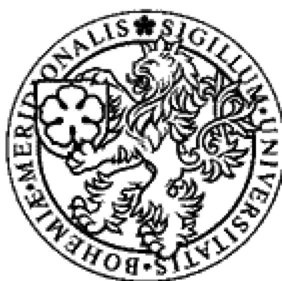
Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra informatiky



UML modelování

(UML návrh jádra modulární aplikace)

Bakalářská práce

Michal Kreuzman

Vedoucí práce : Mgr. Miloš Prokýšek

České Budějovice 2007

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě - v úpravě vzniklé vypuštěním vyznačených částí archivovaných ... fakultou elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.

V Českých Budějovicích dne 20.dubna 2007

Michal Kreuzman

Abstrakt

Bakalářská práce se zabývá návrhem modulární (rozšiřitelné) softwarové aplikace, která má primárně sloužit jako pomůcka pro výuku matematiky, fyziky a chemie pomocí animovaných trojrozměrných scén s využitím stereoskopické projekce. Návrh je vytvořen pomocí modelovacího jazyka UML, CD s kostrou programu v jazyce C# a s modely je přiloženo k práci. Přínosem práce je zdokumentovaný návrh a zdrojové kódy programu (pod open source licencí), který je plně rozšiřitelný a mohou na něm pracovat další zájemci o tuto problematiku. V teoretické části práce je uveden nezbytný teoretický základ jazyka UML, který jsem při návrhu využil. Následuje praktická část, kde se věnuji samotnému návrhu.

Abstract

Bachelor thesis deals with modular (extensible) software aplication which serves mainly as aid tool for teaching mathematics, physics and chemistry through threedimensional animated scenes with usage of stereoscopic projection. Software design is created in modeling language UML, CD with C# source codes and models is included with work. Contribution of work is well documented design and source codes of software program (under open source licence) which is extensible. Other concerned people can join and work on this project. In theoretical part is essential basic of UML that I've used over designing the model. Following by practice part where I focus on modeling.

Poděkování

Děkuji Mgr. Miloši Prokýškovi za odborné a organizační vedení při zpracování této práce.

Obsah

1 Úvod....8
2 Základy jazyka UML....9
2.1 Historie UML....9
2.2 Specifikace UML....10
2.2.1 Diagram Interchange Specification....10
2.2.2 UML Infrastructure....10
2.2.3 UML Superstructure....11
2.2.4 Object Constraint Language....11
2.3 Diagramy v jazyce UML 2.0....11
2.3.1 Strukturální diagramy....12
2.3.2 Diagramy chování....12
2.4 Pohledy v jazyce UML 2.0....13
3 Diagramy případů užití....15
3.1 Hranice systému....15
3.2 Účastníci....15
3.3 Případy užití....16
3.4 Asociace....16
4 Diagramy tříd....18
4.1 Jméno....18
4.1.1 Stereotypy třídy....19
4.2 Atributy....19
4.2.1 Inline atribut....19
4.2.2 Atribut u relace....21
4.2.3 Odvozený atribut....21
4.2.4 Omezení atributů....21
4.2.5 Statické atributy....22
4.3 Operace....22
4.3.1 Omezení operací....23
4.3.2 Výjimky....24
4.3.3 Statické operace....25
4.4 Relace....25
4.4.1 Závislost (Dependency)....25
4.4.2 Asociace (Association)....25
4.4.3 Agregace (Aggregation)....26
4.4.4 Kompozice (Composition)....26
4.4.5 Zobecnění (Generalization)....27
4.5 Rozhraní....27
4.6 Šablony....28
5 Sekvenční diagramy....29
5.1 Objekty a čáry života....29
5.2 Zprávy....30
5.3 Práce s objektem....30
6 Diagramy balíčků....32
6.1 Viditelnost....32
6.2 Speciální zprávy....33
6.3 Vztahy mezi balíčky....33

6.3.1 Importování....33
6.3.2 Přistupování....33
7 Návrh....35
7.1 CASE nástroj Enterprise Architect....36
7.2 Požadavky na software....37
7.3 Komunikační rozhraní....38
7.3.1 Rozhraní pluginů....39
7.3.2 Rozhraní modulů....40
7.4 Jádro....41
7.5 Chování objektů v systému....43
7.5.1 Načítání....43
7.5.2 Renderování....43
7.5.3 Animace s pomocí fyziky....44
7.6 Rozdělení do balíčků....46
8 Závěr....47
Seznam použité literatury....48
Terminologický slovník....49
Příloha A....50
Příloha B....60
Příloha C....61

1 Úvod

V bakalářské práci se zabývám návrhem softwarového řešení pomocí modelovacího jazyka UML (Unified Modeling Language), který slouží pro vizuální zpracování návrhů softwarových řešení. Téma jsem si zvolil z důvodu zájmu o návrhy softwarových řešení. Hlavním prostředkem mé práce je verze jazyka UML 2.0, uvádím ale některé hlavní rozdíly s verzí 1.4. Jazyk UML je v dnešní době standardem pro návrh obecných systémů z důvodu zabudované možnosti rozšíření a tím pádem přizpůsobení se v různých oblastech. Cílem mé práce je návrh softwarové aplikace v jazyce UML, a proto seznamuji čtenáře se základními principy a prostředky tohoto jazyka. Nejedná se o kompletní specifikaci jazyka UML, podrobněji popisuji pouze ty diagramy, které jsem použil při zpracování zadané praktické úlohy. V praktické části je vytvořen návrh modelu (ve specifikaci 2.0) podle zadaných požadavků konkrétního softwaru. Zde popisuji praktické zkušenosti a problémy, na které jsem narazil při sestavování modelu. Cílem mé práce je vytvořit kompletní návrh modulárního systému, který bude spolu s kostrou programu přiložen na doprovodném CD a bude přístupný z internetových stránek. Návrh softwaru si klade za hlavní cíl modularitu (rozšiřitelnost) celého systému a s tím spojené možnosti snadného přidávání funkcionalit kýmkoliv, kdo by o tuto činnost měl zájem.

2 Základy jazyka UML

UML (Unified Modeling Language) je vizuální jazyk, který vznikl za účelem zpracování softwarových návrhů a systémových procesů takovým způsobem, aby mu rozuměli jak programátoři, tak i vývojoví pracovníci. UML nepopisuje jak při analýze a návrhu postupovat, ale poskytuje nástroje pro takovouto činnost. Jazyk podporuje objektově orientovaný přístup a je obecně navržený pro vyjádření a zachycení relací (relationships) a chování (behaviors) tak, aby byl zápis co nejvíce pochopitelný pro širokou skupinu lidí a aby jej všechny nástroje CASE (computer-aided software engineering) mohly snadno implementovat. Toho je dosaženo pomocí unifikace výrazových prostředků a jasně daných pravidel pro jejich rozšiřování. Díky těmto vlastnostem se jazyk UML uplatnil i v jiných oblastech; např. business procesy a stal se standardem nejen pro navrhování softwarových řešení. Je však důležité si uvědomit, že jazyk UML nenabízí žádný druh metodiky modelování, nabízí "pouze" prostředky pro tuto činnost.

2.1 Historie UML

Historie jazyka UML začala v roce 1994 ve firmě Rational Software, kde se sešli Jim Rumbaugh a Grady Booch a začali spojovat své, v té době popularní, modelovací techniky pro návrh objektově-orientovaných systémů (Object-modeling technique a Booch method). Společně pak začali pracovat na sjednocené metodice a nazvali ji Unified Method. V roce 1995 se přidal ještě Ivar Jacobson, tvůrce OOSE (Object-oriented software engineering). V roce 1996 přišla firma Rational Software za těmito pány (sami si nechali říkat Three Amigos) s požadavkem na vytvoření neproprietálního sjednoceného modelovacího jazyka. Pod vedením výše zmiňovaných pak vznikla ještě v témže roce kompletní specifikace jazyka UML. V lednu 1997 byla specifikace verze 1.0 představena skupině OMG (Object Management Group) a v listopadu OMG přijalo verzi 1.1 jako standard, který postupně vytlačil ostatní analytické jazyky.

UML jazyk je v průběhu času významně vyvíjen a upravován, verze 1.4.2. se stala dokonce mezinárodním standardem (ISO/IEC 19501:2005 Information technology -- Open Distributed Processing -- Unified Modeling Language (UML) Version 1.4.2.). V současnosti je nejnovější verze 2.1.1. a jazyk se stal celosvětově používaným standardem pro zachycení požadavků a navrhování softwarových řešení.

2.2 Specifikace UML

UML je souhrn specifikací od skupiny OMG. Skládá se ze čtyř dokumentů: Diagram Interchange Specification, UML Infrastructure, UML Superstructure a Object Constraint Language. Všechny čtyři části jsou zdarma k dispozici na stránkách OMG a představují celou specifikaci tohoto jazyka.

2.2.1 Diagram Interchange Specification

Diagram Interchange Specification (až od verze 2.0) slouží ke snadnému sdílení UML modelů mezi různými softwarovými modelovacími nástroji. Ve verzích UML 1.x byl používán pro výměnu modelů formát XMI (XML Metadata Interchange), bohužel ale tento mechanismus nesplňoval požadavky pro výměnu modelů. Tento formát byl schopen přenášet pouze informace o elementech v diagramu, ale nebyl schopen uchovávat to, jak jsou elementy reprezentovány a v diagramu rozmístěny. S příchodem verze UML 2.0 přislo řešení v podobě přídavného balíčku zaměřeného na graficky orientované informace, což umožnilo nechat XMI[UML] metamodel nedotčený a zpětně kompatibilní.

2.2.2 UML Infrastructure

UML Infrastructure (až od verze 2.0) je nízkoúrovňový metamodel sloužící jako základ pro UML Superstructure. Specifikace ošetřuje jádro architektury, přidává profily a stereotypy. Většinou není tento dokument koncovými uživateli přímo využíván.

2.2.3 UML Superstructure

UML Superstructure (všechny verze) obsahuje přesnou definici UML prvků. Je to kompletní souhrn všech elementů a pravidel pro jejich používání. Celá specifikace je postavena nad UML Infrastructure. Tento dokument je nejobsáhlejší a je určen pro koncového uživatele jazyka.

2.2.4 Object Constraint Language

Object Constraint Language je specifikace určující jednoduchý jazyk na psaní omezení (constraints) a výrazů (expressions), které se hodí při potřebě specifikovat UML model pro určitou oblast, kde vyvstává potřeba určit, jaké hodnoty jsou pro jaký parametr povolené. OCL verze 1.4 definovala pouze jazyk omezení. OCL verze 2.0 byla rozšířena o obecný dotazový jazyk OQL (Object Query Language).

2.3 Diagramy v jazyce UML 2.0

Každý UML model se skládá z jednoho nebo více diagramů, kde každý diagram zachycuje jiný pohled na vyvíjený software. Aspekty žádného komplexnějšího návrhu nelze vyjádřit v jediném diagramu. Je běžné, že právě jedna idea je ztvárněna více diagramy a to z různých úhlů pohledu. "Diagram ve vizuální formě vypráví právě jeden konkrétní příběh o aplikaci."

STEIN, René . Návrh aplikací v jazyce UML [online]. Brno : Zoner software, $2003-2004 [cit. 2007-03-25]. Dostupný z WWW: <http://interval.cz/serial.asp?serial=18>. ISSN 1212-8651.

Verze UML 2.0 zavedla dělení diagramů do dvou skupin: strukturální diagramy (structural diagrams) a diagramy chování (behavioral diagrams). Strukturální diagramy slouží k vyjádření fyzické organizace prvků v systému a znázorňují, jak jeden objekt souvisí s druhým. Diagramy chování naproti tomu vystihují chování elementů v systému. Následuje stručný výpis diagramů. Detailnějším popisem některých z nich se budou zabývat následující kapitoly.

2.3.1 Strukturální diagramy

- Class diagrams

Diagramy tříd popisují jednotlivé třídy a rozhraní a zachycují mezi nimi statické vazby. Hloubka detailů se může diagram od diagramu lišit, od rychlých náčrtů tříd až po kompletní popis všech atributů a operací na třídách, což umožnuje automatické generování zdrojových kódů přímo z diagramu.

- Component diagrams

Diagramy komponent slouží k seskupení menších elementů, například tříd, do větších rozmístitelných částí s danou funkcí. Zaměřují se na detail implementace celého systému.

- Composite structure diagrams

Diagramy spojených struktur jsou novinkou v UML 2.0, zajišťující spojení diagramů tříd a komponent. Diagramy tedy slouží k vyjádření toho, jak se prvky v systému skládají do komplexnějších vzorů.

- Deployment diagrams

Diagramy nasazení slouží k náčrtu toho, jak bude systém realizován na konkrétních částech hardwaru.

- Object diagrams

Diagramy objektů používají stejnou syntaxi jako diagramy tříd, ale znázorňují, jak konkrétní instance tříd fungují s ostatními instancemi za běhu systému.

- Package diagrams

Diagramy balíčků seskupují jednotlivé třídy a rozhraní k sobě. Jedná se v podstatě o jmenné prostory (namespace v c#, package v jave).

2.3.2 Diagramy chování

- Activity diagrams

Diagramy aktivit jsou určeny k modelování toků v systému. Diagramy zachycují proces jako kolekci aktivit a přechodů mezi nimi.

- Sequence diagrams

Sekvenční diagramy znázorňují interakce mezi elementy v čase. V diagramech jsou zobrazeny typy a pořadí zpráv posílaných mezi objekty při běhu systému. Dále z nich lze vyčíst délku života objektu (object lifeline) a to, kdy se s objektem pracuje (focus of control).

- State machine diagrams

Stavové diagramy slouží k zachycení vnitřních stavů elementu a přechodů mezi nimi. Modeluje se pomocí nich konečný stavový automat, který znázorňuje životní cyklus daného elementu.

- Timing diagrams

Časovací diagramy slouží k modelování systémů běžících v reálném čase (real-time systems). Detailně popisují časový průběh zpráv .

- Use case diagrams

Diagramy případů užití slouží k zachycení funkčních požadavků na modelovaný systém. Tyto požadavky představují na pozdější implementaci nezávislý pohled toho, co má systém umět dělat. V podstatě případy užití vytvářejí rámec kolem celého systému.

2.4 Pohledy v jazyce UML 2.0

Takzvané pohledy (Views) nejsou přímo součástí UML, koncepce pohledů na systémy usnadňují návrháři vybrat diagram podle toho, jaké informace do něj chce zanést. Celý model je většinou reprezentován ve čtyřech oddělených pohledech a v jednom speciálním pohledu, který říká, jak spolu v modelu všechno souvisí.

- Deployment view

Pohled nasazení zachycuje konfiguraci systému, jeho instalaci a následný běh.

Většinou se skládá z diagramů komponent, nasazení a diagramů interakce. V pohledu nasazení je ukázáno, jak fyzická hardwarová vrstva komunikuje a vykonává daný systém.

- Design view

V návrhovém pohledu zachycujeme, jak je okruh působnoti systému reprezentován v softwarovém řešení. Návrhový pohled skoro vždy používá diagramy tříd, objektů, diagramy spojených struktur a sekvenční diagramy pro vyjádření návrhu systému.

- Implementation view

Implementační pohled zdůrazňuje, jak jsou komponenty, soubory a jakékoliv zdroje využívány systémem. Tento pohled ukazuje, jak jaká komponenta spolupracují s jinou, jaké zdrojové soubory implemetují jakou třídu atp. Implementační pohled využívá většinou diagramů komponent a spojených struktur.

- Process view

Procesní pohled se používá k zachycení konkurence a funkčních plnění. Pohled používá diagramy interakcí a aktivit na znázornění toho, jak se daný systém chová za běhu.

- Use case view

Pohled případů užití slouží k zachycení požadavků koncového uživatele. Pohled většinou spojuje pomocí diagramů spolupráce všechny čtyři předešlé uvedené pohledy. Pohled případů užití obsahuje tedy diagramy spolupráce a diagramy případů užití.

3 Diagramy případů užití

Na začátku každého návrhu softwaru stojí požadavky zadavatele. Pro zpacovaní těchto požadavků se využívají diagramy případů užití. Ty slouží k zachycení funkcionality systému a tím představují pomyslné hranice, ve kterých se celý systém pohybuje. V těchto diagramech vystupují pouze případy užití, účastníci a vztahy mezi nimi (viz Obrázek 3.1. : Diagram případů užití).

3.1 Hranice systému

"První věcí, kterou musíte udělat, přemýšlíte-li o tvorbě softwarových systémů, je stanovení jeho hranic. Jinými slovy to znamená, že musíte určit, co je součástí systému (uvnitř jeho hranic) a co naopak není jeho součástí (za hranicemi systému). Zní to docela samozřejmě, ale v mnoha projektech vznikly problémy právě v důsledku neurčitých hranic systému. Stanovení hranic systému má obrovský dopad na funkční požadavky (leckdy i na požadavky nefunkční). Už jsme se měli možnost mnohokrát přesvědčit, že neúplné nebo dokonce špatně specifikované požadavky mohou způsobit krach projektu."

ARLOW, Jim, NEUSTADT, Ila. UML a unifikovaný proces vývoje aplikací s. 55. 2003. vyd. Brno Computer Press 2003. 408 s. ISBN 80-7226-947-X.

Hranice systému se zobrazují jako rámeček, ve kterém jsou obsaženy případy užití. Účastníci v systému stojí mimo jeho hranice.

3.2 Účastníci

Účastníci představují v diagramech případů užití spouštěcí mechanismus pro různé funkcionality systému. Účastníkem může být prakticky cokoli, co využívá nebo spouští funkce systému. Účastníci jsou znázorněni ikonou figurky, pod níž je uvedeno jejich jméno.

"Účastník specifikuje roli, kterou určitá externí entita příjmá v okamžiku, kdy

začíná daný systém bezprostředně používat. Může vyjadřovat roli uživatele, roli dalšího systému, který se dotýká hranic vašeho systému."

ARLOW, Jim, NEUSTADT, Ila. UML a unifikovaný proces vývoje aplikací s.56. 2003. vyd. Brno Computer Press 2003. 408 s. ISBN 80-7226-947-X.

3.3 Případy užití

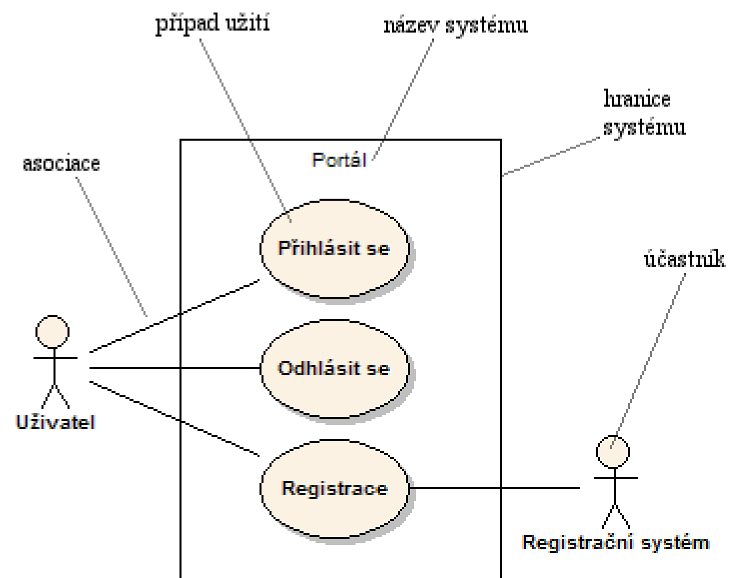
Případy užití vyjadřují jednu funkcionalitu v systému. Zobrazují se v oválu, uvnitř něhož je popis požadované činnosti, což představuje zároveň i jméno případu užití.

"James Rumbaugh definuje v knize The Unified Modeling Language Reference Manual (Referenční příručka jazyka UML) případ užití jako specifikaci posloupností činností, včetně proměnných posloupností a chybových posloupností, které systém, podsystém, nebo třída může vykonat prostřednictvím interakce s vnějšími (externími) účastníky."

ARLOW, Jim, NEUSTADT, Ila. UML a unifikovaný proces vývoje aplikací s.58. 2003. vyd. Brno Computer Press 2003. 408 s. ISBN 80-7226-947-X.

3.4 Asociace

Asociace je vztah mezi účastníkem a případem užití. Relace vyjadřuje volání funkcionality účastníkem, předání výsledku funkcionality účastníkovi, nebo oboje. Diagram se čte zleva doprava, kde účastníci, kteří volají, jsou vlevo od případu užití a účastníci, kteří příjímají výsledek, jsou vpravo. Asociace se znázorňuje pomocí plné čáry a lze u ní uvést otevřenou šipku ve směru vztahu. To se hodí pro složitější modely.



Obrázek 3.1. : Diagram případů užití

4 Diagramy tříd

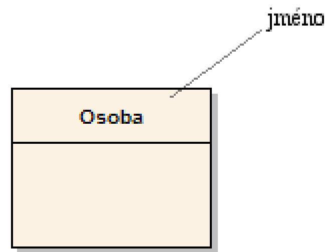
Diagramy tříd jsou statické diagramy, které zobrazují strukturu systému pomocí tříd a vztahů (relationships) mezi nimi. Každá třída může volitelně zobrazovat svoje operace a atributy. Také relace mezi třídami mohou poskytovat další informace o struktuře systému. V této kapitole se seznámíme se syntaxí diagramů tříd.

Třída představuje obecný předpis pro množinu objektů, které sdílejí společné atributy a operace. V jazyce UML je třída reprezentována obdélníkem rozděleným na tři části. V první části je uvedeno jméno třídy, v druhé jsou pak vypsány atributy a ve třetí operace. Podle libosti lze jakoukoli část vynechat, neuvést ji. V praxi se toho využívá velmi často, hlavně v počátcích návrhu není vždy třeba zabývat se přílišnými detaily. Jak se život modelu prodlužuje, vybrušují se i detaily. Stejně jako lze vynechat celou část popisu třídy, může být vynechán atribut nebo operace.

4.1 Jméno

Úplně nejjednodušší reprezentace třídy je označení pouze jménem (viz Obrázek 4.1. : Nejjednodušší reprezantace třídy). Pro pojmenování třídy v jazyce UML jsou daná tyto pravidla :

- Název musí začínat velkým písmenem (nejvíce se využívá styl camel case)
- Název musí být v horní části
- Název musí být napsán tučným písmem
- Jestliže je třída abstraktní, musí být její název napsán kurzívou

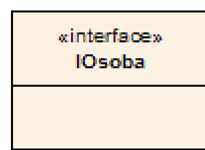


Obrázek 4.1. : Nejjednodušší reprezantace třídy

-

4.1.1 Stereotypy třídy

Stereotyp mění smysl elementu, u kterého je aplikován. Zobrazuje se nad názvem třídy v dvojitých lomených závorkách (viz Obrázek 4.2. : Sterotyp).



Obrázek 4.2. : Sterotyp

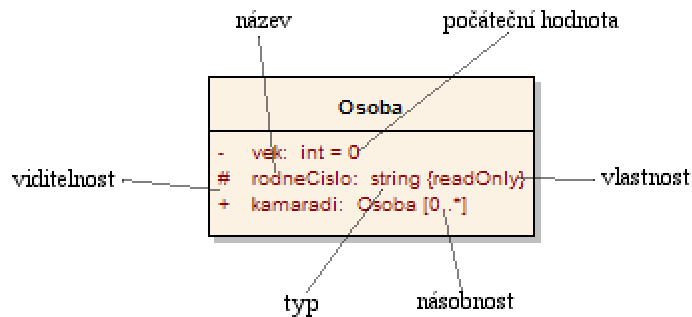
4.2 Atributy

Další částí, kterou třída může obsahovat, jsou atributy, které reprezentují datové detaily třídy. Zobrazit atribut lze pomocí dvou způsobů; tzv. inline nebo u relace.

4.2.1 Inline atribut

Inline atributy jsou umístěny v sekci hned pod jménem třídy, tzv. v těle třídy. Zápis inline atributu má takovýto tvar (viz Obrázek 4.3. : Třída se třemi atributy) :

```
viditelnost název : typ násobnost = počáteční hodnota {vlastnosti}
visibility name : type multiplicity = default {properties}
```



Obrázek 4.3. : Třída se třemi atributy

- viditelnost

Znázornění viditelnosti atributu. Používá se notace + pro veřejnou (public), - pro soukromou (private), # pro chráněnou (protected) a ~ balíčkovou (package) viditelnost atributu.

- název

Udává jméno atributu. Většinou začíná malým písmenem a při notaci jména se dodržuje styl camel case.

- typ

Toto pole udává typ atributu. Typ může být buď primitivum (int, float, boolean atd.), nebo reference na komplexnější typ (třída, rozhraní, enum atd.).

- násobnost

Určuje, kolik instancí typu je obsaženo v atributu. Pokud je instance pouze jedna, lze toto pole vynechat. Jinak se počet udává ve hranatých závorkách, buď jako jedno číslo nebo rozsah čísel.

- počáteční hodnota

Pole udává počáteční (defaultní) hodnutu atributu.

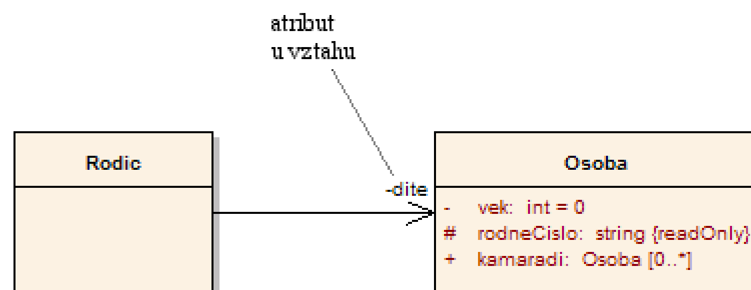
- vlastnosti

Vlastnosti představují soubor omezení a upřesnění pro atribut. Zobrazují se ve

složených závorkách a oddělují se čárkou.

4.2.2 Atribut u relace

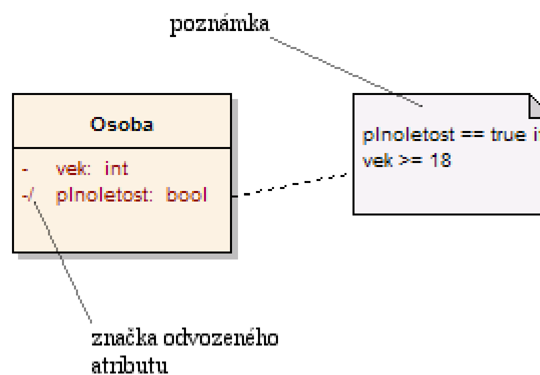
Atributy lze zobrazovat také pomocí označením u relace. Takovýto zápis vede ke komplexnějším a podrobnějším schématům. Atribut se zobrazí u jednoho z typů relace mezi dvěma třídami (viz Obrázek 4.4. : Atribut znázorněný u relace). Pro zápis platí stejná pravidla jako u inline atributů.



Obrázek 4.4. : Atribut znázorněný u relace

4.2.3 Odvozený atribut

Odvozený atribut dovoluje popsat závislost atributu na atributu jiném. Značí se lomítkem (/) umístěným před jménem atributu. Dále je možno uvést omezující podmínku v poznámce (viz Obrázek 4.5. : Odvozený atribut s poznámkou).

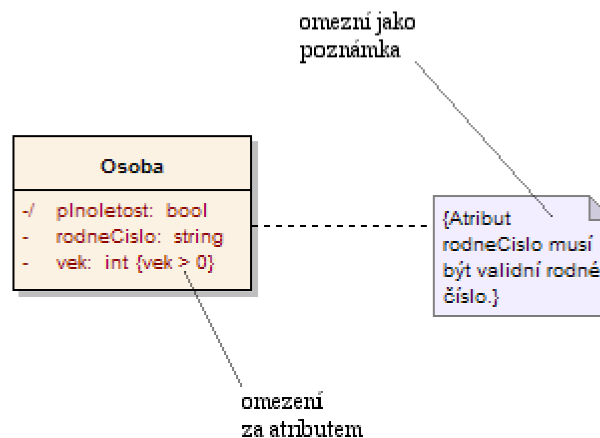


Obrázek 4.5. : Odvozený atribut s poznámkou

4.2.4 Omezení atributů

Omezení atributů slouží k vymezení speciálních požadavků u elementu.

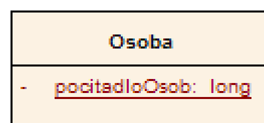
Omezení může být napsáno v jazyce OCL, nebo v normální řeči, avšak musí vyjadřovat svým zněním boolovský výraz. Zobrazuje se buď ve složených závorkách za atributem (v sekci vlastností atributů), nebo v poznámce, která je spojená přerušovanou čarou s omezovaným elementem (viz Obrázek 4.6. : Omezení atributů). Omezení může být pojmenováno tak, že před boolovský výraz přidáme dvojtečku a jméno omezení.



Obrázek 4.6. : Omezení atributů

4.2.5 Statické atributy

Statické atributy jsou atributy náležící ke třídě, nikoli k instanci třídy. Znázorňují se pomocí podtržení celého zápisu atributu (viz Obrázek 4.7. : Statický atribut).



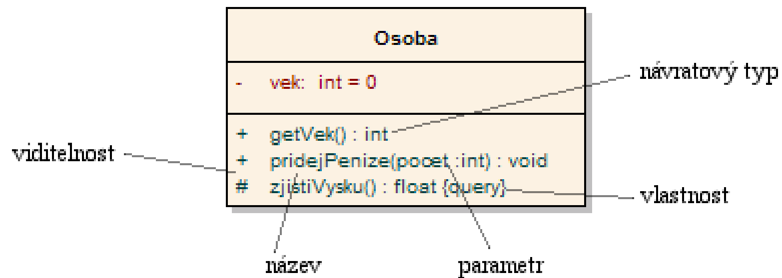
Obrázek 4.7. : Statický atribut

4.3 Operace

Operace jsou vlastnosti vázané k dané třídě a představují určité chování, jsou to v podstatě funkce nad třídou. Zobrazují se do oddílu operací, který následuje hned za oddílem atributů (když je uveden). Operace se znázorňuje následovně (viz Obrázek 4.8.

: Třída se třemi operacemi) :

```
viditelnost název(parametry) : návratovýTyp {vlastnosti}
visibility name(parameters) : return-type {properties}
```



Obrázek 4.8. : Třída se třemi operacemi

- viditelnost a název

Pro viditelnost a název platí stejná pravidla jako u atributů.

- parametry

Pro zobrazování parametru platí obdobná pravidla jako pro atributy, s tím rozdílem, že se neuvádí viditelnost. Parametry se zapisují ve tvaru :

```
název : typ násobnost = počáteční hodnota {vlastnosti}
name : type multiplicity = default {properties}
```

- návratový typ

Návratový typ je informace, kterou operace po jejím provedení vrátí. Jestliže se operace takto nechová, její návratový typ je prázdný (void). Typ může být buď primitivum (int, float, boolean atd.), nebo reference na komplexnější typ (třída, rozhraní, enum atd.).

- vlastnosti

Oddíl obsahuje omezení a upřesnění vztahující se k operaci. Vlastnosti se zobrazují stejně jako obdobný oddíl u atributů.

4.3.1 Omezení operací

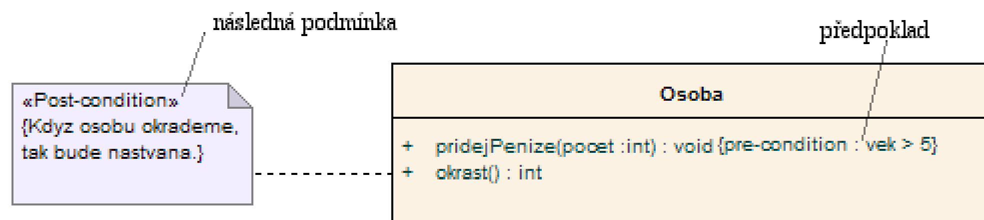
Operace může obsahovat několik omezení, která vymezují interakci s okolím a pozdější implementaci operace. Omezení se zobrazuje stejným způsobem jako omezení u atributů. Omezení lze zapisovat buď pomocí OCL, pseudo kódu, nebo obyčejným textem.

- Předpoklady

Předpoklady říkají, v jakém stavu musí systém být před tím, než lze operaci zavolat. V omezení uvedeme jenom stav souvisejících atributů, nebo stav třídy, který operaci vlastní.

- Následné podmínky

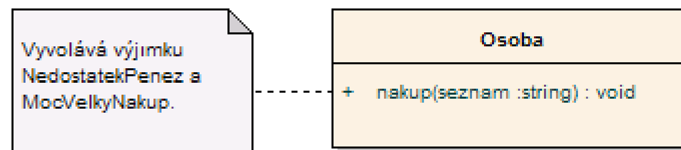
Následné podmínky garantují, v jakém stavu se systém nachází po provedení celé operace. Podmínky se většinou vztahují ke třídě, která operaci vlastní (viz Obrázek 4.9. : Omezení operací).



Obrázek 4.9. : Omezení operací

4.3.2 Výjimky

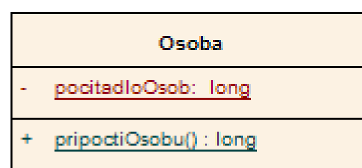
Většina moderních programovacích jazyků pracuje s výjimkami (exceptions), a proto lze u operace uvést, jaké výjimky vyvolává. K zobrazení se používá, podobně jako u omezení, poznámka, která je spojená přerušovanou čarou s operací (viz Obrázek 4.10. : Výjimky).



Obrázek 4.10. : Výjimky

4.3.3 Statické operace

Statické operace náležící ke třídě, nikoli k instanci třídy. Znázorňují se pomocí podtržení celého zápisu operace (viz Obrázek 4.11. : Statická operace).

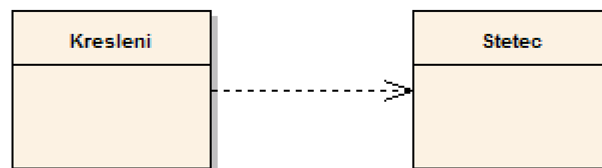


Obrázek 4.11. : Statická operace

4.4 Relace

Relace znázorňují různé vztahy mezi třídami. Bez nich by model tříd nenesl žádnou informaci o chování systému.

4.4.1 Závislost (Dependency)



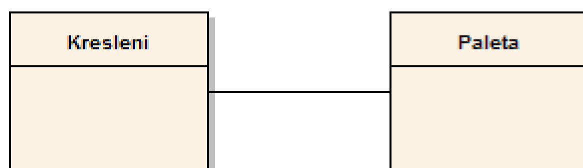
Obrázek 4.12. : Závislost

Nejslabší relací je závislost. Říká nám pouze to, že jedna třída používá druhou, a to většinou po přechodnou dobu. Znázorňuje se pomocí přerušované čáry s šipkou ve směru vztahu (viz Obrázek 4.12. : Závislost).

4.4.2 Asociace (Association)

Asociace je silnějším druhem relace než závislost. Představuje vztah mezi

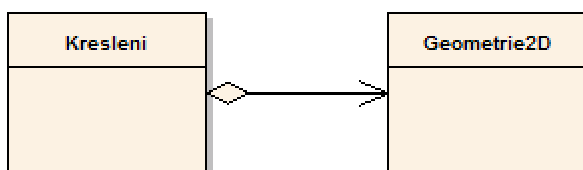
třídami, který trvá delší dobu. Avšak životnost objektů spojených asociací na sobě nezávisí, jeden objekt může být zničen bez toho, aby byl zničen druhý. Asociace se znázorňuje plnou čarou mezi třídami (viz Obrázek 4.13. : Asociace). Pokud je asociace jednosměrná, může být na konci čáry uvedena šipka ve směru vztahu. Bez šipek se jedná o asociaci obousměrnou.



Obrázek 4.13. : Asociace

4.4.3 Agregace (Aggregation)

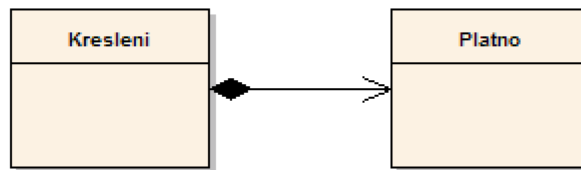
Agregace vyjadřuje silný vztah - jedna třída vlastní druhou, tj. vytváří její instanci, a ta pak spolu s vlastníkem zaniká. Agregace se zobrazuje plnou čarou s kosočtvercem u vlastnící třídy a šipkou po směru vztahu (viz Obrázek 4.14. : Agregace).



Obrázek 4.14. : Agregace

4.4.4 Kompozice (Composition)

Kompozice je nejsilnější vztah mezi třídami. Říká nám, že jedna třída je částí třídy druhé. To znamená, že třídy bez sebe nemohou existovat a při zániku jedné zaniká i druhá. Agregace se zobrazuje jako plná čára s vyplněným kosočtvercem na straně vlastnící třídy a šipkou po směru vztahu (viz Obrázek 4.15. : Kompozice).



Obrázek 4.15. : Kompozice

4.4.5 Zobecnění (Generalization)

Zobecnění představuje vztah mezi obecnější třídou (předpisovou) a třídou konkrétnější, přičemž konkrétnější třída naprosto splňuje svým obsahem třídu obecnější, a tudíž je s ní zaměnitelná (naopak to neplatí). Generalizace se znázorňuje jako plná čára s uzavřenou šipkou na konci, kde je obecnější třída (viz Obrázek 4.16. : Zobecnění).

Obrázek 4.16. : Zobecnění

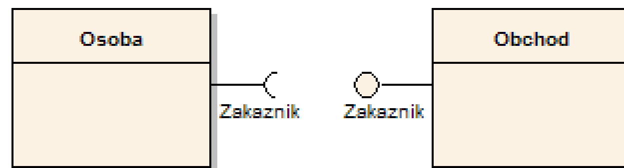
4.5 Rozhraní

"Rozhraní specifikuje pojmenovatelnou množinu operací. Jeho podstata spočívá v oddělení specifikace funkčnosti (rozhraní) od fyzické implementace prostřednictvím klasifikátoru, jakým je například třída nebo podsystém. Rozhraní definuje dohodu, kterou implementuje klasifikátor."

ARLOW, Jim, NEUSTADT, Ila. UML a unifikovaný proces vývoje aplikací s.292. 2003. vyd. Brno Computer Press 2003. 408 s. ISBN 80-7226-947-X.

Rozhraní lze zobrazovat dvěma způsoby. Buď pomocí stereotypu (viz Obrázek

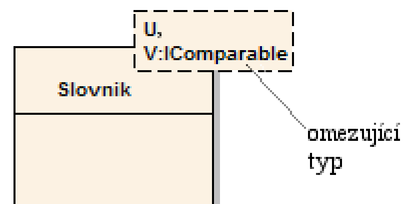
4.2. : Sterotyp), nebo pomocí kuličky a zástrčky. Pomocí druhého způsobu však nejsou vidět názvy operací. Zobrazuje se pouze pomocí kuličky, u které je název rozhraní, a zástrčky u třídy, která na daném rozhraní závisí (viz Obrázek 4.17. : Rozhraní).



Obrázek 4.17. : Rozhraní

4.6 Šablony

V UML lze znázornit šablony pomocí přerušovaného obdélníku v pravém horním rohu třídy, který obsahuje jméno šablonového typu. V případě více typů je lze oddělit čárkou. Pokud je třeba omezit, jaký typ může uživatel použít, přidá se za jeho název dvojtečka a název omezujícího typu (viz Obrázek 4.18. : Šablona).



Obrázek 4.18. : Šablona

5 Sekvenční diagramy

Největší vylepšení, která přinesla verze jazyka UML 2.0, jsou v oblasti diagramů znázorňující různé interakce mezi elementy modelu. Právě sekvenční diagramy jsou k těmto účelům jedny z nejběžněji používaných.

Sekvenční diagramy znázorňují interakce mezi elementy v čase. V diagramech jsou zobrazeny typy a pořadí zpráv posílaných mezi objekty při běhu systému. Dále z nich lze vyčíst délku života objektu (object lifeline) a to kdy se s objektem pracuje (focus of control). V horní části diagramu jsou vedle sebe zobrazeny objekty, z kterých vede vertikálně dolů jejich čára života (lifeline). Zprávy mezi objekty jsou potom zobrazeny v horizontálním směru mezi čarami života.

5.1 Objekty a čáry života

Objekty se znázorňují jako obdélníky, z kterých vertikálně dolů vychází jejich čára života, která říká, jak dlouho objekt existuje. Název se zapisuje dovnitř obdélníku touto notací (viz Obrázek 5.1. : Čára života):

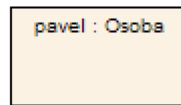
```
jméno objektu : jméno třídy
object name : class name
```

- Jméno objektu

Jméno objektu představuje jméno konkrétní instance, začíná malým písmenem a pro zápis se využívá styl camel case.

- Jméno třídy

Pro jméno třídy se používají stejná pravidla jako u diagramů tříd (viz 4.1 Jméno).



Obrázek 5.1. : Čára života

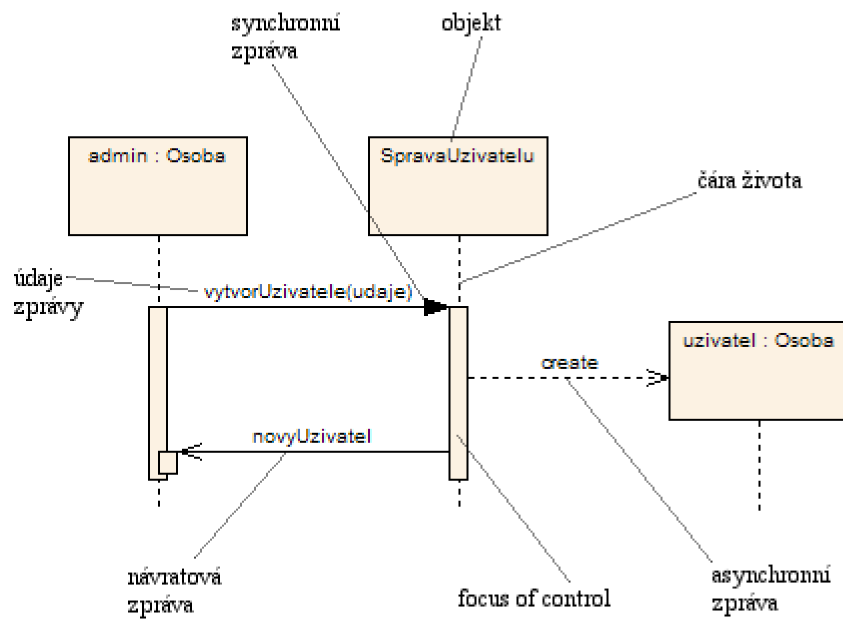
5.2 Zprávy

Zprávy zajišťují komunikaci mezi čarami života objektů. V podstatě se jedná o volání operací mezi dvěma objekty. Zprávy se zobrazují pomocí plné čáry s šipkou na konci ve směru volání. Je-li volání asynchronní (volající objekt nečeká na odpověď a zprávu se okamžitě zpracuje) šipka na konci je otevřená. Je-li naopak volání synchronní, šipka na konci je vyplněná a uvádí se návratová hodnota buď pomocí přerušované čáry ve zpětném směru, nebo v syntaxi volání (viz Obrázek 5.2. : Složitější sekvenční diagram). Syntaxe volání se znázorňuje nad čarou zprávy v tomto tvaru :

```
atribut = jméno (argumenty) : návratová hodnota
```

5.3 Práce s objektem

To kdy se s objektem pracuje (focus of control), lze znázornit pomocí obdélníku, který leží na čáře života. To vyjadřuje délku času, kdy se s objektem pracuje, což záleží na synchronním, nebo asynchronním volání.



Obrázek 5.2. : Složitější sekvenční diagram

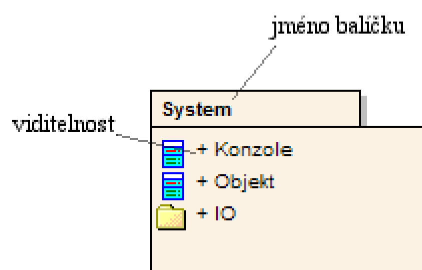
6 Diagramy balíčků

"Balíček je abstrakce sdružování - je to kontejner a vlastník modelovaných elementů. Každý balíček má vlastní jmenný prostor, uvnitř nějž musí být všechny názvy jedinečné. Balíček je ve skutečnosti univerzálním mechanismem uspořádání elementů a diagramů do skupin."

ARLOW, Jim, NEUSTADT, Ila. UML a unifikovaný proces vývoje aplikací s.179. 2003. vyd. Brno Computer Press 2003. 408 s. ISBN 80-7226-947-X.

Diagramy balíčků jsou užitečným nástrojem pro strukturování a zpřehlednění modelovaných systémů. K elementu obsaženém v nějakém balíčku se v UML přistupuje pomocí dvou dvojteček (::), například stejně jako v jazyce C++.

Balíčky se zobrazují pomocí obdélníku se záložkou v levém horním rohu. Jméno se pak zapisuje buď do záložky, nebo dovnitř obdélníku. Elementy, které balíček obsahuje, lze zobrazit dovnitř obdélníku (viz Obrázek 6.1. : Jednoduchý balíček), jméno se pak zapisuje do záložky. Pokud potřebujeme u elementů uvést přesnější detaily, zakreslíme je vně a s balíčkem je spojíme pomocí plné čáry se symbolem plus uvnitř kruhu.



Obrázek 6.1. : Jednoduchý balíček

6.1 Viditelnost

V balíčku lze znázornit u vnořených elementů privátní (private) nebo veřejnou (public) viditelnost. Pokud má element veřejnou viditelnost, je dosažitelný i z venku balíčku. Pokud má ovšem viditelnost privátní, lze s ním pracovat jenom v rámci balíčku. Veřejná viditelnost se zobrazuje pomocí symbolu plus, kdežto privátní pomocí

znaménka mínus (viz Obrázek 6.1. : Jednoduchý balíček).

6.2 Speciální zprávy

V sekvenčních diagramech jsou ještě dvě speciální zprávy, tzv. výchozí bod a konečný bod. Jedná se o zprávy, u kterých není znám odesílatel, resp. příjemce. Používají se pro zjednodušení diagramu u takových případů, kdy se není třeba starat, jak a odkud zpráva přišla, resp. kam směřuje a co se s ní bude dále dít. Místo toho, aby zpráva začínala nebo končila na čáře života, vychází z černě vypleného kruhu nebo tam končí.

6.3 Vztahy mezi balíčky

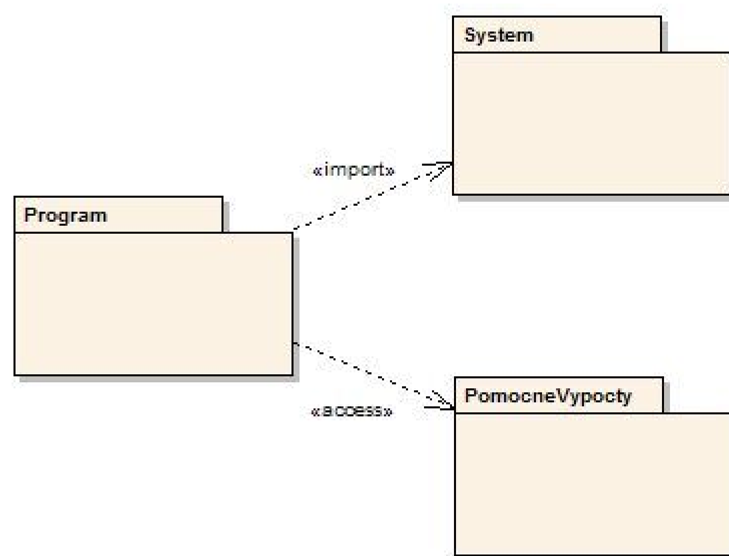
Pokud je potřeba přistupovat z elementu z jednoho balíčku k elementu v jiném balíčku, je třeba uvést celé jméno elementu, tj. i se jménem balíčku. Tím vzniká vztah mezi balíčky, který se zobrazuje pomocí přerušované čáry s šipkou po směru vztahu (viz Obrázek 6.2. : Vztahy mezi balíčky).

6.3.1 Importování

K zjednodušení přístupu elementů může být balíček importován. V tomto případě není potřeba uvádět před názvem elementu jméno balíčku. Tento vztah se znázorní pomocí stereotypu <<import>> u šipky vztahu.

6.3.2 Přistupování

Podobně jako v případě import slouží i přístupový vztah ke zjednodušení přístupu. Na rozdíl od importování slouží access pouze balíčku, který k němu přistupuje, tj. elementy z přistupovaného balíčku obdrží privátní viditelnost. Zobrazuje se pomocí stereotypu <<access>>.



Obrázek 6.2. : Vztahy mezi balíčky

7 Návrh

Projekt Stereopická projekce vznikl za účelem využití tohoto jevu při učení. Hlavním cílem bylo vytvořit kompletní řešení pro učitele tak, aby neměli problémy s integrací výstupu do běžně používaných nástrojů pro tvorbu prezentací nebo jiných výukových materiálů, třeba i na papír. Protože vývoj aplikací, které učitelé pro svoji práci využívají, se stále zvyšuje a je tudíž i různorodější, bylo třeba navrhnout software, který bude plně rozšiřitelný, kde jednotlivé funkce systému půjde snadno přidat pomocí takzvaných pluginů.

Celý systém je postaven modulárně, přičemž jádro systému zprostředkovává komunikaci. Modul představuje zásuvku pro pluginy, soubor funkcionalit systému. Projeví se v grafickém uživatelském rozhraní. Moduly komunikují s pluginy, které představují jednu faktickou funkci, kterou modul využije. Tímto je zajištěna snadná rozšiřitelnost jednotlivých modulů, ale i možnost přidat nebo změnit stávající modul, a tím do celé aplikace přidat úplně nový prvek. V rámci návrhu se počítalo s následujícími základními moduly : Animator, Loader, Physics, Renderer.

Na začátku tvorby návrhu bylo zapotřebí vytvořit komunikační rozhraní mezi modulem a pluginem, ale i jádrem a modulem. Jednalo se v podstatě o klíčové rozhodnutí, jelikož funkcionalita modulů bude realizovatelná pouze v rámci těchto rozhraní a musí uspokojit všechny možné situace, které budou muset moduly řešit.

Celý návrh přihlíží ke skutečnosti, že bude software implemetnován v jazyce C# a bude využívat výhod platformy Microsoft .NET. Pro vytvoření diagramů jsem použil nástroj Enterprise Architect, jenž umí pracovat s nejnovějšími verzemi jazyka UML. Tento nástroj CASE se dále vyznačuje schopnostmi reverse a forward engineeringu, a to pro širokou škálu programovacích jazyků, ale hlavně pak pro jazyk C#.

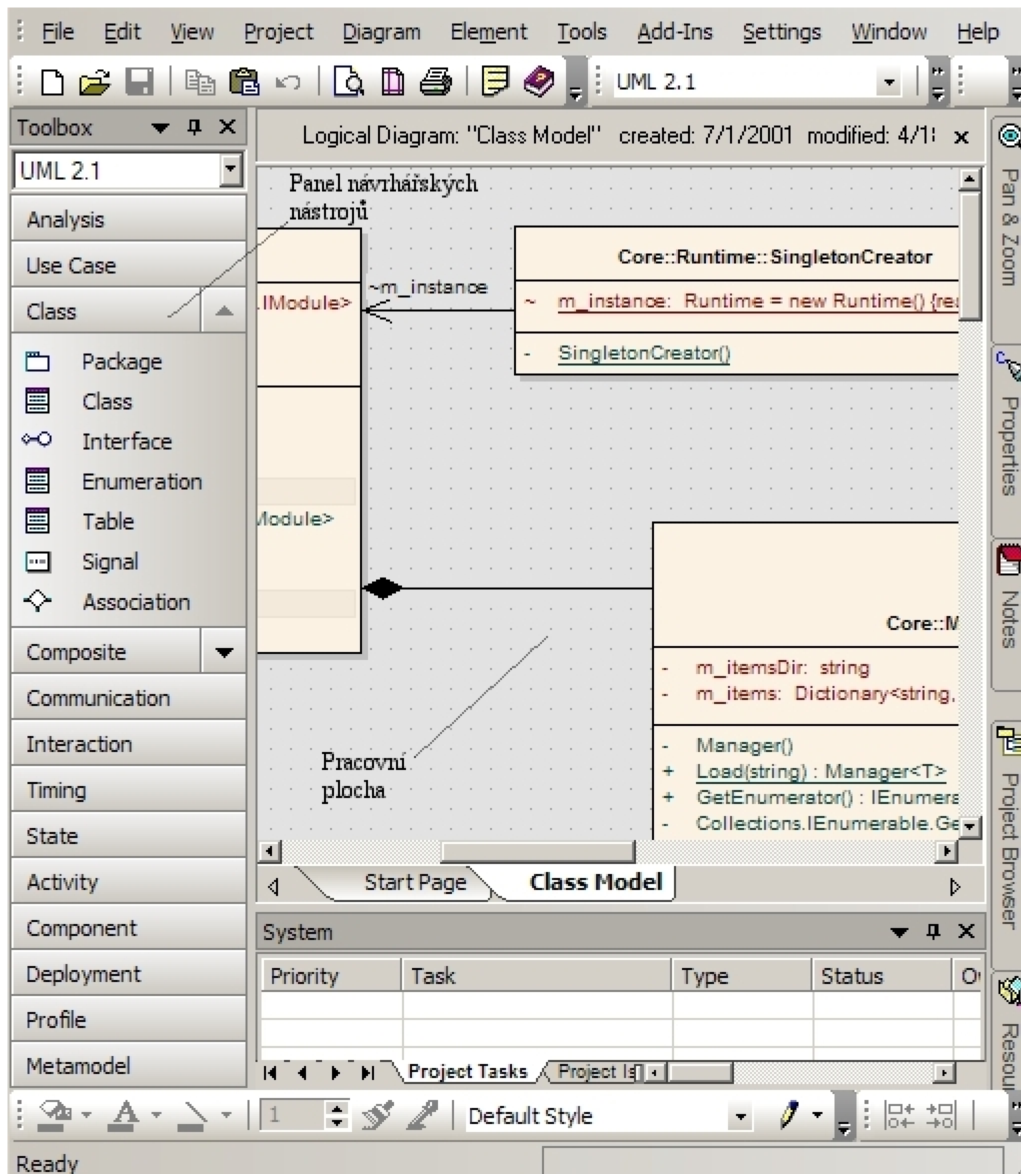
7.1 CASE nástroj Enterprise Architect

Enterprise Architect (EA) je vysoce výkonný nástroj pro modelování UML diagramů, podporuje nejnovější verze tohoto jazyka. Mezi jeho přednosti patří intuitivní grafické uživatelské rozhraní (viz Obrázek 7.1. : Prostředí Eterprise Architect), vysoký výkon, pokročilé modelovací techniky a kompletní řešení pro práci v týmech. Navíc nemá problém s během na normálních desktopových stanicích, takže se hodí i pro návrh malých a středně velkých projektů.

"Enterprise Architect je komplexní nástroj pro analýzu a návrh UML diagramů, pokrývající celý vývoj softwaru od shromáždění požadavků přes analytické fáze, navrhování modelů, testování až po celkové udržování. EA je víceuživatelský, pro platformu Windows, grafický nástroj vytvořený, aby pomáhal vytvářet robusní a udržovatelný software. Jeho vlastnosti jsou flexibilní s vysokou kvalitou výstupní dokumentace. Uživatelská příručka je k dosažení online."

Enterprise Architect - UML Design Tools and UML CASE tools for software [online]. 2000 , 2007 [cit. 2007-04-18]. Dostupný z WWW: <http://www.sparxsystems.com.au/products/ea.html>.

Jednou z nejlepších vlastností EA je možnost forward a reverse code engineering. To znamená, že lze snadno z navržených diagramů vygenerovat zdrojové kódy, a to do široké škály dnes běžně používaných jazyků (C++, C#, Java, Delphi, VB.Net, Visual Basic, ActionScript, PHP). V EA však funguje i opačný postup, takže není problém podívat se jak vypadá diagram námi naprogramovaných aplikací. To vše doplňuje spolupráce s různými vývojářskými aplikacemi, tudíž není žádný problém se synchronizací modelů a zdrojových kódů. Prostředí EA obsahuje i textový editor se zvýrazňováním syntaxe, což umožňuje snadné prohlížení zdrojových kódů přímo z prostředí modelovacího nástroje. Přímo z EA lze také vygenerovat kompletní dokumentaci k projektu.

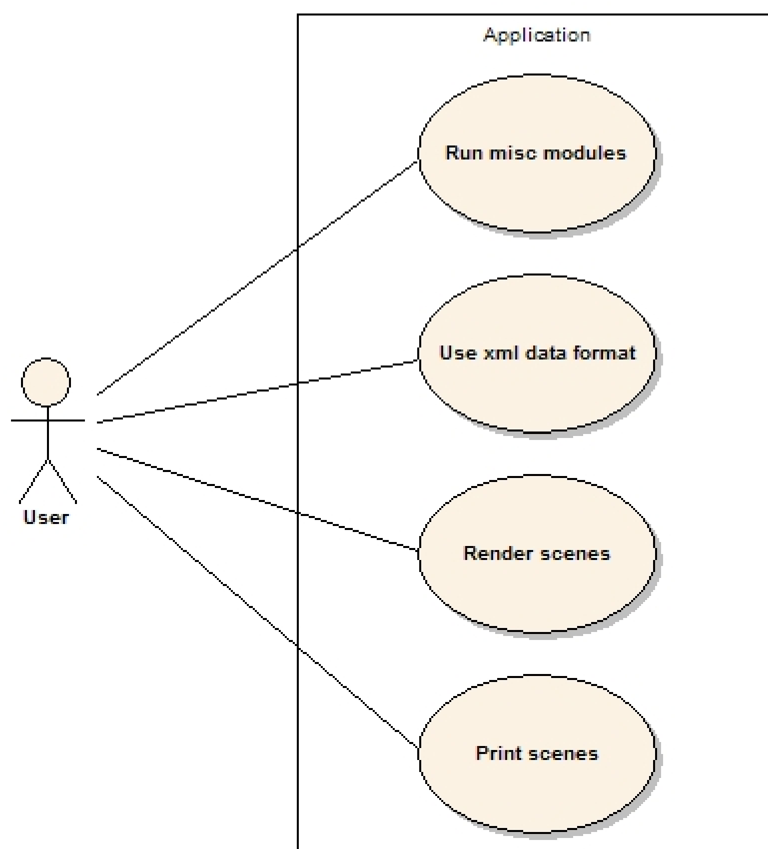


Obrázek 7.1. : Prostředí Eterprise Architect

7.2 Požadavky na software

To jak je nakonec celý systém navržen určují zadané požadavky. Z nich je třeba určit hranice systému, které v podstatě určují funkční požadavky na software. Největším požadavkem na celý systém byla jeho modulárnost a možnost přidávat jednotlivým modulům pluginy, což zaručuje atraktivnost programu, protože kdokoli má zájem

doplnit systém o určitou funkcionalitu, může tak učinit. Dalším požadavkem byla spolupráce s jinými aplikacemi. Tu zajišťují jednotlivé moduly a pluginy, proto kdykoli vyvstane potřeba spolupráce s jinou aplikací, lze snadno doplnit modul, nebo jen příslušný plugin. Dále systém musí umět různé druhy renderování a musí mít snadno pochopitelný datový formát na bázi XML (viz Obrázek 7.2. : Use case diagram).



Obrázek 7.2. : Use case diagram

7.3 Komunikační rozhraní

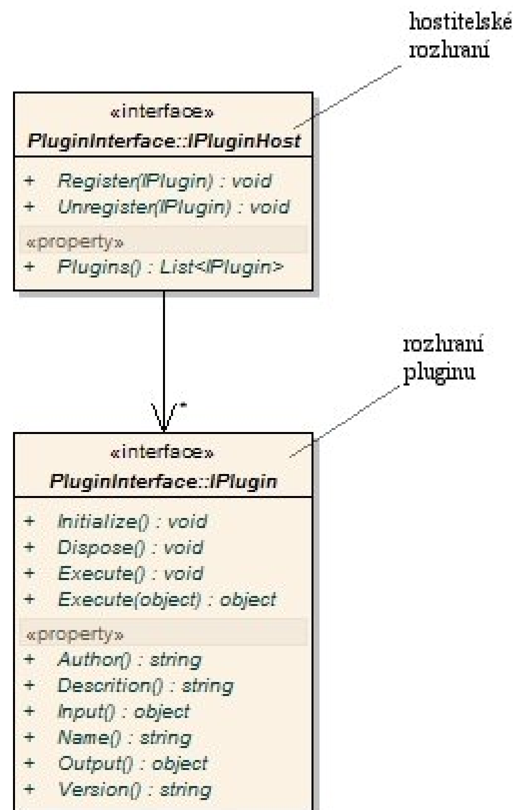
Na začátku návrhu bylo třeba vytvořit komunikační rozhraní mezi pluginem, modulem a jádrem. Navržení takovýchto rozhraní představuje v modulárním systému základní potřebu. Je důležité návrh provést pečlivě, jelikož během vývoje by změny rozhraní mohly znamenat časově nákladnou činnost, spojenou s úpravou všech doposud napsaných modulů nebo pluginů. Rozhraní entity je odděleno od implementace

funkčnosti. Tyto rozhraní určují komunikační rámec, který musí uspokojit všechny funkcionality, které budou moduly a pluginy řešit. Představují abstraktní způsob, jakým jednotlivé elementy komunikují s okolním světem. V důsledku jsou vnější komunikační metody odděleny od interních operací, což umožnuje měnit modul nebo plugin bez zásahu do ostatních entit, s kterými komunikují. Rozhraní jsou základním prvkem pro každý modulární systém. V kódu se projevují jako soubor prázdných metod, které musí každá třída využívající rozhraní implementovat. Za všechny tyto nesporné výhody se však platí navýšením spotřeby výkonu procesoru.

7.3.1 Rozhraní pluginů

Rozhraní pluginů slouží pro komunikaci modulů a pluginů. Modul v tomto případě představuje hostitele pro pluginy. Skládá se ze samotného rozrahní pluginu a rozhraní pro hostitele pluginu, pomocí kterého modul komunikuje s pluginem (viz Obrázek 7.3. : Rozhraní pluginů). K jednomu hostiteli může být takto připojen libovolný počet pluginů. Pluginy nemohou svého hostitele zpětně ovládat, jenom mu předají výstup ze své funkce. Modul si plugin připojí pomocí metody Register, které se připojovaný plugin předává jako parametr. Stejně tak může hostitel plugin odregistrovat pomocí metody Unregister. Tento přístup jsem zvolil z důvodu možnosti dynamického připojování a odpojování pluginů za běhu programu.

Samotné rozhraní pluginu vyžaduje implementaci těchto vlastností (Properties): jméno autora, popis, jméno a verzi. Dále rozhraní požaduje implementaci metod Execute, které provádí danou funkci pluginu, a implementaci inicializační a destrukční metody.



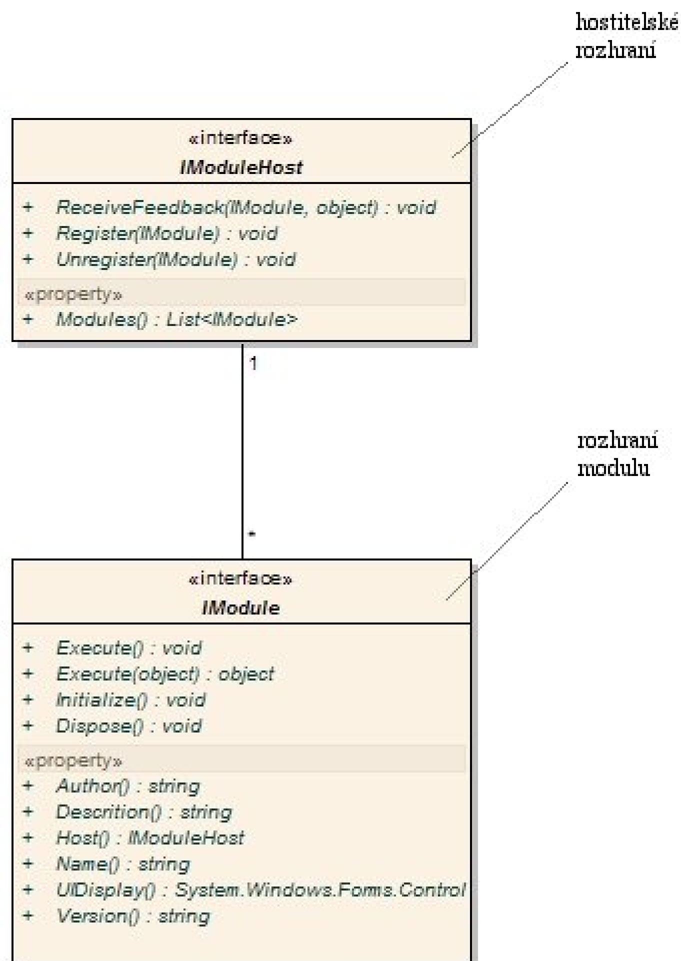
Obrázek 7.3. : Rozhraní pluginů

7.3.2 Rozhraní modulů

Rozhraní modulů slouží pro komunikaci jádra s modulem. Jádro v tomto případě představuje hostitele pro moduly. Skládá se z rozrahní modulu a rozhraní pro hostitele modulu, pomocí kterého jádro komunikuje s modulem (viz Obrázek 7.4. : Rozhraní modulů). K jednomu hostiteli může být připojen libovolný počet modulů. Moduly mohou, narozdíl od pluginů, svého hostitele zpětně ovládat a stejně jako pluginy předají výstup ze své funkce. Jádro si modul připojí pomocí metody Register, které se připojovaný modul předává jako parametr. Stejně tak může hostitel modul odregistrovat pomocí metody Unregister. Pomocí metody ReceiveFeedback může modul ovládat svého hostitele, jako parametry se předávají modul, který odezvu vyvolal, a datový objekt.

Samotné rozhraní modulu vyžaduje implementaci těchto vlastností: jméno

autora, popis, hostitel modulu, jméno, komponentu uživatelského rozhraní a verzi. Dále rozhraní požaduje implementaci metod Execute, které provádí danou funkci modulu, a implementaci inicializační a destrukční metody.



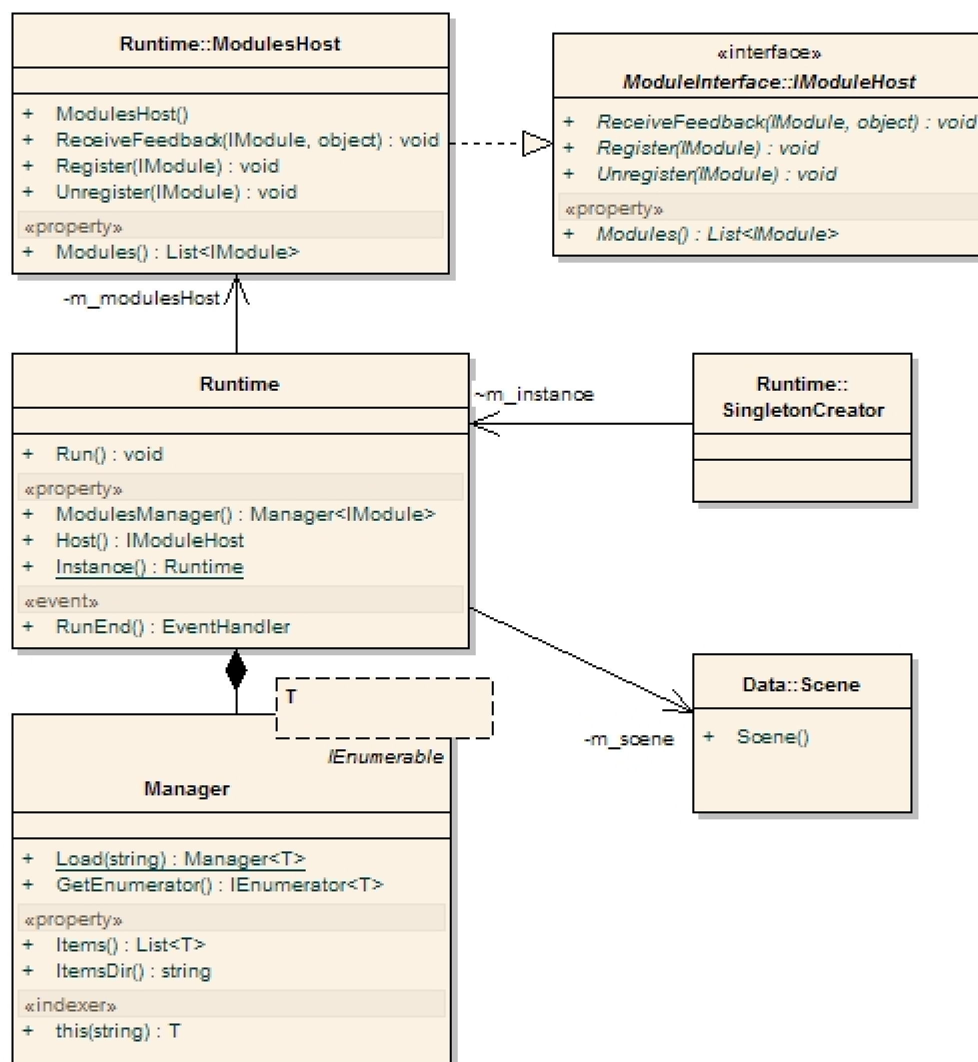
Obrázek 7.4. : Rozhraní modulů

7.4 Jádro

Jádro zastává v sytému funkci řídícího prvku všech modulů, u kterých může spouštět a přerušovat jejich funkce. Jádro je přimo využívano vstupní aplikací (grafické uživatelské rozhraní, příkazový řádek atd.). Jádro pracuje synchronně v diskrétním čase T. Standardní časová jednotka je 1 (rychlost animace se určí pomocí parametru počet snímků za vteřinu). Jádro může být ovládáno buď modulem nebo vstupní aplikací.

Hlavní elementem jádra je třída Runtime, která zajišťuje načtení, registraci

a spouštění modulů. Třída je navržena pomocí návrhového vzoru singleton, protože je nezbytné, aby byla vytvořena vždy pouze jedna instance třídy Runtime. Kdyby tomu tak nebylo, různé instance třídy Runtime by mohly kolidovat při přístupu k modulům. Pro komunikaci třídy Runtime s moduly slouží ModulesHost, třída která implementuje rozhraní IModuleHost. Dále jádro obsahuje datovou část (Scene) a využívá manažer dll souborů pro načítaní modulů (viz Obrázek 7.5. : Model jádra). Scéna je hlavní datový objekt, který se předává mezi moduly, které mohou scénu měnit. Má stromovou strukturu, kde jeden uzel představuje prvek položený v prostoru.



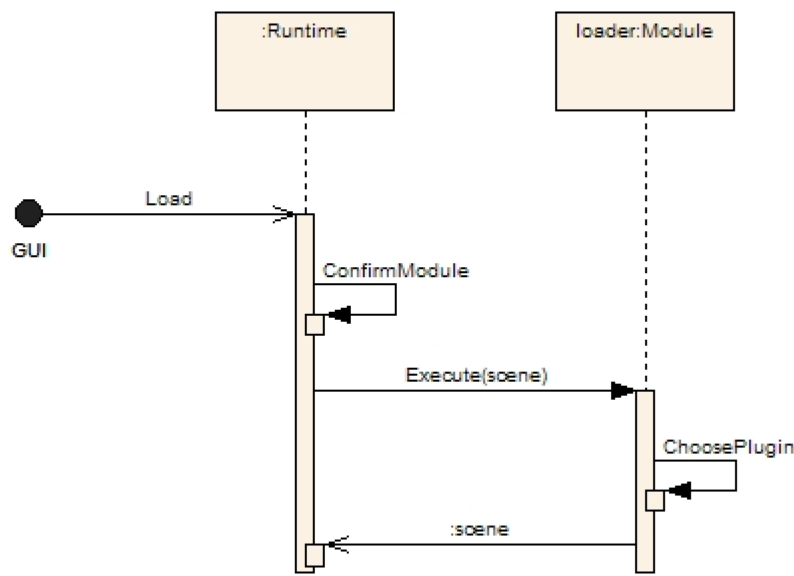
Obrázek 7.5. : Model jádra

7.5 Chování objektů v systému

V této podkapitole popíši chování konkrétních modulů za běhu systému pomocí sekvenčních diagramů. Z nich lze vyčíst, kdy jaká činnost zaměstnává jádro systému a kdy jednotlivé moduly. Pomocí sekvenčních diagramů popíšu následující základní činnosti systému: načítání, renderování a animace s pomocí fyziky.

7.5.1 Načítání

Načítání scény z externích datových zdrojů zařizuje modul Loader. Poté co přijde podnět na načítení scény ze vstupní aplikace, jádro si ověří přítomnost modulu Loader a vynutí si na něm vykonání činnosti metdou Execute, které předá odkaz na prázdnou scénu. V rámci modulu (např. jeho projevu v grafickém uživatelském rozhraní) se zvolí příslušný plugin, který provede parsing vstupních dat a naplní jimi prázdnou scénu. Tu poté vrátí zpět do jádra, které vyvolá událost RunEnd a čeká na další podněty (viz Obrázek 7.6. : Načítání).



Obrázek 7.6. : Načítání

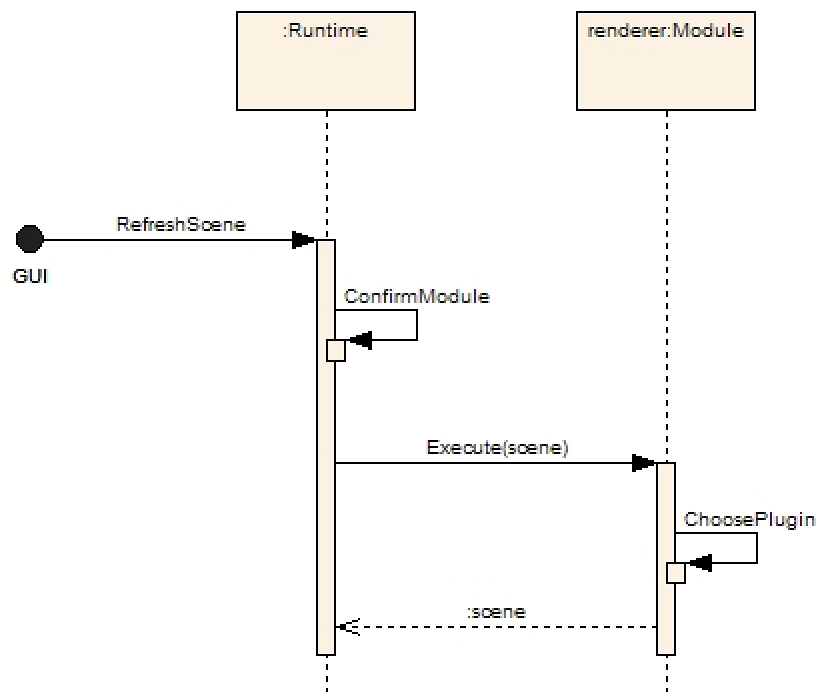
7.5.2 Renderování

"Renderování (anglicky rendering) je proces, při němž ze zadaných dat vzniká

cílový obraz. V případě grafického editoru je výstupem 3D obrázek, v případě webového prohlížeče je to vysázená webová stránka."

Renderování [online]. c1998-2007 [cit. 2007-04-19]. Dostupný z WWW: <http://www.root.cz/slovnicek/renderovani/>. ISSN 1212-8309.

O renderování scény na výstup se stará modul Renderer. Po příchodu podnětu ze vstupní aplikace na vykreslení scény jádro ověří přítomnost modulu Renderer a vynutí si na něm vykonání činnosti metdou Execute, které předá odkaz s aktuální scénou. V rámci modulu se zvolí příslušný plugin, který provede vyrenderování scény. Tu poté vrátí zpět do jádra, které vyvolá událost RunEnd a čeká na další podněty (viz Obrázek 7.7. : Renderování).



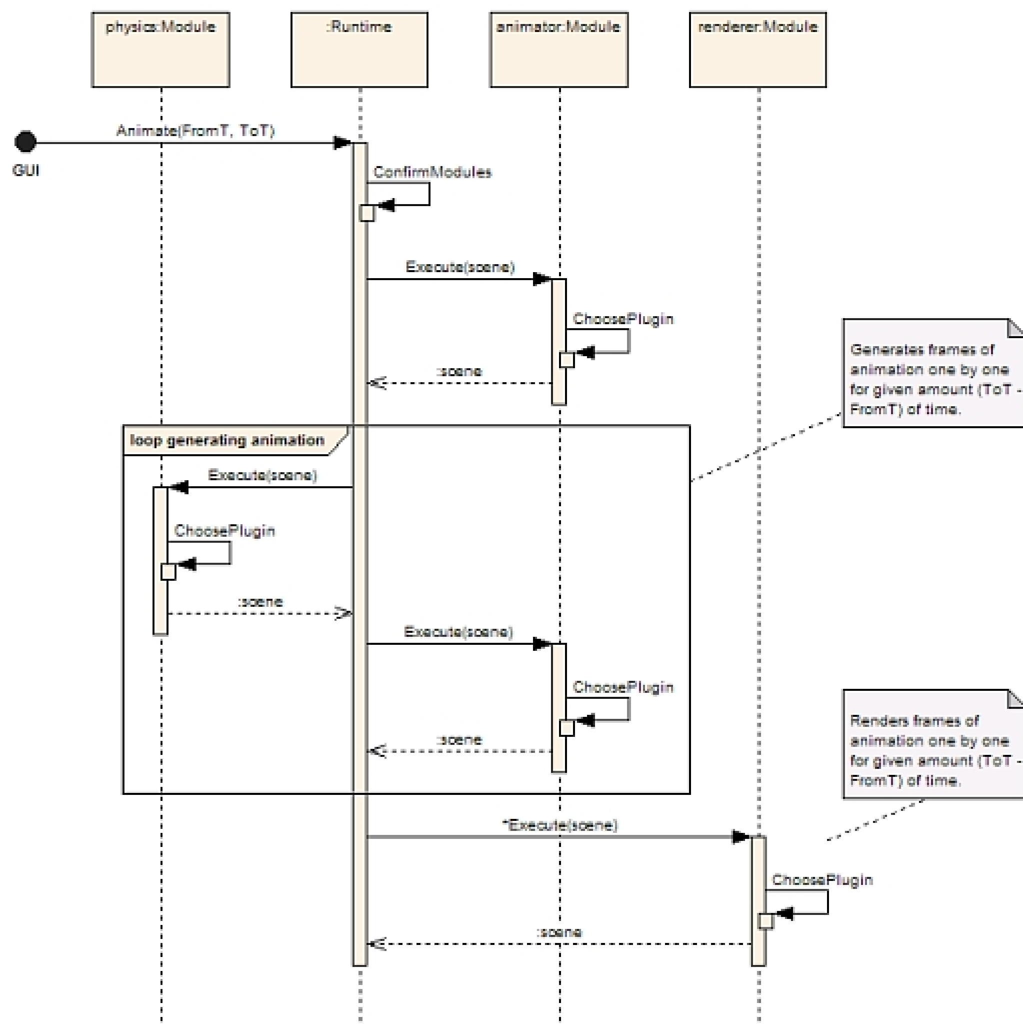
Obrázek 7.7. : Renderování

7.5.3 Animace s pomocí fyziky

Animace je řešena snímek po snímku. Začíná se vytvořením snímku v čase T, poté se změní scéna a vytvoří se snímek v čase T + 1. Takto se generuje animace v určitém časovém rozsahu. Tento přístup by mohl mít problémy se scénami s velkým počtem prvků, díky velkému množství přenášených datových objemů. Proto by do

budoucna bylo vhodné vytvořit animační datový formát, který by s modulem animace celou činnost optimalizoval.

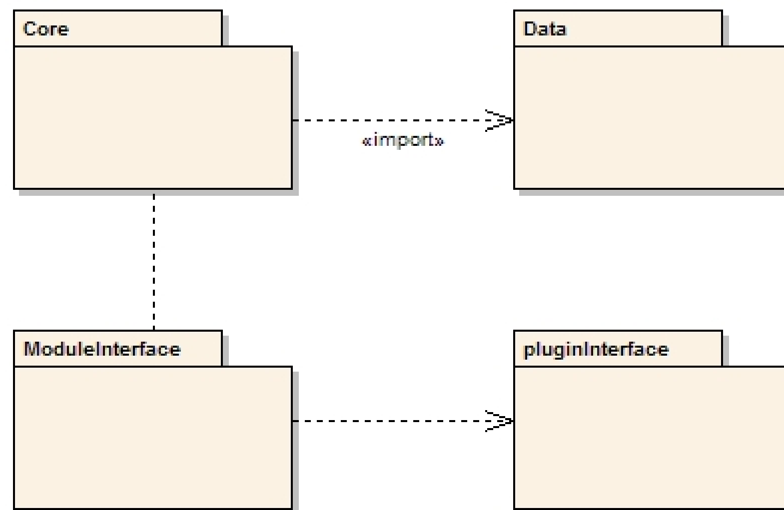
O animování a fyziku ve scéně se stará modul Animator a Physics. Poté, co přijde požadavek na animaci scény, si jádro ověří přítomnost modulů a vytvoří první snímek animace pomocí modulu Animator, kterému se předá odkaz se scénou. Poté se scéna přesune do modulu Physics, kde se podle vybraného fyzikálního pluginu upraví. Následuje opětovné předání do animátoru a celý proces se opakuje do té doby, než je dosaženo požadovaného času T. Poté, co se celá animace vygeneruje, je možné jí snímek po snímku vyrenderovat (viz Obrázek 7.8. : Animace).



Obrázek 7.8. : Animace

7.6 Rozdělení do balíčků

V programovacím jazyku C# je balíček reprezentován jmenným prostorem (namespace). Pro zpřehlednění kódu jsem celý systém rozdělil do čtyř balíčků (Core, Data, ModuleInterface, PluginInterface), přičemž balíček Data je vnořený v balíčku Core (viz Obrázek 7.9. : Balíčky v systému).



Obrázek 7.9. : Balíčky v systému

8 Závěr

V bakalářské práci jsem měl za úkol vymodelovat pomocí jazyka UML softwarový návrh modulární aplikace. Proto jsem v teroretické části popsal nezbytné prostředky tohoto jazyka, které jsem použil při návrhu v praktické části.

Výsledkem mé práce je stručný přehled jazyka UML, který jsem čerpal z uvedených knih a internetových zdrojů. Pokud čtenář při čtení praktické části narazí na nějaký prvek jazyka UML, kterému nerozumí, může si ho zpětně dohledat v teoretickém přehledu. Mým přínosem v bakalářské práci je praktická část, kde je rozebrán samotný návrh. Aplikaci lze díky její flexibilitě dále rozvíjet nebo jednotlivé části předělávat, a to bez nutnosti úpravy ostatních elementů v systému. Návrh si tedy neklade za cíl splnit všechny možné požadavky (kterých může být velké množství), ale vytváří dostatečně obecné prostředí, v kterém nebude problém realizace dalších dílčích problémů. Modularitu jsem zajistil návrhem rozhraní, pomocí nichž jednotlivé prvky v systému komunikují. Dále je k dispozici kompletní návrh vytvořený v nástroji Enterprise Architect a zdrojové kódy kostry programu, a to buď na přiloženém CD nebo z internetu (http://opensvn.csie.org/stereoscopic_projection).

Program se zatím nachází v počáteční fázi, jelikož nejsou dostupné všechny požadované moduly. Prostředí však nabízí prostor pro jejich realizaci, a tudíž je ostatní zájemci mohou začít testovat. Návrh tedy vyšlapává pomyslnou cestičku dalším - mně i následovníkům, kteří by mohli například vylepšit třídu Runtime, která je zatím navržena jen v hrubých obrysech. Osobně mi práce na tomto projektu přinesla velmi cenné zkušenosti z oblasti návrhu objektově orientovaných modulárních systémů a hodlám se této problematice věnovat i v budoucnu (včetně práce na projektu samotném). Jelikož nikde na internetu nenajdete příklad návrhu větších rozměrů, ocenil jsem věcné rady a připomínky vedoucího práce Mgr. Miloše Prokýška.

Seznam použité literatury

(1) ARLOW, Jim, NEUSTADT, Ila. UML a unifikovaný proces vývoje aplikací. 2003. vyd. Brno Computer Press 2003. 408 s. ISBN 80-7226-947-X.

(2) PITMAN, Neil, PILONE, Dan. *UML 2.0 in a Nutshell*. 2005 edition. Sebastopol: O'Reilly Media, 2005. 234 s. ISBN 0-596-00795-7

(3) Object managment group. *(1)Unified Modeling Language: Superstructure*. 2005th edition. [s.l.]: [s.n.], 2005. 694 s. Dostupný z WWW: <http://www.omg.org/technology/documents/modeling_spec_catalog.htm>.

Internetové zdroje

- http://www.omg.org
- http://www.uml.org/
- http://interval.cz
- http://www.objects.cz/
- http://www.root.cz/
- http://en.wikipedia.org/
- http://www.voxcafe.cz/
- http://www.sparxsystems.com.au/
- http://www.codeproject.com/

Terminologický slovník

C#	Moderní objektově orientovaný jazyk z rodiny Microsoft .NET.
CASE	Computer-aided software engineering - Nástroj, který pomáhá při vývoji a údržbě softwaru.
EA	Enterprise Architect - Nástroj CASE.
Microsoft .NET	Platforma pro vývoj softwarových produktů.
Modul	Soubor funkcionalit systému
OCL	Object constrain language - Jazyk na psaní omezujících podmínek.
OMG	Object Managment Group - Konsorcium pro správu standardů pro objektově orientované systémy.
OOSE	Object-oriented software engineering - Metodika modelování objektově orientovaných systémů.
OQL	Object Query Language - Obecný dotazový jazyk.
Plugin	Funkcionalita modulu.
UML	Unified Modeling Language - Jazyk určený k modelování softwarových řešení.
XML	Extensible Markup Language - Snadno rozšiřitelný značkovací jazyk.

Příloha A

Runtime.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;
using ModuleInterface;

namespace Core
{
    /// <summary>
    /// This class represent runtime of module-plugin environment. It runs map of modules.
    /// Class must have private constructor because it's Singleton. You can get instance of
    /// this class throught Instance public static property.
    /// </summary>
    public class Runtime
    {
        #region Private nested classes

        /// <summary>
        /// Internal class used for communication with modules.
        /// </summary>
        class ModulesHost : IModuleHost
        {
            #region Private Variables

            private List<IModule> m_modules;

            #endregion

            public ModulesHost()
            {
                m_modules = new List<IModule>();
            }

            #region IModuleHost Members

            public List<IModule> Modules
            {
                get { return m_modules; }
            }

            public void ReceiveFeedback(IModule from, object data)
            {
                throw new Exception("The method or operation is not implemented.");
            }

            public void Register(IModule module)
            {
                m_modules.Add(module);
```

```
    }

    public void Unregister(IModule module)
    {
        m_modules.Remove(module);
    }

    #endregion
}

/// <summary>
/// Nested private class that instantinate Runtime instance.
/// </summary>
class SingletonCreator
{
    /// <summary>
    /// Private static constructor.
    /// </summary>
    static SingletonCreator()
    {
    }

    /// <summary>
    /// Instance of Runtime class.
    /// </summary>
    internal static readonly Runtime m_instance = new Runtime();
}

#endregion // Private nested classes

#region Private member variables

private Manager<IModule> m_modulesManager;
private IModuleHost m_modulesHost;
private Data.Scene m_scene;

#endregion //Private members

#region Public properties

/// <summary>
/// Returns manager of modules.
/// </summary>
public Manager<IModule> ModulesManager
{
    get { return m_modulesManager; }
}

/// <summary>
/// Returns Module host.
/// </summary>
public IModuleHost Host
{
```

```
        get { return m_modulesHost; }
    }

    internal Data.Scene Scene
    {
        get { return m_scene; }
        set { m_scene = value; }
    }

    #endregion // Public properties

    #region Public static properties

    /// <summary>
    /// Returns instance of Runtime class.
    /// </summary>
    public static Runtime Instance
    {
        get { return SingletonCreator.m_instance; }
    }

    #endregion // Public static properties

    #region Public events

    /// <summary>
    /// Occures when the Run method end.
    /// </summary>
    public event EventHandler RunEnd;

    #endregion // Public events

    /// <summary>
    /// Constructor must be private because this class is Singleton.
    /// </summary>
    Runtime()
    {
        m_scene = new Data.Scene();
        if(!System.IO.Directory.Exists("Modules"))
            System.IO.Directory.CreateDirectory("Modules");

        m_modulesManager = Manager<IModule>.Load(AppDomain.CurrentDomain.BaseDirectory +
"Modules");
        m_modulesHost = new ModulesHost();

        foreach (ModuleInterface.IModule module in m_modulesManager)
        {
            Console.WriteLine("Registring {0} to modules host", module.Name);
            m_modulesHost.Register(module);
        }
    }

    /// <summary>
```

```
        /// Runs the runtime.
        /// </summary>
        public void Run()
        {
            OnRunEnd();
        }

        /// <summary>
        /// Raises the Core.Runtime.RunEnd event.
        /// </summary>
        protected void OnRunEnd()
        {
            if (RunEnd != null)
                this.RunEnd(this, new EventArgs());
        }
    }
}
```

Manager.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using ModuleInterface;
using PluginInterface;

namespace Core
{
    /// <summary>
    /// Generic class that serves as items manager given by type T. It loads all dll files from
    /// directory and make instance of T class form every assembly.
    /// </summary>
    /// <typeparam name="T">Items type</typeparam>
    public class Manager<T> : IEnumerable<T>
    {
        private string m_itemsDir;
        private int m_index;
        private Dictionary<string, T> m_items;

        #region Public properties

        /// <summary>
        /// Creates new list of items and return it.
        /// </summary>
        public List<T> Items
        {
            get
            {
                List<T> list = new List<T>();
                foreach (KeyValuePair<string, T> keyValuePair in m_items)
                {
                    list.Add(keyValuePair.Value);
                }
```

```
        return list;
    }

}

/// <summary>
/// Gets directory of managed items.
/// </summary>
public string ItemsDir
{
    get { return m_itemsDir; }
}

#endregion

#region Public indexers

/// <summary>
/// Gets item by it's name
/// </summary>
/// <param name="name">Name of item</param>
/// <returns>Item</returns>
public T this[string name]
{
    get
    {
        return m_items[name];
    }
}

#endregion

/// <summary>
/// Creates empty manager.
/// </summary>
private Manager()
{
    m_itemsDir = "";
    m_index = -1;
    m_items = new Dictionary<string, T>();
}

/// <summary>
/// Creates manager and search all files from given path with dll extension.Then search for
/// T types from assembly, instantiate it and add it to Directory collection.
/// </summary>
/// <param name="path">Items path</param>
/// <returns>Instance of Manager class</returns>
public static Manager<T> Load(string path)
{
    Manager<T> manager = new Manager<T>();
    manager.m_itemsDir = path;
    Console.WriteLine("Loading modules from {0} directory", path);
```

```
            foreach (string file in Directory.GetFiles(path, "*.dll"))
            {
                System.Reflection.Assembly assembly = System.Reflection.Assembly.LoadFrom(file);

                foreach (Type type in assembly.GetTypes())
                {

                    Type typeInterface = type.GetInterface(typeof(T).ToString(), false);
                    if (type.IsPublic && !type.IsAbstract && typeInterface != null)
                    {
                        Console.Write("Loading " + type.Name);
                        T item = (T)Activator.CreateInstance(type);

                        try
                        {
                            manager.m_items.Add(type.Name.Substring(type.Name.LastIndexOf(".") + 1,
type.Name.Length - type.Name.LastIndexOf(".") - 1), item);
                        }
                        catch (ArgumentException exc1)
                        {
                            Console.WriteLine("-Error:"+exc1.Message);
                            throw new Exception("Loading of module failed.", exc1);
                        }
                        Console.WriteLine("...loading done.");
                    }
                }

            }
            return manager;
        }

        #region IEnumerable<T> Members

        public IEnumerator<T> GetEnumerator()
        {
            return m_items.Values.GetEnumerator();
        }

        #endregion

        #region IEnumerable Members

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
        {
            return m_items.Values.GetEnumerator();
        }

        #endregion
    }
}
```

Scene.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Core.Data
{
  /// <summary>
  /// Internal data structure
  /// </summary>
  class Scene
  {
    public Scene()
    {
    }
  }
}
```

IModule.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ModuleInterface
{
  public interface IModule
  {
    #region Public properties

    /// <summary>
    /// Return creator of plug-in.
    /// </summary>
    string Author { get; }

    /// <summary>
    /// Return descrition of plug-in behavior.
    /// </summary>
    string Descrition { get; }

    /// <summary>
    /// Host which is plug-in connected to.
    /// </summary>
    IModuleHost Host { get; set; }

    /// <summary>
    /// Return name of plug-in.
    /// </summary>
    string Name { get; }

    /// <summary>
    /// Return representation of module to fill graphics user interface.
    /// </summary>
    System.Windows.Forms.Control UIDisplay { get; }
```

```
        /// <summary>
        /// Return version of plug-in
        /// </summary>
        string Version { get; }

        #endregion

        #region Public methods

        /// <summary>
        /// Executes module. Modify data object then returns it.
        /// Module can use additional data through addData object.
        /// </summary>
        /// <param name="dataObject">Data object for execution</param>
        /// <param name="addData">Additional data for execution</param>
        /// <returns>Output dataObject of execution</returns>
        object Execute(object dataObject, object addData);

        /// <summary>
        /// Initialize module. It's called always after module is loaded.
        /// </summary>
        void Initialize();

        /// <summary>
        /// Dispose module.
        /// </summary>
        void Dispose();

        #endregion
    }
}
```

IModuleHost.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ModuleInterface
{
    public interface IModuleHost
    {
        /// <summary>
        /// List of modules registered to module host.
        /// </summary>
        List<IModule> Modules { get; }

        /// <summary>
        /// Object that have registred module can receive feedback from this module throught
        /// this method. Module can have some control over host that is connected to.
        /// </summary>
        /// <param name="from">Referenev to module that sends feedback</param>
```

```
        /// <param name="data">Feedback data</param>
        void ReceiveFeedback(IModule from, object data);

        /// <summary>
        /// Register module to this host
        /// </summary>
        /// <param name="module">Module to be registered</param>
        void Register(IModule module);

        /// <summary>
        /// Unregister module from this host
        /// </summary>
        /// <param name="module">Module to be unregistered</param>
        void Unregister(IModule module);
    }
}
```

IPlugin.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PluginInterface
{
    public interface IPlugin
    {
        #region Public properties

        /// <summary>
        /// Return creator of plug-in.
        /// </summary>
        string Author { get; }

        /// <summary>
        /// Return descrition of plug-in behavior.
        /// </summary>
        string Descrition { get; }

        /// <summary>
        /// Return name of plug-in.
        /// </summary>
        string Name { get; }

        /// <summary>
        /// Return version of plug-in
        /// </summary>
        string Version { get; }

        #endregion

        #region Public methods

        /// <summary>
```

```
        /// Initialize plug-in. It's called always after plug-in is loaded.
        /// </summary>
        void Initialize();

        /// <summary>
        ///
        /// </summary>
        void Dispose();

        /// <summary>
        /// Executes plugin. Modify data object then returns it.
        /// Plugin can use additional data through addData object.
        /// </summary>
        /// <param name="dataObject">Data object for execution</param>
        /// <param name="addData">Additional data for execution</param>
        /// <returns>Output dataObject of execution</returns>
        object Execute(object dataObject, object addData);

        #endregion
    }
}
```

IPluginHost.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PluginInterface
{
    public interface IPluginHost
    {
        /// <summary>
        /// List of plug-ins registered to plug-in host.
        /// </summary>
        List<IPlugin> Plugins { get; }

        /// <summary>
        /// Register plug-in to this host
        /// </summary>
        /// <param name="plugin">Plug-in to be registered</param>
        void Register(IPlugin plugin);

        /// <summary>
        /// Unregister plug-in from this host
        /// </summary>
        /// <param name="plugin">Plug-in to be unregistered</param>
        void Unregister(IPlugin plugin);
    }
}
```

Příloha B

Do práce vložený kompletní diagram tříd ve formátu A3.

Příloha C

Vypracované diagramy v nástroji Enterprise Architect (formát .EAP) jsou k dispozici na přiloženém CD a dostupné z internetu na adrese http://opensvn.csie.org/stereoscopic_projection/StePr/Uml.