VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

LEHCE ROZŠIŘITELNÝ GRAFOVÝ EDITOR

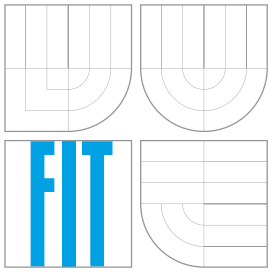
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

FRANTIŠEK SYSEL

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

LEHCE ROZŠIŘITELNÝ GRAFOVÝ EDITOR

EASILY UPGRADABLE GRAPH EDITOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — FRANTIŠEK SYSEL
AUTHOR

VEDOUCÍ PRÁCE — Ing. ALEŠ SMRČKA, Ph.D.
SUPERVISOR

BRNO 2011

Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací editoru grafů. Důraz je kladen na rozdělení celé aplikace do samostatných modulů, které je možno nahradit jinými moduly, či rozšířit aplikaci o nové moduly a tím doplnit její funkčnost. Teoretická část práce se zabývá vybranými technikami pro vývoj modulárních aplikací a praktická popisuje návrh a implementaci jednotlivých zásuvných modulů programu. Součástí práce jsou i automatické testy určené pro vývoj nových modulů.

Abstract

The bachelors thesis deals with design and implementation of graph editor. The emphasis is put on division of the whole application into independent plug-ins which can be replaced by other plug-ins or extended with new plug-ins and thus to add its functionality. Theoretical part deals with techniques for development of module applications while practical part describes design and implementation of particular plug-in modules of a program. The thesis also contains automatic tests for module development.

Klíčová slova

Modulární systém, zásuvný modul, graf, teorie grafů, vykreslení grafu.

Keywords

Modular system, plug-in, graph, graph theory, graph drawing.

Citace

František Sysel: Lehce rozšiřitelný grafový editor, bakalářská práce, Brno, FIT VUT v Brně, 2011

Lehce rozšiřitelný grafový editor

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, PhD.

.....
František Sysel
12. května 2011

Poděkování

Rád bych poděkoval vedoucímu mé práce panu Ing. Aleši Smrčkovi, PhD. za odborné vedení a za čas věnovaný konzultacím.

Obsah

Kapitola 1

Úvod

Graf je důležitý pomocník v mnoha oborech lidské činnosti. Používá se k přehlednému vyjádření vztahů mezi entitami, přičemž grafická podoba těchto vztahů je pro člověka mnohem srozumitelnější a názornější, než jen strohý textový popis. Grafem popisujeme elektrické obvody, chemické vzorce, rodokmeny. S rozvojem softwarového inženýrství se začala používat téměř desítka různých typů grafů, pomocí nichž se modelují objekty reálného světa a popisují programové procesy.

Pokud vytváříme graf, musíme si zodpovědět na otázku, jaký účel bude graf, který chceme vytvořit, mít a podle toho zvolit nástroj, jenž budeme pro vytváření tohoto grafu používat. Zda zvolit jednoúčelový editor pro konkrétní typ grafu, který disponuje editačními funkcemi šitými na míru danému typu grafů, nebo univerzální editor. Dále musíme při volbě nástroje zvážit, zda umí exportovat výsledný graf do vektorových formátů, které můžeme snadno vložit do dokumentů vysázených v typografických systémech, nebo do negrafické podoby, například zdrojového kódu programovacího jazyka.

Tématem této bakalářské práce je navrhnout a vytvořit aplikaci určenou pro vytváření a úpravu grafů, kterou bude možno rozšířit o nové funkce v podobě podpory nových typů grafů, nových formátů pro import a export, ale také odlišných uživatelských rozhraní. Aplikace bude umět načítat, upravovat a ukládat grafy, dále bude umět exportovat grafy do obecných bitmapových a vektorových grafických formátů a krom toho bude umět pracovat v textovém režimu, tzn. bude ji možno použít i ve skriptech pro shell.

Práce je rozčleněna do čtyř částí. Druhá kapitola slouží jako úvod do problematiky teorie grafu, dále se zabývá výčtem a popisem funkcí některých volně dostupných grafových a vektorových programů a technologiemi používanými pro vytváření modulárních aplikací a zásuvných modulů. Třetí kapitola popisuje podrobný návrh modulárního aplikačního systému. Zaměřuje se na popis jádra aplikace, aplikačního rozhraní a způsobu komunikace mezi jednotlivými moduly. Ve čtvrté kapitole jsou rozebrány implementační detaily jednotlivých částí aplikace. Pátá kapitola je věnována testování modulů. Testy jsou zvoleny tak, aby je mohli použít vývojáři nových modulů. Závěrečná kapitola zhodnocuje dosažené výsledky a zabývá se možnostmi dalšího vývoje.

Kapitola 2

Grafové editory a možnosti jejich rozšíření

Celá práce se zabývá vytvářením grafů, proto považuji za užitečné hned na začátku vysvětlit, co je to graf. Informace jsem čerpal z [2, 4].

Po stručném teoretickém úvodu se věnuji zajímavým funkcím vybraných volně dostupných editorů (freeware, opensource, webových). Kromě vektorových a grafových editorů se v této části věnuji i nástrojům na vytváření UML diagramů, protože univerzální grafové editory zpravidla neumějí generovat zdrojové kódy z vytvořených UML diagramů a naopak z kódů generovat grafy, a jednomu zástupci simulačních nástrojů. O komerčních produktech (např. o univerzálním grafovém editoru Microsoft Visio nebo simulačním nástroji Dymola) se nebudu zmiňovat, protože jejich výčet by byl nad rámec kapitoly.

Poslední část kapitoly popisuje možnosti rozšíření programů. Zabývá se obecnými technologiemi používanými v modulárních aplikacích (např. CORBA a COM) a problematikou návrhu zásuvných modulů v jazycích C a C++, o kterých jsem čerpal informace z [8]. Volba není náhodná, neboť chci tyto jazyky použít pro vývoj grafového editoru, jenž by měl být výsledným produktem této práce.

2.1 Stručný úvod do teorie grafů

Graf slouží jako forma popisu vztahů mezi objekty. Objekty nazýváme *vrcholy* nebo také *uzly*, graficky se mohou značit například bodem. Spojením vrcholů pomocí přímky nebo křivky vyjadřujeme určitý vztah mezi těmito vrcholy. Toto spojení se nazývá *hrana*. Hrany mohou být *neorientované* či *orientované*. U orientovaných rozlišujeme počáteční a koncový vrchol. Hranám můžeme přiřadit hodnotu, pak rozlišujeme *ohodnocený* a *neohodnocený* graf. Obsahuje-li graf násobné hrany (tzn. mezi dvěma vrcholy existuje více jak jedna hrana), mluvíme o *multigrafu*.

Existují dvě rozšířené formální definice grafu. První, jednodušší, je dvojice $G = (V, E)$, kde V je neprázdná konečná množina vrcholů (uzlů) a $E \subseteq V \times V$ je množina hran. Tato definice však umožňuje definici multigrafu pouze v případě, že E je multimnožina. Další definice, univerzálnější, je $G = (V, E, \varepsilon)$, kde V je neprázdná konečná množina vrcholů, E je konečná množina hran a ε je zobrazení $\varepsilon : E \to V \times V$, které nazýváme vztahem *incidence*. Takový graf se nazývá *obecným grafem* a může popisovat graf s orientovanými i neorientovanými hranami. Zobrazení ε přiřazuje každé hraně e ∈ E v orientovaném grafu uspořádanou dvojici vrcholů (x, y). Vrchol x nazýváme počátečním vrcholem hrany a vrchol

y koncovým (závisí na pořadí). V neorientovaném grafu existuje pro každou hranu $e_1 \in E, \varepsilon(e_1) = (x, y)$, hrana $e_2 \in E$ taková, že $\varepsilon(e_2) = (y, x)$. V této bakalářské práci si ovšem vystačíme s jednou hranou, pro kterou má dvojice vrcholů (x, y) stejný význam jako (y, x).

Cesta grafem je posloupnost vrcholů a hran $v_0, e_1, v_1, e_2, v_2, ..., e_k, v_k$, kde platí, že se žádný vrchol neopakuje. *Cyklus* (kružnice) je *uzavřená cesta*, v níž se neopakují žádné vrcholy, kromě prvního a posledního (platí $v_0 = v_k$). Existuje-li mezi libovolnými dvěma vrcholy cesta, jedná se o *souvislý graf*. V opačném případě jde o *nesouvislý graf*.

To jsou pojmy, které v této práci používám a které jsou důležité pro její pochopení. Existuje ovšem celá řada dalších pojmů, které si může čtenář nastudovat z citované literatury.

2.2 Editory a jejich funkce

2.2.1 Dia

Univerzální grafový editor Dia[1], který patří do projektu GNOME, slouží pro tvorbu široké škály grafů, kterou je možné ještě rozšířit pomocí balíčků obsahujících další tvary. Editor Dia je napsán v jazyce C a může být rozšířen jak moduly napsanými v jazyce C, tak i moduly v Pythonu. Rozšířením napsaným v jazyce C se budu více věnovat ve třetí části této kapitoly.

2.2.2 Karbon14

Vektorový editor Karbon14 (dříve KIllustrator) je součástí kancelářského balíku KOffice[2], který je určen pro desktopové prostředí KDE. V balíku KOffice by se měl do budoucna objevit ještě program Kivio, který by měl sloužit k tvorbě diagramů.

Karbon14 má téměř stejné funkce jako konkurenční OpenOffice.org Draw z kancelářského balíku OpenOffice.org. Umí vytvářet vrcholy a spojovat je hranami. Výhodou těchto programů je snadný přenos výsledných grafů do dokumentů tvořených ostatními nástroji kancelářského balíku. Nevýhoda spočívá především v absenci většího výběru typů grafů.

2.2.3 yEd

Program yEd[3] je freewarový editor napsaný v Javě. Nabídkou funkcí se řadí na pomyslný vrchol volně dostupných grafových editorů.

Pomocí palety nástrojů můžeme vytvářet UML diagramy, vývojové diagramy a další typy grafů. Poklepáním na vrchol je možné vkládat do vrcholu text, ostatní vlastnosti (barva, tloušťka čar) se nastavují v prohlížeči vlastností. Editor svými funkcemi, zejména napovídáním vhodných bodů a přímek při přesouvání objektů po výkresu a jejich přimykáním, usnadňuje a urychluje tvorbu grafů. Uživatel pomocí těchto funkcí může vytvářet vrcholy ve stejných vzdálenostech, aniž by musel používat funkce pro zarovnání či informace o poloze objektů. Program umožňuje přerovnat rozložení vrcholů a vedení hran do různých seskupení (např. do mřížky nebo do kruhů) a analyzovat základní matematické vlastnosti grafu (acyklicita, rovinnost).

[1] http://live.gnome.org/Dia/
[2] http://www.koffice.org/
[3] http://www.yworks.com/en/products_yed_about.html/

2.2.4 Umbrello

UML nástoj Umbrello[4] je určen pro tvorbu ERD, diagramů tříd a případů užití. Velmi silnou stránkou programu je podpora téměř dvou desítek programovacích jazyků, import zdrojových kódů, ze kterých se vygenerují diagramy, a export, který vytvoří kostry tříd ve zvoleném jazyce.

2.2.5 ArgoUML

ArgoUML[5] je nástroj napsaný v Javě podobný programu Umbrello. Také disponuje funkcemi pro tvorbu UML diagramů, nepodporuje však tak velké množství programovacích jazyků, do kterých je možné generovat kostry programů.

ArgoUML je postaven na modulárním systému používajícím mechanismus zásuvných modulů integrovaného vývojového prostředí Eclipse (OSGi/Equinox), o kterém se zmíním ve třetí části kapitoly.

2.2.6 Gliffy

Gliffy[6] je komerční editor založený na technologii Adobe Flash[7] a Java Scriptu. Nabízí podobné možnosti jako Microsoft Visio – neomezuje se jen na tvorbu grafových editorů, ale je možno pomocí něj vytvářet i mapy podlaží nebo dialogová okna. Program je možno bezplatně vyzkoušet, nicméně funkčnost je omezena pouze na tvorbu výkresu. Pro ukládání či sdílení dokumentu je nutno se zaregistrovat a zaplatit za licenci.

2.2.7 Qfsm

Qfsm[8] je nástroj pro tvorbu a simulaci konečných automatů. Simulační nástroje používají grafy k vizualizaci, či k snadnějšímu popisu a tvorbě simulačního modelu. V případě programu Qfsm uživatel vytváří konečný automat přímo jako graf, může definovat vstupy, výstupy a poté simulovat jeho činnost.

Program umí z grafu konečného automatu vygenerovat obrázek, tabulku stavů a kód automatu ve VHDL a v jiných jazycích určených k popisu hardwaru.

2.2.8 Graphviz

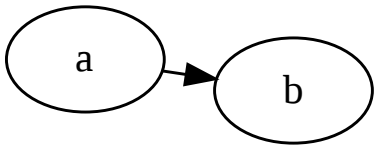
Graphviz[9] je skupina nástrojů pro vizualizaci grafů. Největší výhodou těchto nástrojů je, že pracují v textovém režimu, a proto mohou být spouštěny v rámci skriptů nebo je mohou spouštět jiné programy, které potřebují vytvořit nějaký graf, na pozadí (např. Doxygen[10] – program na generování dokumentace ze zdrojových kódů).

Grafy se popisují pomocí příkazů jazyka DOT[11], ze kterých je posléze vygenerován výstup ve formě bitmapového obrázku, vektorového obrázku nebo dokumentu určeného pro tisk (postscript, pdf).

[4] http://www.kde.org/applications/development/umbrello/
[5] http://argouml.tigris.org/
[6] http://www.gliffy.com/
[7] http://www.adobe.com/products/flash/
[8] http://qfsm.sourceforge.net/
[9] http://www.graphviz.org/
[10] http://www.doxygen.org/
[11] http://www.graphviz.org/content/dot-language

Na obrázku 2.1 je graf vygenerován programem `neato`, vytvořen příkazem `digraph G {a->b}`. Celý příkaz může v prostředí unixového shellu vypadat takto:

```
neato -Tpdf > graf.pdf <<< "digraph G {a->b}"
```



Obrázek 2.1: Jednoduchý graf vygenerovaný programem `neato`.

2.3 Způsoby rozšíření programů

Rozšíření programů pomocí zásuvných modulů je způsob, jak do programu přidat nové funkce, aniž by tvůrce zásuvného modulu musel modifikovat zdrojové kódy programu. Způsoby vytváření modulů silně závisí na vlastnostech pužitých programovacích jazyků. Nespornou výhodu mají interpretované jazyky s vestavěnou podporou dynamického zavádění modulů tříd a funkcí, protože se tvůrci programu nemusejí příliš zabývat rozdíly zavádění modulů na různých platformách a také odpadá problém spolupráce modulů přeložených různými překladači. Moduly jsou většinou distribuovány ve formě komprimovaného balíčku nebo sdílené knihovny (*.so, *.dll, *.jar), ale mohou jimi být i prosté soubory obsahující zdrojové kódy skriptovacích jazyků.

Moduly využívají funkčního rozhraní aplikací (angl. Application Programming Interface, API), jejichž funkčnost rozšiřují. API programu obsahuje funkce pro správu verzí a mechanismy registrace, zavádění nebo odstranění modulů. API může dovolovat vytváření interních i externích rozšíření. Může se omezit pouze na programovací jazyk, ve kterém je napsáno rozhraní nebo může obsahovat interpret jiného jazyka.

Při vytváření modulů se používají některé návrhové vzory [3] – např. továrna (angl. factory), jedináček (angl. singleton) nebo adaptér (angl. adapter, wrapper). Továrna může být funkce, která vytváří instanci třídy obsažené v modulu, jedináček může být třída, která registruje zásuvný modul v hostitelské aplikaci, a adaptér může být třída vytvářející jednotné rozhraní mezi moduly napsanými v jiných jazycích.

2.3.1 Technologie pro vytváření zásuvných modulů a doplňků

V úvodu této podkapitoly bylo řečeno, že záleží na zvolené platformě a na jazyce, který tvůrci doplňků mohou pro vývoj využít. Toto tvrzení nyní rozvedu na příkladu tvorby doplňků pro programy postavené na jádře Gecko[12] (např. Mozilla[13] Firefox a Thunderbird).

[12] https://developer.mozilla.org/en/Gecko
[13] https://www.mozilla.org/

Gecko se stará o vykreslování uživatelského rozhraní a webových stránek. Obsahuje DOM inspektor, interpret JavaScriptu a další webové technologie. Pomocí něj je interpretován celý program a uživatelské rozhraní. Program je tvořen skripty napsanými v JavaScriptu, XML soubory s popisem prvků uživatelského rozhraní a kaskádovými styly, které definují jeho vzhled. Tato technologie se nazývá XUL – XML User Interface Language[14]. Balíčky s doplňky jsou tvořeny stejnou strukturou souborů jako uživatelské rozhraní. Jsou doplněny o manifest s informacemi o verzi doplňku a verzi programu, pro kterou je doplněk určen. Použití JavaScriptu jako platformy pro vývoj programů a doplňků má hned několik výhod: tuto technologii zná velké množství vývojářů, vývoj doplňků se podobá vývoji dynamických webů a zavedení doplňku probíhá prostým načtením jeho kódu do hlavního programu a jeho interpretací. Nevýhodou je, že program pro svoji činnost potřebuje interpret JavaScriptu.

Pro vytváření doplňků v jiných jazycích nabízí Gecko technologii XPCOM[15] (Cross Platform Component Object Model), která je podobná technologiím CORBA [5] (Common Object Request Broker Architecture) a Microsoft COM (Component Object Model). Všechny zmíněné technologie slouží k obecnému zprostředkování objektů a služeb aplikacím. Tvoří jednotné komunikační rozhraní mezi objekty (moduly), které mohou být vytvořeny v různých programovacích jazycích a spuštěny vzdáleně na různých stanicích.

CORBA definuje jazyk IDL (Interface Definition Language) pro jednotný popis rozhraní objektů. Objekty spravuje zprostředkovatel objektů (ORB – Object Request Broker). Odpovídá za distribuované zpracování požadavků a koordinuje předávání parametrů a výsledků zpracování mezi objekty na straně klienta a serveru. Pro komunikaci se používá protokol GIOP (General Inter-ORB Protocol). Obdobnou technologii, UNO[16] (Universal Network Objects), používá pro svoje zásuvné moduly projekt OpenOffice.org.

COM je implementace objektového modelu pro operační systém Windows. V současnosti je nahrazována technologií Windows Communication Service (WCF), která je součástí .NET frameworku[17].

Další používanou technologií je specifikace OSGi frameworku[18], která popisuje dynamický modulární systém postavený na Javě. Specifikace definuje v pětivrstvové architektuře životní cyklus modulů, způsoby zavádění, odebírání modulů, zpřístupnění služeb mezi moduly a bezpečnostní mechanizmy. Existuje několik implementací tohoto frameworku. Mezi nejznámější patří Equinox[19], který je použit v integrovaném vývojovém prostředí (IDE) Eclipse, ArgoUML a v mnoha dalších programech.

Příklad 2.1 ukazuje vytváření modulu frameworku Equinox v programu ArgoUML. Třídy v modulech programu musí být odvozeny od rozhraní `ModuleInterface`, pomocí kterého jsou metody třídy zpřístupněny okolí. Rozhraní obsahuje funkce pro získání informací o modulu (`getName` a `getInfo`), zpřístupnění a zrušení modulu.

```
public class ModuleCpp implements ModuleInterface {
  public String getName() {
    return "GeneratorCpp";
  }
  public String getInfo(int type) { ... }
```

[14] https://developer.mozilla.org/en/XUL
[15] https://developer.mozilla.org/en/XPCOM
[16] http://udk.openoffice.org/
[17] http://msdn.microsoft.com/en-us/library/bb978523.aspx
[18] http://www.osgi.org/
[19] http://www.eclipse.org/equinox/

```
  public boolean enable() { ... }
  public boolean disable() { ... }
}
```

Kód 2.1: Kostra generátoru zdrojových kódů jazyka C++ v ArgoUML

2.3.2 Vytváření modulárních systémů v C/C++

Moduly vytvořené v jazycích C a C++ mají vždy podobu přeloženého binárního souboru – buď jsou částí spustitelného programu nebo jsou uloženy ve sdílené knihovně. Jazyky C a C++ nemají vestavěnou podporu pro načítání sdílených knihoven za běhu aplikace. Místo toho používají systémové funkce pro zavedení knihoven do paměti a zpřístupňování symbolů uložených uvnitř knihovny. Symboly mohou být konstanty, proměnné a funkce.

Při načítání symbolů ze sdílené knihovny se může objevit několik problémů – překladač jazyka C vytváří jednoznačné identifikátory funkcí (které nejsou statické) nebo globálních proměnných ze zdrojového kódu. Jazyk C++ nedokáže rozlišit symboly pouze podle názvů ve zdrojových kódech, protože se v kódu mohou vyskytovat symboly stejného jména, nejčastěji funkce, které mají různé parametry (jsou přetížené) nebo se nacházejí v jiných jmenných prostorech. C++ zavádí dekoraci jmen (angl. name mangling), pomocí které vytváří během překladu jednoznačné identifikátory všech symbolů. Standard jazyka C++ nedefinuje způsob dekorace jmen symbolů. Každý překladač má svůj způsob dekorace, a proto se může stát, že stejná knihovna přeložená různými překladači nabízí odlišné symboly. Problém lze řešit statickým sestavením, zavedením knihoven během spuštění programu nebo použitím sestavení ve stylu jazyka C (angl. C external linkage, deklarace `extern "C"`). Použití tohoto sestavení nemá smysl pro třídy a pro přetížené funkce.

Dalším problémem je dynamické načítání tříd. Neexistuje žádný přímý mechanizmus zpřístupnění tříd, který by fungoval na všech platformách. Instance tříd se musí vytvářet pomocí továrních funkcí (návrhový vzor továrna [3]).

Nejzávažnějším problémem při zavádění symbolů jazyka C++ je nekompatibilita aplikačního binárního rozhraní (ABI). Jazyk C++ nemá standardizované ABI, není proto zaručeno, že instance tříd načtených z různých sdílených knihoven budou správně fungovat. Knihovny mohou být přeloženy různými překladači (verzemi překladačů) a třídy se mohou lišit tabulkami virtuálních metod (Vtable). Tyto problémy je možno alespoň částečně vyřešit použitím knihoven nahrazujících virtuální tabulky tříd (např. knihovna DynObj [20]).

Moduly mohou být podle způsobu zavádění rozděleny do tří typů:

- Statické moduly – k programu se přidávají během sestavování. Nevýhodou statických modulů je nutnost opětovného sestavení programu po přidání nebo odebrání modulů. Moduly napsané v jazyce C se musí ručně zaregistrovat v hostitelské aplikaci, čehož lze dosáhnout pouze zásahem do kódu hostitelské aplikace. Moduly napsané v jazyce C++ se mohou registrovat automaticky pomocí statických tříd. Výhoda statických modulů je v rychlosti komunikace a v binární kompatibilitě.

- Dynamické moduly zaváděné při spuštění programu – pro registraci modulů v jazyce C se nemůže použít funkce, která má ve všech knihovnách stejné jméno, protože by způsobila redefinici existujícího symbolu. Pokud byla dynamická knihovna vytvořena stejným překladačem jako hostitelská aplikace, může exportovat i třídy.

[20] http://www.codeproject.com/KB/library/dynobj.aspx

- Dynamické moduly zaváděné při běhu programu – pokud jsou moduly napsány v jazyce C++ nebo mají rozhraní v jazyce C++, vinou nekompatibility ABI nemusejí fungovat správně. Moduly napsané v jazyce C tento neduh nemají.

Moduly mohou být také rozděleny podle jazyka použitého pro rozhraní:

- Čisté C – rozhraní je tvořeno strukturami s ukazateli na funkce (pseudoobjekty). Z pohledu binární kompatibility mezi moduly vytvořenými různými překladači se jedná o nejspolehlivější rozhraní.
- Čisté C++ – rozhraní je tvořeno bázovými třídami, od kterých jsou odvozeny třídy modulů. Instance tříd jsou zpřístupněny pomocí tovární funkce, která je v globálním jmenném prostoru a je sestavena ve stylu jazyka C. Toto rozhraní by se nemělo používat mezi moduly vytvořenými různými překladači.
- Duální C/C++ – hostitelská aplikace nabízí rozhraní jazyka C i C++. Tvůrci modulů si mohou vybrat, které rozhraní budou používat. Nevýhodou tohoto přístupu je zvýšená složitost rozhraní, které musí být zdvojeno. Pokud komunikují moduly mezi sebou, musí být v aplikaci adaptéry [3] pro přechod mezi těmito jazyky.
- Hybridní C/C++ – moduly mohou být vytvářeny v obou jazycích, ale rozhraní je tvořeno funkcemi (resp. ukazateli na funkce) jazyka C. Hybridní rozhraní zachovává stejnou úroveň kompatibility mezi moduly, jako rozhraní v čistém C. Nevýhodou je, že tvůrci modulů napsaných v jazyce C++ musejí vytvářet adaptér pro rozhraní a nemohou využívat všechny konstrukce jazyka C++ (např. výjimky).

Příklad dynamicky zaváděných modulů s rozhraním napsaným v čistém C nabízí grafový editor Dia. Kód 2.2 ukazuje kostru filtru pro import z formátu SVG. Rozhraní modulu je tvořeno funkcí `dia_plugin_init`, pomocí které se registruje struktura obsahující ukazatel na filtr (struktura `svg_import_filter`).

```
gboolean import_svg(const gchar *filename,
  DiagramData *dia, void* user_data)
{  ... }
static const gchar *extensions[] = {"svg", NULL };
DiaImportFilter svg_import_filter = {
  N_("Scalable Vector Graphics"),
  extensions,
  import_svg
};
PluginInitResult dia_plugin_init(PluginInfo *info)
{
  if (!dia_plugin_info_init(info, "SVG",
      _("SVG import filter"), NULL, NULL))
    return DIA_PLUGIN_INIT_ERROR;
  filter_register_import(&svg_import_filter);
  return DIA_PLUGIN_INIT_OK;
}
```

Kód 2.2: Kostra zásuvného modulu obsahujícího import dat z formátu SVG do programu Dia

Kapitola 3

Návrh grafového editoru

Tato kapitola popisuje detailní návrh editoru. Návrh je psán z pohledu vývojáře, a proto je kladen důraz na podrobný popis jádra aplikace a rozhraní modulů. Nejdříve jsou specifikovány požadavky na výslednou aplikaci, poté jsou popsány komponenty jádra tvořeného správcem modulů a grafovým konejnerem, způsob zavádění modulů, datová reprezentace grafu, formát použitý pro jeho ukládání a nakonec jednotlivé typy modulů. V poslední části této kapitoly se zabývám také návrhem z pohledu uživatele. Tato část popisuje, jakým způsobem bude moci uživatel použít funkce programu.

3.1 Specifikace požadavků

Požadavkem na návrh aplikace je, aby byla univerzální, modulární a snadno rozšiřitelná. Aplikace bude splňovat následující body:

- aplikace bude rozdělena na jádro s omezenou funkčností a moduly, které budou obsahovat hlavní funkční část programu
- aplikace bude podporovat pět typů modulů – modul správce objektů, importu, exportu, grafického uživatelského rozhraní a rozhraní příkazového řádku
- moduly budou uloženy ve sdílených knihovnách, jejich rozhraní bude hybridní
- aplikace bude mít dvě uživatelská rozhraní – grafické a textové
- textové rozhraní bude fungovat jako filtr pro převod mezi souborovými formáty grafu, ale bude také umět vytvářet grafy pomocí jednoduchých příkazů, přičemž se bude starat o automatické rozvržení výsledného grafu
- aplikace bude umět exportovat grafy do formátu PDF, SVG, PNG a PostScript
- součástí aplikace bude sada základních grafických tvarů (vrcholů a hran)

3.2 Správce zásuvných modulů

Úkolem správce modulů je načíst všechny potřebné moduly, které jsou navzájem kompatibilní, a poskytnout jim komunikační rozhraní ostatních modulů. Jeho činnost je rozdělena do několika částí. Nejdříve musí vyhledat všechny sdílené knihovny obsahující pluginy. Poté

proběhne registrace všech modulů načtených z knihoven. Na základě informací o verzích modulů manažer rozhodne, které moduly budou použity, a inicializuje je.

Během načítání knihoven může dojít k registraci různých verzí stejného modulu. Proto manažer vybere vždy modul s nejvyšším číslem verze, zavolá jeho inicializační funkci a přesune jej do seznamu inicializovaných modulů.

3.2.1 Detekce a načtení zásuvných modulů

Proces detekce a načtení zásuvného modulu začíná prohledáním složky se sdílenými knihovnami. Manažer projde všechny soubory v zadané složce a načte všechny sdílené knihovny. Každá sdílená knihovna musí exportovat funkci `pm_module_registration`, kterou manažer načte a zavolá ji. Parametrem funkce `pm_module_registration` je struktura obsahující informace o verzi manažeru a všech rozhraní modulů, funkci registrace modulu (`register_module`) a logovací funkci.

Pokud je v knihovně více verzí stejného modulu, může sdílená knihovna vybrat tu nejvhodnější a zaregistrovat ji vyplněnou strukturou `pm_module` obsahující její jméno, typ modulu a ukazatele na inicializační a deinicializační funkci. Definice funkce a strukur potřebných pro registraci modulů je uvedena v kódu 3.1.

```
typedef struct _pm_module {
  pm_version module_version;
  PM_MODULE_TYPE module_type;
  const char * module_name;

  int (*init_module)(pm_module_interface* interface,
                     const pm_module_helper* helper);
  int (*delete_module)();
} pm_module;

typedef struct _pm_manager_helper {
  pm_version manager_version;
  pm_version ui_interface_version;
  pm_version import_interface_version;
  pm_version export_interface_version;
  pm_version om_interface_version;
  int (*register_module)(const pm_module * module);
  void (*msg_log)(int , const char *, ...);
} pm_manager_helper;

int pm_module_registration(pm_manager_helper* helper);
```

Kód 3.1: Struktura registrovaného modulu `pm_module`, helperu a definice funkce `pm_module_registration`, kterou musí sdílené knihovny exportovat

3.2.2 Inicializace modulů

Inicializace modulů je vykonána zavoláním funkce `init_module` každého modulu, který správce vybral. V této funkci dojde k výměně rozhraní mezi modulem a jádrem aplikace. Rozhraní tvoří struktury s ukazateli na funkce. Pomocí ukazatele na strukturu `interface`,

předá modul správci modulů svoje rozhraní a pomocí ukazatele `helper` se předá struktura s rozhraním správce modulů, obsahující i rozhraní grafového kontejneru. Krom toho si zde mohou moduly inicializovat své interní struktury a alokovat zdroje, protože tato funkce je volána jako párová s funkcí `delete_module`.

Každý typ modulu má odlišné rozhraní tvořené jinou strukturou. Toto řešení je výhodné v případě budoucí změny rozhraní jednoho typu modulu, protože taková změna ovlivní jen moduly daného typu, ne všechny.

V parametru inicializační funkce je použit ukazatel na strukturu `pm_module_interface` (zástupnou strukturu), který si každý modul musí přetypovat na strukturu náležící danému typu modulu. Moduly vyplní strukturu rozhraní tvořenou ukazateli na funkce a získají přístup ke službám správce modulů a grafového kontejneru. Rozhraní modulů jsou popsána v podkapitolách 3.4, 3.5, 3.6 a 3.7, rozhraní správce modulů je popsáno v podkapitole 3.2.4 a grafového kontejneru v podkapitole 3.3.2.

3.2.3 Odstraňování modulů

Proces začíná vyprázdněním grafového kontejneru, poté dojde k odstranění modulu uživatelského rozhraní, následně proces pokračuje moduly importů, exportů a nakonec správci objektů. Správce modulů zavolá všem načteným modulům jejich funkci `delete_module`, aby mohly uvolnit zdroje, které si případně naalokovaly.

Na obrázku 3.1 je sekvenční diagram znázorňující životní cyklus modulu od jeho registrace, přes inicializaci, dále je zde znázorněna činnost aplikace spuštěním modulu uživatelského rozhraní a cyklus je ukončen smazáním modulu.

3.2.4 Rozhraní správce modulů

Rozhraní je modulům předáváno během inicializace pomocí ukazatele *helper*. Jak již bylo zmíněno, je pro každý typ modulu odlišné. Skládá se ze společné části, za kterou následuje vlastní část.

Společná část obsahuje funkci pro logování chyb, získání verze manažeru a získání cesty ke složce, ve které jsou sdílené knihovny obsahující moduly. Kromě sdílených knihoven zde mohou jednotlivé moduly mít složky s vlastními daty (např. ikonami či lokalizačními soubory).

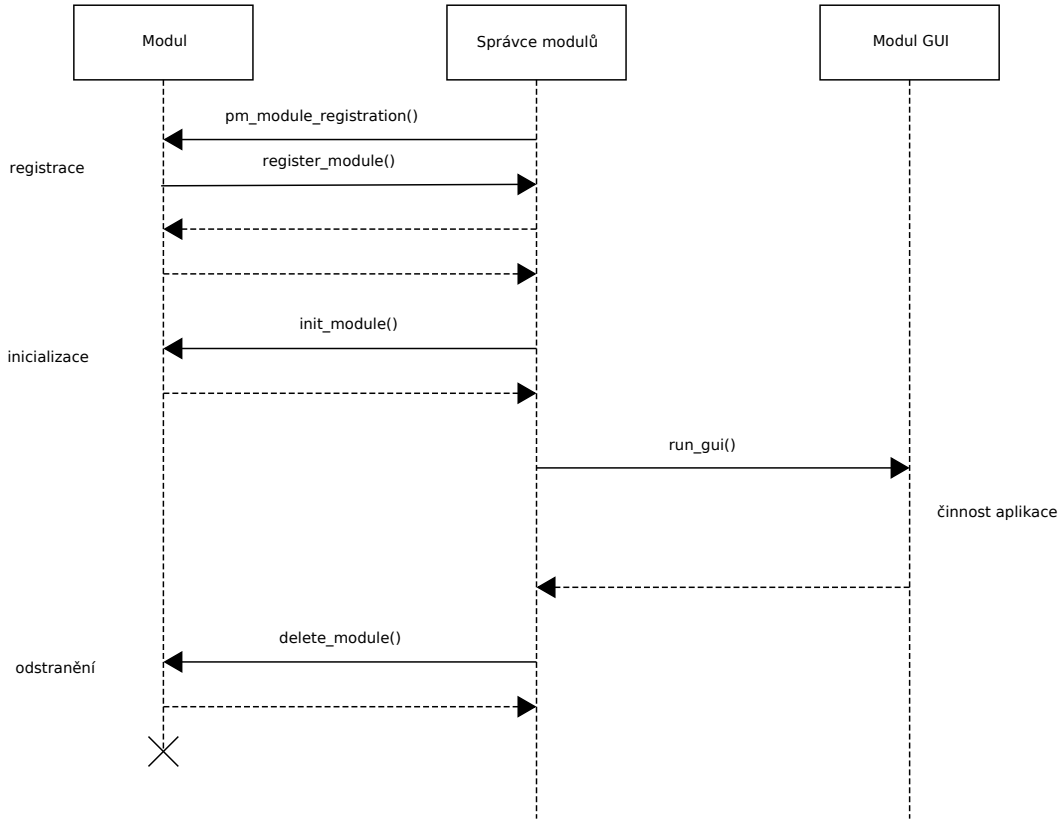
```
typedef _pm_module_helper {
  void (*log_error)(int dlevel, const char * msg, ...);
  const char * (*get_plugin_folder_path)();
  pm_version (*get_manager_version)();
} pm_module_helper;
```

Kód 3.2: Společné rozhraní správce modulů

Struktura předávaná modulu importů obsahuje kromě společného rozhraní pouze funkci pro zpřístupnění rozhraní správců objektů, pomocí nichž probíhá import grafů.

```
typedef _pm_import_helper {
  pm_module_helper helper;
  const pm_object_manager_interface **
     (*object_manager_interfaces)(int * count);
} pm_import_helper;
```

Kód 3.3: Rozhraní pro moduly importů



Obrázek 3.1: Životní cyklus modulu.

Moduly exportů musí mít přístup ke grafovému kontejneru, aby jej mohly procházet. Proto jejich rozhraní obsahuje strukturu s funkcemi grafového kontejneru. Dále potřebují získávat informace o grafu od správců objektů.

```
typedef _pm_export_helper {
  pm_module_helper helper;
  cosnt pm_object_manager_interface **
     (*object_manager_interfaces)(int * count);
  graph_export_interface container_interface;
} pm_export_helper;
```

Kód 3.4: Rozhraní pro moduly exportů

Správcové objektů musí přistupovat ke grafu a vytvářet objekty grafu.

```
typedef _pm_object_manager_helper {
  pm_module_helper helper;
  graph_om_interface container_interface;
} pm_object_manager_helper;
```

Kód 3.5: Rozhraní pro moduly správců objektů

Uživatelská rozhraní musí mít přístup k modulům správců rozhraní, importů a exportů. Kromě toho musí mít možnost procházet graf a také načítat a ukládat grafová data do

souborového formátu aplikace. Způsob ukládání grafu do souboru, načítání ze souboru a formát dat popíši v kapitole 3.3.3.

```
typedef _pm_ui_helper {
  pm_module_helper helper;
  pm_object_manager_interface ** (*object_manager_interfaces)(int
      * count);
  const pm_import_interface ** (*import_interfaces)(int * count);
  const pm_export_interface ** export_interfaces(int * count);
  const pm_file_info * (*get_file_info)();
  int (*load_file)(const char * path);
  int (*save_file)(const char * path);
  graph_ui_interface container_interface;
} pm_ui_helper;
```

Kód 3.6: Rozhraní pro moduly GUI a CLI

3.3 Grafový kontejner

Grafový kontejner uchovává veškeré informace o grafu, se kterým uživatel programu pracuje. Kontejner musí být navržen tak, aby podporoval obecné grafy (orientované, neorientované, smíšené), aby si uchovával jejich topologii, ale aby zároveň umožňoval vést více hran ze stejných počátečních vrcholů do stejných koncových (multigraf), aby mohl vytvářet volné hrany (hrany, které nevedou z vrcholů a nekončí v nich) a napojení hrany na hranu.

3.3.1 Datové struktury grafového kontejneru

K reprezenatci grafu se používají matice souslednosti, incidence nebo seznamy sousedů či vrcholů a hran [1]. Maticová reprezentace grafu není za těchto podmínek příliš vhodná, protože příliš plýtvá paměťovým prostorem a při mazání či přidávání vrcholů musí dojít k smazání (resp. přidání) řádku a sloupce matice a společně se seznamem sousedů (sousedních vrcholů) neumožňuje vytvářet multigrafy a hrany, které nespojují žádné vrcholy. Nejvhodnější řešení je založeno na seznamu vrcholů a hran, které umožňuje vytvářet libovolné typy grafů (včetně konstrukcí nekorektních z hlediska matematické definice grafu, např. volné hrany nebo hrany napojené na jiné hrany).

Kontejner je tvořen dvěma seznamy – seznamem ve kterém jsou uloženy vrcholy, a seznamem, ve kterém je uložena dvojice – hrana a dva ukazatelé na vrcholy (obr. 3.2).

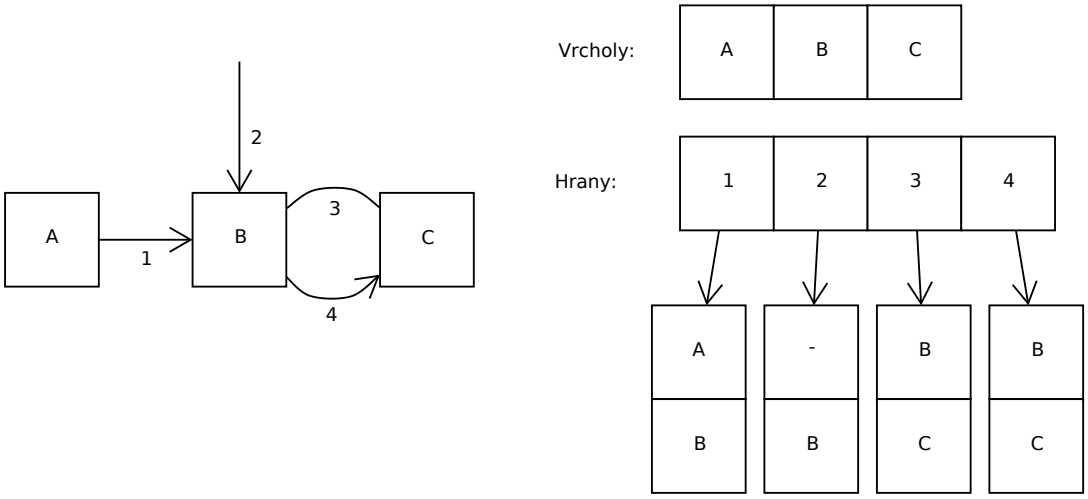
Objekty grafu mají jednotnou strukturu `g_object` 3.7. První část struktury obsahuje jednotné informace dostupné všem modulům – typ grafového objektu, ukazatel na rozhraní správce objektů, číslo vrstvy, odkaz na objekt skupiny, do které objekt patří, a pozice levého horního bodu pravoúhlého čtyřúhelníku, ve kterém je objekt. Druhá část je specifická pro každého správce objektů a skládá se z identifikátoru typu objektu a ukazatele na data objektu.

Pokud objekt nepatří do žádné skupiny (podgrafu), je ukazatel `group` nastaven na nulu.

```
typedef struct _g_object {
        G_OBJECT_TYPES g_object_type;
        struct _pm_object_manager_interface * manager;
        unsigned int layer;
```

```
        struct _g_object * group;
        g_point position;
        unsigned int om_object_type;
        void * data;
} g_object;
```

Kód 3.7: Definice struktury objektů grafu v grafovém kontejneru



Obrázek 3.2: Graf a jeho datová reprezentace pomocí seznamu vrcholů a hran

Typy grafového objektu (výčtový typ `G_OBJECT_TYPES`):

- `G_NULL` – nulový vrchol (uzel). Tento vrchol slouží pouze jako pomocná struktura pro uložení topologie hrany, která není vedena z vrcholu do vrcholu. Kromě typu, pozice a vrstvy jsou všechny hodnoty nulové. Příkladem použití tohoto typu vrcholu je bod (v datové reprezentaci označen pomlčkou), ze kterého vychází hrana č. 2 na obrázku 3.2.
- `G_VERTEX` – vrchol. Význam vrcholu a struktury obsahující popis vrcholu určuje správce objektů. Všechny vrcholy na obrázku 3.2 jsou tohoto typu.
- `G_VERTEX_GROUP` – vrchol obsahující podgraf (skupinu). Ukazatel na data tohoto objektu obsahuje strukturu `g_node_graph`, která je tvořena seznamem ukazatelů na členy skupiny, daty patřícími vrcholu a informací o rozbaleném nebo sbaleném stavu vrcholu. Pokud bude vrchol rozbalený, zobrazí se vrcholy a hrany jeho podgrafu. Pokud bude sbalený, zobrazí se jen tento vrchol.
- `G_EDGE` – neorientovaná hrana. Grafový kontejner ukládá ke každé hraně dvojici objektů, kterými je hrana ohraničena. Objekty mohou být vrcholy, ale i jiné hrany. U neorientovaných hran nezávisí na pořadí ve dvojici. Příkladem je hrana 3 na obrázku 3.2.
- `G_ORIENTED_EDGE` – orientovaná hrana. Závisí na pořadí dvojice objektů. Hrana vychází z prvního objektu a končí ve druhém. Příkladem jsou hrany 1, 2 a 4 na obrázku 3.2.

Na úrovni topologie grafu se nepracuje s ohodnocením hran. Ohodnocení hran a jeho faktický význam je v kompetenci správce objektů.

3.3.2 Rozhraní kontejneru

Služby grafového kontejneru využívají přímo či nepřímo všechny ostatní moduly. Správcové objektů pomocí něj vytvářejí vrcholy a hrany. Moduly exportů procházejí všechny objekty (nebo jen vybranou skupinu) a generují cílový soubor. Moduly uživatelských rozhraní taktéž procházejí objekty a vykreslují je do oblasti dokumentu, ale pokud vytvářejí nové objekty, nekomunikují přímo s kontejnerem, objekty grafu vytvářejí pomocí rozhraní správců objektů. Toto omezení platí i pro moduly importů.

Každý typ modulu má vymezené pole působnosti. Využije jen část funkcí kontejneru. Proto má svoje vlastní rozhraní a ostatní funkce, které by neměl používat, jsou pro něj nedostupné. Například moduly uživatelského rozhraní a importů nesmí vytvářet objekty grafu přímo, ale musí použít funkce správců objektů, tudíž nemají v rozhraní funkce pro vytváření vrcholů a hran.

- Rozhraní pro komunikaci se správcem objektů (`graph_om_interface`) obsahuje funkce pro získání informací o topologii grafu, pro vytváření a editaci grafu a funkce pro vytváření podgrafů.
- Rozhraní pro komunikaci s modulem uživatelského rozhraní (`graph_ui_interface`) obsahuje funkce pro získání informací o topologii grafu, výběr objektů grafu, procházení objektů a funkci `vertex_group` pro práci s podgrafy.
- Rozhraní pro komunikaci s modulem exportu (`graph_export_interface`) obsahuje funkce pro získání informací o topologii grafu, funkce pro procházení objektů, funkci `vertex_group` pro práci s podgrafy a funkci `get_selection` pro zpřístupnění výběru objektů.

3.3.3 Serializace grafu

Grafový kontejner udržuje mnoho specifických informací o objektech a o grafu jako celku. Proto vyžaduje navržení vlastního souborového formátu pro ukládání a načítání všech nezbytných informací.

Jednou z možností je vytvořit moduly importu a exportu souborového formátu, ale tento způsob řešení není příliš vhodný, protože by tyto moduly musely rozumět všem grafovým objektům, což není realizovatelné, protože kdokoli může přidat další modul správce objektů se sadou nových objektů. Vhodnějším řešením je vytvořit jeho implementaci v rámci jádra aplikace a serializaci interních informací o objektech nechat na správcích objektů.

Serializace dat do souboru může být binární nebo textová. Vhodnější pro ladění a přenos mezi platformami je textová podoba. Data budou hierarchicky strukturovaná v několika seznamech, pro něž je vhodné použít formát XML. Samozřejmě by bylo možné použít i jiné formáty, například formát zápisu konfiguračních souborů INI[1] nebo JSON[2].

Všechny formáty mají svoje klady a zápory. Výhoda XML spočívá v možnosti zpracování bez znalosti (nebo jen s částečnou znalostí) sémantiky dokumentu. Nevýhodou je horší čitelnost XML dokumentu a větší nároky na datové uložiště. INI formát je velmi jednoduchý,

[1] http://technet.microsoft.com/en-ie/library/cc722567(en-us).aspx
[2] http://www.json.org/

pro uživatele lépe čitelný a při jeho používání se tolik neplýtvá datovými prostředky, ale je možno jej zpracovávat pouze pomocí nástrojů, které znají sémantiku dokumentu. JSON využívá datový prostor ještě efektivněji než INI, ale pro uživatele je naprosto nepřehledný a nečitelný.

K parsování XML dokumentu se používá externí knihovna. Program využívá pouze omezený počet API funkcí knihovny. Funkce jsou zabaleny do rozhraní, pomocí kterého mohou správci objektů ukládat grafová data. Výhodou tohoto řešení je, že všechny moduly zainteresované v procesu serializace nemusejí být sestaveny s touto knihovnou. Vnitřní implementace a používání knihovny je jim skryto. Pokud v budoucnu dojde ke změně knihovny na zpracování XML souborů (nebo dokonce k výměně formátu ukládání), dotkne se tato změna pouze implementace jádra.

Struktura rozhraní pro ukládání a načítání XML souborů (`pm_file_interface`) obsahuje funkce pro vytváření dceřinných uzlů, nových atributů uzlu a textových hodnot patřících uzlům. Pro čtení nabízí procházení uzlů na stejné úrovni hierarchie, procházení dceřinných uzlů, vyhledávání atributů a zpřístupnění textové části uzlů.

Ukládání a načítání iniciuje uživatel prostřednictvím modulu uživatelského rozhraní (GUI, CLI), který zavolá funkci `save_file` (resp. `load_file`) z rozhraní jádra. Celý proces dále řídí jádro. Do XML souboru uloží informace o verzi správců objektů, jejichž vrcholy a hrany se nacházejí v grafu, seznam vrcholů, hran, napojení hran mezi objekty a podgrafů.

Během ukládání se pro každý objekt zavolá funkce `save_object` z rozhraní manažera objektů, kterému objekt patří. Jako parametry se předají ukazatel na objekt, uzel z hierarchie XML dokumentu, do kterého bude funkce vkládat informace o objektu, a struktura `pm_file_interface` s rozhraním pro práci s XML. Správce objektů do uzlu vytvoří podstrom s informacemi o objektu a vrátí řízení zpět.

Načítání probíhá obdobně – pro každý objekt se zavolá funkce `load_object` a předají se jí jako parametry ukazatel na nově vytvářený objekt, uzel XML dokumentu, ve kterém jsou obsaženy informace o objektu a rozhraní pro práci s XML.

Funkce `save_file` (`load_file`) je navržena tak, aby zapisovala (resp. četla) do standardního výstupu, pokud není v parametru zadána cesta k souboru.

Kód 3.8 ukazuje XML soubor s grafem z obrázku 2.1 vytvořený ve výsledné aplikaci. Datové části objektů jsou zkráceny a nahrazeny tečkami (...). Dokument je uvnitř kořenové značky `graphdoc`, první seznam – `omrequired` je seznam požadovaných správců objektů. Pokud tyto moduly aplikace nemá, není možné soubor načíst. Dalšími seznamy jsou seznam vrcholů (`vertices`), seznam hran (`edges`), seznam napojení hran (`connection`) a seznam skupin (`groups`), který není použit. Objekty obsahují datové části (značka `data`), ve kterých jsou uloženy interní informace o objektech.

Napojení hran mezi objekty je v grafovém kontejneru vyjádřeno pomocí ukazatelů, v uloženém souboru by takový vztah ztratil smysl, proto musí jádro vytvořit pro každý objekt číselný indentifikátor a informaci o napojení hran a o podgrafech přečíslovat. Tento seznam se ukládá jako třetí v pořadí, není tedy přímou součástí hrany v seznamu hran. Důvodem k tomuto kroku je fakt, že je možné napojit hranu na jinou hranu. Díky tomuto uspořádání jsou všechny objekty vytvořeny již před načítáním informací o napojeních hran a nemusí se ukládat dočasné informace nebo vícekrát procházet datový soubor, protože nehrozí vytvoření spoje mezi objekty, které ještě nebyly vytvořeny.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphdoc>
  <omrequired>
    <obmanager name="stdgraph" version="1.0"/>
```

```
  </omrequired>
  <vertices>
    <vertex om="stdgraph" grtype="1" layer="0" x="0" y="0"
       id="0">
      <data .../>
    </vertex>
    <vertex om="stdgraph" grtype="1" layer="0" x="120" y="20"
       id="1">
      <data .../>
    </vertex>
  </vertices>
  <edges>
    <edge om="stdgraph" grtype="4" layer="0" x="40" y="25"
       id="2">
      <data ...>
        <copoints>
          <p x="40" y="25" t="0"/>
          <p x="160" y="45" t="0"/>
        </copoints>
      </data>
    </edge>
  </edges>
  <connections>
    <edge id="2" start="0" end="1"/>
  </connections>
  <groups/>
</graphdoc>
```

Kód 3.8: Příklad XML souboru s grafem.

3.4 Správce objektů

Správce objektů je modul, který definuje typy, tvary a význam objektů v grafu. Má na starosti vytváření vrcholů a hran tvořících grafy konkrétního typu. Graf může obsahovat objekty různých správců objektů. Každý správce má na starost jen svoje vlastní objekty a ostatní ignoruje. Tento návrh nicméně umožňuje vytvářet i hrany spojující vrcholy spravované různými správci.

Nejčastěji se správcem objektů komunikuje modul uživatelského rozhraní (angl. frontend) aplikace. Uživatel pomocí uživatelského rozhraní zadává správcům objektů příkazy, kterými manipuluje s objekty v grafu. Takto komunikuje před vytvářením objektu grafu, při vytvoření, posunu, změně vlastnosti, výběru, mazání a také pokud chce objekt vykreslit. Správce objektů kreslí objekty do grafického kontextu, nemusí to být jen kontext uživatelského rozhraní, ale i kontext modulu exportu. Komunikaci zahajuje vždy uživatelské rozhraní (resp. modul exportu).

3.4.1 Komunikace správce objektů s uživatelským rozhraním

Po procesu inicializace správců objektů musí modul uživatelského rozhraní projít jednotlivé správce a zjistit, jaké objekty spravují. Pomocí funkce `register_objects` si vytvoří seznam všech vrcholů a hran ze správce objektů a poté jej může zobrazit v nabídce objektů. Každý objekt je tvořen strukturou `ui_object`. Ve struktuře je jméno správce objektů (pomocí něj se záznam zařadí do správného seznamu objektů), typ objektu (vrchol nebo hrana), jméno zobrazované v GUI (může podléhat lokalizaci), skutečné jedinečné jméno (používá se při popisování grafu v CLI a identifikaci při vytváření objektu) a cesta k souboru s ikonou.

```
typedef struct _ui_object ui_object;
struct _ui_object {
        const char * obj_manager;
        UI_OBJECT_TYPE object_type;
        const char * caption_name;
        const char * object_name;
        const char * icon_path;
};
```

Kód 3.9: Definice struktury objektů nabízených správcem objektů

Správce objektů nabízí funkce pro vytvoření a úpravu grafů, které obalují funkce grafového kontejneru. Díky nim může grafu přidat význam a vytvořit jen ty konstrukce, které jsou v daném grafu povoleny. Hrany a vrcholy se vytvářejí pomocí funkcí `create_vertex`, resp. `create_edge` pro hrany napojené na vrcholy a `create_void_edge` pro volné hrany.

Každý objekt vytvořený správcem objektů může mít sadu vlastností, které by mělo jít nastavit pomocí uživatelského rozhraní. Modul uživatelského rozhraní získá seznam vlastností funkcí `get_object_options`. Funkce vrátí pole struktur `ui_object_option`, jež obsahuje popis typu vlastností a hodnoty vlastností objektu. Některé vlastnosti jsou společné se všemi ostatními objekty nebo předpokládáme jejich časté používání. Takové vlastnosti jsou označeny prefixem OO_STD a jejich struktury neobsahují název a popisek vlastnosti, který by se zobrazoval v GUI. Ostatní, které mají prefix OO, musí mít i svůj jednoznačný název a popisek, který může být lokalizován.

Vlastnosti může nastavit funkcí `set_object_options`.

Seznam typů vlastností:

- `OO_STD_POSITION` – pozice objektu (levý horní roh).
- `OO_STD_BGCOLOR` – barva pozadí objektu.
- `OO_STD_LINE` – barva, styl (plný, čárkovaný, tečkovaný, střídavě čárka tečka) a tloušťka čáry.
- `OO_STD_SIZE` – velikost objektu.
- `OO_STD_EVALUATION` – ohodnocení hrany.
- `OO_COLOR` – barva.
- `OO_TEXT` – řádek textu, dovoluje nastavit písmo a barvu.
- `OO_VALUE` – hodnota (celočíselná, desetinná nebo textová).

Objekty vykresluje správce objektů pomocí funkce `draw_object`, která kreslí objekty do grafického kontextu `gc_device`. Grafický kontext může patřit modulu grafického rozhraní, ale i modulu exportů. Jeho popis je v kapitole 3.5.1.

3.5 Grafické uživatelské rozhraní (GUI)

Grafické uživatelské rozhraní slouží pro komunikaci uživatele s programem. Rozhraní modulu GUI obsahuje pouze jednu funkci – `run_gui`. Zavolá ji správce objektů po inicializaci všech modulů a předá jí parametry programu. Od té chvíle se řízení aplikace předá modulu uživatelského rozhraní a opustí ji až při ukončování uživatelského rozhraní a programu.

Jediným parametrem, na který by měl modul GUI reagovat, je název vstupního XML souboru, který by měl po svém spuštění načíst. Ostatní parametry, jako třeba `-in=`*vstupní formát*, `-out=`*výstupní formát* nebo `-o` *název souboru*, by měl ignorovat. Tyto parametry mají význam při použití rozhraní příkazového řádku (podkapitola 3.6.1), kde budou vysvětleny. Souhrn všech parametrů programu je uveden v podkapitole 3.8.

Během inicializace získal modul rozhraní správce modulů a pomocí něj může projít všechny moduly správců objektů, importů, exportů a vytvořit v nabídkách a panelech nástrojů položky, které aktivují jejich funkce.

```
typedef struct _pm_gui_interface {
  const char * module_name;
  const pm_version module_version;
  int (*run_gui)(int argc, char ** argv);
} pm_gui_interface;
```

Kód 3.10: Definice rozhraní GUI

Další komunikaci vždy iniciuje uživatelské rozhraní a pokud je to potřeba, předává ukazatel na funkci zpětného volání (angl. callback) nebo na strukturu s funkcemi.

3.5.1 Grafický kontext

Uživatelské rozhraní nerozumí objektům grafu ani jejich významu, tzn. neumí je zobrazit. Pro zobrazení grafů na plátno dokumentu musí uživatelské rozhraní spolupracovat se správci objektů. Pro každý objekt zavolá funkci `draw_object`, které předá strukturu `gc_device` obsahující funkce pro kreslení. Uživatelské rozhraní tím odstíní API grafické knihovny použité v GUI a může pomocí této funkční vrstvy také ovlivňovat výsledné zobrazení (např. která část grafu bude zobrazena nebo velikost vykreslených objektů při přiblížení či oddálení).

Kontext nabízí funkce podobné většině grafických API (vytvoření pera, štětce, kreslení čar, křivek, obdélníků, elips, složitějších uzavřených útvarů, které mohou být ohraničeny úsečkami, křivkami a vyplněny barvou štětce, vytváření písma, získání rozměrů textu a jeho vykreslení na řádek).

V ukázce kódu 3.11 je uvedena definice funkcí pro práci se štětcem – grafickým objektem spravujícím barvu výplně uzavřených těles. Funkce vracející ukazatel `void *` vrací platformě závislé objekty grafických API funkcí. Může se jednat o manipulátory (handle), ukazatele na vnitřní struktury nebo instance tříd. Proto by se měly všechny objekty vytvořené funkcemi `create_` odstranit jejich protějškem `delete_`.

```
void * (*create_brush)(int color);
void * (*set_brush)(void * brush);
```

```
void (*delete_brush)(void * brush);
```

Kód 3.11: Ukázka rozhraní grafického kontextu

Alternativou ke zvolenému řešení je vytvoření komunikačního protokolu s jazykem pro popis objektů, pomocí kterého by probíhalo předávání informací mezi moduly správců objektů a moduly vykreslujícími objekty grafu (moduly GUI a exportu). Výhodou protokolu by byla větší univerzálnost, protože by moduly pracovaly s jednotným popisem všech objektů a jejich vlastností. Nevýhodou by byla zvýšená komunikační režie, a hlavně nutnost vybavit všechny moduly, které by chtěly pracovat s objekty grafu, analyzátorem jazyka pro popis objektů, což by znesnadňovalo jejich vývoj.

3.6 Rozhraní příkazového řádku (CLI)

Modul příkazového řádku komunikuje se správci objektů stejným způsobem jako modul GUI. Liší se pouze tím, že nepoužívá část rozhraní správců objektů pro interaktivní práci s grafem a nepotřebuje implementovat grafický kontext, protože graf nikam nevykresluje. Používá pouze funkce pro vytvoření vrcholů a hran a jejich propojení. Uživatel zadává příkazy, pomocí kterých se vytvářejí vrcholy a hrany grafu.

Rozhraní CLI obsahuje jen funkci `run_cli`, kterou správce modulů zavolá po inicializaci všech modulů programu, předá jí parametry programu a modul CLI může začít číst data ze standardního vstupu nebo ze souboru. Po ukončení čtení se ukončí i celý program. Modul by se měl chovat jako filtr, tzn. pokud uživatel nezadá vstupní soubor, měl by číst ze standardního vstupu, pokud nezadá výstupní soubor, měl by být výstup vložen do standardního výstupu.

3.6.1 Parametry příkazového řádku

Modul CLI by měl zpracovávat parametry, které mu předal správce modulů. Jde především o parametr definující vstupní a výstupní soubor a dále parametry určující vstupní a výstupní formát. Možné kombinace jsou uvedeny v seznamu:

- `-in=`[*formát*] – definice vstupního formátu. Pokud není zadán tento parametr, měl by modul rozhraní číst a zpracovávat příkazy pro tvorbu grafu. Pokud byl zadán prázdný příkaz `-in=`, měl by modul rozhraní číst grafová data ve formátu XML. Pokud je definován formát (přípustná je přípona souboru nebo jméno modulu importu), měl by modul rozhraní použít zvolený modul importu.

- `-out=`[*formát*] – definice výstupního formátu. Pokud není zadán nebo není uveden formát (zadáno pouze `-out=`), je výstupem graf ve formátu XML, pokud je zadán formát (přípustná je přípona souboru nebo jméno modulu exportu), měl by modul rozhraní použít zvolený modul exportu.

- `-o` *výstup* – výstupní soubor. Pokud není zadán, měl by modul rozhraní tisknout výstup na standardní výstup.

- *vstup* – vstupní soubor. Pokud není zadán, měl by modul rozhraní číst vstup ze standardního vstupu.

3.7 Moduly importů a exportů

Modul importů komunikuje se správci objektů a pomocí nich vytváří graf. Rozhraní pro přístup ke správcům objektů je zpřístupněno během inicializace modulu. Modul ve svém rozhraní obsahuje pouze jednu funkci – `import_function`, která provede import ze zadaného souboru. Pokud není zadána cesta k souboru, import proběhne čtením ze standardního vstupu (stdin).

Dále rozhraní obsahuje jméno modulu, verzi, popis modulu, jméno importovaného formátu a koncovku formátu. Poslední dvě položky mohou být zobrazeny v nabídkách GUI.

```
typedef struct _pm_import_interface {
  const char * module_name;
  const pm_version module_version;
  const char * about_module;
  const char * format_name;
  const char * extension;
  int (*import_function)(const char * path);
} pm_import_interface;
```

Kód 3.12: Rozhraní modulu importu

Modul exportů musí komunikovat přímo s grafovým kontejnerem, protože potřebuje procházet všechny objekty uložené v kontejneru. Rozhraní je zpřístupněno během inicializace. Exportem se může vygenerovat jiný grafový formát, obrázek, dokument PDF nebo zdrojový kód. Záleží na implementaci modulu, protože se může specializovat jen na typy objektů, kterým rozumí, a ostatní ignorovat. Pokud je výsledkem exportu obrázek, měl by modul implementovat funkce struktury grafického kontextu `gc_device` a využít funkce `draw_object` z rozhraní správců modulů. Struktura byla představena v kapitole 3.5.1.

Rozhraní obsahuje stejné položky jako rozhraní modulu importu. Liší se jen v názvu ukazatele na funkci – `export_function`. Pokud není modulu exportu zadán výstupní soubor, měl by obdobně jako modul importu tisknout výstup na standardní výstup.

3.8 Grafový editor pohledem uživatele

Z pohledu interakce s uživatelem je možno části systému rozdělit na uživatelské rozhraní (angl. frontend) a výkonnou část (angl. backend). Uživatelské rozhraní aplikace tvoří modul GUI nebo CLI a výkonnou část tvoří správce objektů, moduly importu, exportu a jádro programu.

Moduly uživatelských rozhraní jsou jediné moduly, pomocí kterých program s uživatelem komunikuje. Program vždy po spuštění vybere vhodný modul uživatelského rozhraní. Pokud uživatel nevynutí spuštění jiného modulu, je preferován modul grafického uživatelského rozhraní před moduly rozhraní příkazového řádku. Spuštění jiného modulu je vynuceno parametrem programu `-ui=`*název modulu*. Nenajde-li program žádný modul uživatelského rozhraní, ukončí svoji činnost.

Souhrn parametrů programu:

- `-ui=`*název modulu* – vynucení modulu uživatelského rozhraní
- `-in=`[*formát*] – definice vstupního formátu (pouze pro CLI)
- `-out=`[*formát*] – definice výstupního formátu (pouze pro CLI)

- `-o` *výstup* – výstupní soubor (pouze pro CLI)
- *vstup* – vstupní soubor

Kapitola 4

Implementace grafového editoru

V této kapitole budou popsány implementační detaily. Aplikaci (resp. její jádro) jsem nazval Anve. Jméno Anve je rekurzivní zkratka a znamená Anve Není Vektorový Editor (popř. Anve is Not Vector Editor).

4.1 Jádro aplikace

Jádro aplikace je napsáno v jazyku C++. Je tvořeno dvěma jedináčky, které okolí nabízejí statické metody, protože je nezbytné, aby uměly komunikovat i s moduly, a čtyřmi pomocnými třídami. Jedináčci jsou třídy správce modulů (`pm_manager`) a grafového kontejneru (`graph_container`). Pomocné třídy jsou třída pro zaznamenávání chybových zpráv (`pm_error_log`), procházení složky se soubory (`pm_directory`), načítání sdílených knihoven (`pm_shared_library`) a práci s XML soubory (`pm_doc_file`). Význam správce modulů a grafového kontejneru byl vysvětlen v předchozí kapitole, proto se zaměřím na ostatní třídy tvořící jádro.

Rozhraní třídy `pm_error_log` používají všechny ostatní třídy pro zaznamenávání chybových zpráv. Samotná třída pouze vypisuje zprávy do zvoleného souboru (`err.log`). Umožňuje zprávy rozdělit do tří kategorií (chyby, varování a poznámky) a filtrovat je podle zvolené úrovně nastavení.

Třída `pm_directory` obaluje systémové funkce pro procházení složky. Pomocí ní může správce modulů projít složku s pluginy a detekovat sdílené knihovny.

Třída `pm_shared_library` obaluje systémové funkce pro zavádění knihoven a načítání symbolů.

Třída `pm_doc_file` slouží pro ukládání a načítání XML souborů s grafovým dokumentem. Obaluje funkce knihovny LibXML[1], kterou používá pro parsování a generování stromu XML dokumentu. Využit je pouze zlomek funkcí, které knihovna nabízí – funkce pro vytoření XML dokumentu (`xmlNewDoc`), uzlů a jejich atributů (`xmlNewNode`, `xmlNewText`, `xmlNewProp`, `xmlAddChild`), pro procházení stromu uzlů, atributů (`xmlGetProp`), jeho ukládání (`xmlSaveFormatFileEnc`) a načítání (`xmlReadFile`, `xmlCreatePushParserCtxt`, `xmlCreatePushParserCtxt`, `xmlParseChunk`).

[1] http://xmlsoft.org/

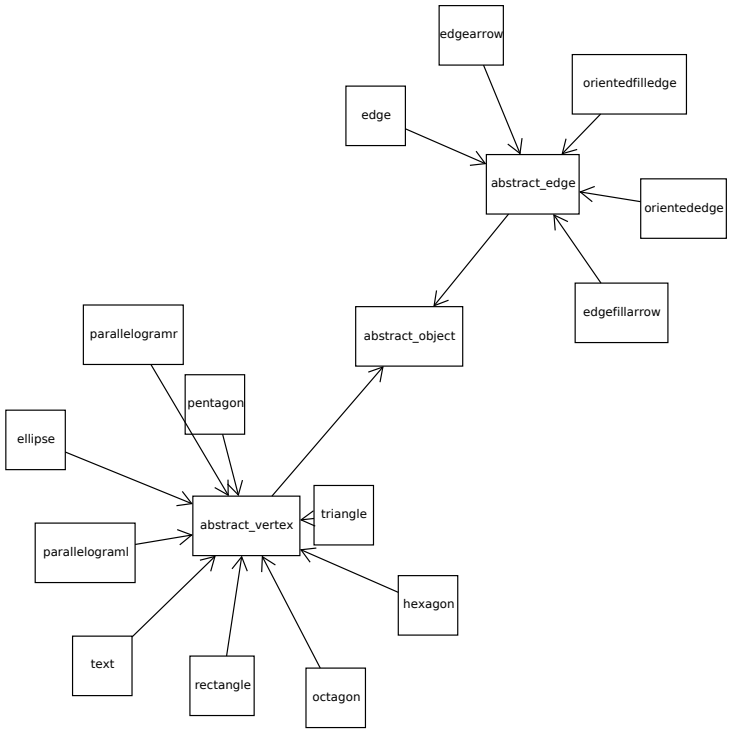
4.2 Modul Stdgraph

Modul správce objektů Stdgraph obsahuje devět typů vrcholů a pět typů hran. Představuje základní sadu objektů, proto umožňuje vytvářet volné hrany a napojovat hrany na kontrolní body jiných hran či objektů jiných správců.

Modul je tvořen třídou `stdgraph`, která vytváří pomocí statických metod rozhraní pro komunikaci modulu s moduly uživatelského rozhraní. Objekty jsou reprezentovány hierarchií tříd – všechny jsou potomky abstraktní třídy `abstract_object`. Vrcholy jsou odvozeny od třídy `abstract_vertex`, která je přímým potomkem třídy `abstract_object` a obsahuje informace o obecných vlastnostech objektů a metody pro jejich nastavování, ukládání a načítání z uzlu XML dokumentu. V odvozených třídách jsou již jen metody pro vykreslení konkrétního objektu (tvaru) a detekce jeho okrajů.

Obdobná je i hierarchie tříd hran. Ty jsou odvozeny od abstraktní třídy `abstract_edge` – potomka třídy `abstract_object`. Tato třída obsahuje vlastnosti hran. V odvozených třídách je již pouze definována metoda pro vykreslení hrany.

Graf na obrázku 4.1 ukazuje hierarchii tříd[2].



Obrázek 4.1: Hierarchie tříd reprezentujících objekty v modulu Stdgraph.

Seznam vrcholů modulu Stdgraph:

- rectangle – obdélník (čtverec)
- ellipse – elipsa (kružnice)
- triangle – trojúhelník

[2]Graf byl vygenerován pomocí modulu Cli, který je popsán v podkapitole 4.4. V této části je uveden i zdrojový kód tohoto grafu (kód 4.1).

- parallelogramL – kosodélník natočený doleva
- parallelogramR – kosodélník natočený doprava
- pentagon – pětiúhelník
- hexagon – šestiúhelník
- octagon – osmiúhelník
- text – popisek

Seznam hran modulu Stdgraph:

- edge – neorientovaná hrana
- edgearrow – neorientovaná hrana zakončená na obou stranách šipkou
- edgefillarrow – neorientovaná hrana zakončená na obou stranách vyplněnou šipkou
- orientededge – orientovaná hrana zakončená šipkou
- orientedfilledge – orientovaná hrana zakončená vyplněnou šipkou

4.3 Modul Qtgui

Modul Qtgui je modul grafického uživatelského rozhraní napsaný pomocí Qt Toolkitu[3].

Hlavní okno modulu reprezentované třídou `main_window` obsahuje paletu objektů, ve které jsou na záložkách zobrazeny ikony všech dostupných objektů. Každá záložka představuje jednoho správce modulů. Pod paletou je umístěn prohlížeč vlastností objektů (třída `option_viewer`), ve kterém může uživatel prohlížet a nastavovat všechny vlastnosti právě označeného objektu. Objekty vkládá na plochu dokumentu (třída `doc_view`), může je posouvat, měnit velikost vrcholů, přidávat řídicí body hran, připojit hranu na jiný objekt nebo smazat označený objekt. Uživatelské rozhraní pracuje ve dvou režimech práce – režimu vytváření a úprav, mezi kterými může uživatel přepínat.

V režimu vytváření může do grafu vkládat nové objekty. Vrcholy se vkládají prostým klepnutím na volné místo v dokumentu. Pokud uživatel vkládá vrcholy, může vybírat a upravovat i ostatní vrcholy. Pokud vkládá hrany, zobrazí se mu v grafu body, na které může hranu napojit. Hranu vytváří tahem myši. Během vytváření hran může upravovat i ostatní hrany.

V režimu úprav není možno vytvářet nové objekty. Uživatel může pouze vybrat libovolný objekt (vrchol i hranu) a upravovat jeho vlastnosti. V obou režimech je možno označený objekt smazat.

Uživatel může pracovat vždy pouze s jedním objektem, operace výběru více objektů stejně jako práce se schránkou (kopírování a vkládání), nebyly implementovány.

[3] qt.nokia.com

4.4 Modul Cli

Modul rozhraní příkazového řádku může sloužit jako prostý filtr převádějící vstup z formátu XML nebo z importů do podoby XML formátu nebo exportů. Kromě toho může sloužit i jako vstup pro příkazy, pomocí kterých může uživatel vytvářet grafy. Příkazy jsou podobné jazyku DOT (viz. 2.2.8), nicméně jsou jednodušší.

Rozlišujeme čtyři typy příkazů. Každý příkaz musí být zadán na samostatný řádek.

- `m` <*název správce objektů*> – přepnutí na správce objektů. Další příkazy budou vytvářet vrcholy a hrany z portfolia zvoleného správce objektů.
- `v` <*jméno vrcholu*> <*typ vrcholu*> `[`<*textový popisek*>`]` – vytvoření vrcholu. Typ vrcholu, jehož název je uveden, musí být podporován právě vybraným správcem objektů. Jméno vrcholu musí být jedinečné, použije se při vytváření hran. Nepovinnou částí příkazu je popisek. Pokud není uveden, použije se jméno vrcholu.
- `e` <*jméno vrcholu 1*> <*jméno vrcholu 2*> <*typ hrany*> `[`<*textový popisek*>`]` – vytvoří hranu mezi zadanými vrcholy. Typ hrany musí být podporován správcem objektů. Orientované hrany vždy začínají ve vrcholu 1 a končí ve vrcholu 2. Nepovinnou částí je popisek.
- `h [m|o]` – nápověda. Přepínač `m` vypíše dostupné správce objektů, přepínač `o` vypíše dostupné objekty právě vybraného správce objektů.

Modul načte a vykoná všechny příkazy. Pokud je příkaz chybný, vypíše na standardní výstup chybovou zprávu a pokračuje dále ve vykonávání dalších příkazů. Pokud řádek začíná znakem mřížky (`#`), přeskočí se.

4.4.1 Automatické rozvržení grafu

Pro výpočet výsledné pozice vrcholů je použit algoritmus založený na silách [9, 6]. Tento algoritmus vychází z fyzikálních zákonů o vztahu mezi elektricky nabitými částicemi (Columbův zákon) a deformací materiálů působením síly (Hookův zákon). Graf si můžeme představit jako soustavu kladně nabitých částic, které se navzájem odpuzují a jsou spojeny pružinami, které spojené částice přitahují. Výpočet probíhá iteračně. Během každé iterace musí algoritmus projít všechny dvojice vrcholů, vypočítat jejich vzájemnou odpudivou sílu a poté aktualizovat polohu a rychlost vrcholů. Výpočet končí ve chvíli, kdy klesne celková kinetická energie pod danou mez (tzn. vrcholy se dostanou do rovnovážného stavu a dále se již nehýbou).

Vytvořené vrcholy musí být umístěny na ploše grafu náhodným způsobem, protože pokud by byly všechny na jednom místě, tento algoritmus by nefungoval.

Pokud je graf nesouvislý, nebude tento algoritmus fungovat správně. Odpudivá síla zařídí, že se nesouvislé části od sebe budou neustále vzdalovat. Proto je do středu grafu vloženo gravitační pole, které takové části grafu drží pohromadě. Nevýhodou gravitačního pole je, že ovlivňuje a mírně deformuje i ostatní souvislé části grafu (např. kružnice tvořená dvaceti vrcholy má tvar čtverce se zaoblenými rohy).

Časová složitost algoritmu je $O(V^3)$, kde V je počet vrcholů, proto je vhodný spíše pro menší a středně velké grafy do cca sta vrcholů.

Kód 4.1 ukazuje příklad grafu vytvořeného pomocí příkazů modulu Cli. Výsledkem tohoto kódu je graf popisující hierarchii tříd modulu Stdgraph (graf 4.1).

```
m stdgraph
v A rectangle abstract_object
v E rectangle abstract_edge
v V rectangle abstract_vertex
v v1 rectangle rectangle
v v2 rectangle ellipse
v v3 rectangle triangle
v v4 rectangle pentagon
v v5 rectangle hexagon
v v6 rectangle octagon
v v7 rectangle parallelogramll
v v8 rectangle parallelogramrr
v v9 rectangle text
e v1 V orientededge
e v2 V orientededge
e V A orientededge
e E A orientededge
e v3 V orientededge
e v4 V orientededge
e v5 V orientededge
e v6 V orientededge
e v7 V orientededge
e v8 V orientededge
e v9 V orientededge
v e1 rectangle edge
v e2 rectangle edgearrow
v e3 rectangle edgefillarrow
v e4 rectangle orientededge
v e5 rectangle orientedfilledge
e e1 E orientededge
e e2 E orientededge
e e3 E orientededge
e e4 E orientededge
e e5 E orientededge
```

Kód 4.1: Příklad grafu popsaného pomocí příkazů modulu Cli

4.5 Moduly exportů

Implementace obsahuje čtyři moduly exportů – do souborových formátů PDF, SVG, PNG a PostScriptu. Všechny moduly jsou napsány v jazyce C a jsou součástí jedné sdílené knihovny.

Moduly k vykreslování grafů používají knihovnu Cairo[4]. Každý modul vytváří při exportu vlastní kreslící plochu (angl. surface), do které se poté vykreslí tvary objektů grafu. Část knihovny, která se stará o vykreslení, je pro všechny moduly totožná, jde pouze o obalení funkcí knihovny Cairo funkcemi grafického kontextu, který se poté předá správci objektů (viz. podkapitola 3.5.1).

[4] http://cairographics.org/

Kapitola 5

Testování aplikace

Součástí zdrojových kódů jsou i testy rozhraní modulů. Zvolil jsem testování jednotek (angl. unit testing). Testuji jednotlivé funkce rozhraní podle vytvořené specifikace, jedná se tedy o kategorii testování černé skříňky (angl. black-box testing) [7].

Všechny testy je možné spustit pomocí shellového skriptu `tests.sh`, který je ve složce `tests` ve zdrojových kódech aplikace. Skript vyžaduje jako povinný parametr cestu ke složce s moduly a jako nepovinný parametr přepínač `-v`, který zapíná režim s výpisy.

Základní část testů je pro všechny moduly stejná. V ní se testuje registrace a inicializace modulů. Další část je určena pro moduly rozhraní správců objektů, exportů a modulů příkazového řádku. Netestují se moduly importu, protože žádný nebyl naimplementován, ani moduly uživatelských rozhraní. U těchto modulů se provede pouze základní část testu.

5.1 Test registrace a inicializace modulů

Test načte všechny sdílené knihovny, zavolá funkci registrace modulů a postupně provede inicializaci a odstranění všech dostupných modulů.

Testují se následující případy:

- Přítomnost funkce `pm_module_registration` ve sdílené knihovně
- Registrace modulu, vyplnění struktury potřebné pro registraci modulu (typ modulu, verze, jméno modulu, ukazatel na inicializační a deinicializační funkci)
- Inicializace modulu, vyplnění struktury rozhraní modulu
- Mazání modulu

5.2 Test modulů CLI

Tento test ověřuje, zda je modul CLI schopen pracovat jako filtr. Test pracuje se dvěma fiktivními moduly importu a exportu, pomocí nichž testuje chování modulu CLI.

Testují se tyto případy:

- Načítání grafu ve formátu XML ze standardního vstupu, ukládání na standardní výstup
- Načítání grafu ve formátu XML ze souboru, ukládání do souboru

- Import ze standardního vstupu do XML formátu
- Import ze souboru do XML formátu
- Import ze standardního vstupu a export do souboru (do jiného formátu)
- Import ze souboru, export do souboru
- Načítání grafu ve formátu XML a export do souboru

5.3 Test modulů exportu

Test ověřuje schopnost modulu zapisovat jak do souboru, tak i na standardní výstup. Zapisuje se prázdný graf a všechny výstupy se navzájem porovnávají.

Testují se tyto případy:

- Export do souboru
- Export na standardní výstup
- Export do existujícího souboru

5.4 Test modulů správců objektů

Test prověřuje operace správce objektů prováděné nad grafovým kontejnerem, předpokládané výsledky testované operace vyhodnocuje podle obsahu kontejneru. V testu nejsou zahrnuty funkce, které jsou pro konkrétní správce modulů specifické (např. testování funkce pro získání přípojných bodů objektů, jejíž výsledek může být jiný v závislosti na konkrétním objektu).

Testují se tyto případy:

- Registrace objektů, kontroluje se, je-li vyplněna struktura registrace (jméno objektu, typ, cesta k ikoně objektu)
- Unikátnost jmen objektů
- Vytváření vrcholů (pokud takové objekty existují)
- Smazání vrcholů
- Pokus o vytvoření vrcholu s neplatným jménem
- Vytvoření hrany (pokud existuje)
- Smazání vrcholu, ze kterého vede hrana
- Smazání vrcholu, ve kterém končí hrana
- Vytvoření volné hrany
- Napojení volné hrany na vrchol
- Odpojení hrany od vrcholu

- Pokus o vytvoření hrany s neplatným jménem
- Smazání hrany
- Nastavení pozice vrcholu
- Pokus o nastavení pozice hrany
- Nastavení velikosti vrcholu
- Pokus o nastavení záporné velikosti vrcholu
- Získání velikosti a pozice vrcholu
- Pokus o nastavení velikosti hrany
- Získání vlastností objektů
- Nastavení neplatné vlastnosti
- Uložení objektu do XML uzlu
- Načtení objektu z XML uzlu
- Pokus o načtení z chybného uzlu

Kapitola 6

Závěr

V práci byl prezentován návrh modulárního grafového editoru, byla popsána všechna programová rozhraní pro komunikaci s rozšiřujícími moduly a poté byly tyto moduly vytvořeny.

Hlavním cílem práce bylo vytvořit program, který by bylo možno dále snadno rozšiřovat. Tento úkol se mi podařilo splnit, neboť jsem vytvořil funkční aplikaci – program Anve, pomocí kterého jsem nakreslil většinu grafů uvedených v této práci.

Aplikace může s uživatelem komunikovat pomocí dvou zcela různých rozhraní a graf je možno exportovat do čtyř různých grafických formátů – do bitmapového formátu PNG, vektorového SVG, PDF a PostScriptu.

Budoucí vývoj Program Anve byl již od začátku navrhován tak, aby bylo možno pokračovat v dalším snadném rozšiřování jeho funkcí. Do budoucna by proto mohl podporovat více typů grafů (např. Petriho sítě, počítačové sítě, elektrické obvody), mohl by umět importovat některé často používané grafové formáty (např. GraphML nebo Dia) a také podporovat jejich export.

Dále by bylo vhodné rozšířit komunikační rozhraní správců objektů o podporu strukturovaných vlastnosti objektů tak, aby bylo možno vytvářet vlastnosti objektů typu seznam nebo struktura, což by velmi usnadnilo vývoj modulu pro vytváření E-R diagramů a diagramů tříd a doplnit modul grafického uživatelského rozhraní o více funkcí.

Aplikaci je možno velmi snadno lokalizovat, protože jsou všechny textové řetězce v aplikaci a v modulech obaleny funkcí `dgettext` (resp. makrem tr v modulu Qtgui).

Zdrojové kódy programu Anve jsem uvolnil pod licencí GPL a jsou dostupné na webu `http://sourceforge.net/projects/anve/`.

Literatura

[1] Cormen, T. H., Leiserson, C. E. a Rivest, R. L. *Introduction to Algorithms*. 1. vyd. Cambridge MA: The MIT Press, 1990. 1048 s. ISBN 0-262-53091-0.

[2] Demel, J. *Grafy a jejich aplikace*. 1. vyd. Praha: Academia, 2002. 257 s. ISBN 80-200-0990-6.

[3] Gamma, E., Helm, R., Johnson, R. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. vyd. Reading MA: Addison-Wesley, 1995. 416 s. ISBN 0-201-63361-2.

[4] Gross, J. L. a Yellen, J. (ed.). *Handbook of Graph Theory (Discrete Mathematics and Its Applications)*. 1. vyd. Boca Raton FL: CRC Press, 2003. 1192 s. ISBN 1-58488-090-2.

[5] Matthew, N. a Stones, R. *Linux: Programujeme Profesionálně*. 1. vyd. Praha: Computer Perss, 2001. Kapitola 20 – CORBA, s. 691–724. ISBN 80-7226-532-6.

[6] Matula, R. *Grafická reprezentace grafů*. Brno: FIT VUT v Brně, 2008. 49 s. Diplomová práce. Dostupné na: <`http://www.fit.vutbr.cz/study/DP/rpfile.php?id=7241`>.

[7] Myers, G. J. *The Art of Software Testing*. 2. vyd. Hoboken, NJ: Wiley, 2004. 256 s. ISBN 0-471-46912-2.

[8] Sayfan, G. *Building Your Own Plugin Framework* [online]. 2007 [cit. 30. dubna 2011]. Dostupné na: <`http://www.drdobbs.com/cpp/204202899`>.

[9] Tollis, I. G., Battista, G. D., Eades, P. et al. *Graph Drawing: Algorithms for the Visualization of Graphs*. 1. vyd. Upper Saddle River NJ: Prentice Hall, 1998. 397 s. ISBN 0-13-301615-3.

Dodatek A

Obsah CD

Kořenový adresář obsahuje tyto složky:

- `anve` – složka se zdrojovými soubory programu Anve. Obsahuje podadresáře:
  - `doc` – složka s dokumentací generovanou programem Doxygen (příkaz `make doc`)
  - `src` – zdrojové kódy jádra aplikace
  - `include` – hlavičkové soubory jádra aplikace, které jsou potřebné pro vývoj zásuvných modulů
  - `plugins` – zdrojové kódy zásuvných modulů
  - `tests` – složka s jednotkovými testy
  - `examples` – příklady grafů
  - `debug`, `release` – prázdné složky, do kterých se po zadání příkazu `make` (resp. `make release=1`) zkopíruje přeložená aplikace
- `text` – elektronická verze této technické zprávy
- `poster` – plakát prezentující výsledky práce
- `win_release` – verze programu pro Windows (x86)
- `linux_release` – verze programu pro Linux