

Ročníkový projekt

Michal Skřivánek

2002

Prohlášení

Prohlašuji, že jsem tento ročníkový projekt vypracoval samostatně pod vedením Ing. Vladimíra Janouška, Ph.D. a uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Abstrakt

Squeak je volně dostupná implementace jazyka Smalltalk-80. FLTK je knihovna v jazyce C++ pro tvorbu grafických uživatelských prostředí. Tento projekt se zabývá praktickými možnostmi komunikace systému Squeak s okolním programovým prostředím a implementuje smalltalkovské rozhraní pro knihovnu FLTK. Věnuje se také způsobům využívání objektů C++ jazykem C.

Klíčová slova

Smalltalk, Squeak, C, C++, FLTK, rozhraní

Obsah

1	ÚVOD.....	6
1.1	SQUEAK.....	6
1.2	FAST LIGHT TOOL KIT	6
2	INTERAKCE SQUEAKU S OKOLÍM	6
2.1	PRIMITIVNÍ METODY.....	7
2.2	MODIFIKACE VIRTUÁLNÍHO STROJE	7
2.3	ZÁSUVNÉ PRIMITIVNÍ METODY	7
2.4	OBJEKTOVÁ ORIENTACE A EXTERNÍ VOLÁNÍ	8
3	NÁVRH ŘEŠENÍ.....	8
3.1	VÝBĚR VHODNÝCH PROGRAMOVACÍCH PROSTŘEDKŮ	8
3.2	VÝBĚR POUŽITÉ METODY KOMUNIKACE.....	8
3.3	SOUČASNÝ STAV	9
4	IMPLEMENTACE VE SMALLTALKU	10
4.1	ZÁPIS PRIMITIVNÍCH METOD	11
4.2	GENEROVÁNÍ C KÓDU	11
4.3	STRUKTURA TŘÍD ROZHRANÍ.....	13
4.3.1	Třída <i>FLTK</i>	13
4.3.2	Třída <i>FLTKStoredData</i>	13
4.3.3	Třída <i>FLTKRun</i>	14
4.3.4	Třída <i>FlAbstract</i> a ostatní s prefixem <i>Fl</i>	14
4.4	METAPROGRAMOVÁNÍ	15
4.5	ODLIŠNOSTI MEZI PŘÍSTUPEM K FLTK V C++ A VE SQUEAKU	16
5	IMPLEMENTACE V C	16
5.1	PROPOJENÍ C A C++	16
5.2	STRUKTURA ADRESÁŘŮ	17
5.3	PODPŮRNÉ SKRIPTY A AUTOMATICKÉ GENEROVÁNÍ.....	17
5.4	SESTAVENÍ MODULŮ	18
6	PROBLÉMY	19
7	DALŠÍ VÝVOJ PROJEKTU	19
	LITERATURA	20

1 Úvod

Téměř veškeré dnešní počítačové aplikace využívají určité grafické uživatelské prostředí. Pro jednoduché vytváření takových aplikací slouží knihovny podprogramů, tzv. toolkity, které obsahují mnoho užitečných funkcí pro snadnou tvorbu grafických ovládacích prvků, podporu grafického výstupu, uživatelského vstupu a podobně. Pokud použijeme stejných knihoven, docílíme pak jednotného vzhledu i ovládání všech aplikací. Tyto knihovny bývají ve velké většině naprogramovány v jazyce C nebo C++. Aby i aplikace napsané v jiných programovacích jazycích mohly snadno využívat již navržených a implementovaných knihoven, existují pro řadu z vyšších jazyků programátorská rozhraní k těmto knihovnám.

Systém Squeak používá pro své prostředí vlastní grafické uživatelské rozhraní Morphic, odlišné od všech ostatních. Toto rozhraní obsahuje velmi mnoho funkcí, ale pro menší a jednodušší programy je zbytečně rozsáhlé a pomalé.

Cílem projektu je, aby i aplikace napsaná pomocí Squeaku mohla vypadat stejně jako ostatní aplikace v systému, aby měla jednotné ovládání. Projekt vytváří rozhraní mezi Squeakem a používaným toolkitem FLTK. Velikou výhodou jsou snížené nároky na paměť a na rychlost procesoru. Tím by se rozšířily praktické možnosti nasazení Squeaku v reálných systémech. Bylo by také možno používat Squeak jako skriptovací jazyk podobně, jako například TCL/TK.

1.1 Squeak

Squeak je volně dostupná, snadno přenositelná implementace jazyka Smalltalk-80, jehož interpret (virtuální stroj) je napsán také ve Smalltalku. Smalltalk je ryze objektově orientovaný programovací jazyk. Je charakteristický svou jednoduchou syntaxí a otevřeností. Podporuje pozdní vazbu, polymorfismus objektů nezávislý na dědění, automatickou správu paměti, znovupoužitelnost kódu, paralelní programování. Jazyk je integrován s programovacím prostředím a tvoří tak kompaktní celek jazyka, vývojového prostředí a rozsáhlé knihovny tříd. Kompilátor jazyka produkuje tzv. bytekód, který je za běhu překládán virtuálním strojem do hostitelského strojového kódu. Bytekód je jednotný pro všechny podporované platformy a i systémové třídy jsou navrženy tak, aby nebyly závislé na konkrétním operačním systému. Samotný Squeak je podporován na platformách Macintosh, Windows, Unix, OS/2, Windows CE a mnoha dalších.

1.2 Fast Light Tool Kit

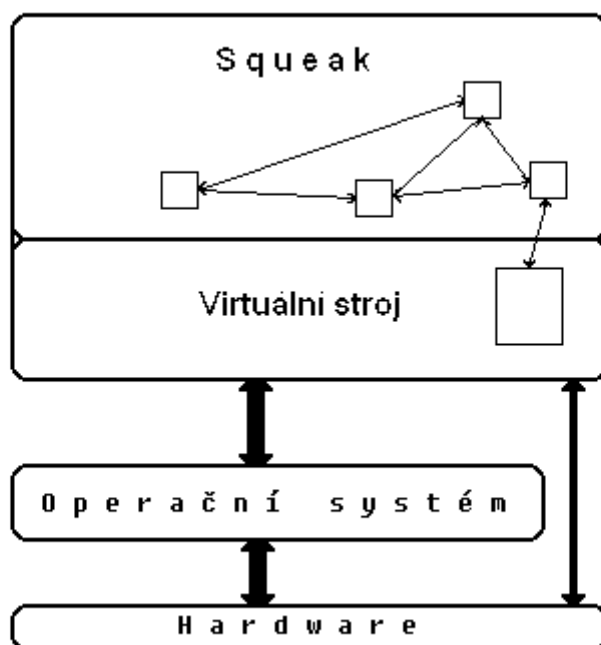
FLTK (Fast Light Tool Kit) je knihovna pro tvorbu grafického uživatelského prostředí pro systémy X Windows, MacOS a Microsoft Windows. Je vytvořena v jazyce C++ a distribuována s licencí LGPL (GNU Library Public License).

2 Interakce Squeaku s okolím

Squeak, stejně jako ostatní implementace Smalltalku, je poměrně uzavřený ke svému okolí. Není vázán na hostitelský operační systém, protože veškeré potřebné funkce poskytuje jeho knihovna tříd. Pro některé základní operace, jako je nízkourovňová práce se soubory nebo s periferními zařízeními, je však nutné opustit systém Squeaku a komunikovat přímo s operačním systémem či hardwarem.

2.1 Primitivní metody

Ve Squeaku existují tzv. primitivní metody, které slouží jako vstupní body do virtuálního stroje. Protože ve Squeaku nelze provést nic jiného než zaslat jinému objektu zprávu, jsou primitivní metody na úrovni zdrojového textu podobné obyčejným metodám jakékoliv třídy. V jejich těle je však uvedena speciální direktiva pro virtuální stroj. Potom se při vyvolání takové metody neprovede její vyhodnocení, ale přímo samotný interpret provede definovanou akci.



Obr. 1: Komunikace Squeaku s okolím

Primitivní metody se používají i pro počítání základních aritmetických operací v pevné a pohyblivé řádové čárce, kde je kladen důraz více na rychlost výpočtu než na korektní provedení všech zpráv.

2.2 Modifikace virtuálního stroje

Klasické primitivní metody obsahují číselný index do tabulky pomocných funkcí, které jsou součástí virtuálního stroje. Případné rozšíření o novou funkci ale vyžaduje vložit nový kód přímo do zdrojového kódu virtuálního stroje a znovu jej přeložit. Takové řešení není příliš šťastné, neboť stroj se pro každou platformu liší, takže by bylo nutné provést překlad na všech. Přidávání dalších a dalších funkcí by také vedlo k přílišnému nárůstu objemu jinak jednoduchého stroje.

Tento způsob je využíván staršími verzemi Squeaku a je odvozen z definice Smalltalku-80 a jeho prvních implementací. Dnes se používá jen u nejzákladnějších operací. Bližší popis je uveden v [5].

2.3 Zásuvné primitivní metody

Od Squeaku verze 2.8 je využíváno tzv. zásuvných primitivních metod (pluggable primitives). Tyto metody jsou již identifikovány jménem. Zdrojový kód už není přímo součástí zdrojového kódu virtuálního stroje, ale je vkládán jako zásuvný modul (plugin). Tento modul lze k němu při

překladu připojit nebo ho používat odděleně, jako dynamickou knihovnu. Virtuální stroj Squeaku v sobě obsahuje mechanismy, jak transparentně získat funkci z modulu ve formě sdílené knihovny na platformě, která toto umožňuje. Jsou to tzv. DLL (Dynamic Linked Library) pod operačními systémy Microsoft Windows nebo DSO (Dynamic Shared Object) pod Unixy. Rozhraní pro přístup k okolnímu systému pak tvoří část napsaná ve Smalltalku a soubor s příslušnou knihovnou.

Sdílená knihovna je výhodná pro svou snadnou aktualizaci bez nutnosti nového překladu virtuálního stroje nebo pro moduly dodávané třetí stranou. Naproti tomu statický překlad je vhodný pro často využívané moduly, například pro již zmíněnou aritmetiku, pro grafický výstup, práci se soubory apod.

Možnost volat funkce z externích knihoven otevřela Squeaku cestu k využívání mnoha funkcí jiných programů. Jako příklad mohu uvést rozhraní k Apple QuickTime [6]. Implementace tohoto rozhraní ve Squeaku je velmi vhodná ke studiu při tvorbě nových zásuvných modulů.

2.4 Objektová orientace a externí volání

Protože jak virtuální stroj, tak zásuvné moduly jsou napsány v jazyce C, přicházíme na této úrovni o jednu z nejdůležitějších vlastností Smalltalku, o objektově orientovaný přístup. Pro volání funkcí operačního systému to není na závadu, ale u využívání rozsáhlejších částí externího kódu je to velký nedostatek. Možná východiska při propojení na jiný objektově orientovaný jazyk jsou popsána v kapitole 5.1.

3 Návrh řešení

3.1 Výběr vhodných programovacích prostředků

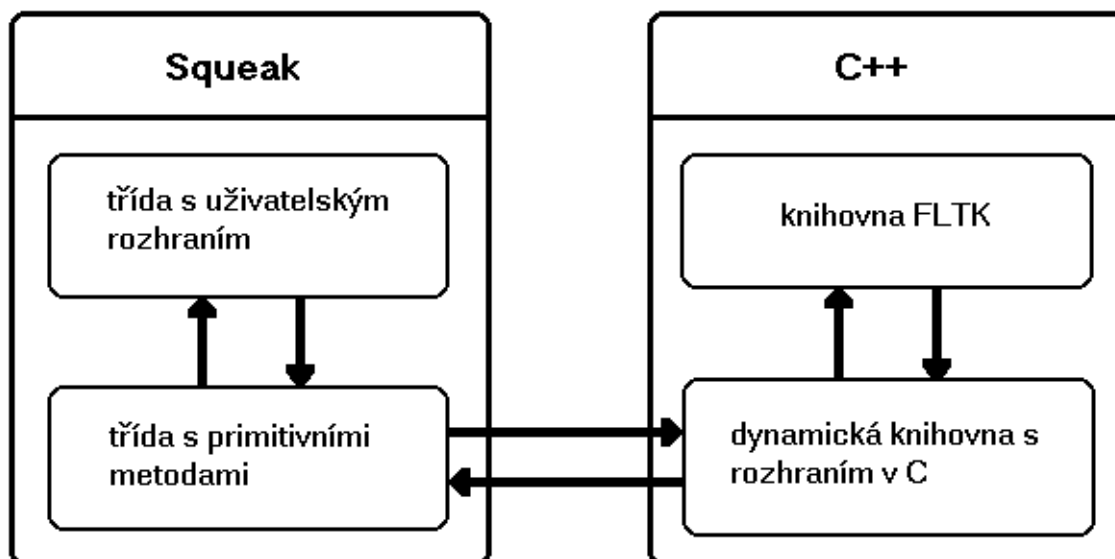
Prostudoval jsem jeden ze dvou nejpoužívanějších toolkitů pro tvorbu uživatelských rozhraní v prostředí operačních systémů Linux, knihovnu Gtk. Gtk je základem uživatelského prostředí GNOME a je napsán v jazyce C. Původně se jevil jako nejvhodnější pro zadaný problém. Při důkladnějším seznámení jsem však toto řešení po konzultaci s vedoucím projektu zavrhl. Gtk je pro účely tohoto projektu příliš náročný a rozsáhlý. I pro vytvoření jednoduché ukázkové aplikace by bylo nutné implementovat z něj spoustu funkcí. Za cílový toolkit byl tedy vybrán FLTK, je sice v jazyce C++, což přineslo nové problémy (viz kapitola 5.1), ale jeho jednoduchost, rychlost a paměťové nároky jsou více v souladu s cílem projektu.

Dalším krokem bylo prostudovat již existující produkty zabývající se touto problematikou. Ke Gtk existuje celá řada rozhraní pro jiné programovací jazyky než C. Všechny jsou odvozeny z řešení pro jazyk Python, program PyGTK[9]. Jedná se však, v důsledku používání Gtk, o poměrně komplikovaný projekt. Také způsob volání externích knihoven v Pythonu a ve Smalltalku je podstatně jiný.

Nejpřínosnější informace jsem však získal z projektu QtC[8]. Je to implementace rozhraní jazyka C k toolkitu Qt v jazyce C++. Od roku 1997 sice již není aktivně udržován, ale pro studijní účely je ideální. Z něj je převzata myšlenka na komunikaci mezi C a C++ a obecně mezi neobjektovým a objektovým jazykem.

3.2 Výběr použité metody komunikace

Z kapitoly 2 vyplývá, že modifikace virtuálního stroje není žádoucí. Nejvhodnější tedy je využít stávajícího a potřebné záležitosti řešit sdílenými knihovnami. Na tomto základě jsem navrhl systém komunikace mezi Squeakem a FLTK, znázorněný na obrázku 2.



Obr. 2: Komunikace mezi Squeakem a FLTK

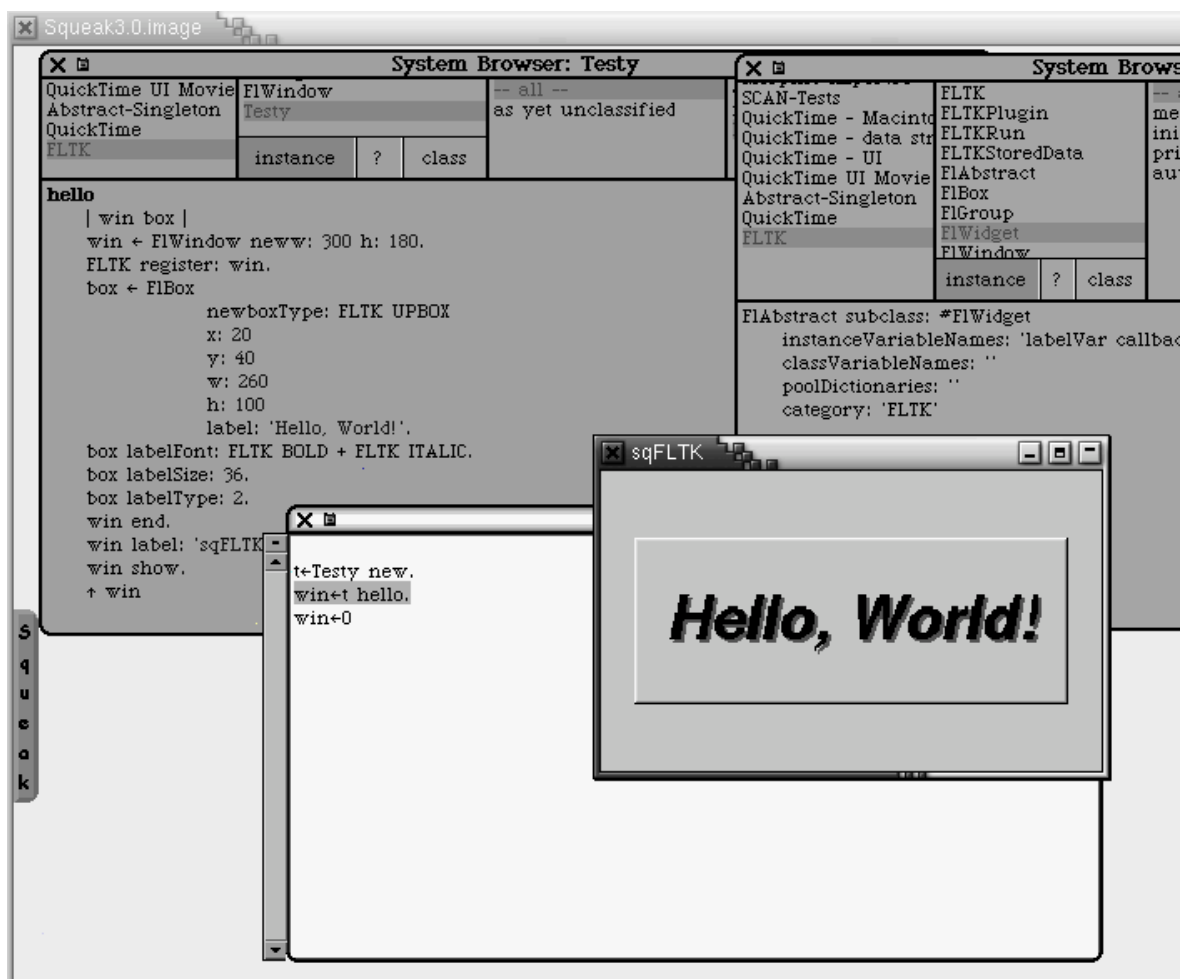
Na nejvyšší úrovni je třída definující uživatelské rozhraní pro ostatní objekty. Při konstrukci nových aplikací budou využívány pro interakci s uživatelem pouze metody této třídy. V rámci ročníkového projektu je toto rozhraní co nejvěrnější podobou rozhraní samotného FLTK. Nutné odlišnosti vyplývají z různého filozofického pojetí jazyků Smalltalk a C. Snahou je, aby rozdílnost práce s rozhraním ve Squeaku a v jazyce C byla minimální. To je blíže rozebíráno v kapitole 4.5. V budoucnu je vhodné postavit nad tuto úroveň ještě jedno, velmi snadno použitelné, jednoduché rozhraní.

Na nižší úrovni se nachází třída s primitivními metodami, popsány v kapitole 2.1. Primitivní metoda tvoří prostředníka mezi klasickou metodou Smalltalku a funkcí v jazyce C. Každá metoda třídy uživatelského rozhraní má jednu svoji primitivní metodu.

V části C++ se na spodní úrovni nacházejí funkce, které odpovídají výše uvedeným primitivním voláním. Musejí být navrženy tak, aby na jedné straně plnily funkci externí metody Squeaku a na druhé straně byly obyčejnou funkcí v C++. Knihovna FLTK se pak používá stejným způsobem, jako kdyby se pro ni psala běžná aplikace a není nijak modifikovaná.

3.3 Současný stav

Při podrobnějším zkoumání se ukázaly nové, zpočátku nepředpokládané potíže. Přesto se podařilo ukázat, že cíle ročníkového projektu i navržené řešení jsou splnitelné. Při samotném vytváření projektu došlo pouze k malým implementačním změnám, detailně rozebíraným v kapitolách 4 a 5. Projekt je ve stavu, kdy je možné předvést funkčnost jednoduché aplikace. Na obrázku 3 je vidět ukázkový program „Hello World” z distribuce FLTK přepsaný do systému Squeak.



Obr. 3: Squeak s aplikací využívající FLTK

Jako hlavní cílová a testovací platforma byl zvolen operační systém Linux. Testovací aplikace byla úspěšně spuštěna na distribucích SuSE 6.4 a Red Hat 7.0 bez sebemenších úprav. Protože jak Squeak, tak FLTK se neomezuje pouze na Linux, ani produkt tohoto ročníkového projektu na něj není nijak vázán a po pouhém přeložení příslušných knihoven byl otestován i pod operačním systémem Solaris. Neměl by být problém přenést produkt na jakoukoliv platformu, která je podporována Squeakem a FLTK a která umožňuje využívat mechanismus sdílených knihoven.

4 Implementace ve Smalltalku

Tato kapitola podrobně probírá, jaké techniky programování byly použity při řešení projektu, respektive jeho části napsané v jazyce Smalltalk v systému Squeak.

Jelikož Squeak je neustále poměrně rychlým tempem rozvíjen, bylo při práci na ročníkovém projektu použito Squeaku nejnovější stabilní řady 3.0, konkrétně sestavení 3545. Vývoj ve Smalltalku probíhal pod operačním systémem Linux i Microsoft Windows.

4.1 Zápis primitivních metod

Základem pro volání mimo Squeak jsou primitivní metody. Jak bylo naznačeno v kapitole 2.1, jsou psány jako běžné metody, ale virtuální stroj s nimi pracuje jinak. Aby je interpret rozpoznal, mají v těle uvedeno, že se jedná o primitivní volání pomocí `<primitive: >`. Na obrázku 4 je ukázka takovéto metody převzatá z třídy `Foo2` v kategorii `VMConstruction-TestPlugins`. Jedná se o metodu, která přičítá hodnotu `x` k instanční proměnné objektu třídy `Foo2`.

primFooIntegerSumWith: x

```
<primitive: 'primFooIntegerSumWith' module: 'FooPlugin2'>
↑"FooPlugin2
  doPrimitive: 'primFooIntegerSumWith:'
  withArguments: {x}" 'Whoops!'
```

Obr. 4: Metoda `primFooIntegerSumWith:` třídy `Foo2`

Direktiva pro překladač `<primitive:>` udává, jaký je název volané funkce a ve kterém modulu se nachází. Obvykle se volí stejný název jako má primitivní metoda. Část uvedená za direktivou se normálně neprovádí. Pouze při chybě volání externí funkce se provede vše, co je za `<primitive:>`, jako kdyby funkce proběhla úspěšně a funguje tedy jako náhradní kód. K selhání může dojít buď již při vyhledávání funkce v knihovně (knihovnu nelze nalézt nebo knihovna neobsahuje funkci s tímto názvem), nebo v samotné implementaci funkce. V prvním případě vypíše Squeak (program spuštěný pod operačním systémem) na chybový výstup zprávu o nenalezené knihovně.

Jako náhradní řešení může být použito tak jako v tomto případě vrácení textu `'Whoops'`, nebo častěji vyvolání chybového okna pomocí zaslání zprávy `self primitiveFailed`. Pokud je možné provést požadovanou funkci metody přímo ve Smalltalku (třeba součet dvou čísel v plovoucí řádové čárce), uvede se jako náhradní řešení a metoda je funkční, i když chybí patřičná knihovna, jen provedení potom trvá déle. V případě našeho projektu toto není možné a proto ve všech případech je zaslána zpráva `primitiveFailed`.

Pro základní studium je vhodné prozkoumat třídy `Foo2` a `FlippyArray2` z kategorie `VMConstruction-TestPlugins`. Velmi užitečné je také prozkoumat poněkud složitější využití primitivních metod pro síťovou komunikaci ve třídě `Socket` kategorie `Network-Kernel`.

Zvyklostí je umisťovat primitivní metody do kategorie `primitives` třídy, v které jsou volání využívána. Jejich název by měl být stejný jako název metody, která je volá, jen je rozšířen o prefix `prim`. V projektu se používá primitivních volání pro každou metodu `FLTK` a jejich název i umístění odpovídá výše uvedeným doporučením. Při vytváření se jedná o víceméně rutinní opisování a kapitola 4.4 popisuje, jak jsou primitivní metody vkládány automaticky.

Pro volání cizích knihovnických funkcí lze použít i `FFI` (`Foreign Function Interface`, kategorie `FFI-Kernel`). Rozhraní je však postaveno tak, aby umělo opravdu všechno, jeho používání je dost těžkopádné a doba provedení volání je dlouhá. Z těchto důvodů není použito pro naše řešení.

4.2 Generování C kódu

Jak bylo zmíněno již v úvodu, virtuální stroj Squeaku je také napsán ve Smalltalku. To by nebylo možné, pokud by neexistoval způsob, jak generovat kód nějakého nižšího jazyka. Z toho důvodu byla napsána třída `CCodeGenerator` z kategorie `VMConstruction-Translation to C`, která překládá Smalltalkovský kód do zdrojového textu v jazyce C. To

umožňuje využívat při vývoji pohodlí Smalltalku a pro běh výsledného kódu efektivitu C. Například interpret Squeaku je možné získat zasláním zprávy `translate` třídě `Interpreter` (tedy instanci třídy `Interpreter class`).

Velmi výhodné by bylo použít stejný přístup i pro psaní zásuvných modulů. Pro toto existuje ve Squeaku třída `PluggableCodeGenerator`, následník třídy `CCodeGenerator`. Liší se jen doplněním vazeb produkovaného kódu na virtuální stroj. V posledních verzích Squeaku přibyla třída `TestCodeGenerator`, druhá generace překladače do C. Obsahuje koerci typů, celkově zjednodušuje přístup k datovým typům C a přístup k objektům Smalltalku v generovaném kódu.

Pro podporu psaní zásuvných modulů ve Smalltalku existuje třída `InterpreterPlugin`. Stejně jako její pomocí napsané moduly je zařazena do kategorie `VMConstrucion-Plugins`. Následníci této třídy obsahují metody, které budou přeloženy do jazyka C. Metody mají přímou návaznost na primitivní metody z předcházející kapitoly, jsou vlastně jejich implementací. `InterpreterPlugin` resp. `TestInterpreterPlugin` obsahuje metodu `translate`, která provede vlastní překlad metod pomocí `CCodeGenerator` resp. `TestCodeGenerator`.

Pokračujme v uvedeném příkladu třídy `Foo2`. Je vytvořena třída `Foo2Plugin` jako následník `TestInterpreterPlugin` a obsahuje metodu se stejným názvem jako jí příslušná primitivní metoda. Na obrázku 5 je uveden její kód.

primFooIntegerSumWith: x

```
|rcvr myInteger|
rcvr ← self
primitive: 'primFooIntegerSumWith'
parameters: #(SmallInteger)
receiver:   #Foo2.

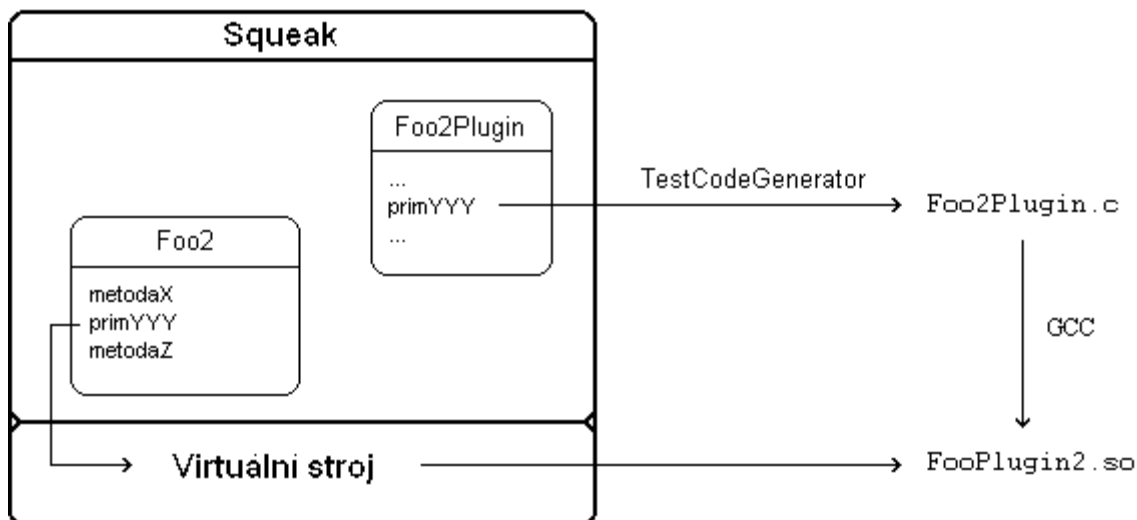
myInteger ← (rcvr asIf: Foo2 var: 'myInteger') asValue: SmallInteger.
↑ (x + myInteger) asOop: SmallInteger
```

Obr. 5: Metoda `primFooIntegerSumWith:` třídy `Foo2Plugin`

Zpráva `primitive:parameters:receiver:` definuje název funkce a použité parametry pro jazyk C a vlastní akci. Typ parametrů se specifikuje jako typ ve Smalltalku a generátor použije odpovídající typ v C, zde to bude `int`, a provede nezbytné převody. Výsledkem je funkce, která má jeden vstupní parametr, který přičte k instanční proměnné `myInteger` objektu třídy `Foo2` a jako výsledek vrátí součet v nové instanci třídy `SmallInteger`.

Uvnitř metody lze využívat třídy `InterpreterProxy`, která má ve vygenerovaném C kódu Smalltalku odpovídající implementaci a umožňuje komunikaci s virtuálním strojem. Poskytuje přímý přístup k instancím pomocí ukazatele v C. V metodě lze také s výhodou využít zprávy `cCode:`, kdy je možno do vygenerovaného textu vložit vlastní řetězec, který je mimo kontrolu Squeaku. Například pomocí: `self cCode: 'i += 5'...`

V ročníkovém projektu jsou všechny primitivní metody implementovány pomocí metod třídy `FLTKPlugin`, odvozené od `TestInterpreterPlugin`. Velká většina z nich je generována automaticky (viz kapitola 4.4). Po provedení `FLTKPlugin translate` dostaneme zdrojový text v jazyce C zásuvného modulu, který je potom využíván Squeakem (viz obr. 6). Detailům překladu se blíže věnuje kapitola 5.4.



Obr. 6: Generování zásuvného modulu

4.3 Struktura tříd rozhraní

Samotné rozhraní FLTK ve Squeaku se skládá z části starající se o inicializaci a globální záležitosti (třída `FLTK`), z pomocné třídy umožňující sdílet data mezi Squeakem a FLTK, z části realizující reakce na události (třída `FLTKRun`) a ze tříd, které odpovídají třídám ve FLTK (`FlWidget`, `FlWindow` atd.).

4.3.1 Třída `FLTK`

Inicializaci rozhraní po spuštění systému má v kompetenci třída `FLTK`. Její oblast působnosti má globální charakter, proto se využívá jen jediné třídní instance `FLTK class`. Poskytuje také konstanty, které jsou v původní knihovně v C (`#define`), protože podobný prostředek Smalltalk nemá. Způsob použití konstant ve Squeaku je navržen tak, aby se výrazně nelišil od používání v jazyce C (viz kapitola 4.5). Třída `FLTK` obsahuje metody, které mají stejný název jako konstanty ve FLTK, jejich jedinou funkcí je vrátit hodnotu původní konstanty. Jelikož jejich definice je v hlavičkových souborech FLTK, je také využito jejich automatického generování (kapitola 4.4).

Třída se také stará o registraci pro tzv. finalizaci objektů (dokončení a uvolnění paměti již dále nepotřebné instance).

4.3.2 Třída `FLTKStoredData`

Původní FLTK používá zejména při práci s řetězci pouze hodnotu ukazatele, nevytváří si svoje vlastní kopie. Protože objekty ve Squeaku nejsou v žádném případě pevně vázány na svoje umístění v paměti, mohou se v libovolném okamžiku (zejména při uvolňování pomocí `garbage-collectingu`) ocitnout na jiném místě. Aby nedocházelo k těmto problémům, byla navržena třída `FLTKStoredData`, která vytváří kopie požadovaných dat v paměťovém prostoru mimo Squeak. Obsahuje zejména metodu `store:`, která provede uložení dat, metodu `loadData` pro opětovné načtení a `release` pro uvolnění této kopie. Aby měla třída přístup mimo Squeak, je její významná část součástí zásuvného modulu generovaného pomocí `FLTKPlugin`.

Při ukládání dat je mimo Squeak volána standardní funkce `malloc` pro alokaci nového prostoru paměti a ukládaný objekt je na toto místo přepokopírován. Instanční proměnná `instancePointer` potom obsahuje jeho novou adresu ve formě `ByteArray` a proměnná

`dataSize` jeho velikost v bytech. Při načítání dat je `instancePointer` opět chápán jako ukazatel. Pomocí `InterpreterProxy instantiateClass:` je vytvořena ve Squeaku nová instance stejné velikosti a data překopírována. Uvolnění dat spočívá v navrácení paměti do systému C funkcí `free`.

Pro snadnější práci se při ukládání odlišují běžná data a řetězce pomocí `isKindOf: String`. Řetězce jsou doplněny koncovým nulovým znakem. Stejně při načítání lze použít `loadData` a `loadString`.

4.3.3 Třída `FLTKRun`

Program ve FLTK typicky obsahuje volání funkce `Fl::run()`. Tím se spouští cyklická kontrola událostí a uživatelské prostředí se vykreslí a začne reagovat. Knihovna FLTK, jako všechny ostatní toolkity, je postavena na událostech. Čeká ve smyčce, až nastane nějaká událost (např. stisk klávesy, pohyb myši) a zavolá obslužnou funkci, která nějak zareaguje (napíše znak na obrazovku, překreslí aktuální okno apod.). Obslužné funkce (callback functions) se registrují u požadovaného objektu pomocí `FlWidget::callback()` a při příchodu události je zainteresovaný objekt zavolá. Tyto funkce má v rozhraní za úkol třída `FLTKRun`.

Nelze však ze Squeaku zavolat funkci `Fl::run()` a dále pokračovat v práci. Squeak sice obsahuje podporu pro paralelní procesy, ale to se netýká externích funkcí v zásuvných modulech, protože ty běží na úrovni virtuálního stroje. FLTK naštěstí poskytuje funkci `Fl::check()`, která neobsahuje smyčku, jen zjistí, zda nastala událost (potom zavolá obslužné funkce) a pokud ne, ihned skončí. Bylo tedy nutné prozkoumat problematiku paralelních procesů ve Squeaku (třídy `Process`, `Delay` a `ProcessorScheduler` v kategorii `Kernel-Processes`) a instalovat nový proces, který bude cyklicky provádět volání `Fl::check()`.

Situace s obslužnými funkcemi je komplikovanější, jelikož neexistuje možnost, jak provést volání do Squeaku iniciované zvenčí. Inspirací pro řešení bylo studium knihovny QtC a třída `Socket` a její třída pro generování modulu `SocketPlugin`. V kategorii `Kernel-Processes` se nachází ještě třída `Semaphore`, jenž umožňuje pomocí vyslaného signálu „probudit“ pozastavený proces. Jedná se pouze o jednobitovou informaci. Metoda pro vyslání signálu je dostupná i v třídě `InterpreterProxy` a tedy i z externího kódu. Byl vytvořen další proces, tentokrát pozastavený, který čeká až bude „probuzen“, poté provede volání zpět do externího kódu, aby zjistil, kterou obslužnou funkci má provést. Obslužné funkce jsou registrovány ve Squeaku pod číselným indexem (metodami `newCallback:` a `removeCallback:`). Při registrování ve FLTK se používá nově vytvořené jednoúčelové obslužné funkce, která reaguje na nastalou událost vysláním signálu Squeaku. Kromě toho si musí „pamatovat“ číselný index obslužné funkce ve Squeaku.

4.3.4 Třída `FlAbstract` a ostatní s prefixem `Fl`

Ostatní třídy v rozhraní mají stejnou hierarchii jako ty původní ve FLTK a začínají na `Fl`. Jen třída `FlAbstract` je výjimkou. Implementuje obecné záležitosti a je proto předchůdcem všech ostatních tříd `Flxxx` a třídy `FLTKStoredData`.

Při vytvoření nové instance `Flxxx` se vyvolá primitivní metoda, jejíž externí kód v zásuvném modulu použije C++ operátoru `new` pro vytvoření nové instance odpovídající třídy v C++. Ukazatel na nový objekt je uložen do instanční proměnné `instancePointer` ve Squeaku. Tento ukazatel se bude používat pro další komunikaci s objektem. Jednoduché metody přesně odpovídající metodám ve FLTK často vypadají tak jednoduše jako na obrázku 7. `FlAbstract` obsahuje také metody pro kontrolu platnosti ukazatele `assigned` a `assignedOrError` a pomocné metody pro automatické generování kódu (kapitola 4.4).

System Browser: FlWidget				
SUnit-Tests	FlAbstract	-- all --	labelFont:	
Framework-Download	FlBox	methods	labelSize:	
Morphic-Imported	FlGroup	initialize	labelSize:	
SCAN-Tests	FlWidget	private	labelType:	
FLTK	FlWindow	autogenerated primitives	labelType:	
	instance	?	class	
labelFont: newfont ↑ self assignedOnError primLabelFontSet: newfont				

Obr. 7: Metoda pro nastavení nového fontu

4.4 Metaprogramování

Množství zdrojového textu má stejný základ nebo se dá odvodit z ostatních informací. Například primitivní metody mají ve svém těle jen specifikaci parametrů, které jsou znovu uvedeny v metodách pro generaci C kódu. I názvy jsou odvozené a velmi podobné. To vedlo k myšlence použít automatickou generaci některých metod. Situace, kdy program vytváří zdrojový text jiného programu nebo části sebe sama, bývá nazývána metaprogramování. Jazyk Smalltalk je pro to velmi vhodný, jeho inkrementální kompilátor umožňuje překládat a modifikovat objekty přímo za běhu. V projektu je využito automatické generování zdrojového textu pro většinu primitivních metod i pro jejich implementaci v zásuvném modulu. To je možné především proto, že se jedná v zásadě pouze o zprostředkování funkce poskytované knihovnou FLTK.

Metody pro generování jsou umístěny do FlAbstract class. Od ní je dědí třída FLTKStoredData a třídy typu Flxxx. V každé třídě se potom nachází metoda generatePrimitives, kde se uvedou názvy generovaných metod.

Mějme třídu FlWidget. Přidáním řádku self generateProcedure: 'LabelFontSet:' do její třídni metody generatePrimitives se přidá kategorie autogenerated primitives a vloží se metoda s primitivním voláním. Zároveň se do třídy FLTKPlugin přidá kategorie se stejným názvem, jako má třída, z které se generuje (zde to bude flwidget) a implementační metoda se správným počtem parametrů. Vygenerování metod všech tříd jednoduše docílíme například spuštěním řádku FlAbstract allSubclassesDo: [:i | i generatePrimitives]. Pro komplikovanější případy, kdy je potřeba předat i ukazatel v jiné instanční proměnné, může generovaný kód vypadat jako na obrázku 8.

```
primFlWidgetSetCallback
| rcvr callbackVarPtr |
rcvr ← self
    primitive: 'primFlWidgetSetCallback'
    parameters: *()
    receiver: #FlWidget.
callbackVarPtr ← (rcvr asIf: FlWidget var: 'callbackVar') asOop: FLTKStoredData.
self cCode: 'FlWidgetSetCallback(getPtr(rcvr), getPtr(callbackVarPtr))'.
↑rcvr
```

Obr. 8: Metoda primFlWidgetSetCallback třídy FLTKPlugin

Také konstanty ve třídě `FLTK` jsou generovány automaticky. Používá se seznamu konstant a jejich hodnot v souboru `sqConstants.in`. Ten je vytvořen pomocnými skripty při překladu části projektu v jazyce C, způsobem blíže popsáným v kapitole 5.3. Konstanty jsou načteny ze souboru a přímo vkládány do třídy `FLTK` jako její metody. Pro větší podobnost s původním `FLTK` jsou názvy metod velkými písmeny, což není ve `Smalltalku` příliš obvyklé. Jiné možné způsoby ale byly pro praktické použití příliš těžkopádné.

4.5 Odlišnosti mezi přístupem k `FLTK` v `C++` a ve `Squeaku`

Rozhraní je koncipováno tak, aby způsob použití byl pokud možno stejný jako původní v `C++`. I když menším odlišnostem není možné se vyhnout, vzhledem k rozdílnosti obou jazyků.

Jedním z výrazných rozdílů je absence konstruktoru ve `Smalltalku`. `Smalltalk` nepotřebuje žádné alokace paměti pomocí `new` jako `C++`. V projektu je konstruktor nahrazen metodou pro vytvoření nové instance dané třídy a provedení inicializace instance. Ta spočívá ve volání externího kódu `C++`, který vytvoří nový objekt pomocí `new` a vrátí ukazatel na něj. Ten je uložen do proměnné `instancePointer`. Pro různý počet parametrů v jednom konstruktoru je třeba ve `Squeaku` udělat speciální metodu s jiným názvem pro každou možnou formu jeho volání.

Ve `Squeaku` také neexistuje destruktorka ani operátor `delete`. Nepotřebné instance se ve `Squeaku` ruší automaticky probíhajícím `garbage-collectingem` na pozadí. V průběhu práce na projektu se ukázalo, že rušení vytvořených objektů je poměrně vážným problémem, který dosud není uspokojivě vyřešen.

`FLTK` také má globální statické metody třídy `Fl` (například již zmíněnou `Fl::run()`) a globální funkce. Ve `Squeaku` takovou funkci plní třída `FLTK`. Tam jsou přístupné i konstanty, kterých je ve `FLTK` celá řada. Jednotlivé číselné hodnoty konstant jsou implementovány jako metody se shodným názvem, vracející příslušné číslo. Zapsání `FL_CONST` v `C++` potom odpovídá použití konstrukce `FLTK CONST` ve `Squeaku`.

Metody mají v obou jazycích stejné názvy. Ve `Squeaku` však dodržují zvyk psaní identifikátorů a selektorů metod, začíná malým písmenem, první písmeno každého dalšího slova je velké, víceslovné názvy jsou psány dohromady. Pokud má metoda ve `FLTK` více parametrů, jsou do selektoru přidány stejné nebo podobné názvy těchto parametrů. Například deklarace metody pro posun okna `Fl_Window::move(int x, int y)` by zapsána ve `Squeaku` vypadala takto: `FlWindow>>movex: paramx y: paramy`. „Konstruktory“ mají navíc v názvu slovo `new`. Vytvoření nového okna o šířce 300 a výšce 180 pixelů pomocí `Fl_Window *win = new Fl_Window(300,180)` by bylo přepsáno do `Squeaku` jako `win ← FlWindow neww: 300 h: 180`.

5 Implementace v C

Kapitola se zabývá použitými technikami při psaní obslužné části rozhraní a problematikou překladu. Zdrojové texty byly překládány `GCC` (`GNU C Compiler`) verze 2.95. Pomocné skripty využívají standardní unixové programy `make`, `ksh`, `gawk`, `sed` apod.

5.1 Propojení C a C++

Pro vzájemnou spolupráci kódu v `C` a `C++` je použito stejného principu jako v projektu `QtC`[8]. Ke každé metodě v `C++` je vytvořena obyčejná funkce, která je pomocí `extern C` potom dostupná z kódu v jazyce `C`. Pro rozlišení původní třídy musí být funkce rozšířena o tuto informaci. Například použitím názvu třídy jako prefixu názvu funkce. Jako vstupní parametr funkce musí být

přítomen i ukazatel na objekt, se kterým se pracuje. Potom lze pracovat s objekty podobně, i když jazyk C nepodporuje polymorfismus ani zapouzdření.

Mějme například volání metody `move` objektu `win` třídy `Fl_Window` v C++:

```
// volání metody Fl_Window::move(int x, int y);
win->move(10, 20);
```

Transformace do jazyka C se pak skládá z vytvoření pomocné funkce v C++:

```
extern "C" void FlWindowMove(Fl_Window *win, int x, int y) {
    win->move(x, y);
}
```

Samotné volání se potom provede z jazyka C pomocí

```
FlWindowMove(win, 1, 2);
```

Důležitým rozdílem je, že informaci o třídě objektu si musíme v C uchovávat sami.

5.2 Struktura adresářů

Zdrojové texty jsou umístěny ve třech podadresářích, podle rozsahu své působnosti. Největší část kódu se zabývá propojením C na C++, zdrojové soubory jsou umístěny v podadresáři `interface`. Ze Squeaku vygenerovaný zdrojový soubor pro překlad zásuvného modulu se nachází v podadresáři `plugin`, tam jsou umístěny i hlavičkové soubory pro komunikaci s virtuálním strojem příslušné platformy. V podadresáři `constants` se nacházejí pomocné soubory pro vytvoření vstupního souboru s konstantami pro Squeak.

V propojovací části se nacházejí funkce v C++ podle předchozí kapitoly. Jsou umístěny v souborech podle příslušnosti implementovaných metod k dané třídě. Funkce implementující rozhraní k třídě `Fl_Window` jsou tedy v souboru `sqFl_Window.cpp`. V podadresáři `autoh` jsou automaticky generované hlavičkové soubory, obsahující definici `extern "C"`. V propojovací části jsou i soubory implementující podporu pro obslužné funkce (callback) a kontrolu událostí. Při implementaci nových metod FLTK stačí přidat příslušnou obslužnou funkci do odpovídajícího souboru.

5.3 Podpůrné skripty a automatické generování

Projekt obsahuje části, které lze generovat automaticky. Jedná se především o hlavičkové soubory k souborům v podadresáři `interface` a o soubor s konstantami FLTK pro Squeak.

Hlavičkové soubory se generují pomocným skriptem `makeh.sh`. Ten vytvoří ze zdrojového textu deklarace funkcí, přidá správný hlavičkový soubor FLTK pro danou třídu a deklaruje implementované funkce C++ tak, aby je bylo možné volat z jazyka C.

Konstanty se získávají komplikovanější cestou. Ve FLTK začínají povinně všechny konstanty prefixem `FL_` a jsou psány většinou celé velkými písmeny. Skript `makeconst.sh` z hlavičkových souborů FLTK `Enumerations.H` a `Fl_Export.H` vybere konstanty začínající na `FL_` a vytvoří zdrojový text v jazyce C++ tisknoucí název a hodnotu konstant. Generovaný řádek pro konstantu `FL_SHADOW_BOX` vypadá takto:

```
printf("SHADOW_BOX ^%d\n", FL_SHADOW_BOX);
```

Zdrojový text se poté přeloží a spustí. Jeho standardní výstup se přesměruje do souboru `sqConstants.in`, který je posléze využíván při instalaci rozhraní ve Squeaku. Jeden řádek vytištěný programem odpovídá přímo zdrojovému textu metody ve Smalltalku. Znak `^` je ekvivalentní znaku `↑` v prostředí Squeaku.

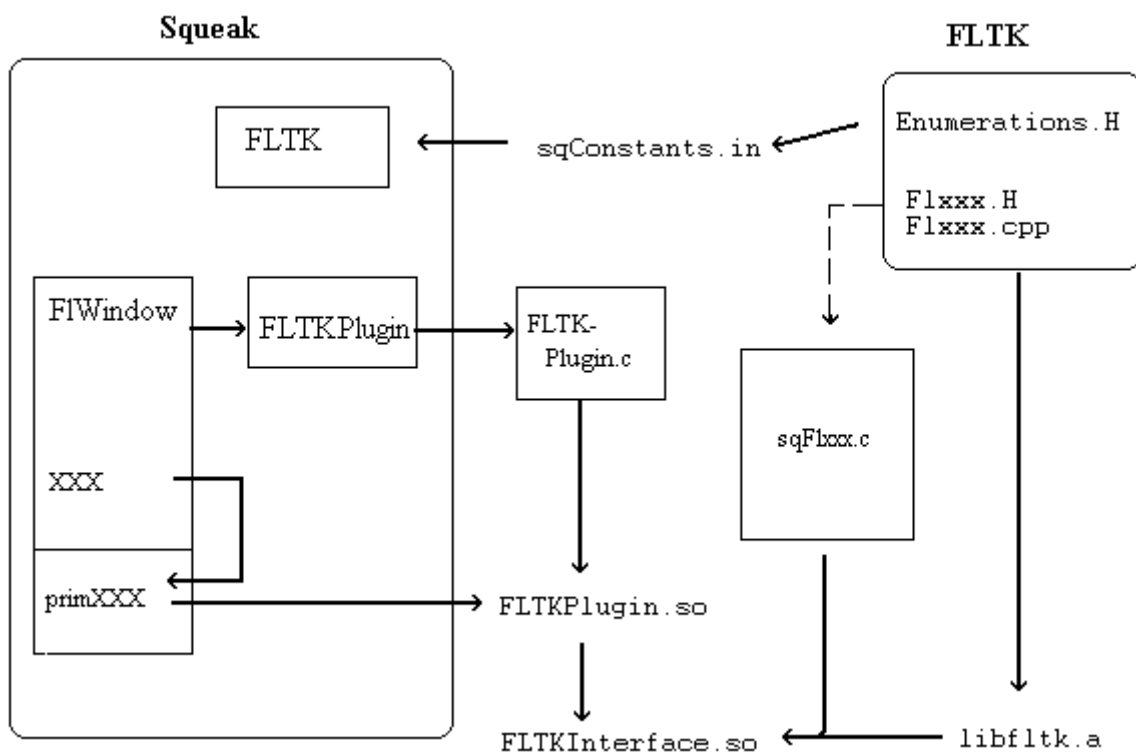
5.4 Sestavení modulů

Pro snadný překlad zdrojových souborů se používá program GNU `make`, který automatizuje zpracovávání jednotlivých souborů. K tomu využívá soubor s popisem překladu, který má tradičně název `Makefile`. Ten specifikuje, který překladač se má použít, jaké má mít parametry a jaké jsou závislosti mezi vstupními soubory. Při změně jednoho zdrojového souboru také program `make` automaticky zajistí aktualizaci všech na něm závislých modulů a programů.

Výsledným produktem překladu tohoto projektu jsou tři soubory, které se umístí do adresáře `bin`. Při spuštění `make` v hlavním adresáři se nejprve provede překlad podadresáře `interface`. V něm se nachází pro něj specifický `Makefile`. Pro každý vstupní zdrojový soubor s automaticky generovaným hlavičkovým souborem se provede skript `makeh.sh`. Je vyžadována i přítomnost knihovny `libfltk.a`, která se získá překladem FLTK nebo je již nainstalována v systému. Produktem překladu adresáře je sdílená knihovna `FLTKInterface.so`. Ta je v C++, ale její funkce jsou lze volat z C (viz kapitola 5.1).

Dále se pokračuje v podadresáři `plugin`. Překládá se soubor `FLTKPlugin.c`, tedy vygenerovaný soubor, získaný ze Squeaku provedením `FLTKPlugin translate`. `FLTKPlugin.c` je přímo propojen s virtuálním strojem a proto potřebuje jeho hlavičkové soubory. Ty musejí být do tohoto adresáře ručně přepokopány ze zdrojových textů virtuálního stroje příslušné platformy. Součástí projektu jsou hlavičkové soubory pro systémy Unix. Produktem překladu je sdílená knihovna `FLTKPlugin.so`. Její funkce jsou přímo volány při provádění primitivních metod.

V podadresáři `constants` se pomocí skriptu `makeconst.sh` získá soubor konstant pro Squeak a umístí také ke sdíleným knihovnám do adresáře `bin`. Na obrázku 9 je vidět konečné závislosti všech částí rozhraní.



Obr. 9: Celková struktura rozhraní

6 Problémy

Zatím ne zcela dořešené jsou problémy s inicializací a ukončováním rozhraní. Obrazy objektů ve Smalltalku jsou uloženy do tzv. *image*, z kterého se při spuštění systému znovu načtou a systém je v naprosto stejném stavu. To nelze provést při využívání externího kódu. Pokud má například objekt ve Squeaku uložen ukazatel na odpovídající objekt v C++ a tento stav se uloží, nemůže se při obnovení tento ukazatel již používat. Po restartu systému se objekt v C++ nenachází na stejné adrese a není v totožném stavu. To je třeba ošetřit v metodách `startup` a `shutdown`, které umožňují provést uživatelské akce při spouštění a ukončování Squeaku. Vhodné řešení by bylo projít všechny instance FLTK a zneplatnit ukazatele, které obsahují. Musí se ale vyřešit problém závislostí mezi objekty, neboť objekty Squeaku, které rozhraní využívají, předpokládají, že vše je ve stále stejném stavu.

Obslužné funkce (callback funkce) by bylo vhodné doplnit o nějakou formu synchronizace mezi vyvoláním a obsluhou funkce. Pokud se v externím kódu vyvolají dvě žádosti o obslužnou funkci ihned po sobě, Squeaku trvá delší dobu, než zareaguje a první z nich se nezachytí. Situaci ale nelze řešit zadržením v externím kódu, protože by se celý systém zastavil (viz 4.3.3). Dalo by se snad využít fronty požadavků, které by byly postupně obsluhovány.

Problémem je také uvolňování objektů. Při ztrátě poslední reference na instanci se po čase provede její zrušení. Pokud obsahuje ukazatel na externě alokovaný objekt, je vhodné ho v tom okamžiku také uvolnit. Squeak umožňuje použít finalizaci (provedení metody objektu před jeho zrušením) speciálně registrovaných objektů pomocí třídy `WeakArray`. Toto řešení je ale nedostačující. Při konstrukci uživatelského rozhraní ve FLTK se velmi často využívá třídy `Fl_Group`, která sdružuje vkládané objekty do jedné skupiny. Při použití ve Squeaku však po vložení objektu do `Fl_Group` už na něj nemáme referenci. To by vedlo k nesprávnému uvolnění objektu, se kterým se stále pracuje. Možným řešením by bylo ukládání informace, jakým způsobem se má objekt rušit, v závislosti na tom, do které skupiny patří.

7 Další vývoj projektu

Cíle projektu se podařilo splnit. Do budoucna by bylo vhodné především vyřešit problémy popsané v předchozí kapitole. Je třeba také postupně převádět více tříd a metod z FLTK do Squeaku, projekt zatím obsahuje pouze jejich malou část.

Dalším krokem by mohla být nadstavba nad tímto rozhraním, která by se ve svém pojetí více přibližovala Squeaku a využívala by jeho potenciálu pro co největší část implementace. Rozhraní lze pak využít pro programování aplikací ve Smalltalku, které mají stejný vzhled a ovládání jako ostatní aplikace v systému.

Literatura

- [1] Hopkins, T., Horan, B.: Smalltalk – an introduction to application development using VisualWorks. Prentice Hall 1995, ISBN 0-13-318387-4
- [2] Squeak. <http://www.squeak.org>
- [3] Spitzak, B.: Fast Light Tool Kit. <http://www.fltk.org>
- [4] Squeak 3.0 image. <ftp://st.cs.uiuc.edu/Smalltalk/Squeak/3.0/platform-independent/>
- [5] Project Digitaslis. <http://www.phaidros.com/DIGITALIS>
- [6] Benson, J.: QuickTime for Squeak. <http://home.attbi.com/~zurgle/quicktim.htm>
- [7] Squeak Swiki. <http://minnow.cc.gatech.edu/squeak/1>
- [8] Alsina, R.: QtC 0.0.2. <http://linux.tucows.com/x11html/preview/9909.html>
- [9] Henstridge, J.: PyGTK. <http://www.daa.com.au/~james/pygtk/>