Zadání

Vytvořte metamodelář na platformě NetBeans. Součástí metamodeláře budou grafické knihovny pro vykreslování symbolů použitých v meta-modelu. Jedna z knihoven bude využívat NetBeans Visual library druhá framework JGraph.

České vysoké učení technické v Praze  
Fakulta elektrotechnická  
Katedra počítačů  
  
  
  
  
  
  
  
Bakalářská práce  
  
Meta-modelovací nástroj v prostředí NetBeans*Vrchlavský Petr*  
  
  
  
  
  
  
Vedoucí práce: Ing. Ondřej Macek  
  
  
Studijní program: Softwarové technologie a management, strukturovaný, bakalářský  
  
Obor: Softwarové inženýrství  
  
2. dubna 2011

Poděkování

Zde bych rád poděkoval svému vedoucímu práce Ing. Ondřeji Mackovi za jeho cenné rady a ukázání správného směru vždy, když jsem se na něj obrátil.

Prohlášení  
  
Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 2. 4. 20011 ……………………………………………......

podpis

Abstract

The primary goal of this thesis is design and implementation of a meta-modeling tool with support for graphic notations. The application will let to define meta-models along with their visualization rules. In the opening part, the process of meta-modeling is explained and also the description of an existing solution can be found there. The second part concentrates to the architecture of the application and its implementations details. The close part contains brief description of testing and a user manual.

Abstrakt

Hlavním cílem práce je návrh a implementace meta-modelovacího nástroje s podporou pro grafické notace. Ten umožní nadefinování meta-modelu a pravidel pro jeho vizualizaci. V první části práce je vysvětleno, co je to meta-modelování a popis existující alternativy. Druhá část se věnuje architektuře aplikace a dále popisu její implementace. Závěr práce obsahuje stručný popis testování a uživatelskou příručku.

Obsah

1. [Úvod 1](#_Toc293405850)
2. [Meta-modelování 3](#_Toc293405851)

[2.1 Meta-modelování 3](#_Toc293405852)

[2.2 Modelování vs. meta-modelování 4](#_Toc293405853)

[2.2 Meta-Object Facility (MOF) 5](#_Toc293405854)

[3.3 Eclipse Modeling Framework (EMF) 6](#_Toc293405855)

1. [Cíle a analýza požadavků 9](#_Toc293405856)

[3.1 Zadání a cíle 9](#_Toc293405857)

[3.2 Požadavky 10](#_Toc293405858)

[3.2.1 Požadavky nefunkční 10](#_Toc293405859)

[3.2.2 Požadavky funkční 11](#_Toc293405860)

1. [Architektura návrhu 13](#_Toc293405861)

[4.1 Problematika návrhu meta-modelovacích nástrojů 13](#_Toc293405862)

[4.2 Návrh řešení 13](#_Toc293405863)

[4.2 Jádro aplikace 15](#_Toc293405864)

[4.3 Vizualizace symbolů meta-modelu 19](#_Toc293405865)

[4.4 Grafické uživatelské rozhraní 22](#_Toc293405866)

[4.5 Serializace 24](#_Toc293405867)

1. [Implementace 27](#_Toc293405868)

[5.1 NetBenas platforma 27](#_Toc293405869)

[5.2 NetBeans IDE 28](#_Toc293405870)

[5.3 Použité API 28](#_Toc293405871)

[5.3.1 Nodes API a Explorers & Property Sheet API 29](#_Toc293405872)

[5.3.2 Project API 29](#_Toc293405873)

[5.3.3 Správa souborů a jejich typů 30](#_Toc293405874)

[5.3.4 Podpora souborového typu 30](#_Toc293405875)

1. [Testování 33](#_Toc293405876)

[6.1 Domény testování 33](#_Toc293405877)

[6.1.1 Testování systémových požadavků 33](#_Toc293405878)

[6.1.2 Testování návrhu a implementace 34](#_Toc293405879)

[6.1.3 Testování v provozu 35](#_Toc293405880)

1. [Závěr 37](#_Toc293405881)

[7.1 Shrnutí 37](#_Toc293405882)

[7.2 Budoucí práce 37](#_Toc293405883)

[7.3 Závěr 38](#_Toc293405884)

[Literatura 39](#_Toc293405885)

[A. Instalační a uživatelská příručka 41](#_Toc293405886)

[A.1 Instalace Metamodeláře 41](#_Toc293405887)

[A.2 Odinstalování Metamodeláře 43](#_Toc293405888)

[A.3 Uživatelská příručka 43](#_Toc293405889)

[A.3.1 Založení projektu 43](#_Toc293405890)

[A.3.2 Práce se souborem meta-modelu 44](#_Toc293405891)

[A.3.3 Tvorba meta-modelu 44](#_Toc293405892)

[B. Obsah přiloženého CD 49](#_Toc293405893)

Seznam obrázků

[Obrázek 1 - Grafické znázornění MOF architektury [12] 5](#_Toc293349247)

[Obrázek 2 - EMF sjednocuje Javu, XML a UML 6](#_Toc293349248)

[Obrázek 3 - Jádro EMF [11] 8](#_Toc293349249)

[Obrázek 4 - UML diagram znázorňující vymezení hranic Metamodeláře 10](#_Toc293349250)

[Obrázek 5 - Zjednodušené zobrazení vrstev abstrakce v podání Metamodeláře 14](#_Toc293349251)

[Obrázek 6 - Návrh jádra Metamodeláře 16](#_Toc293349252)

[Obrázek 7 - Vazba mezi dvěma referencemi 17](#_Toc293349253)

[Obrázek 8 - Model jako rozhrani mezi daty a klienty 19](#_Toc293349254)

[Obrázek 9 - Vizualizační vrstva 20](#_Toc293349255)

[Obrázek 10 - Sekvenční diagram vykreslování meta-objektů, v tomto případě se jedná o Element 21](#_Toc293349256)

[Obrázek 11 - Ilustrace spojení vrstvy meta-modelu a vizualizační vrstvy 21](#_Toc293349257)

[Obrázek 12 - Schéma stromové struktury znázorňující meta-model 23](#_Toc293349258)

[Obrázek 13 - Schéma souboru s příponou *.mm* nebo *.MM* 24](#_Toc293349259)

[Obrázek 14 - Vztah mezi NetBeans platformou, NetBeans IDE a jednotlivými moduly, které dělají z NetBeans IDE použitelný nástroj. (Zdroj: [6]) 28](#_Toc293349260)

[Obrázek 15 - Definice MIME typu pomocí konfiguračního souboru modulu 30](#_Toc293349261)

[Obrázek 16 - Vztah mezi *File*, *FileObject* a *DataObject* [13] 31](#_Toc293349262)

[Obrázek 17 - Dialog NetBeans IDE pro instalaci pluginů 41](#_Toc293349263)

[Obrázek 18 - Označte moduly, které chcete do NetBeans IDE nahrát a potvrďte *Open* tlačítkem 42](#_Toc293349264)

[Obrázek 19 - Zvolené moduly nainstalujete stisknutím tlačítka *Install* 42](#_Toc293349265)

[Obrázek 20 - Odinstalace modulů/plaginů 43](#_Toc293349266)

[Obrázek 21 - Nově založený projekt Metamodeláře 44](#_Toc293349267)

[Obrázek 22 - Nově vytvořený meta-model rodiny 45](#_Toc293349268)

Seznam zkratek

API Application programming interface [aplikační programovací rozhraní]

EMF Eclipse modeling framework

GIT

GUI Graphical user interface [grafické uživatelské rozhraní]

IDE Integrated development environment [integrované vývojové prostředí]

IT Information technology [informační technologie]

MDE Model driven engeneering [vývoj řízený modely]

MOF Meta-object Facility

MVC Model-View-Controller

OOP Object oriented programming [objektově orientované oprogramování]

UI User interface [uživatelské rozhraní]

UML Unified modeling language [unifikovaný modelovací jazyk]

UP Unified process [unifikovaný proces]

XMI XML metadata interchange

XML Extensible Markup Language [rozšiřitelný značkovací jazyk]

ZIP

Kapitola 1

Úvod

Pomocí techniky meta-modelování vytváříme šablony pro budoucí modely, které nám velmi usnadní práci při jejich realizaci. Každý standard, každá specifikace ve světě výpočetní techniky je meta-model. Implementace těchto standardů a specifikací jsou konkrétní modely vycházející ze své abstrakce – čili meta-modelu.

Principu meta-modelování využíváme dennodenně, aniž bychom si to uvědomili. Například při tvorbě tohoto textu jsem nejprve vytvořil šablonu definující styly jednotlivých elementů, respektive nadpisů, seznamů, obsahu, rozvržení stránek, záhlaví, zápatí aj., jinými slovy jsem vytvořil meta-model obsahu bakalářské práce a postupně, jak jsem přidával obsah, jsem vytvářel konkrétní dokument.

Meta-modelování je pojem relativně nový a je spjat s informačními technologiemi (dále IT), nicméně už z principu meta-modelování je jasné, že se s ním lidstvo potýkalo od nepaměti. Rozdíl oproti dnešku je ve formě vyjádření meta-modelu, respektive jeho chápání jako abstrakce nad konkrétnem. Pokud se zaměříme na oblast IT, existuje několik specifikací a nástrojů, poskytující meta-modelovací schopnosti. Těmito specifikacemi a nástroji se zabývá kapitola 2.

Tato práce posouvá meta-modelování, oproti dnešním standardům, o trochu více k abstraktní rovině. Neopírá se o principy objektově orientovaného návrhu softwaru, jak je tomu u dnešních meta-modelovacích nástrojů, ale naopak se snaží poskytnout meta-modelovací prostředí, schopné poskytnou jednodušší náhled na modelovanou problematiku a nezávislou na principech ostatních meta-modelovacích nástrojích. Cílem je tedy vytvořit aplikaci, která nebude spjata s meta-modelováním objektově orientovaných systémů, ale bude zaměřena na více obecnou rovinu problému. Meta-modelování není triviální záležitost. To se také odrazilo v této práci. V důsledku rozsáhlosti konceptu meta-modelování nebylo v mých silách vytvořit nástroj držících se ve všech ohledech současných specifik meta-modelování. Jako příklad uvedu modelovací (meta-modelovací) jazyky, které jsou definované na všech úrovních abstrakce. Tyto jazyky slouží k definici jednotlivých vrstev abstrakce. Metamodelář definuje tyto jazyky svojí architekturou. Uživatel nemá přímý vliv na jejich definici.

Kapitola 2

Meta-modelování

2.1 Meta-modelování

V této kapitole bych se rád zaměřil na pojem meta-modelování a jeho popis z pohledu modelování.

Meta-modelování lze chápat jako nadstavbu nad samotným modelováním s výraznými prvky abstrakce. Stejně jako modelování chápeme jako abstrakci a zjednodušení skutečnosti, tak i meta-modelování zaujímá určitou pozici na pomyslné úrovni abstrakce. Co je to ve skutečnosti abstrakce a jaký je rozdíl mezi abstrakcí na úrovni meta-modelu a modelu samotného? Toto jsou zásadní otázky, které dále rozepíši a pokusím se o co nejvíce výrazné rozdělení mezi těmito dvěma úrovněmi abstrakce.

Význam pojmu abstrakce nelze vyjádřit jednoznačně a v jednotlivých oborech lidské činnosti se může v konkrétních detailech významně lišit (abstrakce v umění vs. abstrakce v OOP). Proto zde popíši pojem abstrakce v kontextu modelování.

Začněme definicí modelu. Model je všeobecně považován za reprezentaci určitého objektu nebo systému z daného úhlu pohledu. Modely v informačních technologiích vytváříme za účelem popisu problému a jeho řešení, k výměně informací, ale hlavně ke komunikaci mezi jednotlivými subjekty dané infrastruktury. Tím mám na mysli jednotlivé účastníky podílející se svou činností na úpravě, údržbě či pozorování modelu.

Model v sobě nese informaci o určitém objektu nebo systému. Při nespočetném množství činností, které lidé, stroje či programy vykazují, vzniká ohromné množství modelů. Můžeme říct, že je logické předpokládat, že se v určité podmnožině všech těchto modelů najdou takové, které mají určité rysy stejné a stále se opakující. Zde přichází abstrakce a její přednosti. U zmíněné množiny modelu můžeme vytvořit všeobecná pravidla, která budou obsahovat společné rysy/vlastnosti těchto modelů a prohlásit, že se jedná o jakýsi základ pro stávající a budoucí modely stejného charakteru. Tento základ nazveme abstrakce jednotlivých modelů.

Co je to tedy ta abstrakce? Abstrakce představuje všeobecnou informaci, která je nezávislá na jednotlivých objektech, které z dané abstrakce vychází. Jinými slovy, abstrakce je sjednocení společných vlastností prvků v dané množině s tím, že dané prvky vycházejí ze vzniklé abstrakce. Vlastnosti abstrakce jsou zřejmé. Ačkoliv na úrovni jednotlivých prvků, můžeme pozorovat rozdílné vlastnosti či chování, na úrovní dané abstrakce se chovají všechny prvky zcela stejně.

Jako příklad ze života můžeme uvést rodné číslo. Rodné číslo v sobě obsahuje několik důležitých informací a má daný formát. Zejména datum narození a identifikační prvek dané osoby, 6 čísel pro datum a 4 čísla pro jednoznačnou identifikaci osoby pro daný datum. Tyto vlastnosti můžeme nazvat jako abstraktní vlastnosti. Ty jsou společné pro všechna rodná čísla. Rodná čísla můžeme rozdělit do dvou skupin, a to rodná čísla ženská a mužská. Obě skupiny mají výše zmíněné vlastnosti, a přesto se liší. Liší se zejména v zápisu. V České republice se rozlišuje výpočet rodného čísla na výpočet pro mužskou část populace a pro ženskou část. Při výpočtu rodného čísla pro ženskou populaci se pro datum měsíce přičítá 50. Pro rodná čísla určená mužům se nepřičítá, za normálních okolností, žádná hodnota. Díky tomu lze, mimo jiné, rozeznat rodné číslo ženy od muže. Z pohledu jednotlivých rodných čísel vidíme na první pohled rozdílnosti. U ženských rodných čísel je minimální možná hodnota představující měsíc narození 51. Avšak z abstraktního hlediska jsou si všechna rodná čísla rovna – všechna rodná čísla obsahují datum narození, jednoznačný identifikátor spjatý s daným datem narození a zároveň se každé rodné číslo skládá z 10 číslic. Toto lze nazvat abstrakcí rodných čísel, přičemž rodná čísla pro ženy a pro muže jsou „implementacemi“ této abstrakce.

Nyní se můžeme vrátit k začátku kapitoly a popsat vztah mezi modelem a meta-modelem. Zjednodušeně řečeno meta-model je abstrakce modelu. Meta-model obsahuje informace o daném modelu, o jeho syntaxi (obsah, vazby mezi jednotlivými prvky obsahu, jejich vlastnosti atd.) a sémantice (jakým způsobem se s daným obsahem nakládá, jak lze vyhodnotit, zpracovat atd.).

Z výše uvedeného lze pojem meta-modelování vysvětlit, jako vytváření šablony pro budoucí modely. Konkrétněji, definice obsahu budoucích modelů, interakce mezi jednotlivými prvky obsahu a omezení/hranice budoucích modelů. Dále meta-modelováním určujeme význam jednotlivých prvku obsahu (sémantika) a vlastnosti vztahů mezi jednotlivými prvky obsahu (syntaxe). Konkrétní implementace meta-modelu můžou, a pravděpodobně budou mít, své specifické rysy, ale i přesto budou mít rysy vycházející z meta-modelu (abstrakce modelu) všechny modely stejné.

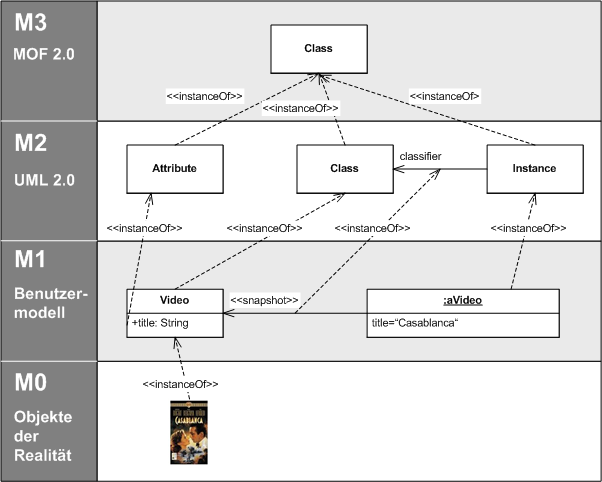
2.2 Modelování vs. meta-modelování

Modelování meta-modelů a modelování modelů je odlišná činnost a je nutné rozlišovat rozdíly. I přesto, že přístupy k meta-modelování nebo modelování jsou velmi podobné, vyžadují jiný základ.

Základem pro vytváření meta-modelů je taktéž meta-model. V tomto případě mluvíme o meta-meta-modelu. Ten definuje schopnosti meta-modelů, respektive jaké vlastnosti lze u meta-objektů definovat, jakých hodnot tyto vlastnosti mohou nabývat. Úroveň abstrakce, které můžeme dosáhnout je v podstatě nekonečná. Každý meta-model je zároveň modelem a ke každému modelu můžeme vytvořit jeho meta-model. Zpravidla se definují tzv. vrstvy abstrakcí, kterých je konečný počet a v praxi nepřesáhne hodnoty 3 - 4. Vrstvy jsou na sobě závislé a téměř výhradně vrstva vyšší abstrakce definuje prostředí pro vrstvu nižší abstrakce.

2.2 Meta-Object Facility (MOF)

MOF [[1](#lit1)] je standardem Object Management Group (OMG[[1]](#footnote-1)) pro vývojovou metologii Model-Driven engineering (MDE). MOF má původ v Unified Modeling Language (UML) [[3](#lit3)], což je souhrn grafických notací pro účel modelování doménové problematiky. MOF byl potřeba pro meta-modelování architektur pro definování UML. MOF je definován jako čtyřvrstvá architektura, kde na nejvyšší vrstvě, označované jako M3 vrstva, se nachází meta-meta-model. Tato vrstva představuje jazyk pro vytváření meta-modelů zvaných M2 modely. M2 model představuje meta-model UML, což je model, který popisuje UML jako takové. M2 vrstva popisuje elementy vrstvy M1 a tím i zároveň modely vrstvy M1. Těmi by mohly být například modely napsané v UML. Poslední vrstva je vrstva M0 a představuje objekty reálného světa (viz [obrázek 1](#obr1)).



**Obrázek 1 - Grafické znázornění MOF architektury [**[**12**](#lit12)**]**

3.3 Eclipse Modeling Framework (EMF)

Už jsme si dříve vysvětlili co je to model, meta-model a vazbu mezi nimi. Pro snazší pochopení co je to EMF a k čemu slouží, si pomůžu názornou ukázkou. Řekněme, že jste dostali za úkol napsat program pro správu objednávek v nějakém obchodě. Víte, že uskutečnění objednávky zahrnuje operace „fakturovat“ a „doručit“ na adresu. Předmět objednávky obsahuje jméno produktu, kvantitu a cenu. Vypadá to velice jednoduše, a proto začnete s tvorbou rozhraní. Po vytvoření požadovaných rozhraní, máme vše, co potřebujeme pro započetí práce na aplikační logice, uživatelském rozhraní (UI – user interface), persistentní vrstvě atd. Ale předtím, než začnete psát kód, budete požádáni nadřízeným, abyste vytvořili i modely, nejlépe v UML. Protože chcete vyhovět, uděláte UML diagramy popisující strukturu aplikace. Nebude dlouho trvat, a dostanete se do situace, kdy budete potřebovat model uschovat na bezpečné místo. Pravděpodobně nalezení bezpečného místa nebude problém, co vás bude spíše zajímat, jakým způsobem bude model uložen, respektive v jakém formátu. Rozhodnete se, že uložit model v XML je rozumná volba. Začnete psát XML Schéma pro popis vašeho XML dokumentu.

Nyní je ten správný čas zaregistrovat, že vytváříte už 3. model jedné a té samé věci. Zde přichází EMF jakožto framework a generátor kódu, který vám umožní definovat model v jakékoliv z výše uvedených alternativ – Java rozhraní, UML nebo XML Schéma. Navíc, a to je velmi důležitá vlastnost frameworku, EMF podporuje konverzi mezi jednotlivými alternativami (viz [obrázek 2](#obr2)).



Obrázek 2 - EMF sjednocuje Javu, XML a UML

Aby bylo možné jednoduše, účinně a hlavně bezpečně provést konverzi z jedné technologie na druhou, je zapotřebí zvolit pevný základ, reprezentaci modelu, která bude schopna popsat daný model a zároveň bude možné provést konverzi bez ztráty informace. Proto EMF definuje meta-model modelu a nazývá jej EMF ECore ([obrázek 3](#obr3)). ECore je uložen jako XML dokument, přesněji pomocí XML Metadata Interchange (XMI).

XMI [[7](#lit7)] je standart pocházející od konsorcia OMG a slouží pro výměnu meta-dat pomocí značkovacího jazyka XML (Extensible Markup Language). Abych přesněji popsal vztah XML a XMI, pokusím se to vyjádřit objekty z reálného světa. XML si můžeme představit jako cihlu a XMI jako dům postavený z cihel. Tam kde potřebujeme použít XML nelze použít XMI. Každý dokument XMI je zároveň XML dokumentem, ale ne každý XML dokument je XMI dokumentem. Jinými slovy, XMI používá XML jako jazyk k popisu meta-dat. XMI používá i další standardy, jmenovitě: UML, MOF a mapování MOF na XMI.

XMI nachází uplatnění především jako jeden z nástrojů MDE[[2]](#footnote-2), kde se předpokládá generování kódu z jednotlivých modelů. Bohužel v současné době existuje několik rozdílných implementací XMI v jednotlivých modelovacích nástrojích. Naneštěstí to znamená téměř nemožnou výměnu souborů mezi jednotlivými UML modelovacími nástroji.

EMF podporuje MDE koncept, tj. z modelů (vstup) generuje zdrojový kód (výstup). Protože jsou reprezentace modelů jednotlivých technologií uložené v abstraktním formátu ECore, je generace zdrojového kódu nezávislá na své reprezentaci a v podstatě se jedná o mechanickou záležitost.

Vraťme se zpět k ECore. Na [obrázku 3](#obr3) je zobrazena architektura jádra EMF ECore. Pro lepší pochopení popíši nejdůležitější části:

* *EClass* - *EClassifier*, od které *EClass* dědí, si můžeme představit jako signaturu dané třídy – obsahuje jméno třídy, reflexi (instanceClass: *EJavaClass<?>* parameter) a počáteční hodnotu, pak si *EClass* můžeme představit jako vnitřní strukturu dané třídy – metody, parametry, reference na jiné třídy atd.
* *EReference* – Představuje jeden konec asociace mezi dvěma třídami. *eReferenceType* odkazuje na třídu na druhém konci asociace a *eOpposite* odkazuje na referenci taktéž na opačné straně asociace.
* *EAttribute* – Reprezentuje jednu položku mezi třídními atributy. *EAttribute* může být buďto primitivní datový typ, nebo datovým typem odvozeným od třídy. *EAttribute* má dolní a horní mez a jméno.
* *EObject* – Plní funkci podobné třídě *Object* v Jave. Navíc disponuje užitečnou funkcionalitou v podobě notifikačního mechanismu (v podstatě se jedná o návrhový vzor *Observer*). Každá třída v *ECore* balíku dědí od *EObject*.
* *EPackage* – Je to kontejner pro *EClassifier*, tj. třídy a datové typy. Balík je identifikován na základě URI (Uniformresourceidentifier).
* *EDataType* – Modeluje jednoduchý datový typ, který není na úrovni meta-modelu blíže specifikován. Slouží jako wrapper (objekt pro zapouzdření informací) pro primitivní datové typy nebo objektové typy plně specifikované Javou (např. *String*, *Date*, *Color* …).



Obrázek 3 - Jádro EMF [[11](#lit11)]

Kapitola 3

Cíle a analýza požadavků

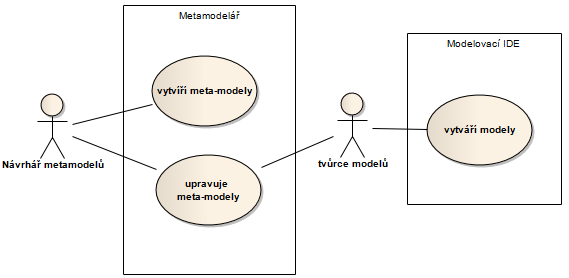
3.1 Zadání a cíle

Ze zadání plyne, že výstupem práce má být nástroj schopný modelovat meta-modely s tím, že zde bude podpora pro vizualizaci jednotlivých elementů meta-modelu. To znamená, vytvoření aplikace, pomocí které bude uživatel definovat meta-modely, které budou dále nějakým způsobem využitelné a zobrazitelné. Ať už ve stejné aplikaci, nebo výstupní meta-modely poslouží jako vstup jiné aplikace, která na tomto bude stavět. Aplikace bude dále umět vykreslovat prvky definované v meta-modelu, protože se předpokládá, že nástroje vytvářející modely na základě meta-modelů se nebudou zabývat vytvářením vizualizačních objektů, ale pouze manipulací s nimi. Důvodem je co nejvíce usnadnit práci modelovacím nástrojům.

V čase vytváření zadání pro tuto práci se počítalo s ještě dalším souběžným projektem, který měl využívat výstupní meta-modely a služby související s vykreslováním jednotlivých prvků meta-modelu. Z důvodů nejasné budoucnosti souběžného projektu a po domluvě s vedoucím práce se od spolupráce výstupních aplikací jednotlivých projektů sešlo. Jelikož není v současné době nástroj, který by uměl podle generovaných meta-modelů vytvářet modely, tudíž není ani zapotřebí, aby se vizualizace prováděla pomocí dvou různých knihoven. Po dohodě s vedoucím práce se s podpory pro grafickou knihovnu *JGraph* [[8](#lit8)] sešlo a zůstalo se pouze u podpory nativní knihovny *VisualLibrary* [[9](#lit9)], která je součástí NetBeans [[4](#lit4)][[5](#lit5)][[6](#lit6)] platformy. Nicméně návrh aplikace umožňuje do budoucna doimplementovat podporu jiné knihovny pro vykreslování, respektive aplikace není závislá na konkrétní knihovně. Pouze stačí dodržet definovaná rozhraní.

Cílem je tedy vytvořit aplikaci, která bude schopna vytvářet meta-modely, definovat grafickou stránku meta-modelu s tím, že ji bude umět i reprezentovat pomocí grafické knihovny *VisualLibrary*. Takto navržené meta-modely budou uloženy do souboru, aby bylo možné s nimi dále pracovat. Aplikaci jsem nazval *Metamodelář*. Na [obrázku 4](#obr4) je znázorněno vymezení hranic Metamodeláře. Uživatel v roli návrháře meta-modelů využívá aplikaci k definici meta-modelů, jejich struktury, možného obsahu a vizuální stránky. Návrhář se také stará o údržbu meta-modelů, jejich korektnost vůči požadovanému výstupu. Zatímco tvůrce modelů využívá meta-modely jako šablonu pro tvorbu modelů a v krajních situacích může meta-modely poupravit, aby vyhovovaly jeho potřebám. Tvorba modelů neprobíhá v aplikaci Metamodeláře. V současné době neexistuje aplikace pro práci výstupních meta-modelů.

Do budoucna se aplikace může rozšířit o podporu dalších grafických knihoven (zejména *JGraph*) a pro podporu modelování na základě meta-modelu. Tyto plány do budoucna nejsou součástí této práce, ale aplikace je koncipována tak, aby bylo možné tyto funkce přidat.



Obrázek - UML diagram znázorňující vymezení hranic Metamodeláře

3.2 Požadavky

Základem každého vývoje softwarového díla je pevná základna specifikace systémových požadavků. Názory na to, jestli se mají požadavky vytvářet vždy, nebo jestli existuje hranice, od kdy požadavky ještě nemá smysl vytvářet a od kdy už je to nutnost, se velice liší. Osobně zastávám názor „Požadavky se mají sbírat, analyzovat a vyhodnotit vždy, u jakéhokoliv projektu“.

U větších projektů jsou specifikace systémových požadavků (SRS – system requirement specification) velmi důležitou částí vývoje softwaru. Díky nim máme přehled, co nás do budoucna čeká, můžeme odhadovat rizika, plánovat, vyhodnocovat aktuální pokrok a hlavně nám usnadňují komunikaci se zákazníkem. V následující části popíši základní požadavky na Metamodelář.

3.2.1 Požadavky nefunkční

Nefunkční požadavky definují vlastnosti systému jako celku a jeho omezení.

1. Metamodelář bude koncipován jako plugin do NetBeans IDE.

Protože NetBeans IDE je prostředí, které přímo vybízí k rozšiřování pomocí pluginů, a také z důvodů nevytvářet pro každý nástroj nové vývojové prostředí, aby se zamezilo záplavě různých nástrojů se svými specifickými vývojovými prostředími, rozhodnul jsem se vytvořit Metamodelář jako plugin pro NetBeans IDE. Z toho vyplývá i nutnost použít Javu, jako programovací jazyk.

1. Metamodelář nebude realizován jako samostatná aplikace.

3.2.2 Požadavky funkční

Především vůči funkčním požadavkům se provádí tzv. validace, která má za cíl ověřit, jestli jsme udělali produkt, který odpovídá požadavkům zákazníka. Proto je nutné co nejvýstižněji a co nejpřesněji popsat každý požadavek, aby se v budoucnu předešlo k chybám vzniklých nepochopením, nejednoznačností, dvojsmysly nebo neúplností jednotlivých požadavků.

Toto ne zcela platí u této práce, protože smyslem práce je vyřešit zadaný problém a ne vyřešit zadaný problém podle přesně daných specifik. Díky tomu nemusí být SRS příliš rozepsané a přesně definované, ony ani být nemohou, protože to je součást práce, navrhnout a vyřešit daný problém, tudíž není možné mít od začátku kompletní a přesné požadavky. Velké množství specifik a jejich řešení se odhalí až v průběhu realizace.

1. Metamodelář bude vytvářet meta-modely

Obecný požadavek vycházející ze zadání.

* 1. Metamodelář umožní vytváření meta-modelů pomocí intuitivního grafického uživatelského rozhraní.

Na Metamodelář se lze dívat jako na aplikaci založenou na architektonickém návrhovém vzoru MVC (*Model-View-Controller*) [[10](#lit10)].

* 1. Bude možné definovat vlastní elementy a jejich spojitost s ostatními elementy.

Toto je kritický požadavek a v podstatě pokrývá hlavní funkcionalitu celého systému. Je zde zapotřebí si uvědomit, co vše je třeba nadefinovat, aby se docílilo daného požadavku.

* + 1. Bude možné nadefinovat obecný meta-objekt – element, který bude sloužit jako stavební kámen meta-modelu a bude obsahovat základní identifikační informace v podobě jmenného prostoru a popisu.
    2. Každý meta-objekt se může odkazovat na jiné meta-objekty.
    3. Každý meta-objekt bude mít sadu vlastností
  1. Vztah mezi dvěma elementy se bude definovat pomocí referencí a relací.  
     1. Reference bude přípojný bod elementu k dané relaci a bude nést informaci o násobnosti a orientaci. Orientace budou 2 druhy – zdrojová a cílová.
     2. Relace poskytne základní informace o vztahu mezi dvěma objekty, který je definován pomocí referencí. Relace umožní nadefinovat symetrii. Dále bude možné nastavit, zda se jedná o kompozitní relaci.

Požadavky celé pod-větve P 3.3 jsou velmi důležité vlastnosti systému. Díky nim můžeme definovat vztahy mezi elementy, definovat vlastnosti těchto vztahů a díky tomu vybudovat komplexnější meta-modely a poté i modely.

* 1. Bude možné definovat vizuální stránku meta-modelu. Především elementů a relací mezi nimi, aby bylo možné vytvářet grafy.
     1. Bude možné přiřadit obrázek jednotlivým elementům.
     2. Bude možné přiřadit popisky k jednotlivým elementům.
     3. Barva pozadí bude nastavitelná u jednotlivých elementů.
     4. Vizualizace přípojných bodů, tedy referencí, bude možné zobrazit jako šipky konkrétního tvaru. Základními tvary budou, plná šipka, dutá šipka, jednoduchá šipka a bez šipky.

Vizualizace modelu je velmi důležitá funkce. Model převedený do grafické podoby je mnohem přirozenější formou komunikace mezi lidmi než shluk dat v textovém souboru. Samozřejmě záleží na kontextu, který daný model zobrazuje, ale dobře navrženou grafickou stránkou modelu můžeme řádově zvýšit jeho čitelnost.

1. Metamodelář bude generovat XML soubor s popisem meta-modelu.

V podstatě se bude jednat o perzistenci meta-modelu, která bude použitelná jak pro uložení meta-modelu mezi jeho jednotlivými úpravami tak i o formu komunikace meta-modeláře a dalších nástrojů, které budou mít zájem meta-data dále zpracovat.

1. Tvorba meta-modelů se bude provádět v rámci projektu.

Jelikož je správa projektů v prostředí NetBeans IDE jednoduchou záležitostí a zároveň zabalení práce do projektu zvyšuje přehlednost a manipulovatelnost s modely, je tento požadavek logickým krokem směrem k lepšímu a intuitivnějšímu ovládání.

Kapitola 4

Architektura návrhu

4.1 Problematika návrhu meta-modelovacích nástrojů

V kapitole 2 věnující se mimo jiné meta-modelování, jsou zmíněné vrstvy abstrakce. Abstrahovat můžeme teoreticky donekonečna. Volba úrovně minimální a maximální abstrakce je kritický bod vývoje každého nástroje zaměřeného na meta-modelování. Minimum je tedy určit úrovně abstrakce, s kterými bude daný nástroj pracovat a jasně definovat účel, vlastnosti a provázanost jednotlivých vrstev.

Každá vrstva definuje jazyk modelování, který je definovaný syntaxí, sémantikou a v případě grafického modelovacího jazyka i notací. Tento jazyk slouží k tvorbě modelů nižší úrovně. Vytváření jednotlivých úrovní není omezeno. Důležitou věcí pro „zakončení“ meta-modelovací hierarchie je nalezení požadované úrovně abstrakce. Nalezení úrovně definující elementy jako „věc“, „vlastnost“ nebo „vztah“ je často ukazatel nalezení požadované úrovně a hranice abstrakce. Záleží na konkrétním nástroji a specifikaci, ze které nástroj vychází, při určení konečné úrovně abstrakce. V tomto ohledu jsou veliké rozdíly mezi jednotlivými meta-modelovacími nástroji. Zvolená úroveň abstrakce by měla být schopna popsat vlastnosti a hranice budoucích požadovaných modelů. Což je velice složitá procedura, ne li nemožná, protože většinou nejsme schopni zmapovat potřebu všech uživatelů na požadovanou úroveň abstrakce.

Najít správnou úroveň abstrakce je nejtěžší úkol z celého návrhu. Aby bylo možné tuto úroveň nalézt, je zapotřebí určit, čeho se to vlastně meta-modely budou vytvářet, jaké budou nižší úrovně abstrakce, jaké jsou na ně kladené nároky, jestli budou tyto nižší úrovně abstrakcí pro další, nižší úrovně atd. Toto vše je třeba mít při návrhu na mysli, protože na nižší vrstvy abstrakce, tedy na konkrétní modely meta-modelů jsou kladeny nároky na jejich obecnost, na informaci, kterou mohou nést a vše co ve vyšší vrstvě abstrakce neumožníme vyjádřit, to nevyjádříme ani v nižších vrstvách.

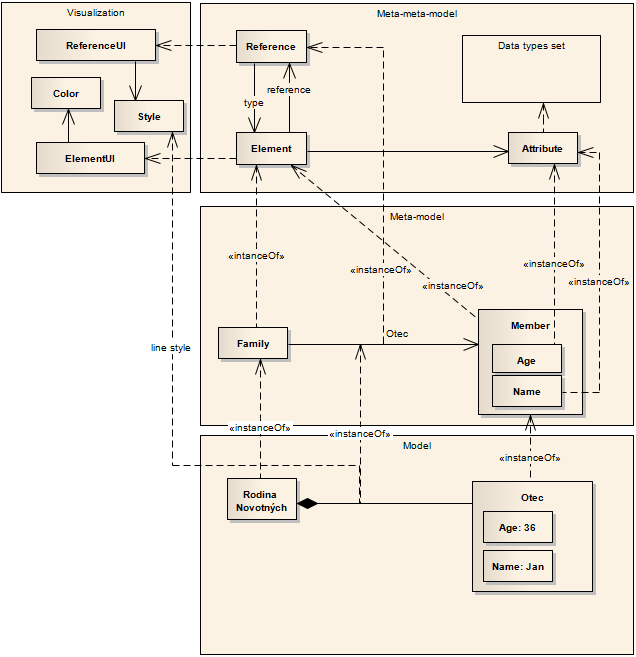
4.2 Návrh řešení

V kapitole 2 je zmíněn meta-modelovací nástroj Eclipse modeling framework. Tento framework je určen pro platformu Eclipse a jeho vývojové prostředí. Cíl této práce je vytvořit podobný meta-modelovací nástroj ve zjednodušeném provedení pro platformu NetBeans. Oproti EMF, Metamodelář se nezaměřuje na meta-modelování doménové problematiky softwaru, ale je pojat více obecně. Metamodelář se v některých principech a mechanismech inspiroval právě v EMF. Důvod proč jsem se neinspiroval v MOF, je prostý. MOF je pouze specifikace a naproti tomu EMF je implementace této specifikace. V následujících odstavcích a kapitolách popíši problémy, které bylo nutné řešit (vycházejí především z požadavků) a návrh jejich realizace.

Návrh je koncipován do 4 základních vrstev, z nichž se tato práce zabývá 3. Těmi čtyřmi základními vrstvami jsou vrstva *meta-meta-modelu*, vrstva *meta-modelu*, vrstva *modelu* a vrstva *vizualizace*. Vrstva *modelu*, ač není zahrnuta v této práci, je nedílnou součástí principu meta-modelování. V úvodní kapitole textu je zmíněn souběžný projekt s tímto projektem. Naneštěstí byl právě tento projekt zodpovědný za vrstvu *modelu*. Na druhou stranu není tvorba meta-modelů závislá na vrstvě *modelu*. Po skončení spolupráce, jsem do návrhu zavedl mechanismus, který umožní vrstvu modelu v budoucnu přidat.

Základními částmi návrhu jsou:

* Jádro – vrstva *meta-meta-modelu / meta-modelu*
* Vizualizace symbolů meta-modelu – vrstva *vizualizace*
* Grafické uživatelské rozhraní
* Serializace



Obrázek - Zjednodušené zobrazení vrstev abstrakce v podání Metamodeláře

Na [obrázku 5](#obr5) jsou zobrazeny jednotlivé vrstvy a vztah mezi nimi je znárodněn na příkladu velmi jednoduchého meta-modelu rodiny.

4.2 Jádro aplikace

Jádro představuje základ celé aplikace. Od jádra se odvíjí vše ostatní, a proto tvoří kritický bod návrhu. Při návrhu jádra je zapotřebí znát přesnou požadovanou funkcionalitu a účel zamýšlené aplikace, protože jádro je to, co definuje možnosti, schopnosti a hranice systému.

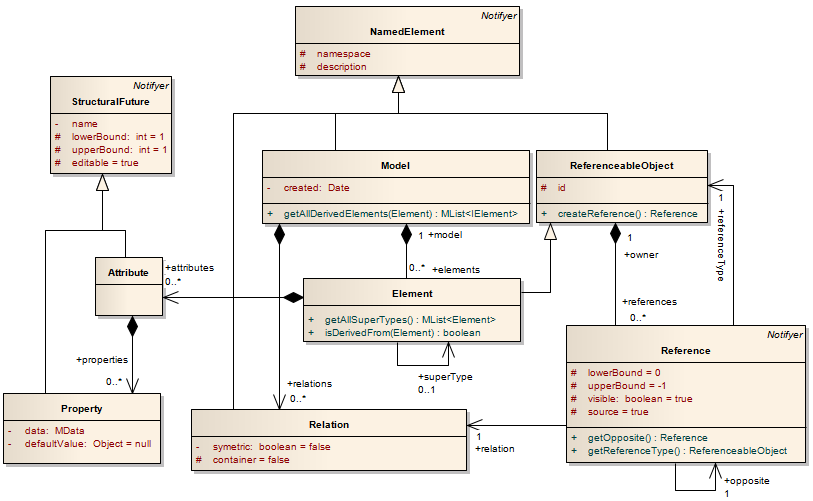
Při práci s EMF jsem nabyl dojmu, že ač se jedná o poměrně komplexní nástroj, práce s ním je poměrně jednoduchá. Při základních znalostech UML a Javy, lze snadno vytvořit meta-model doménového modelu. Procitnutí přišlo při bližším zkoumání zdrojových kódů. K zajištění na první pohled relativně jednoduché funkcionality je zapotřebí ohromné množství kódu. Samotný doménový model EMF ([obrázek 3](#obr3)) představuje minimum z celkového kódu. Zbytek zabírá aplikační logika a uživatelské rozhraní. To mě přivedlo k tomu, že je třeba věnovat velkou pozornost návrhu jádra, tedy doménového modelu a návrh koncipovat tak, aby obsahoval pouze nutné minimum k pokrytí systémových požadavků. Čím více bude jádro složité, tím více bude pracný návrh aplikační logiky a uživatelského rozhraní a tím více bude celý návrh náchylný na chyby. Na druhou stranu, jádro nesmí být příliš jednoduché, jinak by mohla nastat situace, kdy není možné pokrýt požadavky, které nepřímo s jádrem souvisí.

4.2.1 Návrh jádra Metamodeláře

Návrh jádra Metamodeláře má za úkol pokrýt požadavky P 3 až P 3.3.2. Tyto požadavky v podstatě definují vlastnosti meta-modelování, definují nejvyšší vrstvu abstrakce meta-modelů. Na [obrázku 6](#obr6) je znázorněn UML class diagram návrhu jádra Metamodeláře. V následující části textu popíši nejdůležitější části návrhu.

NamedElement

Aby bylo možné od sebe objekty rozlišovat jinak než na základě referencí v programovacím jazyce, je třeba jim přiřadit jednoznačný identifikátor. Důvodu, proč zavést identifikátor je několik. 1) je velmi jednoduché na základě identifikátoru ověřit identitu objektů, respektive jejich rovnost/nerovnost. 2) serializace a zpětná deserializace objektů si přímo vynucuje identifikátory, protože chceme, aby se zachovaly reference mezi objekty. 3) chceme zajistit unikátnost objektů a při odvolání na identifikátor chceme, abychom obdrželi pouze jeden jediný objekt. Z těchto důvodů je zavedena hlavička objektů v podobě *NamedElement.* Navíc je k identifikátoru přidán popis. Popis slouží k stručnému popisu objektu. Využití této informace je především v nižších vrstvách abstrakce. Popis bych nejlépe přirovnal k *tool tip[[3]](#footnote-3)* mechanismu.



Obrázek - Návrh jádra Metamodeláře

Element

Element reprezentuje meta-objekt. Jeho účelem je sdružovat informace o meta-objektu a jeho vazby na další meta-objekty. Zároveň si každý element drží sadu vlastností v podobě atributů.

Každý element může dědit od maximálně jednoho jiného elementu. Díky dědičnosti je možné vytvářet hierarchii. Obecné elementy můžeme konkretizovat. Příkladem může být element ***člen rodiny****,* který má vlastnosti *jméno* a *věk* a jeho potomek ***otec****,* který svého předka konkretizuje o vlastnosti *žena* a *potomci.* Výhody dědičnosti jsou všeobecně známé a věřím, že je není třeba rozepisovat. Akorát bych podotknul, že Metamodelář nepodporuje více násobnou dědičnost a ani cykly v dědičnosti. Vlastnost acyklické dědičnosti zajišťuje business logika.

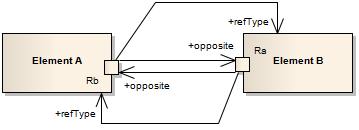
Relace

Elementy lze spojovat pomocí relací. Relace je mechanismus, který umožňuje modelovat vlastnosti vazby mezi dvěma Elementy. Relaci lze nastavit její symetrii. Symetrie v tomto případě má stejný význam jako symetrie matematické relace. Tj. relace (bavíme se o Metamodeláři) je symetrická, pokud platí, jestliže je Element1 v relaci s Elementem2, pak i Element2 je v relaci s Elementem1. Využití symetrie v Metamodeláři přináší jednu zásadní výhodu, pokud chceme mezi jednotlivými elementy nadefinovat relace různých typů a zároveň požadujeme, aby relace platily oboustranně, pak si díky možnosti nastavení symetrie ušetříme polovinu práce. Odpadne definování dvou relací (aRb, kde R je symetrická, místo aRb a bRa, kde u R nelze definovat symetrii). Element může vystupovat v relaci jako kontejner (container) pro jiné Elementy, nebo naopak jako část kontejneru. Pokud není definována relace jako kontejner, pak se jedná o standardní relaci bez speciálních vlastností.

Reference

Aby celý mechanismus relací mohl efektivně fungovat, je třeba určit, jakým způsobem se budou jednotlivé elementy dané relace účastnit. K tomu slouží třída Reference, která definuje fyzické spojení dvou Elementů. Zatímco Relace definuje vlastnosti tohoto spojení jako celku, Reference definuje role dílčích Elementů, které se dané relace účastní. Pomocí reference lze určit horní a dolní mez, čili násobnosti a ukazatel, který říká, jestli element vystupuje v relaci jako zdrojový element nebo jako cílový element.

Vazba mezi dvěma referencemi je znázorněna na [obrázku 7](#obr7). Každá reference má svůj protějšek. Neexistuje způsob jak vytvořit referenci, která by se na nic neodkazovala. Reference si drží informace o objektu, na který se odkazuje spolu s odkazem na protější referenci, která odkazuje zpět na původní element. Takto vytvořená dvojice referencí dohromady tvoří relaci (viz část Relace). Opět, je povinností business logiky, aby zajistila konzistenci v podobě odkazu obou referencí z dvojice na stejnou relaci.



Obrázek - Vazba mezi dvěma referencemi

Vlastnosti a jejich kontejnery

Způsobů, jak vyjádřit vlastnosti elementů je nepřeberné množství. Požadavek P 3.2.3 vyžaduje u každého elementu možnost vybavit ho sadou vlastností. Toto je obecný požadavek a blíže nespecifikuje komplexnost řešení. Protože má být Metamodelář více obecným meta-modelovacím nástrojem bez bližší specifikace zaměření, je mechanismus vlastností pojat také více obecně. Z toho plyne, že existuje pouze vazba vlastnosti na element. Neexistuje zde vazba element – element, kde první element vystupuje jako vlastnost druhého (toto je kompenzováno relací typu kontejner – část kontejneru a je zcela v režii nižších vrstev abstrakce, jak se s tímto vypořádají). Místo toho je definována pevná základna datových typů, které lze použít pro definici vlastností.

Vztah mezi vlastnostmi a jejich kontejnery je snaha o seskupení logicky k sobě patřících vlastností elementu a jejich jednotné manipulaci (mazání např.). *Atribut* ([obrázek 6](#obr6)) zde vystupuje jako kontejner pro vlastnosti ([obrazek 6](#obr6) - *Property*), které modelují požadovanou vlastnost. *Atribut* je identifikován pomocí jména. Každá *property* je definována svým datovým typem, jménem, multiplicitou a hodnotou.

Datové typy

Metamodelář definuje pevnou základnu datových typů. Datové typy mají vliv jak na sémantiku, tak na syntaxi meta-modelování. Sémantiku ovlivňují ve smyslu významu datového typu, tedy informace, kterou může nést a její vlastnosti. Zatímco syntaktický význam datového typu definuje, kde se může objevit daný datový typ. Pro názornost si představme element, který má definovaný atribut datového typu *String*. Syntaxe takového meta-modelu říká, že máme element, který má přiřazen atribut s datovým typem *String* a nic jiného mu přiřadit nemůžeme. Zatímco sémantický význam tohoto atributu nám říká, že s daty uloženými v atributu lze nakládat jako s posloupností znaků. Tyto aspekty, tedy sémantický a syntaktický vliv datového typu na meta-modelování, rozšiřuje bázi informací, které je schopen meta-model obsáhnout.

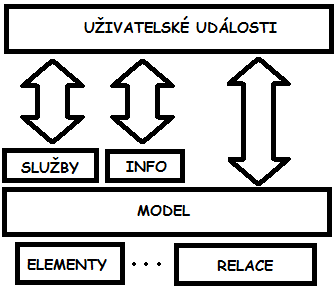
Metamodelář definuje následující datové typy:

* STIRNG – řetězec znaků
* DOUBLE – číslo s desetinnou čárkou
* INTEGER – celé číslo
* CHARACTER – znak
* BOOLEAN – logická proměnná
* COLOR – barva definovaná ve smyslu RGB standardu

Datová základna byla vybrána s ohledem na obecnost meta-modelování. Cílem je umožnit definovat obecné datové typy, které pokryjí standardní nároky na uložení nejpoužívanějších typů dat.

Model

Model zde hraje roli rozhraní mezi „syrovými daty“ a zbytkem aplikace. Na [obrázku 7](#obr7) je znázorněna přístupová vrstva (**MODEL**) k datům modelu a klienti, kteří tuto vrstvu využívají. Do skupiny **UŽIVATELSKÉ UDÁLOSTI** se řadí zprávy zaslané od uživatele (prostřednictvím uživatelského rozhraní) a ostatní části systému, které vyžadují spolupráci s modelem. Mezi ně patří například *Serializer*, který se stará o persistentní vrstvu. Komponenta **SLUŽBY** poskytuje služby spojené s tvorbou a modifikací meta-modelu v bezpečném „módu“. Komponenta **INFO** je informační centrála nad modelem. Poskytuje sadu strukturovaných dat, která popisují momentální stav modelu. Navíc slouží jako „průvodce“ modelem, který na základě předložených identifikačních údajů zjistí, zda hledaný objekt existuje či ne, a v případě existence hledaného objektu je schopen ho předložit klientovi. Vrstva klienta má i přístup přímo k modelu samotnému. Zejména komponenty, které vyžadují model “pouze pro čtení“, nebo potřebují získat specifickou informaci o modelu, kterou jim není komponenta **INFO** schopna poskytnout, přistupují k modelu přímo. Příkladem je již výše zmíněný *Serializer*. Při serializaci (tj. uložení modelu např. na vnější paměť) je třeba znát veškeré detaily modelu. Komponenta **INFO** poskytuje informace pouze o hrubé struktuře modelu, které jsou dostačující pro uživatele při tvorbě meta-modelu, ale nejsou dostačující pro serializaci. Při deserializaci (tj. opětovné obnovení modelu v paměti počítače ze souboru z vnější paměti) již není potřeba přímý přístup k modelu a namísto toho může *Serializer* použít komponentu **SLUŽBY** při sestavování modelu.



Obrázek 8 - Model jako rozhrani mezi daty a klienty

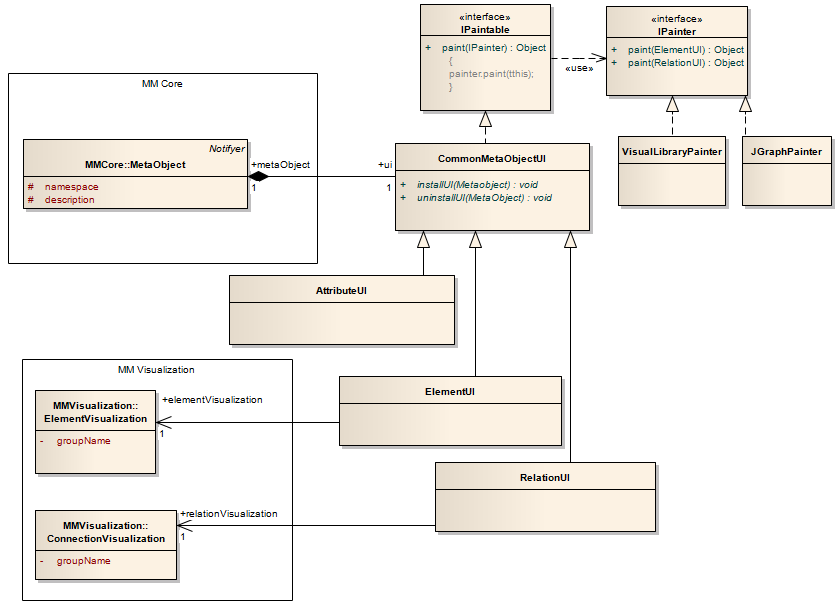
V úvodu podkapitoly [4.2](#vrstvy_metamodelovani_navrh) jsem zmínil vrstvu *modelu* a důvod, proč není součástí práce. Dále jsem zmínil, že Metamodelář zavádí mechanismus, aby bylo možné podporu modelování na základě meta-modelu přidat. Právě komponenty **SLUŽBY** a **INFO** definují, zda se jedná o vrstvu *modelu* nebo *meta-modelu*. Metamodelář definuje obecné rozhraní pro obě komponenty. Implementací těchto rozhraní definujeme sémantiku jazyka meta-modelování, potažmo modelování. Syntaxe zůstává stejná. Metamodelář implementuje pouze sémantiku pro meta-modelování.

4.3 Vizualizace symbolů meta-modelu

Každý meta-objekt má referenci na svoje uživatelské rozhraní (dále UI – user interface). Na [obrázku 9](#obr9) je znázorněn mechanismus vizualizační vrstvy a na [obrázku 10](#obr10) je znázorněn sekvenční diagram vykreslování jednotlivých meta-objektů.

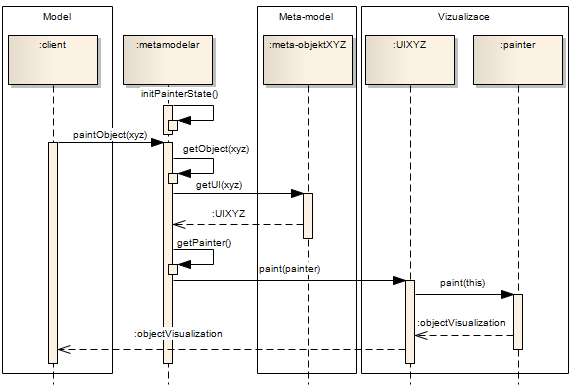
Je zde použit návrhový vzor *Visitor*. Roli *visitora* zde hrají jednotlivé třídy pro vykreslování (Metamodelář podporuje pouze vykreslování pomocí knihovny VisualLibrary, pro kterou je implementovaný „vykreslovač“ – *VisualLibraryPainter*), které „navštěvují“ jednotlivé UI meta-objektů. Zde je prostor pro budoucí podporu dalších grafických knihoven. Pro rozšíření podpory pro novou grafickou knihovnu stačí implementovat rozhraní *IPainter*.

Jednotlivé prvky konkrétních typů UI se liší. Např. u *ElementUI* lze nastavit obrázek elementu, jeho ohraničení, popisek atd. Naproti tomu u *RelationUI* lze nastavit styl čáry spojující dva meta-objekty, tvary šipek u zdrojového a cílového meta-objektu, popisky atd. Nicméně, ne každý meta-objekt se musí povinně umět vykreslit, např. u Attribute a Property není potřeba je jakýmkoliv způsobem vykreslovat. Nicméně ta možnost tady je, ale Metamodelář ji nevyužívá.

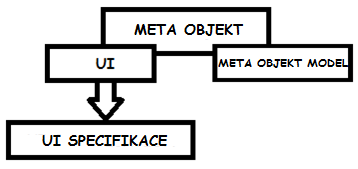


Obrázek 9 - Vizualizační vrstva

Sekvenční diagram na [obrázku 10](#obr10) znázorňuje pořadí volání jednotlivých částí Metamodeláře a zároveň ilustruje, které části vrstvy se podílejí na vykreslení. Standardně je Metamodelář nastaven na vykreslování pomocí knihovny *VisualLibrary* (jediná implementace). První krok je získání grafického kontextu meta-objektu. V kroku druhém je získán „vykreslovač“ – implementace obecného rozhraní *IPainter*. Jakmile máme grafický kontext a objekt, který jej umí vykreslit, vyžádáme si od „vykreslovače“ instanci grafické reprezentace na základě grafického kontextu. Obdržená instance grafické reprezentace meta-objektu je vrácena volajícímu klientovi.



Obrázek 10 - Sekvenční diagram vykreslování meta-objektů, v tomto případě se jedná o Element



Obrázek 11 - Ilustrace spojení vrstvy meta-modelu a vizualizační vrstvy

Spojení vrstvy meta-modelu a vizualizační vrstvy je realizováno pomocí přípojného bodu vizualizace u jednotlivých meta-objektů (viz [obrázek 11](#obr11)). **UI** má na starosti spojení meta-objektu s patřičnou specifikací pro daný typ meta-objektu. Tento mechanismus zajišťuje konzistenci mezi meta-objekty různých typů (atribut, element, relace atd.) a jejich grafické reprezentace. Díky tomu je zaručená „typová bezpečnost“ grafické reprezentace.

4.4 Grafické uživatelské rozhraní

Požadavek P 3.1 vyžaduje pro tvorbu meta-modelů grafické uživatelské rozhraní (dále GUI – graphical user interface). Dále zadání definuje platformu NetBeans [[4](#lit4)][[5](#lit5)][[6](#lit6)] jako prostředí pro Metamodelář. Nabízí se dvě možnosti realizace GUI. 1) Metamodelář bude samostatná aplikace, nebo 2) Metamodelář bude implementován v podobě pluginu do NetBeans IDE[[4]](#footnote-4).

Vybral jsem si druhou variantu, protože Metamodelář není natolik rozsáhlá a komplexní aplikace, aby musela existovat jako samostatný celek. Navíc je možné do budoucna Metamodelář rozšířit o služby, které může poskytovat veřejným rozhraním vnějším modulům (toto byl původní záměr Metamodeláře, dokud se nesešlo ze spolupráce s projektem zmíněným v úvodu textu). To by nebylo možné v konceptu samostatné aplikace. Jako druhotnou výhodu tohoto rozhodnutí vidím v NetBeans IDE samotném, jelikož poskytuje funkce, na které se stačí pouze připojit. V samostatné aplikaci by se tyto funkce musely implementovat. Mezi tyto funkce patří správce projektů, správce oken, menu, standardní dialogy a jiné funkce a prvky převážně z GUI.

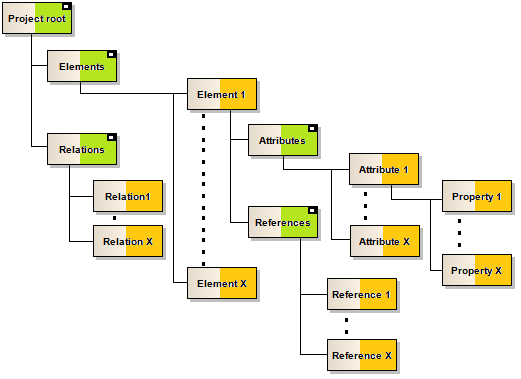
Protože NetBeans IDE již samo o sobě definuje hlavní okno aplikace spolu s dalšími prvky, je zapotřebí pouze vytvořit obsah, který bude s hlavním oknem a jeho prvky spolupracovat. Nejdůležitější částí návrhu GUI bylo rozhodnout, jakým způsobem se bude definovat meta-model. Rozhodoval jsem se mezi třemi základními způsoby:

1. Definice meta-modelů pomocí sady tabulek. Každý element je reprezentován pomocí několika tabulek. Mezi základní tabulky patří tabulka s relacemi, která mapuje vztahy mezi jednotlivými elementy, tabulka s atributy, která definuje sadu vlastností elementu a další tabulky.
2. Pomocí grafové sítě založené na teorii grafů. Uzly představují jednotlivé elementy, zatímco hrany představují vazby mezi těmito elementy.
3. Reprezentace meta-modelu v podobě stromové struktury. Kde uzly vyšší úrovně reprezentují elementy, které obsahují uzly vlastností a referencí na jiné elementy.

1. možnost zahrnuje práci s tabulkami. Tuto možnost jsem zavrhnul v momentě, kdy jsem si uvědomil množství tabulek, které by musel uživatel spravovat v případě rozsáhlejšího meta-modelu. Pokud by model obsahoval 10 a více elementů, znamenalo by to minimálně 20 a více tabulek. 2. možnost by byla ideální do té doby, než by uživatel začal definovat komplexní síť vztahů mezi elementy. Jelikož by každá hrana představovala jednu relaci mezi elementy, pak už od malého počtu vztahu by začal být celý graf nepřehledný až chaotický. 3. možnost je kompromisem prvních dvou. Při velkém počtu elementů a relací roste složitost stromu lineárně. Toto řešení na druhou stranu trpí menší přehledností než v případě řešení s tabulkami. Nicméně jsem se rozhodl právě pro tuto třetí možnost. Důvodem je snadné a přirozené ovládání a přijatelná složitost úměrná komplexnosti meta-modelu.

Stromová struktura

Návrh stromové struktury byla poměrně jednoduchá záležitost. Návrh se skládá z definování pevně daných uzlů a uzlů, jejichž počet se bude měnit. Struktura stromu vychází z jádra Metamodeláře.



Obrázek 12 - Schéma stromové struktury znázorňující meta-model

Na [obrázku 12](#obr12) je zobrazena obecná struktura stromu meta-modelu. Obdélníky s černým čtvercem v pravém horním rohu označují povinné uzly, které jsou přítomny vždy, když je přítomen i jejich rodič. Tyto uzly tvoří hierarchii stromu. Obdélníky bez černého čtverce zobrazují uzly, které jsou volitelné, a jejich výskyt záleží na datech v meta-modelu. Tyto uzly představují data v meta-modelu.

Každému typu uzlu (*Elements, Element, Reference* …), je přiřazena množina akcí, které jsou nepřímo (přes komponentu *Controller* v návrhovém vzoru MVC [[10](#lit10)] – požadavek P 3.1) napojené na komponentu **SERVICES** ([obrázek 8](#obr8)). Charakter akce záleží na uzlu a datech, které reprezentuje. Uzlům tvořící hierarchii stromu jsou přiřazeny akce, které mají za úkol tvorbu svých potomků – tj. přidání nových prvků meta-modelu. Uzly znázorňující obsah meta-modelu (oranžově označené bez černého čtverce) mají přiřazeny akce, které slouží k modifikaci dat, které reprezentují.

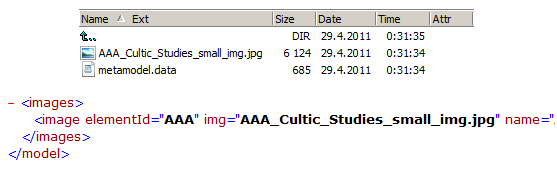
Každý uzel ve stromové struktuře si registruje posluchače u patřičné části jádra Metamodeláře. Pokud je stav jádra změněn, je notifikován daný uzel o změně. Uzel se poté aktualizuje na základě nových dat. To může znamenat i samotné smazání či přidání nového uzlu. Nejčastěji se však jedná o aktualizaci popisku uzlu, či jiných hodnot, které je uzel schopen zobrazit. Zde jsou možnosti široké. NetBeans platforma poskytuje API, které je určené právě pro rozšíření schopnosti zobrazit dodatečné informace u jednotlivých uzlů. Rozhraní, která se podílí na vizualizaci pomocí uzlů se jmenují Nodes API a Explorers & Property Sheet API a jsou podrobněji vysvětlena v kapitole věnující se implementaci.

4.5 Serializace

Uschovat data mezi jednotlivými spuštěními aplikace je záležitost, s kterou se musí vypořádat většina dnešních aplikací. Zvolit správný formát je velmi důležité. Výstupní formát definuje, jakým způsobem je možné data dále zpracovat, definuje složitost zpracování či bezpečnost.

Souborový formát

Metamodelář definuje svůj vlastní formát. Data jsou identifikována na základě koncovky jména souboru. Konkrétně se jedná o koncovky “.*mm*“ a “.*MM*“. Soubor je založen na principu *ZIP[[5]](#footnote-5)* souboru. *Soubor.mm* obsahuje samotná data meta-modelu a další data, která přímo s meta-modelem souvisí. Jedná se především o obrázky, které jsou využité ve vizualizační vrstvě. Data meta-modelu jsou uložená v XML formátu. XML formát jsem vybral pro jeho snadné zpracování a lidmi čitelnou formou. Navíc se jedná o celoplošně uznávaný standard. *Soubor.mm* lze „rozbalit“ jakýmkoliv nástrojem, který podporuje standart *zip* archivů. Výsledkem bude soubor *metamodel.data* – XML s popisem meta-modelu a množina obrázků, které jsou v meta-modelu použity. Výhodu v tomto řešení vidím v ucelenosti. S daty meta-modelu se pracuje jako s jedním celkem – jedním souborem, ale přesto jsou čistá data meta-modelu oddělená od zbylých dat.



Obrázek 13 - Schéma souboru s příponou *.mm* nebo *.MM*

Na [obrázku 13](#obr13) je zobrazen příklad obsahu souboru definovaného Metamodelářem. Horní půlka obrázku zobrazuje obsah souboru po rozbalení. Spodní půlka obrázku zobrazuje výňatek z *metamodel.data*, konkrétně z části odkazující se na obrázek, který je mapován na konkrétní element. Povinná položka archivu je *metamodel.data,* která představuje XML soubor s popisem meta-modelu. Ostatní položky jsou volitelné. Pokud jsou v archivu souboru *xyz.mm* obsažené jiné položky než *metamodel.data*, pak se na ně meta-model odkazuje podle potřeby. Struktura souboru *metamodel.data* není příliš důležitá. Nicméně je vhodné popsat některé její rysy. XML soubor v podstatě kopíruje strukturu stromu popsaného v předchozí kapitole. Navíc je obohacen o reference na externí data v archivu, ve kterém se nachází.

Metamodelář je koncipován tak, aby se o strukturu souboru staral automaticky sám. Jako příklad uvedu správu obrázků. V případě definice vizuální vrstvy meta-modelu, uživatel může přiřadit jednotlivým elementům obrázek. Ten je vybrán pomocí „file chooser“ dialogu z vnější paměti (pevný disk např.). Při serializaci se provedou standardní procesy pro převod meta-modelu do XML souboru. Obrázek, který byl vybrán jako grafická reprezentace elementu je archivován (vytvořena kopie) spolu s XML souborem a v XML souboru je vytvořen odkaz na tento obrázek. Při deserializaci se obrázek přečte tentokrát již z archivu. Díky tomu není meta-model závislý na vnějších zdrojích a nemůže dojít k situaci, kdy se meta-model odkazuje na data, která již neexistují, protože byla smazaná nebo změněná jiným procesem.

Metamodelář dále definuje akce nad soubory typu “.mm“ nebo “.MM“. Akce jsou definované v souladu s NetBeans platformou [[4](#lit4)][[5](#lit5)][[6](#lit6)]. Patří mezi ně akce *OPEN* – otevření souboru v editoru, který obsahuje stromovou strukturu meta-modelu dle předchozí podkapitoly, *DELETE* – odstraní soubor z projektu, *SAVE* – uloží změny v meta-modelu a *RENAME* – přejmenuje meta-model.

Projektový formát

Metamodelář také definuje projektový formát. Jedná se o velmi jednoduchou záležitost, a proto její popis zestručním. Definice projektového formátu vychází z prostředí NetBeans platformy a NetBeans IDE samotného. NetBeans IDE již v základním oknu definuje dokovatelné okno - správce projektů. Je pouze na pluginu, jestli toto okno využije a nadefinuje vlastní projektový formát. Pokud je projektový formát nadefinován, NetBeans IDE se postará o jeho správu. Definice projektového formátu se skládá z následujících částí:

* Definice obsahu, který musí složka splnit, aby byla prohlášená za složku projektu.
* Definice položek, které se mají zobrazit ve správci projektů
* Definice akcí nad projektem, jako jedním celkem.

Metamodelář deklaruje složku za projekt, pokud obsahuje alespoň jeden soubor s koncovkou *“.mm*“ nebo *“.MM*“. V tomto případě vytvoří stromovou strukturu projektu pojmenovanou po složce, ve které se soubory nacházejí. Ve stromové struktuře se zobrazí pouze tyto soubory jako uzly stromu. Každý uzel představuje soubor s definovaným souborovým typem. S každým takovýmto uzlem je možné pracovat v rámci definovaných akcí (viz předchozí podkapitola).

Kapitola 5

Implementace

V této kapitole se věnuji technologiím, prostředím a API[[6]](#footnote-6), které jsem využil při realizaci Metamodeláře.

5.1 NetBenas platforma

Již samo zadání určilo, na jaké platformě se bude Metamodelář zakládat. NetBeans platforma [[4](#lit4)][[5](#lit5)] je založena na Swing API[[7]](#footnote-7) a nabízí širokou škálu API ulehčující vývoj desktopových aplikací. Časté využití API je na poli oken, akcí, souborů a další. NetBeans platforma podporuje modulární vývoj. Modul je základní jednotka aplikace poskytující specifickou funkci. Modul se skládá z Java tříd, konfiguračních souborů a knihoven. Moduly mohou být jeden na druhém závislé. Přínosem modulární aplikace je rozložení celého konceptu na menší celky, které se lépe spravují. Moduly podporují znovu použitelnost, udržovatelnost a lepší testovatelnost. Na [obrázku 14](#obr14) je znázorněna vazba mezi NetBeans Platform a NetBeans IDE. Všimněte si členění do modulů na všech úrovních. Popis modulů (API NetBeans Platformy) využitých v Metamodeláři naleznete v kapitole [5.3](#APIs).



Obrázek - Vztah mezi NetBeans platformou, NetBeans IDE a jednotlivými moduly, které dělají z NetBeans IDE použitelný nástroj. (Zdroj: [[6](#lit6)])

5.2 NetBeans IDE

NetBeans IDE [[6](#lit6)] (dále NB IDE) je vývojové prostředí s grafickým uživatelským rozhraním založeným na NetBeans Platformě (viz [obrázek 14](#obr14)). Funkcionalita NB IDE je závislá na pluginech. Na [obrázku 14](#obr14) je zobrazeno NB IDE jako nadstavba nad NetBeans Platformou a jako kontejner pro pluginy. Samotné NB IDE je bez pluginů pouze prázdná schránka bez jakékoliv funkcionality. To co tvoří NB IDE populárním vývojovým prostředím je právě velká základna pluginů, které je možno do NB IDE přidat. Důvodů, proč využít NB IDE jako kontejner pro své pluginy je hned několik. NB IDE, mimo jiné, zahrnuje následující:

* Grafické uživatelské rozhraní (GUI)
* Možnost využít funkce pluginů a nástrojů třetích stran
* Zkrácení času vývoje softwaru
* Pluginy lze spustit a vyzkoušet přímo v prostředí, ve kterém se vyvíjí a to i uprostřed vývojového cyklu

5.3 Použité API

API NetBeans Platformy hrály při vývoji Metamodeláře významnou roli. Především ulehčily a zkrátily vývojový cyklus aplikace. V této podkapitole popíši základní API využité při realizaci Metamodeláře.

5.3.1 Nodes API a Explorers & Property Sheet API

Velkou nevýhodou Swingových komponent v Jave je nesourodost jejich datových modelů. *JTree*, *JTable*, *JList* a další, všichni mají svůj specifický datový model. V případě, že se rozhodneme změnit vizualizaci dat, musíme vytvořit nový datový model pro námi vybranou komponentu.

Nodes API [[4](#lit4)][[5](#lit5)] toto řeší elegantním způsobem. Definuje datový model v podobě takzvaných nodes (uzlů), které zapouzdřují doménová data. Každý uzel poskytuje prezentační informace, jakými jsou: zobrazení jména uzlu, ikona uzlu, vlastnosti uzlu (respektive objektu, který je uzlem zabalen) a akce nad doménovými daty, které zaobaluje. Důležité je také zmínit vlastnost vnoření, kde každý uzel může obsahovat jiné uzly.

Explorers & Property Sheet API [[4](#lit4)][[5](#lit5)] slouží k zobrazení dat a jako datový model je zde využit výše zmíněné Nodes API. Eplorers API poskytuje různé způsoby zobrazení dat. Jsou jimi např. *List View* – obdoba *JList*, *Choice View* – obdoba *JChoice* nebo *Been Tree View* – obdoba *JTree* (využit v Metamodeláři).

Property Sheet API poskytuje mechanismus pro manipulaci s daty zapouzdřenými pomocí Nodes API. Pomocí tohoto API lze vytvořit tzv. *Sheets*, který spravuje sadu tzv. *Properties*, které mapuje na doménová data. Pro každé Property je nutno definovat editor, který bude schopen manipulovat s daty jemu svěřenými. Pro primitivní datové typy (int, boolean, double…) a pro základní objektové typy (Color, Date…) jsou už v základu tyto editory definovány. Pro specifická data je třeba vytvořit vlastní editor.

Metamodelář využívá Nodes API společne s Explorers & Property Sheet k vizualizaci a manipulaci doménových dat pomocí GUI. Při tvorbě GUI bylo zapotřebí kompromisu v podobě zapouzdření doménových dat. Sheets totiž manipulují s daty přímo bez možnosti vložení vlastního mezičlánku, který by události plynoucí z konkrétních editorů zachytil, zpracoval a až poté změnil doménová data. Což vyžaduje architektura MVC. Tento problém jsem vyřešil pomocí návrhového vzoru *Proxy.* Data, s kterými se manipuluje pomocí *Sheets*, jsou zabalena *proxy* objektem, který má stejné rozhraní jako data, která zaobaluje a při pokusu jednotlivých editorů daná data změnit, *proxy* objekt přepošle požadavek patřičnému manažeru v *Controller* komponentě (MVC).

5.3.2 Project API

Project API umožňuje definovat tzv. projektový typ. Termín projektový typ je v NetBeans Platformě použit pro skupinu složek a souborů, s kterými se zachází jako s jedním celkem. Projekt ulehčuje uživatelům manipulaci s daty, které daný projekt tvoří. Jeho předností je definice obsahu, který se zobrazí v projektovém správci[[8]](#footnote-8). Pomocí Project API můžeme nadefinovat akce nad námi definovaným projektovým typem. Tyto akce budou přístupny v projektovém správci. Projektový správce pracuje s uzly (viz Node API) a vyžaduje definici stromu uzlů po projektovém typu.

5.3.3 Správa souborů a jejich typů

Před tím, než vysvětlím, jak se vytváří podpora pro souborový typ, nejprve vysvětlím principy podpory souborového systému v NetBeans platformě.

Layer.xml

Každý modul může obsahovat konfigurační soubor *layer.xml,* který obsahuje pokyny pro NetBeans IDE. Při startu aplikace se načtou všechny tyto konfigurační soubory a vykonají se operace podle dat v nich uložených. Ve většině případů se jedná o registraci služeb jednotlivých modulů. Tj. i definice souborových typů.

FileObject

FileObject je kontejnerem pro data v souborovém systému počítače. Neví nic o struktuře nebo typu dat, obsahuje pouze množinu bytů. Lze jej srovnat s *java.io.File,* ale navíc nabízí podporu pro notifikaci změn nad daty či zamykání dat.

DataObject

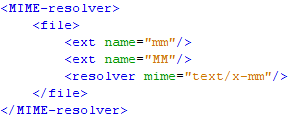
*DataObject* je zodpovědný za mapování a reprezentování obsahu souboru. *DataObject* zná datový typ souboru (může reprezentovat 1 a více souborů).

DataLoader

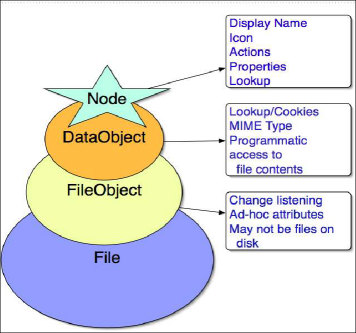
Je to most mezi *FileObject* a *DataObject*. Pokud je definována podpora pro konkrétní souborové typy, *DataLoader* přiřadí konkrétnímu *FileObjectu* příslušný *DataObject*. S *DataLoaderem* nepřijde Metamodelář do styku, je plně v režii NetBeans IDE.

5.3.4 Podpora souborového typu

První věc, kterou je potřeba udělat, je definice tzv. *MIME* typu (viz [obrázek 15](#obr15)). Je to v podstatě meta-informace o typu dat, která slouží pro *DataLoader*, aby byl schopen vytvořit *DataObject* na základě *MIME*.



Obrázek 15 - Definice MIME typu pomocí konfiguračního souboru modulu



Obrázek 16 - Vztah mezi *File*, *FileObject* a *DataObject* [[13](#lit13)]

*MIME* definice říká, jaký charakter má daný typ souboru, a určuje jeho tzv. resolver, tj. v tomto případě koncovku jména souboru. Je to velice jednoduchý způsob, jak na základě jména souboru v počítači určit jeho datový typ a tudíž operace nad tímto souborem. Druhou věcí je definovat akce nad naším novým datovým typem. To se děje taktéž pomocí konfiguračního souboru modulu, kde se definuje mapování akcí na jejich implementace. Mezi časté akce patří: otevři, ulož, zavři atd.

Souborové typy v NetBeans IDE mají své ikony, menu položky a chování. To co vidíme ve správci projektů jsou uzly (viz Node API), které poskytují funkcionalitu jako akce nad objekty jako soubory či složky. Pokud rozbalíme uzel v projektovém správci, IDE se zeptá pro každou položku v  uzlu každého registrovaného *DataLoaderu*, jestli poznává „co to je“. První *DataLoader*, který se přihlásí, vytvoří *DataObject* pro daný soubor. *DataObject* je pak zodpovědný za vytvoření uzlu pro každý soubor/složku, kterou rozpoznává jako jeho datový typ. Tyto uzly jsou to, co ve skutečnosti vidíme ve správci projektů. Vztah mezi *File*, *FileObject*, *DataObject* a uzly je zobrazen na [obrázku 16](#obr16).

Kapitola 6

Testování

6.1 Domény testování

Ověření správné funkčnosti aplikace je nedílnou součástí procesu vývoje softwaru. Testuje se od započetí práce na projektu až po odstavení softwaru. V různých stádiích procesu vývoje softwaru se používají různé techniky testování. Při vývoji Metamodeláře jsem se zaměřil na stádia:

* sběr a analýza požadavků
* návrh a implementace
* nasazení do provozu

Testování v každém stádiu vyžaduje jiný přístup testera. Vstupní a výstupní data závisí na charakteru testované domény. Nástroje, kterými se ověřuje správnost domény, jsou závislé na konkrétním obsahu domény. Volba správného nástroje je klíčovým bodem při návrhu testů. V následujících odstavcích popíši jednotlivé domény, nástroje, které jsem použil pro ověření správného řešení a způsob, jakým jsem jednotlivé domény testoval. Každá část je rozdělena na popis problematiky testování včetně postupu testování a na výsledky testování.

6.1.1 Testování systémových požadavků

Popis

Systémové požadavky jsou souhrnem chtěných vlastností aplikace. Definicí popisující, jak by měl takový požadavek vypadat, lze nalézt mnoho. Při ověřování správnosti požadavků jsem se držel specifik Unified Process (UP – [[3](#lit3)]). Ověření proběhlo výhradně manuální formou. Každý požadavek byl validován vůči správné formulaci požadavku. Tj. každý požadavek musí být ve formátu *<id><systém> bude <funkce>*. Zároveň byla báze požadavků zkontrolovaná z hlediska konzistence. Byl kladen důraz na vyhledání potencionálních protiřečení. Nakonec byla báze zkontrolovaná vůči vylučovacímu efektu. Tj. zda se požadavky navzájem nevylučují.

Výsledek

Požadavky se v průběhu vývoje aplikace v malé míře měnily. Tudíž bylo zapotřebí po každé změně provést nové testování. V průběhu testování se neobjevily žádné závažné nedostatky. Jediné co bylo třeba předělat, byly identifikátory jednotlivých požadavků.

6.1.2 Testování návrhu a implementace

Popis

Testování návrhu a implementace je s porovnáním s ostatními testy nejrozsáhlejší, co se týče věnovaného času. Techniky testování bych rozdělil na dvě hlavní části. A to testování manuální a automatické. Testy probíhaly v průběhu celé fáze návrhu a implementace. Návrhová část vývoje byla testována především vůči požadavkům. Tj. zda návrh zahrnuje veškeré body požadavků a zda dostatečně robustně tyto body reprezentuje. Dále byl návrh testován i v průběhu fáze implementace. Především pro korektnost spolupráce jednotlivých částí návrhu (tzv. integrační testování). Testování implementace bylo provedeno automatickými testy za pomocí frameworku JUnit4[[9]](#footnote-9) [[14](#lit14)] a manuálními testy, které se týkaly především grafického uživatelského rozhraní.

Manuální testování bylo započato až ve fázi implementace, přesněji ve fázi implementace grafického rozhraní. V této fázi bylo možné pomocí rozhraní testovat implementované funkce. Cílem testování bylo ověřit správné zpracování vstupů, reakce na správné a chybné vstupy, správné rozložení grafických komponent a jednoduchost a srozumitelnost ovládání. Manuální testování hrálo významnou roli především v ověřování nově implementovaných funkcí. Protože automatické testy v době těsně po implementaci konkrétní funkce neexistovaly, byl toto jediný způsob testování. Ověření správného výstupu testování bylo provedeno čistě visuální formou. Tj. např. kontrola správné reakce na zmáčknutí tlačítka či ověření správného změnění hodnoty v meta-modelu na základě informací zobrazených v GUI.

Pro automatické testování jsem zvolil framework JUnit4 [[14](#lit14)]. Jeho výhodou je jednoduché ovládání. Psaní testovacích případu nevyžaduje žádné speciální znalosti (nepočítám znalosti spojené s návrhem testů) mimo Javy. Jednotkovými testy jsem ověřil kritické části implementace. Jednotky jsem také použil na integrační testování. I když jednotkové testování není přímo určené pro tento druh testů, lze s nimi do značné míry ověřit spolupráci jednotlivých komponent. Mezi kritické části implementace patří jádro a komunikace mezi komponenty.

Výsledek

I přesto, že je manuální testování méně efektivní ve větších projektech než testování automatické, tvořilo kritický bod fáze testování. Manuálním testováním jsem byl schopen velice rychle ověřit jednoduché případy, jakými jsou reakce GUI na akce uživatele, vyvolání dialogu s chybovou hláškou tam, kde se vyvolat měl, správná reakce na vstupy atd. Díky manuálnímu testování jsem odhalil veliké množství chyb, které sice nebyly závažné ani nijak rozsáhlé, ale jejich odstranění bylo potřebné pro další vývoj. Pro jednoduché případy představovalo manuální testování rychlejší formu ověření správnosti, než automatické testování.

Automatickým testováním jsem odhalil celou řadu chyb. Chyby spadaly do skupin od méně závažných až kritických. Mezi ty méně závažné chyby patřily chyby typu špatné ukládání hodnot do objektů meta-modelu. Mezi kritické chyby se řadily chyby typu nesprávné provedení vymazání objektů z meta-modelu nebo špatné zpracování vstupů u služeb spojených s manipulací meta-modelu. Pak zde byly chyby, které se vyskytovaly nepravidelně, závislé především na vstupu. Tyto chyby většinou spočívaly v chybně definované podmínce, či špatném pořadí vykonávání instrukcí. Dále byly odhaleny chyby neinicializovaných proměnných. Tyto chyby byly nalezeny velice rychle a byly i snadno opravitelné.

6.1.3 Testování v provozu

Popis

Aplikace byla testovaná v provozu formou manuálního testování. Testovacím prostředím bylo zvoleno prostředí Microsoft Windows 7 professional 64b a Ubuntu 9.04. Testováno bylo ve verzi Javy v1.7.0, v1.6.20 a v případě Ubuntu v1.6.19. Verze NetBeans IDE bylo v případě Windows 6.9 a 7.0, v případě Ubuntu byla použita verze 6.9.

Testy se skládaly z verifikace[[10]](#footnote-10) a validace[[11]](#footnote-11). Verifikace probíhala manuální formou a spočívala v projetí celé aplikace a odzkoušení všech funkcí. I když není v silách žádného testera vyzkoušet všechny možné vstupy u všech funkcí, snažil jsem se volit vstupy ve smyslu: správný vstup, žádný vstup a chybný vstup (nepovolený vstup, vstup mimo meze atd.). Validaci jsem provedl způsobem projetí jednoho požadavku po druhém a ověření, že byl požadavek implementován a odpovídá specifikaci.

Výsledek

Výsledkem validace a verifikace bylo odhalení několika nedostatků. Závažným nedostatkem bylo nedodržení jednoho z požadavků. Konkrétně se jednalo o požadavek P 3.3.2. Definice typu kontejner/část kontejneru byla v režii Reference nikoliv Relace. To si vyžádalo poměrné rozsáhlé změny v kódu. Dále byly objeveny minoritní chyby především spojené s uživatelským rozhraním. Ve většině případů se jednalo o špatné popisky v GUI a o špatné zobrazení informací ač byly informace v modelu správně uložené.

Kapitola 7

Závěr

7.1 Shrnutí

K realizaci nástroje byly použity standardní techniky a nástroje všeobecně známé. K implementaci byl použit jazyk Java, jehož volba vyplývá ze zadání. Nicméně to považuji za dobrou volbu, protože umožňuje dobrý objektově orientovaný vývoj a aplikace založené na Javě jsou platformě nezávislé. Prostředí, pro které je aplikace vytvořena taktéž vyplývá ze zadání. Tím prostředím je NetBeans platforma a ukázalo se to jako dobrou volbou. NetBeans platforma poskytuje veliké množství funkcionality, která díky tomu nemusí být vývojářem implementována. Jako stinnou stránku NetBeans platformy bych vyzdvihl její poměrně velkou komplexnost. V průběhu vývoje jsem se musel seznámit s relativně velkým množstvím API poskytovaných NetBeans platformou a osvojit si principy fungování jednotlivých částí a hlavně způsoby, jakým mezi sebou tyto části komunikují.

Jako stěžejní část práce bych označil část návrhu a implementace. Nicméně, jak čas ukázal, nejsložitější částí byla implementace uživatelského rozhraní. Nešlo ani tak o náročnost intelektuální (zde dominovala část návrhu jádra, především část týkající se konceptu meta-modelování), ale o implementaci a zároveň ověření správné funkčnosti uživatelského rozhraní, které bylo velice časově náročné s porovnáním s ostatními částmi.

7.2 Budoucí práce

Jako hlavní cíle dalšího vývoje bych označil následující:

* Rozšíření možností visuální vrstvy. Především vytvořit více flexibilní a variabilní prostředí pro definici grafické podoby jednotlivých objektů meta-modelů
* Implementace vrstvy modelu. Toto vidím jako nejdůležitější bod pro budoucí práci. Jelikož v současné době neexistuje nástroj, který by uměl výstupní meta-modely dále zpracovat, vidím toto jako logický krok.
* Implementace podpory dalších grafických knihoven, aby nástroje pracující s meta-modely nebyly závislé výhradně na *VisualLibrary* [[9](#lit9)].

7.3 Závěr

Cílem práce bylo vytvořit meta-modelovací nástroj s podporou grafických notací. Realizovaným výstupem je analýza a realizace meta-modelu, který definuje obecnou strukturu požadovaného modelu s možností rozšíření o grafické prvky. Druhým výstupem je nástroj Metamodelář, který umožňuje práci s meta-modely a umí zobrazit u jednotlivých elementů a relací jejich grafickou podobu. Zadání se tak podařilo splnit.

Literatura

1. OMG [online]. 2007 [cit. 2011-04-02]. Catalog of OMG Modeling and Metadata Specifications. Dostupné z WWW: <http://www.omg.org/spec/MOFVD/2.0/>.
2. STEINBERG, Dave, et al. EMF: Eclipse Modeling Framework. Second Edition. [s.l.] : Addison-Wesley Professional, 2008. 744 s. ISBN 978-0-321-33188-5.
3. ARLOW, Jim; NEUSTADT, Ila. UML 2 : a unifikovaný proces vývoje aplikací. Holandská 8, 639 00 Brno : Computer Press, a.s., 2008. 539 s. ISBN 978-80-251-1503-9.
4. Heiko Böck. The Definitive Guide to NetBeans™ Platform. [s.l.] : Apress, 2009. 370 s. ISBN 978-1-4302-2418-1.
5. JÜRGEN, Petri. NetBeans Platform 6.9 Developer. 32 Lincoln Road Olton Birmingham, B27 6PA, UK. : Packt Publishing Ltd., 2010. 288 s. ISBN 978-1-849511-76-6.
6. MYATT, Adam. Pro NetBeans™ IDE 6 : Rich Client Platform Edition. [s.l.] : Apress, 2008. 522 s. ISBN 978-1-4302-0439-8.
7. GROSE, Timothy J.; DONEY, Gary C.; BRODSKÝ, Stephen A. Mastering XMI : Java Programming with XMI, XML, and UML. [s.l.] : John Wiley & Sons, 2002. 480 s. ISBN 0471384291.
8. JGraph Ltd. JGraph : User Manual [online]. 35 Parracombe Way, Northampton NN3 3ND U.K. : [s.n.], 2009 [cit. 2011-04-17]. Dostupné z WWW: <http://www.jgraph.com/>.
9. Visual Library 2.0 - Documentation [online]. 2008 [cit. 2011-04-17]. NetBeans. Dostupné z WWW: <http://bits.netbeans.org/dev/javadoc/org-netbeans-api-visual/org/netbeans/api/visual/widget/doc-files/documentation.html>.
10. Aspiringcraftsman [online]. 2007 [cit. 2011-04-29]. Interactive Application Architecture Patterns. Dostupné z WWW: <<http://www.aspiringcraftsman.com/2007/08/25/interactive-application-architecture>/>.
11. Eclipsecon : javadoc [online]. 2010 [cit. 2011-03-21]. Eclipse modeling framework javadoc. Dostupné z WWW: <<http://download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/>>
12. Wikipedia [online]. 2010 [cit. 2011-03-21]. Meta-Object Facility. Dostupné z WWW: <<http://en.wikipedia.org/wiki/Meta-Object_Facility>>
13. NetBeans platform [online]. 2009 [cit. 2011-04-17]. File type. Dostupné z WWW: <http://platform.netbeans.org/tutorials/nbm-filetype.html>
14. JUnit.org : Resources for Test Driven Development [online]. 2011, 2011 [cit. 2011-05-07]. Dostupné z WWW: <http://www.junit.org>.

Příloha A

A. Instalační a uživatelská příručka

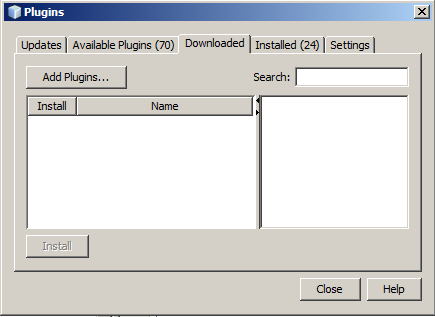
A.1 Instalace Metamodeláře

Metamodelář se skládá ze čtyř základních modulů. Každý je třeba nainstalovat do NetBeans IDE. Moduly se jmenují:

* *org-cvut-vrchlpet-MCore.nbm*
* *org-cvut-vrchlpet-MEditor.nbm*
* *org-cvut-vrchlpet-MFileType.nbm*
* *org-cvut-vrchlpet-MProject.nbm*

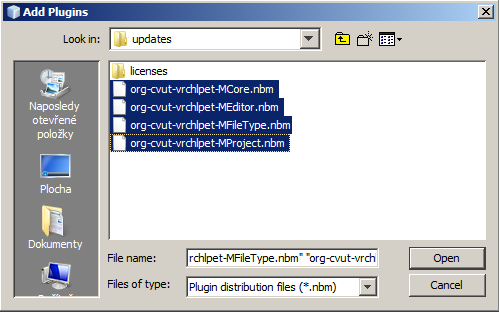
Postupně následujte tyto kroky:

1) Instalace probíhá přes instalační dialog. V hlavním menu zvolte ***Tools***->***Plugins***. Objeví se dialog, viz [obrázek 17](#obr17). Na záložce ***Downloaded*** zvolte ***Add Plugins…***



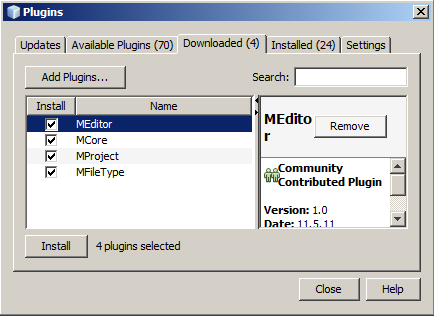
Obrázek - Dialog NetBeans IDE pro instalaci pluginů

2) Otevřete adresář s jednotlivými moduly, označte je a potvrďte tlačítkem ***Open*** viz [obrázek 18](#obr18).



Obrázek - Označte moduly, které chcete do NetBeans IDE nahrát a potvrďte *Open* tlačítkem

3) Moduly se objeví v dialogu pro instalování (viz [obrázek 19](#obr19)) a stisknutím tlačítka ***Install*** je nainstalujete.

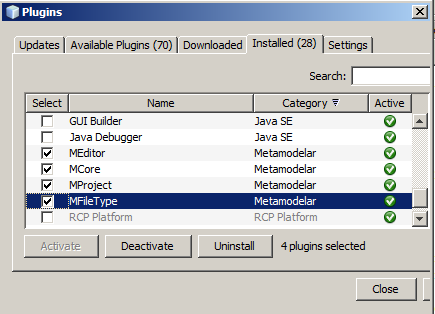


Obrázek - Zvolené moduly nainstalujete stisknutím tlačítka *Install*

4) Po stisknutí tlačítka ***Install*** budete vyzváni akceptovat licenci. Po akceptování licence je možné, že vás IDE upozorní, že moduly nejsou podepsané. S tím by neměl být problém, protože veškeré provedené testy nevykazovaly sebemenší poškození či nestabilitu IDE po instalaci Metamodeláře. Po tomto kroku je Metamodelář připraven k použití.

A.2 Odinstalování Metamodeláře

Odinstalování Metamodeláře probíhá podobným způsobem jako instalace. V hlavním menu zvolte ***Tools***->***Plugins***. Objeví se dialog, viz [obrázek 17](#obr17). Na záložce ***Installed*** označte moduly (pluginy), které chcete deaktivovat či odinstalovat (Metamodelář je v kategorii *Metamodelar*) a stiskněte tlačítko ***Deactivate***/***Uninstall*** (viz [obrázek 20](#obr20)). Po tomto kroku bude vyžadován restart IDE, který lze však odložit na později, takže je popřípadě možné uložit rozpracovanou práci. Dialog zavřete.

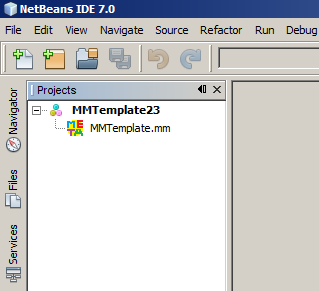


Obrázek - Odinstalace modulů/plaginů

A.3 Uživatelská příručka

A.3.1 Založení projektu

V hlavním menu zvolte ***File*** -> ***New project…*** V dialogovém okně *New Project* zvolte kategorii ***Metamodelar*** a v kolonce ***Projects***: zvolte ***Metamodelar project.*** Potvrďte tlačítkem ***Next.*** Do kolonky ***Project Name*** vyplňte jméno projektu a do kolonky ***Project Location*** zvolte umístění projektu. Potvrďte stisknutím tlačítka ***finish***. Ve správci projektů se objeví nový projekt, viz [obrázek 21](#obr21).



Obrázek - Nově založený projekt Metamodeláře

V nově vytvořeném projektu je vytvořen soubor *MMTemplate.mm*. Jedná se o prázdný meta-model. Pokud vám nevyhovuje jeho jméno, pak jej lze přes kontextové menu přejmenovat. Kontextové menu vyvolané nad projektem, v našem případě nad *MMTemplate23*, nabízí tyto možnosti:

* *New* – umožní vložit nové položky do projektu. Pokud vložíte jakoukoliv jinou položku než soubor meta-modelu, pak se sice položka vytvoří v adresáři projektu, ale nezobrazí se v projektovém správci.
* *Delete* – smaže projekt jak z IDE tak z místa, kde byl vytvořen.
* *Close* – zavře projekt. IDE projekt odstraní z projektového správce, ale ponechá ho v místě vytvoření projektu. Tzn. bude možné projekt znovu otevřít.

A.3.2 Práce se souborem meta-modelu

Soubor meta-modelu poznáte ve správci projektů snadno podle ikony s písmeny META. Nad meta-modelem jsou definované tyto akce v kontextovém menu:

* *Open* – otevře editor meta-modelu
* *Save* – uloží meta-model
* *Delete* – vymaže meta-model z projektu. Včetně ze složky, kde byl uložen.
* *Rename* – přejmenuje soubor meta-modelu (nikoliv meta-model samotný!)
* *Properties* – zobrazí základní informace o souboru meta-modelu

A.3.3 Tvorba meta-modelu

Definici meta-modelu vysvětlím pomocí názorného příkladu. V následujících řádcích popíší postup při tvorbě jednoduchého meta-modelu rodiny. Elementy meta-modelu budou:

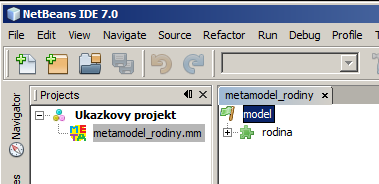
* *Rodina*
* *Člen rodiny*
* *Rodič*
* *Otec a matka*
* *Syn a dcera*

Relace meta-modelu budou:

* *Člen*
* *Potomek*
* *Nevlastní příbuzný*

Postup

Vytvoříme projekt (viz kapitola A.3.1). Přejmenujeme soubor s meta-modelem (viz kapitola A.3.2) na „*metamodel\_rodiny.mm*“ (koncovku není třeba zadávat, doplní se automaticky). Otevřeme editor poklepáním na soubor meta-modelu. Po otevření přejmenujeme meta-model samotný na „*rodina*“ a to vyvoláním kontextového menu nad meta-modelem (po vytvoření je jméno nastaveno na „*metamodel*“) a zvolením ***change namespace***. Podobným způsobem můžeme změnit popis (*description*). Stávající stav je zobrazen na [obrázku 22](#obr22).



Obrázek - Nově vytvořený meta-model rodiny

Jako první, definujeme relace. Po rozbalení uzlu *rodina* se objeví uzly *elements* a *relations*. Vyvoláme kontextové menu nad uzlem *relations* a zvolíme ***Add relation***. V dialogu vyplníme jméno relace a potvrdíme tlačítkem ***Add***. Tento postup provedeme pro všechny tři relace. Po rozbalení uzlu *relations* uvidíme naše nově vytvořené relace. V kontextovém menu vyvolaném jednotlivě nad každou relací zvolíme položku ***Property***. Objeví se dialog pro editaci relace. U jednotlivých relací nastavíme parametry takto:

* *Člen*: zaškrtneme *container*, nezaškrtneme *symmetric*
* *Potomek*: necháme beze změn
* *Nevlastní příbuzný*:zaškrtneme *container*, zaškrtneme *symmetric*

Nyní máme nadefinované relace. Přejdeme k definici elementů. Přidání elementu probíhá podobně jako je tomu u relace. Z kontextového menu vyvolaném nad uzlem *elements* zvolíme ***Add element***. Po vložení všech 7 elementů dialog zavřeme. Každý uzel *elementu* obsahuje uzly *attributes* a *references*.

Dalším krokem je vytvoření referencí mezi jednotlivými elementy. Reference mezi elementy se vytváří přes kontextové menu uzlu *references* nebo v dialogu vlastností u jednotlivých elementů. V dialogovém oknu pro přidání reference zvolíme relaci, kterou chceme použít a element, s kterým chceme v dané relací být. Vytvoříme tyto relace (je důležité dodržet, nad kterým elementem vytváříme referenci):

* *Rodina* – *člen rodiny*: relace - *člen*, horní hranice: -1, dolní hranice: 0
* *Rodič* – *syn*: relace – *potomek*, horní hranice: -1, dolní hranice: 0
* *Rodič – dcera*: relace – *potomek*, horní hranice: -1, dolní hranice: 0
* *Člen rodiny – člen rodiny*: relace: *nevlastní příbuzný*: horní hranice:-1, dolní hranice: 0

Pro úplné definování vztahů mezi elementy je zapotřebí dodefinovat reference na opačných koncích jednotlivých relací. Otevřete dialog vlastností reference nad těmito referencemi a nastavte:

* *Syn*: reference: *rodič* – dolní hranice: 1
* *Dcera* – stejně jako u *syna*
* *Člen rodiny*: reference: *rodina* – dolní hranice: 1, horní hranice:

V dalším kroku nadefinujeme dědičnou hierarchii našeho meta-modelu, aby výše definované relace dávaly smysl. Dědičnost se definuje ve vlastnostech elementu. Dialog vlastností se vyvolá z kontextového menu elementu položkou *properties*. Předka konkrétního elementu nastavíme pomocí rozbalovacího seznamu v rámečku *Super type* v dialogu vlastností. Dědičnosti nastavíme takto:

* *Rodič* dědí od *člen rodiny*
* *Otec* dědí od *rodič*
* *Matka* dědí od *rodič*
* *Syn* dědí od *člen rodiny*
* *Dcera* dědí od *člen rodiny*

Nyní zbývá pouze nastavit vlastnosti jednotlivých elementů. Vlastnosti lze přidat přes kontextové menu uzlu *attributes* u jednotlivých elementů. Vytvoříme tyto vlastnosti:

* Element *Rodina*:
  + Atribut: jmeno, dolní a horní hranice 1, *editable* true
    - Property: *String*, *name*: jméno rodiny
* Element *člen rodiny*:
  + Attribut: jmeno, dolní a horní hranice 1, *editable* true
    - Property: *String*, *name*: jméno člena
  + Attribut: věk, dolní a horní hranice 1, *editable* true
    - Property: *Integer*, *name*: věk člena

Další atributy nejsou důležité pro tento příklad. Důležité je si uvědomit, že díky dědění jsou tyto atributy propagovány do ostatních elementů meta-modelu. Na úrovni meta-modelu se to nijak neprojeví, ale na úrovni modelu to hraje významnou roli. V této fázi máme nadefinovaný meta-model rodiny.

Pokud budeme chtít tvořit modely na základě tohoto meta-modelu za použití grafické notace, pak můžeme nadefinovat visuální stránku jednotlivých elementů, popřípadě relací. Uzly *elementů* a *relací* mají v kontextovém menu položku ***editUI***. Tato položka vyvolá dialog pro editaci visuální stránky elementu/relace. U elementu lze nastavit obrázek, barva pozadí, ohraničení a popisky. U relace lze nastavit tvar šipky u zdroje a u cíle a popisky. Aktuální grafickou podobu elementu/relace lze zobrazit volbou ***show graphics*** z kontextového menu.

**Nezapomeňte uložit[[12]](#footnote-12) meta-model před zavřením NetBeans IDE!**

Příloha B

B. Obsah přiloženého CD

**| --** *readme.txt* Informace o struktuře CD a instalační pokyny.

**\ --** *src* Zdrojový kód aplikace včetně GIT repozitáře.

**\ --** *text*

**| --** *BP.pdf* Text práce.

**| --** *zadani.txt* Zadání práce.

**\ --** *Metamodelar* Zkompilované moduly určené pro instalaci do NetBeans IDE**.**

**\ --** *Ukazkovy projekt* Projekt meta-modelu rodiny vytvořeném podle návodu v příloze A.3.

1. OMG je konsorcium zaměřené na správu standardů pro distribuované objektově orientované systémy. OMG se zaměřuje pouze na specifikace standardů a neposkytuje jejich implementaci. Předtím, než je specifikace přijata za standard, musí navrhovatel garantovat, že se její implementace do roka objeví na trhu. Toto je pokus o prevenci přijmutí neimplementovaných a neimplementovatelných standardů. [↑](#footnote-ref-1)
2. MDE (Model-driven engineering) je metodologie softwarového vývoje, která se zaměřuje na vytváření a využívání doménových modelů (abstraktní reprezentace vědomostí a aktivit problémové domény – například bankovní systém), kterým dává přednost před výpočetními (algoritmickými) koncepty. MDE je zaměřeno na zvýšení produktivity maximalizací komptability mezi jednotlivými systémy a napomáhá při vylepšení komunikace mezi jednotlivými účastníky vývoje a týmu pracujícího na daném systému. V ideálním případě je možné z modelů, které jsou jádrem MDE, vygenerovat funkční software, který vyžaduje minimální zásah pro doladění detailů. [↑](#footnote-ref-2)
3. *Tool tip* je krátká informace, nejčastěji popis konkrétní komponenty v grafickém uživatelském rozhraní. *Tool tip* je často reprezentován bublinou s textem zobrazujícím popis, nápovědu či obecné informace o dané komponentě. [↑](#footnote-ref-3)
4. *NetBeans IDE* je vývojové prostředí postavené na platformě *NetBeans platform*. NB IDE je koncipována jako modulární aplikace, tj. její funkcionalita je určena moduly, které tvoří celou aplikaci. NB IDE poskytuje širokou škálu funkcí, mezi ně patří i podpora pluginů. [↑](#footnote-ref-4)
5. **ZIP** je populární, všeobecně rozšířený souborový formát pro kompresi a archivaci dat. ZIP soubor vytvořený kompresí obsahuje jeden či více komprimovaných souborů, což ve výsledku pomůže zredukovat velikost uložených dat. [↑](#footnote-ref-5)
6. **API** (**Application Programming Interface**) označuje v informatice rozhraní pro programování aplikací [↑](#footnote-ref-6)
7. **Swing** je knihovna uživatelských prvků na platformě Java pro ovládání počítače pomocí grafického rozhraní [↑](#footnote-ref-7)
8. Projektový správce je součástí NetBeans IDE. Při startu IDE se projdou všechny moduly, jestli nedefinují projektový typ. Pokud ano, pak se každý definovaný typ zaregistruje. Práce s projektovým typem je v režii pluginu, který daný typ definuje. Správce projektů pouze poskytuje rozhraní pro komunikaci s uživatelem. Obsah, kterým daný projektový typ správce zaplní je čistě na definici tohoto projektového typu. Stejně tak akce, které projektový typ definuje [↑](#footnote-ref-8)
9. **JUnit** je framework pro jednotkové testy psaný v programovacím jazyce Java. [↑](#footnote-ref-9)
10. ***Verifikace*** je ověření, že software správně implementoval specifické funkce. -> „Vytvořili jsme produkt správně?“ [↑](#footnote-ref-10)
11. ***Validace*** je ověření, že software odpovídá požadavkům zákazníka. - > „Vytvořili jsme správný produkt?“ [↑](#footnote-ref-11)
12. Meta-model rodiny vytvořený podle tohoto návodu je na přiloženém CD k tomuto textu. [↑](#footnote-ref-12)