VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EDITOR ZDROJOVÝCH KÓDŮ PRO QDEVKIT

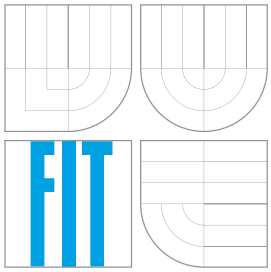
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

BŘETISLAV KÁBELE

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EDITOR ZDROJOVÝCH KÓDŮ PRO QDEVKIT

QDEVKIT SOURCE CODE EDITOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — BŘETISLAV KÁBELE
AUTHOR

VEDOUCÍ PRÁCE — Ing. VAŠÍČEK ZDENĚK
SUPERVISOR

BRNO 2010

Abstrakt

Práce se zabývá návrhem a implementací editoru zdrojových textů a souborového manageru do aplikace QDevKit. V teoretické části jsou předvedeny trendy moderních editorů a vlastnosti použitých technologií. Dále popisuje návrh, implementaci a způsob začlenění vzniklého řešení.

Abstract

This work deals with the design and implementation of the QDevKit source code editor and file manager. The theoretical part of this thesis describes trends of modern source code editors and features of used technologies. It describes the concept, the implementation and the way of integration the incurred solution.

Klíčová slova

QDevKit, editor, Qt, QScintilla, FITkit

Keywords

QDevKit, editor, Qt, QScintilla, FITkit

Citace

Břetislav Kábele: Editor zdrojových kódů pro QDevKit, bakalářská práce, Brno, FIT VUT v Brně, 2010

Editor zdrojových kódů pro QDevKit

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zdeňka Vašíčka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Břetislav Kábele
17. května 2010

Poděkování

Tímto bych chtěl poděkovat Ing. Zdeňkovi Vašíčkovi za odbornou pomoc při tvorbě praktické i teoretické části mé práce a lidem kteří se podíleli na testování programu.

Obsah

Kapitola 1

Úvod

QDevKit je skriptovatelný terminál určený pro komunikaci s FITkitem. Usnadňuje přeložení aplikací a práci s FITkitem. Jeho hlavní nevýhodou je, že při vytváření aplikací se musí přepínat mezi ním a externím editorem. Vzniká tak nepříjemný stav, kdy musíme mít spuštěné dva programy, které spolu nekomunikují. Není možné zautomatizovat rutiny jako například překlad aplikace po uložení nebo přenahrání editovaného zásuvného modulu.

Pro vývoj jakýchkoliv aplikací je zapotřebí vývojového prostředí. Jednou ze součástí vývojového prostředí je textový editor. V dřívějších dobách stačil editor pro psaní jednoduchého textu. Nebylo třeba vyšších funkcí kvůli jednoduchosti syntaxe programovacího jazyka. S rozšířením počítačů do domácností se nároky na aplikace zvýšily a současně se zvýšily i nároky na programovací jazyky. S komplexností programovacích jazyků vzrostly požadavky na vývojová prostředí, potažmo na editory. V dnešní době je téměř samozřejmostí, že kvalitní editor pro vývoj aplikací bude mít zvýrazňování kódu, doplňování slov, zakomentování (odkomentování) řádku či číslování řádků. Tyto vlastnosti pak dovolí vývojářům věnovat se více samostatnému vývoji a pomáhají jim urychlit práci.

Práce je členěna následovně. První část se zabývá teoretickým úvodem do moderních editorů a jejich vlastností. Druhá část informuje o požadovaných technologiích. Třetí se zabývá návrhem řešení a čtvrtá implementací.

Kapitola 2

Moderní textové editory

V dnešní době, kdy požadavky na vývoj aplikací jsou nemalé, je třeba se zamyslet, jak aplikace vyvíjet rychle a levně. Právě jejich cena je jedním z důležitých aspektů úspěšnosti. Pro rychlý a levný vývoj aplikací přispívají vlastnosti textových editorů. Je žádoucí, aby se vývojář dobře orientoval v textu, který napíše, nebo který napsal někdo jiný. Dotyčný člověk se pak soustředí pouze na to, co ho zajímá, a ostatní text už nevnímá.

Pro podporu tohoto přístupu bylo zavedeno několik metod k usnadnění čtení a psaní textu. Především zvýrazňování syntaxe vede ke zvýšené přehlednosti textu. Doplňování kódu urychlí psaní zdrojového textu stejně tak jako kontrola kódu při psaní a kontextové nápovědy a tipy. Ke zpřehlednění textu také přispívá automatické odsazování, číslování řádků, zvýrazňování řádků a skrývání částí kódu. Pomocí zakomentování a odkomentování textu můžeme docílit pohodlného testování. Vyhledáváním se můžeme snadno přesunout na tu část kódu, která nás zajímá. Historie úprav nám uchovává naše změny v textu a poznámky mohou doplnit informace pro lepší pochopení kódu. Při psaní kapitoly jsem čerpal z [13] a [10]. Postupně se jednotlivými metodami zabývají následující odstavce.

2.1 Zvýrazňování syntaxe

Při tvorbě zdrojového kódu je možné přispět k čitelnosti textu zvýrazněním důležitých (tzv. klíčových) slov a dalších řetězců, jako jsou například konstanty, komentáře či jiné aspekty jazyka. Pro programátora je pak snadné pochopit části kódu.

Zásadní roli při použití této metody hraje lexikální analýza. Ve své podstatě rozděluje vstupní text na tzv. tokeny. Tokeny si můžeme jednoduše představit jako jednotlivá slova. K tokenu se přidává atribut, který označuje jeho druh – například token typu klíčové slovo, komentář, atd. Rozdělování textu na tokeny je řízeno lexikálním analyzátorem. Programově je lexikální analyzátor řešen jako konečný automat, jehož výstupem je právě token s atributem. Z teoretické informatiky víme, že konečný automat zpracovává pouze regulární jazyky. Další informace například v [7].

Zvýrazňování pomocí regulárních výrazů

Nejjednodušší zvýrazňovače používají pouze regulární výrazy pro zjištění typu tokenu. Tato lexikální analýza je řízena na základě poskytnutých regulárních výrazů. Analyzátoru se zadá regulární výraz a pro něj definovaný způsob zvýrazňování (nejčastěji barva a tučnost písma) a požadovaná slova budou zvýrazněna.

Příklad regulárního výrazu pro zvýraznění všech hexadecimálních čísel začínajících sekvencí 0x, poté alespoň jedno či více písmen nebo číslic se zvýrazní modře tučně. Zápis je zhotoven pouze pro ukázku:

exp = 0x[0-9a-fA-F]$^+$
color = blue
bold = true

Tento přístup má své výhody i nevýhody.

Výhody: Poměrně rychlý způsob zvýrazňování textů, jednoduché zadávání pravidel v podobě regulárních výrazů a snadné přidání nových pravidel. Při změně textu stačí zanalyzovat jen změněné slovo.

Nevýhody: Nedokáže zvýrazňovat složitější zápisy, které mají návaznosti slov. To znamená, že rozpoznává jednotlivá slova, ale neudržuje si informace o kontextu. Nedokáže zvýraznit například řetězce či komentáře a tokeny, ve kterých by se měl vyskytnout znak ohraničení tokenu.

Optimalizace: Pro zvýraznění komentářů a řetězců si stačí pamatovat předchozí stav a podle něj a aktuálního tokenu se rozhodovat. Nicméně složitější konstrukce se tímto nevyřeší.

Příklad je možno vidět na obrázku 2.1. První volání funkce je chybně zvýrazněno, druhý parametr neměl být zvýrazněn. Je možno si toho všimnout v porovnání s druhým voláním funkce nebo na obrázku 2.2.

```
void function(char* p1, int p2, char* p3);

int main()
{
        char* p1 = "Ahoj";
        int p2 = 2010;
        char* p3 = "světe!";
        function("Ahoj", p2, "světe!");
        function(p1, p2, p3);
        return 0;
}
```

Obrázek 2.1: Příklad zvýrazňování pomocí regulárních výrazů v demonstrační aplikaci „Syntax Highlighter“ frameworku Qt.

Zvýrazňování na základě syntaktické analýzy

S využitím syntaktické analýzy získáme silnější nástroj než je lexikální analýza založená na regulárních výrazech. Syntaktický analyzátor využívá tokeny, které dostane od lexikálního analyzátoru. Pro popis syntaxe se používají bezkontextové gramatiky. Programovací jazyky jsou zpravidla koncipované jako bezkontextové. Syntaktické analyzátory se realizují zásobníkovým automatem, který na základě definovaných pravidel provádí syntaktickou analýzu. Výstupem analýzy je derivační strom.

Tento přístup využijeme ke zvýrazňování složitějších struktur, jako jsou komentáře, řetězce a vnořené struktury. Zásobníkový automat nám dovoluje pamatovat si nejen minulý

stav, ale i všechny předchozí stavy, kterými jsme prošli. Pokud se budeme do struktury zanořovat, zaznamenáme vždy stav do zásobníku. Při vynořování postupně stav obnovíme ze zásobníku v předpokládaném pořadí.

Uvedu příklad vymyšleného jazyka pro ilustraci. Po každém použití otevírací složené závorky se změní zvýraznění písma a při použití uzavírací složené závorky se obnoví stav před otevírací závorkou:

typ1 { *typ2* **{ typ3** { `typ4` } **typ3 }** *typ2* } typ1

Výhody: Dokáže rozpoznat a zvýraznit i velmi složité struktury. Pamatuje si předchozí stavy. Syntaktická analýza se může dále použít k dalším pokročilým funkcím editoru.

Nevýhody: Zpomaluje celý proces zvýrazňování. Při změně textu musí celá analýza proběhnout znova. Pro každý jazyk se musí vytvořit samostatný analyzátor.

Optimalizace: Není potřeba vytvářet derivační strom, ke zvýrazňování se nevyužije. Pro analýzu se použije jen viditelná část textu.

Příklad zvýrazňování na základě syntaktické analýzy je na obrázku 2.2. K porovnání poslouží obrázek 2.1.

```
void function(char* p1, int p2, char* p3);

int main()
{
	char* p1 = "Ahoj";
	int p2 = 2010;
	char* p3 = "světe!";
	function("Ahoj", p2, "světe!");
	function(p1, p2, p3);
	return 0;
}
```

Obrázek 2.2: Příklad zvýrazňování na základě syntaktické analýzy editoru Scite.

2.2 Doplňování kódu

Dalším usnadněním pro rychlý vývoj aplikací je automatické doplňování kódu. Základní myšlenkou je uhodnout, co uživatel píše, a doplnit zbytek slova. Výhodou plynoucí z tohoto přístupu je to, že si uživatel nemusí pamatovat celý používaný slovník. Chyby způsobené psaním slov se minimalizují. Doba psaní výsledného kódu se značně zkrátí. Navíc při používání doplňování se uživatel nemusí bát používat delší samopopisující názvy.

Předpokladem doplňování je, aby slovník používaného jazyka byl známý a měl konečný počet slov. Jednotlivá slova ve slovníku by se měla od sebe lišit v prvních pár písmenech, aby bylo možné co nejrychleji zmenšit prohledávaný prostor slovníku. Důležité je, zda tato metoda dokáže hledané slovo či text umístit na první místo v seznamu slov. Doplňování samotné je založeno na prohledávání stavového prostoru slovníku.
V doplňování se uplatňují dva druhy přístupu:

Nerozlišování velikosti písma snižuje zátěž na uživatele, který si nemusí velikost písma pamatovat. Na druhou stranu zvětšuje nabízený počet slov. Vyhledávané slovo se velmi často nevyskytuje na předních pozicích kolekce nabízených slov.

Rozlišování velikosti písma zmenšuje počet nabízených slov. Pravděpodobnost výskytu požadovaného slova na předních místech je větší. Pokud ale uživatel nenapíše přesně prvních pár písmen, slovo se v nabízené kolekci nemusí vůbec objevit.

Přehled algoritmů a jejich hlavních myšlenek:

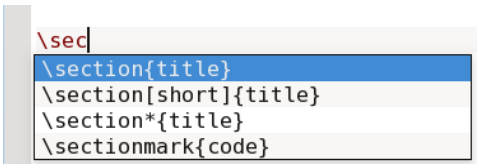
- Optimistické doplňování – u každého slova ve slovníku se ukládá počet použití a podle toho se řadí slova v nabízené kolekci. Pokud totiž uživatel slovo již jednou použil, je dosti pravděpodobné, že ho použije znova.
- Optimistické strukturní doplňování – vychází z domněnky, že lokální metody a funkce se volají častěji než vzdálené, proto se jejich názvy posunou do předních pozic.
- Poslední vložené slovo do slovníku – pokud uživatel přidal nějaké slovo do slovníku, předpokládá se, že ho bude chtít v nejbližší době použít. V kolekci se nově přidané slovo objeví na první pozici. Ostatní slova se řadí podle pořadí přidání.
- Nedávno vložený text – kolekce nabízených slov se upravuje podle posledního vloženého textu. Slova, která uživatel napsal naposledy, se budou posouvat na přední místa.

Samozřejmě existuje spousta dalších algoritmů, které jsou použitelné. Záleží pouze na situaci, kdy a který z nich je výhodnější. Výše jmenované algoritmy se dají mezi sebou kombinovat. Také záleží na tom, jestli u nich implementujeme rozpoznávání velkých písmen či ne.

Doplňování se běžně implementuje tím způsobem, že po napsání určitého počtu písmen, například tří, se prohledá slovník a shody se slovy se zobrazí. Poněkud složitější je postup při nabízení názvů metod tříd. V tomto případě musí být již známý typ instance třídy, aby se vědělo, jaký slovník se má použít. Tyto informace se většinou zjišťují v průběhu lexikální a syntaktické analýzy. Obě analýzy se provádějí při jiných vyšších funkcích moderních editorů. Proto se doplňování velmi často vyskytuje v kombinaci s jinými metodami.

V editorech se běžně používají dva druhy slovníků. Jeden skládáme z již napsaných slov. To znamená, že na začátku psaní textu nebudeme mít k dispozici žádná slova k doplnění. Avšak na konci psaní budeme mít dost velký slovník na doplnění čehokoliv. Druhý slovník máme již složený na základě použitého programovacího jazyka, který dříve specifikujeme. Tento přístup se používá více v objektově orientovaných jazycích, kdy nám editor nabízí názvy metod známých tříd. Slovník se také může měnit. Většinou se skládá z menších podslovníků jednotlivých tříd. Právě tyto podslovníky můžeme přidávat.

Na obrázku 2.3 je příklad doplňování pro systém LaTeX.



Obrázek 2.3: Příklad zobrazení slov pro doplnění v programu Kile.

Sémantické doplňování

Sémantické doplňování je poměrně nový trend převážně internetových vyhledávačů a portálů. Jeho myšlenkou je poskytnout nápovědu nejen prostřednictvím slov, která mají stejná

počáteční písmena, ale i slov, která mají podobný význam nebo jsou se slovem spojena významem či kontextem.
Příklad pro doplnění slova prezide...:

prezident
Václav Havel
hlava státu

Typy sémantického doplňování:

- Ekvivalence relací – podporuje porovnávání slov na základě synonym, homonym a víceznačnosti slov. Například anglická zkratka „NYC“ se může doplnit jako „New York City“ nebo „Big Apple“, jelikož všechna slova odkazují na stejné město. Všechna sémanticky podobná slova se doplní do preferované formy daného editoru.
- Nepřímé sémantické doplňování – dokáže doplnit slova za hranice porovnávání a ekvivalencí. Například spojení „Evropská unie“ dokáže doplnit na jména členských států.
- Sémantické doplňování rolí – toto doplňování dokáže doplnit slovo na základě role nějakého objektu. Pro pochopení postačí příklad: Máme editor, který zná názvy muzeí a divadel. My napíšeme město a on nám nabídne seznam muzeí a divadel v tomto městě.

Sémantické doplňování je mocným nástrojem doplňování převážně v internetových prohlížečích. Pokud píšeme text na nějaké stránce a známe již kontext dané stránky, pak vyhledávací slovník není tak velký. Informace byli získány v [2, 6].

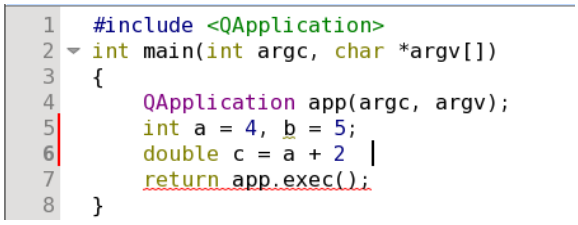
2.3 Kontrola kódu při psaní

Kontrola kódu je dobrým pomocníkem při psaní zdrojových kódů v dříve specifikovaném jazyce. Editory zvýrazňují části kódu, které jsou syntakticky či sémanticky nečisté z hlediska použitého jazyka. Syntax a sémantiku jazyka musíme již znát, aby bylo možné předem vytvořit množinu syntaktických a sémantických pravidel. Pro kontrolu se používají dva základní přístupy – syntaktická a sémantická analýza.

Syntaktická kontrola kódu

Syntaktická kontrola kódu je nejpoužívanějším způsobem kontroly u mnohých editorů. Vychází ze syntaktické analýzy kódu, při které dochází k vytváření derivačního stromu. Pokud není možné ho vytvořit, je někde v kódu chyba. V praxi se dost často nevytváří derivační strom, ale jen se provádí kontrola syntaktických pravidel. Ta zjistí, na jakém řádku je chyba, a editor posléze zvýrazní celý řádek.

Na obrázku 2.4 je možno vidět červené podtržení jakožto zvýraznění syntaktické chyby, chybějící středník na předchozím řádku. Příklad je z jazyka C++. Je možno si všimnout zeleného podtržení proměnné „b“. Během syntaktické kontroly si lze poznamenat u každé proměnné, jestli již byla použita. Pokud ne, pak ji editor zvýrazní.



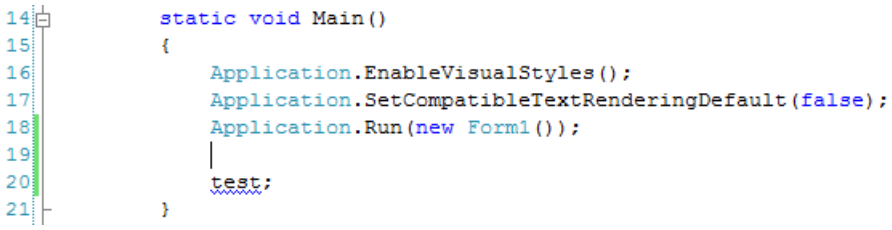
Obrázek 2.4: Příklad kódu z aplikace QtCreator se zvýrazněním syntaktické chyby.

Sémantická kontrola kódu

Výstupem sémantické analýzy je tzv. abstraktní syntaktický strom. Nepodaří-li se tento strom sestavit, pravděpodobně došlo k sémantické chybě. V moderních editorech se ovšem strom nevytváří. Kontrolují se převážně typové konverze nebo deklarace proměnných.

Celá sémantická analýza by byla pro editor příliš náročná, a proto se příliš v moderních editorech nevyskytuje. Pokud editor poskytuje sémantickou kontrolu kódu, bývá začleněn do většího komplexu vývojového prostředí. Zde sémantickou analýzu provádí překladač a jím ohlášené chyby editor pouze zvýrazňuje.

Na obrázku 2.5 je zvýrazněna sémantická chyba v jazyce C# ve vývojovém prostředí MS Visual Studio 2008. Proměnná „test“ nebyla deklarována. Před zvýrazněním této chyby bylo nutné provést překlad programu, protože zde sémantickou kontrolu provádí překladač. Editor chybu zachytil a zvýraznil.



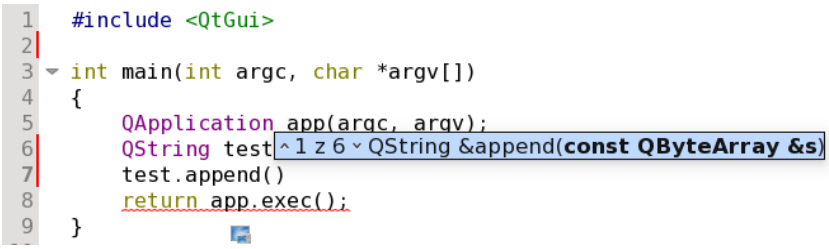
Obrázek 2.5: Příklad kódu z aplikace Microsoft Visual Studio 2008 se sémantickou chybou.

2.4 Kontextové nápovědy a tipy

Nápovědy a tipy nejsou jen výsadou editorů. Jsou nedílnou součástí každé aplikace a umožňují snazší orientaci v programu. Jsou svázané s nějakým slovem nebo objektem. Informují nás o tom, co daný objekt dělá nebo co dané slovo znamená. V případě editorů zdrojových kódů jsou nejčastěji použity k informacím o parametrech funkcí a parametrech metod a jejich typů. V některých případech též informují o tom, co daná funkce nebo metoda dělá.

V prostředí editorů jsou tyto metody úzce spojené s doplňováním kódu, jelikož mohou přesněji zobrazovat informace o tom, co hledáme. Také se nápověda nemůže zjevit z ničeho nic, musí být zobrazena v nějakém kontextu, aby uživatel mohl využít její pomoci.

Příklad nápovědy je možno vidět na obrázku 2.6, kde je zachyceno napovídání parametrů pro metodu třídy.



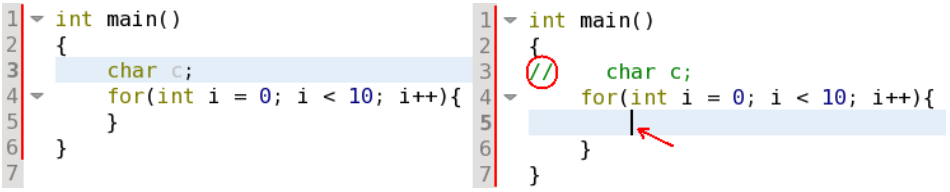
Obrázek 2.6: Příklad nápovědy v programu QtCreator.

2.5 Odsazování, zakomentování, odkomentování textu

I v běžně psaném projevu je známo, že odsazování jistých částí textu zvyšuje přehlednost. V programování to není výjimkou, ba dokonce je silně naléháno na to, aby každá logická část kódu byla odsazena stejně. Je také dobré dodržovat stejné zvyšování a snižování odsazení například o dvě mezery. Je nežádoucí, aby část více se zanořující byla odsazena méně. Editory přispívají k automatickému odsazování pomocí jednoduchých metod. Nejčastěji stačí čítač, který říká, o kolik je nutné odsadit. Pro zvětšení či zmenšení čítače se používá pravá či levá složená závorka. Ve speciálních případech se čítač zvyšuje při vynechání středníku. Po napsání středníku se však musí čítač vrátit na původní hodnotu. Problém nastává, když se změní pozice v textu. Pak se musí hodnota pro čítač přepočítat. Někdy bývá čítačů pro odsazení řádků více.

Zakomentování a odkomentování jsou techniky, které se mi osobně velmi osvědčily při testování. Je dobré si nežádoucí kód snadno zakomentovat a sledovat, jak se chová zbytek. Způsob provedení komentáře se liší dle použitého programovacího jazyka. Nejjednodušší je zakomentování prostřednictvím řádkových komentářů. Při označení většího bloku textu stačí na začátek každého řádku vložit řádkový komentář. Pro použití víceřádkových komentářů je situace složitější, jelikož máme počáteční a ukončovací znaky. Dnešní editory jsou schopny používat oba druhy komentářů.

Ukázka zakomentování řádku a správného odsazování je na obrázku 2.7.



Obrázek 2.7: Příklad zakomentování řádku a odsazování v editoru QtCreator.

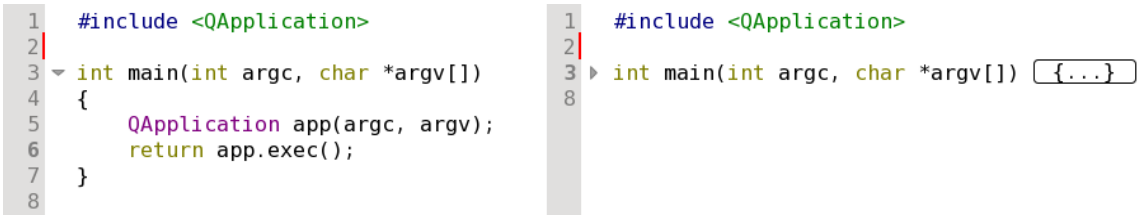
2.6 Číslování a zvýrazňování řádků, skrývání částí kódu

Číslování řádků velice přispívá k základní orientaci v kódu. V případě chyby po překladu překladačem programovacího jazyka obdržíme číslo řádku, kde se chyba vyskytuje. Toto číslo by se zpravidla mělo shodovat s číslem řádku v editoru. Číslo řádku se obvykle zobrazuje nalevo od řádku, nicméně některé editory, ne však editory zdrojových kódů, nabízejí možnost zobrazit čísla napravo. Zobrazení čísel napravo je rozšíření pro země používající arabské písmo.

Zvýraznění řádku usnadní čtení textu s dlouhými řádky. V kombinaci s číslováním je lepší podporou pro hledání chyb. Některé editory zavádějí i druhotné zvýraznění řádku (jiný řádek, než na kterém se nacházíte). Využívá se pro zvýraznění řádku s chybou, jehož číslo editor obdrží od překladače.

Další často vyskytující se technikou usnadňující čtení zdrojového kódu je skrývání jeho částí (angl. folding). Uživatel si pomocí této techniky může skrýt text, který ho nezajímá, aby ho nijak nerušil. Využívá se vlastnost jazyků, a to taková, že bloky kódu se uvozují nějakými párovými značkami. Text mezi těmito značkami se může skrýt a viditelná zůstává jen jedna vstupní značka, aby mohl být kód znovu odkryt.

Příklad skrývání kódu je na obrázku 2.8. Levá část ukazuje nezakrytý kód a pravá část kód po zakrytí.



Obrázek 2.8: Příklad skrývání části kódu v programu QtCreator.

2.7 Historie úprav, poznámky, vyhledávání

Historie úprav je jednou z vlastností, která se stala nezbytnou součástí moderních textových editorů. Dovoluje vrátit se v editoru k textu, který jsme napsali dříve či později. K přechodu mezi texty se používají operace „zpět“ nebo „vpřed“. Konkrétní implementace se liší ve způsobu uchovávání historie. Nejsnazší přístup je při každé úpravě textu si uložit celý text. Tento přístup je velice náročný na prostředky počítače. Ale v případě přechodu zpět nebo vpřed nemusíme dělat složité operace, jednoduše přenačteme celý obsah textu. Jiný způsob ukládání – u každého vloženého písmene uchováme několik atributů (například pořadí vložení, místo v textu), poté při operacích s historií výsledný text poskládáme. I tato metoda má svá úskalí, a proto moderní editory používají složitější metody.

Poznámky v textu jsou spíše výsadou textových procesorů, ale v editorech také najdou své uplatnění, a to v případě, kdy programátor nechce, aby jeho poznámky ke kódu viděl někdo jiný než on sám. Poznámky se udržují většinou v oddělených souborech. Udržuje se v nich informace o tom, k jakému textu patří a na jaké místo v kódu se mají zařadit.

Vyhledávání slov je dnes v editorech samozřejmostí. Moderní editory nabízejí i možnost přepsání vyhledaného slova, ukončení hledání na konci dokumentu či hledání citlivé na malá a velká písmena.

Kapitola 3

Použité technologie

3.1 Qt

Qt je „framework“ od společnosti Trolltech, nyní vlastněné společností Nokia. Jedná se o multiplatformní knihovnu pro vývoj aplikací s grafickým uživatelským rozhraním. První verze byla vydaná v březnu roku 1995. V dnešních dnech je plnohodnotným nástrojem též pro vývoj konzolových aplikací (od Qt v4). Nabízí možnost pracovat se sítěmi, SQL, XML, OpenGL, podporuje internetový vývoj a multimédia.

Knihovny jsou dostupné i pro jiné programovací jazyky, než je C++. Pro krátký přehled stačí zmínit: Python, Ruby, Perl, Pascal, C# nebo Java. Pro vývoj aplikací využívajících Qt existuje několik vývojových prostředí:

- Qt Creator – editor zdrojových textů vyvíjen přímo společností Nokia.
- Visual Studio – vývojové prostředí od společnosti Microsoft, Qt knihovny se přivedou do aplikace jako zásuvný modul.
- Eclipse – vývojové prostředí původně pro jazyk Java, Qt port pro Javu se nazývá Jambi, Qt knihovny se přivedou do aplikace jako zásuvný modul.

Framework je dostupný nejen na platformě MS Windows, ale také na X Window System, Mac OS X a vestavěných zařízeních. Tato vlastnost je důležitým znakem Qt. Další vlastností je vydávání pod licencí GPL, LGPL a komerční verzí.

Pro překlad aplikací s knihovnou Qt se používá následující „trojice“:

- qmake -project – vytvoří soubor s příponou .pro s informacemi o projektu.
- qmake – bez parametrů hledá soubor s příponou .pro a na jeho základě vytvoří soubor Makefile.
- make – přeloží projekt.

Celý projekt je dobře zdokumentován na internetu [11]. Užitečné informace lze nalézt i v publikaci [5], ze které jsem čerpal při psaní práce.

3.2 QScintilla

Qscintlla[1] je založená na editoru Scintilla napsaná ve frameworku Qt. Jedná se o komponentu realizující plnohodnotný textový editor poskytující řadu funkcí. Knihovna podporuje

[1] http://www.riverbankcomputing.co.uk/software/qscintilla/intro

většinu funkcí moderních editorů – zvýrazňování syntaxe, doplňování kódu, odsazování, zvýrazňování řádku, číslování řádků, historii úprav, kontextové nápovědy/tipy a vyhledávání. Vyvíjena je firmou Riverbank pod licencí GNU GPL (v2 a v3) a komerční licencí.
Dostupná je ve dvou verzích:

QScintilla1 je postavená na Qt v3, je zastaralá a nepodporuje tolik funkcí.

QScintilla2 je kompatibilní s Qt v3 a v4, je novější verzí, nekompatibilní s QScintilla1. Podporuje více moderních funkcí.

Pro implementaci QScintilli se používají následující třídy:

- QsciScitilla – základní třída implementující okno zobrazující samotný editor. Dokáže doplňovat kód prostřednictvím slovníku, který získává z napsaného textu. Zobrazí čísla řádků a zvýraznění řádku. Obsahuje odsazování, historii úprav a vyhledávání. Dědí ze třídy QsciScintillaBase, která obsahuje nižší funkce.
- QsciLexer – třída pro zvýrazňování syntaxe, provádí lexikální a částečnou syntaktickou analýzu při psaní textu. Dokáže také definovat skrývání kódu pro konkrétní jazyk. Je to abstraktní třída, takže z ní dědí třídy pro jednotlivé podporované jazyky.
- QsciAPIs – třída, která uchová informace a připraví je pro použití v doplňování a pro kontextovou nápovědu.
- QsciDokument – třída pro reprezentaci celého dokumentu. Podporuje sdílení dokumentu mezi více QsciScintilla instancemi.
- QsciCommand – třída pro registraci příkazů a kláves spojených s příkazy.
- QsciStyle a QsciStyledText – třídy pro definování stylů.
- QsciMacro a QsciPrinter – podpůrné třídy pro tisk a definování maker.

Vytvoření editoru

Pro zavedení editoru nám stačí vytvořit instanci třídy QsciScintilla. Pro vstup textu se používá metoda „void setText(const QString &text)“ a pro výstup metoda „QString text() const “. Je nutné poznamenat, že metody přijímají nebo vracejí instanci třídy QString frameworku Qt i s jeho vnitřní reprezentací. To znamená, že správnost kódování QsciScintilla nezajišťuje.

Dále QsciScintilla poskytuje standardní metody pro práci s textem (kopírovat, vložit, vyjmout, atd.) a metody pro práci s historií úprav („void undo()“ – zpět, „void redo()“ – vpřed). V této třídě se nastavuje způsob doplňování. Máme tři možnosti: doplňování vypnuto, doplňování z textu a doplňování ze třídy QsciAPIs (pokud je k dispozici). Samozřejmě poskytuje i jiné další možnosti.
Příklad vytvoření instance editoru v jazyce C++:

```
#inculde <Qsci/qsciscintilla.h>

  QsciScintilla editor(this);   //vytvoření instance
        //nastavení zdroje doplňování na dokument
  editor.setAutoCompletionSource(Qsciscintilla::AcsDocument);
        //nastavení počtu písmen, pro aktivaci doplňování
  editor->setAutoCompletionThreshold(2);
  editor.setAutoIndent(true);   //nastavení odsazování
```

Zvýrazňování kódu

Pro tuto vlastnost se využívá abstraktní třídy QsciLexer. Pro konkrétní jazyk se zavádí třídy QsciLexerXXX, kde XXX značí název jazyka. Například QsciLexerCPP je třída pro zvýrazňování jazyka C++. Pro zvýrazňování se využívá lexikální analýzy a částečné syntaktické analýzy (viz. 2.1). Toto řešení odstraňuje nedostatky samotné lexikální analýzy, takže řetězce a komentáře jsou zvýrazňovány správně. Jelikož syntaktická analýza je jen částečná (nevytváří se derivační strom), není editor tak náročný na své zdroje, jako v případě kompletní syntaktické analýzy.

Pro připojení lexeru k editoru musíme vytvořit instanci lexeru s požadovaným jazykem. Lexeru je nutné v konstruktoru nebo pomocí metody „void setEditor(QsciScintilla *editor)“ předat ukazatel na instanci editoru . Lexer sám již zná klíčová slova pro daný jazyk a začátky a konce bloků. Má předdefinované styly pro zvýrazňování a dokáže komunikovat s editorem.

QScintilla umožňuje dodefinovat si své vlastní styly a navázat je na lexer. QsciLexer má vlastnost ukládat své nastavení a později si ho načíst. Pro jeho vyšší funkce je nutné mu poskytnout třídu QsciAPIs s definovanými slovy pro doplňování a kontextové tipy.

Definování vlastního lexeru

Existují dvě možnosti definice vlastního lexeru. První možnost je využití dědičnosti třídy QsciLexer a k tomu doimplementovat minimálně čistě virtuální metody. Pak podle dokumentace bychom měli celý projekt QScintilla přeložit. Druhá možnost spočívá ve zdědění ze třídy QsciLexerCustom. V tomto případě nemusíme překládat celý projekt, ale musíme navíc doimplementovat metodu „void styleText(int start, int end)“.

Uvedu příklad vytvoření jednoduchého lexeru pro restrukturovaný text, který se používá při psaní dokumentací k projektům pro FITkit. Patří do rodiny značkovacích jazyků. Jeho zákonitosti je možno nalézt na [12]. Ukázka vychází z použití QsciLexerCustom.

Aby příklad nebyl moc složitý, omezím se na zvýraznění direktivy, jednoho „inline“ značení a skrývání bloku.
Direktiva:

> „..“ typ direktivy „::“ direktiva
> blok

Inline značení:

> ‘text‘

Blok:

> ::
> <odsazení> text

Vytvoří se výčtový typ, aby bylo možné snadno identifikovat jednotlivá zvýraznění.

```
enum {
 DIRECTIVE,
 INLINE,
 BLOCK,
 DEFAULT
};
```

První metodu, kterou je nutné naimplementovat je „const char *language() const“. Tato metoda vrací pouze název jazyka.

```
const char *QsciLexerreST::language() const
{
  return "reST";
}
```

Druhou metodu, kterou je nutné doimplementovat, je „QString description(int style) const“. Metoda vrací název pro daný styl „style“. Příklad předefinování:

```
QString QsciLexerreST::description(int style) const
{
  switch (style)
  {
  case DIRECTIVE:
      return "Direktiva";
  case INLINE:
      return "Inline␣značení";
  case BLOCK:
      return "Blok";
  case DEFAULT:
      return "Default";
  }
}
```

Poslední metodou je „void styleText(int start, int end)“. V parametrech se předává začátek a konec textu, který je třeba zvýraznit. Nejprve je nutné se ujistit, že proměnná editor je nastavená. Ke zjištění ukazatele na editor se používá metoda „QsciScintilla* editor()“.

```
void QsciLexerreST::styleText(int start, int end)
{
  if (!editor()) return;
```

Dále je potřeba se editoru dotázat, jaký text se má zvýraznit. Použije se metoda „long QsciScintillaBase::SendScintilla(unsigned int msg, long cpMin, long cpMax, char* lpstrText) const“.

```
  char *text = (char *)malloc(end - start);
  editor()->SendScintilla(QsciScintilla::SCI_GETTEXTRANGE,
                          start, end, text);
  QString source(text);
  free(text);
```

Pro začátek zvýrazňování se musí zavolat metoda „void startStyling(int pos, int style_bits = 0)“, která přepne editor do módu zvýrazňování a nastaví pozici, odkud se začne zvýrazňovat. Část textu se zvýrazňuje metodou „void setStyling(int length, int style)“, té se předá délka textu a styl. Tato metoda také posouvá pozici zvýrazňování nastavenou v předchozí metodě.

```
      //zjištění čísla řádku v editoru
  long index = editor().sendScintilla(
               QsciScintilla::SCI_LINEFROMPOSITION, start);
  startStyling(start);
  QString line;
  foreach(line, source.split("\n")) { //rozdělení po řádcích
    int style = DEFAULT;
    long length = line.length();
    if(length == 1)     //odřádkování
      setStyling(length, DEFAULT); //zvýraznění textu
    else {
```

Nyní se nastaví zvýraznění direktivy. Direktiva se pozná podle dvou teček na začátku řádky.

```
      if(line.startsWith("..")) {
        setStyling(line.find(":"), DIRECTIVE);
        setStyling(length - line.find(":"), DEFAULT);
        style = DIRECTIVE;
```

Pro zvýraznění symbolu bloku se hledají na začátku řádky dvě dvojtečky.

```
      } else if(line.startsWith("::")) {
        setStyling(length, BLOCK);
        style = BLOCK;
```

Následující kód zvýrazní klasický text a inline značení. Pro jednoduchost se předpokládá pouze jeden jeho výskyt. Ukázka poukazuje především na to, že každá část textu musí mít přiřazeno zvýraznění.

```
      } else {
        if((long pos1 = line.indexOf("`")) != -1) {
          long pos2 = line.indexOf("`", pos1);
          setStyling(pos1, DEFAULT);
          setStyling(pos2 - pos1, INLINE);
          setStyling(length - pos2, DEFAULT);
        } else
          setStyling(length, DEFAULT);
      }
    }
```

Dále se sdělí editoru, kde může skrývat kód. QsciScintilla si ke každému řádku značí, do jaké úrovně bloku patří. Právě tato úroveň se musí u každého řádku nastavit.

```
    long level = QsciScintilla::SC_FOLDLEVELBASE; //základní úroveň
    if(style == BLOCK)
      level |= QsciScintilla::SC_FOLDLEVELHEADERFLAG; //vstup do bloku
    else if(style == DEFAULT) {
      if((line[0] == "␣" || line.isEmpty()) && index != 0) {
        level = editor().sendScintilla(QsciScintilla::SCI_GETFOLDLEVEL
                                       index - 1);
        if(level & QsciScintilla::SC_FOLDLEVELHEADERFLAG)
                //řádek pod vstupem do bloku
          level = QsciScintilla::SC_FOLDLEVELBASE + 1;
        else
          level &= QsciScintilla::SC_FOLDLEVELNUMBERMASK; //blok
      }
    }
    editor().sendScintilla(QsciScintilla::SCI_SETFOLDLEVEL,
                           index, level);
    index++;
  }
}
```

Příklad je velmi zjednodušený. Neprovádí žádnou složitou analýzu. Ukazuje převážně komunikaci s editorem při zadávání zvýraznění. Pro definici barev a stylů písma by se musely předefinovat metody „QColor defaultColor(int style)“, „QFont defaultFont(int style)“ a „QColor defaultPaper(int style)“.

Doplňování kódu

Kód je možno doplňovat slovníkem generovaným z textu (rozhraní třídy QsciScintilla) nebo vytvořením vlastního slovníku za pomoci třídy QsciAPIs. Aby doplňování fungovalo musí se nastavit práh, po kterém se začne slovník prohledávat. K tomuto účelu slouží metoda „void setAutoCompletionThreshold(int thresh)“. Je-li cílem zvýšit pravděpodobnost umístění hledaného slova na předních pozicích v kolekci nabízených slov, můžeme využít metody „void setAutoCompletionCaseSensitivity(bool cs)“. Metoda zajistí citlivost na velká písmena. Efektivitu doplňování lze vylepšit použitím metody „void setAutoCompletionReplaceWord(bool replace)“. Ta po doplnění vymaže zbytek slova vpravo od kurzoru.

Definování vlastního slovníku

Vlastní slovník je možné definovat za pomoci třídy QsciAPIs. Metodě „void add(const QString &entry)“ se předá slovo jako parametr. Třída si slovo zaregistruje a metodou „void prepare()“ se všechna zaregistrovaná slova převedou do vnitřní formy. Pokud se posledně jmenovaná metoda neprovede, slovník nebude používán. Třída vytváří vnitřní reprezentaci k tomu, aby byl celý editor schopen reagovat na uživatelské vstupy a nezdržoval se vyhledáváním ve slovníku.

Před vytvořením instance třídy QsciAPIs musí být již k dispozici instance třídy QsciLexer. Editor komunikuje se slovníkem výhradně skrze třídu QsciLexer. Informace předávaná metodě „void add(const QString &entry)“ je ve tvaru:

název(parametry) popis,

přičemž parametry a popis se mohou vynechat. Pokud se vynechají jen parametry, název a popis se berou jako jeden výraz do slovníku. Parametry se oddělují čárkou a musí se uzavřít do závorek.

Slovník lze uložit do souboru. Formát zápisu je stejný. Na každé řádce může být jen jeden výraz. Rozhraní třídy samo dovoluje uložit slovník ve vnitřním formátu pro rychlejší načítání.
Příklad definice slovníku v jazyce C++ i s vytvořením všech náležitostí:

```
#inculde <Qsci/qsciscintilla.h>
#inculde <Qsci/qscilexerTest.h>
#inculde <Qsci/qsciapis.h>

    QsciScintilla* editor = new QsciScintilla;
    QsciLexetTest* lexer = new QsciLexerTest(editor);
    QsciAPIs* api = new QsciAPIS(lexer);
    api->add("funkce(int␣parametr)␣popis␣funkce"); //přidání záznamu
    api->load("název␣souboru");          //přidání záznamů ze souboru
    api->prepare();                      //vytvoření vnitřní formy
    editor->setLexer(lexer);             //nastavení přístupu
      //nastavení počtu písmen, pro aktivaci doplňování
    editor->setAutoCompletionThreshold(2);
      //nastavení zdroje doplňování na QsciAPIs
    editor->setAutoCompletionSource(QsciScintilla::AcsAPIs)
```

Slovník můžeme za běhu aplikace libovolně měnit přidáním nebo ubráním slov. Po každé změně se musí volat metoda „void prepare()“. Příklad metody pro přidání slova do slovníku:

```
void addToDictionary(const QString &word)
```

```
{
  api->add(word);
  api->prepare();
}
```

Podobně odebrání slova ze slovníku:

```
void removeFromDictionary(const QString &word)
{
  api->remove(word);
  api->prepare();
}
```

Kontextová nápověda a tipy

Kontextové nápovědy a tipy jsou výhradně pod správou třídy QsciAPIs (viz. 3.2). Napovídají se hlavně parametry a popisy funkcí. Nápověda se zobrazí u funkce po napsání levé závorky. Složitější nápovědy projekt QScintilla není schopen. Příklad je na obrázku 3.1.

Obrázek 3.1: Příklad kontextové nápovědy v QScintilla.

Kapitola 4

Návrh řešení

4.1 Zadání práce

Cílem práce bylo vytvořit editor zdrojových kódů pro platformu *QDevKit* a následně editor do platformy začlenit. Součástí řešení bylo vytvořit manager souborového systému a propojení všech komponentů dohromady.

Na editor byly kladeny následující požadavky: vhodné začlenění do stávající aplikace, podpora číslování řádků, jednoduché vyhledávání, podpora zvýrazňování důležitých částí kódu pro jazyky C, VHDL, Python a XML. Pro implementaci bylo možné použít stávající řešení, například technologie *QScintilla*.

Základem manageru souborového systému bylo poskytnout možnost prohlížení souborového systému počítače a otevření souborů v editoru zdrojových kódů. Tento požadavek se rozšířil o možnost otevření projektu v jednoduchém stromu, vytváření souborů a pohled na samotnou složku s projektem. Vše vytvořené mělo poskytnout logické rozhraní pro komunikaci mezi sebou a popřípadě s dále rozšiřujícími komponenty.

Je třeba mít na mysli, kdo bude potencionálním uživatelem aplikace. Výsledné řešení by se nemělo lišit od již hotových a v praxi používaných editorů a managerů. Uživatelské rozhraní by mělo být jednoduché a srozumitelné, aby se v něm kdokoliv rychle zorientoval. Potencionálními uživateli budou převážně studenti Vysokého učení technického v Brně.

Požadavky na operační systém byly takové, že aplikace měla být multiplatformní a přeložitelná na většině běžně používaných systémů. Vzhledem k vybranému programovacímu jazyku a „frameworku“ bylo nutné dodržet kompatibilitu se systémem Windows a GNU/-Linux.

Bylo požadavkem používat programovací jazyk C++ [3] a „framwork“ Qt od společnosti Nokia. Dále bylo doporučeno použít řešení od společnosti Rivrbank QScintilla. Celý projekt bylo nutné překládat systémem CMake od společnosti Kitware.

4.2 Současný stav

FITkit

FITkit je přípravek, který Fakulta výpočetní techniky VUT v Brně používá pro výukové účely. Obsahuje mikrokontrolér, hradlové pole FPGA a periferie. Aplikace se mohou tvořit buď v jazyce VHDL nebo v jazyce C. VHDL se používá pro popis „hardware“, jeho využití je převážně pro programování hradlového pole. Jazyk C je využíván pro popis programů pro mikrokontrolér. Stávající verze FITkitu je 2.0. FITkit obsahuje: MCU rodiny

MSP430 (Texas Instruments), FPGA Spartan 3 XC3S50-4PQ208C (Xilinx), USB převodník FT2232C, audio rozhraní, konektory PS2, rozhraní VGA , konektor RS232, DRAM 8x8Mbit, klávesnice, řádkový LCD displej, rozšiřující konektory.

Cílem je, aby se student naučil pracovat s mikrokontroléry a FPGA. Práce pro FITkit jsou dostupné na internetu ve zdrojové podobě. Každý uživatel si tak může aplikaci stáhnout, upravit a opět produkovat ve zdrojové podobě. Více informací je možno získat na [8].

QDevKit

Pro komunikaci s FITkitem se doposud používaly terminály. Práce s nimi byla přinejmenším nepohodlná a zdlouhavá. Jejich hlavní nevýhody byly například vzájemná nekompatibilita, nemožnost pracovat s FITkitem přímo, pouze přes emulovaný sériový port, nutnost nastavit řadu parametrů, neexistence jednotného multiplatformního řešení.

Z těchto důvodů bylo vytvořeno jednotné řešení pro přístup k FITkitu pomocí aplikace *QDevKit*. Tato aplikace je složena z několika nástrojů. Jedním je aplikace *fcmake*, která poskytuje informace o projektu a je vhodná pro použití v manageru. Dalšími jsou *fkflash* a *fkterm*, které slouží pro ovládání FITkitu. *PythonQt* je knihovna pro práci s QDevKitem pomocí jazyka Python. *QDevKit* je samotné grafické uživatelské rozhraní, které spojuje funkčnost výše zmíněných programů a bude využito pro implementaci editoru.

QDevKit umožňuje vytvářet zásuvné moduly, a tak přidat další funkčnost. Je implementováno jednoduché rozhraní, které umožní snadné nahrání či odebrání zásuvného modulu. Přes toto rozhraní je možné přistoupit k důležitým částem QDevKitu.

Dosavadní funkčnost umožňuje přeložit projekt pro FITkit, jeho nahrání do FITkitu a následnou komunikaci s FITkitem. Aplikace poskytuje pohled na již vytvořené projekty, zobrazení „README“ souboru nebo dokumentace. Dokáže spustit externí editor od společnosti Xilinx nebo simulaci v programu ModelSIM (pokud jsou dostupné).V neposlední řadě dovoluje stáhnout projekty pomocí klienta SVN.

Aplikace je psaná z velké části v jazyce C++, druhým použitým jazykem je Python. Pro grafické rozhraní byl zvolen „framework“ Qt. Pro překlad se díky platformní nezávislosti používá program cmake. Na systém Windows existuje též instalátor, který všechny závislosti nese s sebou. Pro více informací odkazuji na [9].

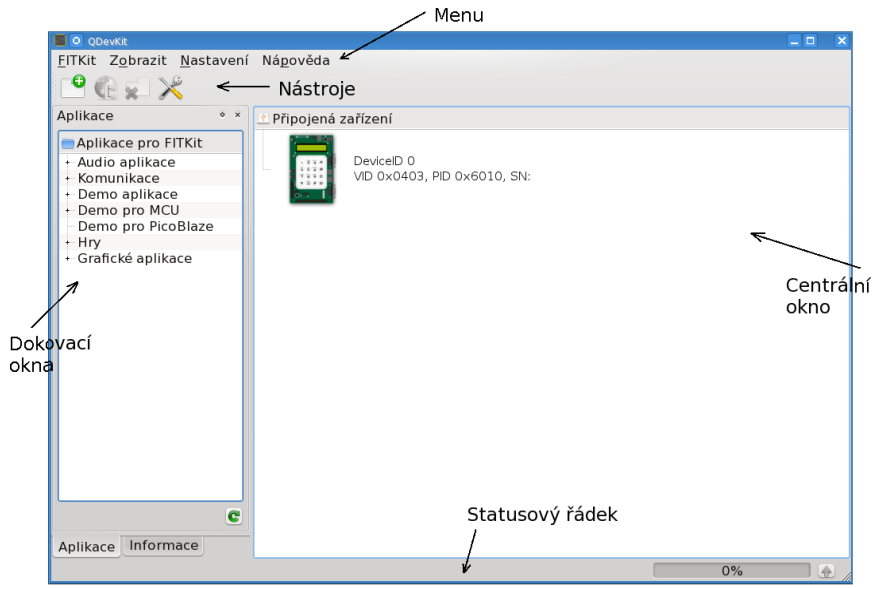
4.3 Návrh editoru

Editorů a vývojových prostředí je dnes na trhu velké množství. Některé z nich jsou velice úspěšné, a proto by bylo dobré si je vzít jako vzor. Zvolil jsem pro inspiraci vývojové prostředí QtCreator, Visual Studio 2008 a Kate. Každý editor má svoje menu, svoji nástrojovou lištu a stavový řádek. V nastavení je možné zapnout manager projektů nebo kolekce otevřených souborů.

V rozložení a nabídce komponent se editory příliš neliší. Ústřední komponentou je okno se zdrojovým kódem. Tento koncept jsem zvolil i já.

Hlavní okno editoru bude vloženo na pozici centrálního okna (obrázek 4.1) místo zobrazování připojených zařízení. Docílí se tím využití největší plochy, kde se bude zobrazovat kód.

Jako přístup k zavedení editoru do QDevKitu se použije již vytvořené menu a to tak, že se do menu přidá další záložka s názvem editor. Záložka po rozbalení bude nabízet operace



Obrázek 4.1: Obrázek aplikace QDevKit

pro otevření souboru jak nového, tak i dříve vytvořeného, a standardní funkce pro ukládání. Instance záložky bude pro všechny editory sdílená.

Každý editor bude mít svoji nástrojovou lištu, aby nedocházelo ke kolizím jeho operací. Zavede se tím i oddělení od stávající nástrojové lišty. Operace v něm budou reprezentovány obrázky. Po najetí kurzorem se zobrazí krátká nápověda ve stavovém řádku. Minimální vložené operace musí poskytovat práci s historií úprav, možnost vyhledávání, výběr jazyka pro zvýrazňování a výběr kódování. Další operace se mohou zvolit podle potřeby.

Stavový řádek bude také pro každý editor vlastní, aby uživatel věděl, ke kterému textu patří. Budou se v něm zobrazovat nápovědy, vyhledávání a na levé straně pozice kurzoru v textu.

Rozpoznávání typů souborů

Editor se na základě názvu otevřeného souboru bude snažit odvodit typ souboru. Rozpoznávání typu souboru není triviální. Využívají se různé techniky, ale nejpodstatnější jsou tři základní:

Přípona souboru: název souboru se skládá z vlastního jména souboru a tzv. přípony. Je to část názvu za poslední tečkou (první zprava). Podle přípony je možné odhadnout typ souboru. Potřebujeme databázi přípon k rozpoznávání.
Výhody: rychlý a jednoduchý způsob rozpoznávání u mnoha přípon a typů souborů, snadná přenositelnost.
Nevýhody: snadná záměna souboru přepsáním koncovky.

Vnitřní metadata: nejrozšířenější způsob na systémech UNIX. Na začátku souboru je „magické číslo“, které udává typ souboru. Databáze typů se pak nachází v /etc/magic. Rozpoznání se provádí porovnáním „magického čísla“ s výskytem v databázi.
Výhody: soubor nemusí obsahovat příponu, záměna souboru je ztížena a je snadná

přenositelnost mezi systémy.
Nevýhody: rozšířeno pouze na UNIXu, je třeba ořezávat začátek souboru.

Vnější metadata: informace o souboru se uchovávají mimo soubor v nějaké databázi (file system). Ta pak obsahuje jméno souboru a jeho typ, popřípadě další možné informace. Speciálním případem je „MIME“ typ, který se používá v elektronické poště pro identifikaci připojených souborů. Obsahuje název typu a podtypu oddělených lomítkem.
Výhody: informace o souboru jsou mimo soubor, velmi těžká záměna souboru.
Nevýhody: nepřenositelnost informací o souboru. Je třeba přenášet se souborem i metadata.

Existují i další metody identifikace souborů. Jako příklad bych uvedl identifikaci na základě metody „Fingerprint“ nebo „Fileprint“. Pro více informací postačují [4] a [1].

Rozhraní

Hlavním požadavkem rozhraní editoru je schopnost komunikovat s co možná největším počtem komponent. Tomuto požadavku bude vyhověno zavedením metod, které zprostředkují komunikaci s okolím. Dále je nutné propojit editor s modulem PythonQt, aby bylo rozhraní přístupné přes jazyk Python.

Nejzákladnější službou, kterou musí rozhraní poskytovat, je jednotný přístup ke všem editorům. Nejlépe v podobě kolekce se všemi vytvořenými instancemi editoru.

Rozhraní by mělo poskytnout možnost otevření souborů, ať už nových nebo vytvořených, a jejich uložení. Dalšími jednoduchými funkcemi, které jsou vhodné přidat do rozhraní, jsou zavření editoru, vrácení názvu souboru asociovaného s editorem, přidání textu na určité místo, přístup ke kontextovému menu, přístup k nástrojové liště a operace s pozicí kurzoru.

Zpřístupnit by se měly i služby pro práci s vyššími funkcemi editoru, jako jsou zapnutí či vypnutí číslování řádků, zapnutí či vypnutí doplňování, změna lexeru a hledání slova. Editor musí také informovat jiné komponenty o otevření souboru, jeho uložení a modifikování.

V zájmu zvýšení rychlosti práce s editorem by se měly některé funkce zpřístupnit přes klávesové zkratky.

4.4 Návrh manageru

Aby byla možná snadná navigace v souborech projektu, je nedílnou součástí i souborový manager. Jeho hlavním úkolem bude poskytovat seznam souborů v projektu. Manager poskytne tři druhy pohledů:

Souborový systém zobrazí všechny soubory a složky na počítači.

Projekty zobrazí soubory spadající pod projekt.

Složka projektu bude zobrazovat soubory a adresáře ve složce projektu.

Všechny pohledy budou sjednoceny v jednom okně a nebudou zobrazeny dva najednou. V tomto případě musí existovat přístup k jednotlivým pohledům.

Pohled „Projekty“ může zobrazovat více projektů najednou. V tomto případě musí umět projekt odstranit z pohledu a na požádání vytvořit soubor ve složce projektu. Pro přidání nového projektu se zvolí snadný přístup, například akce v kontextovém menu stromu aplikací v QDevKitu. Další funkčnost může být přiřazena.

Požadavky na rozhraní manageru nejsou velké. Jeho hlavním úkolem je využívat funkcionalitu ostatních prvků v celém QDevKitu, proto manager nemusí poskytovat rozsáhlé rozhraní.
Popisy metod, které se mají objevit v rozhraní:

- Otevření projektu.
- Informace o tom, že byl projekt otevřen.
- Informace o tom, že byl projekt uzavřen.
- Informace o cestě k souboru/složce, který(á) byl(a) právě označen(a).

4.5 Začlenění řešení do QDevKitu

K začlenění editoru i manageru máme dvě možnosti. Každá má svoje nevýhody i výhody. Je třeba uvážit, pro kterou komponentu jaký přístup použít. Po začlenění se ještě naimplementuje nastavení editoru. Vše zmíněné je rozepsáno v následujících odstavcích.

Přepis kódu

Použijeme již napsané zdrojové kódy a dopíšeme požadovanou funkčnost popřípadě vytvoříme pár nových souborů. Je nutné se seznámit s adresářovou strukturu projektu a vhodně začlenit nové soubory.

Výhody: Je možné celou aplikaci od základů přepracovat a graficky vytvořit lepší řešení. Přístup k veškeré funkčnosti QDevKitu.

Nevýhody: Nebezpečí ztráty funkčnosti při přepisování kódu. Je třeba dát si dobrý pozor, aby to, co funguje, fungovalo dál.

Tento přístup se použije pro přidání editoru. Je žádoucí, aby editor nabídl své rozhraní dalším komponentám. Jeho začlenění do QDevKitu je složitější a musíme mít možnost upravit stávající grafické rozhraní.

Zásuvný modul

QDevKit nabízí možnost psát vlastní zásuvné moduly a začlenit je do připravených míst v QDevKitu. Tato místa se nazývají „doky“. Musíme si prostudovat psaní zásuvných modulů do QDevKitu. To je v podstatě jednodušší, než přepisovat kód.

Výhody: Komunikace je přes definované rozhraní. Nemusíme se bát, že ztratíme funkcionalitu aplikace.

Nevýhody: Nemáme přístup k veškeré funkčnosti QDevKitu. Nemůžeme si grafické rozhraní přepracovat.

Pro psaní manageru nepotřebujeme grafiku měnit, proto bude realizován jako zásuvný modul. Navíc je to komponenta, která může být v průběhu práce vypnuta, aby nepřekážela.

Nastavení

QDevKit má možnost měnit a ukládat nastavení. Vytvoří se vlastní záložka pro nastavení editoru. Nastavení se bude skládat z číslování řádků, doplňování a kódování. Podle vhodnosti se rozhodneme, jakým způsobem se budou nastavení zobrazovat a ukládat. Na záložce vzhled doplní nastavení písma a jeho velikost. Další nastavení mohou být podle uvážení doplněny.

Kapitola 5

Implementace

Již od počátku samotné implementace jsem měl veliký problém cokoliv do QDevKitu připsat. Po experimentování se zdrojovým kódem jsem zjistil, že QDevKit nebyl zrovna navržen tak, aby se mohl rozšířit o editor zdrojových kódů. Řešení zásuvným modulem bylo nevhodné z důvodu neefektivního využití plochy a nepřístupnosti rozhraní při nezavedení modulu do aplikace. V případě přepisování kódu aplikace jsem narazil na problém. Spočívá v tom, jak graficky zakomponovat editor do aplikace a při tom nechat přístupné všechny ovládací prvky. Další odstavce popisují výsledné řešení, které je možno zhlédnout na obrázku 5.1.

5.1 Editor

Největším problémem při vytváření editoru bylo jeho zakomponování do QDevKitu. Centrální okno je řešeno komponentou Qt, která se nazývá „QStackedWidget“. Jeho vlastnost je, že zobrazuje v daný čas jen jednu komponentu. Vloženy do něj byly dvě komponenty – první bylo okno pro zobrazování připojených zařízení a druhou byl panel s otevřenými terminály k jednotlivým zařízením. Přede mnou stála otázka. Jak přidat editor, aniž bych ztratil stávající funkčnost? Jako první možnost mě napadlo přidávat editor do panelu s terminály. Ale tento panel byl navržen pouze pro přidávání terminálů a ničeho jiného. Přidáním editoru by se muselo přepsat mnoho z kódu. Druhou možností bylo přidat panel s editory do „QStackedWidgetu“. Problémem by bylo zobrazování panelu. Proto jsem zvolil vytvoření panelu jako hlavní okno a do něj jsem vložil původní „QStackedWidget“. Do panelu jsem poté mohl vkládat jakoukoliv další komponentu, v mém případě editor. Jiná řešení se zdála neschůdná kvůli možnosti ztráty přístupu k některým ovládacím prvkům QDevKitu.

Třída MainWindow

Třída MainWindow byla součástí QDevKitu. Využil jsem ji pro správu více editorů. Nové editory se ukládají do kolekce, aby k nim byl hromadný přístup. Přepsal jsem metodu, ve které se vytvářelo centrální okno. Vytvoří se tak panel pro vkládání záložek a do něj se vloží záložka pro komunikaci s FITkitem. Takto je vše připraveno na vkládání editorů.

Do menu jsem přidal záložku editor. Po rozbalení záložky se objeví funkce, které byly popsány v návrhu. Tímto byl splněn grafický návrh editoru (viz 4.3).

Dále jsem zde implementoval metody pro přístup k editoru:

Vytvořit editor lze pomocí jedné ze tří metod. První umožňuje vytvoření prázdného editoru, další dvě umožňují otevření editoru se souborem. Všechny metody vrací ukazatel

na editor. Při otvírání souboru se zkontroluje, jestli již soubor není otevřen. Jestliže ano, tak se pouze přepne na odpovídající záložku. Navíc je možno otvírat soubor pomocí dialogu nebo zadání cesty. Pokud soubor neexistuje, otevře se prázdný editor.

Pro ukládání existují následující metody. Jedna ukládá změněný soubor, druhá soubor s jiným jménem a třetí uloží všechny změněné soubory.

Zavírání provádí jedna přetížená metoda. Buď zavírá aktuální editor aktivní v panelu nebo editor podle čísla záložky.

Zjištění změn Jestliže byl soubor změněn jiným editorem, je možno tuto změnu detekovat při přepnutí záložky.

Změna názvu provádí změnu názvu záložky při uložení editoru a při změně jeho modifikace.

Přenačítání zásuvného modulu Editorem je možné přepisovat soubory zásuvných modulů v jazyce Python. Po uložení souboru si QDevkit sám zjistí, že byl zásuvný modul změněn a modul přenačte.

Signály jsou zavedeny z důvodu možného napojení dalších komponent. K dispozici jsou signály informující o jedné z těchto operací: otevření, zavření, uložení a změnění editoru.

Rozhraní jsem implementoval pro použití jak v jazyce C++, tak v jazyce Python. Bral jsem ohledy hlavně na to, aby se mohlo přistupovat k editorům z různých zásuvných modulů. Implementoval jsem signál informující o otevření editoru, který v parametru předává ukazatel na editor. Po připojení na signál je možné jakýkoliv vytvořený editor měnit. Signál vyslaný při uložení souboru dovoluje zjistit název souboru a na jeho základě přenačítá zásuvný modul v jazyce Python.

Třída Editor

Třída Editor v sobě zapouzdřuje veškerou funkčnost jednoho editoru. Ústředním oknem je instance třídy QScintilla pro psaní zdrojového kódu. Nad ní je nástrojová lišta pro přístup k funkcím QScintilli. Pod ní je stavový řádek, kde se zobrazuje nápověda a číslo řádku s kurzorem.

V nástrojové liště jsou akce zobrazeny jako obrázky. Po najetí kurzorem myši na obrázek se ve stavovém řádku zobrazí nápověda. Obrázky jsem převzal z projektu QScintilla, který patří pod licenci LGPL, nebo jsem některé vytvářel sám v nástroji Gimp[1].

Do nástrojové lišty jsem vložil následující akce:

- Ukládání – uložit a uložit jako ...
- Operace se schránkou – kopírovat, vložit vyjmout.
- Zvětšení písma – přiblížit, oddálit.
- Operace s historií úprav – zpět a vpřed.
- Vyhledávání – zobrazit vyhledávání, najít další, najít předchozí.

[1] http://www.gimp.org/

- Změna kódování, změna Lexeru, doplňování, číslování řádek, zakomentování, odkomentování.

Zajímavým nedostatkem QScintilli se stala skutečnost, že sice dokáže vyhledávat v textu *další* slova, ale ne *předchozí*. Po každém vyhledání slova se kurzor posune na konec slova, ale právě od pozice kurzoru se vyhledává další text. Při vyhledání předchozího slova se proto objeví to samé slovo, které jsme už jednou našli. Odstranění tohoto nedostatku se dá jednoduše řešit tak, že před vyhledáním předchozího výskytu slova posuneme kurzor na začátek slova, což je mé použité řešení.

Dalším nedostatkem se stalo to, že QScintilla neposkytuje způsob jak zakomentovat či odkomentovat řádek. Problém jsem vyřešil jednoduchým přidáním řádkového komentáře na začátek řádky, který jsem zjistil na základě použitého lexeru. Odkomentování řeším podobným způsobem – odebráním komentáře na začátku řádky.

Již v návrhu bylo zmíněno, že QScintilla neřeší kódování textu. Tuto funkčnost dodává framework Qt a jeho třídy *QTextCodec* a *QTextStream*. *QTextStream* zajišťuje převod mezi kódováním a *QTextCodec* dodává potřebná pravidla. V implementaci jsem využil rozbalovacího seznamu, kde si uživatel vybere daný kodek a zbytek zajistí jmenované třídy. Editor se také sám snaží uhodnout použité kódování. Pokud je na prvních dvou řádcích uveden text „coding: *kódování* “ nebo „encoding=„*kódování*“ “, porovná se text *kódování* se známými typy kódování. Vyhledávání se provádí na základě regulárních výrazů. Pokud je kódování nalezeno, je použito.

Pokud se editor stal aktivním, musí zkontrolovat, jestli náhodou editovaný soubor někdo nezměnil. Stane-li se editor aktivní, zachytí se signál panelu s editory. Objevení změn jsem implementoval na základě poslední změny souboru. Když se soubor otevře, editor si poznačí jeho poslední čas změny. Když se editor stane aktivním, zkontroluje čas poslední změny a informuje o tom uživatele.

Kontextové menu bylo potřeba vytvořit celé nové, aby k němu mohl být zaveden přístup. QScintilla je pouze nástavbou editoru Scintilla, její instance má své vlastní kontextové menu, do kterého není z Qt přístup kvůli vzájemné nekompatibilitě tříd. Jediný možný způsob je vypnutí zobrazování menu Scintilli a zobrazovat vlastní.

Rozhraní jsem přizpůsobil pro použití v jazyce C++ i Python:

- Název souboru – je možno zjistit celou cestu k souboru nebo jen zkrácené jméno
- Přístup k nástrojové liště
- Zjištění aktuálního textu – buď celý text nebo jedna řádka
- Nastavení pozice kurzoru v textu
- Zadání vlastního lexeru
- Přidání textu – na místo kurzoru nebo zadání souřadnic
- Vyhledávání, číslování řádků a doplňování

Při návrhu rozhraní byl brán ohled na další rozšiřování funkčnosti. Je možno vytvořit vlastní lexer a přiřadit ho do editoru, vytvořit vlastní šablony a na klávesovou zkratku je připsat do textu, zjišťovat cestu k souboru, vybrat řádky z textu, posouvat kurzor v textu, vyhledávat a na jeho základě přepisovat programově text. Jsou i další možnosti.

Třída Lexer

Pro implementaci zvýrazňování a doplňování se používá třída QsciLexer. Moje třída Lexer tovární třídou ostatních tříd QsciLexerXXX. Zajišťuje vytvoření správného lexeru zadáním souboru. Sama rozpozná typ souboru nebo vybere lexer „na žádost“. Také poskytuje třídě Editor řádkový komentář rozpoznaného jazyka. Lexery, které jsou rozpoznány:

- QsciLexerCMake
- QsciLexerMakefile
- QsciLexerCPP
- QsciLexerXML
- QsciLexerPython
- QsciLexerVHDL

Rozhodl jsem se implementovat rozpoznávání pomocí přípon souboru. Je to jednoduchý, rychlý a přenositelný způsob, jelikož se i na UNIXu objevují přípony. Databáze přípon je uložena v souboru „lexerfiles.xml“. Kořenový element se jmenuje *files*. Po něm následuje element s typem souboru. Listový element je nazýván *file* a obsahuje text přípony. Element s typem souboru může obsahovat nepovinný parametr se způsobem porovnání. Lze porovnávat pouze koncovku nebo celý název. Rozpoznání se provádí načtením databáze, porovnáním přípony a vrácením typu souboru.

Propojení Editoru a Lexeru je jednoduché. Stačí vytvořit instanci Lexuru a zavolat metodu „void setLexer(QsciLexer *lexer)“ a vše je zajištěno.

Přidání vlastního lexeru je možné s využitím metody „void SetLexer(QsciLexer* lex)“ třídy MainWindow. Za správu lexeru poté třída Lexer nezodpovídá.

Třída Find

Třída Find zapouzdřuje možnosti vyhledávání. Vytvořil jsem ji v designeru, abych mohl vzhled interaktivně měnit. Třída komunikuje s Editorem pomocí signálů, které říkají, jaké operace se mají provádět. Vyhledávat je možno směrem vpřed i vzad, dají se rozlišovat velká a malá písmena. Pro úplnost jsem doplnil metody přepisující vyhledaný text.

Nastavení

V nastavení pod záložkou vzhled jsem přidal možnost měnit velikost a typ písma v editoru. Abych sjednotil celé nastavení písma v záložce, musel jsem ukládat třídu Highlighter, která zde byla použita. Zvolil jsem na první dojem ne příliš šťastné řešení, protože kvůli dvěma hodnotám se musí vytvořit celá velká třída, ale programový přístup je jednotný a třída se může dále výhodně využít. Znemožnil jsem také při nastavování editoru použití záložky „Zvýrazňovač“. Nastavení zvýrazňování je v mém případě složitější, jelikož se musí nastavit zvýraznění pro každý lexer zvlášť, ale tato funkčnost může být přidána. Přesunutí tohoto nastavení pod jinou záložku by mohlo být matoucí z hlediska logického členění struktury nastavení.

Editor má i své vlastní nastavení pod záložkou *Editor*. Implementoval jsem možnost nastavit výchozí kódování textu, zobrazení číslování řádků a používání doplňování. Ukládání je uděláno jednoduše, a to přes třídu „QSettings“ s prefixem *editor/*.

5.2 Manager

Manager je implementován jako zásuvný modul. Využil jsem návodu, který je na stránkách projektu. Zásuvný modul se dědí ze třídy PluginInterface, který má čtyři čistě virtuální metody, a to „bool isLoaded()“ – informuje o tom, zda je modul zaveden, „int load()“ – zavádí modul do aplikace, „int unload()“ – odpojuje modul od aplikace a „int configure()“ – reaguje na změnu nastavení. Všechny metody se musí doimplementovat, jinak nebude možné modul přeložit.

Podle návrhu jsem implementoval tři druhy pohledu. První zobrazuje otevřené projekty, umožňuje je otevírat a uzavírat a přidávat do nich soubory. Druhý zobrazuje pohled na soubory v počítači a expanduje se na základě kliku do prvního pohledu. Třetí zobrazuje složku s projektem, jeho obsah se mění se na základě kliku do prvního pohledu.

Pohled „projekty“

Pohled se zobrazuje jako strom, kde první uzel je název projektu. Další uzly se skládají ze souborů v projektu:

- project.xml – informace o projektu, autorovi a všech souborech
- index.rst – dokumentace k projektu
- README – soubor s nápovědou
- zdrojové a hlavičkové soubory v jazyce C
- soubory v jazyce VHDL

Celý strom s projektem se vyváří po předání struktury definované ve *FCMake*u. Projekt je možno zavřít nebo do něj přidat další soubory. V pohledu může být i více projektů najednou. Je vytvářen ručně tak, jak je popsáno v dokumentaci Qt[2].

Výhodou pohledu je, že se jednotlivé soubory v projektu zobrazují v logických skupinách a zobrazují se pouze soubory, které jsou potřeba.

Pohledy „souborový systém“ a „složka projektu“

Jejich začlenění nebylo složité, jelikož framwork Qt nabízí možnost využití předdefinované třídy, která vytvoří a spravuje celý strom sama. V pohledu souborový systém je zobrazen celý strom souborového systému počítače. V pohledu složka projektu je pak už jen část souborového systému týkající se složky s projektem.

Oba pohledy jsou propojeny s pohledem projekty, aby reagovaly na změnu aktuální složky.

Pohled *souborový systém* je výhodný při otevírání souboru, který nepatří do žádného z projektů, a je možné vidět soubory z SVN.

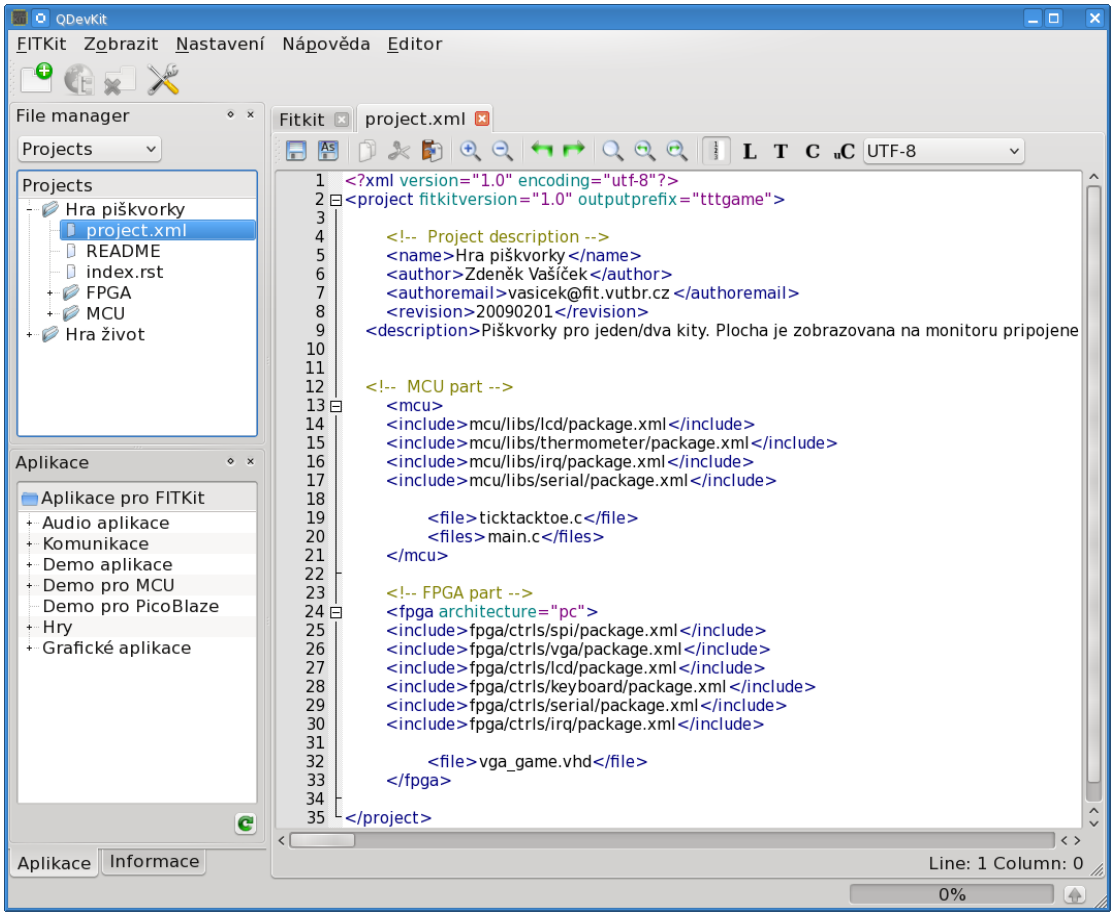
Výhodou pohledu *složka projektu* je přístup i k ostatním souborům v projektu než v pohledu *projekty*.

[2] http://doc.qt.nokia.com/4.6/qabstractitemmodel.html

5.3 Propojení

Veškeré propojení všech komponent jsem provedl v Manageru. Z něj je možné otevírat editor po zavolání metody *open(QString path)* a předání cesty k souboru. Otevírat soubory je možné ze všech pohledů. Zpáteční interakce není.

Aby bylo možné přidávat projekty do stejnojmenného pohledu, přidal jsem do stromu aplikací možnost otevřít projekt. Manager pak dostane informace o projektu ve struktuře *FCMake*u.



Obrázek 5.1: Obrázek aplikace QDevKit po úpravách.

Kapitola 6

Závěr

Cílem práce bylo navrhnout rozšíření, které by dovolovalo upravovat a psát zdrojové kódy v platformě QDevKit. Nastudováním vlastností a funkcí moderních editorů jsem získal cenné znalosti, které jsem využil k realizaci rozšíření. Výstupem se stal editor zdrojových kódů pro jazyky C, VHDL, Python, XML, CMAke a Makefile. Zabudována byla i podpora zvýrazňování, doplňování kódu, vyhledávání a další užitečné vlastnosti.

Aby bylo možné přistupovat k jednotlivým projektům, byl vytvořen správce projektů se schopností komunikace s editorem a QDevKitem. Dokáže zobrazit soubory v otevřených projektech a otevírat je v editoru.

Aplikace byla testována skupinou lidí, kteří se zabývají informatikou nebo ji studují. Objevilo se několik závažných chyb, které jsem opravil. Důležitá byla zpětná odezva testujících. Zjistil jsem, co by bylo podle nich vhodné předělat a co jim naopak vyhovuje. Na základě jejich připomínek jsem dodělal do aplikace klávesové zkratky.

V budoucnu bych chtěl editor rozšířit o doplňování z jiných zdrojů, než je samotný text, vyřešit možnost definovat typy písma pro doplňování a zabudovat vložení šablony do editoru na klávesovou zkratku. Správce souborů bych rád rozšířil o vytvoření nového projektu. Objevil by se průvodce, který by si od uživatele vyžádal informace. Na jejich základě by se vygenerovaly všechny potřebné soubory a do nich by se vložila šablona.

Chtěl bych v tomto projektu pokračovat i po ukončení celé práce a rozšiřovat jeho funkčnost. Tak by se stal mocným pomocníkem při vyvíjení aplikací pro FITkit.

Literatura

[1] Ahmed, I.; suk Lhee, K.; Shin, H.; aj.: Fast File-type Identification. http://www.alphaminers.net/thesis/Internationaldf.

[2] Hyvönen, E.; Mäkelä, E.: Semantic Autocompletion. http://www.cs.helsinki.fi/u/eahyvone/publications/2005/autocompletion.pdf.

[3] Liberty, J.: *Naučte se C++ za 21 dní*. Computer Press, a. s., 2007, iSBN 978-80-251-1583-1.

[4] McDaniel, M.: *Automatic File Type Detection Algorithm*. Diplomová práce, James Madison University, 2001.

[5] Pollock, W.: *The Book of Qt 4: The Art of Building Qt Applications*. No Starch Press, Inc., 2006, iSBN 978-3-937514-12-3.

[6] Robbes, R.; Lanza, M.: Improving Code Completion with Program History. http://www.dcc.uchile.cl/ rrobbes/pdfs/JASE2010-completion.pdf.

[7] Russkih, I.; Ketkov, Y.: Colorer: Syntax Analysis Framework for Source Code Editors and Integrated Development Environments. http://colorer.sourceforge.net/papers/irusskih-colorer-overview-eng.pdf.

[8] Vašíček, Z.: FITkit. http://merlin.fit.vutbr.cz/FITkit/uvod.html.

[9] Vašíček, Z.; Vavruša, M.: QDevKit - Windows. http://merlin.fit.vutbr.cz/FITkit/docs/navody/qdevkit.html, 2009-03-19 [cit. 2010-04-16].

[10] WWW stránky: Eclipse documentation. http://www.eclipse.org/documentation/.

[11] WWW stránky: Online Reference Documentation. http://doc.qt.nokia.com/.

[12] WWW stránky: reStructuredText. http://docutils.sourceforge.net/rst.html.

[13] WWW stránky: Scintilla Documentation. http://www.scintilla.org/ScintillaDoc.html, Last edited 4/April/2010 NH [cit. 2010-04-16].

Dodatek A

Obsah CD

- Zdrojový kód programu (`src/`),
- Binární verze programu pro systém Windows (`win/`),
- Text práce v elektronické podobě (`text/projekt.pdf`),
- Projektová dokumentace vygenerovaná programem Doxygen (`doc/`),
- Seznam klávesových zkratek programu (`doc/shortCut.pdf`)
- Návod na zprovoznění aplikace (`INSTALL`)