru

DataArchitect

Splňuje požadavky nejnáročnějších datových modelářů a DB administrátorů. DataArchitect nabízí řízení požadavků a plný rozsah funkcionality pro víceúrovňovou analýzu a design včetně dopředného i zpětného inženýrství pro téměř 60 poskytovatelů/verzí relačních databází.

Developer

Varianta určená pro objektové modelování včetně řízení požadavků. Plná podpora UML. Obsahuje funkcionalitu pro zvýšení produktivity při použití s těmito jazyky/nástroji: Java, C#, VB.NET, XML, PowerBuilder, aj. Variantu Developer je možné propojit s většinou rozšířených vývojových nástrojů pro zajištění automatické model-to-code synchronizace.

Studio

Naplňuje potřeby zejména vedení IT a obchodu a poskytuje možnosti sladění cílů obou těchto oddělení. Kombinuje funkcionalitu variant DataArchitect, Developer a ještě

přidává možnost modelovat obchodní procesy. Varianta Studio tak umožňuje vytvořit komplexní pohled na strukturu a chování celé firmy.

Viewer

Tuto variantu využijí zejména týmy tvořící dokumentaci, techničtí manažeři a další uživatelé, kteří potřebují přístup ke čtení modelů. Je možné z něj přistupovat do Repository, tisknout modely a vytvářet reporty. Viewer je k dispozici zdarma na adrese http://www.sybase.com/products/powerdesigner.

Enterprise

Varianty s přívlastkem Enterprise obsahují navíc připojení do Repository pro každého uživatele. Na Enterprise varianty je možné upgradovat i ze základních verzí PowerDesigneru v okamžiku potřeby.

3 Vznik a význam UML

UML je v současnosti nejrozšířenější standardizovaný prostředek pro objektové modelování, který je též využíván pro datové a procesní modely. Jedná se o vizuální jazyk, který umožňuje popsat strukturu, chování (dynamiku) a architekturu softwarových systémů. Ovšem využít jej lze i pro popis systémů a jejich procesů obecně (BPM), nemusí se jednat nutně o software, vyjadřovací schopnosti UML jsou mnohem širší. Více informací o historii UML a jeho použití přímo od jeho tvůrců v [1], dále pak v [3] a na [11]. Pro popis základů jazyka UML viz přílohu A.

S tím jak se v 80. letech minulého století objektově orientované jazyky stávaly stále více přijímané a široce rozšířené, vyvstala potřeba výrazového prostředku, který by umožnil zachytit nové přístupy v návrhu a tvorbě stále složitějších aplikací. V reakci na to vzniklo mnoho jazyků, potažmo metodik, které si vzájemně konkurovaly, ovšem žádná z nich nebyla dokonalá. Postupem času se ustálilo několik nejvýznamnějších, např. Booch, OMT, OOSE. Ani jedna ale nebyla univerzální a měla své specifické slabé stránky. Tento neutěšený stav bylo třeba řešit.

Jelikož autoři tří hlavních metodik, Grady Booch (Booch), James Rumbaugh (OMT) a Ivar Jacobson (OOSE), usilovali o to samé, spojili své síly a zkušenosti nabité při vývoji vlastních metodik. Rozhodli se vytvořit sjednocený modelovací jazyk, Unified Modeling Languge - UML.

Tato snaha vyústila v roce 1996 vydáním UML specifikace verze 0.9. Mnoho softwarových společností, včetně odborné veřejnosti rozpoznalo význam UML pro softwarové inženýrství. Přijaly tedy UML za své a chtěly se účastnit na jeho rozvoji. Tak vzniklo UML konsorcium, mezi jehož zakládajícími členy byly společnosti jako IBM, Hewlett-Packard, Oracle, Microsoft, Texas Isntruments a mnoho dalších. Výsledkem této spolupráce byla UML verze 1.0.

V lednu 1997 došlo ke standardizaci UML 1.0 organizací OMG (Object Managment Group – více informací o této organizaci na [10]). Rozvoj UML pokračoval pod patronátem OMG a následovalo několik dalších verzí UML. V roce 2005 byl však přijat standard UML 2.0, který zaváděl mnoho vylepšení na základě zkušeností s předešlými verzemi. Verze 2.0 je nejrozšířenějším standardem UML (vývoj však probíhá i nadále).

4 Základní principy a význam modelování

Realita je nekonečně složitá. Každý IS je nějakým způsobem na svět kolem nás informačně napojen a musí této složitosti čelit. To, jakým způsobem se tak bude dít, zásadně ovlivní jeho funkčnost. Hlavní úlohou analytika (a nejen jeho) je zajistit, aby IS, který je vždy modelem reality, postihoval přesně tu její část, která je pro systém podstatná a rušivými, zbytečnými detaily se nezabýval. Tomuto základnímu principu se říká abstrakce. Ve správném zacházení s abstrakcí tkví umění modelování.

Jedním z nejpoužívanějších postupů či paradigmat při zacházení s abstrakcí je tzv. hierarchická abstrakce. Hierarchickou abstrakci používáme dvěma způsoby:

- Konkretizace
- Generalizace

Konkretizace

Při analýze problému postupujeme od obecných celků ke konkrétním (tzv. top-down princip), snažíme se problém dekomponovat na snadněji zvládnutelné části. Pomocí tohoto tradičního principu dokážeme čelit složitosti modelované problematiky.

Generalizace

Opakem principu konkretizace je generalizace. Při generalizaci budeme posupovat „od zdola nahoru“ (bottom-up) - z konkrétních celků skládat obecnější, hledat mezi nimi vzory, zobecňovat a vytvářet tak vyšší úrovně modelu. Pomocí principu generalizace dokážeme v modelu nalézt společné vlastnosti pro více prvků a tímto zobecněním si velmi ulehčit práci s modelem.

4.1 Princip tří architektur

Princip tří architektur (P3A) definuje, jakým způsobem využít abstrakci při vývoji IS. Při vývoji pomocí P3A zavádíme jednotlivé abstrakční vrstvy tak, abychom postihli všechny hlavní aspekty vyvíjeného systému. Jedná se o:

- Obsah
- Technologii
- Implementaci

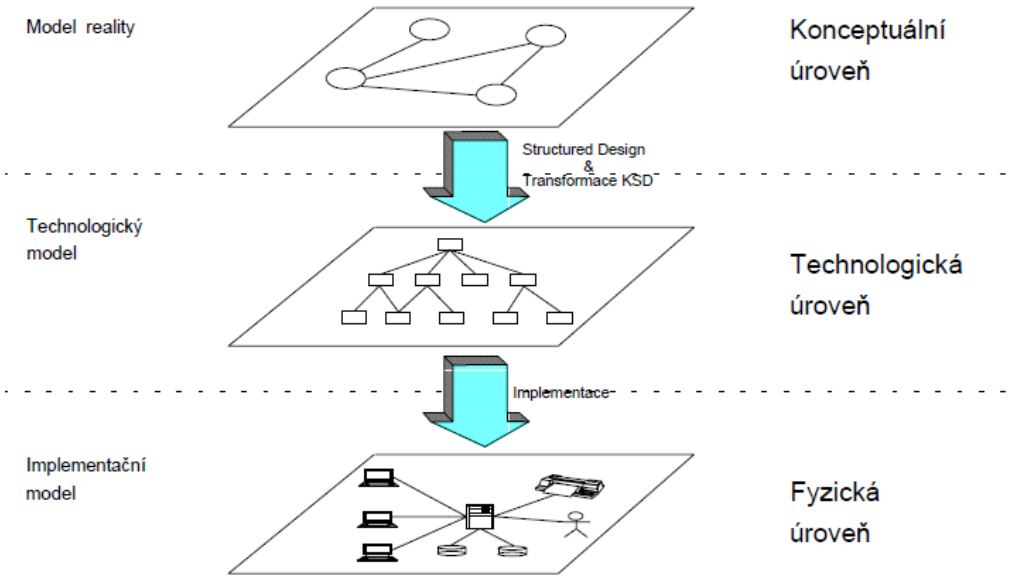
Tyto aspekty spolu velmi souvisí a dokonce jeden vyplývá z druhého. Pokud se řídíme principy P3A, vývoji IS pak probíhá v těchto třech fázích:

- Analýza
- Návrh
- Implementace

Analýza systému spočívá v nalezení logického modelu, který se nezabývá technologickými ani implementačními specialitami, nýbrž základním smyslem systému, jelikož o ten jde v první řadě. Během analytické fáze určujeme, **CO** je obsahem systému.

K **návrhové** fázi můžeme přikročit, až když jsme si jisti, že náš analytický model odpovídá požadavkům na systém. Návrhový model se zabývá především technologickou koncepcí IS. Ovšem nadále platí, že by tato koncepce neměla být zatížena implementačními specifiky. Během návrhové fáze se zajímáme o to, **JAK** bude obsah realizován technologicky.

S hotovým návrhem IS můžeme přikročit k vypracování **implementačního** modelu. Ten zahrnuje již veškeré implementační prostředky, jako konkrétní databázové stroje, programovací jazyky, vývojová prostředí apod. Implementační návrh určuje, **ČÍM** je technologické řešení realizováno. Princip tří architektur je znázorněn na Obr. 7. Více informací o principu tří architektur na [5].



Obr. 7: Princip tří architektur (zdroj: [5]).

4.2 Princip separace různých pohledů

Když provádíme analýzu IS (ale platí to i pro návrh a ostatní vývojové fáze), nikdy nemůžeme na systém nahlížet jako na celek v tom smyslu, že bychom obsáhli veškeré souvislosti a důsledky. Takového přístupu není člověk schopen. Proto je nutné na vyvíjený IS nahlížet vždy z určitého úhlu pohledu. Jednotlivé pohledy, volíme tak aby se vzájemně doplňovaly. Tyto pohledy jsou vyjadřovány a zaznamenávány UML a jinými diagramy. Díky moderním CASE nástrojům, jako je PD, můžeme tyto modely spravovat najednou a tím tak docílit uceleného pohledu na systém.

5 Zájmová doména

5.1 Volba domény

Pro náš vzorový příklad analýzy bylo třeba vybrat takovou doménu, jejíž fungování je každém důvěrně známo. Nemělo by se jednat o systém statický, měl by mít určitou vnitřní dynamiku, aby bylo možné tyto děje zachytit a demonstrovat vlastnosti jednotlivých analytických modelů. Zároveň bylo třeba, aby zájmová doména nebyla příliš rozsáhlá a složitá nebo aby její případné zjednodušení neubíralo vzorovému příkladu na názornosti. Tyto podmínky výborně splňuje malý hotel či pension. Jako vzorový příklad tedy poslouží analýza IS hotelu.

5.2 Popis domény

Jedná se o běžný hotel o několika desítkách pokojů. Pokoje mohou být dvojlůžkové či jednolůžkové, vybavené případně manželskou postelí. Pokoje mohou mít různou cenu. Středobodem veškerého dění v hotelu je recepce. Skrze recepci lze vytvářet rezervace, ubytovat hosta, či ho odhlásit. Hotelový management sleduje vytíženost hotelu a další parametry a dle toho upravuje ceny ubytování. Provozní personál dbá na pravidelný úklid pokojů a čistotu v ostatních prostorách hotelu. Součástí hotelu je též restaurace a různé služby pro rekreační vyžití, ty ale při analýze IS zanedbáme, budeme se zbývat pouze ubytovací částí hotelu.

6 Příklad: informační systém hotelu

Náš vzorový příklad se zaměří na analytickou fázi vývoje IS. Cílem je postupně na navrhovaný systém pomocí analytických modelů (diagramy UML a další) nahlédnout z různých úhlů pohledu a tím demonstrovat jak význam samotného UML, tak i roli CASE nástrojů, bez nichž je správa a údržba souvisejících analytických modelů a tím i samotný vývoj software dnes již velmi krkolomný. Zvolený postup analýzy (mimo jiné) volně vychází z metodiky popsané v [2] a na [8].

Konkrétně se bude jednat o tyto modely (či diagramy):

- Model požadavků
- Diagram rozkladu procesů
- Diagram případů užití
- Diagram aktivit
- Diagram tříd
- Sekvenční diagram
- Konceptuální datový model
- Fyzický datový model

Složitost IS byla pečlivě volena tak, aby čtenáře nezahltil zbytečnými detaily, ale aby zároveň byl pro svůj účel dostačující.

6.1 Diagram případů užití

Diagram případy užití je při analýze IS velmi důležitým prvkem. Je úplným popisem funkčnosti IS z pohledu uživatele, tzv. funkční obálkou systému. Vymezuje tak hranice IS vůči okolnímu světu. Tento pohled na systém je velmi podstatný. Účelem každého IS je naplnit požadavky uživatele, jež jsou vyjádřeny právě případy užití. To je důvodem, proč se z případů užití vychází při tvorbě dalších analytických modelů. Nejinak tomu bude i u našeho vzorového příkladu. Případy užití nehrají ovšem roli pouze ve fázi analýzy IS, ale i během návrhu a implementace. Protože popisují funkčnost systému, prostupují celým jeho vývojem, až dokud není požadované funkčnosti dosaženo. To jakým způsobem určíme případy užití, velmi zásadně ovlivní další vývoj IS. Pokud jsou

v této fázi (analýza) případy užití určeny chybně, znamená to vážné problémy pro celý vývoj IS jednoduše proto, že systém, který neplní přání svého uživatele, nikdo nechce.

6.1.1 Jak nalézt případy užití

Známe-li nyní význam případů užití pro vývoj IS, hodilo by se vědět, jak pro konkrétní systém odpovídající případy užití nalézt. Povede-li se nám již nějaké ty případy užití nalézt, můžeme si být jisti, že jsme nalezli všechny? Jsou správně členěny? Identifikovali jsme správně všechny aktéry? Toto jsou velmi podstatné otázky, ty nejdůležitější při hledání případů užití. Jejich zodpovězení ovšem není triviální záležitost.

Ukážeme si dva hlavní způsoby, jak nalézt případy užití pro náš systém (detailněji viz [2]). Tím prvním, dnes již klasickým způsobem, je hledání případů užití skrze aktéry systému. Postup je v takovém případě následující:

- Nalezneme všechny aktéry systému
- Zjistíme, jaké služby či informace aktér od systému vyžaduje
- Určíme konkrétní případy užití

Ukázalo se, že tento přístup, ač mnohdy plně dostačující, má své slabiny. Identifikace jednotlivých aktérů je často složitá (nemusí se nám povést identifikovat spolehlivě všechny). Rozdíly mezi aktéry též mohou být zpočátku nejasné, důsledkem čehož jsou špatně určené případy užití. Ukáže-li se např., že prvek, jehož jsme považovali za jednoho aktéra, jsou ve skutečnosti aktéři dva, mohou následně potřebovat oba více či méně odlišné případy užití, ovšem my už jsme případy užití určili pro původního jednoho aktéra, zde vzniká prostor pro vážnou chybu. Hledání případů užití skrze aktéry systému je často spolehlivým způsobem jejich nalezení. Ovšem pokud je systém tvořen pro procesně složitější prostředí, tato metoda kvůli složité identifikaci aktérů selhává. Netřeba ji ovšem nadobro zatracovat, vždy se bude hodit např. pro ověřování případů užití, stává se doplňkovou metodou metody jiné, dokonalejší.

Důvodem, proč metoda hledání aktérů selhává, je složité procesní prostředí. Druhá metoda hledání případů užití, kterou si představíme, se proto bude týkat procesního mapování okolí, ve kterém IS působí – podnikových procesů. Tento přístup je

v současnosti široce přijímaný a stává se dominantní metodou pro získávání případů užití. Postupujeme takto:

- Identifikujeme všechny podnikové procesy.
- Určíme, které z procesů využívají IS, a na ty se více zaměříme. Tyto procesy „vyvolávají“ nebo „spouštějí“ jednotlivé případy užití.
- Nalezneme všechny případy užití vyvolané podnikovými procesy.

Tato metoda se ukázala jako velmi spolehlivá, protože nám při správném provedení umožní beze zbytku určit všechny případy užití. Již se neptáme „kdo - jaký aktér“ systém používá, ale „co - jaký proces“ vede k použití systému. Pokud zmapujeme podnikové procesy, můžeme jednoznačně určit rozhraní mezi IS a jeho procesním okolím. To nám umožní spolehlivě určit i případy užití.

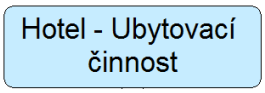
6.1.2 Hledání případů užití

Nyní když máme základní přehled o metodách hledání případů užití, aplikujeme je na náš příklad. Nejdříve se zaměříme na procesní model hotelu, ze kterého určíme případy užití. Dále provedeme sběr požadavků na IS, určíme aktéry systému a nalezené případy užití pomocí nich ověříme.

Diagram rozkladu procesů a diagram podpory IS

Začneme tedy hledáním podnikových procesů. Podnikový proces je takový proces, který pokrývá činnost nebo jiným způsobem souvisí s činností podniku. Při hledání podnikových procesů se zaměříme pouze na ubytovací část hotelu. Každý proces můžeme dělit na podprocesy (pokud se nejedná o proces o jedné atomické akci). Při hledání podnikových procesů je možné a velmi vhodné využít jejich Top-Down hierarchie a postupovat od „vyšších“ procesů k „nižším“. Tento postup se nazývá procesní rozklad. Ukažme si ho na příkladu.

Nejprve identifikujeme hlavní proces podniku, v našem případě hotelu. Ptáme se: „Co je hlavním účelem hotelu?“ Hlavním účelem hotelu je jeho ubytovací činnost, která přináší zisk. Zavedeme si tedy základní, „top“ proces hotelu (viz Obr. 8), který v sobě zahrnuje vše, co se v hotelu za tímto účelem děje.

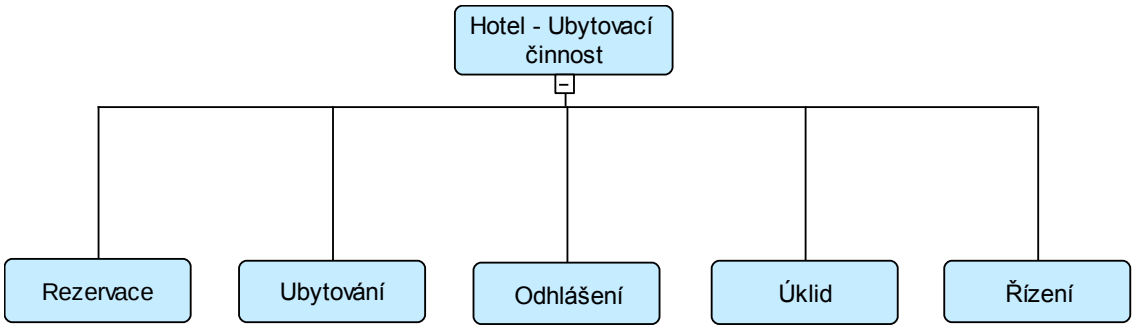


Obr. 8: Základní proces hotelu - ubytovací činnost.

Nyní, když máme zaveden hlavní proces hotelu, můžeme přikročit k jeho rozkladu. Aby ubytovací služba správně fungovala, je nutné hotel udržovat čistý a uklizený, někdo musí hotel řídit a někdo musí vyřizovat rezervace, ubytování hostů a jejich odhlašování. Rozpoznali jsme hned několik dalších procesů – podprocesů procesu ubytovací činnost:

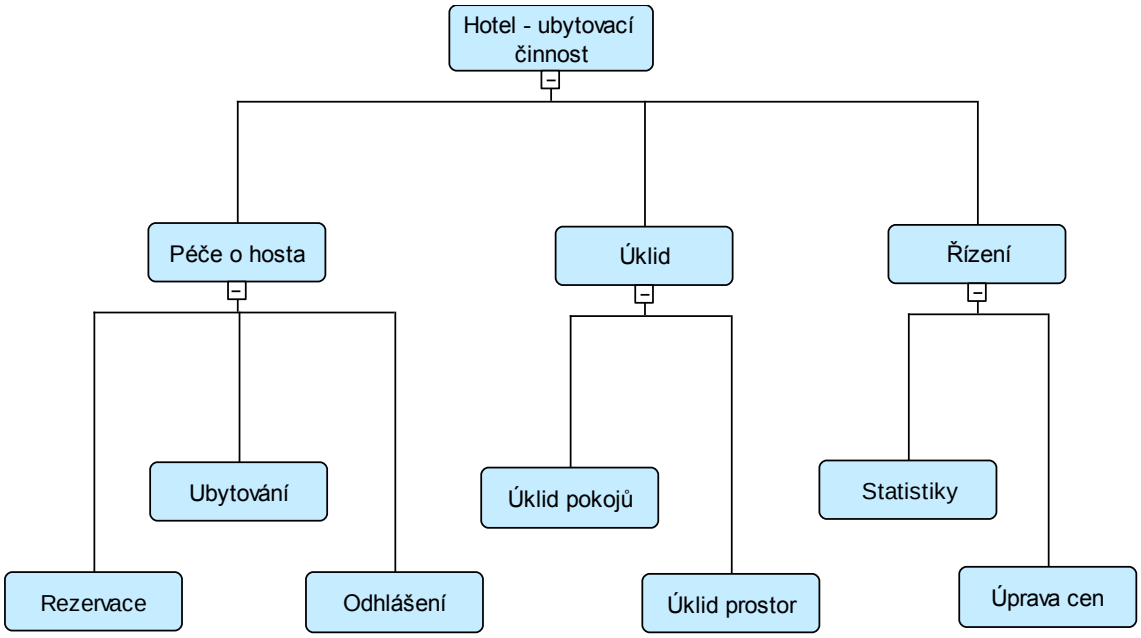
- Úklid
- Řízení
- Rezervace
- Ubytování
- Odhlášení

Rozložený proces ubytovací činnost je zobrazen na Obr. 9.



Obr. 9: Rozložený proces ubytovací činnost.

Proces úklid bychom dále mohli rozložit na podprocesy úklid pokojů a úklid ostatních prostor hotelu. Podobně bychom mohli postupovat u procesu řízení, šel by rozložit na získávání statistik z provozu hotelu a následnou úpravu cen, dle toho jaká je aktuální vytíženost apod. Všimněme si, že jiná situace ovšem nastává v případě zbývajících procesů. Tyto procesy, mimo to, že jsou podprocesy základního procesu ubytovací služba, mají společnou ještě jinou věc. Všechny se přímo týkají zákazníka – hosta. Zasloužily by být zastřešeny pod novým, zvláštním procesem, nazvěme jej třeba péče o hosta. Odpovídající procesní rozklad je zobrazen na Obr. 10.



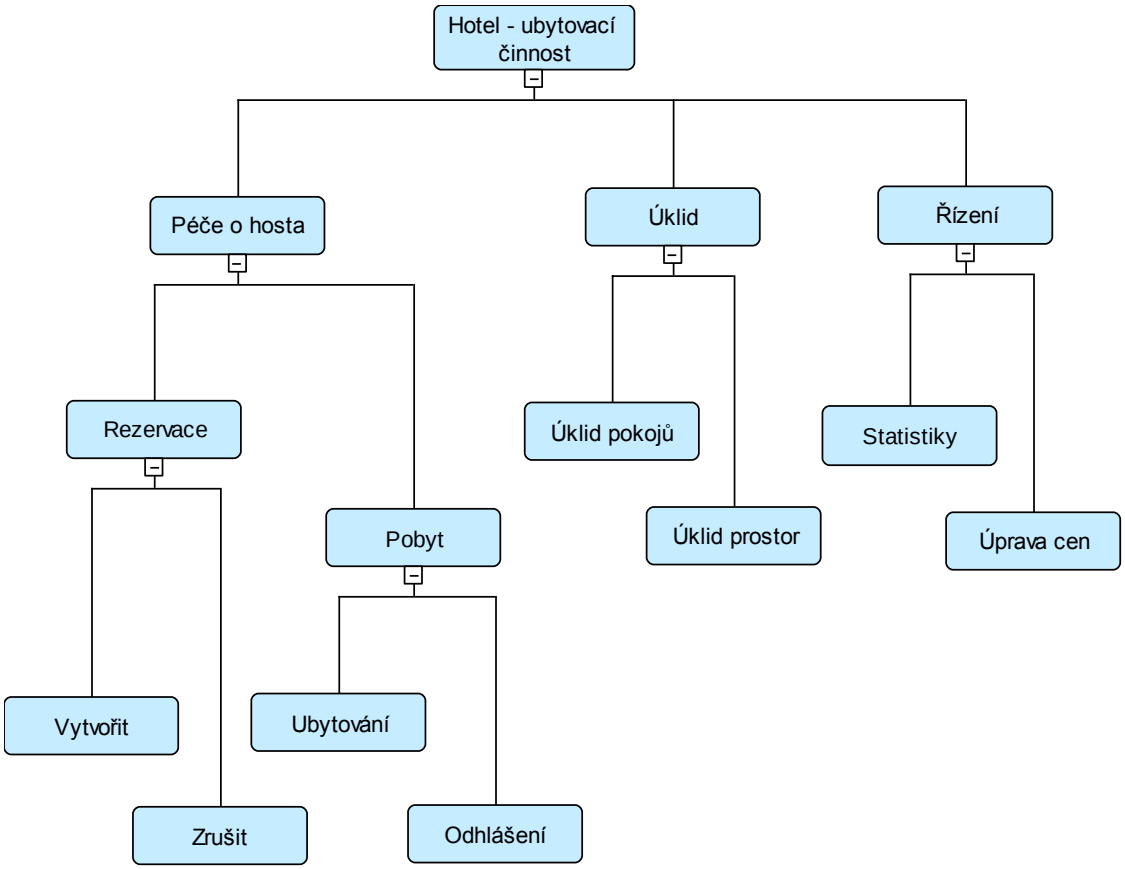
Obr. 10: Procesní rozklad hotelu - další členění.

To, co jsme nyní provedli, je přesný opak rozkladu procesu (více o použití hierarchické abstrakci v kapitole 4). Sjednotili jsme několik procesů a vytvořili tím nový, zastřešující proces. Důvodů, proč jsme tak učinili, je několik:

- Vyváženost rozkladu - na stejné úrovni by měli být procesy vyjadřující stejnou „úroveň detailu“.
- Procesy, které jsme sjednotili, spolu souviseli a měli spolu mezi sebou společného více, než s ostatními procesy.

Všimněme si toho, že ač rozkladem podnikových procesů vzniká strom, rozhodně se celý nedá vytvořit prostou dekompozicí od shora dolů. Ve skutečnosti se nám povede tu nějaký proces rozložit, tu identifikovat nové procesy, které umístíme na příslušné místo ve stromě nebo procesy sjednotit pod nový zastřešující proces. Rozkladový strom vzniká postupně, často z nesouvislých částí, které nakonec při správném postupu vytvoří celek. Vzniklému rozkladovému stromu říkáme diagram rozkladu procesů. Dalším zpřesněním našeho rozkladu vznikne diagram rozkladu procesů na Obr. 11.

V PD vytvoříme diagram rozkladů procesů volbou *File > New Model*. Jako typ modelu zvolíme *Bussines Process Model* a jako typ diagramu *Process Hierarchy Diagram*.



Obr. 11: Diagram rozkladu procesů ubytovací činnosti hotelu.

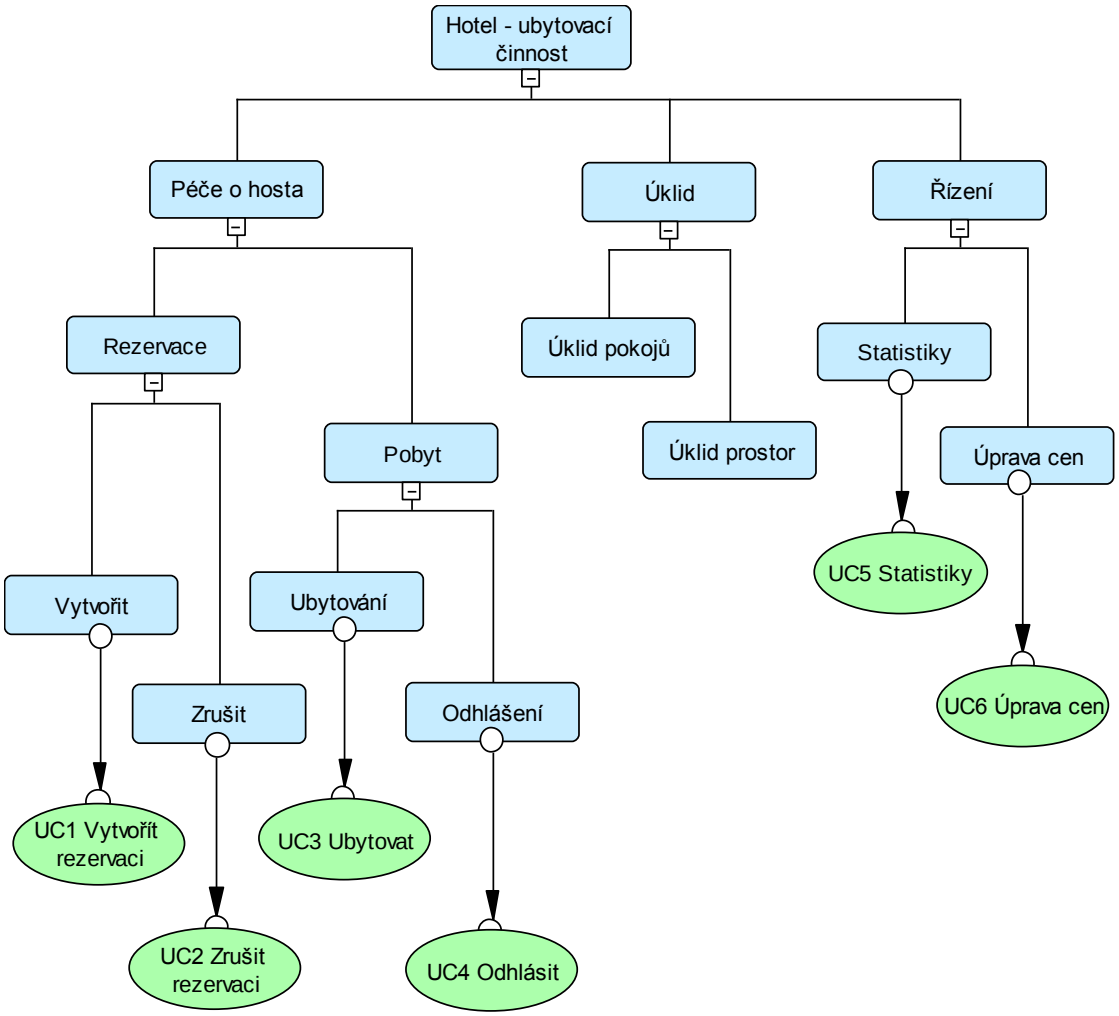
Při tvorbě rozkladového stromu dbáme na vyváženost stromu a na postupné „zjemňování“ od „vyšších“, obecnějších, k procesům „nižším“, konkrétnějším. Nyní ovšem vyvstává otázka: kdy a proč rozklad procesů zastavit? Teoreticky bychom mohli rozkládat procesy až dokud bychom nenarazily na atomické operace. Vždy musíme mít na paměti, že důvodem, proč vůbec nějaký strom procesů tvoříme, je, že **hledáme případy užití** pro náš IS. Tím se musí řídit i samotný procesní rozklad a jeho konec. Rozkládat přestáváme v případě, že:

- Daný proces spouští případ užití našeho IS.
- Když jsme si jisti, že dalším rozkladem procesu již žádný další případ užití nemůže být aktivován (celá procesní větev se odehrává mimo IS).

Každý případ užití by tak měl být spouštěn procesy, které jsou listy rozkladového stromu. Vše co je „pod“ procesy, které spouštějí případy užití IS, už do diagramu rozkladu procesů v našem případě nepatří. Tyto procesy už se totiž odehrávají přímo v IS a popisem těchto dějů se zabývají případy užití. Procesy, které stojí „mimo“ či

„nad“ případy užití nás z pohledu IS též nezajímají, jelikož se odehrávají mimo jeho hranice. Diagram, který zobrazuje aktivaci případů užití procesy, se nazývá diagram podpory IS a vypadá takto (viz Obr. 12).

PD bohužel neumožňuje vytvořit přímo diagram podpory IS. Nabízí ovšem širokou škálu doplňkových grafických nástrojů, můžeme si tedy upravit diagram rozkladu procesů tak, aby plně odpovídal našim potřebám (do podoby diagramu podpory IS).



Obr. 12: Diagram podpory IS.

Díky procesnímu rozkladu jsme nalezli hned několik případů užití, věnovat se jim budeme v dalších podkapitolách.

Rozklad procesů není jednoznačný a diagram podpory IS může mít na vyšších úrovních mnoho podob. Podstatné je to, že ať provedeme rozklad jakkoliv, listy stromu by vždy měli vést ke stejným případům užití.

Diagram podpory IS, krásně znázorňuje hranici našeho IS v procesním prostředí podniku. To byl hlavní cíl našeho procesního modelování. Díky tomu můžeme nalézt všechny případy užití pro náš IS.

Sběr požadavků

Správně provedený sběr požadavků je pro úspěšný IS velmi důležitý. Požadavky jsou pro IS významné po celou dobu jeho životního cyklu, ovšem na počátku jeho tvorby je jejich význam nejvyšší, prakticky rozhodují o úspěchu či neúspěchu vývoje IS. Jejich sběr ve fázi údržby např. slouží jako zpětná vazba od uživatele a pomocí ní je IS udržován a dále vylepšován. Jelikož se náš příklad zabývá především analýzou IS, budeme se významem požadavků zabývat pouze z pohledu analýzy. Existuje mnoho druhů požadavků, které se dělí do dvou hlavních kategorií:

- Funkční požadavky
- Mimofunkční požadavky
    - Výkonnostní požadavky
    - Návrhové požadavky
    - Technologické požadavky
    - Odvozené požadavky
    - …

Funkční požadavky se zabývají čistě funkčností systému, popisují co má systém umět. Mimofunkční požadavky se týkají téměř všeho ostatního, tedy požadavků na výkon systému, jeho dostupnost, zabezpečení apod. (příslušné metriky pro daný požadavek jsou jeho součástí). Mohou se týkat podmínek, které mají být dodrženy při návrhu, jak softwarové, tak i hardwarové části systému.

My se dále budeme zabývat pouze funkčními požadavky, pro náš příklad analýzy jsou nejpodstatnější.

Sběr požadavků bychom měli provést velmi důkladně a opakovaně jej konzultovat se zákazníkem. Jedná se o iterativní proces, zákazník může postupně při opakovaných konzultacích upřesňovat svoji představu.

Požadavky zásadně ovlivňují správný vývoj IS, jsou-li požadavky specifikovány chybně či nepřesně, znamená to téměř jistě neúspěch projektu. Skrze požadavky se totiž

dovídáme co má systém umět, představují jednotku potřebné funkcionality nebo jiné vlastnosti systému. Spolu s procesním modelem pak slouží ke specifikaci případů užití, které jsou v dalších fázích životního cyklu IS implementovány. Bez správného procesního modelu a správně sepsaných požadavků je představa úspěšného vývoje IS holou utopií.

Při sběru požadavků se snažíme nic dopředu nepředjímat, naslouchat zákazníkovi, zaznamenávat každý detail komunikace. Pokud existuje nejasné místo v požadavcích, je třeba se na něj dotázat a ne se pouze domnívat, jak se věci mají. Může se jednat o věc z našeho pohledu triviální a samozřejmou, ale zákazník na to může nahlížet naprosto opačně a může se jednat o kritickou záležitost pro správný chod systému. To, co je správné, vždy určuje zákazník. Samozřejmě, že můžeme navrhovat, doporučovat vysvětlovat nové přístupy, ale je třeba mít na paměti, že zákazník má poslední slovo. S tímto vědomím bychom měli přistupovat ke sběru požadavků, s maximální otevřeností a smyslem pro detail. V PD vytvoříme model požadavků volbou *File > New Model*. Jako typ modelu zvolíme *Requirments Model* a jako typ diagramu *Requirments document view*.

Požadavky pro náš hotelový IS jsou uvedeny v následující tabulce (Tabulka 1).

1 Funkční požadavky
1.1 Recepce
1.1.1 Rezervace
Host si může rezervovat pokoj. Při výběru pokoje postupuje dle těchto kritérií: počet lůžek, vybavenost, cena, datum pobytu. Je-li vybraný pokoj k dispozici v požadovaném termínu, host může provést rezervaci na jméno hosta buď prostřednictvím recepční a nebo prostřednictvím webové aplikace. Při rezervaci je automaticky založen též účet na jméno hosta. Rezervace je neplacená služba.
1.1.1.1 Prostřednictvím recepční
Host se dostaví na recepci, naváže kontakt s recepční a sdělí jí svá kritéria (viz 1.1.1. Rezervace) výběru pokoje. Je-li pokoj k dispozici, recepční provede rezervaci.
1.1.1.2 Prostřednictvím webové aplikace

Host si prostřednictvím webové aplikace vyhledá vhodný pokoj (kritéria výběru - viz 1.1.1. Rezervace). Je-li požadovaný pokoj k dispozici, host provede rezervaci.
1.1.2 Ubytování hosta
1.1.2.1 S rezervací
Jakmile se host s platnou rezervací dostaví na recepci, je ubytován. Rezervace je označena jako vyřízená, od tohoto okamžiku je hostovi účtován pobyt.
1.1.2.2 Bez rezervace
Host, který se dostaví na recepci s požadavkem na ubytování a nemá platnou rezervaci, se může recepční dotázat, zda-li není k nějaký pokoj dle jeho představ (kritéria výběru - viz 1.1.1. Rezervace) volný. Volný znamená, že v něm momentálně není ubytován žádný host a pokud je pokoj rezervován, tak tato rezervace nesmí kolidovat s předpokládaným pobytem hosta. Nalezne-li recepční volný pokoj dle představ hosta, je host ubytován. V opačném případě mu nelze vyhovět a ubytován není.
1.1.3 Odhlášení hosta
Po skončení pobytu se host na recepci odhlásí. Je mu vystaven účet za pobyt, který zahrnuje cenu pobytu.
1.2 Management
Management má za úkol úspěšně řídit hotel, je tedy nutné, aby k tomu systém poskytoval podklady. Na základě těchto podkladů musí být možné upravit ceny.
1.2.1 Statistiky
Výstup ve formě statistik poskytuje přehled o vytíženosti hotelu, v jakých obdobích je hotel zcela obsazen a naopak kdy je poptávka slabší.
1.2.2 Úprava cen
Na základě statistik management hotelu upravuje ceny ubytování pro konkrétní období.

Tabulka 1: Požadavky IS hotelu.

Výsledný diagram případů užití

Nyní, když víme, jak vypadá procesní model a model požadavků, můžeme se detailněji zabývat jednotlivými případy užití. Pomocí procesního modelu (viz Obr. 12) jsme identifikovali následující případy užití:

- *UC1 Vytvořit rezervaci*
- *UC2 Zrušit rezervaci*
- *UC3 Ubytovat*
- *UC4 Odhlásit*
- *UC5 Statistiky*
- *UC6 Úprava cen*

Jedná se o tzv. hraniční případy užití. Jsou to ty případy užití, které jsou spouštěny zvenčí procesem podniku. Ovšem proces jako takový, je ryze abstraktní pojem. Ve skutečnosti, fyzicky, je případ užití spouštěn jiným člověkem nebo strojem – aktérem. Můžeme tedy říct, že proces spouští případ užití IS vždy prostřednictvím nějakého aktéra.

Problémům, které mohou vzniknout při hledání případů užití prostřednictvím aktérů jsme se již krátce věnovali a proto jsme tuto metodu nepoužili. Místo toho jsme k nalezení hraničních případů užití využili procesní model hotelu, což je metoda o mnoho spolehlivější.

Nicméně, protože proces spouští případ užití vždy právě skrze aktéra, je možné aktéry využít k ověření správnosti našeho postupu. Postup při takovém ověření je následující:

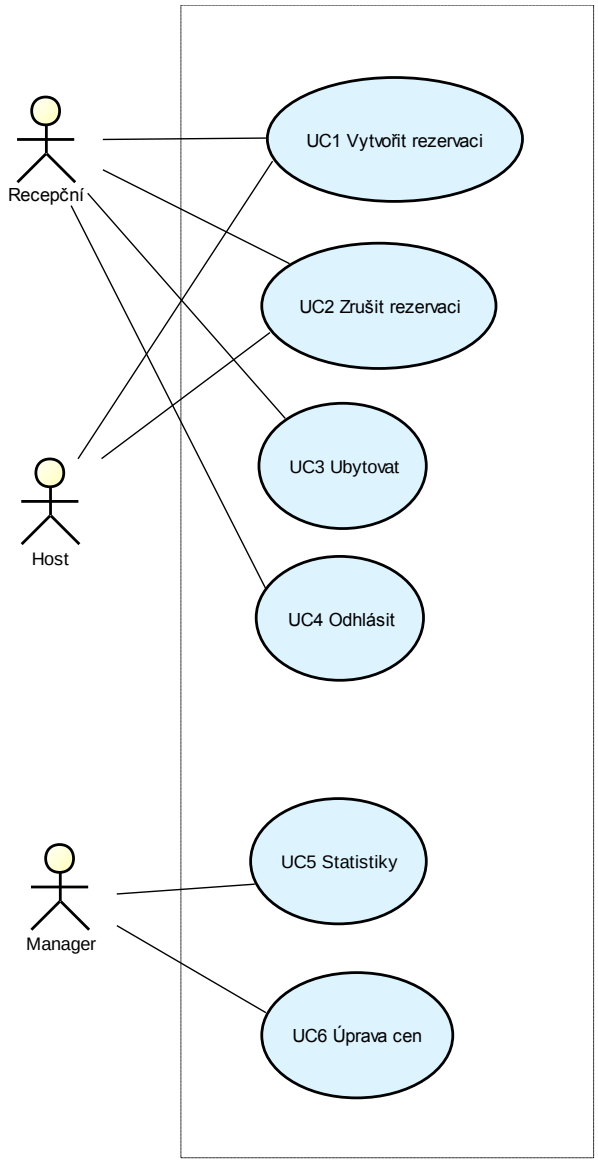
- Nalezneme všechny aktéry.
- Mezi případy užití určenými pomocí procesního modelu nalezneme ty případy užití, které aktéři spouštějí.
- Ověříme, zda-li je vše v souladu s modelem požadavků.

Při hledání aktérů a případů užití, které spouštějí, využíváme především model požadavků. Výsledné případy užití by měli odpovídat těm, které jsme nalezli pomocí procesního modelu. Samozřejmě, že se drobné odlišnosti mohou vyskytnout, důležité je, jestli se ve své podstatě shodují. Tuto kontrolu provádíme hlavně proto, abychom se

ujistili, že náš procesní model (a tím i nalezené případy užití) dává smysl. Mimo to, nalezení aktérů se hodí též při komunikaci se zákazníkem. Aktéři jsou též důležití i při implementaci IS (kvůli uživatelskému rozhraní apod.).

Z modelu požadavků vyplývá, že IS hotelu komunikuje se třemi aktéry. Jsou to Recepční, host a manažer hotelu. Tyto aktéry v souladu s požadavky přiřadíme k jednotlivým případům užití, které jsme nalezli pomocí procesního modelu a tím si ověříme jejich smysl. Viz diagram na Obr. 13.

Nový diagram případů užití v prostředí PD vytvoříme volbou *File > New Model*. Jako typ modelu zvolíme *Object-Oriented Model* a jako typ diagramu *Use Case Diagram*.



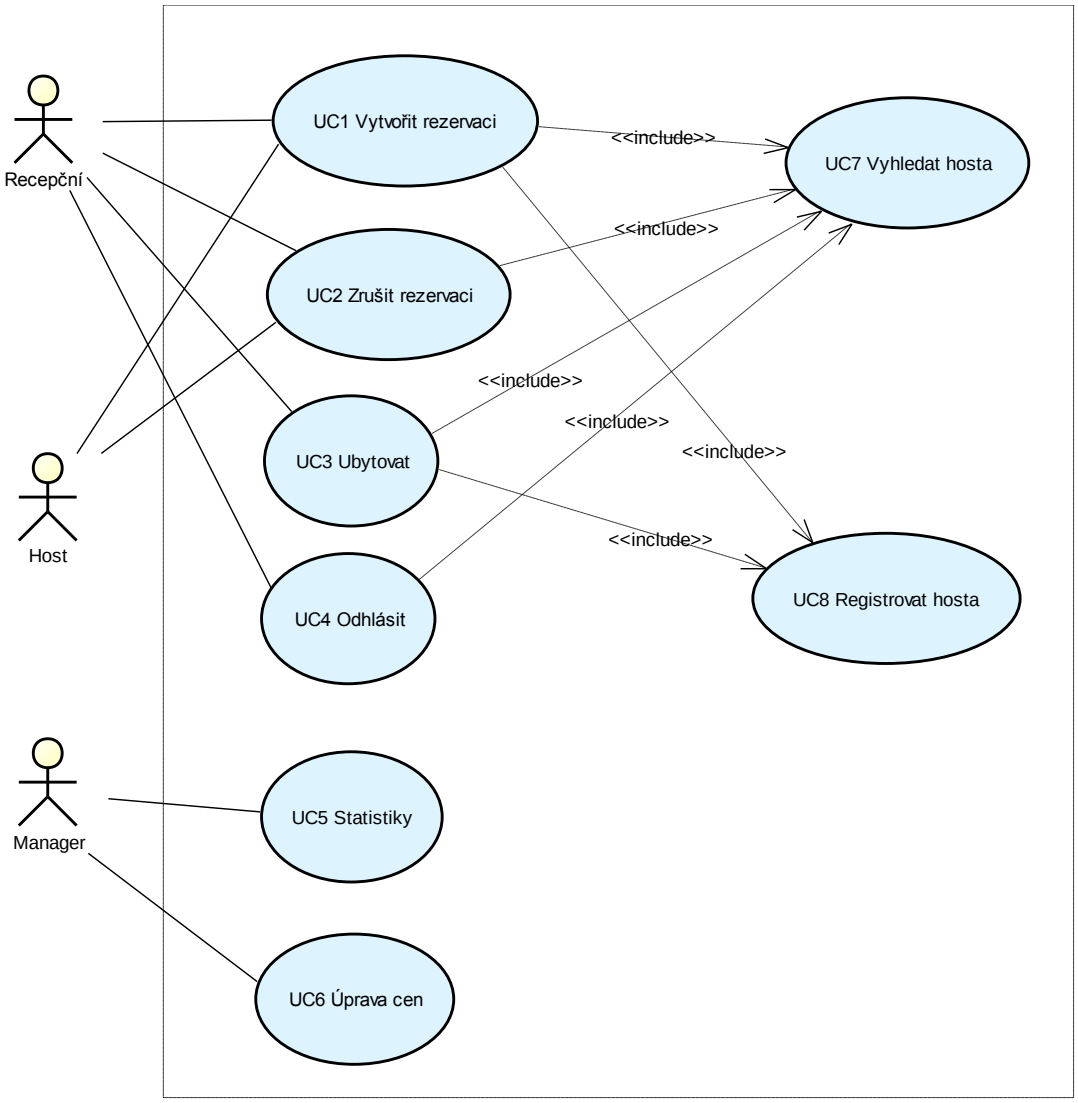
Obr. 13: Diagram případů užití.

Nyní se můžeme zaměřit na to, co se odehrává v případech užití samotných. Případy užití, které jsme nalezli, jsou tzv. hraničními případy užití. Jsou to ty případy užití, jež jsou v kontaktu s okolím IS. Ne všechny případy užití ale musí komunikovat přímo s okolím IS, místo toho mohou komunikovat (nebo být spouštěny, být součástí) s jiným případem užití. Tyto tzv. vnitřní případy užití, vznikají rozložením či rozšířením hraničních případů užití a umožnují nám všechny případy užití strukturovat a lépe spravovat. Pro popis možných vazeb mezi případy užití viz příloha A.

Ukažme si to na našem příkladu. Vezmeme-li v úvahu např. případy užití *UC2 Zrušit rezervaci*, *UC3 Ubytovat* a *UC4 Odhlásit*, ve všech případech budeme muset vyhledat hostav IS hotelu. Dokonce i v případě užití *UC1 Vytvořit rezervaci*, budeme muset hosta vyhledat, pokud již byl někdy v minulosti v hotelu registrován. Pokud bychom nechali vše beze změny, došlo by k tomu, že by čtyři případy užití obsahovaly jednu a tu samou činnost. Je nesmyslné implementovat stejnou funkčnost vícekrát. Vytvoříme tedy nový případ užití *UC7 Vyhledat hosta* a pomocí vazby *include* ho později zahrneme do všech čtyř případů užití. Podobný případ může nastat při registraci hosta. K registraci totiž může dojít při tvorbě rezervace, ale taky při ubytování hosta „z ulice“, který si nic dopředu nerezervoval a v hotelu nikdy předtím ubytován nebyl. Konečný výčet případů užití pro náš hotelový IS vypadá takto:

- *UC1 Vytvořit rezervaci*
- *UC2 Zrušit rezervaci*
- *UC3 Ubytovat*
- *UC4 Odhlásit*
- *UC5 Statistiky*
- *UC6 Úprava cen*
- *UC7 Vyhledat hosta*
- *UC8 Registrovat hosta*

Na Obr. 14 je zobrazena konečná podoba diagramu případů užití.



Obr. 14: Konečná podoba diagramu případů užití.

Scénář a diagram aktivit

Každý případ užití by měl být popsán pomocí tzv. scénáře (viz příloha A). Ve výjimečně složitých případech ho můžeme doplnit o diagram aktivit, jenž je přeci jenom názornější a člověk si udělá lepší představu o tom, co se v případě užití odehrává. Ovšem scénář jako takový, který zachytí veškeré detaily, by neměl chybět. Nicméně je třeba podotknout, že ve velké většině případů postačuje popis případu užití pomocí scénáře a je-li scénář až příliš složitý, měli bychom zvážit rozložení případu užití na jednodušší celky (ale pouze zvážit, nemusí to být vhodné).

Použití scénáře při popisu si ukážeme na případech užití UC1 Vytvořit rezervaci a UC8 Registrovat hosta (viz Tabulka 2 a Tabulka 3). Abychom si vyzkoušeli i použití diagramu aktivit, oba související případy užití si pomocí něj též popíšeme.

V PD přidáme scénář k jednotlivým případům užití tak, že v diagramu případů užití, otevřeme dvojklikem na konkrétní případ užití okno s jeho vlastnostmi. Na kartě *Specification* pak můžeme popsat scénář případu užití (viz Obr. 15).

Název případu užití:
Vytvořit rezervaci
Stručný popis:
Vytvořit rezervaci pokoje na přání hosta.
ID:
UC1
Scénář:
1. Zadání požadavků na rezervaci pokoje (co: parametry pokoje a kdy: termín ubytování) 2. IS vyhledá požadovaný pokoj v daném termínu. 3. KDYŽ existuje volný pokoj s požadovanými parametry v daném termínu: 3.1. [INCLUDE UC7 Vyhledat hosta] 3.2. KDYŽ host není registrován v hotelovém IS: 3.2.1. [INCLUDE UC8 Registrovat hosta] 3.3. IS provede rezervaci pokoje. 4. JINAK 4.1. IS informuje o nemožnosti rezervovat požadovaný pokoj.

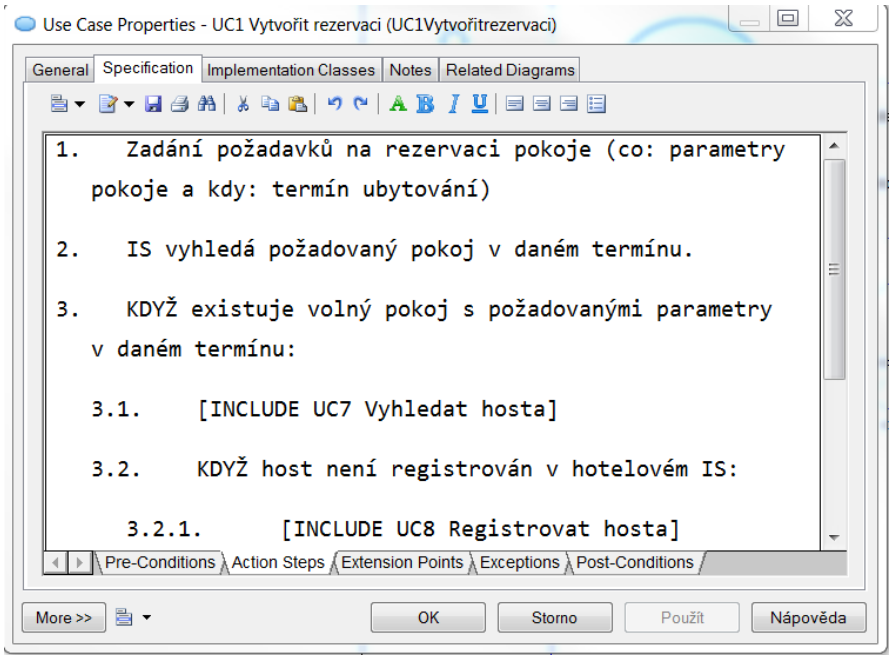
Tabulka 2: Scénář případu užití UC1 Vytvořit rezervaci.

Název případu užití:
Registrovat hosta
Stručný popis:
Registrovat hosta v IS hotelu.
ID:
UC8
Scénář:
1. IS požaduje základní údaje o hostovi 2. Host předá základní údaje 3. IS zaregistruje hosta 4. IS založí hostovi účet

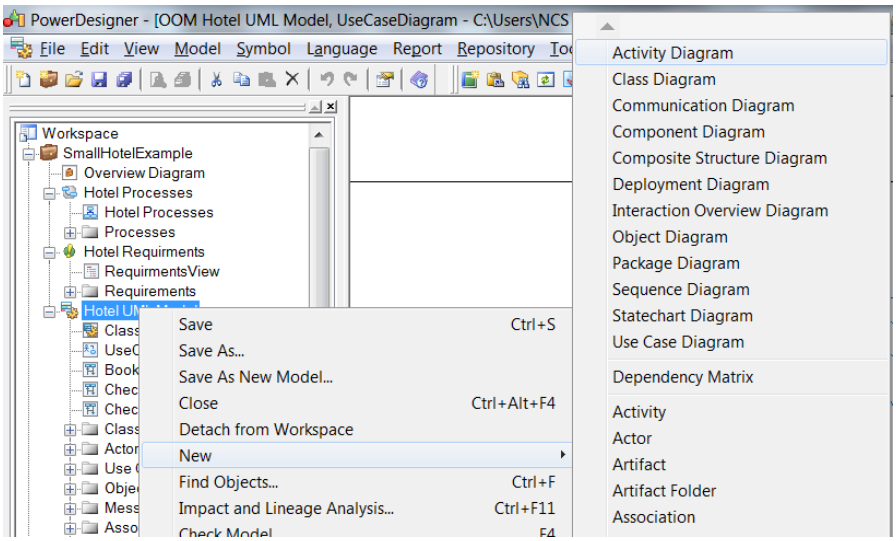
Tabulka 3: Scénář případu užití UC8 Registrovat hosta.

Ve většině případů si lze vystačit s popisem případů užití pomocí scénáře. Za pozornost stojí realizace vazby *include* mezi případy užití v textu scénáře (např. Tabulka 2 bod 3.2.1). Tento zápis chápeme tak, že se na konkrétním místě ve scénáři začne vykonávat vázaný případ užití a po jeho skončení se pokračuje v původním scénáři. Na Obr. 17 a Obr. 18 jsou zobrazeny diagramy aktivit pro případy užití *UC1 Vytvořit registraci* a *UC8 Registrovat hosta*. Zde je vazba *include* realizována ještě názorněji, diagram na Obr. 17 představuje popis aktivity *registrovat hosta* diagramu na Obr. 18.

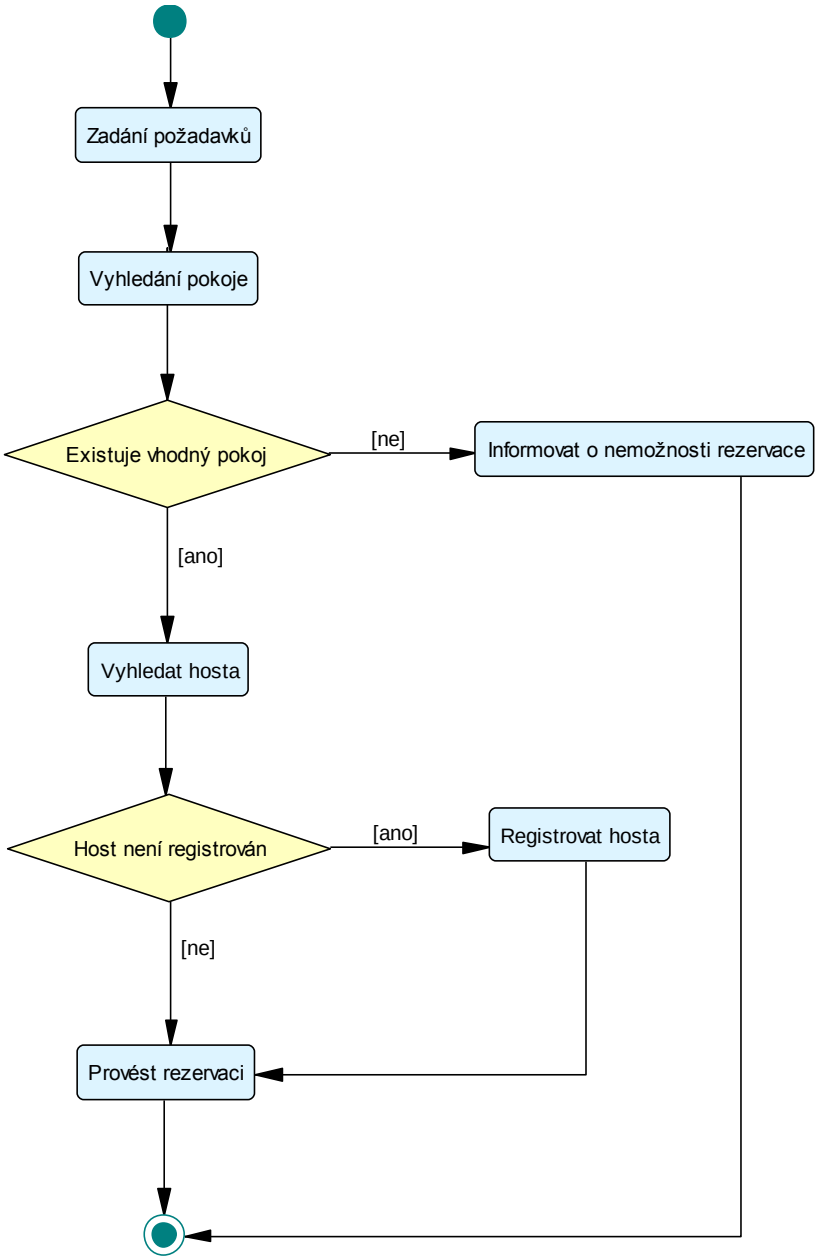
Diagram aktivit v PD vytvoříme v rámci již exitujícího UML modelu (vytvořili jsme ho spolu diagramem případů užití). V levém navigačním okně klikneme pravým tlačítkem na UML Model a zvolíme *New* > *Activity Diagram*. Viz Obr. 16. Pokud chceme provázat aktivitu s jiným diagramem (abychom vyjádřili vztah mezi případy užití *include*), klikneme v diagramu aktivit pravým tlačítkem na aktivitu a zvolíme *Related Diagram* > *New*. Takto vytvoříme nový diagram, který popisuje danou aktivitu.



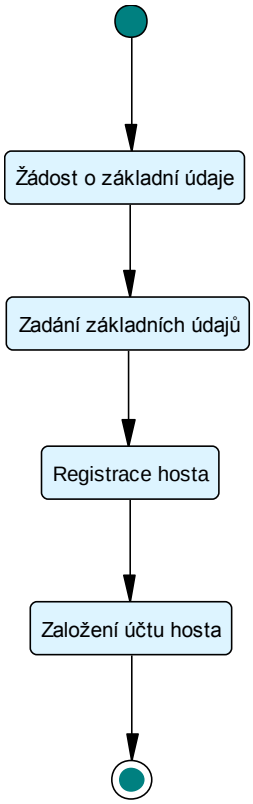
Obr. 15: Popis scénáře případu užití v prostředí PD.



Obr. 16: Tvorba diagramu aktivit v prostředí PD.



Obr. 17: Diagram aktivit případu užití UC1 Vytvořit registraci.



Obr. 18: Diagram aktivit případu užití UC8 Registrovat hosta.

6.2 Diagram tříd

Diagram tříd je jedním z vůbec nejdůležitějších diagramů ve vývoji IS. Dokáže pojmout mnoho podstatných informací o systému naráz a bez pohledu na něj se při vývoji IS jen těžko můžeme obejít. Jelikož se (jak z názvu vyplývá) zabývá třídami, nikoliv jejich instancemi, poskytuje nám statický pohled na systém. Diagram tříd též můžeme chápat jako jakýsi předpis, vyznačené mantinely pro vznik objektů v IS. To ale nastává až za běhu systému, proto chování objektů, jejich vznik, zánik a jejich vzájemnou komunikaci označujeme za děje dynamické, kterými se už zabývají jiné diagramy (např. sekvenční diagram nebo diagram aktivit).

V analytické fázi slouží diagram tříd k popisu hlavních prvků (které abstrahujeme pomocí tříd), z nichž se IS bude skládat. Diagram také zaznamenává, jak budou tyto prvky vzájemně provázány, strukturovány. Ve fázi návrhu má diagram tříd též svou roli, zabývá se již konkrétním technologickým návrhem. Z jedné třídy ve fázi analýzy mohou ve fázi návrhu např. vzniknout dvě, upraveny mohou být i vazby mezi třídami apod. Diagram tříd, který vznikne během analytické fáze, je ve fázi návrhu dále

zjemňován, zpřesňován, doplňován o další detaily tak, aby IS co nejlépe odpovídal zadání (my se ale zabýváme pouze analýzou IS).

6.2.1 Vyhledání tříd

Při vytváření diagramu tříd, musíme ze všeho nejdříve vyhledat jednotlivé třídy. Vycházíme při tom z modelu požadavků resp. scénářů případů užití IS. Při analýze textu se zaměříme na podstatná jména a slovesa. Podstatná jména mohou představovat prvek (objekt z reálného světa), který pomocí třídy abstrahujeme, slovesa pak mohou představovat vazby mezi nimi, např. věta v požadavku 1.1.1: „Host si může rezervovat pokoj". Toto je ovšem pouze základní pomůcka. Pro IS zásadní třídy v textu vůbec být zmíněny nemusí, přímo či nepřímo z něj mohou vyplývat. Nazývají se tzv. skrytými třídami.

Při určování atributů a operací dbáme na to, aby odpovídaly, požadavkům na IS a případům užití. Atributy popisují vlastnosti modelovaného objektu. Skrze operace IS naplňuje požadavky na něj kladené, což tedy jeho hlavní smysl existence. A proto bychom měli smyslu a členění operací mezi jednotlivé třídy věnovat zvýšenou pozornost.

Na druhou stranu by analytické třídy neměly obsahovat příliš mnoho atributů a operací a neměly by se zabývat naprosto všemi důsledky. Analytický diagram tříd slouží k zachycení a ověření hlavní myšlenky IS. Detailní rozpracování je pak otázkou návrhové fáze.

Dalším pravidlem, kterého se při hledání tříd snažíme držet, je to, že nejdříve vyhledáváme konkrétní třídy. Abstraktní třídy a rozhraní zavádíme, až když jsme si naprosto jisti, že máme hotový základní model. Stejně tak přistupujeme ke generalizaci. Nejdříve je třeba znát všechny třídy, jež systém obsahuje a až poté je možné s nimi dále pracovat (např. zavedení dědičnosti apod.). Častou chybou bývá, že se snažíme např. nejdříve navrhnout dědění a až poté hledáme konkrétní třídy. Děje se tak, protože už máme v mysli hrubou představu o třídách a chceme ji dále vylepšovat. To je ale nebezpečná chyba, velmi často se totiž stává, že poté, co „si to hodíme na papír“, objevíme nové souvislosti, které mohou zásadně měnit náš původní pohled na věc. Proto je lepší nejdříve pečlivě vyhledat konkrétní třídy a až když jsme si jisti, že odpovídají potřebám IS, přikročit k pokročilejším principům typu dědičnost.

Při analýze se snažíme, aby diagram nebyl příliš složitý, přílišná granularita a velký počet tříd velmi znepřehledňuje celý diagram. O co složitější náš návrh bude ve fázi analýzy, tím složitější bude další vývoj IS. Náš model nikdy nebude dokonalý (ve vztahu k realitě), na to nesmíme zapomínat. Důležité je, aby plně odpovídal potřebám našeho IS a aby zároveň neznemožnil jeho pozdější rozvoj.

V tomto duchu přikročíme ke hledání tříd i u našeho hotelového IS. Nejprve se zaměříme na hledání konkrétních tříd a až poté hledáme asociační a generalizační vazby mezi nimi. Využíváme při tom jak případů užití, tak modelu požadavků. Základní třídy hotelového IS budou:

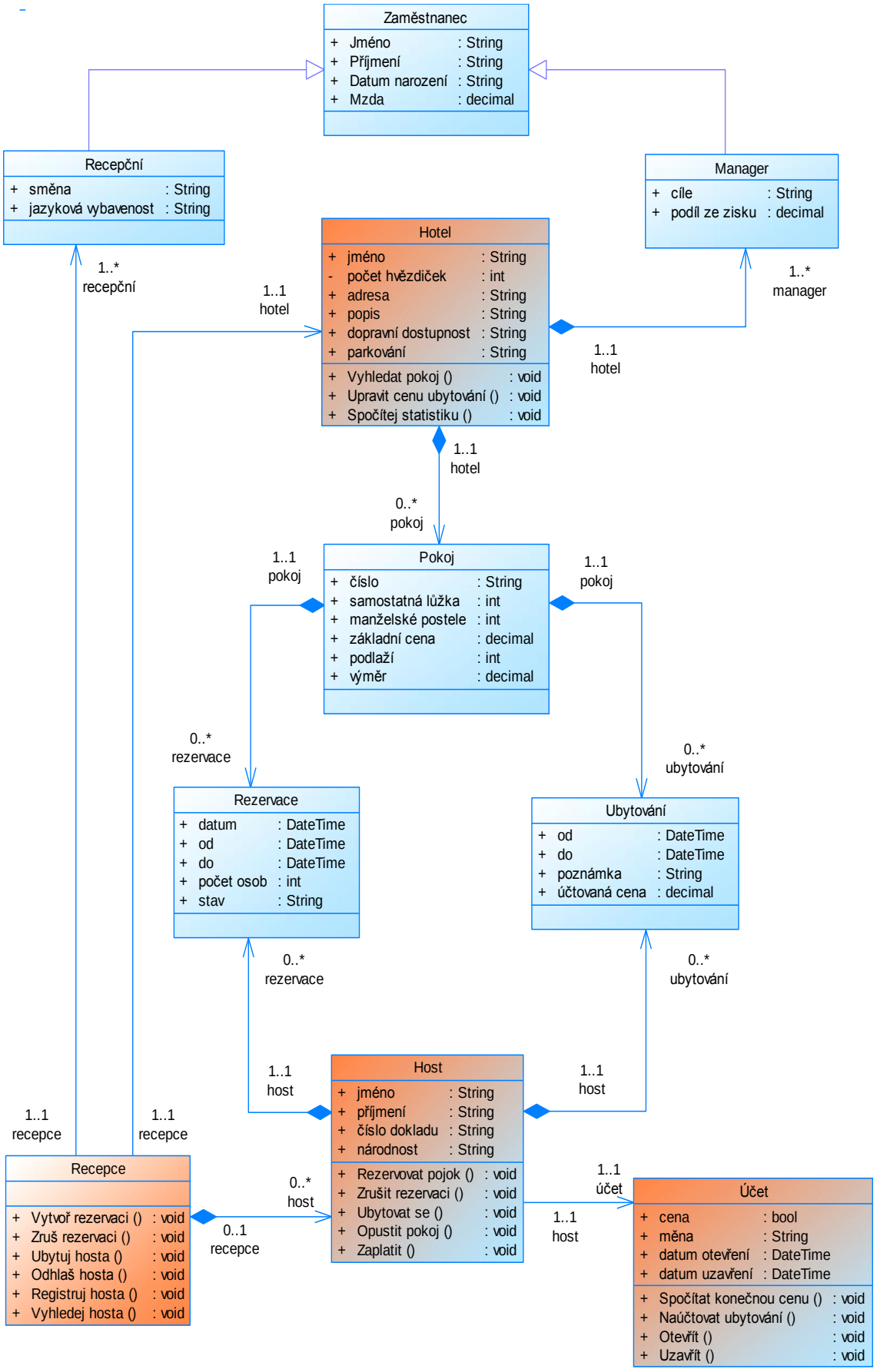
- *Pokoj*
- *Host*
- *Účet*
- *Hotel*
- *Recepce*
- *Recepční*
- *Manažer*

Postupným tříbením dospějeme až ke konečné podobě diagramu tříd na Obr. 19. Důležité je se držet zásad zmiňovaných v předešlých odstavcích, začít od základních tříd a diagram postupně zpřesňovat. Za povšimnutí stojí především zavedená generalizace v podobě třídy *Zaměstnanec* a vazební třídy *Rezervace* a *Ubytování*, které vyjadřují dva možné vztahy mezi třídami *Pokoj* a *Host*. Nabízelo by se třídy *Rezervace* a *Ubytování* sjednotit do jedné třídy, jelikož mají podobné atributy i funkci. Rozdíl mezi samotnou rezervací a ubytováním bychom pak rozlišovali pomocí stavového atributu. Rezervace je ovšem v našem hotelu neplacená služba, což se o ubytování říci nedá. Z tohoto důvodu je vedení tříd *Rezervace* a *Ubytování* zvlášť „čistějším“ řešením.

Dále si popišme vztah mezi třídami *Hotel* a *Pokoj*. Jedná se o jednosměrnou kompozici. *Hotel* může mít až N pokojů, *Pokoj* však může patřit vždy pouze do jednoho hotelu. Jedná se o kompozici, protože pokud by hotel zanikl, pokoje samy o sobě nemají žádný smysl (ve smyslu hotelových pokojů) a zanikly by taktéž. A nakonec, tento vztah je jednosměrný protože z analytického hlediska nemá žádný smysl, aby pokoj „měl ponětí“ o svém hotelu, ale hotel samozřejmě o svých pokojích „mít přehled“ musí.

Zastavme se ještě u volby atributů třídy. Je velmi důležité nezavádět zbytečné atributy, které nikdy nevyužijeme, naopak bychom neměli pominout ty podstatné. To, jaké atributy zvolíme při modelování, bude vymezovat stavový prostor třídy, její vlastnosti.

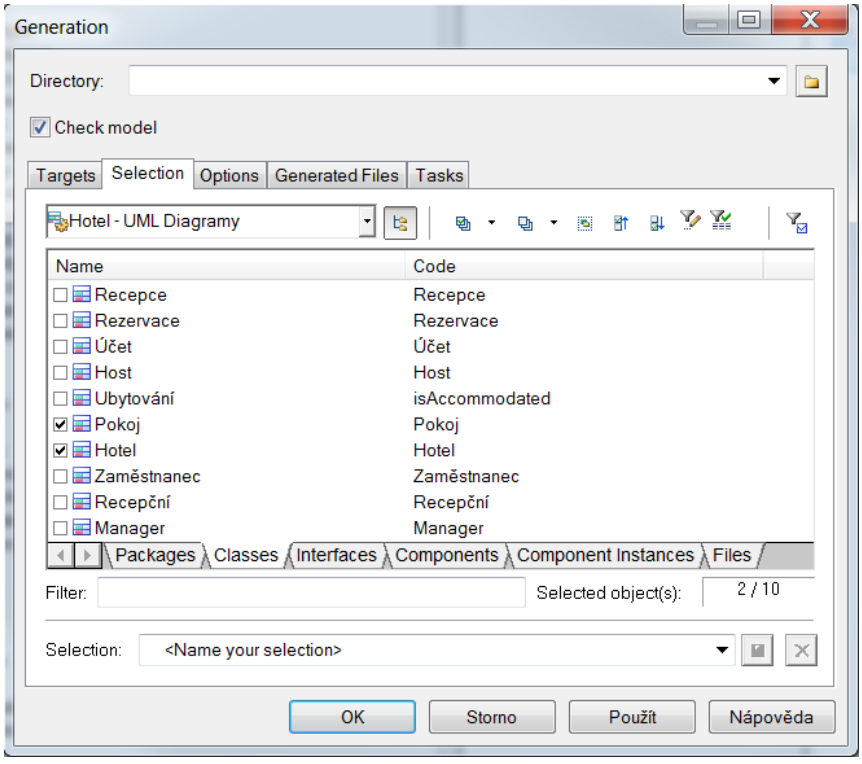
Diagram tříd vytvoříme v PD, stejně jako diagram aktivit, v rámci již existujícího UML modelu. V levém navigačním okně klikneme pravým tlačítkem na UML Model a zvolíme *New > Class Diagram*.



Obr. 19: Konečná podoba diagramu tříd.

Ukázka dopředného inženýrství

Náš příklad se sice zabývá analýzou IS, krátce ale odbočíme k dalším vývojovým fázím IS, k návrhu a implementaci. Velmi zajímavou možností, jež PD poskytuje, je možnost tzv. dopředného inženýrství. Jedná se o to, že z návrhového modelu (diagram tříd) uloženém v PD lze přímo generovat „kostru“ implementačního kódu. Vygenerovány jsou třídy s atributy, s metodami a se všemi vzájemnými vazbami. Implementovat pak zbývá těla metod, to je však již úkol pro programátora. Implementační kód můžeme generovat pro všechny hlavní objektové jazyky (Java, C# a další). Volbu jazyka pro náš model provádíme při zakládání projektu v PD, změnit ji můžeme ale i kdykoliv později. Provedeme to takto, zvolíme *Language > Change Current Object Model Language.* Ve stejné nabídce pak můžeme provést samotné generování kódu (*Language > Generate Code*). Odpovídající dialog je zobrazen na Obr. 20. Po nastavení všech vlastností již dojde k vytvoření projektu pro konkrétní vývojové prostředí (dle volby jazyka), ve kterém můžeme pokračovat v implementaci.



Obr. 20: Dialog pro generování implementačního kódu.

Následuje příklad vygenerovaného kódu pro třídu *Hotel*. Těla metod nejsou implementována. Doplněny byly pouze základní metody pro obsluhu kolekcí.

```
// File:    Hotel.cs
// Created: 26. listopadu 2012 12:19:41
// Purpose: Definition of Class Hotel

using System;

public class Hotel
{
   private int pocetHvezdicek;

   public void VyhledatPokoj()
   {
      throw new NotImplementedException();
   }

   public void UpravitCenuUbytování()
   {
      throw new NotImplementedException();
   }

   public void SpočítejStatistiku()
   {
      throw new NotImplementedException();
   }

   public String jmeno;
   public String adresa;
   public String popis;
   public String dopravniDostupnost;
   public String parkovani;

   public System.Collections.ArrayList pokoj;

   /// <summary>
   /// Property for collection of Pokoj
   /// </summary>
   /// <pdGenerated>Default opposite class collection property</pdGenerated>
   public System.Collections.ArrayList Pokoj
   {
      get
      {
         if (pokoj == null)
            pokoj = new System.Collections.ArrayList();
         return pokoj;
      }
      set
      {
         RemoveAllPokoj();
         if (value != null)
         {
            foreach (Pokoj oPokoj in value)
               AddPokoj(oPokoj);
         }
      }
   }

   /// <summary>
```

```
   /// Add a new Pokoj in the collection
   /// </summary>
   /// <pdGenerated>Default Add</pdGenerated>
   public void AddPokoj(Pokoj newPokoj)
   {
      if (newPokoj == null)
         return;
      if (this.pokoj == null)
         this.pokoj = new System.Collections.ArrayList();
      if (!this.pokoj.Contains(newPokoj))
         this.pokoj.Add(newPokoj);
   }

   /// <summary>
   /// Remove an existing Pokoj from the collection
   /// </summary>
   /// <pdGenerated>Default Remove</pdGenerated>
   public void RemovePokoj(Pokoj oldPokoj)
   {
      if (oldPokoj == null)
         return;
      if (this.pokoj != null)
         if (this.pokoj.Contains(oldPokoj))
            this.pokoj.Remove(oldPokoj);
   }

   /// <summary>
   /// Remove all instances of Pokoj from the collection
   /// </summary>
   /// <pdGenerated>Default removeAll</pdGenerated>
   public void RemoveAllPokoj()
   {
      if (pokoj != null)
         pokoj.Clear();
   }
}
```

6.3 Sekvenční diagram

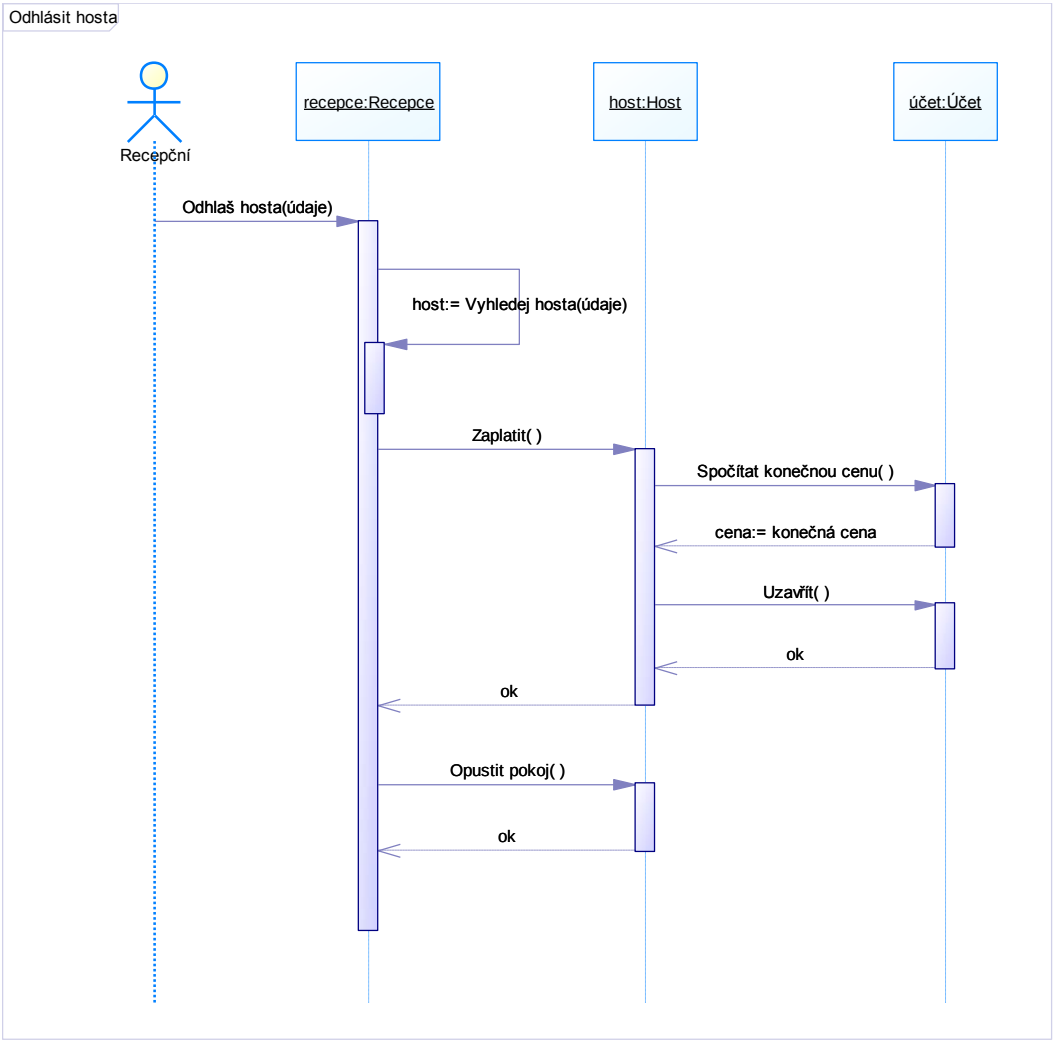
Sekvenční diagramy slouží především k popsání komunikace mezi objekty. Tato komunikace probíhá za různých podmínek za běhu systému, je to dynamický děj (již samotný vznik objektu je dynamickým dějem, natož pak komunikace mezi vícero objekty).

U našeho příkladu IS, ale mělo by to platit i obecně, využijeme sekvenční diagramy pouze pro ty nejsložitější případy komunikace. U jednodušších případů je to zbytečné. Snažíme se, aby sekvenční diagram popisoval komunikace v rámci jednoho případu užití. Takový přístup nám umožní zkontrolovat, zda jsme správně provedli dekompozici operací mezi jednotlivé třídy v diagramu tříd tak, abychom co nejlépe naplnily scénář

konkrétního případu užití. Je např. daleko lepší mít více jednoduchých metod v různých třídách, které spolu vzájemně komunikují, přispívá to k lepší čitelnosti a udržovatelnosti systému. Toto se netýká pouze operací, ale celého diagramu tříd. Z toho, jak probíhá komunikace mezi jednotlivými objekty, poznáme, zda je náš diagram tříd správný, zda má ta či ona vazba smysl apod. Toto je největší přínos sekvenčních diagramů, přináší nový pohled na systém, vnáší světlo do složitějších dynamických dějů.

My pomocí sekvenčního diagramu popíšeme případ užití *UC4 Odhlásit*, který se zbývá odhlášením hosta na konci pobytu. Sekvenční diagram pro tento případ užití bude vypadat následovně (viz Obr. 21).

Je-li výsledný sekvenční diagram až příliš složitý, např. obsahuje-li mnoho cyklů a větvení, přestává být čitelný a tím se ztrácí i jeho smysl. Vzniká prostor pro možnou chybu, napříč vývojovými fázemi IS. V takovém případě bychom měli zvážit buď rozložení případu užití na více menších, nebo pro zobrazení použít více sekvenčních diagramů. Sekvenční diagram vytvoříme v PD, stejně jako diagram aktivit, v rámci již existujícího UML modelu. V levém navigačním okně klikneme pravým tlačítkem na UML Model a zvolíme *New > Sequence Diagram.* Tím, že jsme vytvořili sekvenční diagram ve stejném modelu jako diagram tříd, dojde automaticky k jejich provázání a v sekvenčním diagramu lze pracovat přímo s objekty a třídami, které jsou definovány v diagramu tříd.



Obr. 21: Sekvenční diagram pro UC4 Odhlásit.

6.4 Datový model

Aby náš hotelový IS mohl správně fungovat, chybí mu jedna podstatná věc – datová persistence. Dosud jsme se při naší analýze zabývali de facto algoritmy prováděnými nad daty, nikoliv samotným datovým modelem. V této kapitole se budeme zabývat:

- Konceptuálním datovým modelem
- Fyzickým datovým modelem

Využijeme při tom možností PD pro práci s těmito modely.

Při návrhu IS jsme využili objektového přístupu, což je dnes široce přijímaný standard. Nyní ale nastává problém – jak tyto objekty uložit do databáze. Kdybychom používali čistě objektové databáze, bylo by to snadné – objekty bychom do nich přímo zapisovali.

Objektové databáze sice existují, nicméně se z mnoha důvodů zatím neujaly. Relační databáze jsou naopak v současnosti hlavní databázovou technologií. Díky své spolehlivosti, letům vývoje, odladěnosti a hlavně jejich výkonu jsou relační databáze stále v nejlepší formě.

Jak tedy tento konflikt objektového a relačního světa vyřešit? Musíme provést tzv. objektově relační mapování. Objekty, které chceme uložit do databáze, nazýváme perzistentní. Abychom je mohli do databáze uložit a zachovat jejich vlastnosti a vazby mezi nimi, musíme pro ně vytvořit vhodný datový model. To je hlavním smyslem datového modelování.

Nejdříve vytvoříme ERA model. Jedná se o logický datový model, jenž je odstíněn od technologických a implementačních specialit jednotlivých databází. Než přikročíme k tvorbě fyzického datového modelu (který je již svázán s konkrétní databázovou technologií), můžeme konceptuální model libovolně upravovat. V našem případě přistoupíme k zavedení tzv. systémových primárních klíčů, aby fungování datového modelu bylo nezávislé na hodnotách ukládaných dat.

Pří tvorbě konceptuálního modelu využijeme jednu z hlavních výhod, jakou nám PD nabízí. Díky sdílenému slovníku (repository) je možno konceptuální datový model generovat z diagramu tříd (viz Obr. 22). Z tříd, označených jako perzistentní (perzistenci lze určit ve vlastnostech třídy), vznikají entity. Při převodu třídy na entitu jsou brány v potaz pouze atributy, protože vyjadřují stav objektu. Generování provedeme následovně. Otevřeme diagram tříd a zvolíme *Tools* > *Generate Conceptual Data Model*.

Než přikročíme k tvorbě fyzického datového modelu (který již tvoříme pro konkrétní databázi), je třeba provést několik nezbytných úprav (s modelem nemusíme být vždy na 100% spokojeni). Jak už bylo zmíněno, zavedeme systémové primární klíče do každé entity. Tím zajistíme nezávislost a libovolnou modifikovatelnost dat v budoucnu, aniž bychom narušili vazby mezi tabulkami.

6.4.1 Problematika volby primárního klíče

V objektovém modelu je asociační vazba realizována atributy. Tyto atributy obsahují referenci na konkrétní objekt dané třídy v paměti a tím je vazba realizována. Standardně je tento referenční atribut součástí zdrojové třídy a obsahuje referenci na objekt cílové třídy. Jméno tohoto atributu je pak obvykle shodné s jeho rolí, tzn. s označením třídy na kterou odkazuje. Toto platí pokud realizujeme vazbu 1..1, ale pokud bychom měli realizovat vazbu 1..*, musí být vazební atribut ve zdrojové třídě vektor, množina referenčních atributů, které odkazují na konkrétní objekty. Tomuto způsobu realizace vazby se říká **explicitní**.

V relačním modelu je naopak vazba realizována **implicitně.** Je vyjádřena tzv. cizími klíči. Jsou to primární klíče zdrojové tabulky, umístěné v cílové tabulce. Primární klíč je minimální množina atributů, jejichž hodnoty jednoznačně určují výskyt entity. Skládá-li se primární klíč z více atributů (tzv. složený klíč), není možné žádný atribut ze složeného klíče vyloučit, aniž bychom porušili podmínku jednoznačné identifikace entity.

Zásadní otázka při tvorbě datového modelu, která má dalekosáhlé implementační důsledky je volba primárních klíčů. Pomocí klíčů se implementují v relačním datovém modelu vazby mezi entitami, klíče udržují integritu dat na všech úrovních.

Možnosti volby primárního klíče:

- Volba přirozeného, významového atributu. Např. pro čtenáře jde o jeho identifikační číslo přidělené knihovnou, pro knihu je to její ISBN apod.
- Systémem generovaný, umělý klíč. Obvykle je to číslo přidělené databázovým systémem fyzickému záznamu v okamžiku jeho založení do databáze.

Výhody systémových klíčů:

- Jsou automaticky generované, stabilní po celou dobu existence relace. Vstupují do vazeb jako cizí klíče, vytvářejí stabilní referenční integritu.
- Celočíselný formát cizích klíčů je nejvhodnější formát pro vytváření indexů, rychlý výběr záznamů.

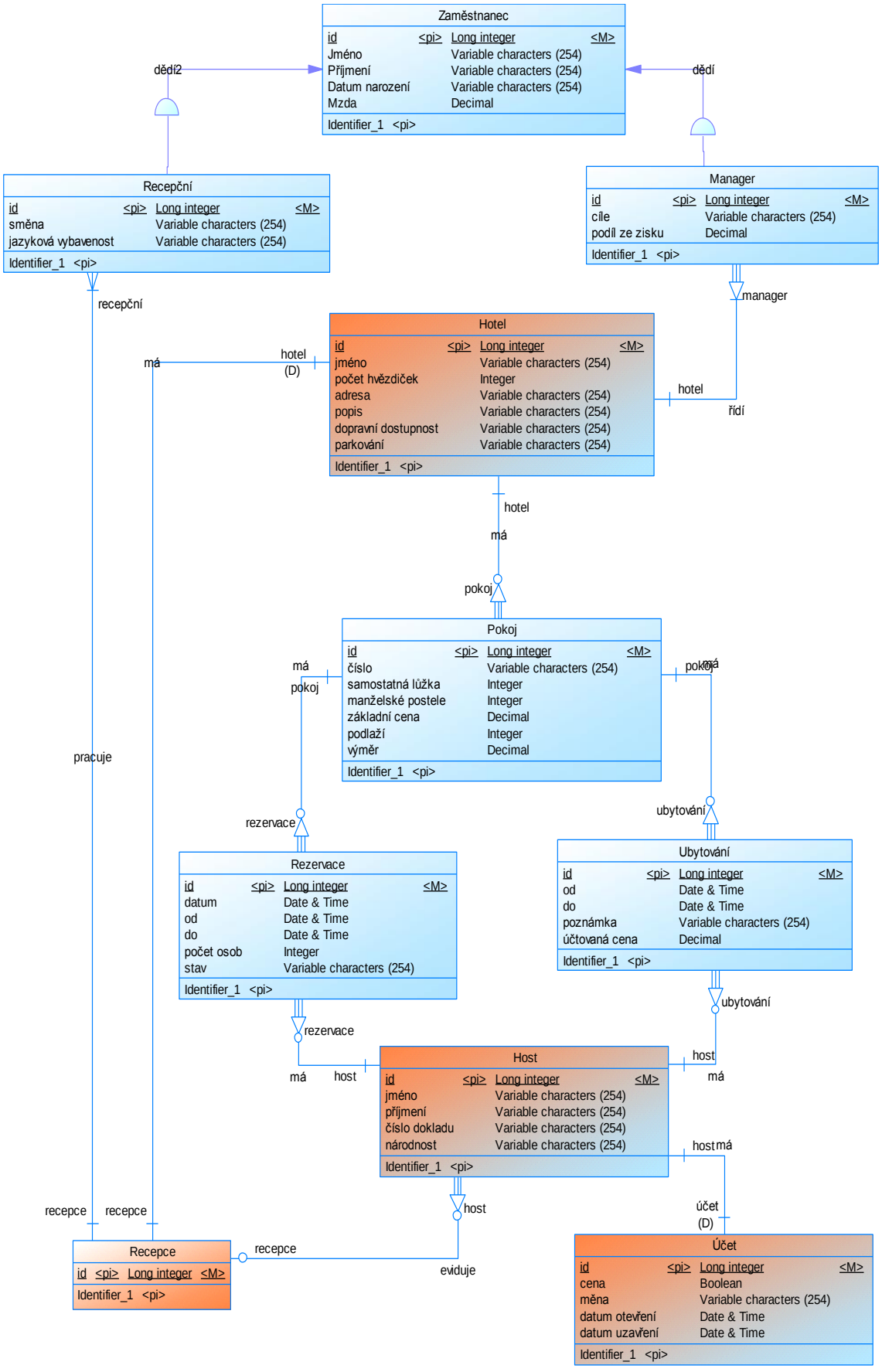
Nevýhody systémových klíčů:

- V závislých tabulkách není uložena žádná informace (např. číslo čtenáře). Pro získání informace je třeba složitějšího dotazu, který pracuje s více tabulkami.

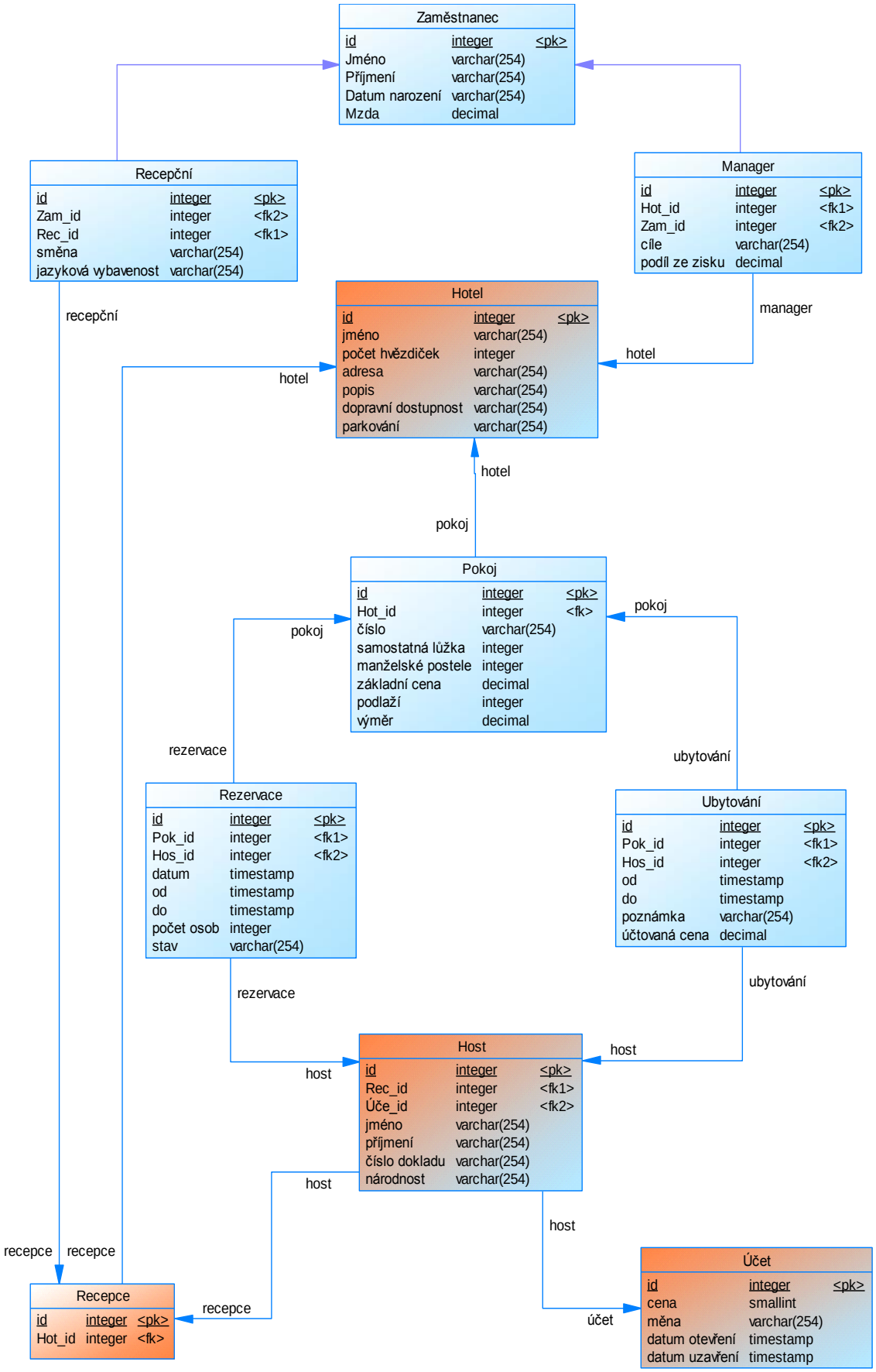
Nevýhody přirozených klíčů:

- Zásadní problémy s datovým modelem při vynucené změně klíče, např. přečíslování čtenářů v knihovně, sloučení dvou knihoven apod.
- Závislost tabulek vede ke složeným klíčům, problémy s indexováním.

Fyzický datový model vytvoříme takto. Zvolíme *Tools* > *Generate Physical Data Model*, vybereme databázi, pro kterou chceme model generovat a potvrdíme. Výsledný model je na Obr. 23. Model je generován pro SŘBD Sybase SQL Anywhere 12, vygenerovaný skript pro tvorbu databáze v tomto SŘBD je součástí přílohy B.



Obr. 22: Konceptuální datový model.



Obr. 23: Fyzický datový model.

7 Závěr

Cílem této práce bylo na relativně jednoduchém příkladu analýzy IS demonstrovat použití UML modelů, které spolu souvisí a vzájemně se doplňují. Abychom mohli naplno využít takového přístupu, je dnes již praktickou nutností využití některého z CASE nástrojů, který zajistí kontrolu konzistence a vzájemných vazeb mezi modely. A to nejen během analytické fáze vývoje IS, ale prakticky během celého jeho životního cyklu. V případě této práce byl zvolen nástroj Sybase PowerDesigner.

Zvolená doména příkladu – hotel – je všeobecně známa, ale zároveň není natolik složitá, aby to znemožnilo její srozumitelný popis během analýzy (diagramy lze přehledně zobrazit na A4). Většina učebnic a textů popisující analýzu IS vysvětlují význam a použití jednotlivých modelů na jednoduchých, ale samostatných příkladech, které spolu nesouvisí. Tím se ztrácí to podstatné z metodik využívajících CASE nástroje. V této práci spolu všechny modely od počátku až do konce analýzy souvisí, tvoří jeden konzistentní celek a tím vzrůstá jejich didaktická hodnota.

Po prvotním přehledu o CASE nástrojích následuje detailnější seznámení s problematikou UML, která je dále aplikována během celého příkladu analýzy. Během analýzy je pozornost nejprve věnována modelu požadavků a procesnímu prostředí, ve kterém bude IS působit. Z tohoto prostředí jsou následně odvozeny případy užití IS. Z případů užití dále vychází všechny ostatní UML modely (resp. diagramy), jako digram tříd, sekvenční diagramy a další. Naplno se projevily výhody použití CASE nástroje, jednotlivé modely nejsou ve vzájemném konfliktu a jakákoliv změna je snadno proveditelná v globálním měřítku. Závěrem byl vytvořen relační datový model, který vychází z analytických objektově orientovaných modelů a byl vygenerován implementační skript tohoto datového modelu pro konkrétní SŘBD Sybase SQL Anywhere 12.

Literatura

[1] BOOCH, Grady. The unified modeling language user guide. 2nd ed. Upper Saddle River: Addison-Wesley, 2005.

[2] KRAVAL, Ilja. Analytické modelování UML systémů v praxi. 1. vyd. Object Consulting s.r.o., 2010.

[3] ARLOW, Jim a Ila NEUSTADT. UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky. Vyd. 1. Computer Press, 2007.

[4] History of CASE. [online]. [cit. 2013-04-21]. Dostupné z: http://it.toolbox.com/wiki/index.php/History_of_CASE

[5] ŘEPA, Václav. Metodika vývoje informačního systému s pomocí nástroje PowerDesigner. [online]. [cit. 2013-04-21]. Dostupné z: http://www.sybase.cz/buxus/docs/Metodika_vyvoje_IS_06_2006.pdf

[6] Sybase PowerDesigner Documentation & Manual. [online]. [cit. 2013-04-21]. Dostupné z: http://www.sybase.com/products/modelingdevelopment/powerdesigner

[7] Sybase Power Designer, česká stránka produktu. [online]. [cit. 2013-04-21]. Dostupné z: http://www.sybase.cz/index.php?option=com_content&view=article&id=3&mid=24

[8] Server objektových technologií. [online]. [cit. 2013-04-21]. Dostupné z: http://objects.cz/

[9] UML Specification. [online]. [cit. 2013-04-21]. Dostupné z: http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/

[10] Object Managment Group Homepage. [online]. [cit. 2013-04-21]. Dostupné z: http://www.omg.org/

[11] UML Official Homepage. [online]. [cit. 2013-04-21]. Dostupné z: http://uml.org/

[12] Computer Aided Software Engineering Tools (CASE). [online]. [cit. 2013-04-21]. Dostupné z: http://www.c-sharpcorner.com/UploadFile/nipuntomar/computer-aided-software-engineering-tools-case/

Příloha A

Základy UML 2.0

Rozsah celého standardu UML je mírně řečeno značný (viz specifikace UML na [9]). Pokrývá celou řadu oblastí a zabývat se všemi jeho úskalími by bylo mimo smysl této práce. Tato podkapitola se tedy zbývá pouze základy UML, nezbytnými pro naši vzorovou úlohu.

Diagram tříd

Diagram tříd (Class Diagram) je jeden z nejpoužívanějších UML diagramů. Poskytuje pohled na statickou strukturu systému. Diagram tříd může obsahovat:

- Třídy
- Vazby
- Rozhraní

Třída

Třída je nejdůležitějším prvkem v objektově orientovaném návrhu. Představuje základní stavební prvek, jenž reprezentuje objekt či věc (ať už abstraktního nebo konkrétního charakteru) z reálného světa, jehož část náš objektově orientovaný návrh modeluje. Každá třída musí/může mít:

- Jméno
- Atributy
- Operace

Jméno

Jméno třída mít musí. Nejenže ji odlišuje od ostatních a činí ji v rámci návrhu unikátní, ale především vyjadřuje smysl této třídy a odkazuje na původní prvek z reálného světa, který abstrahuje.

Atributy

Atributy jsou volitelné. Představují vlastnosti reálného objektu, které jsme se rozhodli při modelování zohlednit. Atribut může být znázorněn pouze svým jménem, je ale možné uvést i datový typ, či počáteční hodnotu atributu.

Operace

Operace jsou též volitelné. Představují chování reálného objektu, které jsme se rozhodli modelovat. Podobně jako s atributy i operace může být znázorněna pouze svým jménem, ke kterému je možné doplnit např. návratovou hodnotu operace nebo vstupní/výstupní parametry.

V diagramu je třída reprezentována obdélníkem, vodorovně rozděleným na tři části. V horní části se nachází jméno třídy, v prostřední jsou atributy a ve spodní části se nachází operace třídy (viz Obr. 24).

Obr. 24: Třída.

Vazby

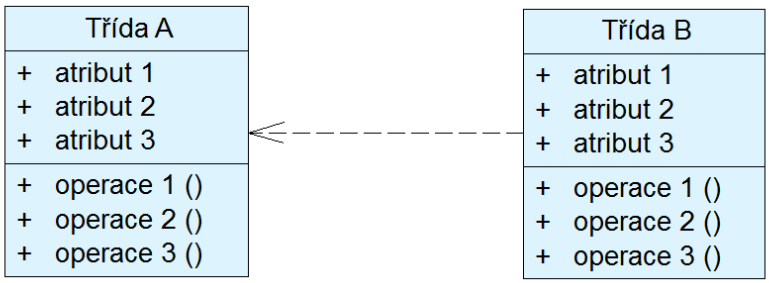
Při modelování pomocí diagramu tříd je velmi nepravděpodobné, že bychom se obešli bez vazeb. Jsou to právě vazby, které určují způsob spolupráce mezi třídami a rozhraními a jejich vzájemné uspořádání. Základními druhy vazeb jsou:

- Závislost
- Generalizace
- Asociace

Závislost

Závislost je druh vazby, kterým vyjádříme, že jeden prvek využívá služeb či informací druhého prvku. Prvek, který těchto služeb využívá, můžeme označit jako závislý,

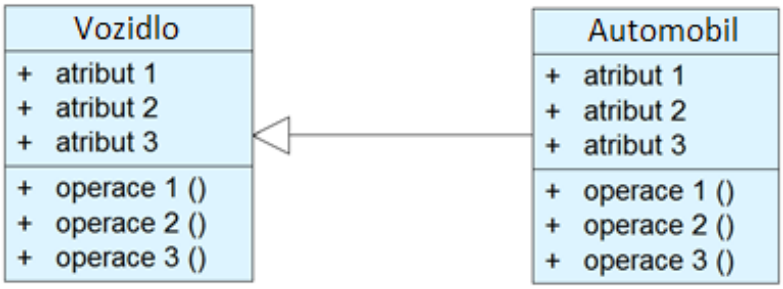
využívaný prvek pak jako nezávislý. Je tomu tak, protože jakákoliv změna v prvku (nejčastěji třída), který poskytuje služby (metody), ovlivní především uživatele této služby (též nejčastěji třída) i jeho vlastní chování. V diagramu je závislost zobrazena jako přerušovaná čára s šipkou, určující směr závislosti. Na Obr. 25 je zobrazena závislost mezi dvěma třídami, třída A je závislou třídou.



Obr. 25: Závislost mezi třídami.

Generalizace

Tento druh vazby vyjadřuje vztah mezi obecnějším a konkrétnějším prvkem. Nejčastěji je využívána pro popis generalizace, v OOP terminologii tzv. dědičnosti, tříd. Na úrovni diagramu tříd nejsou pro dědičnost určena žádná omezení, ta jsou většinou dána až konkrétními programovacími jazyky. Je možné, aby třída měla např. více nadřazených tříd (tzv. vícenásobná dědičnost). Graficky je dědičnost (potažmo generalizace) znázorněna plnou čárou zakončenou prázdným trojúhelníkem, určujícím směr dědění (generalizace). Viz Obr. 26.



Obr. 26: Generalizace mezi třídami.

Asociace

Asociace je vazbou, která určuje, jak budou mezi sebou propojeny instance třídy. Pomocí asociací tedy definujeme strukturu modelu. Přímo v diagramu jsou samozřejmě propojeny třídy, jelikož diagram tříd instance nijak nezachycuje (tím se zabývá diagram objektů), pouze určuje pravidla pro jejich vznik. Může se jednat o dvě různé třídy, tedy binární vazbu. Stejně tak lze ale propojit i jednu třídu samu se sebou (unární vazba) - znamená to, že mezi sebou budou propojeny instance té samé třídy. Možné jsou i tzv. n-ární vazby, kdy je k sobě vázáno několik tříd najednou. V takovém případě je nezbytností zavést tzv. vazební třídu, skrze kterou se n-ární vazba realizuje. Mimo tento účel slouží vazební třída k detailnějšímu popisu vazby, lze ji tedy využít i u unárních a binárních vazeb, pokud je to pro náš objektový model žádoucí, samy o sobě to ale tyto vazby nevyžadují. Asociační vazba je znázorněna pouze plnou čárou, která ale může být zakončena různými symboly podle toho, jaké vlastnosti má vazba. Mezi tyto základní vlastnosti patří:

- Jméno
- Směr
- Multiplicita
- Druh

Jméno asociace je vhodné využít pro přesné určení významu asociace. Ten většinou vyplývá z toho, jaké instance jsou k sobě vázány, nicméně i tak se vyplatí pojmenováním asociace jednoznačně určit jaký její význam. V případě, že třída je vázána vícero asociacemi najednou, stává se využití jména, pokud chceme zachovat čitelnost modelu, nutností. V diagramu jméno znázorníme textem, který umístíme nad prostředek vazby.

Směr asociace je poměrně málo využívanou vlastností. Většina asociací je totiž obousměrná. Existují ovšem případy, kdy je vhodné, aby asociace byla pouze jednosměrná, tedy aby objekt A věděl o objektu B, ale nikoliv naopak. Směr vazby v takovém případě znázorníme v digramu jednoduchou šipkou na konci vazby. Obousměrné vazby by tedy měly být znázorněny s oběma šipkami na koncích, pro zjednodušení se ale v takovém případě neuvádí ani jedna.

Multiplicita určuje, kolik objektů může být mezi sebou pomocí asociace provázáno. Je zapsána pomocí dvou celých čísel oddělených tečkami, jež představují interval – a minimální a maximální hodnotu počtu objektů na konci vazby. Nemusí se však jednat o konkrétní hodnoty, neurčitost se zapisuje znakem hvězdičky. Pokud jsou oba symboly (číslo či hvězdička) shodné, lze zápis zkrátit uvedením pouze jednoho symbolu. V tabulce níže (Tabulka 4) je uveden přehled značení multiplicity.

Značení	Význam
0..1	0 nebo 1
0..*	0 až více
1..1	právě 1
1..*	1 až více
*	0 až více

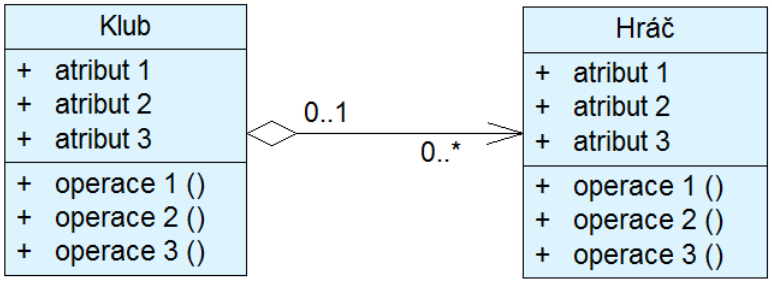
Tabulka 4: Značení multiplicity.

Druh vazby je velmi důležitou vlastností. Určuje relativní význam vázaných tříd. Rozlišujeme tyto základní druhy vazeb:

- Běžná vazba
- Agregace
- Kompozice

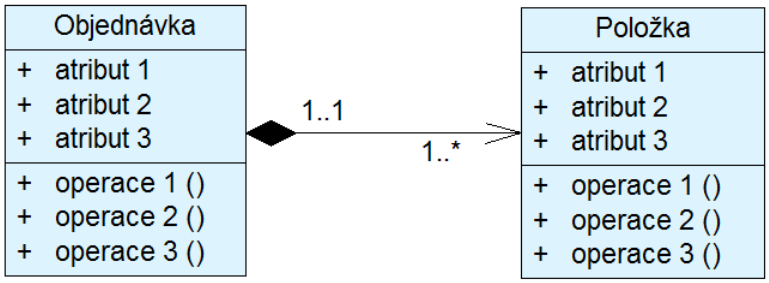
Běžná asociační vazba mezi dvěma třídami, nijak nevyzdvihuje jednu nebo druhou třídu, obě dvě mají stejný význam, z hlediska modelu jsou si rovny. Běžná vazba je znázorněna pouze plnou čárou.

Může ovšem nastat případ, kdy budeme potřebovat význam jedné ze tříd zdůraznit – jedná se o vztah „celek-část“. Tedy modelovaný objekt označený jako „celek“ (prostřednictvím třídy) se skládá z objektů označených jako „část“. Pokud dojde k zániku „celku“, „část“ sama o sobě může dále existovat. Takovému vztahu říkáme agregace. Agregaci znázorníme prázdným kosočtvercem na konci asociační vazby (viz Obr. 27).



Obr. 27: Agregace mezi třídami.

Posledním druhem vazby je kompozice. Jedná se o speciální druh agregace, s tím rozdílem, že „celek“ je v tomto případě pevně svázán s „částmi“, ze kterých se skládá, pokud dojde k zániku „celku“, zaniknou i „části“, jelikož samy o sobě nemůžou existovat. Kompozicí vyjadřujeme silnější druh vlastnictví a moci „celku“ nad „částmi“, „části“ nesmí být součástí jiné kompozice, na rozdíl od běžných agregací, kde může „část“ patřit do mnoha „celků“. Kompozici v diagramu poznáme podle plného kosočtverce na konci asociační vazby (viz Obr. 28).



Obr. 28: Kompozice mezi třídami.

Rozhraní

Rozhraní slouží k oddělení definice operací od jejich implementace. Toto samo o sobě má dalekosáhlé důsledky – slouží k definování hranic v návrhu, modularizaci systému. V teorii OOP hraje rozhraní velmi důležitou roli.

V UML diagramu znázorníme rozhraní podobně jako třídu bez atributů, s tím rozdílem, že nad jméno třídy přidáme tzv. stereotyp, označující, že se jedná o rozhraní. Každé rozhraní musí mít:

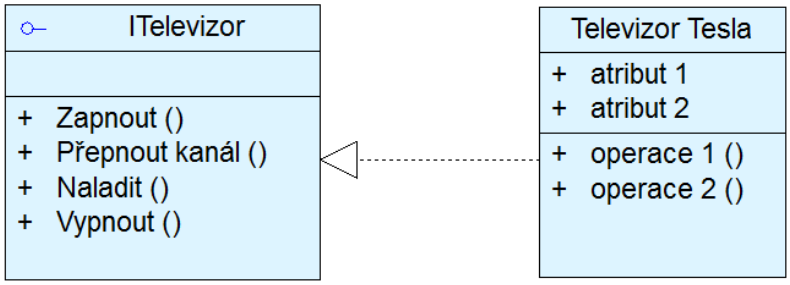
- Jméno

- Operace

Pro jméno rozhraní platí stejné náležitosti jako u třídy. Tedy slouží k rozlišení od ostatních rozhraní a vyjadřuje jeho smysl. Doporučuje se, aby každé jméno rozhraní začínalo písmenem „I“ (jako Interface), aby bylo ze jména na první pohled zřejmé, že se jedná o rozhraní a ne o třídu.

Operace musí obsahovat každé rozhraní, jejich definice je jeho smyslem.

Rozhraní, podobně jako třída, může být součástí vazeb jako závislost, generalizace a asociace. Oproti třídě může ovšem využít další vazby a tou je realizace. Bez této vazby by totiž rozhraní nemělo smysl. To že třída realizuje rozhraní, můžeme znázornit dvěma způsoby. Jedním je přerušovaná čára zakončená prázdným trojúhelníkem, směrem od třídy k rozhraní (viz Obr. 29). Druhý způsob obsahuje kromě realizace rozhraní i požadavek na jeho využití (podmínka komunikace mezi dvěma třídami).



Obr. 29: Implementace rozhraní.

Diagram případů užití

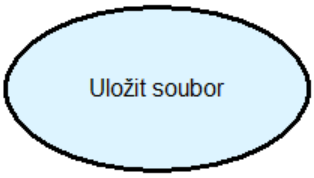
Diagram případů užití (Use Case diagram) je jedním z digramů UML, které slouží k mapování chování systému. Diagram případů užití, Use Case diagram, se ve většině případů skládá z těchto částí:

- Případ užití (Use Case)
- Hranice systému (Subject)
- Aktér (Actor)
- Vazby
    - Include
    - Extend

- Generalizace

Případ užití, aktér, hranice systému

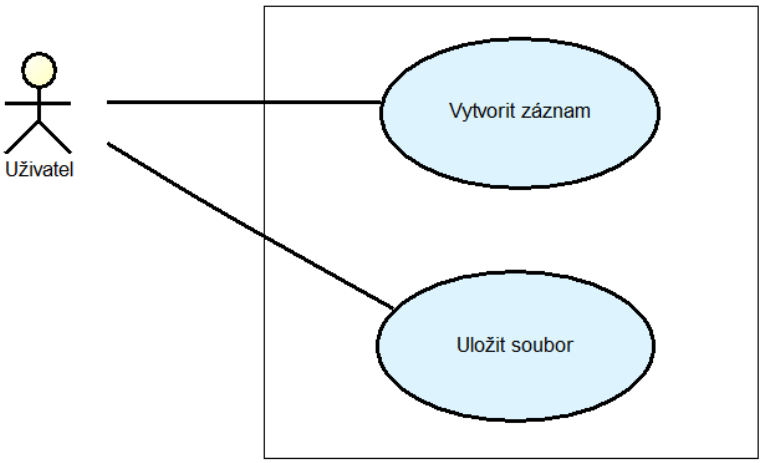
Jelikož žádný systém nemůže existovat v úplné izolaci - v opačném případě by postrádal smysl – komunikuje nějakým způsobem se svým okolím. Okolí může představovat např. člověk, který systém používá – poskytuje mu podněty k určité činnosti a očekává její výsledný výstup. Tuto systémovou činnost nazýváme případ užití, neboli Use Case. Případ užití je v diagramu znázorněn pomocí oválu. Každý případ užití musí mít jednoznačné a unikátní jméno (viz Obr. 30).



Obr. 30: Případ užití.

Tímto způsobem lze zmapovat celý modelovaný IS, případy užití, které jsou v IS navíc, jsou zbytečné a naopak pokud některý chybí, funkčnost IS nebude úplná, což se dá považovat za kritické selhání IS, jelikož nenaplňuje požadavky na něj kladené. Případy užití lze tedy úspěšně využít jako metriku při postupné tvorbě IS (a to ve všech fázích vývoje, od analýzy až po implementaci), pokud jsou všechny případy užití uspokojivě pokryty, lze IS považovat za hotový.

Prvek z vnějšího okolí, který vyvolává případ užití, se nazývá aktér. Jak bylo řečeno, může se jednat o člověka, ale obecně téměř o cokoliv jiného co žádá nějaký výstup, či provedení určité operace - např. další počítač apod. Vnější aktér je pro systém velmi podstatný, on totiž klade na IS požadavky, které vedou k jednotlivým případům užití. Aktér je vyznačen pomocí symbolu figurky. Obr. 31 zobrazuje interakci aktéra s IS.

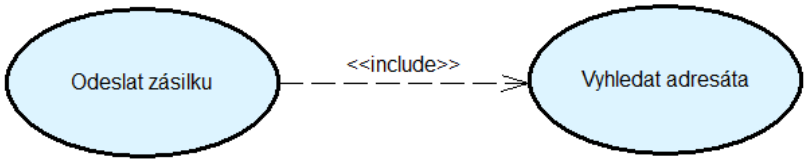


Obr. 31: Interakce aktéra s IS.

Pro přehlednost je lepší do diagramu vždy zaznamenat i pomyslnou hranici IS, která jej vymezuje vůči jeho okolí. Jediný prvek vnějšího prostředí, který smí tuto hranici „porušit“, je právě aktér, který vyžaduje od IS provedení některého z případů užití.

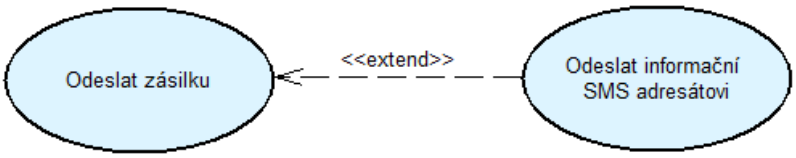
Vazby

Vazba include slouží k vyjádření vztahu „celek-část“. Případ užití A v tomto zahrnuje případ užití B. Případ užití A nelze provést bez provedení případu užití B (viz Obr. 32).



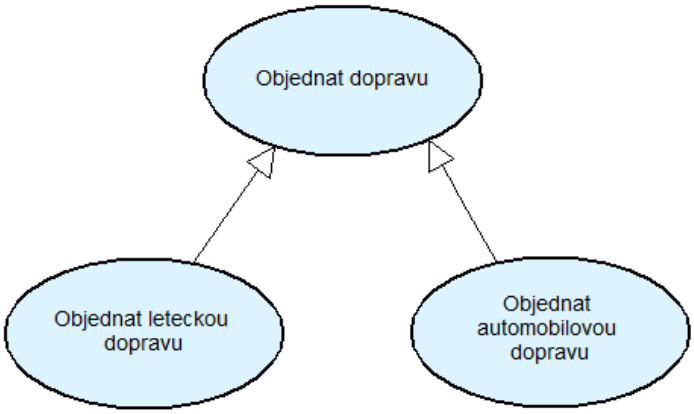
Obr. 32: Vazba include mezi dvěma případy užití.

Vazba extend vyjadřuje rozšíření původního případu užití. V tomto případě však musí být případ užití B proveditelný samostatně, jedná se o samostatný případ užití. Případ užití A o případu užití B ovšem „nic netuší“, naopak případ užití B o případu užití A „má povědomí“ (vyjadřuje to i směr vazby) a je vykonáván při jeho běhu a tím rozšiřuje jeho funkčnost (viz Obr. 33).



Obr. 33: Vazba extend mezi dvěma případy užití.

Generalizaci lze mezi případy užití též využít. Chceme-li např. definovat z nějakého důvodu obecnější případ užití, ze kterého chceme odvozovat konkrétní případy užití, generalizace nám to umožní (viz Obr. 34).



Obr. 34: Generalizace mezi případy užití.

Specifikace případu užití

Diagram případů užití je specifikován standardem UML zcela jasně, ovšem to je pouze vizualizace případů užití a jejich vztahů mezi nimi. Chceme-li však rozepsat konkrétní případ užití, žádný standard se nám nenabízí. Rozepsat každý jednotlivý případ užití je nutnost, jelikož symbol v diagramu s názvem případu užití nám moc konkrétních informací o něm nepředá. Potřebujeme tedy zaznamenat strukturovaný, jasný a výstižný text, který daný případ užití popíše.

Takovému textu říkáme scénář. Existuje více způsobů psaní scénáře. Scénář můžeme psát přímo, či ho větvit apod. Ale vždy při tom musíme dbát na jednoznačnost, strukturovanost, úplnost. Při větvení bychom měli využívat klíčových slov (KDYŽ, JINAK, POTOM, PRO apod.). Klíčová slova jsou velmi důležitá, protože scénářem se řídí programátor při implementaci konkrétního případu užití. Scénář by měl být ale

zároveň srozumitelný i běžným uživatelům, aby mohli případ užití v případě potřeby doplnit, upozornit na nepřesnosti apod.

Detailní specifikaci případu užití bychom měli provést dle nějaké šablony. Např. takto:

- Název případu užití
- Stručný popis podstaty případu užití
- ID – jednoznačný identifikátor případu užití
- Vstupní podmínky – podmínky, kritéria, která musí být splněna ještě před „spuštěním“ případu užití, omezení stavu systému
- Popis scénáře – tok událostí, jednotlivé kroky případu užití. Jednoduchá sekvence číslovaných kroků, co dělá aktér a co systém.
- Následné podmínky – podmínky, kritéria, která musí být splněna na konci případu.

Jednoduchý příklad specifikace případu užití:

Název případu užití: Práce se souborem

ID: UC01_Prace_se_souborem

Scénář:

Uživatel může otevřít následující typy souborů:

- .xsl
- .doc
- .prj

KDYŽ je soubor poškozen, je zobrazena varovná hláška.

JINAK je soubor nahrán do aplikace a uživatel s ním může pracovat. To znamená:

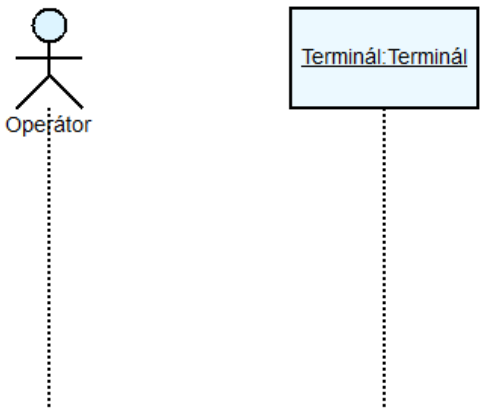
- upravovat načtená data
- tyto úpravy následně ukládat

Sekvenční diagram

Sekvenční diagram je jedním z diagramů, který slouží k modelování dynamiky systému. V téměř každém systému (až na triviální případy) spolu musí objekty komunikovat tak, aby bylo dosaženo kýženého výsledku. Tuto komunikaci, jelikož se jedná o dynamický děj však diagram tříd nemůže postihovat (zachycuje systém v jeho statické podobě). Smyslem sekvenčních diagramů je právě mapovat komunikaci, spolupráci mezi objekty. Komunikace mezi objekty probíhá prostřednictvím zpráv, velký důraz je přitom kladen na jejich časovou souslednost. Jeden sekvenční diagram zpravidla zachycuje komunikaci v modelu, během realizace jednoho případu užití. Sekvenční diagram běžně sestává z těchto částí:

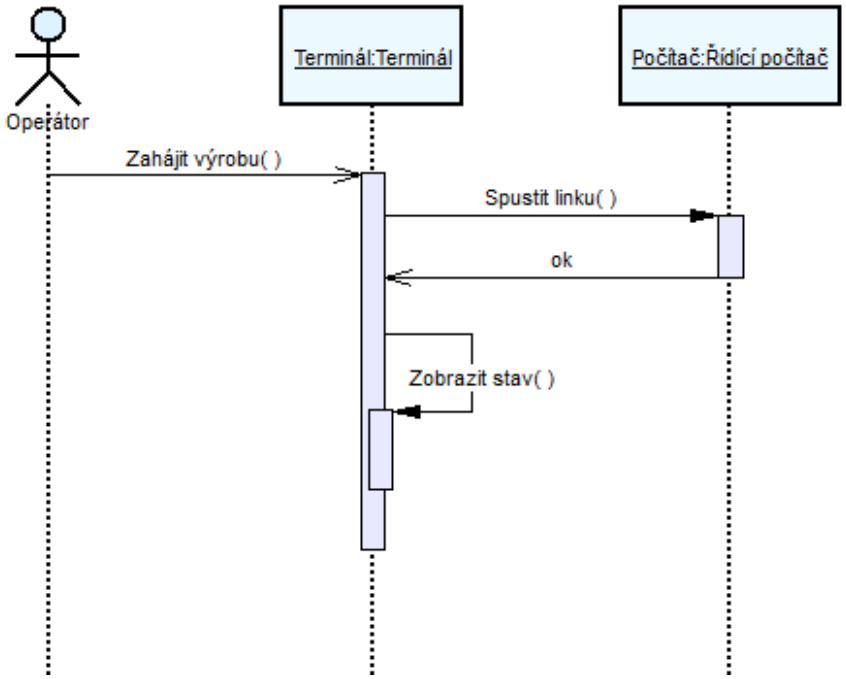
- Objekty
    - Instance
    - Aktér
- Zprávy
    - Call
    - Return
    - Create
    - Destroy

Objektem může být myšlen objekt ve smyslu OOP, tedy instance třídy – může se ale jednat i o objekt z reálného světa, který se systémem komunikuje – tedy aktér. Jelikož sekvenční diagram slouží k popisu komunikace v rámci jednoho případu užití, přítomnost aktéra v diagramu není výjimkou. Graficky je objekt znázorněn buď jako obdélník se jménem třídy a instance nebo symbolem figurky, představující aktéra. V obou případech je pod objekty svislá přerušovaná čára, zobrazující časový rozměr komunikace (viz Obr. 35).



Obr. 35: Aktér a instance třídy.

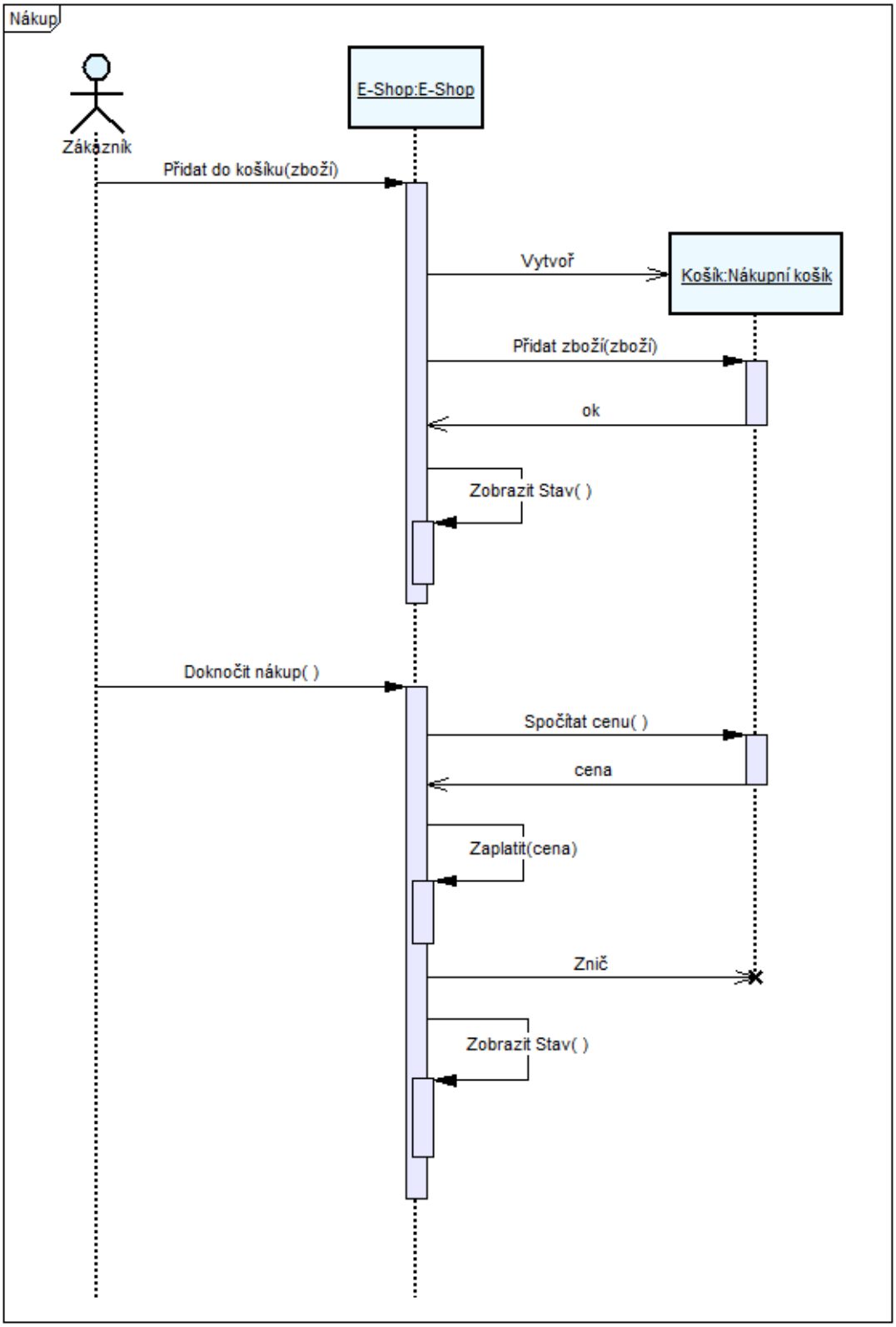
Zprávy mohou být několika typů. Pokud se jedná např. o volání procedury, zpráva může obsahovat i její parametry, podobně návratové volání může vracet hodnotu. Zprávy mohou být zasílány synchronně a asynchronně. Jsou reprezentovány čarou, která vede od jednoho objektu k druhému (dle pozice na časové ose lze určit souslednost volání). Na Obr. 36 je zobrazen příklad komunikace mezi objekty.



Obr. 36: Komunikace mezi objekty.

Pokud potřebujeme vytvořit objekt „za běhu“, lze to provést pomocí zprávy Create. Podobně lze objekt zničit (ukončit jeho existenci) voláním zprávy Destroy. Na Obr. 37 je zobrazen příklad těchto volání.

Dobře navržená komunikace mezi objekty sestává z jednoduchých operací rozdělených mezi odpovídající třídy. Pomocí sekvenčního diagramu tak můžeme ověřit, zda jsme navrhli komunikaci pro daný případ užití správně.



Obr. 37: Tvorba a zánik instance v průběhu komunikace.

Diagram aktivit

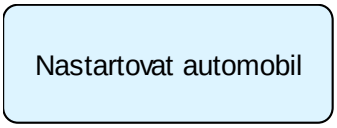
Diagram aktivit je dalším z diagramů UML, který slouží k modelování dynamických aspektů systému. Zobrazuje uspořádaný běh činností. Používá se především k popisu business procesů, ale lze jej využít obecně pro popis jakékoliv procedury či logiky, jelikož umožnuje zobrazit paralelní běh činností, jejich větvení apod. (viz dále). Diagram aktivit běžně sestává z:

- Akce (Actions)
- Aktivity (Activities)
- Přechody (Flows)
- Alternativní běh
    - Větvení (Branch)
    - Sjednocení (Merge)
- Paralelní běh
    - Větvení (Fork)
    - Sjednocení (Join)
- Zóna odpovědnosti (Swimlanes)
- Signály (Signals)

Každý proces se skládá z posloupnosti nějakých činností. V diagramu aktivit existují dva prvky takového typu. Jsou to tzv. akce a aktivity. Akce představuje atomickou činnost, která nemůže být v modelovaném procesu dále rozložena - dekomponována.

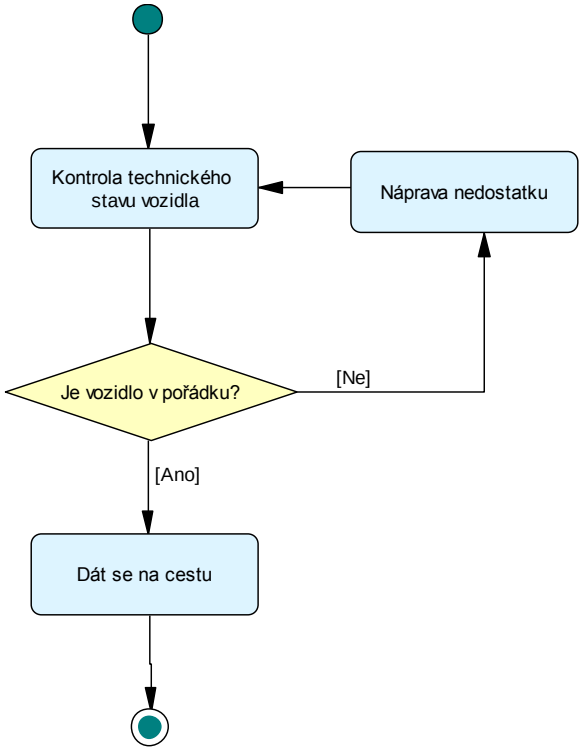
Naproti tomu aktivita je definována jako činnost, která sestává z atomických akcí. Aktivity můžeme nořit do sebe, aktivita se může skládat i z jiných aktivit. Aktivita slouží k organizaci akcí do struktur. V konečném důsledku se tak každá aktivita skládá z atomických činností. Aktivity nám umožňují modelovat procesy v různé míře detailu, kteroukoliv z aktivit můžeme zobrazit jako nový diagram a naopak.

Aktivita i akce se znázorňují shodně – zaobleným obdélníkem (viz Obr. 38).



Obr. 38: Znázornění aktivity či akce.

Abychom určili, v jakém pořadí se mají aktivity nebo akce provádět, potřebujeme je nějakým způsobem propojit. K tomuto účelu slouží tzv. přechody. Přechod znázorníme jednoduchou čarou od jedné aktivity (či akce) k druhé. Čáru zakončíme šipkou a tím určíme směr toku řízení procesu (pořadí vykonávaných činností). Většina procesů má definován svůj počátek a konec. Počátek procesu zobrazíme plným kruhem, konec procesu prázdným kruhem, který má v sobě menší plný kruh.

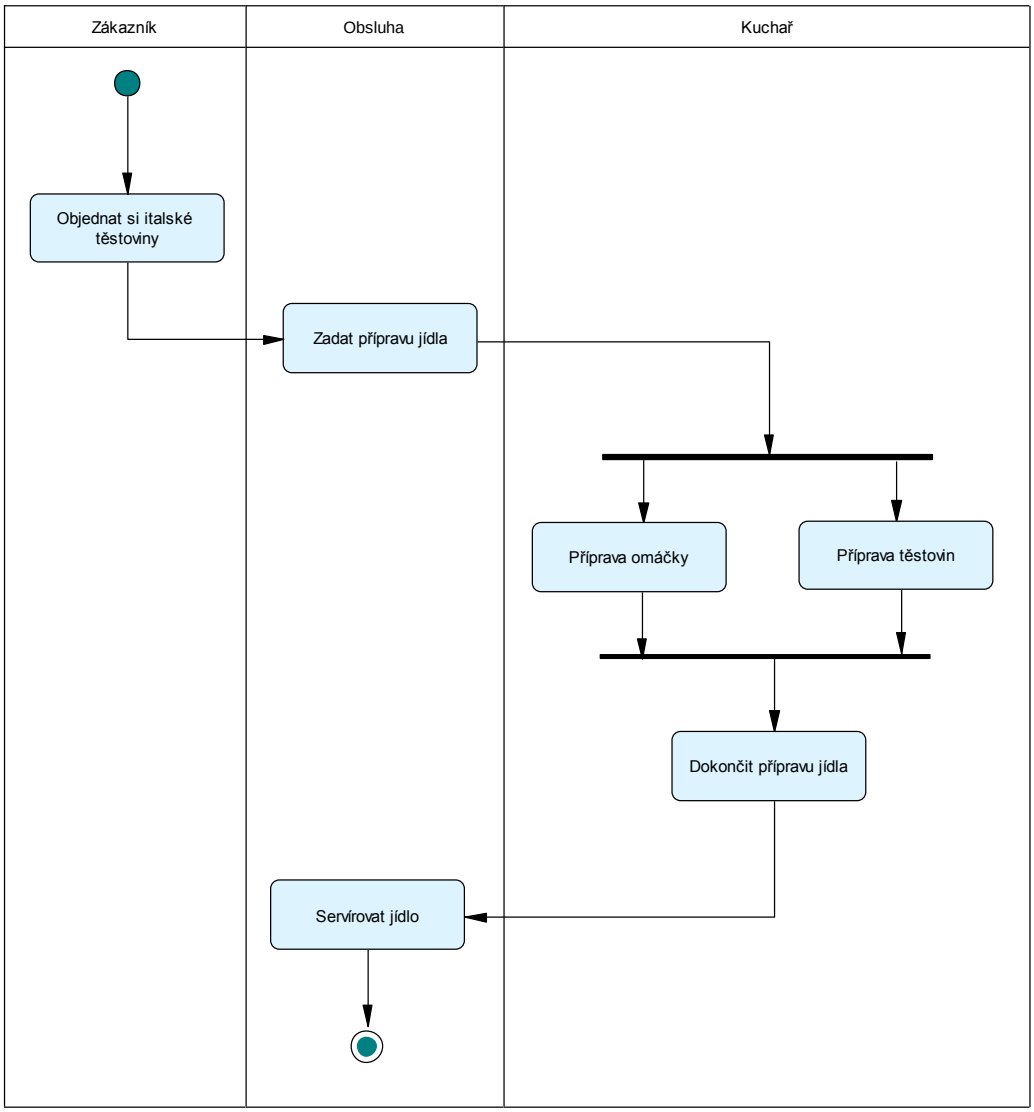


Obr. 39: Alternativní větvení toku.

Pokud námi modelovaný proces není pouze prostou sekvencí příkazů, budeme pravděpodobně potřebovat větvit řízení toku procesu – definovat alternativy běhu. Sémantika takového alternativního větvení je naprosto shodná s příkazem IF – řídící strukturou známou z programovacích jazyků. Proces pokračuje tou větví, která splní definovanou podmínku (tzv. guard). Alternativní větvení je graficky znázorněno kosočtvercem, ze kterého vedou jednotlivé přechody. Nad každým přechodem je uvedena v hranatých závorkách příslušná podmínka. Pokud chceme rozvětvený běh opět sjednotit, provedeme to svedením přechodů do společného bodu, bez jakýchkoliv dalších podmínek. Na Obr. 39 je zobrazen příklad alternativního větvení toku.

Modelujeme-li složitější proces, nemusí nám alternativní větvení postačovat. Bude-li třeba, aby se několik aktivit vykonávalo současně, využijeme paralelního větvení řízení toku procesu (fork). Jedna větev se může rozvětvit na dvě a více paralelních větví. Pokud chceme paralelní běh opět sjednotit, větve synchronizovat, použijeme paralelní sjednocení, tzv. bariéru. Jak známo z paralelního programování, dokud všechny větve procesu nedosáhnou bariéry (join), jsou ostatní větve ve stavu nečinnosti. Po sjednocení vede z bariéry opět jedna větev (přechod). V diagramu je paralelní větvení i sjednocení znázorněno tlustou plnou čarou. Na Obr. 40 je zobrazen příklad alternativního větvení.

Pomocí tzv. zón odpovědnosti (swimlanes) lze diagram aktivit členit tak, abychom měli přehled o tom, které činnosti kdo nebo co vykonává, či za ně odpovídá. Jsou znázorněny vertikálními plnými čarami, které člení celý diagram na několik částí, každá část představuje člověka, oddělení firmy, či jiný prvek, který může provádět aktivity v rámci procesu. Do těchto částí pak umisťujeme jednotlivé aktivity (viz Obr. 40).



Obr. 40: Příklad paralelního větvení toku a členění diagramu.

Příloha B

Skript pro SŘBD Sybase SQL Anywhere 12

```
if exists(select 1 from sys.sysforeignkey where role='FK_HOST_EVIDUJE_RECEPCE') then
    alter table Host
       delete foreign key FK_HOST_EVIDUJE_RECEPCE
end if;

if exists(select 1 from sys.sysforeignkey where role='FK_HOST_MA_UCET') then
    alter table Host
       delete foreign key FK_HOST_MA_UCET
end if;

if exists(select 1 from sys.sysforeignkey where role='FK_MANAGER_GENERALIZ_ZAMESTNA')
then
    alter table Manager
       delete foreign key FK_MANAGER_GENERALIZ_ZAMESTNA
end if;

if exists(select 1 from sys.sysforeignkey where role='FK_MANAGER_RIDI_HOTEL') then
    alter table Manager
       delete foreign key FK_MANAGER_RIDI_HOTEL
end if;

if exists(select 1 from sys.sysforeignkey where role='FK_POKOJ_MA_HOTEL') then
    alter table Pokoj
       delete foreign key FK_POKOJ_MA_HOTEL
end if;

if exists(select 1 from sys.sysforeignkey where role='FK_RECEPCE_MA_HOTEL') then
    alter table Recepce
       delete foreign key FK_RECEPCE_MA_HOTEL
end if;

if         exists(select         1         from         sys.sysforeignkey         where
role='FK_RECEPCNI_GENERALIZ_ZAMESTNA') then
    alter table Recepcni
       delete foreign key FK_RECEPCNI_GENERALIZ_ZAMESTNA
end if;

if exists(select 1 from sys.sysforeignkey where role='FK_RECEPCNI_PRACUJE_RECEPCE')
then
    alter table Recepcni
       delete foreign key FK_RECEPCNI_PRACUJE_RECEPCE
end if;

if exists(select 1 from sys.sysforeignkey where role='FK_REZERVAC_MA_HOST') then
    alter table Rezervace
       delete foreign key FK_REZERVAC_MA_HOST
end if;

if exists(select 1 from sys.sysforeignkey where role='FK_REZERVAC_MA_POKOJ') then
    alter table Rezervace
       delete foreign key FK_REZERVAC_MA_POKOJ
end if;

if exists(select 1 from sys.sysforeignkey where role='FK_UBYTOVAN_MA_HOST') then
    alter table Ubytovani
       delete foreign key FK_UBYTOVAN_MA_HOST
end if;
```

```
if exists(select 1 from sys.sysforeignkey where role='FK_UBYTOVAN_MA_POKOJ') then
    alter table Ubytovani
       delete foreign key FK_UBYTOVAN_MA_POKOJ
end if;

drop index if exists Host.ma_FK;

drop index if exists Host.eviduje_FK;

drop index if exists Host.Host_PK;

drop table if exists Host;

drop index if exists Hotel.Hotel_PK;

drop table if exists Hotel;

drop index if exists Manager.Generalization_2_FK;

drop index if exists Manager.ridi_FK;

drop index if exists Manager.Manager_PK;

drop table if exists Manager;

drop index if exists Pokoj.ma_FK;

drop index if exists Pokoj.Pokoj_PK;

drop table if exists Pokoj;

drop index if exists Recepce.ma_FK;

drop index if exists Recepce.Recepce_PK;

drop table if exists Recepce;

drop index if exists Recepcni.Generalization_1_FK;

drop index if exists Recepcni.pracuje_FK;

drop index if exists Recepcni.Recepcni_PK;

drop table if exists Recepcni;

drop index if exists Rezervace.ma_FK2;

drop index if exists Rezervace.ma_FK;

drop index if exists Rezervace.Rezervace_PK;

drop table if exists Rezervace;

drop index if exists Ubytovani.ma_FK2;

drop index if exists Ubytovani.ma_FK;

drop index if exists Ubytovani.Ubytovani_PK;

drop table if exists Ubytovani;

drop index if exists Ucet.Ucet_PK;

drop table if exists Ucet;

drop index if exists Zamestnanec.Zamestnanec_PK;
```

```
drop table if exists Zamestnanec;

/*==============================================================*/
/* Table: Host                                                  */
/*==============================================================*/
create table Host
(
   id                   integer                        not null,
   Rec_id               integer                        not null,
   Uce_id               integer                        not null,
   jmeno                varchar(254)                   null,
   prijmeni             varchar(254)                   null,
   cisloDokladu         varchar(254)                   null,
   narodnost            varchar(254)                   null,
   constraint PK_HOST primary key (id)
);

/*==============================================================*/
/* Index: Host_PK                                               */
/*==============================================================*/
create unique index Host_PK on Host (
id ASC
);

/*==============================================================*/
/* Index: eviduje_FK                                            */
/*==============================================================*/
create index eviduje_FK on Host (
Rec_id ASC
);

/*==============================================================*/
/* Index: ma_FK                                                 */
/*==============================================================*/
create index ma_FK on Host (
Uce_id ASC
);

/*==============================================================*/
/* Table: Hotel                                                 */
/*==============================================================*/
create table Hotel
(
   id                   integer                        not null,
   jmeno                varchar(254)                   null,
   pocetHvezdicek       integer                        null,
   adresa               varchar(254)                   null,
   popis                varchar(254)                   null,
   dopravniDostupnost   varchar(254)                   null,
   parkovani            varchar(254)                   null,
   constraint PK_HOTEL primary key (id)
);

/*==============================================================*/
/* Index: Hotel_PK                                              */
/*==============================================================*/
create unique index Hotel_PK on Hotel (
id ASC
);

/*==============================================================*/
/* Table: Manager                                               */
/*==============================================================*/
create table Manager
(
```

```
   id                   integer                        not null,
   Hot_id               integer                        not null,
   Zam_id               integer                        not null,
   cile                 varchar(254)                   null,
   podilZeZisku         decimal                        null,
   constraint PK_MANAGER primary key (id)
);

/*==============================================================*/
/* Index: Manager_PK                                            */
/*==============================================================*/
create unique index Manager_PK on Manager (
id ASC
);

/*==============================================================*/
/* Index: ridi_FK                                               */
/*==============================================================*/
create index ridi_FK on Manager (
Hot_id ASC
);

/*==============================================================*/
/* Index: Generalization_2_FK                                   */
/*==============================================================*/
create index Generalization_2_FK on Manager (
Zam_id ASC
);

/*==============================================================*/
/* Table: Pokoj                                                 */
/*==============================================================*/
create table Pokoj
(
   id                   integer                        not null,
   Hot_id               integer                        not null,
   cislo                varchar(254)                   null,
   samostatnaLuzka      integer                        null,
   manzelskePostele     integer                        null,
   zakladniCena         decimal                        null,
   podlazi              integer                        null,
   vymer                decimal                        null,
   constraint PK_POKOJ primary key (id)
);

/*==============================================================*/
/* Index: Pokoj_PK                                              */
/*==============================================================*/
create unique index Pokoj_PK on Pokoj (
id ASC
);

/*==============================================================*/
/* Index: ma_FK                                                 */
/*==============================================================*/
create index ma_FK on Pokoj (
Hot_id ASC
);

/*==============================================================*/
/* Table: Recepce                                               */
/*==============================================================*/
create table Recepce
(
   id                   integer                        not null,
   Hot_id               integer                        not null,
```

```
   constraint PK_RECEPCE primary key (id)
);

/*==============================================================*/
/* Index: Recepce_PK                                            */
/*==============================================================*/
create unique index Recepce_PK on Recepce (
id ASC
);

/*==============================================================*/
/* Index: ma_FK                                                 */
/*==============================================================*/
create index ma_FK on Recepce (
Hot_id ASC
);

/*==============================================================*/
/* Table: Recepcni                                              */
/*==============================================================*/
create table Recepcni
(
   id                   integer                        not null,
   Zam_id               integer                        not null,
   Rec_id               integer                        not null,
   smena                varchar(254)                   null,
   jazykovaVybavenost   varchar(254)                   null,
   constraint PK_RECEPCNI primary key (id)
);

/*==============================================================*/
/* Index: Recepcni_PK                                           */
/*==============================================================*/
create unique index Recepcni_PK on Recepcni (
id ASC
);

/*==============================================================*/
/* Index: pracuje_FK                                            */
/*==============================================================*/
create index pracuje_FK on Recepcni (
Rec_id ASC
);

/*==============================================================*/
/* Index: Generalization_1_FK                                   */
/*==============================================================*/
create index Generalization_1_FK on Recepcni (
Zam_id ASC
);

/*==============================================================*/
/* Table: Rezervace                                             */
/*==============================================================*/
create table Rezervace
(
   id                   integer                        not null,
   Pok_id               integer                        not null,
   Hos_id               integer                        not null,
   datum                timestamp                      null,
   rezervaceOd          timestamp                      null,
   rezervaceDo          timestamp                      null,
   pocetOsob            integer                        null,
   stav                 varchar(254)                   null,
   constraint PK_REZERVACE primary key (id)
);
```

```
/*==============================================================*/
/* Index: Rezervace_PK                                          */
/*==============================================================*/
create unique index Rezervace_PK on Rezervace (
id ASC
);

/*==============================================================*/
/* Index: ma_FK                                                 */
/*==============================================================*/
create index ma_FK on Rezervace (
Pok_id ASC
);

/*==============================================================*/
/* Index: ma_FK2                                                */
/*==============================================================*/
create index ma_FK2 on Rezervace (
Hos_id ASC
);

/*==============================================================*/
/* Table: Ubytovani                                             */
/*==============================================================*/
create table Ubytovani
(
   id                   integer                        not null,
   Pok_id               integer                        not null,
   Hos_id               integer                        not null,
   ubytovaniOd          timestamp                      null,
   ubytovaniDo          timestamp                      null,
   poznamka             varchar(254)                   null,
   uctovanaCena         decimal                        null,
   constraint PK_UBYTOVANI primary key (id)
);

/*==============================================================*/
/* Index: Ubytovani_PK                                          */
/*==============================================================*/
create unique index Ubytovani_PK on Ubytovani (
id ASC
);

/*==============================================================*/
/* Index: ma_FK                                                 */
/*==============================================================*/
create index ma_FK on Ubytovani (
Pok_id ASC
);

/*==============================================================*/
/* Index: ma_FK2                                                */
/*==============================================================*/
create index ma_FK2 on Ubytovani (
Hos_id ASC
);

/*==============================================================*/
/* Table: Ucet                                                  */
/*==============================================================*/
create table Ucet
(
   id                   integer                        not null,
   cena                 smallint                       null,
   mena                 varchar(254)                   null,
```

```
   datumOtevreni        timestamp                      null,
   datumUzavreni        timestamp                      null,
   constraint PK_UCET primary key (id)
);

/*==============================================================*/
/* Index: Ucet_PK                                               */
/*==============================================================*/
create unique index Ucet_PK on Ucet (
id ASC
);

/*==============================================================*/
/* Table: Zamestnanec                                           */
/*==============================================================*/
create table Zamestnanec
(
   id                   integer                        not null,
   jmeno                varchar(254)                   null,
   prijmeni             varchar(254)                   null,
   datumNarozeni        varchar(254)                   null,
   mzda                 decimal                        null,
   constraint PK_ZAMESTNANEC primary key (id)
);

/*==============================================================*/
/* Index: Zamestnanec_PK                                        */
/*==============================================================*/
create unique index Zamestnanec_PK on Zamestnanec (
id ASC
);

alter table Host
   add constraint FK_HOST_EVIDUJE_RECEPCE foreign key (Rec_id)
      references Recepce (id)
      on update restrict
      on delete restrict;

alter table Host
   add constraint FK_HOST_MA_UCET foreign key (Uce_id)
      references Ucet (id)
      on update restrict
      on delete restrict;

alter table Manager
   add constraint FK_MANAGER_GENERALIZ_ZAMESTNA foreign key (Zam_id)
      references Zamestnanec (id)
      on update restrict
      on delete restrict;

alter table Manager
   add constraint FK_MANAGER_RIDI_HOTEL foreign key (Hot_id)
      references Hotel (id)
      on update restrict
      on delete restrict;

alter table Pokoj
   add constraint FK_POKOJ_MA_HOTEL foreign key (Hot_id)
      references Hotel (id)
      on update restrict
      on delete restrict;

alter table Recepce
   add constraint FK_RECEPCE_MA_HOTEL foreign key (Hot_id)
      references Hotel (id)
      on update restrict
```

```
      on delete restrict;

alter table Recepcni
   add constraint FK_RECEPCNI_GENERALIZ_ZAMESTNA foreign key (Zam_id)
      references Zamestnanec (id)
      on update restrict
      on delete restrict;

alter table Recepcni
   add constraint FK_RECEPCNI_PRACUJE_RECEPCE foreign key (Rec_id)
      references Recepce (id)
      on update restrict
      on delete restrict;

alter table Rezervace
   add constraint FK_REZERVAC_MA_HOST foreign key (Hos_id)
      references Host (id)
      on update restrict
      on delete restrict;

alter table Rezervace
   add constraint FK_REZERVAC_MA_POKOJ foreign key (Pok_id)
      references Pokoj (id)
      on update restrict
      on delete restrict;

alter table Ubytovani
   add constraint FK_UBYTOVAN_MA_HOST foreign key (Hos_id)
      references Host (id)
      on update restrict
      on delete restrict;

alter table Ubytovani
   add constraint FK_UBYTOVAN_MA_POKOJ foreign key (Pok_id)
      references Pokoj (id)
      on update restrict
      on delete restrict;
```