Žilinská univerzita v Žiline
Fakulta riadenia a informatiky

Diplomová práca

Študijný program: Informačné systémy
Zameranie: Aplikovaná informatika

Tomáš Bača

Návrh a tvorba systému zásuvných modulov pre nástroj UML .FRI

Vedúci: Ing. Ján Janech

Reg. č.: 171/2008 jún 2009

Žilina

Anotačný list

Anotačný list diplomovej práce študenta v štúdijnom programe Informačné systémy na Žilinskej univerzite v Žiline v akademickom roku 2008.

1.	Meno a priezvisko	: *Tomáš Bača*
2.	Fakulta	: *Fakulta riadenia a informatiky*
3.	Názov práce	: *Návrh a tvorba systému zásuvných modulov pre nástroj UML .FRI*
4.	Počet strán	: *60*
5.	Počet obrázkov	: *8*
6.	Počet príloh	: *1*
7.	Počet titulov použitej literatúry	: *14*
8.	Kľúčové slová	: *zásuvný modul, UML .FRI, sieťový graf, kritická cesta*

Resumé

Diplomová práca analyzuje možnosti rozšírenia aplikácie UML .FRI o systém zásuvných modulov a jeden zo spôsobov implementuje. Obsahuje jeden zásuvný modul, hľadajúci kritickú cestu v sieťovom grafe, na demonštráciu možností tohto systému.

Summary

The diploma thesis analyses possible ways of extending UML .FRI application by plug-in system and implements one of them. A plug-in for finding critical path in network is included as an demonstration.

Poďakovanie

Ďakujem svojmu vedúcemu diplomovej práce Ing. Jánovi Janechovi za výdatnú pomoc a množstvo konzultácií.
Ďakujem svojim rodičom za pomoc pomoc pri oprave chýb v práci.
Ďakujem Ing. Ľubomírovi Sadloňovi za zapožičanie literatúry a konzultácie o sieťových grafoch.

Prehlásenie

Prehlasujem, že som diplomovú prácu spracoval samostatne a že som uviedol všetky použité pramene a literatúru, z ktorých som čerpal.

V Žiline dňa 19. 5. 2009 Podpis.....

Obsah

Zoznam obrázkov

Zoznam výpisov

Zoznam použitých označení

CASE	*Computer Aided Software engineering* – Softvérový nástroj na uľahčenie vývoja s využitím počítača.
CPU	*Central Processing Unit* – centrálna výpočtová jednotka počítača. V súčastnosti predstavuje jedno jadro procesora jedno CPU.
DSL	*Domain Specific Language* – jazyk závislý od domény. Konkrétne to znamená, že je možné upravovať metamodel a navrhnúť si vlastný na každú príležitosť.
DSM	*Domain Specific Modeler* – CASE nástroj na prácu s DSL.
DTD	*Data Type Definition* – Starší spôsob definície štruktúry XML.
ERA	*Entity Relationship Attribute* – Diagram na entitno-relačný návrh databázy.
GUI	*Graphical User Interface* – Grafické rozhranie, ktoré slúži na interakciu používateľa a programu.
GPL	*GNU General Public License* – Licencia pre slobodný softvér
HTTP	*HyperText Transfer Protocol* – Protokol, využívaný v počítačových sieťach na prenos predovšetkým hypertextových dokumentov. Principiálne umožňuje prenos ľubovoľnej informácie.
IP	*Internet Protocol* – protokol používaný na prenos dát cez siete s prepínaním paketov. Zabezpečuje prenos paketov postupne cez uzly v sieti.
RFC	*Request For Comments* – Textový dokument, v ktorom je popísaný návrh alebo aj finálna podoba internetového štandardu.
SIP	*Session Initiation Protocol* – Štandard využívaný na signalizáciu a vytváranie spojení (predovšetkým hlasové služby) cez Internet.
SMTP	*Simple Mail Transfer Protocol* – Protokol používaný na prenos elektronickej pošty cez Internet.
TCP	*Transmission Control Protocol* – protokol používaný na spoľahlivý prenos dát cez *IP* sieť. Vytvára spojenie medzi koncovými účastníkmi spojenia.

UDP — *User Datagram Protocol* – protokol používaný na prenos dát cez *IP* sieť. Vytvára spojenie medzi koncovými účastnínkmi spojenia, ale nezabezpečuje spoľahlivý prenos.

UML — *Unified Modeling Language* – vizuálny jazyk na modelovanie a komunikáciu o systéme pomocou diagramov a podporného textu [8]

URI — *Uniform Resource Identifier* – jednotný identifikátor zdroja. Je zložený z postupnosti znakov a slúži na identifikáciu zdroja na Internete.[13]

URL — *Uniform Resource Locator* – jednotný lokátor zdroja. Je to typ *URI* a obsahuje informáciu kde, a akým mechanizmom je zdroj na Internete prístupný.[14]

XML — *eXtensible Markup Language* – v preklade rozšíriteľný značkovací jazyk, ktorý bol vyvinutý a štandardizovaný konzorciom W3C (World Wide Web Consortium) ako pokračovanie jazyka SGML a HTML. Umožňuje jednoduché vytváranie konkrétnych značkovacích jazykov na rôzne účely a široké spektrum rôznych typov údajov. [9]

XSD — *XML Schema definition* – Novší a kvalitnejší popis štruktúry XML v porovnaní s DTD.

Kapitola 1

Úvod

S projektom UML .FRI som sa prvýkrát stretol pred dvoma rokmi, keď som pracoval na svojej bakalárskej práci. V tom čase ma najviac zaujal fakt, že aplikácia je vyvíjaná programovacím jazykom Python, s ktorým som vtedy experimentoval. Jazyk sa mi páčil pre svoju špecifickú syntax a prenosnosť medzi platformami. Rozhodol som sa, že sa k projektu počas inžinierskeho štúdia pripojím. Dva roky prešli a ja v rámci tohto projektu obhajujem diplomovú prácu.

Aplikácia UML .FRI je vizuálny editor diagramov rôznych typov, aké bežne používajú študenti a pracovníci v oblasti IT. Vyvíjajú ho študenti Fakulty Riadenia a Informatiky Žilinskej Univerzity v rámci svojej projektovej výučby. Nástroj je vydaný pod licenciou GNU/GPL, to znamená, že je voľne dostupný. Radíme ho medzi tzv. *DSM*[1] *CASE*[2] nástroje a preto je preň charakteristické, že sám v sebe neobsahuje gramatiku nejakého špecifického modelovacieho jazyka. Takáto gramatika je dodaná v samostatnom balíčku, ktorý sa nazýva metamodel[3].

Keď sa pozrieme na profesionálne modelovacie nástroje, okrem modelovania samotného, ponúkajú aj iné, pokročilejšie funkcie. Hovoríme však o funkciách, ktoré sú špecifické pre jeden konkrétny modelovací jazyk. Je to spôsobené predovšetkým faktom, že sú zamerané práve na ten jeden jazyk. Ale skúsení používatelia tieto funkcie vyžadujú. Príkladom môže byť generovanie SQL skriptov z ERA diagramu.

Tím, starajúci sa o vývoj UML .FRI, považuje podobné funkcie za potrebné a rád by ich do svojho nástroja začlenil. Lenže takéto funkcie, práve

[1] *Domain Specific Modeler* – CASE nástroj na prácu s DSL.

[2] *Computer Aided Software engineering* – Softvérový nástroj na uľahčenie vývoja s využitím počítača.

[3] Model popisujúci model. Aké typy entít dokáže používať, aké vzťahy medzi nimi dokáže zaviesť atď...

pre svoju špecifickosť a previazanosť s konkrétnym metamodelom, musia byť k nástroju dodávané spoločne s metamodelmi vo forme zásuvných modulov. Úlohou mojej diplomovej práce je preskúmanie možnosti rozširovania aplikácie práve týmto spôsobom a implementácia systému, ktorý by to aj umožnil.

V Kapitole 2 podrobne rozoberám, aké požiadavky boli postupne kladené na diplomovú prácu. Špecifikácia sa týka predovšetkým implementačnej zložky, pretože je predpoklad, že v budúcnosti bude aplikácia značne rozširovaná práve cez zásuvné moduly.

V tretej kapitole predstavujem aplikáciu UML .FRI, jej hlavnú vývojovú vetvu, v ktorej ešte nie je moja diplomová práca začlenená. Opisujem architektúru aplikácie, vývojové nástroje a technológie, ktoré sú použité pri tvorbe tohto nástroja.

V nasledujúcej kapitole sa venujem analýze problematiky zásuvných modulov. V prvej časti predstavujem všeobecný pohľad na zásuvné moduly a spôsoby, ako s nimi môžu aplikácie spolupracovať. V druhej časti premietam tieto všeobecné informácie do kontextu UML .FRI a vyberám konkrétne technológie pre implementáciu.

Piata kapitola popisuje samotnú implementáciu systému zásuvných modulov. Vymenúvam množinu príkazov, ktoré môže zásuvný modul aplikácii poslať a spôsob, akým aplikácia odpovedá. Opisujem, ako tento mechanizmus funguje a ako sú naprogramované jednotlivé prvky tohto systému. Posledná časť kapitoly vysvetľuje, ako aplikácia jednotlivé zásuvné moduly štartuje.

Kapitola 6 dokumentuje jeden konkrétny zásuvný modul, na ktorom som chcel deklarovať, že systém zásuvných modulov skutočne funguje. Ukážkový zásuvný modul obsahuje algoritmus hľadania kritickej cesty v sieťovom grafe.

Kapitola 2

Ciele diplomovej práce

Aplikácia UML .FRI je vizuálny nástroj na modelovanie diagramov rôznych typov. Radí sa medzi *DSM*. Nástroj samotný neobsahuje ani jeden metamodel (gramatiku modelovacieho jazyka). Metamodely sú dodávané k aplikácii ako samostatné balíky. Aplikácia prečíta jeden konkrétny metamodel až v momente, keď otvára projekt, v ktorom sa tento metamodel používa.

Pretože aplikácia nie je napísaná špecificky pre nejaký konkrétny modelovací jazyk, neobsahuje ani funkcionalitu špecifickú pre modely konkrétnych typov. (Neumožňuje napríklad generovať zdrojový kód z UML diagramov.) Takáto funkcionalita je však veľmi nápomocná pri mnohých typoch úloh a skúsenejší používatelia takéto funkcie od svojich nástrojov vyžadujú.

Aby mohlo UML .FRI byť úspešné, potrebuje preto obsahovať aj funkcie určené špeciálne pre konkrétne typy modelov. Tieto však musia byť dodávané ako samostatné balíčky, podobne ako metamodely. Takéto balíčky budú k aplikácii pripájané vo forme zásuvných modulov a teda môžu byť distribuované nezávisle na aplikácii.

Systém pre podporu zásuvných modulov však aplikácia doteraz neobsahuje a nebolo naň myslené ani v pôvodnom návrhu. Ciele tejto diplomovej práce preto sú:

- Analýza možností rozšírenia funkcionality nástroja pomocou zásuvných modulov.
- Analýza možností implementácie systému zásuvných modulov v programovacom jazyky Python.
- Návrh a implementácia systému zásuvných modulov pre nástroj UML .FRI. Na systém zásuvných modulov sú kladené nasledujúce podmienky:
  - Systém zásuvných modulov musí zachovávať vnútornú štruktúru aplikácie, tzn. dôsledné oddelenie vrstiev aplikácie.

  - Zásuvné moduly nesmú mať priamy prístup k vnútorným objektom aplikácie, pristupovať k nim smú iba sprostredkovane, cez presne definované programové rozhranie.
  - Chyba, ktorá vznikne v zásuvnom module, sa nesmie preniesť do jadra aplikácie – nesmie ohroziť stabilitu aplikácie.
  - Tak ako aplikácia, tak aj systém zásuvných modulov má byť prenosný na podporované počítačové platformy.
  - Súčasťou systému zásuvných modulov bude knižnica, určená tvorcom zásuvných modulov v jazyku Python.
  - Systém musí byť navrhnutý tak, aby ho bolo možné v budúcnosti jednoducho rozšíriť.
- Implementácia ukážkového zásuvného modulu v jazyku Python. Zásuvný modul bude hľadať kritickú cestu v sieťovom grafe.

Kapitola 3

Súčasný stav UML .FRI

V tejto kapitole predstavím hlavnú vývojovú vetvu aplikácie UML .FRI, v ktorej ešte moja diplomová práca nie je začlenená.

3.1 UML .FRI – hlavná vývojová vetva

Aplikácia UML .FRI je vizuálny editor diagramov rôznych typov, aké bežne používajú študenti a pracovníci v oblasti IT. Vyvíjajú ho študenti Fakulty Riadenia a Informatiky Žilinskej Univerzity v rámci svojej projektovej výučby. Nástroj je vydaný pod licenciou GNU/*GPL*[1], to znamená, že je voľne dostupný.

Nástroj vznikol ako školský projekt v akademickom roku 2005 na Fakulte Riadenia a Informatiky Žilinskej Univerzity. Jeho vývoj prebieha v rámci predmetov Projekt 1, 2, 3. Vytvárajú ho teda predovšetkým študenti inžinierskeho štúdia. V súčastnosti už projekt možno nazývať generačným, pretože sa naň každý rok prihlasuje niekoľko nových študentov, aby preberali prácu po svojich starších kolegoch. Spolu sa takto na projekte podieľalo už 14 študentov.

Nástroj je od začiatku vyvíjaný tak, aby s ním bolo možné pracovať na všetkých rozšírených počítačových platformách. To bol jeden z dôvodov pre výber programovacieho jazyk Python. Na vytvorenie GUI je použitá knižnica *GTK+* a rozloženie prvkov *GUI*[2] je vytvorené v nástroji *Glade*. Kresliaca plocha je zobrazovaná s pomocou knižnice *Cairo*. Metamodel, konfigurácia programu a projekt sú uložené vo formáte *XML*[3].

[1] *GNU General Public License* – Licencia pre slobodný softvér

[2] *Graphical User Interface* – Grafické rozhranie, ktoré slúži na interakciu používateľa a programu.

[3] *eXtensible Markup Language* – v preklade rozšíriteľný značkovací jazyk, ktorý bol vy-

Výsledný produkt preto rovnako dobre funguje na operačných systémoch Linux, Windows, MacOS, BSD, Solaris. Predpokladáme, že by nemali byť prekážky na jeho spúšťanie ani na ďalších platformách, kde sú spomínané nástroje k dispozícii. A čo je hlavné, prenos medzi platformami je možný bez špeciálnych zásahov do zdrojových kódov alebo častí zdrojových kódov napísaných špeciálne pre jednu platformu.

Zatiaľ bola posledná verzia programu vydaná v marci 2009. Označená je ako 1.0-beta20090309. Stále ešte nebola vydaná finálna verzia aplikácie.

3.2 Implementačné prostriedky

3.2.1 Programovací jazyk Python

Python je všeobecne využiteľný vysoko úrovňový programovací jazyk. Navrhnutý bol tak, aby pri práci s ním programátor maximalizoval produktivitu a čitateľnosť zdrojových kódov. Umožňuje používať pri programovaní viaceré paradigmy - objektovo orientovaný, imperatívny, štruktúrovaný a funkcionálny prístup. V jazyku je implementovaná automatická správa pamäte.

Výhodou jazyka Python je prístupnosť na mnohých platformách. To znamená, že jeden zdrojový kód možno používať bez zásahov na väčšine platforiem. Podmienkou je, že sa programátor vyhne použitiu vlastností špecifických iba pre jednu platformu. (Napríklad napevno použité znaky spätnej lomky pri určení cesty k súboru.) Medzi podporované platformy patrí Linux, Windows, MacOS, BSD, Solaris...

V projekte používame de facto štandard jazyka Python, tzv. *CPython*. Je to implementácia vykonávacieho jadra v jazyku C. Pre väčšinu operačných systémov je minimálna povolená verzia jazyka 2.5, na operačnom systéme Windows je minimálna povolená verzia jazyka 2.6.

3.2.2 GTK+

GTK+ je na platforme nezávislý súbor nástrojov pre tvorbu grafických používateľských rozhraní (widget toolkit), resp. tvorbu *GUI*. Pôvodná implementácia bola napísaná v C jazyku, ale existujú knižnice, ktoré umožňujú používať GTK+ aj v jazykoch ako napríklad C++, Python, Ruby, Perl. Vydaná je pod slobodnou *GNU Less General Public License (LGPL)*, ktorá umožňuje vývoj slobodného, ale aj proprietárneho softvéru.

vinutý a štandardizovaný konzorciom W3C (World Wide Web Consortium) ako pokračovanie jazyka SGML a HTML. Umožňuje jednoduché vytváranie konkrétnych značkovacích jazykov na rôzne účely a široké spektrum rôznych typov údajov. [9]

V súčastnosti je GTK+ založená predovšetkým na štyroch knižniciach, taktiež vyvíjaných GTK projektom - Glib, Pango, Cairo a ATK.

- *Glib* umožňuje použitie vlákien, prináša programovú slučku čakajúcu na udalosti *event loop*, dynamické načítavanie a objektový systém. Prináša rôzne utility na prácu so súbormi, reťazcami, atď.
- *Pango* je knižnica určená na kreslenie textu s dôrazom na internacionalizáciu a tvorí hlavnú zložku pre prácu s textom a typmi písma.
- *Cairo* je 2D grafická knižnica bližšie popísaná v kapitole 3.2.4.
- pri podpore *ATK* alebo Accessibility Toolkit rozhraní môže aplikácia používať nástroje ako alternatívne vstupné zariadenia.

Pôvodne bola GTK+ vyvinutá pre GNU Image Manipulation Program alebo GIMP, odtiaľ pochádza aj názov knižnice - GTK, teda The GIMP ToolKit. V súčastnosti je knižnica využívaná v projektoch ako GNOME alebo XFCE, prípadne v aplikáciách ako Wireshark, Gnumeric.

UML. FRI využíva knižnicu PyGTK, ktorá predstavuje premostenie[4] jazyka Python do GTK+. Pôvodným autorom je James Henstridge - jeden z popredných vývojárov GNOME. PyGTK tak spája výhody jazyka Python a knižnice GTK+. Pre samotný návrh GUI využívame v aplikácii nástroj *Glade*.

3.2.3 Glade

Glade je nástroj, ktorý uľahčuje návrh a implementáciu používateľských rozhraní pre GTK+. GUI navrhnuté v Glade je uložené ako *XML* súbor. Za použitia *libglade* knižnice je možné toto používateľské rozhranie, podľa potreby dynamicky, načítať aplikáciou. Vytvorené *XML* súbory je možné využiť v programovacích jazykoch, pre ktoré existujú premostenia, napríklad Python, Java, Perl, Ruby.

3.2.4 Cairo

Cairo je 2D grafická knižnica s podporou rôznych výstupných formátov. V súčastnosti podporuje X Window Systém, Quartz, Win32, PostScript, PDF, SVG výstup. Cairo je navrhnuté tak, aby produkovalo konzistentný výstup

[4]Po anglicky *binding* alebo *wrapper*. Mapuje volania jedného jazyka na volania knižnice napísanej v inom jazyku.

za podpory hardvérovej akcelerácie, ak je možná - napr. pomocou *X Render Extension*.

Pôvodne je napísaná v jazyku C, ale existujú premostenia pre jazyky ako napríklad Python, Perl, Ruby. Cairo je slobodný softvér licencovaný pod GNU Lesser General Public License (GPL) a Mozilla Public License (MPL).

UML .FRI používa Cairo knižnicu pomocou jej premostenia do jazyka Python - PyCairo. Je napísaná tak, aby pokiaľ možno, čo najvernejšie kopíroval svoju C predlohu. V UML. FRI využívame PyCairo v dvoch dôležitých oblastiach:

- Kreslenie diagramov na kresliacu plochu. Tu využívame schopnosti ako vyhladzovanie hrán alebo možnosť približovať a odďaľovať vykreslený diagram (zoom).
- Exportovanie diagramov do formátov *Portable Document Format (PDF)*, *Portable Network Graphics (PNG)*, *PostScript (PS)*, *Scalable Vector Graphics (SVG)*

3.2.5 *XML*

XML (eXtensible Markup Language, po slovensky rozšíriteľný značkovací jazyk) je všeobecný značkovací jazyk, ktorý bol vyvinutý a štandardizovaný konzorciom W3C. Špecifikácia *XML* konzorcia W3C je zadarmo prístupná všetkým. Každý tak môže bez problémov do svojich aplikácii implementovať podporu *XML*.

XML umožňuje jednoduché vytváranie konkrétnych značkovacích jazykov pre rôzne účely a široké spektrum rôznych typov dát. Jazyk je predovšetkým určený pre výmenu dát medzi aplikáciami a publikovanie dokumentov. Umožňuje popísať štruktúru dokumentu z hľadiska vecného obsahu jednotlivých častí a sám sa nezaoberá vzhľadom dokumentu alebo jeho častí.

XML dokument je text, ktorý je implicitne kódovaný v *Unicode* (utf-8). Efektivita *XML* je silne závislá na štruktúre, obsahu a integrite. Aby bol *XML* dokument považovaný za správne štruktúrovaný (well formed), musí spĺňať nasledovné vlastnosti:

- Musí mať práve jeden koreňový (root) element.
- Neprázdne elementy musia byť ohraničené štartovacou a ukončovacou značkou.
- Prázdne elementy môžu byť označené značkou „prázdny element".

- Všetky hodnoty atribútov musia byť uzavreté buď v jednoduchých alebo dvojitých úvodzovkách. Opačný pár úvodzoviek môže byť použitý vo vnútri hodnôt atribútu.

- Elementy môžu byť vnorené, ale nemôžu sa prekrývať.

XML neobsahuje preddefinované značky (tagy), preto je potrebné definovať si vlastné, ktoré budú v dokumente použité. Tieto značky možno definovať pomocou *DTD*[5] súboru. Potom môžeme kontrolovať, či daný dokument zodpovedá tejto definícii. O kontrolu sa stará prekladač (parser). Ten dokáže na základe *DTD* zistiť chyby v dátach v *XML* dokumente.

DTD nie je jediný definičný jazyk pre *XML*. Nedokáže definovať a kontrolovať typy dát a typy hodnôt atribútov. Z toho dôvodu boli vytvorené konzorciom W3C tzv. *XML* schémy. *XSD*[6] (XML Schema Definition) je alternatívou k popisu *XML*.

XML schémy:

- definujú miesta v dokumente, kde sa môžu vyskytovať rôzne elementy,

- definujú atribúty,

- definujú, ktoré elementy sú potomkami iných elementov,

- definujú poradie elementov,

- definujú kardinalitu elementov,

- definujú, či môže byť element prázdny, alebo musí obsahovať text,

- definujú dátové typu elementov a ich atribútov,

- definujú štandardné hodnoty elementov a atribútov.

V aplikácii UML .FRI sa *XML* využíva na viacerých miestach. V tomto formáte je zapísaný konfiguračný súbor aplikácie. Sadou *XML* súborov sú tvorené metamodely. Predloha pre nový projekt aj projekt samotný je uložený v *XML* formáte, ktorý je následne komprimovaný do formátu ZIP.

Ukážku *XML* dokumentu si môžete všimnúť na výpise 3.1. Obsahuje zápis poštovej adresy vo Veľkej Británií. Na výpise 3.2 je schéma, ktorá definuje formát takejto adresy.

[5] *Data Type Definition* – Starší spôsob definície štruktúry XML.
[6] *XML Schema definition* – Novší a kvalitnejší popis štruktúry XML v porovnaní s DTD.

```
<?xml version="1.0" encoding="utf-8"?>
<Address xmlns:xsi="http://www.w3.org/2001/XMLSchema
   -instance"
         xsi:noNamespaceSchemaLocation="
            SimpleAddress.xsd">
  <Recipient>Mr. Walter C. Brown</Recipient>
  <House>49</House>
  <Street>Featherstone Street</Street>
  <Town>LONDON</Town>
  <PostCode>EC1Y 8SY</PostCode>
  <Country>UK</Country>
</Address>
```

Výpis 3.1: *Ukážka XML dokumentu – poštová adresa*

3.3 Metamodel

Zaujímavou vlastnosťou aplikácie je, že sama o sebe neobsahuje definíciu, ako má ten ktorý diagram vyzerať. Program je v skutočnosti univerzálny nástroj, ktorý dokáže vykresliť a editovať diagramy, zložené z ľubovoľných elementov a spojení. Dokáže ich ukladať do súboru a opätovne ich načítať, exportovať ich do niekoľkých grafických formátov, alebo vytlačiť na tlačiarni. Niekoľko diagramov môže byť spolu uložených v jednom projekte. Spolu tak vytvárajú model systému, ktorý chcel používateľ popísať.

Aby aplikácia dokázala vôbec s nejakým projektom pracovať, potrebuje k nemu získať gramatiku modelovacieho jazyka, čiže definíciu pravidiel, elementov a spojení. Celú túto definíciu spoločne nazývame metamodel. Vzťah metamodelu a modelu možno prirovnať aj ku vzťahu gramatiky programovacieho jazyka a zápisu programu v programovacom jazyku. Jeden metamodel môže obsahovať definíciu väčšieho množstva prvkov a tiež niekoľkých diagramov. Každý projekt v sebe nesie informáciu, na základe ktorého metamodelu bol vytvorený. Identifikuje ho na základe dvojice *URI*[7] a verzie metamodelu.

Metamodel je tvorený sadou *XML* súborov v definovanej adresárovej štruktúre. Jeden súbor popisuje jeden prvok, z ktorého sa model môže skladať.

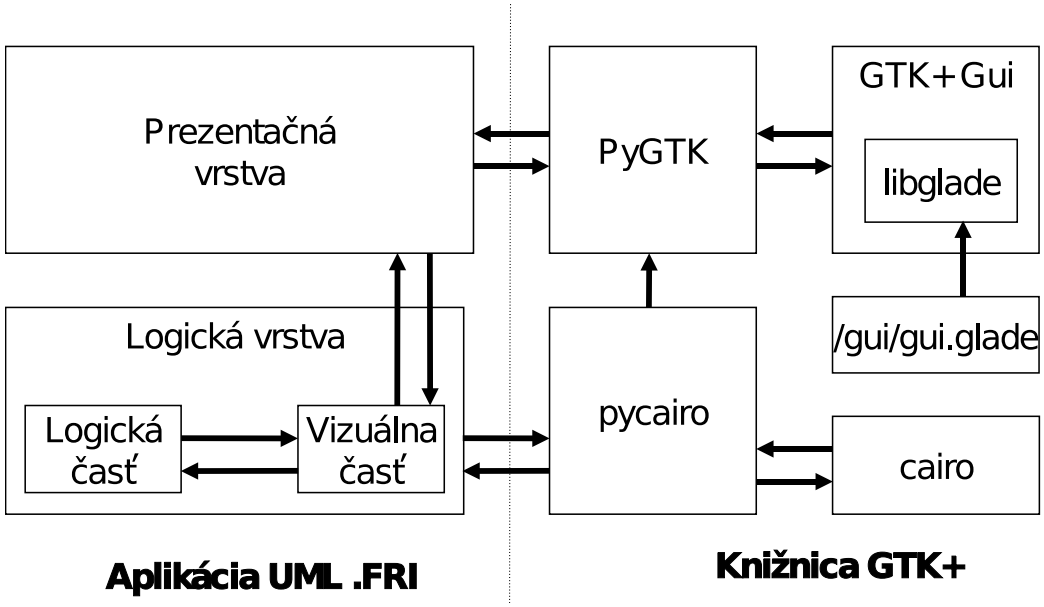
[7] *Uniform Resource Identifier* – jednotný identifikátor zdroja. Je zložený z postupnosti znakov a slúži na identifikáciu zdroja na Internete.[13]

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema elementFormDefault="qualified" xmlns:xs="
   http://www.w3.org/2001/XMLSchema">
  <xs:element name="Address">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Recipient" type="xs:string
           " />
        <xs:element name="House" type="xs:string" />
        <xs:element name="Street" type="xs:string" /
           >
        <xs:element name="Town" type="xs:string" />
        <xs:element name="County" type="xs:string"
           minOccurs="0" />
        <xs:element name="PostCode" type="xs:string"
            />
        <xs:element name="Country">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="FR" />
              <xs:enumeration value="DE" />
              <xs:enumeration value="ES" />
              <xs:enumeration value="UK" />
              <xs:enumeration value="US" />
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Výpis 3.2: *Príklad XML Schémy – schéma pre poštovú adresu*

3.4 Architektúra aplikácie

Aplikácia UML .FRI bola navrhnutá a implementovaná tak, aby mala jednotlivé podsystémy navzájom oddelené a nezávislé. Dala by sa rozdeliť na tri základné vrstvy, ale častejšie býva popísaná ako hierarchická.



Obr. 3.1: *Architektúra systému UML .FRI*

Na ľavej strane schémy sú zaznačené podsystémy aplikácie. Na pravej strane sú naviazané na knižnice, ktoré aplikácia používa na zobrazovanie *GUI* a kresliacej plochy (viď. kapitola 3.2).

3.4.1 Prezentačná vrstva

Prezentačná vrstva slúži na komunikáciu s používateľom. Obsahuje logiku obsluhy jednotlivých grafických komponentov a zachytáva udalosti, ktoré vyvolá používateľ svojimi akciami. Je naprogramovaná špecificky pre použitie s GTK+.

3.4.2 Logická vrstva – logická časť

Logická vrstva je rozdelená na dve časti. Logická časť logickej vrstvy má niekoľko funkcií:

- priamy prístup k súboru, v ktorom je uložený projekt,
- pri otváraní projektu prečíta a interpretuje metamodel,
- udržiava strom projektu – štruktúru, v ktorej sú uložené jednotlivé elementy,
- pracuje s hodnotami atribútov elementov a spojení. (Atribútom v tomto prípade môže byť názov triedy v prípade UML modelu tried.)

Logická časť je nezávislá na implementácii zvyšku aplikácie.

3.4.3 Logická vrstva – vizuálna časť

Vizuálna časť logickej vrstvy pri svojej činnosti vychádza z informácií, ktoré získava od logickej časti. Zároveň jej prezentačná vrstva posiela správy o udalostiach, ktoré vyvolal používateľ svojou činnosťou na kresliacom plátne.

Hlavná činnosť vizuálnej časti je práca s diagramami. To znamená, že udržiava informácie o tom, ktoré elementy a spojenia majú byť zobrazené na danom diagrame. Dôležité je si uvedomiť, že jeden element môže byť zobrazený na niekoľkých diagramoch súčasne. Na každom z týchto diagramov môže mať inú pozíciu a veľkosť. To isté platí o spojeniach medzi elementmi.

Vizuálna časť má za úlohu daný diagram aj nakresliť. Toto kreslenie je však nezávislé na type použitého kresliaceho plátna. Kresliace plátno vyberie prezentačná vrstva na základe toho, či má byť diagram vykreslený na obrazovku, alebo má byť exportovaný ako obrázok do súboru, alebo odoslaný na tlačiareň.

Kapitola 4

Analýza

Zásuvný modul (anglicky „plugin", „plug-in", „addon", „add-on", „add-in", . . .) je počítačový program, ktorý spolupracuje s hlavnou (hostiteľskou) aplikáciou. Poskytuje tým konkrétnu a obyčajne veľmi špecifickú funkciu „na požiadanie používateľa". Aplikácia samotná dokáže pracovať aj bez zásuvného modulu, ale zásuvný modul býva závislý na aplikácii – jej programovom rozhraní. [11]

4.1 Zásuvný modul – všeobecný pohľad

V tejto časti opisujem zásuvné moduly zo všeobecného pohľadu. Oboznamujem s dôvodmi, prečo sa zásuvné moduly v niektorých aplikáciách nepoužívajú a naopak, prečo iné aplikácie zásuvné moduly podporujú. Taktiež uvádzam aj niektoré konkrétne prípady použitia zásuvných modulov.

4.1.1 Monolitické aplikácie

Veľká skupina aplikácií je vyvíjaná monoliticky. To znamená, že výsledok funguje dopredu určeným spôsobom. Môže sa skladať z viacerých knižníc, alebo vykonateľných súborov. Môže fungovať v distribuovanom prostredí – na viacerých uzloch v sieti, ale i tak má dopredu určenú množinu svojich funkcií.

Takýto prístup je zvolený najmä v prípade, ak chce jeden vývojový tím, prípadne úzko spolupracujúce tímy, kontrolovať celý projekt. V prípade, že vznikne potreba rozšíriť funkcionalitu aplikácie, upravia aplikáciu samotnú a vydajú novú verziu.

Niekto by mohol namietať, že vo svete je veľký počet aplikácií s otvorenými zdrojovými kódmi. O týchto skutočne nemožno vyhlásiť, že na ich

vývoji pracuje iba jeden vývojový tím. Mnohokrát k nim totiž prispieva aj „komunita" programátorov, ktorí si s hlavným tímom vymenia iba niekoľko správ.

V oboch spomínaných prípadoch však platí, že všetci zúčastnení programátori musia poznať vnútornú štruktúru aplikácie, alebo aspoň tú jej časť, na ktorej pracujú. Pri zásahu do zdrojového musia zvážiť všetky dôsledky, ktoré to pre aplikáciu bude mať. Taktiež na túto zmenu musia upozorniť dotknutých spolupracovníkov.

Opäť možno namietať, že ak má aplikácia vhodne navrhnutú štruktúru, jednotlivé časti správne oddelené a rozhrania medzi nimi dôsledne definované, tak sa zmena jednej časti neprenesie do inej časti. Toto je samozrejme pravda, ale takto to funguje v čase, keď sa aplikácia ladí, alebo pribúda iba málo nových a hlavne izolovaných funkcií. Ale podstatou novej funkcie obyčajne je, že prinesie zmenu, ktorú je cítiť a preto sa zákonite musí prejaviť vo viacerých častiach programu.

Väčšina vývojárov to však berie ako samozrejmosť. Všetci vedia, že takto je to normálne, že pokrok vyžaduje zmenu a inak to byť nemôže. Alebo, keby mohlo, bolo by to zbytočne nákladné. Alebo by to otvorilo cestu do aplikácie niekomu cudziemu, a to by znamenalo riziko. Jednoducho nežiadúca situácia.

4.1.2 Zásuvný modul ako spôsob rozšírenia

V predošlej časti som ukázal klasický prístup k vývoju aplikácie. Uviedol som niekoľko dôvodov, prečo je mnoho aplikácií vyvíjaných práve monoliticky. Najdôležitejším z nich je však asi ten, že prakticky všetky projekty začínajú práve takto. Systém podpory zásuvných modulov totiž vyžaduje prácu programátorov navyše a dôvody, prečo by aplikácia mohla alebo mala byť rozširovaná aj takto, sa prejavia až neskôr, keď trošku narastie a začne sa pýtať do sveta.

Jeden problém monolitickej aplikácie je jej sústavný rast a nabaľovanie nových funkcií. S tým sa môžu stretnúť všetky projekty, bez ohľadu na to, či sú vyvíjané uzatvorene alebo otvorene. Ako aplikácia naberá nové funkcie, zväčšuje svoje nároky na systémové prostriedky. Čím má aplikácia viac používateľov, tým viac požiadaviek na ňu kladú. Zároveň s tým sa však zmenšuje podiel funkcií, ktoré jednotliví používatelia reálne používajú. Ak takéto rozširovanie presiahne únosnú mieru, výsledný softvérový produkt začne spadať do kategórie tzv. *elephantware*[1].

[1]softvér, ktorý spotrebúva príliš veľa systémových prostriedkov, pretože obsahuje veľké množstvo funkcií. Väčšina týchto funkcií nesúvisí priamo s pôvodnou úlohou softvéru.

Ponúka sa riešenie, aby aplikácia pracovala s dynamicky pripojiteľnými modulmi, ktoré obsahujú určité balíky funkcií. Tieto sa budú pripájať iba v prípade, že ich používateľ potrebuje. Aplikácia tak môže spotrebúvať menšie množstvo systémových prostriedkov bez toho, aby prišla o časť funkcionality.

Na správu takýchto balíčkov bude aplikácia potrebovať jednotné rozhranie, ktoré sa nebude príliš často meniť. Alebo, ak by sa malo dôjsť k zmene, tak pribudne nová funkcia, ale staré funkcie zostanú zachované. To preto, aby už existujúce balíčky mohli fungovať naďalej.

Iný problém, s ktorým sa stretnú skôr otvorené projekty, je rozširovanie aplikácie nezávislými programátormi. Títo si často chcú rozšíriť funkcionalitu podľa vlastných potrieb. Nové funkcie znamenajú zásah do rozhraní medzi jednotlivými časťami aplikácie. Ak takúto zmenu spraví nezávislý programátor, môže zmenu zaslať hlavnému vývojovému tímu (správcovi zdrojových kódov). Zmena však môže byť odmietnutá a keď je následne na to vydaná nová verzia programu, používateľova úprava bude pravdepodobne s novou verziou nekompatibilná.

Aj v takomto prípade budú dynamicky pripojiteľné moduly riešením. Ak aplikácia poskytne nezávislému používateľovi – programátorovi dostatočne stabilné programové rozhranie, môže si pridať vlastnú funkcionalitu. Túto bude môcť používať aj s novšou verziou aplikácie a zároveň nebude musieť žiadať hlavný vývojový tím o začlenenie vlastných úprav do aplikácie.

4.1.3 Spôsoby pripojenia zásuvného modulu k aplikácii

Z definície zásuvného modulu v úvode tejto kapitoly vyplýva, že zásuvný modul je samostatná programová jednotka. Do istej miery sa to podobá na knižnicu. Skôr však, ako prejdem k spôsobom realizácie, rozšírim spomínanú definíciu o predpoklady, ktoré o zásuvných moduloch treba vo všeobecnosti mať.

- Zásuvný modul vytvárame pre nejakú konkrétnu úlohu – všeobecná úloha je obyčajne začlenená do samotnej aplikácie.
- Bol vytvorený iným programátorom než aplikácia – zásuvný modul môže vytvoriť ktokoľvek.
- Vývoj zásuvného modulu prebieha čiastočne oddelene od vývoja samotnej aplikácie – vydanie novej verzie aplikácie neznamená vydanie novej verzie zásuvného modulu ani naopak.
- Zásuvných modulov je viac a sú od rôznych autorov – každý z týchto autorov sleduje svoj vlastný cieľ a málokedy sa stará o „tých druhých".

Zásuvný modul možno pripojiť k aplikácii principiálne tromi základnými spôsobmi s rôznym stupňom oddelenia modulu od aplikácie. Modul možno pripojiť ako knižnicu, ktorej kód bude spúšťaný v spoločnom adresnom priestore a programovom vlákne ako hlavný program. Možno ho mať pripojený v adresnom priestore, ale jeho vykonávanie prebieha v samostatnom programovom vlákne. Alebo môže byť zásuvný modul spustený ako samostatný proces.

Zásuvný modul vykonávaný v spoločnom programovom vlákne je najtesnejšie pripojený k aplikácii. Zásuvný modul zdieľa spoločný adresný priestor s aplikáciou, čím získava priamy prístup k objektom a stavovým premenným aplikácie. Dokáže tak veľmi rýchlo a efektívne komunikovať so zvyškom aplikácie a prípadne s ďalšími zásuvnými modulmi.

Ak je aplikácia jednovláknová – čo je asi hlavný dôvod k výberu tohto riešenia – neobjavia sa v nej problémy s prístupom k zdieľaným prostriedkom. Na akciu používateľa aplikácia najskôr zareaguje sama, potom postupne vyvoláva príslušnú akciu u všetkých zavedených zásuvných modulov. Tieto vykonajú svoju prácu a odovzdajú riadenie ďalej. Aplikácia nakoniec ešte môže skontrolovať výsledky a zobrazí výsledok používateľovi.

Na prvý pohľad sa to môže zdať ako jednoduché a efektívne riešenie a v ideálnom svete funguje veľmi dobre. Ak však vezmeme do úvahy rôznorodosť autorov, predstava, že cudziemu kódu odovzdáme na chvíľu riadenie celej aplikácie, nie je práve príjemná. Spoliehali by sme sa pri tom na korektné fungovanie všetkých zásuvným modulov. V prípade, že jeden spôsobí chybu, vyvolá tým nestabilitu celej aplikácie. Navyše, odovzdanie kontroly naspäť do hlavného programu nie je možné vynútiť. Jeden chybný zásuvný modul tak môže spôsobiť „zamrznutie" celej aplikácie.

Zásuvný modul v samostatnom vlákne je už čiastočne oddelený od hlavnej aplikácie. Stále však zdieľa spoločný adresný priestor, takže prístup k stavu aplikácie je i naďalej rýchly.

Celá aplikácia sa tým automaticky stáva viacvláknová. To so sebou prináša komplikácie spôsobené prístupom viacerých účastníkov ku zdieľaným prostriedkom, napríklad k stavovým premenným. Môžu tým vzniknúť všetky druhy problémov známe z teórie operačných systémov (uviaznutie, súperenie o premenné a i.). Okrem nich je tu ešte jeden problém – Grafické používateľské prostredie. Väčšina systémov grafických rozhraní je postavených tak, že je k nim možné pristupovať iba z jedného programového vlákna.

Na druhú stranu, oddelenie vykonávania kódu aplikácie a zásuvného modulu dodáva aplikácii určitú autoritu. V prípade, že sa zásuvný modul začne

správať nekorektne, môže ho vypnúť. V prípade, že výpočet v zásuvnom module trvá pridlho, aplikácia neprestáva reagovať na podnety od používateľa. No i tak, v prípade závažnej chyby zásuvného modulu, môže dôjsť k pádu celej aplikácie.

Zásuvný modul ako samostatný proces je už úplne oddelený od aplikácie. Nemajú spoločný pamäťový priestor a prepojený sú len prostriedkami medziprocesovej komunikácie (viď. 4.1.4). Znamená to, že táto komunikácia je výrazne pomalšia v porovnaní s predchádzajúcimi dvomi prípadmi.

Rovnako, ako vo viac-vláknovom prístupe, aj tu sa objavia problémy s prístupom viacerých účastníkov k spoločným zdrojom. V tomto prípade sú to však už len stavové premenné, ku ktorým aplikácia sprostredkúva prístup zásuvným modulom prostredníctvom svojho rozhrania.

Spravovanie *GUI* sa v tomto prípade dostane do inej roviny, ako v predchádzajúcich prípadoch. Každý zo zásuvných modulov si teoreticky môže vytvoriť vlastné okno a sám si ho môže aj spravovať. Na druhú stranu, vznikla by zvláštna situácia, keď by mal používateľ otvorených niekoľko okien, každé z nich by zdanlivo patrilo inej aplikácii a napriek tomu by mali tvoriť jeden celok. Pre väčšinu používateľov by to bola úplne neprijateľná situácia (známa z niektorých linuxových aplikácií.). Zásuvný modul teda bude musieť žiadať v prípade potreby o rozšírenie okna aplikácie.

Úplné oddelenie aplikácie od zásuvného modulu okrem spomínaných komplikácií prináša zároveň aj určitú formu ochrany. Ak sa v zásuvnom module vyskytne problém a proces zásuvného modulu „spadne", stabilitu hlavnej aplikácie to neohrozí. Táto môže takýto zásuvný modul opäť zaviesť, alebo iba informovať používateľa o chybe.

Taktiež je chránená od nekorektného správania sa zásuvného modulu. V prípade spoločného adresného priestoru môžu byť stavové premenné aplikácie čiastočne izolované od zásuvného modulu, do akej miery, to závisí skôr od použitého programovacieho jazyka. V prípade oddelených procesov sa zásuvný modul dostane iba k tým informáciám, ktoré mu aplikácia na požiadanie poskytne a zároveň môže ovplyvniť stav aplikácie iba presne definovaným spôsobom.

4.1.4 Možnosti komunikácie medzi procesmi

Na komunikáciu medzi procesmi možno využiť nasledujúce spôsoby:

- Zdieľaná pamäť
- Rúry (pipes)

- Sokety (sockets)
- Fronty správ
- D-Bus

Každý zo spomínaných spôsobov komunikácie bol vyvíjaný s určitým zámerom a má preto, v porovnaní s inými, svoje výhody aj nevýhody.

Zdieľaná pamäť predstavuje adresný priestor, ku ktorému majú spoločne prístup viaceré procesy. V prípade, že je práca s ním vhodne navrhnutá, môže predstavovať najrýchlejší spôsob komunikácie, najmä ak medzi procesmi potrebujeme prenášať väčšie objemy informácií.

Jeho rýchlosť spočíva práve v tom, že v skutočnosti sa informácia neprenáša z jedného procesu do druhého, ale že jeden proces informáciu niekam uloží a druhý ju tam nájde.

Avšak, niektoré moderné programovacie jazyky (Java, C#, Python) už prakticky úplne skrývajú fyzické adresy objektov. Prípadne umožnia programátorovi zistiť, kde v pamäti sa objekt nachádza, ale vytvárajú prekážky tomu, aby umiestnil objekt na ľubovoľné miesto v pamäti.

Rúry sú pre programátora veľmi podobné súborom. Správajú sa dosť podobne ako textovým súbory známe z jazyka Pascal. To znamená, že sa v nich nedá vyhľadávať pozícia, dá sa iba čítať prúd znakov.

Operačné systémy rozoznávajú dva typy rúr:

- Anonymné – v procese sú identifikované iba popisným číslom,
- Pomenované – existujú akoby virtuálne súbory v súborovom systéme.

Anonymné rúry môžu používať medzi sebou dva procesy iba v tedy, ak je jeden potomkom druhého. Potomok musí zdediť popisné číslo, inak sa k rúre nedostane. K pomenovanej rúre sa môže pripojiť ľubovoľný proces, ak má oprávnenie na prístup k súboru v danom súborovom systéme.

Soket má svoj pôvod v komunikácii po počítačovej sieti. Na unixových operačných systémoch sú k dispozícii aj sokety určené na komunikáciu medzi procesmi v rámci jedného počítača, neposkytuje ich však operačný systém Windows.

Obmedzím sa preto na *IP*[2] sokety, ktoré sú prístupné prakticky vždy. To je ich najväčšou výhodou. Ak chcú komunikovať procesy v rámci jedného

[2] *Internet Protocol* – protokol používaný na prenos dát cez siete s prepínaním paketov. Zabezpečuje prenos paketov postupne cez uzly v sieti.

operačného systému prostredníctvom *IP* soketov, môžu využiť virtuálne *IP* rozhranie nazvané *loopback*[3].

Cez *IP* soket môžu využiť komunikáciu prostredníctvom *TCP*[4] protokolu, čím získajú charakter komunikácie podobný ako pri rúrach. Ak použijú protokol *UDP*[5], ktorý pri prenose cez *loopback* možno považovať tiež za spoľahlivý, získajú tým komunikáciu podobnú frontu správ.

Nevýhodou tohto systému môže byť dlhší čas doručenia správy v porovnaní s ostatnými spôsobmi. Dôvodom je, že odoslaná správa musí prejsť cez *TCP* alebo *UDP* a potom *IP* vrstvu operačného systému, pričom sa najskôr zabalí a rozdelí na IP pakety a následne opäť spojí a rozbalí. Ak by bola rýchlosť komunikácie kritickým faktorom pri výbere prenosového kanálu, môže toto spomalenie znamenať problém. Ak však hovoríme o aplikácii, kde zásuvný modul aj aplikácia samotná fungujú predovšetkým iba ako reakcia na používateľovu akciu, tak toto spomalenie nie je dôležité.

Podstatne závažnejší problém predstavujú rôzne bezpečnostné prvky, predovšetkým *firewall*[6]. Táto môže byť nastavená tak, aby blokovala *IP* komunikáciu aj na rozhraní *loopback*. V takomto prípade by bola komunikácia medzi procesmi znemožnená aspoň do momentu, pokiaľ ju používateľ nepovolí.

Fronty správ umožňujú zasielanie pevne definovaných správ medzi viacerými procesmi. Umožňujú komunikáciu M:N. Pevná definícia správ je však určená skôr pre nižšie programovacie jazyky ako napr. C.

Hlavnou nevýhodou tohto systému je odlišnosť implementácie na operačnom systéme Windows a iných operačných systémoch.

D-Bus je jedným z novších systémov pre komunikáciu medzi procesmi. Vyvýjaný predovšetkým firmou *Red Hat*, je dnes súčasťou väčšiny Linuxových distribúcií. Umožňuje komunikáciu veľkého počtu aplikácií súčasne. Jednotlivé procesy môžu vytvárať virtuálne kanály, po ktorých sa správy šíria. Taktiež sa môžu zaregistrovať, aby dostávali iba vybrané typy správ od vybraných procesov.

Nevýhodou tohto systému je jeho absencia na operačnom systéme Windows. V čase písania tejto diplomovej práce síce už existuje projekt starajúci

[3]Virtuálne sieťové rozhranie vytvorené na testovacie účely a možnosť komunikácie „sieťovým protokolom" medzi procesmi bežiacimi na jednom počítači.

[4]*Transmission Control Protocol* – protokol používaný na spoľahlivý prenos dát cez *IP* sieť. Vytvára spojenie medzi koncovými účastníkmi spojenia.

[5]*User Datagram Protocol* – protokol používaný na prenos dát cez *IP* sieť. Vytvára spojenie medzi koncovými účastnínkmi spojenia, ale nezabezpečuje spoľahlivý prenos.

[6]Bezpečnostný sieťový prvok, filtrujúci neautorizovanú sieťovú komunikáciu

sa o prenos *D-Bus* aj sem, nie je však ešte oficiálne začlenený do hlavnej vývojovej vetvy.

4.1.5 Príklady použitia zásuvných modulov

Medzi počítačovými programami je množstvo takých, ktoré využívajú zásuvné moduly. V tejto časti uvediem niekoľkých zástupcov.

Rhythmbox je aplikácia pre správu hudobného obsahu. Je vyvíjaná pod *GNU/GPL* licenciou a určená najmä pre prostredie *Gnome*. Postavená je na kostre projektu *GStreamer*.

Aplikácia samotná je napísaná v jazyku C. Zásuvné moduly je možné pre ňu napísať v jazyku C alebo v jazyku Python. Napriek tomu, že niektoré časti programového rozhrania nie sú do Pythonu premostené, tento jazyk odporúčajú pre tvorbu zásuvných modulov samotní vývojári.

Rhythmbox je typická jedno-vláknová aplikácia. Vlákna dokonca nie sú povolené ani v rámci zásuvných modulov. Preto je ich vývoj pomerne priamočiary. Ak zásuvný modul potrebuje k svojej činnosti svoje vlastné okno, definuje ho pomocou nástroja *Glade*[7].

Pidgin je aplikácia umožňujúca zasielanie okamžitých správ (instant messages) cez viaceré známe siete (ICQ, GoogleTalk, XMPP, MSN, ...). Je vyvíjaná pod *GNU/GPL* licenciou a je prenosná na väčšinu známych operačných systémov (Linux, Windows, MacOS, ...). Aplikácia samotná je grafickou nadstavbou knižnice *libpurple*, ktorá zabezpečuje komunikáciu cez jednotlivé siete. Aplikácia aj knižnica sú napísané v jazyku C.

Zásuvné moduly pre *Pidgin* môžu byť napísané v jazyku C (odporúčaný tvorcami), ale s určitými obmedzeniami aj v jazykoch Perl a Tcl. Tieto zásuvné moduly pracujú v rovnakom vlákne ako jadro aplikácie. To im umožňuje ovplyvniť reakciu programu na niektoré vzniknuté udalosti. (Napríklad môžu zrušiť akciu prehrania zvuku.)

Pidgin však umožňuje pripojiť zásuvný modul aj ako samostatný proces. Na komunikáciu sa v takomto prípade používa *D-Bus*. Je teda možné naprogramovať zásuvný modul aj v iných jazykoch (napríklad C++, Python, ...). Zásuvný modul, komunikujúci týmto spôsobom, však má určité obmedzenia. O udalostiach je iba informovaný, že nastali, nemôže ich zrušiť alebo ovplyvniť. Môže však dodatočne vyvolávať iné akcie.

[7] Popis nájdete v kapitole 3.2.3

Winamp je proprietárny mediálny prehrávač vyvíjaný firmou `Nullsoft`. Je určený pre operačný systém Windows. Napísaný je v jazyku C++.

Zásuvné moduly do tohto prehrávača sa delia na viacero typov podľa svojeho použitia. Sú dodávané vo forme samostatných DLL knižníc a tvorcovia odporúčajú na ich tvorbu programovací jazyk C alebo C++. V závislosti na spôsobe použitia môže zásuvný modul bežať v niektorom vlákne hlavnej aplikácie (napríklad modifikujúce zvukový výstup), alebo v samostatnom.

Špeciálnu kategóriu zásuvných modulov pre *Winamp* tvoria vzhľadové moduly (tzv. „skins"). Tieto sú ale iba definíciou vzhľadu a rozloženia *GUI* prehrávača. Neobsahujú vykonateľný kód.

4.2 Zásuvné moduly v aplikácii UML .FRI

V kapitole 3.3 som písal, že aplikácia UML .FRI samotná je nezávislá na modelovacom jazyku. A teda, až keď si používateľ vyberie projekt a s ním metamodel, tak bude jasné, či pri modelovaní použije *UML*[8], *ERA*[9], vývojový diagram alebo niečo iné. Z toho dôvodu aplikácia neobsahuje žiadne funkcie, ktoré by mali byť špecifické pre jeden konkrétny modelovací jazyk.

Moderné modelovacie nástroje však okrem základného modelovania umožňujú s diagramami vykonávať činnosti, ktoré sú však už špecifické pre konkrétny modelovací jazyk, či skôr typ diagramu. Môžeme sem zaradiť napríklad:

- generovanie štruktúry zdrojových kódov z *UML* diagramu tried a spätné generovanie *UML* diagramov zo zdrojových kódov,
- Generovanie SQL skriptov z *ERA* diagramu a spätné generovanie *ERA* diagramov.
- Hľadanie najkratšej cesty v grafe, hľadanie kritickej cesty v sieťovom grafe,
- a iné.

Hoci sa metamodel odlišuje od zásuvného modulu predovšetkým tým, že nie je vykonateľný, môžeme na neho aplikovať predpoklady z kapitoly 4.1.3. Platia všetky, okrem posledného z nich, pretože projekt je vytváraný vždy len podľa jedného metamodelu a teda nemôže dôjsť ku konfliktu dvoch metamodelov.

[8] *Unified Modeling Language* – vizuálny jazyk na modelovanie a komunikáciu o systéme pomocou diagramov a podporného textu [8]

[9] *Entity Relationship Attribute* – Diagram na entitno-relačný návrh databázy.

Zo spomínaných predpokladov teda možno usudzovať, že aj keby sme mali ambíciu rozširovať aplikáciu o špecifické algoritmy, pravdepodobne by sme neuspeli. Ktorýkoľvek používateľ so znalosťou jazyka *XML* a štruktúry metamodelu bude schopný vytvoriť, alebo upraviť nejaký metamodel pre svoje potreby. Takáto úprava je vo všeobecnosti menej náročná, ako zásah do zdrojových kódov aplikácie, čo by však mohlo byť potrebné v prípade úpravy metamodelu.

Okrem toho by bolo nepríjemné, keby mal byť priamo do aplikácie implementovaný napríklad algoritmus pre hľadanie najkratšej cesty v grafe. Tento algoritmus je totiž navrhnutý pre jednoduchú štruktúru grafu zloženého z uzlov a hrán. Teraz by musel byť prispôsobený vnútornej štruktúre aplikácie UML .FRI. Tá je však stavaná skôr na všeobecné informácie, potrebné pre kreslenie diagramov.

V konečnom dôsledku by sa tu prejavili problémy monolitickej aplikácie, ktoré sú popísane v kapitole 4.1.1. Na základe všetkých spomínaných dôvodov došlo k rozhodnutiu, že všetky funkcie, ktoré sú nejakým spôsobom špecifické a teda určené iba pre jeden metamodel, budú naprogramované vo forme zásuvných modulov.

4.2.1 Výber stupňa oddelenia zásuvného modulu od aplikácie

V kapitole 4.1.3 som popísal tri všeobecné možnosti, ako pripájať zásuvný modul k aplikácii. Teraz tieto spôsoby zhodnotím v kontexte aplikácie UML .FRI s aplikovaním požiadavkov z kapitoly 2.

Zásuvný modul, vykonávaný v spoločnom programovom vlákne, je vo svetle požiadaviek, vymenovaných v kapitole 2, takmer úplne nevyhovujúci.

Aj keby som zabezpečil, že žiadna výnimka, ktorá vznikla v zásuvnom module, neprenikne až do kódu samotnej aplikácie, nezabránim sa problémom s dočasným „zamrznutím" aplikácie. Ak aplikácia raz odovzdá kontrolu zásuvnému modulu, nevynúti si jej opätovné vrátenie.

Túto možnosť preto vylúčim.

Zásuvný modul v samostatnom vlákne je, aj s prihliadnutím na použitý programovací jazyk, pomerne silne oddelený od samotnej aplikácie. V Pythone, ak totiž vznikne výnimka v niektorom z vlákien (okrem prvého vlákna), stabilitu ostatných to priamo neohrozí.

Na druhú stranu, práve pre použitý programovací jazyk a jeho konkrétnu implementáciu (CPython) nie sú vlákna efektívne v prípade, že sa objaví niekoľko výpočtovo náročných úloh. Súvisí to s tzv. všeobecným zámkom interpretra (General Interpreter Lock), ktorý prakticky znemožňuje paralelný beh vlákien na počítači s viacerými *CPU*[10].

Objavuje sa tu však problém s *GUI*. Tak, ako mnohé iné nástroje na tvorbu *GUI*, tak aj *GTK+* môžu byť ovládané iba z jedného vlákna. To znamená, že ak by chcel zásuvný modul nejakým spôsobom zmeniť *GUI*, musel by k nemu pristupovať sprostredkovane. Ak by sme sa chceli vyhnúť možnosti „zamrznutia" počas spracovania udalosti zásuvným modulom, musela by byť udalosť spracovaná v samostatnom vlákne a to znamená asynchrónne.

Pre programátora zásuvného modulu by tento spôsob znamenal však jedno závažné obmedzenie. Bol by limitovaný na používanie programovacieho jazyka Python. Hoci je tento programovací jazyk veľmi vhodný pre vývoj aplikácií, ktoré pracujú interaktívne s používateľom, mohol by predstavovať bariéru pre značný počet ľudí.

Zásuvný modul ako samostatný proces je od hlavnej aplikácie úplne oddelený. Komunikácia prebieha len s použitím prostriedkov medziprocesovej komunikácie (viď. kapitola 4.1.4). Aplikácia plne kontroluje, k čomu zásuvnému modulu umožní prístup. Autor zásuvného modulu nie je limitovaný programovacím jazykom, pretože komunikácia medzi procesmi je univerzálna.

Prináša však obmedzenie vo forme spomalenia, lebo všetky volania musia byť prenesené z jedného procesu do druhého. Taktiež práca s *GUI* je značne obmedzená na súbor niekoľkých volaní, ktoré aplikácia poskytuje.

Napokon sa však táto možnosť, s prihliadnutím na požiadavky, ukazuje ako najvýhodnejšia. Z toho dôvodu som si ju zvolil.

4.2.2 Prístupnosť jednotlivých častí aplikácie

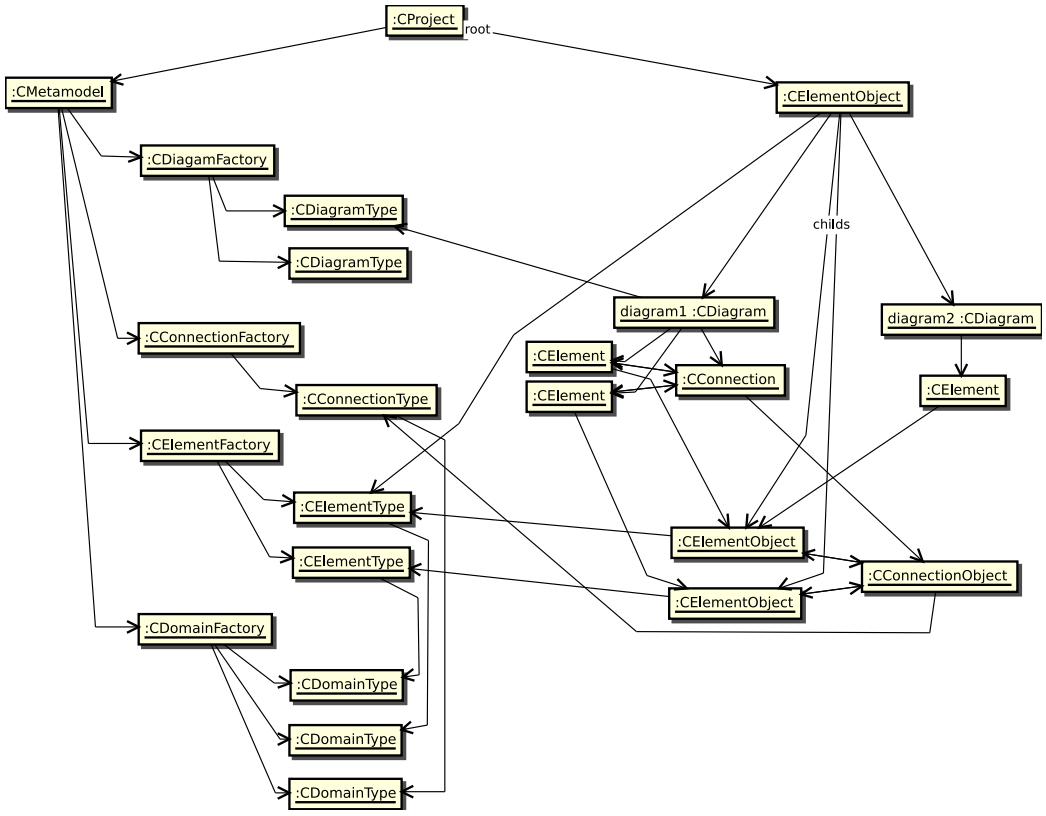
Je zrejmé, že zásuvný modul nebude potrebovať prístup ku všetkým objektom aplikácie a že k niektorým objektom by mal byť prístup iba na čítanie. Postupnou úvahou som dospel k nasledujúcim záverom:

- Bude potrebný prístup k aktuálne načítanému metamodelu iba na čítanie.
- Bude potrebný prístup k aktuálnemu modelu na čítanie aj zapisovanie.

[10] *Central Processing Unit* – centrálna výpočtová jednotka počítača. V súčastnosti predstavuje jedno jadro procesora jedno CPU.

- Objektová štruktúra, reprezentujúca model v aplikácii, je príliš podrobná pre potreby zásuvného modulu. Z toho dôvodu budem abstrahovať od objektov tried `CTreeNode` a `CDomainObject`.
- Zásuvný modul bude potrebovať sprostredkovaný prístup ku *GUI*.
- Vzhľadom k oddeleniu jadra aplikácie od použitého nástroja na vytváranie *GUI*(v súčastnosti GTK+), mal by byť oddelený aj zásuvný modul. Aplikácia mu preto neposkytne plnú objektovú štruktúru, ale iba niekoľko štandardizovaných volaní.

Na základe týchto záverov bude pre zásuvný modul prístupná objektová štruktúra znázornená na obrázku 4.1.



Obr. 4.1: *Diagram objektov viditeľných pre zásuvný modul*

4.2.3 Typy správ

Je zrejmé, že aplikácia bude so zásuvným modulom komunikovať prostredníctvom nejakých správ. Správy bude potrebné posielať v nasledujúcich situáciách:

- zásuvný modul sa pripája k aplikácii,
- zásuvný modul sa odpája od aplikácie,
- aplikácia informuje zásuvný modul o udalosti, ktorú vyvolal používateľ alebo iný zásuvný modul,
- zásuvný modul reaguje na udalosť,
- zásuvný modul chce vykonať iniciatívne nejakú akciu,
- zásuvný modul žiada o informácie,
- aplikácia odpovedá na žiadosť zásuvného modulu.

Z daného zoznamu situácií vyplýva, že správy sa budú principiálne deliť na tieto typy:

- príkaz na vykonanie činnosti,
- žiadosť o nejakú hodnotu,
- odpoveď na žiadosť,
- informatívna správa o udalosti.

Z týchto štyroch typov správ budú prvé dve posielať zásuvné moduly a druhé dve aplikácia. Okrem toho, prvé dva typy správ sa navzájom principiálne odlišujú iba v tom, či na ne zásuvný modul očakáva spätnú odpoveď. Druhá dvojica správ sa tiež navzájom odlišuje iba príznakom, či je správa odpoveďou alebo informáciou o udalosti.

4.2.4 Výber prenosového média

Z rozhodnutia pripájať zásuvné moduly k aplikácii ako samostatné procesy vyplýva, že na ich vzájomnú komunikáciu musím použiť jednu z možností popísaných v kapitole 4.1.4. Pri výbere, najvhodnejšej z nich zohľadním predovšetkým nasledujúce:

- Pre komunikáciu medzi zásuvným modulom a aplikáciou bude charakteristické, že relatívne dlhé intervaly bez komunikácie bude prerušovať výmena niekoľkých správ.
- Tieto správy budú vyvolané nejakou akciou používateľa, preto priepustnosť prenosového kanálu nie je hlavným kritériom výberu.
- Jedna správa bude pomerne malá, ale nedá sa dopredu určiť jej presná veľkosť.
- Prenosové médium by malo byť univerzálne v zmysle prenosnosti medzi platformami (operačný systém aj programovací jazyk).
- Svoje osobné skúsenosti s medziprocesovou komunikáciou.

Z jednotlivých možností medziprocesovej komunikácie sa ako najuniverzálnejšie javia byť rúry a sokety, pretože umožňujú posielanie správ variabilnej dĺžky. Aj programové rozhranie a správanie sa na rôznych operačným systémoch majú veľmi podobné. Ale vzhľadom k vlastným bohatším skúsenostiam so soketmi a jednoduchšej manipulácii s nimi pri testovaní, som sa pre prvú verziu rozhodol uprednostniť práve sokety.

4.2.5 Výber komunikačného protokolu

Pri výbere komunikačného protokolu treba brať do úvahy predovšetkým tieto kritériá:

- Zásuvný modul môže byť napísaný v ľubovoľnom programovacom jazyku. Z toho dôvodu treba pamätať pri výbere komunikačného protokolu predovšetkým na univerzálnosť a prenosnosť medzi platformami.
- Jednoduché ladenie celého systému zásuvných modulov. Pri ladení pomáha, ak je možné plynulo čítať a zapisovať ľubovoľné správy v komunikácii medzi zásuvným modulom a aplikáciou.
- Nie je možné dopredu vymenovať celú množinu správ, ktoré bude potrebné zasielať. Nie je dokonca možné ani ohraničiť veľkosť jednej správy.

Pri výbere komunikačného protokolu možno principiálne uvažovať o dvoch typoch protokolov. Binárny a textový. Ak však zvážime kritériá, ktoré som práve položil, jedinou nevýhodou textového protokolu oproti binárnemu môže byť, že správa s tou istou informačnou hodnotou zaberá viac miesta v pamäti. Naopak, pri ladení sa s ním pracuje podstatne ľahšie, pretože čítanie textu je

```
GET / HTTP/1.1
Host: localhost:3000
Accept: text/html,application/xhtml+xml
Accept-Language: en-gb,en;q=0.7,sk;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: UTF-8,*
Keep-Alive: 300
Connection: keep-alive
```

Výpis 4.1: *Ukážka HTTP požiadavky.*

pre ľudí jednoduchšie ako čítanie číselných kódov a ľahšie sa rozširuje o nové správy a parametre správ. Celkovo sa textový protokol javí ako výhodnejší.

Na základe svojich predošlých skúseností so sieťovými protokolmi a programovaním aplikácií komunikujúcich po sietí som sa rozhodol, že na formátovanie správ využijem štandard popísaný v *RFC 2822*. Je to textový protokol, v ktorom sú jednotlivé časti správy rozdelené do riadkov. Zjednodušene povedané, prvý riadok správy obsahuje príkaz samotný a nasledujúce riadky potom parametre príkazu. Parametre sú zapísané ako dvojice názov parametra, hodnota parametra, oddelené dvojbodkou. Toto formátovanie je známe aj z protokolov *HTTP*[11], *SMTP*[12], *SIP*[13]. Výhodou tohto formátovania je veľmi dobrá čitateľnosť pre človeka a pomerne prijateľná hustota prenášanej informácie v správe (v porovnaní napríklad s *XML*).

[11] *HyperText Transfer Protocol* – Protokol, využívaný v počítačových sieťach na prenos predovšetkým hypertextových dokumentov. Principiálne umožňuje prenos ľubovoľnej informácie.

[12] *Simple Mail Transfer Protocol* – Protokol používaný na prenos elektronickej pošty cez Internet.

[13] *Session Initiation Protocol* – Štandard využívaný na signalizáciu a vytváranie spojení (predovšetkým hlasové služby) cez Internet.

Kapitola 5

Implementácia systému zásuvných modulov

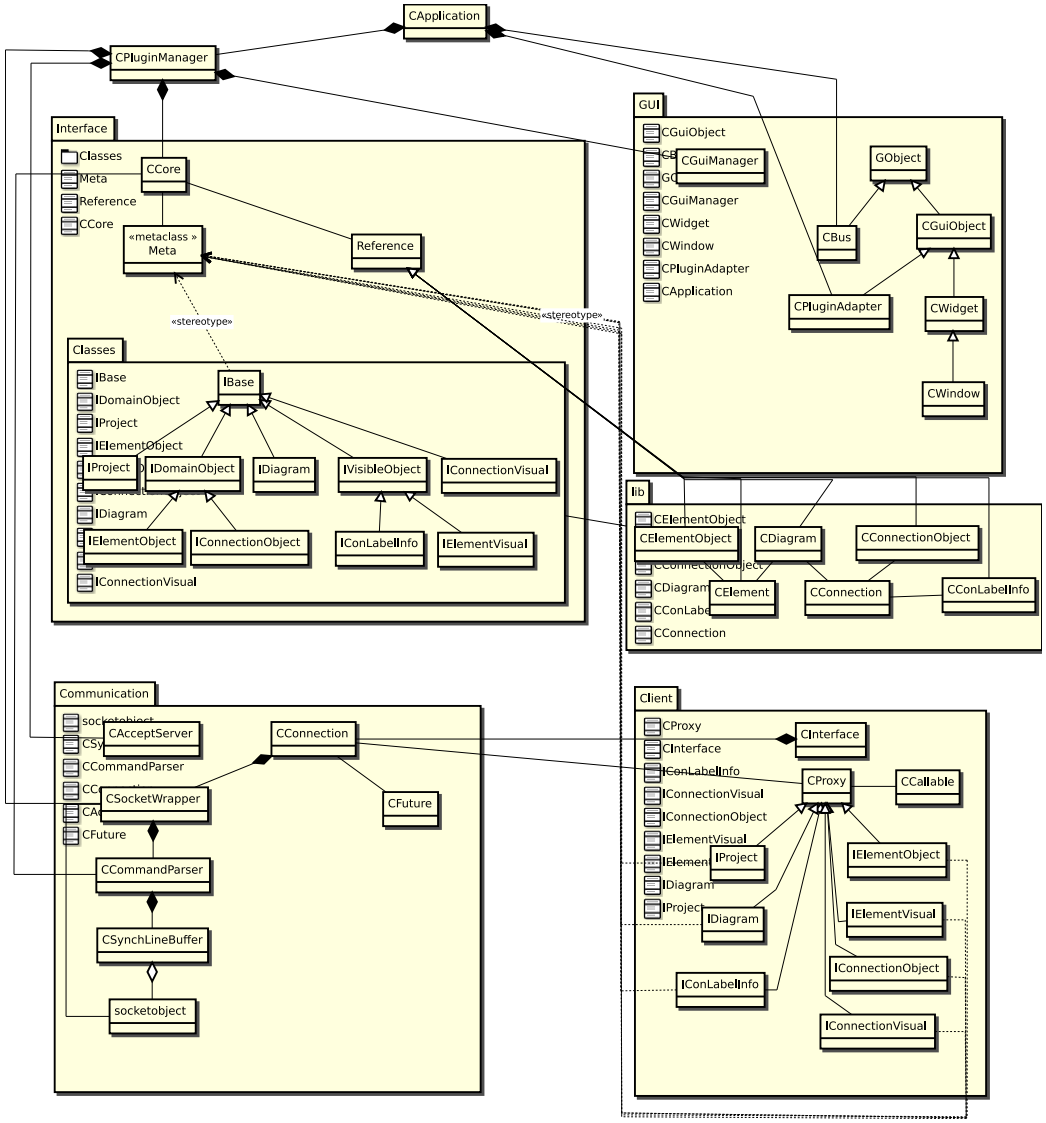
V tejto kapitole je popísaný spôsob implementácie systému zásuvných modulov v aplikácii UML .FRI. Prvá časť sa venuje komunikačnému kanálu medzi zásuvným modulom a aplikáciou. Druhá časť popisuje príkazy, ktoré môže zásuvný modul zaslať aplikácii. Tretia rozoberá spôsob distribúcie udalostí vyvolaných používateľom alebo zásuvným modulom tak, aby sa o tom dozvedeli všetky zúčastnené prvky systému. Štvrtá časť popisuje implementáciu knižničného rozhrania pre zásuvný modul napísaný v jazyku Python.

Obrázok 5.1 znázorňuje usporiadanie tried systému zásuvných modulov a knižnice pre zásuvné moduly do jednotlivých častí – balíčkov.

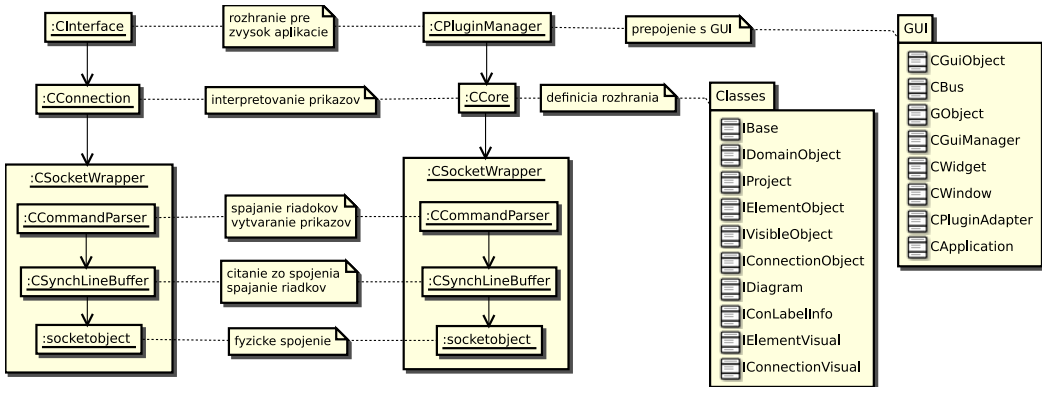
5.1 Prenos a spracovanie informácie medzi zásuvným modulom a aplikáciou

Pri implementácii komunikačnej časti systému zásuvných modulov som sa snažil rozdeliť jednotlivé časti do viacvrstvovej architektúry. Jednotlivé vrstvy sú implementované tak, aby jedna vrstva vždy žiadala o službu svoju podriadenú vrstvu a odovzdávala štandardizované správy svojej nadradenej vrstve. Usporiadanie jednotlivých objektov v aplikácii a zásuvnom module je znázornené na obrázku 5.2.

Dôvodom pre takéto oddelenie je najmä fakt, že zvolené komunikačné médium – TCP soket – nie je jediné riešenie, s ktorým je do budúcnosti počítané. V budúcnosti bude umožnené spojenie aj pomocou rúr.



Obr. 5.1: *Diagram tried systému zásuvných modulov*



Obr. 5.2: *Znázornenie usporiadania objektov do komunikačných vrstiev*

5.1.1 Parsovanie správ

V kapitole 4.2.5 som písal, že na prenos správ medzi aplikáciou a zásuvným modulom budem používať textový protokol a správy budú formátované podľa *RFC 2822*. To znamená, že jedna správa je tvorená textom, ktorý je rozdelený na riadky. Prvý riadok správy obsahuje samotný príkaz a nasledujúce riadky obsahujú parametre príkazu. Správa sa končí prázdnym riadkom.

Vzhľadom k tomu, že TCP soket prenáša všetky dáta ako prúd bajtov a operácia čítania z prázdneho soketu je blokujúca, vytvoril som dvojicu tried `CSynchLineBuffer` a `CCommandParser`. Jeden objekt každej z nich je vytvorený pre každé spojenie aj na strane aplikácie aj na strane zásuvného modulu.

Objekt triedy `CSynchLineBuffer` postupne číta prúd bajtov zo soketu a rozdeľuje ho na riadky. Objekt triedy `CCommandParser` ho obaľuje a necháva bežať čítanie v samostatnom vlákne, aby ním neblokoval zvyšok aplikácie. Prichádzajúce riadky analyzuje s využitím regulárnych výrazov a spája ich do príkazov s parametrami. Keď je jeden príkaz poskladaný, použije odkaz na svoj proxy objekt a vyvolá jeho metódu `Command`. Proxy objektom je v prípade aplikácie objekt triedy `CCore` a v prípade zásuvného modulu objekt triedy `CConnection`.

Objekty spomínaných tried slúžia na prijímanie správ. Každý z nich, tak ako boli postupne vymenované, tvoria samostatné vrstvy v reťazi spracovania správ. Aby bolo možné na správy odpovedať, vytvoril som triedu `CSocketWrapper`. Ku každému soketu je vytvorený jeden objekt tejto triedy. Podstatná je metóda `Send`, ktorá zo zadaných parametrov spätne poskladá textovú reprezentáciu správy a túto odošle. Ukážku protokolu si pozriete na

```
exec #26.SetValue UML.FRI/0.1
__id__: 108
path: critical
value: True
```

Výpis 5.1: *Ukážka komunikačného protokolu*

výpise 5.1.

5.1.2 Interpretovanie prijatých správ

Interpretovanie prijatých správ prebieha buď v objekte triedy `CCore` (v hlavnej aplikácii), alebo v objekte triedy `CConnection` (v zásuvnom module). Objekt jednej alebo druhej triedy je iba jeden v celej aplikácii / zásuvnom module.

Správy, ktoré posielajú zásuvné moduly aplikácii, majú príkaz rozdelený na dve časti – na príkaz a podpríkaz. V praxi sa teda príkazy roztrieďujú pomocou vnorených podmienok. (V slovníku jazyka Pascal by sa dalo povedať, že pomocou vnorených `CASE` príkazov.)

Správy, ktoré posiela aplikácia do zásuvného modulu (informatívne alebo návratové hodnoty volaní) majú iba jednu hodnotu pre príkaz. Príkazy sú však kódované trojciferným číselným kódom (podobne ako odpovede v protokoloch *HTTP*, *SMTP*, *SIP*) a cifra na mieste stoviek vypovedá o druhu správy. Podobne tak dochádza k triedeniu prijatých správ systémom vnorených podmienok.

5.1.3 Spracovanie chybných správ

Prijatá správa môže byť vyhodnotená ako chybná z rôznych dôvodov a preto môže byť chyba zachytená na rôznych miestach. Pri postupnom prechode cez jednotlivé vrstvy sa v správe môže zistiť jedna z týchto chýb:

1. V triede `CSynchLineBuffer`, pri rozdeľovaní prúdu bajtov na riadky. V tomto prípade však chybový stav môže vyvolať iba zatvorenie soketu (druhou stranou). Ak sa nadradená vrstva pokúsi čítať aj po uzatvorení soketu, namiesto riadku textu dostane hodnotu `None`.

2. Pri analýze riadkov a skladaní príkazov v triede `CCommandParser`. Skladanie príkazov funguje ako stavový automat a v každom konkrétnom stave musí prijatý riadok vyhovovať konkrétnemu regulárnemu výrazu. Ak je prijatý riadok, ktorý nevyhovuje, celý doteraz skladaný príkaz

je vyhodnotený ako chybný a zahodený. Nadradená vrstva je informovaná o chybe. V prípade, že soket bol uzatvorený, nadradená vrstva je informovaná o ukončení spojenia.

3. Pri triedení príkazov v triede `CCore` alebo `CConnection`. V tomto prípade môže dôjsť k niekoľkým typom chýb:

   - zadaný príkaz je neznámy,
   - príkaz obsahuje nesprávne parametre, alebo nesprávne hodnoty parametrov, alebo nejaké parametre chýbajú,
   - príkaz aj parametre sú syntakticky správne, ale pokus o jeho vykonanie so zadanými hodnotami skončí chybou,
   - podriadená vrstva zistila chybu.

V prípade že chyba vznikne na prostrednej vrstve (2. bod), predpokladám, že na druhej strane je proces, s ktorým som v skutočnosti nechcel komunikovať. Pravdepodobne komunikuje iným komunikačným protokolom a nemalo by zmysel posielať mu správu o chybe v komunikácii. Aplikácia spojenie jednoducho zruší.

Pri detekovanej chybe na najvyššej vrstve už môžem predpokladať, že skutočne komunikujú navzájom aplikácia a zásuvný modul. V takomto prípade má zmysel upozorniť druhú stranu na vzniknutý problém. Aplikácia preto posiela zásuvnému modulu spätnú správu s kódom chyby.

Zásuvný modul aplikácii správy o chybe neposiela a to z dvoch dôvodov:

- aby nevznikol cyklus, v ktorom by krúžili správy o chybách,
- bol to zásuvný modul, kto sa pripojil k aplikácii, nie naopak. Ak sa mu nepáči, môže sa odpojiť.

5.2 Definícia množiny volaní v aplikácii

Aplikácia rozoznáva tri základné príkazy a každý z týchto príkazov obsahuje svoj podpríkaz.

- `gui` – príkazy týkajúce sa vzhľadu grafického používateľského prostredia,
- `metamodel` – čítanie informácií z metamodelu,
- `exec` – volanie metódy niektorého z objektov modelu.

Príkazy, ktoré pošle zásuvný modul, sa môžu v aplikácii vykonávať aj asynchrónne. Nie je teda garantované, že odpovede na príkazy prídu naspäť v rovnakom poradí, v akom boli zaslané príkazy. Preto je potrebné párovanie odpovedí s príkazmi. Na tento účel som vyhradil parameter `''__id__''`, ktorý môže byť prítomný v ľubovoľnom príkaze. Jeho hodnotou môže byť ľubovoľný reťazec a je v záujme zásuvného modulu, aby používal unikátne identifikátory, aplikácia to nesleduje.

V prípade, že sa tento parameter v príkaze nachádza, na príkaz bude zaslaná príslušná odpoveď, aj keby mal byť výsledok prázdny (mal hodnotu `None`). V prípade, že sa parameter vo volaní nenachádza, aplikácia vykoná požadovanú činnosť, ale výsledok nepošle naspäť.

5.2.1 Príkaz `gui`

Keďže je systém zásuvných modulov implementovaný nezávisle na použitej knižnici vytvárajúcej GUI, nemá to zaujímať ani zásuvný modul. Každá z týchto knižníc má svoje špecifiká. Majú však aj dostatok spoločných znakov na to, aby som mohol vytvoriť základnú sadu funkcií, ktoré sa dajú použiť univerzálne.

Napríklad, každá z nich umožňuje aplikácii vytvoriť hlavné menu, lištu s tlačidlami a kontextové menu. Taktiež môžu byť tieto položky aktívne alebo neaktívne (necitlivé). Spracovanie udalostí typu „bolo kliknuté na konkrétne tlačidlo", býva vždy riešené tak, že je zaregistrovaná jedna konkrétna metóda/funkcia, ktorá je vyvolaná, ak vznikne nejaká konkrétna udalosť. Taktiež býva k dispozícii možnosť zobraziť jednoduché dialógové okno.

Pre potreby úpravy a riadenia GUI zo zásuvného modulu som preto vytvoril príkaz `gui`. Tento má implementovanú sadu podpríkazov umožňujúcich práve spomínané všeobecné funkcie:

- `add` – pridáva položku do programového menu,
- `sensitive`, `insensitive` – aktivuje/deaktivuje položku menu,
- `warning` – zobrazí varovnú správu s textovým obsahom.

Je jasné, že aj keď je systém zásuvných modulov všeobecný, napokon musí dôjsť k vykonaniu konkrétnych úkonov s konkrétnymi triedami a volaniami konkrétnej knižnice *GUI*. Na tento účel slúži trieda `CGuiManager`. Objekt tejto triedy je pridružený k objektu triedy `CPluginManager`, a je tiež iba jeden v aplikácii. Jeho úlohou je interpretovanie všeobecných príkazov na príkazy priamo pre použité *GTK+*. Okrem toho si pamätá, čo ktorý zásuvný modul v *GUI*spravil.

5.2.2 Príkaz `metamodel`

Tento príkaz sa používa na prečítanie informácií o aktuálne načítanom metamodeli. Príkaz umožňuje získať všetky informácie, ktoré sú relevantné pre logickú časť logickej vrstvy architektúry aplikácie (viď. kapitola 3.4.2). Napriek tomu predpokladám, že väčšina zásuvných modulov sa bude zaujímať iba o to, či je načítaný práve ten jeden metamodel, pre ktorý sú vytvorené. Všetky ostatné potrebné informácie si ponesú v sebe.

Podtyp príkazu `metamodel` sa skladá z dvoch častí oddelených bodkou. Prvá časť špecifikuje typ prvku, na ktorý sa v príkaze pýtame:

- `metamodel` – informácie o metamodeli,
- `connection` – zoznam typov spojení alebo informácie o type spojenia (identita a doména),
- `element` – zoznam typov elementov alebo informácia o type elementu (doména, identita, s akými spojeniami sa môže viazať),
- `diagram` – zoznam typov diagramov alebo informácia o type diagramu (zoznam typov elementov a spojení, ktoré možno použiť),
- `domain` – zoznam typov domén alebo informácia o type domény (zoznam analyzátorov domény a zoznam atribútov domény).

Druhá časť môže byť:

- `list` – Pre získanie zoznamu prvkov daného typu. Príkaz nenesie žiadne parametre. Nie je možné získať zoznam načítaných metamodelov.
- `detail` – Pre získanie detailných informácií o jednom konkrétnom prvku. Príkaz obsahuje parameter `name` – názov typu.

Návratová hodnota (zoznam, alebo detail) je kódovaná s použitím syntaxe jazyka Python pre štandardné typy objektov (zoznam, slovník, n-tica, reťazec). To znamená, že knižnica pre zásuvný modul napísaná v Pythone ich môže opäť poskladať jediným príkazom `eval`.

5.2.3 Príkaz `exec`

Tento príkaz sa používa, ak chce zásuvný modul zavolať metódu nejakého objektu zo samotného modelu. Identifikátor objektu a názov metódy sú obsahom podpríkazu. Sú spojené znakom bodka.

Identifikátory objektov sú väčšinou tvorené znakom mriežky (#) a číselnou hodnotou. Takýto identifikátor majú všetky objekty, ktorých trieda je

potomkom triedy `Reference` (viď. obrázok 5.1). Vzhľadom k tomu, že v aplikácia môže môže iba jeden objekt triedy `CProject`, pristupuje sa k nemu pomocou identifikátora `project`.

Zoznam tried objektov a metód, ktoré môžu byť nad nimi zavolané vytvára *metatrieda*[1] `Meta`. Je vytvorený z tried, ktoré sa nachádzajú v balíčku `Classes` a sú potomkami triedy `IBase`.

Registrovanie objektov – trieda `Reference`

Trieda `Reference` je predkom pre všetky triedy, ktorých inštancie majú byť viditeľné zo zásuvných modulov. Výnimku tvorí trieda `CProject`, jej inštancia je prístupná cez rezervovaný identifikátor.

Trieda `Reference` má uložené v atribútoch triedy naposledy použitý identifikátor a slovník so slabými referenciami[2] na registrované objekty. Pri vytvorení novej inštancie tejto triedy sa zvýši hodnota naposledy použitého identifikátora a táto sa inštancii priradí. Zároveň sa inštancia zaregistruje. Okrem toho má metódu triedy `GetObject`, ktorá ako parameter berie identifikátor objektu a ak objekt existuje, vráti ho.

Definícia rozhrania – trieda `Meta`

Aby bolo možné ľahko vyhľadať, o ktorú metódu zásuvný modul žiada a aké má parametre, použil som *reflexiu*[3] a *metaprogramovanie*[4] – schopnosti jazyka Python. Definoval som triedu `Meta`, ktorá je metatriedou pre triedu `IBase` a triedy od nej odvodené. Okrem toho sú metódy týchto tried upravené *dekorátormi*[5], ktoré definujú spôsob spracovania jednotlivých parametrov a návratovej hodnoty.

Keď interpreter jazyka narazí na definíciu triedy, ktorá má ako metatriedu nastavenú triedu `Meta`, skôr ako spojí jednotlivé časti do novej triedy, vyvolá konštruktor triedy `Meta`. Súčasťou tohto konštruktora je vytvorenie samotnej

[1] Trieda, ktorej inštancie sú iné triedy. Jej konštruktor metatriedy sa vyvolá, keď normálna trieda vzniká v pamäti.

[2] Slabá referencia je špeciálna konštrukcia jazyka Python, ktorá zvyšuje efektivitu automatickej správy pamäte. Ak na nejaký objekt odkazujú už iba slabé referencie, objekt sa môže uvolniť z pamäte.

[3] Pozorovanie a modifikovanie počítačového programu sebou samým. Je to jedna z techník metaprogramovania[12]

[4] Programovacia technika, pri ktorej program modifikuje sám seba alebo iný program. Takýto program je schopný lepšie reágovať na zmeny a programátor musí zapísať menej kódu.[10]

[5] Dekorátory sú formou metaprogramovania. Sú to funkcie, ktoré upravujú alebo vylepšujú fungovanie iných funkcií alebo metód.

triedy, ale dôležitejšia časť je, že sa vytvorí mapa tejto triedy pre rýchly prístup v budúcnosti.

Trieda `IBase` a triedy od nej odvodené majú každá vlastný atribút triedy `__cls__`. Ten ukazuje na skutočnú triedu objektu, na ktorý sa mapuje. Napríklad `IElementVisual.__cls__` je nastavené na `CElement`, čo znamená, že k objektom triedy `CElement` budú zásuvné moduly pristupovať cez rozhranie definované triedou `IElementVisual`. Ak má niektorá trieda nastavenú hodnotu `__cls__` na `None`, nebude použitá v mapovaní, ale triedy od nej odvodené môžu dediť jej metódy.

Trieda `Meta` obsahuje atribút interface typu slovník (`dict`). Vzhľadom k tomu, že ide v skutočnosti o slovník slovníkov, ktorý obsahuje slovníky... celkovo v štyroch vrstvách, jednotlivé vrstvy sú popísane v zozname:

1. Kľúčmi sú skutočné triedy, na ktoré sú triedy rozhrania mapované.
2. Kľúčmi sú názvy metód triedy rozhrania. Okrem toho tu je ešte špeciálny kľúč `''__class__''`, ktorého hodnota je samotná trieda rozhrania.
3. Obsahuje kľúče:
   - `''result''` – hodnotou je modifikátor výsledku,
   - `''fname''` – hodnotou je skutočný názov metódy,
   - `''params''` – hodnotou je slovník parametrov metódy
4. Kľúčmi sú názvy parametrov danej metódy. Hodnotami sú modifikátory parametrov metódy.

Okrem vytvorenia zoznamu tried a metód rozhrania sa všetky metódy tried rozhrania transformujú na statické metódy. Dôvodom je fakt, že počas behu aplikácie nevzniká ani jedna inštancia triedy rozhrania. Pri vyvolávaní ich metód býva namiesto implicitného parametra[6] vložený skutočný objekt modelu. Preto som namiesto zaužívaného `self` pre názov prvého parametra použil názov `him`.

Modifikátory parametrov a návratovej hodnoty

Vzhľadom k tomu, že komunikačný protokol medzi aplikáciou a zásuvnými modulmi je textový, je potrebné transformovať hodnoty parametrov z textovej podoby na typ, aký očakávajú jednotlivé metódy. Opačne, výslednú hodnotu metódy treba opäť transformovať do textovej podoby.

[6]Pri volaní metódy inštancie býva ako prvý parameter implicitne vložený objekt samotný. Na rozdiel od väčšiny programovacích jazykov, je v Pythone nutné tento parameter explicitne deklarovať na prvom mieste v zozname parametrov metódy.

Na tento účel slúži skupina funkcií definovaných v module `ComSpec`. Sú implementované tak, aby sa správali rovnako, ako štandardné transformačné funkcie napríklad z textu na číslo. To znamená, že akceptujú jednu hodnotu a vrátia výsledok. V prípade nesprávnej hodnoty vyhodia výnimku `ValueError`.

Dekorátory metód

Dekorátor v Pythone je zavolateľný objekt (`callable`), ktorý ako parameter zoberie nejaký iný objekt (najčastejšie funkciu alebo metódu) a vráti ho buď pozmenený, alebo nejaký iný, ktorým má byť nahradený ten pôvodný. Pre účely registrácie tried rozhrania používam nad ich metódami niekoľko dekorátorov s nasledujúcim významom:

- `parameter` – pridá metóde informáciu o tom, akým modifikátorom bude spracovaný jeden z jej parametrov,
- `result` – pridá metóde informáciu o tom, akým modifikátorom bude spracovaná návratová hodnota
- `not_interface` – pridá metóde príznak, aby nebola zaradená medzi metódy rozhrania
- `factory` – takto označená metóda bude použitá v prípade, že zásuvný modul chce vytvoriť nový objekt. Metóda nie je konštruktorom, lebo nevytvára inštanciu triedy rozhrania, ale triedy, na ktorú je rozhranie mapované.

Vzdialené vyvolanie metódy – `Meta.Execute`

Pri spracovaní príkazu `exec` sa trieda `CCore` dostane iba po získanie objektu samotného s pomocou príslušného modifikátora. Potom vyvolá metódu triedy `Meta.Execute`. Tam sa prehľadáva slovník `Meta.interface`. Najskôr podľa triedy objektu, nad ktorým má byť vykonaná metóda. Potom podľa názvu metódy. Výsledkom je informácia, či je daná metóda prístupná zásuvnému modulu a popis jej parametrov.

Parametre príkazu, ktoré poslal zásuvný modul, sú usporiadané v slovníku, ale na začiatku sú všetky v textovom tvare. Preto musí prebehnúť na základe získaného popisu metódy transformácia hodnôt parametrov na príslušné typy. Okrem toho sa k parametrov pod kľúčom `''him''` priradí samotný objekt, nad ktorým má byť metóda vykonaná.

```
exec #45.GetDestination UML.FRI/0.1
__id__: 108


UML.FRI/0.1 205
__id__: 108
result: #26
```

Výpis 5.2: *Ukážka príkazu* `exec` *a odpovede vo forme návratovej hodnoty*

Pri vyvolaní metódy sa využíva syntaktická možnosť jazyka Python – rozbalenie slovníka na zoznam pomenovaných parametrov. Návratová hodnota metódy je potom transformovaná do textovej reprezentácie príslušným modifikátorom a takto vrátená ako výsledok volania `Meta.Execute`. Ten je potom odoslaný naspäť zásuvnému modulu.

Na výpise 5.2 je príkaz , ktorý poslal zásuvný modul aplikácii. Objekt identifikovaný ako `#45` je spojenie a zásuvný modul sa pýta aplikácie, do ktorého elementu spojenie smeruje. Po príkaze nasleduje odpoveď aplikácie, že cieľovým elementom je objekt s identifikátorom `#26`. Všimnite si previazanie príkazu a odpovede pomocou parametra `__id__`

5.3 Distribúcia udalostí medzi časťami aplikácie

Doteraz si v aplikácii zasielali signály medzi sebou iba jednotlivé komponenty GUI. Napríklad, ak používateľ označil nejaký element na kresliacom plátne, kresliace plátno vyslalo signál, že sa zmenil stav označenia. Táto správa je zaujímavá napríklad pre bočný panel, na ktorom sú zobrazené hodnoty atribútov. Preto, keď bočný panel zachytí takýto signál, prepíše svoj obsah aktuálnymi hodnotami.

Teraz je však už potrebné o niektorých udalostiach informovať aj zásuvné moduly a niektoré akcie zásuvných modulov môžu vyžadovať, aby na ne patrične reágovalo *GUI*.

Vznikli z toho dva problémy, ktoré bolo potrebné vyriešiť:

- Väčšina udalostí mohla doteraz vzniknúť iba v jedinom komponente, čo predstavuje komunikáciu 1:N. Po novom je však viacero signálov šírených spôsobom M:N.

- Udalosti sa šírili iba v GUI. Preto boli implementované špecificky pre

GTK+. Systém zásuvných modulov však má byť nezávislý od tejto vrstvy.

Riešením prvého problému bolo vytvorenie triedy `CBus`. Aplikácia obsahuje jednu inštanciu tejto triedy. Táto inštancia predstavuje „signalizačnú zbernicu" aplikácie. Ak komponent, namiesto toho, aby vyslal signál o udalosti sám, použije zbernicu, opäť sa komunikácia vráti k spôsobu 1:N.

Triedu `CPluginAdapter` som vytvoril na odtienenie závislosti GUI na GTK+. V aplikácii je opäť iba jedna inštancia tejto triedy. Táto „počúva" signály o udalostiach na „zbernici" a preposiela ich do systému zásuvných modulov. Na druhú stranu, pre tento systém poskytuje rozhranie, ktorým môže zásuvný modul informovať zvyšok aplikácie ak spôsobil nejakú udalosť. Adaptér túto udalosť preloží do formy signálu a vyšle ho na „zbernicu".

5.4 Knižnica pre zásuvný modul

V tejto časti popisujem knižnicu, ktorá je určená pre zásuvné moduly napísané v programovacom jazyku Python. Pri jej implementácii som sa snažil čo najviac využiť už existujúce triedy, ktoré som vytvoril pre samotú aplikáciu. Dôvodom bolo zabránenie vzniku duplicít tried rozhrania. Ak by ich potom bolo nutné definovať na dvoch miestach, definície by museli byť totožné. Chcel som takto predísť anomáliám, ktoré by mohli vzniknúť pri rozširovaní alebo úprave rozhrania, ak by výsledok nebol totožný na oboch miestach.

Rozhranie knižnice smerom k zásuvnému modulu simuluje objektovú štruktúru aplikácie tak, ako ju má zásuvný modul vidieť. Tieto objekty dávajú prístup iba k metódam, ktoré presmerujú do aplikácie a odpoveď aplikácie vrátia ako návratovú hodnotu metódy. Programátor je takto odtienený od skutočnosti, že zásuvný modul nemá priamy prístup do aplikácie a že pracuje v samostatnom procese.

Všimnite si ešte raz obrázok 5.1. Knižnica je navrhnutá tak, aby programátor zásuvného modulu na začiatku dostal prístup iba k jedinému objektu triedy `CInterface`. Táto poskytuje rozhranie ku všetkým príkazom, ktoré môže zásuvný modul aplikácii zadať. Zapúzdruje objekt triedy `CConnection`, ktorá tvorí vrstvu parsovania na spojení medzi aplikáciou a zásuvným modulom. Pri jednotlivých volaniach sa však transparentne dostane k objektom ďalších tried z balíčka `Client`.

5.4.1 Trieda `CProxy` a jej potomkovia

Keby ste sa pozreli na zdrojový kód triedy `CProxy`, zdal by sa vám až príliš malý. Navyše, definície tried jej potokov by ste ani vôbec nenašli. Napriek

tomu sa v zásuvných moduloch aktívne používajú. V tomto prípade som opäť využil *metaprogramovanie*.

V čase, keď trieda `Meta`, ktorá je `metatriedou` pre triedy definujúce rozhranie aplikácie, tieto triedy vytvára, vytvorí k nim párové triedy. Tieto sú všetky odvodené od triedy `CProxy`. Umiestni ich do balíčka `Client` a ako atribút triedy im pridá odkaz na seba samú.

Triedy, ktoré takto vznikli, v skutočnosti nemajú žiadnu z metód, ktoré používa zásuvný modul. Trieda `CProxy` implementuje metódu `__getattr__`. Táto je vyvolaná v prípade, že program chce pristupovať k atribútu/metóde, ktorú objekt nemá. Objekt v nej získa z triedy `Meta` informácie o metóde s rovnakým názvom, aký chcel program použiť. Metóda `CProxy.__getattr__` vráti inicializovaný objekt triedy `CCallable`. Opäť som tu využil *reflexiu*.

5.4.2 Trieda `CCallable`

Trieda `CCallable` implementuje metódu `__call__`, to znamená, že tento objekt tejto triedy môže byť použitý na mieste funkcie alebo metódy. Môžu byť preto použité ako metódy objektov tried odvodených od `CProxy`.

Keď program túto „metódu" zavolá, volanie sa presmeruje do nižšej vrstvy komunikácie (do objektu triedy `CConnection`) a prepošle sa do aplikácie. Príkaz je blokujúci a čaká na odpoveď aplikácie. Pre programátorom zásuvného modulu sa takto skryje fakt, že volanie metódy je asynchrónne.

5.5 Štartovanie zásuvných modulov

Všetka funkcionalita, ktorú som popísal doteraz, predstavovala pre aplikáciu rozhranie pre vzdialené vyvolanie metód. Nerobilo to z nej však systém pre podporu zásuvných modulov. Chýbal mu mechanizmus, ktorý by zásuvný modul samostatne spustil a odovzdal mu parametre, ako sa má k aplikácii pripojiť.

Aplikácia má zatiaľ implementovaný mechanizmus iba pre automatické štartovanie zásuvných modulov, ktoré sú napísané v programovacom jazyku Python. Zásuvný modul musí byť balíček jazyka Python, z ktorého sa dá importovať funkcia `main`. Táto funkcia akceptuje jeden parameter, inicializovanú a pripojenú inštanciu triedy `CInterface`. V tejto funkcii sa zásuvný modul inicializuje. Ak chce zásuvný modul fungovať čisto reaktívne – na podnety z aplikácie, môže túto funkciu opustiť, lebo zásuvný modul sa vypne až v momente, keď dôjde k prerušeniu spojenia s aplikáciou.

Zásuvný modul musí byť umiestnený v adresári `etc/plugins`. Tento aplikácia prehľadáva hneď po tom, ako vytvorí svoje *GUI*. Pre každý nájdený

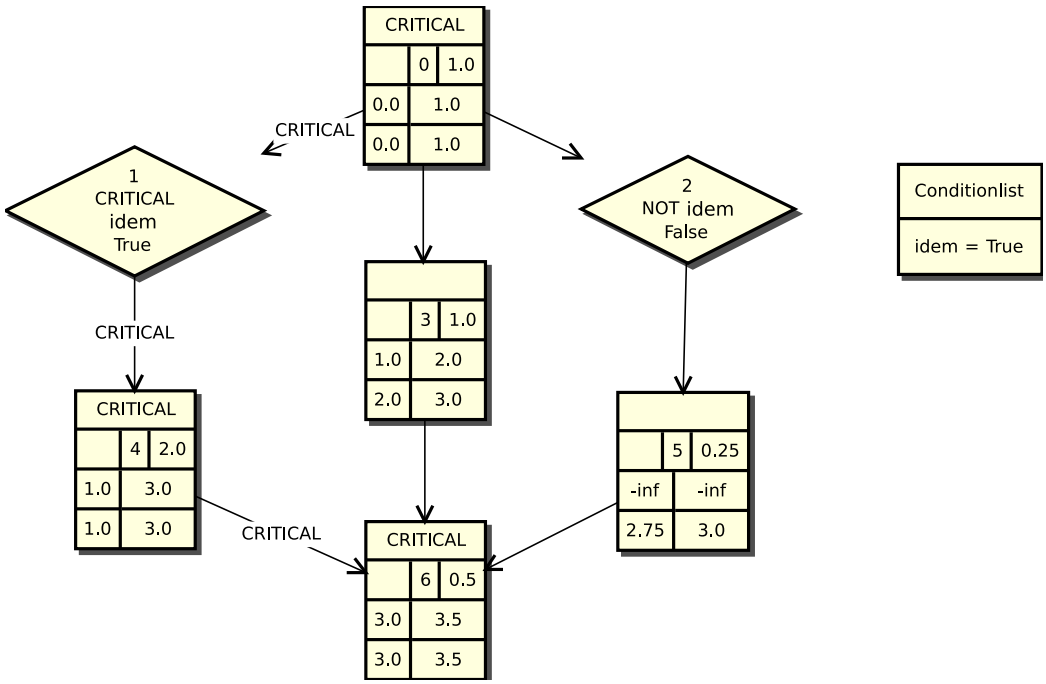
balíček spustí program `pl_runner.py`, ktorý je v spoločnom adresári ako hlavný vykonateľný súbor aplikácie. Tento program akceptuje dva parametre – číslo portu, na ktorom aplikácia počúva prichádzajúce spojenia a názov zásuvného modulu, ktorý sa má pripojiť. V programe sa následne vytvorí objekt triedy `CInterface`, ktorý sa pripojí k aplikácii. Potom importuje požadovaný zásuvný modul ako balíček (package) a vyvolé jeho metódu `main`.

Pri vykonávaní programu `pl_runner.py` sa mi však nepodarilo zachovať úplnú nezávislosť na použitom operačnom systéme. Chcem totiž, aby sa proces zásuvného modulu vykonával na pozadí. Používam teda iný spôsob volania na operačnom systéme Windows a iný na ostatných. V oboch prípadoch vyvolám nový proces pomocou systémového volania `os.system`. Jeho parametrom je text, ktorý sa má vykonať vo virtuálnom príkazovom riadku.

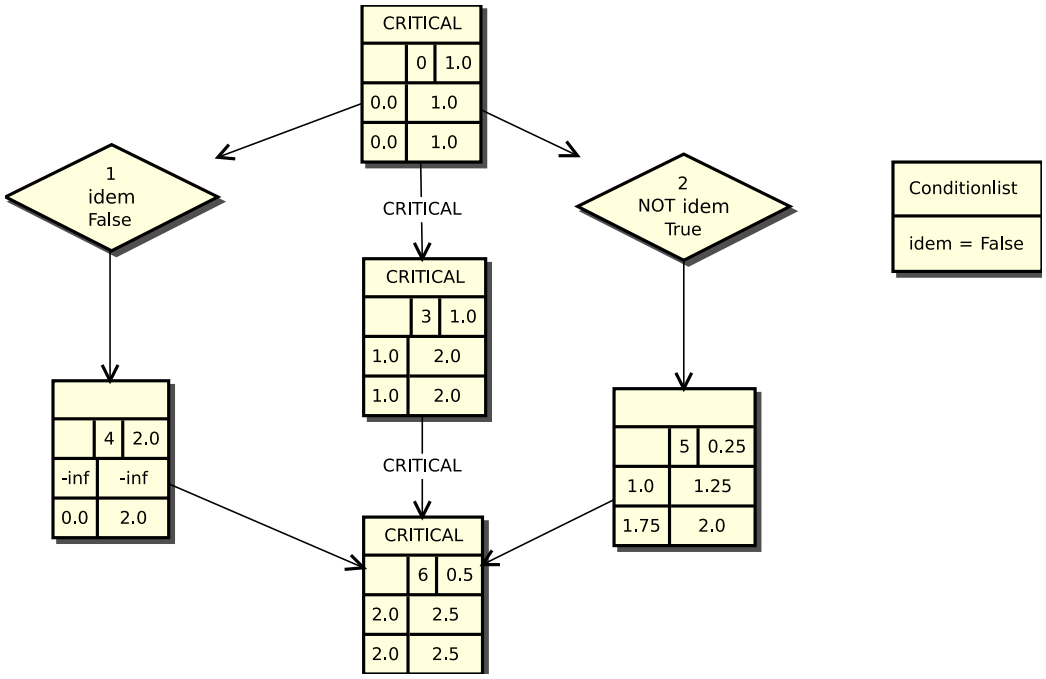
- Na operačnom systéme Windows spúšťam proces pomocou príkazu `start` s parametrom `/B`.
- Na ostatných operačných systémoch spúšťam proces s posledným parametrom `&`.

Kapitola 6

Ukážkový zásuvný modul – Hľadanie kritickej cesty v sieťovom grafe



Obr. 6.1: *Príklad sieťového grafu*



Obr. 6.2: *Ten istý sieťový graf so zmenenou podmienkou*

Jedným z cieľov diplomovej práce bola tvorba ukážkového zásuvného modulu, na ktorom bude preukázané, že je systém zásuvných modulov funkčný. Na základe odporúčania vedúceho diplomovej práce som implementoval vo svojom zásuvnom module algoritmus hľadania kritickej cesty.

Na implementáciu som použil rozšírený sieťový graf, doplnený o podmienky. Ukážka sieťového grafu s podmienkou je na obrázku 6.1. Ten istý sieťový graf, avšak s obrátenou hodnotu podmienky je na obrázku 6.2.

6.1 Algoritmy použité v zásuvnom module

Pri tvorbe zásuvného modulu bolo potrebné implementovať algoritmus na monotónne očíslovanie vrcholov v digrafe a algoritmus na nájdenie kritickej cesty v sieťovom grafe s podmienkami.

Digraf je taká usporiadaná dvojica $G = (V, H)$ kde V je neprázdna množina vrcholov grafu a H je množina usporiadaných dvojíc typu (u, v), kde $u \neq v$, t.j. $H \subseteq \{(u, v)|u \neq v; u, v \in V\} = V \times V$ [6]

Monotónne očíslovanie vrcholov digrafu $G = (H, V)$ je také očíslovanie vrcholov $v_1, v_2, \ldots v_n$, pre ktoré platí $[v_i, v_k] \in H \rightarrow i < k$ V prípade, že digraf obsahuje orientovaný cyklus, digraf nie je možné monotónne očíslovať. [6].

Pri reprezentácii technologického postupu pomocou sieťového grafu (digraf bez orientovaného cyklu), som použil vrcholovo orientovaný prístup. Vrchol teda reprezentuje činnosť a orientované hrany reprezentujú vzájomné závislosti medzi činnosťami. To znamená, že ak sa v grafe nachádza hrana *(u, v)*, tak činnosť reprezentovaná vrcholom v nesmie začať skôr ako skončí činnosť reprezentovaná vrcholom u.

Keď rozšírime sieťový graf o podmienky, dodáme doň špeciálny typ vrchola. Podmienku si možno predstaviť ako činnosť, ktorá má nulovú dobu trvania v prípade, že podmienka platí. V prípade, že podmienka neplatí, doba jej trvania nadobudne hodnotu záporného nekonečna.

Diagramy oboch algoritmov sú na obrázku 6.3. Použité premenné v algoritmoch majú nasledujúci význam:

- `V` – množina vrcholov v grafe
- `H` – množina hrán v grafe
- `inf` – hodnota nekonečno
- `v.ideg` – počet hrán vstupujúcich do vrcholu
- `v.odeg` – počet hrán vystupjúcich z vrchola

- `v.duration` – trvanie aktivity reprezentovanej vrcholom grafu
- `v.minstart`, `v.minend`, `v.maxstart`, `v.maxend` – hodnoty najskoršieho a najneskoršieho možného začiatku a konca aktivity.

6.2 Implementácia zásuvného modulu

Celá práca na ukážkovom zásuvnom module spočívala v dvoch fázach:

1. Tvorba nového metamodelu.
2. Programovanie zásuvného modulu.

Nový metamodel bol potrebný vzhľadom k tomu, že nástroj UML .FRI doteraz neobsahoval nič, čo by sa podobalo na sieťový graf. Metamodel obsahuje jeden typ diagramu – Sieťový diagram. Na ten sa dajú vkladať tri typy elementov:

- Aktivita – predstavuje štandardnú činnosť.

  Používateľ by mal vyplniť predovšetkým dĺžku trvania aktivity (duration). Taktiež môže vyplniť názov aktivity (name). Všetky ostatné atribúty automaticky nastaví zásuvný modul.

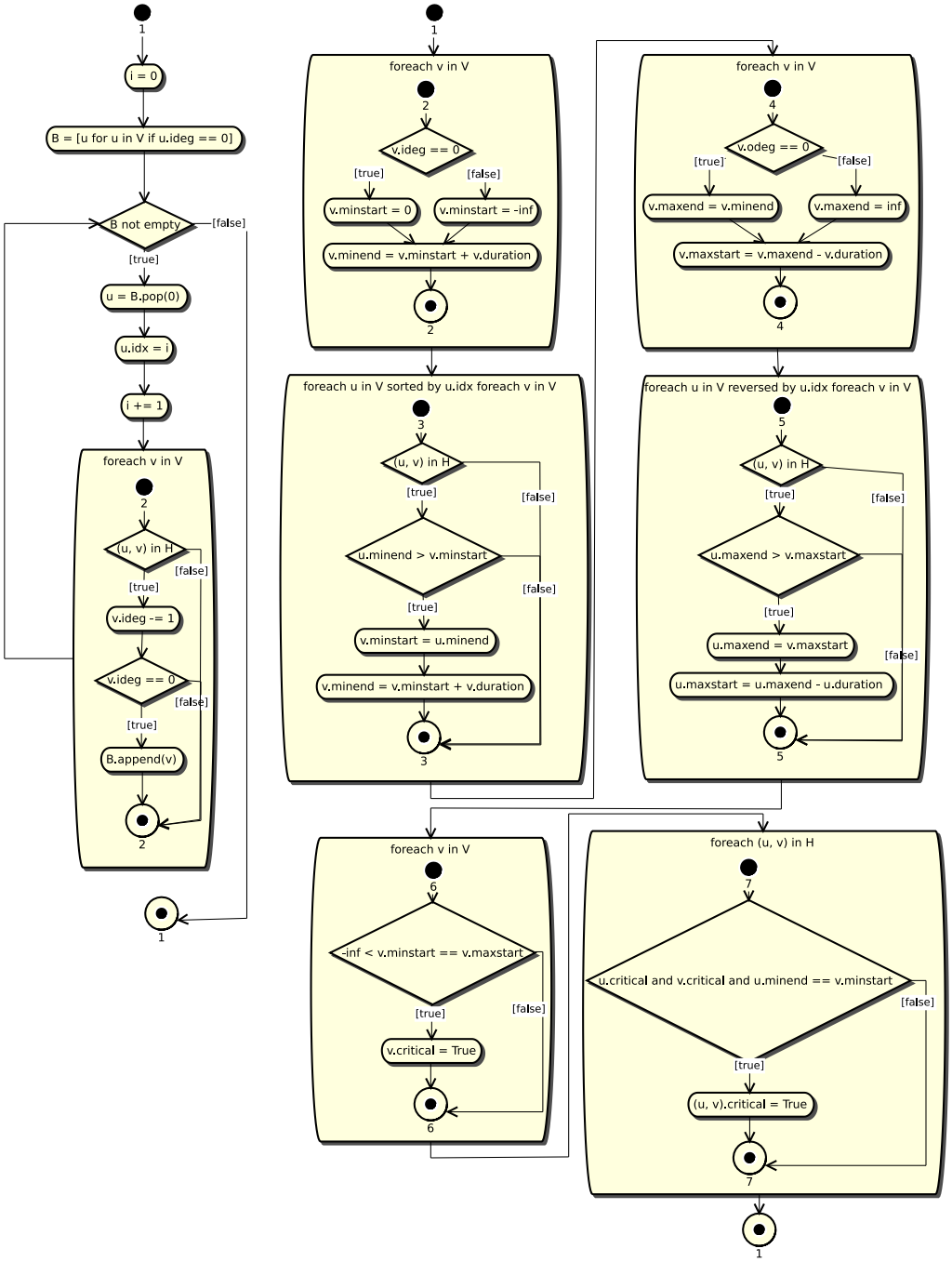
- Zoznam hodnôt podmienok – obsahuje názvy a hodnoty podmienok.

  Používateľ dopĺňa prvky do zoznamu podmienok. Názov (name) je identifikátorom podmienky. Hodnota (value) obsahuje hodnotu podmienky.

- Podmienka – špeciálny typ aktivity (viď. predošlá kapitola)

  Používateľ musí zadať identifikátor podmienky (condition) a môže zmeniť hodnotu negácie. Hodnoty ostatných atribútov nastaví zásuvný modul.

Diagram obsahuje iba jeden typ spojenia – prechod (transition) predstavujúci priamu závislosť medzi aktivitami. Atribút spojenia (critical) nastavuje automaticky zásuvný modul.



Obr. 6.3: *algoritmus monotónneho očíslovania vrcholov v digrafe a algoritmus hľadania kritickej cesty v sieťovom grafe*

Programovanie zásuvného modulu spočívalo v naprogramovaní balíčka v jazyku Python, ktorý implementuje predovšetkým algoritmy z obrázka 6.3. To však samozrejme nie je všetko. Zásuvný modul počas svojej incializácie požiada aplikáciu o doplnenie hlavného menu o svoju položku, pomocou ktorej sa spúšťa hlavný algoritmus.

Keď používateľ klikne na *„Find critical path"*, aktivuje tým hlavný algoritmus zásuvného modulu. V tom sa vykoná nasledujúca postupnosť krokov:

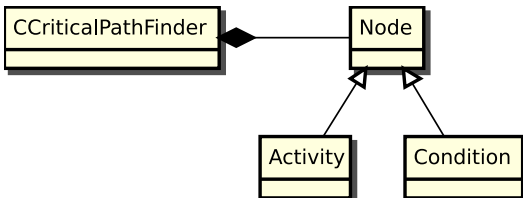
1. Kontrola, či je otvorený projekt so správnym metamodelom a či je otvorený diagram so sieťovým grafom. Ak nie, zobrazí sa výstražné okno a algoritmus nepokračuje.

2. Čítanie zoznamu elementov a spojení na diagrame a vytvorenie internej objektovej štruktúry reprezentujúcej aktuálny sieťový graf.

3. Vykonanie algoritmu na monotónne očíslovanie vrcholov v grafe. Ak sa nepodarí monotónne očíslovať všetky vrcholy, znamená to, že v grafe je orientovaný cyklus.

4. Vykonanie algoritmu na nájdenie kritickej cesty v sieťovom grafe.

5. Premietnutie výsledkov v aplikácii.



Obr. 6.4: *Diagram tried zásuvného modulu*

Vnútorná štruktúra sa skladá z tried zobrazených na obrázku 6.4.

Kapitola 7

Záver

Na začiatku tejto diplomovej práce bol cieľ umožniť rozširovanie aplikácie UML .FRI aj pomocou zásuvných modulov. Jedna podmienka bola, aby nový systém fungoval na všetkých platformách, na ktorých doteraz fungovala aplikácia. Tiež mal umožniť, aby zásuvné moduly mohli byť vyvíjané v rôznych programovacích jazykoch. A napokon som mal na príklade ukázať, že systém skutočne funguje.

Keď vedúci mojej diplomovej práce vypísal tému, očakával, že na jej základe bude môcť vypísať celú sériu ďalších prác s rôznou tématikou. Napríklad programovanie algoritmov z teórie grafov, pričom sa na kreslenie grafov použije nástroj UML .FRI. Alebo tvorba modulov, ktoré umožnia generovanie zdrojových kódov z diagramov UML. Nápady na rôzne témy sa postupne vynárajú aj teraz.

Na tomto mieste môžem konštatovať, že svoje ciele, aj očakávania vedúceho svojej práce, sa mi podarilo splniť. Vytvoril som ukážkový zásuvný modul, ktorý hľadá kritickú cestu v sieťovom grafe. Na tomto príklade je vidieť, že systém pre podporu zásuvných modulov funguje a dá sa používať.

Moja práca sa však ešte stále nekončí. Súbežne vytváral svoju diplomovú prácu aj môj kolega na tému „Návrh a implementácia systému Undo/Redo[1] do aplikácie UML .FRI". Obaja sme vyvíjali svoje vetvy oddelene od hlavnej aj od seba navzájom. Našou spoločnou úlohou v najbližšom čase bude zaradenie týchto nových systémov do hlavnej vetvy a ich vzájomné prepojenie. Spoločným termínom celého tímu okolo UML .FRI na vydanie novej verzie, je september 2009. Vtedy začne nasledujúci akademický rok a my všetci dúfame, že si v ňom UML .FRI získa pevné postavenie medzi učiteľmi aj študentmi Fakulty riadenia a informatiky. Verím, že som svojou diplomovou prácou k tomu cieľu výrazne pomohol.

[1]Operácia *Undo* znamená vrátenie predchádzajúcej používateľovej akcie. Operácia *Redo* zruší predchádzajúcu *Undo* operáciu.

Referencie

[1] Bachratý, Hynek – Sadloň, Ľubomír: *Modifikácia zovšeobecnených sieťových grafov.* 2005. nepublikované.

[2] —: Výpočet prevádzkových intervalov pomocou sieťových grafov. In: *INFOTRANS 2005.* Pardubice : DPJF Univerzita Pardubice, september 2005. ISBN 80-7194-792-X.

[3] Bača, Tomáš – Juríček, Milan – Odlevák, Pavol: *Case tool development.* In: *Journal of Information, Control and Management Systems.* 2008. vol.6, n°1. ISSN 1336-1716.

[4] Janech, Ján – Bača, Tomáš – Odlevák, Pavol a i.: *Projektová dokumentácia.* 2008. nepublikované.

[5] Lukáč, Marek: *Softvérový nástroj na návrh a výpočty sieťových grafov s podmienkovými hranami.* 2005.

[6] Palúch, Stanislav: *Skriptá z teórie grafov.* [online]. 2001. Dostupné online: <http://frcatel.fri.uniza.sk/users/paluch/grafy-pdf-pics.zip>.

[7] *RFC 2822 – Internet Message Format.* 2001. Dostupné online: <http://www.ietf.org/rfc/rfc2822.txt>.

[8] Ružbarský, Ján: *Základy UML.* [online]. 2008. Dostupné online: <http://kst.uniza.sk/predmety/uml/>.

[9] Wikipédia: *XML.* [online]. 2008. prístup 19-december-2008. Dostupné online: <http://sk.wikipedia.org/XML>.

[10] —: *Metaprogramming.* [online]. 2009. prístup 12-máj-2009. Dostupné online: <http://en.wikipedia.org/wiki/Metaprogramming>.

[11] —: *Plug-in.* [online]. 2009. prístup 12-máj-2009. Dostupné online: <http://en.wikipedia.org/wiki/Plugin>.

[12] —: *Reflection (computer science)*. [online]. 2009. prístup 12-máj-2009. Dostupné online: <http://en.wikipedia.org/wiki/Reflection_(computer_science)>.

[13] —: *URI*. [online]. 2009. prístup 12-máj-2009. Dostupné online: <http://en.wikipedia.org/wiki/URI>.

[14] —: *URI*. [online]. 2009. prístup 12-máj-2009. Dostupné online: <http://en.wikipedia.org/wiki/URL>.