

Technická dokumentácia projektu

(Textový editor obohatený o grafické prvky)

Tímový projekt

Autor:	tím č.10 – Innovators
Téma projektu:	textový editor obohatený o grafické prvky (TrollEdit)
Vytvorený:	02.10. 2011
Stav:	finálny
Vedúci projektu:	Ing. Peter Drahoš
Vedúci tímu:	Bc. Lukáš Turský
Členovia tímu:	Bc. Marek Brath Bc. Adrián Feješ Bc. Maroš Jendrej Bc. Jozef Krajčovič Bc. Ľuboš Staráček
Kontakt:	tp-team-10@googlegroups.com

Obsah

1	Úvod	1
2	Analýza.....	2
2.1	Existujúce riešenia editorov	2
2.1.1	eTextEditor (e)	2
2.1.2	SciTE	2
2.1.3	Notepad++	4
2.2	Analýza predchádzajúceho riešenia nástroja TrollEdit	5
2.2.1	Inicializácia editora, otvorenie súborov	5
2.2.2	Práca s editorom	5
2.2.3	Programovanie v editore	6
2.2.4	Komentáre	6
2.2.5	Bloky	7
2.2.6	Práca so súbormi, prílohami	7
2.2.7	Syntaktický analyzátor	8
2.2.8	Gramatika	8
2.2.9	Literate programming	8
2.3	Analýza použitých technológií	8
2.3.1	Qt	8
2.3.2	Qt Quick a jazyk QML	9
2.3.3	Jazyk Lua	10
2.3.4	Knižnica LPeg	11
2.4	Analýza spracovávania syntaktického stromu	11
2.4.1	Gramatiky	12
2.4.2	Rozhranie Lua - Qt	13
3	Špecifikácia požiadaviek	15
3.1	Funkcionálne požiadavky	15
3.2	Nefunkcionálne požiadavky	16
3.3	Analýza požiadaviek pre paralelizmus	16
4	Návrh riešenia	18
4.1	Diagram prípadov použitia	18
4.2	Architektúra programu	21
4.3	Návrh GUI	22

4.4	Návrh funkcionality UNDO/REDO	24
4.4.1	Qt Undo Framework.....	24
4.4.2	QScintilla.....	25
4.4.3	QTextDocument	25
4.5	Návrh funkcionality pre shortcuts	26
4.5.1	Dialógové okno	26
4.5.2	Proces editovania.....	28
4.5.3	Editovanie, načítanie a ukladanie skratiek	29
4.6	Návrh spracovania syntaktického stromu.....	30
4.6.1	Experimentovanie s LUA C API / FFI library	32
4.7	Návrh riešenia paralelizmu	34
4.7.1	QRunnable prístup.....	34
4.7.2	QtConcurrent prístup.....	35
4.7.3	Zdieľanie zdrojov medzi paralelnými vláknami	36
4.7.4	Zhodnotenie.....	36
5	Implementácia prototypu.....	37
5.1	Implementácia 2 módov editácie	37
5.2	Experimentovanie a implementácia funkcionality Undo/Redo.....	38
5.2.1	Experimentovanie.....	38
5.2.2	Samotná implementácia	39
5.3	Experimentovanie a implementácia paralelizmu.....	39
5.3.1	Experimentovanie.....	39
5.3.2	Samotná implementácia	40
5.4	Implementácia spracovania AST pomocou LUA C API.....	40
6	Testovanie	43
6.1	Akceptačné testy pre overenie funkcionality.....	43
7	Zhodnotenie a návrhy do ďalšej fázy riešenia.....	45
7.1	Návrhy pre optimalizáciu riešenia	45
7.1.1	Optimalizácia paralelizmu.....	45
7.1.2	Optimalizácia spracovania AST.....	45

Zoznam obrázkov

Obr. 1 Pracovný cyklus s použitým Qt Quick.....	9
Obr. 2 Diagram prípadov použitia.....	18
Obr. 3 Architektúra programu.....	22
Obr. 4 Hierarchické rozdelenie okien programu.....	22
Obr. 5 Splash screen programu.....	23
Obr. 6 Predbežný návrh hlavného menu programu.....	24

Zoznam tabuliek

Tab. 1 Porovnanie funkcionalít.....	5
Tab. 2 Funkcionálne požiadavky.....	15
Tab. 3 Nefunkcionálne požiadavky.....	16
Tab. 4 Prípad použitia UC13 Undo/redo.....	19
Tab. 5 Prípad použitia UC14 Prepínanie módov písania.....	19
Tab. 6 Prípad použitia UC14 Použitie shortcuts.....	19
Tab. 7 Prípad použitia UC16 Vyhľadávanie v kóde.....	20
Tab. 8 Prípad použitia UC17 Export súborov.....	20
Tab. 9 Prípad použitia UC18 Nastavenie programu.....	21
Tab. 10 Prípad použitia UC19 Zobrazenie sw metrík.....	21
Tab. 11 Akceptačný test funkcionality Undo/redo.....	43
Tab. 12 Akceptačný test funkcionality skratiek (Shortcuts).....	43
Tab. 13 Akceptačný test funkcionality 2 modov editora.....	43

1 Úvod

Súčasný textový editor zdrojových kódov len minimálne využíva možnosti grafickej reprezentácie, čo je veľká škoda vzhľadom na to, že práve obohatenie editorov o grafické prvky by mohlo v mnohých veciach uľahčiť prácu s takýmto editorom. Sprehľadnil by sa zdrojový kód, zjednodušila a zefektívnila nie len jeho tvorba, ale aj údržba a prezentácia, a vnieslo by to možnosť nového pohľadu na integráciu dokumentácie s programom.

Práve to by malo byť výsledkom tohto projektu, ktorého cieľom bude pokračovať vo vývoji multiplatformového editora „TrollEdit“ (ktorý bol riešený v roku 2009/10 tímom s názvom UFOPAK) pre editovanie najmä zdrojových kódov, ktorý bude využívať grafické prvky na zjednodušenie a zefektívnenie práce programátora. Naším zameraním bude rozšírenie stávajúcej funkcionality do podoby vhodnej pre reálne nasadenie editora do praxe.

Tento dokument obsahuje zhrnutie všetkých riešení nášho tímu na tomto projekte od analýzy až po implementáciu.

2 Analýza

V rámci tejto časti sme sa zamerali na analýzu existujúceho riešenia nie len z pohľadu toho, čo všetko už editor dokáže, prípadne nedokáže, ale boli analyzované aj kľúčové funkcionality, ktoré by sme radi do editora zakomponovali.

2.1 Existujúce riešenia editorov

V súčasnosti na trhu existuje mnoho editorov od jednoduchších až po zložitejšie, s rôznymi funkcionalitami a metódami ktoré uľahčujú prácu používateľa. Pri vytváraní projektu sa môžeme inšpirovať súčasnými ako sú eTextEditor (e), SciTE alebo Notepad++.

2.1.1 eTextEditor (e)

Textový editor pre Microsoft Windows s výkonnými funkciami pre úpravu textu. Vznikol ako alternatíva pre TextMate, pretože práve tento editor bol oslavovaný mnohými programátormi. Umožňuje rýchlu a jednoduchú manipuláciu s textom, automatizuje všetku manuálnu prácu, čím vám napomáha lepšiemu sústredeniu sa na písanie. Medzi jeho pozoruhodné vlastnosti patrí osobný systém pre správu revízií, rozvetvené, viacstupňové, grafické undo, možnosť prevádzkovať TextMate zväzkov pomocou Cygwin. Významný prvok propagácie a marketingu „e“ je jeho schopnosť púšťať mnoho TextMate zväzkov priamo z repozitára MacroMates CVS.

„E“ podporuje viacnásobný výber textu. Ak je podržaný kláves Ctrl, potom dvojklik/viacnásobný výber slov, je vtedy možné editovať všetky tieto slová naraz. Vlastnosť nájsť a premiestniť, dáva okamžitú vizuálnu spätnú väzbu, zvýraznenie požiadaviek, ktoré sú písané. Táto vlastnosť je užitočná najmä pri používaní regulárnych výrazov. Keďže väčšina zväzkových príkazov sa spolieha na Unixové príkazy, ktoré nie sú k dispozícii pre Windows, e používa sadu nástrojov Cygwin. Menšou nevýhodou je trochu pomalé otváranie súborov.

2.1.2 SciTE

Editor založený na Scintille. V SciTE nenájdete žiadneho správcu súborov, Project Manager či integrovaného FTP klienta, je to teda čistý editor. SciTE môže držať viac súborov v pamäti naraz, pričom len jeden súbor bude viditeľný. SciTE zvýrazňuje syntax a podporuje množstvo jazykov (HTML, PHP, SQL, CSS, Java, . . .). Má otvorený zdrojový kód. Obdĺžnikové bloky textov je možné vybrať podržaním klávesy Alt, zatiaľ čo je myš ťahaná ponad text. Používajú

sa rôzne funkcie ako skratky, nápoveda, editačné možnosti, vyhľadávanie, pohyb kurzora, kompilácia, dopĺňanie textu, makrá, komentáre, zobrazenie výstupu.

Tu uvádzame krátky prehľad základných a často používaných vlastností:

Skratky: Napíšete slovo, stlačíte klávesu Ctrl+B a rozvinie sa skratka, napr. if môže byť namapované, ako if (|) {\n\t\n}. Ich využitie je efektívne z hľadiska času, ak označíme kus kódu, stlačíme klávesy Ctrl+Shift+R, napíšeme if a kód sa obalí kompletnou konštrukciou if.

Nápoveda: Kláves F1 zobrazí nápovedu k funkcii, na ktorej je kurzor. Aj tu je možnosť namapovať si pre ľubovoľný jazyk to, čo vám najviac vyhovuje.

Editačné možnosti: Základné editačné možnosti sú samozrejmosťou. Duplikácia riadka pomocou Ctrl+D či jeho prehodenie s predchádzajúcim riadkom Ctrl+T.

Vyhľadávanie: Ctrl+F3 vyhľadá slovo pod kurzorom alebo označený text. Ctrl+Shift+F vyhľadáva vo viac súboroch štandardnými nástrojmi grep alebo findstr. Je možné doplniť si aj vlastnú funkciu na vyhľadávanie, teda môžete napríklad vyhľadávať len v reťazcoch a text nájdený inde sa odignoruje.

Pohyb kurzora: Klávesová skratka Ctrl+E presunie kurzor k odpovedajúcej zátvorke. Šikovník je aj funkcia pre prechod medzi časťami slov, na rozdiel od Ctrl+šípky zohľadňuje aj podtržník a zmeny veľkosti písmen v slove či odseku (bloky textu oddelené prázdny riadkom).

Kompilácia: Skontrolovanie syntaxe a prenesenie na riadok, kde sa daná chyba nachádza.

Dopĺňanie textu: Ctrl+Space doplní slovo z pevného zoznamu a Ctrl+Enter potom zo slov obsiahnutých v zozname.

Makrá: Funkčnosť je možné rozširovať makrami písanými v jazyku Lua.

Komentáre: Ctrl+Q prehodí zakomentovanosť označených riadkov, Ctrl+Shift+Q zakomentuje označený text.

Zobrazenie výstupu: Výstup externých programov sa zobrazuje v samostatnom okne priamo v rámci editora. Okno sa dá zapnúť či vypnúť pomocou klávesy F8.

2.1.3 Notepad++

Voľne dostupný editor zdrojového kódu [4], ktorý aj podporou viacerých jazykov nahrádza Notepad. Beží v prostredí MS Windows pod licenciou GPL. Avšak môže byť viacplatformovým využitím softvéru, napr. WINE. Je založený na komponente Scintilla a napísaný v jazyku C++ a využíva čisté Win32 API a STL, ktoré zabezpečuje vyššiu rýchlosť a menšiu veľkosť programu.

Podporuje zvýraznenie syntaxe pre 44 jazykov, skriptovacie a značkovacie jazyky. Užívatelia môžu tiež definovať svoj vlastný jazyk pomocou zabudovaného zásuvného panelu. Pre väčšinu podporovaných jazykov môže užívateľ urobiť svoj vlastný zoznam API (alebo stiahnuť API súbory zo sekcie). Akonáhle je API súbor pripravený, zadajte Ctrl+Space na začatie tejto akcie.

Podporuje multi-dokument, čo umožňuje úpravu viacerých dokumentov naraz. Poskytuje dva pohľady v rovnakom čase. To znamená, že môžete zobraziť dva rôzne dokumenty súčasne. Môžete vizualizovať (editovať) v dvoch náhľadoch jeden dokument a v dvoch rôznych pozíciách. Úprava dokumentu v jednom zobrazení sa bude vykonávať v inom náhľade.

Hľadanie a nahrádzanie reťazca v dokumente pomocou regulárnych výrazov. Úplná podpora drag-and-drop. Môžete otvoriť dokument pomocou tejto funkcie, presunúť tak dokument z pozície. Užívateľ si môže nastaviť pozíciu pohľadov dynamicky (len v režime dvoch zobrazení: oddelovač môže byť nastavený horizontálne alebo vertikálne). Ak máte upraviť či vymazať súbor, ktorý sa otvoril v Notepad++, ste upozornení na aktualizáciu dokumentu (reload súbor alebo odstránenie súboru). Možnosť funkcie priblíženia a oddialenia, ktorá je zložkou Scintilly.

Podporuje viacjazyčné prostredie. Takže je možné používať napríklad aj čínštinu, hebrejčinu, kórejštinu či arabčinu. Poskytuje funkciu záložky, kde si užívateľ môže kliknúť na rozpätie alebo pomocou Ctrl+F2 prepínať návestia. Pre dosiahnutie záložiek stačí stlačiť F2 (ďalšie záložky), alebo Shift+F2 (predchádzajúca záložka). Vymazanie všetkých záložiek sa koná pomocou Menu, kde kliknete na Hľadať -> Odstrániť všetky záložky. Ak vsuvka zostane pri jednom zo symbolov {}()[], symbol vedľa vsuvky a jeho opak budú zvýraznené, rovnako ako smernice za účelom ľahšieho nájdania bloku.

Tab. 1 Porovnanie funkcionalít

	TextEditor (e)	SciTE	Notepad++
Spell checking	plugin	nie	plugin
Viacnásobné undo/redo	áno	áno	áno
Selekcie blokov	áno	áno	áno
Zvýraznenie syntaxe	áno	áno	áno
Automatické dopĺňanie	áno	áno	áno
Integrácia kompilátora	áno	áno	áno
Spoločné editovanie na viacerých počítačoch	áno	nie	plugin

2.2 Analýza predchádzajúceho riešenia nástroja TrollEdit

Vzhľadom na to, že pokračujeme na projekte, ktorý bol vyvíjaný v rámci minuloročného tímového projektu bolo nutné vykonať podrobnú analýzu predchádzajúceho riešenia. Výsledkom je porovnanie medzi reálnym stavom editora a technickou dokumentáciou minulého tímu. Správa o stave bola rozdelená podľa jednotlivých funkčných častí.

2.2.1 Inicializácia editora, otvorenie súborov

Implementované:

- pri načítaní súboru určenie správnej gramatiky a jej kontrola
- pri otvorení súboru automatická analýza a zobrazenie do blokov
 - komentáre sú prepojené v blokoch avšak umiestnené sú mimo riadku, na ktorý sa odvolávajú, bolo by vhodné ich umiestniť vedľa textu
- história naposledy otvorených súborov
- obsahuje modul pre syntaktickú analýzu
- novšia LuaJit verzia – rýchle spracovanie menších súborov

Chýba:

- veľké súbory stále dosť pomalé na prácu
- pri otvorení napr. Analyzer.cpp nevyrobí správne bloky častí súboru, všetko brané ako samostatný riadok

2.2.2 Práca s editorom

Implementované:

- zvýrazňovanie syntaxe až na úrovni blokov, teda je možné určiť grafické vlastnosti pre všeobecné prvky naprieč viacerým jazykom a gramatikám
- popis zvýrazňovania jednotlivých blokov, ktoré majú byť zvýraznené, je obsiahnutý v konfiguračnom súbore
- všetko v rámci editačného okna editora je možné presúvať

- existuje možnosť „*Edit plain text*“ pre úpravu textu, vtedy je zobrazené v samostatnom okne všetko ako čistý text
- v súbore `text_item.cpp` implementovaný pohyb medzi blokmi po stlačení kláves
- samostatné zoomovanie každého otvoreného súboru nehl'adiac na tie ostatné
- presúvanie blokov v editore
- vyhľadávanie v texte zobrazí bloky, v ktorých sa text nachádza

Chýba:

- klávesové skratky veľmi chýbajú
 - nejaké už sú implementované (v Menu -> File sa dajú vidieť)
 - priamo na začiatku v súbore `main_window.cpp` sa priradujú skratky k akciám
- chýba možnosť Undo, Redo, Copy, Paste
 - len cez kontextovú ponuku cez pravé tlačítko je možná
- selekcia textu aj v rámci viacerých blokov
- malé možnosti vyhľadávania
 - chýba možnosť pri veľkých súboroch krokovania nájdených výskytov vyhľadávania
 - vyhľadáva len v aktuálnom súbore
 - lupa nie je klikateľné tlačítko
- naraz otvorená len jedna pracovná plocha (workspace)
 - triedu `BlockGroup` je v budúcnosti možné využiť na paralelné zobrazenie viacerých hierarchií (`BlockGroup`) na jednej scéne (`DocScene`) – viacero pracovných plôch
- konfiguračný súbor sa načíta len raz, pri spustení editora (sprevádza ho)
- základná štruktúra menu pre prácu s textom a options je zakomentovaná a neimplementovaná

2.2.3 Programovanie v editore

- analýza zdrojového kódu je časovo náročnejšia a preto sa spúšťa iba v čase prechodu písania na ďalší riadok.
- funguje automatické odsadzovanie pri analýze
 - medzery a tabulátory sa nezobrazujú a realizujú sa len ako prázdny priestor pred príslušným blokom
 - prebytočné zbavenie sa medzier
- znak konca riadku je nahradený nastavením príznaku v bloku

2.2.4 Komentáre**Implementované:**

- posúvanie komentárov – plávajúce komentáre
- funguje zobrazenie jednoriadkových aj blokových komentárov ako samostatný blok

- šípka ku blokovému komentáru začína vždy na začiatku riadku
- šípka ku jednoriadkovému komentáru začína od konca daného riadku
- funguje CTRL + Ľavé tlačítko myši = vytvorí na danom mieste nový ale len bežný blok (podobný ako ten pre komentár), pričom ho prepojí šípkou z miestom kde sa nachádzal kurzor

Chýba:

- textové komentáre ako samostatné bloky, nie je možné napísať tvrdú medzeru
- šípka by mohla byť aj zmysluplnejšie ukazujúca na daný blokový komentár
- textové komentáre len ako bežné bloky, nie je možnosť rovno písať dokumentáciu ako bolo spomínané cez dokumentačné bloky
- chýba možnosť vytvárania dokumentačných blokov, ale funkčne je implementovaná

2.2.5 Bloky**Implementované:**

- možnosť presúvať bloky, alebo časti blokov
- pomocná čiara pri presune

Chýba:

- plávajúce bloky sa nedajú zmazať priamo, jedine postupným vymazaním ich obsahu
- chýba možnosť samostatne vytvárať bloky a prepájacie šípky
- šípka odkazuje len na jeden blok
- vždy možné presúvať len jeden blok naraz, chýba výber viacerých blokov
- chýba skrývanie blokov, nezobrazuje možnosť na skrytie blokov
- pri inicializácii sa nedeteguje prekryvanie viacerých blokov na jednom mieste
- pri vkladaní bloku rozostupovanie ostatných blokov

2.2.6 Práca so súbormi, prílohami**Implementované:**

- pridanie súboru ako prílohy v podobe bloku,
 - treba mať označený nejaký blok, aby bolo možné určiť časť súboru od ktorej sa priraduje
- prílohy vkladá ako odkaz
- obrázky vie rovno zobrazit'
- možnosť úpravy rozmerov obrázka
- ukladanie ako pôvodný súbor s komentármi, súbor bez komentárov, alebo ako PDF tlačit' (len printscreen v rámci ohraničenia pri tlači)

- žiadne pokročilé prvky popisujúce obsah dokumentačných blokov ako bolo spomínané

2.2.7 Syntaktický analyzátor

Implementované:

- analýza realizovaná v jazyku LUA za pomoci knižnice LPeg
 - možnosť rozširovania o ďalšie gramatiky (načítavajú sa z priečinka grammars)
 - výstupom je LUA tabuľka obsahujúca ďalšie tabuľky a tento systém tabuliek zodpovedá syntaktickému stromu
- vytváranie AST na strane jazyka LUA a jeho prenos do C++

2.2.8 Gramatika

Implementované:

- základná gramatika default_grammar.lua
 - rozloženie ľubovoľného textu na slová a riadky
 - popísané povinné konštanty, ktoré musia gramatiky obsahovať
 - funkcie na testovanie gramatík
- gramatika pre C, LUA a XML

2.2.9 Literate programming

Implementované:

- možnosť vkladať ku kódu okrem klasických komentárov aj obrázky

Chýba:

- neukladajú sa pridané obrázky a iné formátovacie zmeny v dokumente
- ukladanie dokumentácie do RTF formátu

2.3 Analýza použitých technológií

Pri implementácii budeme používať nástroje a technológie, ktoré používal predchádzajúci tím UFOPAK počas vývoja editora. Nosnými technológiami sú Qt SDK, Lua a využitie RTF, ktoré v skratke predstavíme ako aj dôvod, prečo sme sa rozhodli pokračovať v ich používaní.

2.3.1 Qt

Qt je implementačný nástroj založený na jazyku C++. Je to technológia, pomocou ktorej je možné vyvíjať aplikácie pre rôzne platformy. Qt umožňuje vytvárať a jednoducho nasadzovať aplikácie pre počítače, mobilné telefóny, ale aj vnorené systémy (MP3prehrávače), bežiace pod operačnými systémami Windows, Linux, MAC OS, Symbian. Multiplatformovosť je

práve jedna z rozhodujúcich výhod, kvôli ktorým je editor implementovaný pomocou tohto nástroja. Qt je v súčasnosti dostupné pod komerčnou ale aj GNU GPL v3.0 licenciou.

Nástroj Qt ponúka okrem množstva tried a knižníc pre tvorbu GUI aplikácií aj vlastné vývojové prostredie Qt Creator. Uvažované možnosti práce s nástrojom Qt boli nasledovné:

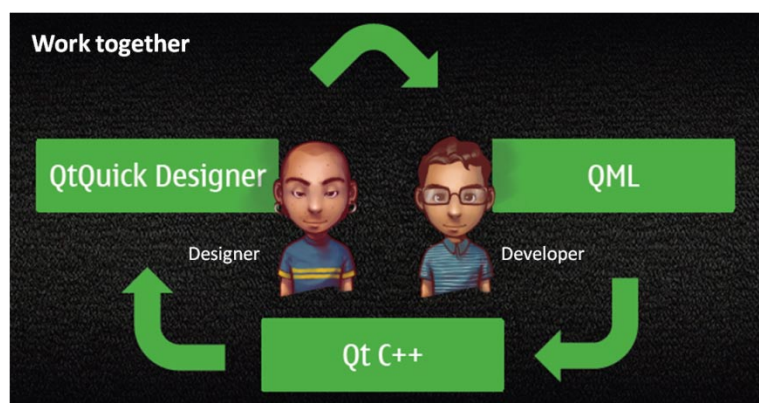
- Qt modul pre vývojové prostredie Eclipse
- Qt modul pre vývojové prostredie Visual Studio
- Integrované vývojové prostredie Qt Creator

Rozhodli sme sa pre použitie prostredia Qt Creator, keďže poskytuje samostatné vývojové prostredie, čiže pre naše potreby by malo byť ideálne, a taktiež integruje v sebe viacero novších technológií a prístupov, ktoré nám pomôžu pri vývoji. Napr. obsahuje novú technológiu Qt Quick.

2.3.2 Qt Quick a jazyk QML

Qt Quick je nová technológia určená pre rýchle vytváranie jednoduchých a bohatých používateľských rozhraní aplikácií pre rôzne platformy. Qt Quick obsahuje jazyk QML, ktorý je navrhnutý vychádzajúc z jazykov HTML, CSS, JavaScript, pričom spája ich výhody.

S použitím technológie Qt Quick je možné aby dizajnér navrhol UI podľa vlastnej fantázie a vývojár len doplnil logiku aplikácie, čo prináša obrovskú výhodu keďže vývojár a dizajnér majú každý iný pohľad na svet a nie vždy bolo možné nájsť konsenzus pri vytváraní aplikácie.



Obr. 1 Pracovný cyklus s použitým Qt Quick

Qt Quick umožňuje vytvárať rôzne animácie, ktoré využívajú knižnicu OpenGL. Takisto umožňuje navrhnuť dizajn jednotlivých používateľských prvkov aplikácie ako napr. tlačítka v grafických editoroch Adobe Photoshop, Autodesk Maya, Gimp.

Jednoduchosť technológie Quick možno vidieť v rozdiel medzi definovaním jednoduchého tlačítka klasickým spôsobom cez actionscript a novým pomocou jazyka QML.

Actionscript: **MenuButton.as**

```
public class MenuButton extends MovieClip
{
    public function MenuButton() {
        this.x = 60;
        this.addEventListener(MouseEvent.CLICK,
ClickBt);
    }
    function ClickBt(e:MouseEvent) {
        trace("clicked");
    }
}
```

QtQuick: **MenuButton.qml**

```
Item {
    x:60;
    MouseArea: {
        anchors.fill: parent;
        onClicked: print("clicked");
    }
}
```

Použitie technológie Qt Quick by mal pre nás veľký význam keďže nám umožňuje navrhnúť UI pre TrollEdit podľa našej potreby, ktorý by bol zaujímavejší ako súčasné UI riešenia editorov. To nám dáva možnosť v tomto smere vytvoriť kvalitnejší produkt.

Príklad dizajnu navrhnutého s použitím technológie Qt Quick je možno vidieť v takých aplikáciách ako Skype, VLC Media Player atď.

2.3.3 Jazyk Lua

Lua je rýchly procedurálny skriptovací jazyk, určený hlavne na vnorené používanie. Programátorské rozhranie (API) je navrhnuté tak, aby umožňovalo integráciu s programami napísanými v iných jazykoch (C, C++, Java, C#, . . .) vrátane skriptovacích (Perl, Ruby).

Filozofiou jazyka Lua je jednoduchosť a rozšíriteľnosť. Obsahuje základnú funkcionálnu a mechanizmy ako definovať čokoľvek, čo považujeme za potrebné. Týmto spôsobom je možné získať aj schopnosti objektovo orientovaných (rozhrania, dedenie) alebo funkcionálnych jazykov. Lua je dynamicky typovaná a obsahuje niekoľko atomických dátových typov doplnených o jednu dátovú štruktúru – tabuľku. Tabuľka funguje ako asociatívne pole a jej pomocou je možné simulovať iné štruktúry (pole, množina, hash tabuľka, strom, atď.) a tiež objekty v zmysle OO paradigmy.

Lua patrí medzi najrýchlejšie skriptovacie jazyky. Je implementovaná v štandardnom ANSI (ISO) C, čo sa prejavuje na jej vysokej prenositeľnosti. Funguje pod všetkými známymi platformami. Výhodou Lua je jej veľkosť (aktuálna verzia Lua 5.1.4 má 860KB aj s dokumentáciou), vďaka ktorej nie je problém pripojiť ju celú k aplikácii, ktorá ju používa.

Lua je vyvíjaná pod voľnou licenciou (MIT) a môže byť používaná zdarma na akékoľvek (aj komerčné) účely. Lua sa dnes často používa pri skriptovaní počítačových hier, ale využívajú ju aj iné programy ako napríklad Skype, Wireshark, VLC media player atď.

2.3.4 Knížnica LPeg

LPeg je knižnica jazyka Lua určená na hľadanie vzoriek v texte (pattern matching). Snaží sa odstrániť problémy spojené s používaním regulárnych výrazov, ktoré môžu byť pri komplikovanejších úlohách neprehľadné. Je postavená na gramatikách typu PEG (Parsing Expression Grammar) a formalizme podobnom bezkontextovým gramatikám. Na rozdiel od bežných gramatík, PEG nedefinuje jazyk, ale algoritmus na jeho rozpoznanie. LPeg poskytuje dva moduly s rozličným spôsobom práce. V prvom module `re` (skratka z `regex`) sú vzory popisované reťazcami so syntaxou odvodenou z regulárnych výrazov. Druhý modul `lpeg` pracuje so vzormi ako s premennými vlastného dátového typu a obsahuje viac spôsobov na ich vytváranie a spájanie. Obidva moduly podporujú vyhľadávanie (vyjadrené priamo vzorom) rovnako ako zachytávanie reťazcov na pokročilej úrovni. Vybraný text je možné ukladať do tabuliek, ľubovoľne zamieňať a inak transformovať. LPeg používa tzv. limitovaný backtracking, vďaka ktorému je veľmi rýchly a efektívny.

2.4 Analýza spracovávania syntaktického stromu

Na špecifikáciu gramatiky v jazyku Lua je využitá knižnica LPeg. Je vytvorená gramatika pre jazyk C, pričom vychádza zo zápisu v Bakchus-Naurovej forme (BNF). Gramatika sa nachádza v skripte a je dynamicky kompilovaná za behu aplikácie. Spoluprácu s jadrom systému zabezpečuje C API (štandardná súčasť jazyka Lua) a funguje na báze zásobníka, z ktorého čítajú a zapisujú obe strany. Komunikácia prebieha nasledovne:

- aplikácia spustí skript s gramatikou a gramatika sa skompiluje
- aplikácia zavolá LPeg funkciu `match`, ktorej vstupom je gramatika a text (kód), ktorý chceme analyzovať
- výstupom je Lua tabuľka obsahujúca ďalšie tabuľky a tento systém tabuliek zodpovedá syntaktickému stromu

- výstup je umiestnený na zásobník, z ktorého je postupne čítaný
- z Lua tabuliek sa zrekonštruuje AST strom v C++

Gramatika využíva funkcie na zachytávanie časti vstupu, ktoré zodpovedajú daným LPeg výrazom (podobným regulárnym výrazom). Ku každej zachytenej (lexikálnej) jednotke alebo skupine jednotiek je pripojený identifikačný kľúč (napr. `storage_class_specifier`, `number_constant`, `parameter_list`), ktorý v AST slúži na identifikáciu uzlov. Zachytené sú všetky znaky, teda aj tie, ktoré nie sú priamo lexémami, ako napríklad biele znaky (kľúč `whitechar`) alebo text, ktorý nezodpovedá gramatike jazyka (označený kľúčom `unknown`). Gramatika dosiaľ nie je úplne kompletná, neobsahuje podporu inštrukcií preprocesora v tele funkcií a vyžaduje si ďalšie testovanie.

2.4.1 Gramatiky

Gramatiky jazykov sú písané v jazyku Lua a nachádzajú sa v priečinku `/grammars`. Pre fungovanie programu je nutná existencia základnej gramatiky `default_grammar.lua`. Táto gramatika slúži na rozloženie ľubovoľného textu na slová a riadky a obsahuje funkcie používané na testovanie gramatík. V súbore s touto gramatikou sú tiež popísané povinné konštanty, ktoré musí každý súbor s gramatikou obsahovať ako napríklad prípony spracúvaných súborov, zoznam párových znakov a podobne. Pomocou knižnice LPeg vytvára Lua interpret tabuľkovú reprezentáciu stromu ako systému hierarchicky vnorených tabuliek bez explicitných kľúčov v tvare:

```
uzol1; uzol1:1; uzol1:1:1; :: :: :: ::; uzol1:2; :: ::; uzol1:3; :: :: :: ::
```

Názov uzla je vždy nasledovaný tabuľkami zodpovedajúcim jeho priamym potomkom. Zároveň by každá gramatika mala vrátiť len jednu tabuľku, ktorá ale nemusí nutne obsahovať len jeden koreň (napr. `a`; `b` je korektný výstup). Tabuľková štruktúra je definovaná priamo v syntaktických pravidlách gramatiky pomocou štandardných LPeg funkcií `lpeg.C`, `lpeg.Ct` a `lpeg.Cc`. Vo všeobecnosti sa predpokladá, že súbor s gramatikou obsahuje jednu kompletnú gramatiku (`full_grammar`) a ľubovoľný počet čiastkových gramatík (`other_grammars`). Plná gramatika sa využíva pri analýze celého súboru a ostatné gramatiky pri analýze menších častí. Napríklad, pre jazyk C sa používajú gramatiky *program* (analyzuje celý program v C), *top_element* (analyzuje funkcie, mimo-funkčné deklarácie alebo direktívy preprocesora) a *in_block* (analyzuje obsah bloku príkazov). Zmysel čiastkových gramatík je v tom, že

napríklad na vyhodnotenie syntaktickej správnosti skupiny príkazov nemôžeme použiť kompletnú gramatiku, pretože skupina príkazov nie je platným programom.

2.4.2 Rozhranie Lua - Qt

Celú komunikáciu medzi Lua a Qt/C++ zapuzdruje trieda *Analyzer*. Pri požiadavke na analýzu textu je vytvorená inštancia Lua interpretu a vykonaný príslušný skript. Daný text je potom spracovaný pomocou funkcie `lpeg.match` a výsledok je načítaný z Lua zásobníka.

Tabuľková hierarchia je paralelne prevádzaná do stromovej štruktúry reprezentovanej triedou *TreeElement*. Pomocnú funkciu má trieda *LanguageManager*, ktorá uchováva zoznam objektov triedy *Analyzer* pre všetky podporované jazyky.

Funkcie, ktoré sa starajú o analýzu kódu:

- Táto funkcia analyzuje celý kód

```
// analyze string, creates AST and returns root
TreeElement* Analyzer::analyzeFull(QString input)
```

- Funkcia analyzuje konkrétny podstrom AST, bez nutnosti prepisovať celú štruktúru

```
// reanalyze text from element and it's descendants, updates AST and
returns first modified node
TreeElement *Analyzer::analyzeElement(TreeElement* element)
```

- Analyzuje sa kód pomocou vybranej gramatiky, využíva pritom funkciu na prepis
LUA tabuliek do AST

```
// analyze string by provided grammar
TreeElement *Analyzer::analyzeString(QString grammar, QString input)
```

- Táto funkcia konvertuje tabuľky z jazyka LUA do stromovej štruktúry typu
TreeElement

```
// creates AST from recursive lua tables (from stack), returns root(s)
TreeElement *Analyzer::createTreeFromLuaStack()
```

Trieda *TreeElement* obsahuje smerník na predchodcu a svojich potomkov, takto vytvára stromovú štruktúru.

```
class TreeElement
{
    ...
protected: TreeElement *parent;
```

```
private:
    QList<TreeElement*> children;
    QString type;
    Block *myBlock;
    TreeElement *pair;
    bool lineBreaking;
    bool selectable;
    bool paragraphsAllowed;
    bool paired;
    bool floating;
    . . .
};
```

3 Špecifikácia požiadaviek

Keďže vychádzame z už existujúceho multiplatformového textového editora „TrollEdit“ obohateného o grafické prvky, popisujeme iba tie požiadavky na systém, ktoré chceme implementovať prípadne modifikovať v súčasnej verzii.

Hlavné ciele tohto projektu sú:

- Rozšíriť súčasnú - implementovanú funkcionálnu
- Modifikovať používateľské rozhranie - GUI
- Vytvoriť kvalitný produkt, ktorý bude úspešný a mohol by sa presadiť aj v praxi

3.1 Funkcionálne požiadavky

Pre TrollEdit boli identifikované funkcionálne požiadavky na základe dôkladnej analýzy predchádzajúceho riešenia a taktiež na základe podnetov od nášho vedúceho tímu, ktoré sú spísané v Tab.1.

Tab. 2 Funkcionálne požiadavky

ID	Požiadavka	Charakteristika	Priorita
F01	Možnosť Undo/Redo	Možnosť vrátiť zmeny naspäť a opačne	vysoká
F02	Podpora skratiek v editore (Shortcuts)	Možnosť spustiť funkcie programu pomocou klávesových skratiek	vysoká
F03	Dopytovanie sa do Lua		vysoká
F04	Podpora paralelizmu	Syntaktický strom by mal bežať na pozadí pod vlastným vláknom a program pod vlastným	vysoká
F05	2 módy písania	Prvý by bol klasický editor na úpravu kódu a po prepnutí by editor prešiel do druhého grafického módu.	vysoká
F06	Nastavenie programu	Možnosť rozšírených nastavení priamo v editore	stredná
F07	Podpora intellisense	Rozpoznávanie bežných kľúčových slov programovacích jazykov, ale aj najčastejšie používané bloky kódu (napr. funkcie, cykly, podmienky)	stredná
F08	Rozšírenie funkcionality	Možnosť rozširovať funkcionálnu pomocou zásuvných modulov	stredná
F09	Vyhľadávanie	Určitý druh fulltextového vyhľadávania s prípadnou optimalizáciou pre najčastejšie vyhľadávané výrazy	stredná
F10	Export súborov	Možnosť exportovania súboru do iných formátov (.csv, .doc)	nízka
F11	Podpora SW metrik	Schopnosť detegovať určité ukazovatele v zdrojovom kóde ako index udržateľnosti, cyklomatická zložitosť, hodnoty Fan in a Fan out,	nízka

		ktoré by boli zobrazené v tabuľke, prípadne vizuálne v podobe grafov	
Legenda: Vysoká priorita – nevyhnutia funkcia systému, je základom funkcionality systému. Stredná priorita – funkcia, ktorú možno implementovať neskôr netvorí základ funkcionality systému. Nízka priorita – funkcionality bude implementovaná v ďalších verziách			

3.2 Nefunkcionálne požiadavky

Pre TrollEdit boli identifikované nasledujúce nefunkcionálne požiadavky pre správne zabezpečenie fungovania programu.

Tab. 3 Nefunkcionálne požiadavky

ID	Požiadavka	Charakteristika
N01	Rýchlosť a spoľahlivosť	Zrýchlenie programu hlavne čo sa týka parsovania. Program by mal byť schopný pracovať aj na menej výkonnom hardvéri
N02	Modulárnosť	Možnosť rozširovania jeho funkcií pomocou dodatočnej implementácie nových modulov. Tým pádom nie je v zásade nutné zasahovať do samotnej implementácie systému pri rozširovaní jeho funkcionality
N03	Redesign používateľského rozhrania GUI	Musí byť jednoduché a prehľadné, pričom najčastejšie funkcie systému by mali byť prístupné používateľovi bez náročného hľadania

3.3 Analýza požiadaviek pre paralelizmus

Hlavnou myšlienkou paralelného spracovávania bolo využitie zdrojov a výkonu zariadení na ktorých by mal editor bežať. V dnešnej dobe už väčšina výpočtových zariadení vie pracovať a poskytuje na prácu minimálne dve samostatné procesorové jadrá. Aj myšlienkou nášho editora a jednou z jeho hlavných výhod by mala byť jeho rýchlosť a pokročilé možnosti vizualizácie textu aby sa mohol páčiť používateľovi a teda by mal bez väčšieho meškania fungovať a reagovať na vstup používateľa. Toto sa bohužiaľ nedá dosiahnuť použitím len jedného hlavného vlákna, ktoré by malo na starosti všetky úlohy. Práve preto sme rozhodli zaviesť do editora paralelizmus.

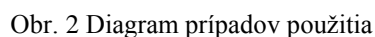
Týmto by sa vyriešila otázka efektívnosti práce editora a teda aj problém rýchlosti práce a výkonnosti editora pri spracovávaní náročnejších operácií. Pri pôvodnom riešení editora bol problém z rýchlosťou behu niektorých operácií akými je napríklad syntaktická analýza. Analýza aktuálneho textu chvíľu trvá a pri väčších textoch nastáva problém s dlhou dobou odozvy a zobrazenia analyzovaných prvkov, kedy používateľ zažíva značné omeškanie vizualizácie blokov.

Spomínaný proces syntaktickej analýzy aktuálneho súboru je celkovo zdĺhavý najmä z dôvodu pomalej copy fázy AST stromu na stranu Qt zo strany Lua, kde prebieha analýza. Toto je už z časti spojené z pôvodným riešením tvorby stromu, ktorý by sa mal zostavovať už na strane Lua jazyka, no nič to nemení na skutočnosti, že analýza aktuálneho súboru by mala byť schovaná a bežať na pozadí aplikácie, kedy ničím neruší používateľa.

Aby bolo využitie paralelizmu čo najefektívnejšie, plánuje sa najmä jeho spolupráca zo skriptovacím jazykom Lua a teda plné využitie ich výhod.

V tejto kapitole je popísaný návrh programu TrollEdit podľa požiadaviek definovaných v predchádzajúcej kapitole. Funkcionálne požiadavky sa premietnu do diagramu prípadov použitia a nefunkcionálne do architektúry systému.

Na diagrame sú znázornené prípady použitia popisujúce funkcionality, ktorá je už implementovaná v programe TrollEdit a taktiež novu funkcionality, ktorú sme identifikovali na základe analýzy. Nové prípady použitia sú odlišené od tých existujúcich červenou farbou.



Tab. 4 Prípád použitia UC13 Undo/redo

Názov	Undo/ redo		
ID	UC13		
Opis	Možnosť voľby undo/ redo nad vykonanými zmenami v zdrojovom kóde	Priorita	vysoká
Vstupne podmienky	História vykonaných zmien		
Výstupne podmienky	-		
Participant	Používateľ (použ.)		
Základná postupnosť	Krok	Rola	Činnosť
	1.	Použ.	vyberie možnosť undo/ redo v pop menu na zdrojovom kódom
	2.	Systém	urobí zmeny v kóde podľa histórie výkonných akcií
Alternatívna postupnosť	Krok	Rola	Činnosť
-			
Poznámky	-		

Tab. 5 Prípád použitia UC14 Prepínanie módov písania

Názov	Prepínanie módov písania		
ID	UC14		
Opis	Prvý mód pre klasický editor na úpravu kódu a po prepnutí by editor prešiel do druhého grafického módu	Priorita	vysoká
Vstupne podmienky	-		
Výstupne podmienky	-		
Participant	používateľ (použ.)		
Základná postupnosť	Krok	Rola	Činnosť
	1.	Použ.	V pop menu si zvolí možnosť prepnutia do druhého módu písania kódu
	2.	Systém	prepne úpravu kódu do grafického módu
	3.	Systém	rozšíri možnosti funkcionality pre grafický mód úpravy kódu
Alternatívna postupnosť	Krok	Rola	Činnosť
-			
Poznámky	-		

Tab. 6 Prípád použitia UC14 Použitie shortcuts

Názov	Použitie shortcuts		
ID	UC15		
Opis		Priorita	vysoká
Vstupne podmienky	-		
Výstupne podmienky	-		
Participant	používateľ (použ.)		
Základná postupnosť	Krok	Rola	Činnosť
	1.		
	2.		

	3.		
Alternatívna postupnosť	Krok	Rola	Činnosť
Poznámky	-		

Tab. 7 Prípad použitia UC16 Vyhľadávanie v kóde

Názov	Vyhľadávanie v kóde		
ID	UC16		
Opis	Vyhľadanie zvoleného výrazu v zdrojovom kóde	Priorita	stredná
Vstupne podmienky	-		
Výstupne podmienky	Zobrazenie výsledku hľadaného výrazu		
Participant	Používateľ (použ.)		
Základná postupnosť	Krok	Rola	Činnosť
	1.	Použ.	zadá hladný výraz do textboxu pre vyhľadávanie a potvrdí tlačidlom hľadať
	2.	System	vyhľadá zvolený výraz v aktuálnom zdrojovom kóde
	3.	System	zobrazí výsledky hľadaného výrazu
Alternatívna postupnosť	Krok	Rola	Činnosť
	3.a	System	v prípade nenájdenia hľadaného výrazu zobrazí modálne okno s upozornením
Poznámky	-		

Tab. 8 Prípad použitia UC17 Export súborov

Názov	Export súborov		
ID	UC17		
Opis	Export súborov zdrojového kódu do iných formátov	Priorita	nízka
Vstupne podmienky	Parsovaný zdrojový kód		
Výstupne podmienky	Vyexportovaný súbor		
Participant	Používateľ (použ.)		
Základná postupnosť	Krok	Rola	Činnosť
	1.	Použ.	vyberie možnosť exportu súborov zo zdrojového kódu
	2.	System	ponúkne možnosti do akých formátov ma exportovať súbory
	3.	Použ.	vyberie formát súboru pre uloženie
	4.	System	uloží súbory vo zvolenom formáte
Alternatívna postupnosť	Krok	Rola	Činnosť
-			
Poznámky	formát pdf, doc.		

Tab. 9 Prípád použitia UC18 Nastavenie programu

Názov	Nastavenie programu		
ID	UC18		
Opis	Podrobne nastavenie možností programu	Priorita	stredná
Vstupne podmienky	-		
Výstupne podmienky	Zmena nastavenia programu		
Participant	Používateľ (použ.)		
Základná postupnosť	Krok	Rola	Činnosť
	1.	Použ.	si zvolí možnosť nastavenia programu z hlavného menu
	2.	Systém	zobrazí modálne okno s možnosťami nastavenia programu
	3.	Použ.	vykoná zmeny v nastaveniach a uloží zmeny
	4.	Systém	uloží vykonane zmeny a reštartuje program
Alternatívna postupnosť	Krok	Rola	Činnosť
	4.a	Systém	v prípade nekorektného nastavenia oznámi používateľa varovaním oknom s popisom chyby
Poznámky	-		

Tab. 10 Prípád použitia UC19 Zobrazenie sw metrík

Názov	Zobrazenie sw metrík		
ID	UC19		
Opis	Zobrazenie sw metrík zdrojového kódu	Priorita	nízka
Vstupne podmienky	Zdrojový kód pre vygenerovanie metrík		
Výstupne podmienky	Zobrazenie výsledkov metrík vo forme grafov		
Participant	Používateľ (použ.)		
Základná postupnosť	Krok	Rola	Činnosť
	1.	Použ.	si zvolí možnosť zobrazit' sw metriky z menu
	2.	Systém	vygeneruje metriky zo zdrojového kódu a zobrazí výsledky vo forme grafov
Alternatívna postupnosť	Krok	Rola	Činnosť
-			
Poznámky	-		

4.2 Architektúra programu

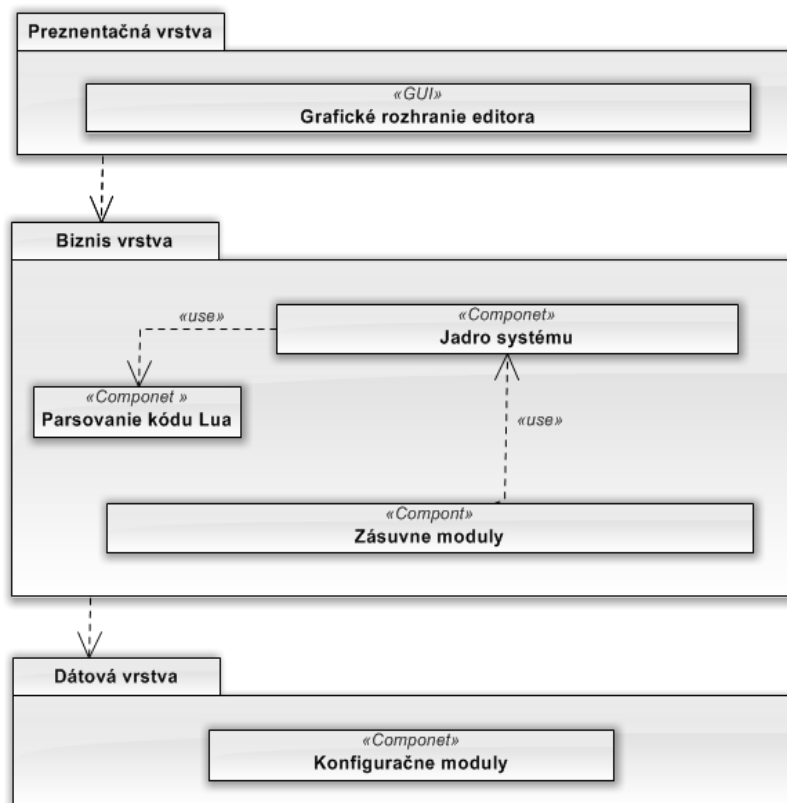
Architektúru programu by sme chceli postaviť na klasickom trojvrstvom princípe, t.j. rozdelená na prezentačnú, biznis a dátovú vrstvu vid. Obr. 3.

V prezentačnej vrstve budú implementované triedy pre grafické rozhranie editora od hlavného menu až po nápovedu. Prezentačná vrstva bude komunikovať s biznis vrstvou, v ktorej bude spracovávaná aplikačná logika programu.

Biznis vrstva sa bude skladať z troch komponentov. Jeden pre jadro systému kde bude implementovaná základná funkcionálna program. Druhý pre parsovanie zdrojového kódu,

kde sa bude vytvárať AST strom v skriptovacom jazyku Lua. A tretí pre pridávanie novej funkcionality, ktorá nenaruší základnú funkcionality jadra programu.

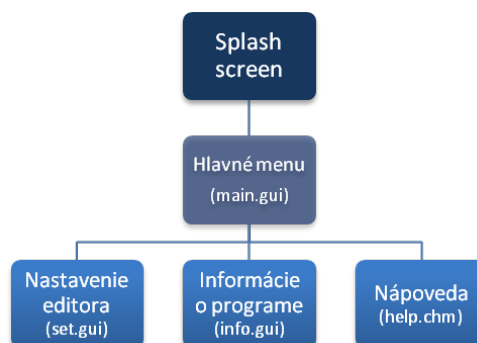
V dátovej vrstve budú dáta ktoré si bude program ukladať ako nastavenie programu dočasnej históriu zmien nad zdrojovým kódom a údaje o projekte.



Obr. 3 Architektúra programu

4.3 Návrh GUI

Na základe analýzy sme identifikovali 5 okien, ktoré budú v programe implementované. Tieto okná budú mať hierarchický význam, t.j. okno na vyššej úrovni môže volať iba okná na nižšej úrovni, nie však opačne.



Obr. 4 Hierarchické rozdelenie okien programu

Hierarchia:

- Splash screen
 - úvodné modálne okno, ktoré sa zobrazí vždy pri spustení TrollEditu
 - odstraňuje problémy studeného štartu
- Hlavne menu
 - nemodálne okno, ktoré slúži ako hlavné menu programu
 - z tohto okna je možné volať iné okna
- Nastavenie editora
 - modálne okno, ktoré slúži pre detailné nastavenie editora
- Informácie o programe
 - modálne okno, ktoré zobrazuje informácie o programe ako popis programu, dátum vytvorenia, verzia programu.
- Nápoveda
 - nemodálne okno pre zobrazenie nápovedy programu



Obr. 5 Splash screen programu



Obr. 6 Predbežný návrh hlavného menu programu

4.4 Návrh funkcionality UNDO/REDO

4.4.1 Qt Undo Framework

Koncepcia

Je to implementácia návrhového vzoru Command. Základnou myšlienkou tohto vzoru je, že každá zmena v aplikácii je vykonávaná pomocou vytvárania „Command“ objektov. Tieto objekty aplikujú zmeny a sú uložené do „Command stack“-u. Vďaka tomu každý Command objekt vie ako vrátiť predchádzajúci stav dokumentu. Pomocou prechádzania stacku a zavolania funkcií undo() a redo() vieme vykonávať akcie UNDO/REDO.

Návrh riešenia

Do triedy MainWindow by sme vytvorili QUndoStack pre uchovávanie Command objektov a QUndoView na sledovanie a interakciu so stackom.

```
QUndoStack *undoStack;  
QUndoView *undoView;
```

Pomocou funkcie createUndoView() vytvoríme QUndoView

```
void MainWindow::createUndoView()  
{  
    undoView = new QUndoView(undoStack);  
    undoView->setWindowTitle(tr("Command List"));  
    undoView->show();  
    undoView->setAttribute(Qt::WA_QuitOnClose, false);  
}
```

Vo funkcií createActions() vytvoríme akcie UNDO/REDO

```
void MainWindow::createActions()  
{  
    undoAction = undoStack->createUndoAction(this, tr("&Undo"));  
    undoAction->setShortcuts(QKeySequence::Undo);  
    redoAction = undoStack->createRedoAction(this, tr("&Redo"));  
    redoAction->setShortcuts(QKeySequence::Redo);  
}
```

Vytvoríme "Command" na zmazanie.

```
class DeleteCommand: public QUndoCommand
{
public:
    DeleteCommand (QGraphicsScene *graphicsScene, QUndoCommand
*parent = 0);
    void undo();
    void redo();
private:
    QGraphicsScene *myGraphicsScene;
};
```

Implementujeme funkcie undo() a redo()

```
void DeleteCommand::undo()
{
    myGraphicsScene->addItem(myDiagramItem);
    myGraphicsScene->update(); }
}
```

```
void DeleteCommand::redo()
{
    myGraphicsScene->removeItem(myDiagramItem); }
}
```

Výhoda: rýchla, prehľadná a škálovateľná implementácia funkcionality UNDO/REDO

Nevýhoda: vytváranie „Command“ objektov pre každú akciu môže byť neefektívne.
potreba zmeny existujúcej implementácie.

4.4.2 QScintilla

Uloží akcie vykonávané na dokumente. Umožňuje kombináciu viacerých akcií do transakcií, na ktorými je potom možné vykonať UNDO/REDO.

Obsahuje rôzne príznaky, ktoré určujú či je možné alebo nie je možné vykonať UNDO/REDO.

Výhoda: poskytuje možnosť riadenia zmien na dokumentoch pomocou základných operácií UNDO/REDO.

QScintilla poskytuje aj ďalšie operácie vhodné pre prácu so zdrojovými súbormi.
(Syntax highlighting, Searching, Replacing, Key bindings, Copy, Paste)

Nevýhoda: preštudovanie novej technológie a neprehľadná implementácia

4.4.3 QTextDocument

QTextDocument je grafický element, ktorý obsahuje formátovaný text.

Poskytuje sloty a funkcie na podporu funkcionality UNDO/REDO.

void	redo ()
void	setModified (bool <i>m</i> = true)
void	undo ()

Atribút modified poskytuje informáciu o tom či bol alebo nebol dokument modifikovaný. Okrem toho obsahuje funkcie aj na zistenie dostupnosti operácií UNDO/REDO.

Výhoda: nepotrebuje nové technológie, malý zásah do existujúceho kódu

Nevýhoda: poskytuje len triviálne riešenie

4.5 Návrh funkcionality pre shortcuts

Spoločným použitím object model a action system v Qt môžeme vytvoriť prispôsobiteľné funkcie v editore akcií, ktoré sa dajú integrovať do už existujúcich aplikácií. Qt action system je založený na triede QAction, ktorá uchováva informácie o všetkých rôznych spôsoboch ako je možné spustiť určitý príkaz v aplikácii.

4.5.1 Dialógové okno

Vytvoríme si dialógové okno, ktoré umožní používateľovi prispôbovať klávesové skratky v aplikácii. V jednom stĺpci budú vypísané akcie a v druhom stĺpci budú vypísané klávesové skratky, ktoré bude môcť používateľ editovať.

Definícia triedy ActionsDialog:

```
class ActionsDialog : public QDialog
{
    Q_OBJECT

public:
    ActionsDialog(QObjectList *actions,
                  QWidget *parent = 0);
protected slots:
    void accept();

private slots:
    void recordAction(int row, int column);
    void validateAction(int row, int column);

private:
    QString oldAccelText;
    QTable *actionsTable;
    QList<QAction*> actionsList;
};
```

Keď sa text klávesovej skratky edituje, tak sa využijú funkcie `recordAction()` a `validateAction()`. Funkcia `accept()` sa využije, keď dialógové okno akceptuje skratku.

Konštruktor plní úlohu vytvorenia používateľského rozhrania. Na minimalizovanie vplyvu na aplikáciu sa trieda vytvorí v `QObjectList` a použijeme `QTable` na zobrazenie informácií.

```
ActionsDialog::ActionsDialog(QObjectList *actions,
                             QWidget *parent)
    : QDialog(parent)
{
    actionsTable = new QTable(actions->count(), 2, this);
    actionsTable->horizontalHeader()->setLabel(0,
        tr("Description"));
    actionsTable->horizontalHeader()->setLabel(1,
        tr("Shortcut"));
    actionsTable->verticalHeader()->hide();
    actionsTable->setLeftMargin(0);
    actionsTable->setColumnReadOnly(0, true);
}
```

V tabuľke je povolené editovať len jeden stĺpec a odstránime z nej aj vertikálnu hlavičku na ľavej strane.

Každá akcia je taktiež pridaná do zoznamu, ktorý nám umožní vyhľadať akciu zodpovedajúcu danému riadku v tabuľke. Toto ešte budeme používať na modifikovanie akcií.

```
QAction *action =
    static_cast<QAction *>(actions->first());
int row = 0;

while (action) {
    actionsTable->setText(row, 0, action->text());
    actionsTable->setText(row, 1,
        QString(action->accel()));
    actionsList.append(action);
    action = static_cast<QAction *>(actions->next());
    ++row;
}
```

- Dialógové okno bude mať dve tlačidlá: OK a Cancel

```
QPushButton *okButton = new QPushButton(tr("&OK"), this);
QPushButton *cancelButton = new QPushButton(tr("&Cancel"), this);
connect(okButton, SIGNAL(clicked()),
    this, SLOT(accept()));
connect(cancelButton, SIGNAL(clicked()),
    this, SLOT(reject()));
```

- Dva signály z tabuľky slúžia na editačný proces

```
connect(actionsTable, SIGNAL(currentChanged(int, int)),
    this, SLOT(recordAction(int, int)));
connect(actionsTable, SIGNAL(valueChanged(int, int)),
    this, SLOT(validateAction(int, int)));
```



```
...
    setCaption(tr("Edit Actions"));
}
```

4.5.2 Proces editovania

Keď používateľ začne upravovať bunku, actionsTable vyšle signál currentChanged() a zavolá sa recordAction() s riadkom a stĺpcom bunky.

```
void ActionsDialog::recordAction(int row, int col)
{
    oldAccelText = actionsTable->item(row, col)->text();
}
```

Predtým, než používateľ dostane šancu modifikovať obsah, uložíme si aktuálny text bunky. Keď nebude vyhovovať zmenený text, tak sa text bunky vráti na pôvodný.

```
void ActionsDialog::validateAction(int row, int column)
{
    QTableWidgetItem *item = actionsTable->item(row, column);
    QString accelText = QString(QKeySequence(
        item->text()));

    if (accelText.isEmpty() && !item->text().isEmpty()) {
        item->setText(oldAccelText);
    } else {
        item->setText(accelText);
    }
}
```

Použijeme QKeySequence na kontrolu nového textu. Ak nový text nemôže byť použiteľný QKeySequence, tak sa do bunky uloží pôvodný text. Keď používateľ z bunky zmaže starý text a nezadá nový, tak bunka zostane prázdna. Keď nový text môže byť použiteľný, tak sa uloží do bunky a nahradí starý text.

```
void ActionsDialog::accept()
{
    for (int row = 0; row < actionsList.size(); ++row) {
        QAction *action = actionsList[row];
        action->setAccel(QKeySequence(
            actionsTable->text(row, 1)));
    }

    QDialog::accept();
}
```

Pri stlačení tlačidla OK sa zmeny uložia a pri stlačení Cancel sa všetky zmeny stratia.

4.5.3 Editovanie, načítanie a ukladanie skratiek

V aplikácii musíme zabezpečiť otvorenie dialógu s používateľského rozhrania, načítanie nastavenia skratiek a ich ukladanie.

```
private slots:
    ...
    void editActions();
    void saveActions();

private:
    void loadActions();
    ...

ApplicationWindow::ApplicationWindow()
    : QMainWindow(0, "example application main window",
                  WDestructiveClose)
{
    ...
    QPopupMenu *settingsMenu = new QPopupMenu(this);
    menuBar()->insertItem(tr("&Settings"), settingsMenu);

    QAction *editActionsAction = new QAction(this);
    editActionsAction->setMenuText(tr(
        "&Edit Actions..."));
    editActionsAction->setText(tr("Edit Actions"));
    connect(editActionsAction, SIGNAL(activated()),
            this, SLOT(editActions()));
    editActionsAction->addTo(settingsMenu);

    QAction *saveActionsAction = new QAction(this);
    saveActionsAction->setMenuText(tr("&Save Actions"));
    saveActionsAction->setText(tr("Save Actions"));
    connect(saveActionsAction, SIGNAL(activated()),
            this, SLOT(saveActions()));
    saveActionsAction->addTo(settingsMenu);
}
```

Na prepísanie predvolených klávesových skratiek vložíme nasledovný kód na koniec konštruktorov:

```
...
    loadActions();
    ...
}
```

Funkcia LoadActions () slúži na zmenu skratky každého QAction, ktorého názov zodpovedá záznamu v nastavení:

```
void ApplicationWindow::loadActions()
{
    QSettings settings;
    settings.setPath("trolltech.com", "Action");
    settings.beginGroup("/Action");
}
```

```
QObjectList *actions = queryList("QAction");
QAction *action =
    static_cast<QAction *>(actions->first());

while (action) {
    QString accelText = settings.readEntry(
        action->text());
    if (!accelText.isEmpty())
        action->setAccel(QKeySequence(accelText));
    action = static_cast<QAction *>(actions->next());
}
```

Táto funkcia závisí na predvolené hodnoty z QSettings:: readEntry (), aby zabezpečili, že každá akcia je zmeniť iba v prípade, že je vhodný vstup s platnou skratku k dispozícii.

editActions () vykonáva úlohu otvoriť dialóg so zoznamom všetkých akcií hlavného okna:

```
void ApplicationWindow::editActions()
{
    ActionsDialog actionsDialog(queryList("QAction"),
                                this);
    actionsDialog.exec();
}
```

QObjectList vrátené QueryList () obsahuje všetky QActions v hlavnom okne, vrátane tých nových, ktoré sme pridali do menu.

saveActions() je volané vždy keď Save Action menu je aktivované:

```
void ApplicationWindow::saveActions()
{
    QSettings settings;
    settings.setPath("trolltech.com", "Action");
    settings.beginGroup("/Action");

    QObjectList *actions = queryList("QAction");
    QAction *action =
        static_cast<QAction *>(actions->first());

    while (action) {
        QString accelText = QString(action->accel());
        settings.writeEntry(action->text(), accelText);
        action = static_cast<QAction *>(actions->next());
    }
}
```

4.6 Návrh spracovania syntaktického stromu

Ako spôsob spracovania syntaktického stromu navrhujeme dokopy tri možné riešenia.

C++ -> LUA C API -> LUA -> AST

Ak by bol zmenený kód, tak sa z C++ cez LUA C API pošle požiadavka na vykonanie analýzy zmeneného kódu na strane Lua, pomocou knižnice Lpeg. Lua sprístupní tabuľky reprezentujúce AST, nad ktorými sa bude ďalej pracovať v C++ kóde.

Výhody :

- Rýchlejšie volanie
- Žiadna dátová redundancia

Nevýhody :

- Problém pri spätnej zmene dát (grafické vykresľovanie)

C++ -> LUA FFI C štruktúra -> AST

Ak sa zmení kód, tak sa pošle požiadavka na vykonanie analýzy pomocou knižnice Lpeg. Potom sa vytvorí Lua skript, v ktorom sa tabuľky vygenerované knižnicou Lpeg transformujú do C štruktúr prepojených smerníkmi pomocou FFI knižnice. Následne by sa do C++ kódu poslal smerník na tieto štruktúry, kde by sa s nimi ďalej pracovalo.

Výhody :

- Čiastočné zrýchlenie
- Zachovaná myšlienka manipulácie s AST

Nevýhody :

- Redundancia dát

C++ -> C štruktúra -> LUA FFI Lpeg (AST)

Lpeg knižnicu prepíšeme pomocou FFI knižnice tak, že nebude reprezentovať AST pomocou Lua tabuliek, ale priamo bude vytvárať AST v C štruktúre. Následne pošle smerník na túto štruktúru do C++ kódu, kde sa s ňou bude ďalej pracovať. Týmto by malo byť možné dosiahnuť niekoľkonásobné zrýchlenie, nakoľko práca s C štruktúrou je oveľa rýchlejšia. To je spôsobené najmä dynamickým určovaním dátového typu Lua tabuľky.

Výhody :

- Veľmi veľké zrýchlenie
- Zachovaná myšlienka manipulácie s AST

Nevýhody :

- Náročná implementácia

Rozhodli sme sa pre prvú variantu, a teda dopytovanie sa na AST cez LUA C API. Druhú variantu sme zamietli kvôli redundancii dát, a teda by nemusela poskytovať želané zrýchlenie

programu. Tretiu variantu sme zamietli kvôli náročnosti implementácie, ale v prípade potreby čo najväčšej optimalizácie výkonu aplikácie budeme nad touto variantou znova uvažovať.

Návrh spracovania AST pomocou prvej varianty bude teda vyžadovať nasledovné úpravy. Funkcia `TreeElement *Analyzer::createTreeFromLuaStack()` by sa zrušila a jej funkcionality by bola nahradená čiastkovými volaniami z tabuľky na strane jazyka LUA. Tieto čiastkové volania by musela implementovať nová trieda, ktorá by plne nahradila triedu `TreeElement`. Takáto zmena reprezentácie AST by znamenala značnú zmenu architektúry celej aplikácie, lebo ostatné triedy sú s triedou `TreeElement` silno previazané. V C++ by sme už viac nevytvárali AST, len by sme sa dopytovali na strane LUA, kde by bol trvalo prístupný a menil sa len pri jeho modifikácií.

4.6.1 Experimentovanie s LUA C API / FFI library

Natívny spôsob mapovania C funkcie do LUA skriptu

Táto funkcia slúži na registrovanie C funkcie pre LUA skript

```
void lua_register (lua_State *L,
                  const char *name,
                  lua_CFunction f);
```

Príklad použitia:

Definícia funkcie, ktorú voláme v LUA skripte

```
static int average(lua_State *L)
{
    int n = lua_gettop(L);           /* get number of arguments */
    double sum = 0;
    for (int i = 1; i <= n; i++)     /* loop through each argument */
    {
        if (!lua_isnumber(L, i))
        {
            lua_pushstring(L, "Incorrect argument to 'average'");
            lua_error(L);
        }
        sum += lua_tonumber(L, i);    /* total the arguments */
    }
    lua_pushnumber(L, sum / n);       /* push the average */
    lua_pushnumber(L, sum);           /* push the sum */
    return 2;                         /* return the number of results */
}
```

Ukážka registrácie funkcie pre LUA skript

```
L = luaL_newstate();                /* initialize Lua */
luaL_openlibs(L);                   /* load Lua base libraries */
lua_register(L, "average", average); /* register our function */
luaL_dofile(L, "C:\\\\TEST\\\\lua_call.lua"); /* run the script */
lua_close(L);                       /* clean up LUA */
```

Ukážka LUA skriptu, ktorý používa C funkciu

```
avg, sum = average(10, 20, 30, 40, 50)
print("The average is ", avg)
print("The sum is ", sum)
```

Natívny spôsob mapovania LUA funkcie do C kódu

Definícia, ktorú voláme v LUA skripte

```
function f (x, y)
    io.write("The table the script received has:\n")
    local xx = 1
    for i = 1, #foo do
        print(i, foo[i])
        xx = xx + foo[i]
    end
    io.write("Returning data back to C\n")
    return x + y + xx
end
```

Ukážka kódu, kde voláme LUA funkciu v C kóde, tiež na stranu LUA posielame data do tabuľky a dva parametre pre funkciu

```
L = luaL_newstate();
luaL_openlibs(L); // Load Lua libraries
int status = luaL_dofile(L, "C:\\\\TEST\\\\script.lua");
lua_newtable(L); // We will pass a table
for (int i = 1; i <= x; i++) {
    lua_pushnumber(L, i); // Push the table index
    lua_pushnumber(L, i*3); // Push the cell value
    //lua_rawset(L, -3); // Stores the pair in the table
    lua_settable(L, -3);
}
lua_setglobal(L, "foo");// By what name is the script going to reference
our table?
lua_getglobal(L, "f");
lua_pushnumber(L, 5);
lua_pushnumber(L, 5);
int result = lua_pcall(L, 2, 1, 0); // Ask Lua to run our little script
if (!lua_isnumber(L, -1)) //Get the returned value at the top of the stack
(index -1)
int sum = lua_tonumber(L, -1);
lua_pop(L, 1); // Take the returned value out of the stack
lua_close(L);
```

Natívna FFI (Foreign Function Interface) knižnica

FFI knižnica umožňuje volanie externých C funkcií a používanie C dátových štruktúr v Lua skripte. FFI knižnica do veľkej miery obmedzuje nutnosť použitia natívnych Lua\C volaní v C kóde. Tento spôsob je jednoduchý, keďže FFI parsuje deklarácie C kódu.

Ukážka jednoduchého použitia C funkcií v Lua skripte:

```
local ffi = require("ffi")
ffi.cdef[[
    int printf(const char *fmt, ...);
```

```
int rename(const char* oldname, const char* newname);  
int MessageBoxA(void *w, const char *txt, const char *cap, int  
type);  
}  
ffi.C.printf("Hello %s!", "world")  
ffi.C.rename("pokus.txt", "success.txt")  
ffi.C.MessageBoxA(nil, "Hello world!", "Test", 0)
```

Tento skript vykoná nasledovné akcie. V prvom riadku načíta FFI knižnicu. Potom nasleduje ffi.cdef blok, v ktorom sú deklarácie C funkcií, ktoré chceme skriptu sprístupniť. Potom nasledujú volania týchto funkcií s už konkrétnymi volaniami. Výstupom týchto volaní bude:

→ Vypíše text "Hello world"

→ Premenuje súbor "pokus.txt" na "success.txt" v aktuálnom adresári

→ Zobrazí okno s textom "Hello world!", názvom "Test" a tlačítkom "OK"

4.7 Návrh riešenia paralelizmu

V rámci návrhu riešenia a následného experimentovania boli preštudované možnosti implementácie paralelizmu, ktoré nám rovno poskytuje Qt framework. V predchádzajúcich verziách Qt možnosti paralelného spracovania neboli veľmi rozsiahle, v podstate celková práca vychádzala z použitia QThread triedy. S nasadením aktuálnej 4.4 verzie Qt frameworku boli značne rozšírené prístupy pre paralelizmus, ktoré už sú vhodne implementované a teda je možné ich použiť pre riešenie pokročilejších problémov. V podstate princípy a práca s vláknami je v Qt podobná prístupu, ktorý je využívaný v rámci jazyka Java.

Pre následnú integráciu boli naštudované viaceré prístupy a triedy poskytujúce vhodné paralelné riešenia, pričom sme celkovo identifikovali dva možné prístupy pre zavedenie paralelného spracovania:

4.7.1 QRunnable prístup

Trieda QRunnable slúži ako základná trieda pre všetky bežiacie objekty. V skutočnosti si ju môžeme predstaviť ako interface, ktorý sa používa pre reprezentovanie úlohy alebo časti kódu, ktorý je potrebné vykonať. Pri tomto spôsobe sú využívané upravené objekty pre potreby behu paralelného vlákna. Teda pre spustenie paralelného vlákna je nutné implementovať funkciu run(), ktorú poskytuje daný interface.

Oproti staršiemu riešeniu cez dedenie triedy QThread zo sebou priniesol viaceré výhody:

- QRunnable nemá žiadnu základnú triedu
- bohužiaľ nevyužíva signáli a sloty
- má jednoduchú konštrukciu, naopak využitie QObject je veľakrát ťažkopádne

- beží na akomkoľvek voľnom vlákne a tým rieši cenu vytvárania nového vlákna
- novinka zavedená v Qt 4.4

Pri použití daného interfacu hovoríme o asynchrónnom prístupe, pri ktorom sú všetky vlákna pre vykonanie kódu spúšťané a uložené v rámci `QThreadPool` štruktúry. `QThreadPool` manažuje a má kontrolu nad vláknom, ktoré po skončení automaticky zmaže.

```
class HelloWorldTask : public QRunnable {
    void run()
    {
        qDebug() << "Hello world from thread" << QThread::currentThread();
    }
}
HelloWorldTask *hello = new HelloWorldTask();
QThreadPool::globalInstance()->start(hello);
```

Je možné ovplyvniť cez flag, čo sa má po zbehnutí úlohy s vláknom stať.

```
QRunnable::setAutoDelete() - to change the auto-deletion flag
```

V súčasnej dobe je to stále jeden z najpoužívanějších spôsobov, pomocou ktorého je implementovaný paralelizmus v súčasných aplikáciách.

4.7.2 QtConcurrent prístup

Tento prístup k paralelizmu je založený na Concurrent frameworku. V podstate vznikol ako odpoveď na riešenie otázky kedy už máme implementovanú funkciu, ale nie je nad ňou postavený žiaden `QThread` alebo `QRunnable` prístup? V tom prípade je `QtConcurrent` ideálne riešenie. V tomto prípade ide rovnako o o asynchrónny prístup, ktorého API je ale už postavené na vyššej úrovni synchronizačných princípov.

Na rozdiel od iných spôsobov sa sústreďuje priamo na spustenie funkcie v samostatnom vlákne. V rámci tohto prístupu je výhodou, že nie je nutné robiť žiaden zásah do už existujúcich funkcií a metód. Priamo za pomoci poskytnutého frameworku vieme paralelne spustiť jednotlivé funkcie.

The `QtConcurrent::run()` cez funkciu `run` sa spúšťa funkcia v samostatnom vlákne.

```
QFuture<void> future = QtConcurrent::run(function_name, arg1, arg2, ...);
```

Ako je vidieť, argumenty je možné posielat' do funkcie v rámci `run()` hneď za menom spúšťanej funkcie. Trieda `QFuture` nám tu poskytuje užitočné funkcie ako *isFinished*, *isRunning*, *isStarted*, *waitForFinished*, alebo *result*.

```
QString result = future.result();
```


Výsledky spusteného vlákna sú sprístupnené cez QFuture API. Trieda QFuture a ďalšia trieda QFutureWatcher sa používajú na monitorovanie stavu funkcií vo vláknach.

Funkcia *QFuture::result()* blokuje spracovanie a čaká na výsledok, avšak pri použití QFutureWatcher ide o neblokujúcu synchronizáciu, kedy je možné požadovať notifikovanie až beh funkcie a teda aj vlákna skončí.

Pre nesynchronný prístup je lepšie nastaviť pri vytváraní vlákna Signal aký bude vyslaný po skončení vlákna a Slot, ktorý registruje a spracuje tento signál.

4.7.3 Zdieľanie zdrojov medzi paralelnými vláknami

V rámci návrhu sme sa zamerali ešte na jednu problematiku, ktorá s paralelizmom priamo súvisí a to je zdieľanie spoločných zdrojov v pamäti. Qt frameworku na riešenie tohto problému poskytuje triedu QSharedMemory. Táto štruktúra poskytuje prostriedky pre vytvorenie priestoru v pamäti, ku ktorému môžu pristupovať všetky vlákna, ktoré majú prístupový kľúč. Zároveň obsahuje aj základné synchronizačné prostriedky pre zabezpečenie súbežného prístupu do takejto pamäte.

4.7.4 Zhodnotenie

Aj keď obidve metódy spĺňajú požiadavky paralelizmu v editore, predsa len sme sa pre potreby zavedenia paralelizmu rozhodli využiť druhý prístup a to z viacerých dôvodov:

- aby sme sa vedeli čo možno najviac priblížiť k tým skutočným častiam a funkciám editora, ktoré majú byť vykonávané paralelne
 - napr. samostatná analýza na strane Lua sa spúšťa v rámci jednej metódy *analyze*, ktorá je ale súčasťou komplexnejšej triedy *Analyzer* pre celkovú prácu s Lua
- keďže staviame na už existujúcom riešení, ktoré ďalej rozširujeme a upravujeme, využitie druhého popísaného znamená fakt, že už existujúce časti nemusíme dodatočne prerábať
- ide o pokročilejšiu paralelnú techniku, a teda by mala byť efektívnejšia a rýchlejšia ako staršie riešenia
- keďže ide o grafický editor, ktorý pre svoju prácu vo veľkom využíva princípy signal a slot, nemal by byť problém asynchrónne spracovávať výsledok vlákna, ktorého signal by mal byť odchytávaný

5 Implementácia prototypu

Táto kapitola popisuje implementáciu systému t.j. prevedenie návrhu do výsledného funkčného kódu.

5.1 Implementácia 2 módov editácie

Pre potrebu implementácie dvoch módov editácie bola vytvorená trieda `TextGroup`, ktorá poskytuje klasické možnosti editácie textu ako iné textové editory. Táto trieda dedí svoje vlastnosti od triedy `QGraphicsTextItem` zlučuje v sebe textový editor a schopnosť grafického zobrazenia v triede `QGraphicsScene`. Pri prepnutí módu sa nahrádza miesto triedy `BlockGroup`. Pri opätovnom prepnutí módu posielajú triede `BlockGroup` editovaný obsah textu a zaniká. V takomto minimalistickom riešení dosahujeme uspokojivé implementovanie funkcionality 2 módov, v ktorom sa zachováva konzistencia editovaného textu a pri tom aj výhody formátovania blokov na grafickej scéne.

```
TextGroup::TextGroup(BlockGroup *block, QGraphicsScene *scene)
    : QGraphicsTextItem(0, scene)
{
    this->block = block;
    this->setPlainText(block->toText());
    this->setPos(block->pos().x(), block->pos().y());
    this->setScale(block->scale());
    this->setFlags(QGraphicsItem::ItemIsSelectable |
QGraphicsItem::ItemIsFocusable | QGraphicsItem::ItemIsMovable);
    this->setTextInteractionFlags(Qt::TextEditorInteraction);
    Block *temp = new Block(new TreeElement("temp"), 0, block);
    QFont *f = new QFont(temp->textItem()->font());
    this->setFont(*f);
}
```

Prepnutie módu nastáva stlačením `alt` a ľavým tlačidlom myšky. Nastáva poslanie aktuálnych parametrov a zobrazenie bloku.

```
void TextGroup::mousePressEvent(QGraphicsSceneMouseEvent *event)
{
    if (event->button() == Qt::LeftButton){
        if ((event->modifiers() & Qt::AltModifier) == Qt::AltModifier)
        {
            block->setContent(this->toPlainText());
            block->setPos(this->pos().x(), this->pos().y());
            this->setVisible(false);
            block->setVisible(true);
            scene->update();
            event->accept();
        }
    }
    QGraphicsTextItem::mousePressEvent(event);
}
```

```
}
```

5.2 Experimentovanie a implementácia funkcionality Undo/Redo

Keďže ide o textový editor pri implementácii funkcionality Undo/Redo sme sa zamerali na prácu s textom. Väčšina akcií vykonaných v editore pracuje priamo s textom a až potom sa rieši vizualizácia jednotlivých blokov. Preto najdôležitejšie je identifikovať všetky miesta, kde sa text zmení a uchovávať stav pred a po vykonaní akcie, ktorá zmení text. Pri implementácii tejto funkcionality je možné využiť návrhový vzor Command.

5.2.1 Experimentovanie

Pri experimentovaní sme sa hlavne zamerali na využitie Qt Undo Frameworku, ktorá umožňuje implementáciu návrhového vzoru Command.

- To triedy MainWindow sme pridali zásobník na uloženie jednotlivých akcií a „view“ na zobrazenie histórie vykonaných akcií

```
class MainWindow : public QMainWindow {  
.  
.  
.  
QUndoStack *undoStack;  
QUndoView *undoView;  
.  
undoStack = new QUndoStack(this);  
.
```

- Potom sme vytvorili súbor commands.cpp, v ktorom sa nachádzajú implementácie jednotlivých akcií. Pre každú akciu sme vytvorili vlastnú triedu.

```
class AddTextCommand : public QUndoCommand  
{  
    Q_OBJECT  
public:  
    AddTextCommand(QString text, TextItem* myTextItem, DocType *docType,  
Arrow* arrow, QString* path);  
    void undo(); // implementácia akcie pre undo  
    void redo(); // implementácia akcie pre redo  
private:  
    QString text;  
    TextItem *myTextItem;  
    DocType *docType;  
    QString *path;  
};
```

- Pri zavolaní funkcie na pridávanie textu sa vytvorí nový objekt triedy AddTextCommand, ktorý vykoná akciu. Po vykonaní akcie sa objekt uloží do zásobníka.

```
void MainWindow::addText()
```

```
{
    if (diagramScene->selectedItems().isEmpty())
        return;

    QUndoCommand *addTextCommand = new addTextCommand(diagramScene);
    undoStack->push(addTextCommand);
}
```

- Funkcionalitu Undo/Redo dosiahneme vložení a výberom jednotlivých objektov zo zásobníka a zavolaním funkcie `undo()` resp. `redo()`

5.2.2 Samotná implementácia

Pri implementácii sa v prvom rade zameriavame na prácu s textom a až potom riešime veci ako sú napr. dokumentačné bloky. Na implementáciu využijeme Qt Undo Framework a vychádzame z výsledkov, ktoré sme získali experimentovaním.

5.3 Experimentovanie a implementácia paralelizmu

V rámci tejto časti sme sa zamerali na dve veci. Prvou je experimentovanie s navrhovanými možnými riešeniami tak ako to býva zvykom pri využívaní nových technológií. Nasleduje popísanie riešenia samotnej implementácie, v rámci ktorej sú rozvinuté otázky, ktoré je vhodné riešiť.

5.3.1 Experimentovanie

- QRunnable implementácia a čítanie zo zdieľanej pamäte vo vlákne

```
class Work : public QRunnable
{
public:
    void run(){
        qDebug() << "Hello from child thread " << QThread::currentThread();
        //zavolanie ďalšej funkcie na vykonávanie napr.
        loadFromSharedMemory()
    }
    void loadFromSharedMemory()
    {
        QSharedMemory sharedMemory("SM_key");
        sharedMemory.attach();
        sharedMemory.lock();
        char *to = (char*)sharedMemory.data();
        qDebug() << to;
        sharedMemory.unlock();
        sharedMemory.detach();
    }
};
```

- QSharedMemory – inicializácia a zapísanie do zdieľanej pamäte

```
// QSharedMemory - inicializácia
QSharedMemory sharedMemory("SM_key");
sharedMemory.create(1024);
```

```
// QSharedMemory - zapísanie do pamäte, treba poskytnúť len meno
void writeToSM(QString &SM_key)
{
    QSharedMemory sharedMemory(SM_key);
    sharedMemory.attach();
    sharedMemory.lock();
    char *to = (char*)sharedMemory.data();
    char *text = "hello world printed from SharedMemory by writeToSM
function in separate thread";
    memcpy(to, text, strlen(text)+1);
    sharedMemory.unlock();
    sharedMemory.detach();
}
```

- Využitie QtConcurrentRun

```
// QtConcurrent::run spustí metódu writeToSM v novom vlákne a pošle meno
QFuture<void> future2 = QtConcurrent::run(writeToSM, (QString) "SM_key");
QFutureWatcher<void> watcher2.setFuture(future2);
```

5.3.2 Samotná implementácia

V prvom rade sme sa mali v rámci paralelizmus zamerať na uľahčenie syntaktickej analýzy textu, ktorá by mala bežať na pozadí aby nerušila používateľa a nebrzdila ho pri práci.

Na základe aktuálneho spôsobu práce v editore, sme sa rozhodli zvlášť vytvárať lokálne *lua_state* v novo spustenom vlákne, ktorým bude po skončení nahradený predchádzajúci už neaktuálny *lua_state*.

Počas implementácie bude potrebné vyriešiť nasledujúce problémy:

- Ako preklopiť novú analýzu na tú s ktorou pracuje editor. Teda vymeniť nový *lua_state* a jeho obsah za ten pracovný, s ktorým sa pracuje zatiaľ čo beží analýza na pozadí.
- Ako signalizovať dobehnutie analýzy
- Čo robiť ak už beží analýza a editor zaznamená ďalšie zmeny pri práci so súborom. Zhodiť vlákno a spustiť odznova alebo vytvoriť ďalšie vlákna? Problém, že analýza by nemusela nikdy dobehnúť pri príliš akčnom používateľovi.
- Ako pristupovať k *lua_state*, teda ako to obaliť na strane editora

5.4 Implementácia spracovania AST pomocou LUA C API

Rozhodli sme sa do prototypu implementovať spracovanie abstraktného syntaktického stromu pomocou LUA C API. Postupovali sme prepisom jednotlivých funkcií, ktoré sme nahradili v triede *TreeElement* za ich dynamickú verziu.

Ukážka funkcie ktorá vracia deti (children) aktuálneho elementu v zásobníka a ukladá ich do listu.

```
QList<TreeElement*> Analyzer::getElementChildrenAST(){
    QList<TreeElement*> children;
    int limit = getCountElementChildrenAST();
    if( limit > 0 ){
        TreeElement* child;
        for(int i = 0; i < limit ; i++ ){
            lua_next(L, -2);                                //!< iterate on child
            QString nodeName;
            nodeName = getChildAST();
            bool paired = false;
            if (pairedTokens.indexOf(nodeName, 0) >= 0)
            {
                paired = true;
            }
            child = new TreeElement(nodeName,
                                    selectableTokens.contains(nodeName),
                                    multiTextTokens.contains(nodeName),
                                    false, paired);
            if (floatingTokens.contains(nodeName))
                child->setFloating(true);
            child->analyzer = this;
            children.append(child);                            //!< add children to list
        }
        lua_pop(L, 1);
        lua_pushnumber(L, 1);
    }
    return children;
}
```

Pomocná rekurzívna funkcia na zistenie jednotlivých detí.

```
QString Analyzer::getChildAST(){
    QString child;
    if(lua_isstring(L, -1)){
        child = QString(lua_tostring(L, -1));
        lua_pop(L, 1);
    }else{
        lua_pushnil(L);
        while(lua_next(L, -2) != 0)
        {
            if(lua_isstring(L, -1)){
                child = getChildAST();
                break;
            }else{
                child = getChildAST();
                lua_pop(L, 2);
            }
        }
        lua_pop(L, 2);
    }
    return child;
}
```

Nahradenie za dynamickú verziu funkcie v triede TreeElement.

```
QList<TreeElement*> TreeElement::getChildren() const
{
    if(DYNAMIC)
        return this->analyzer->getElementChildrenAST();
    else
        return children;
}
```

Podarilo sa nám implementovať väčšinu funkcií v triede TreeElement. Neskôr sa však testovaním prišlo na problém spätného vnárania sa na predchádzajúce elementy, predovšetkým z dôvodu reprezentácie len jedného lua_state stavu.

6 Testovanie

V rámci tejto časti sú popísané spôsoby testovania a navrhnuté jednotlivé akceptačné testy.

6.1 Akceptačné testy pre overenie funkcionality

Slúžia na overenie jednotlivých funkcionalít systému.

Tab. 11 Akceptačný test funkcionality Undo/redo

Názov		Použitie Undo/redo	
Rozhranie		dokument	ID testu 01
Účel testu		Overenie funkcionality Undo/redo	ID UC 13
Vstupne podmienky		Otvorený dokument obsahujúci text (zdrojový kód)	
Výstupné podmienky		Dokument je po editovaní a využití funkcie Undo/redo v pôvodnom stave	
Krok	Akcia	Očakávaná reakcia	Skutočná reakcia
1.	Vymaž časť textu	Z dokumentu je vymazaná časť textu.	
2.	Stlač ctrl + z (undo)	Vymazaný text je obnovený.	
3.	Napiš časť textu	V dokumente pribudol text.	
4.	Stlač ctrl + y (redo)	Nový text je odstránený.	
Úroveň splnenia testu		Musí – Mal by – Mohol by	
Poznámka		-	

Tab. 12 Akceptačný test funkcionality skratiek (Shortcuts)

Názov		Použitie skratiek (Shortcuts)	
Rozhranie		hlavne menu	ID testu 02
Účel testu		Overenie funkcionality skratiek	ID UC 15
Vstupne podmienky		Nakonfigurované skratky	
Výstupné podmienky		Vykonanie príslušnej funkcionality podľa nastavenej skratky	
Krok	Akcia	Očakávaná reakcia	Skutočná reakcia
1.	Dopíš text do dokumentu	V dokumente pribudol text.	
2.	Stlač ctrl + s (save)	Zobrazí voľbu uloženia dokumentu.	
3.	Stlač ctrl + q (quit)	Ukončí sa editácia a program.	
4.	Stlač ctrl + <znak>	Vykoná príslušnú funkcionality.	
Úroveň splnenia testu		Musí – Mal by – Mohol by	
Poznámka		-	

Tab. 13 Akceptačný test funkcionality 2 modov editora

Názov		2 módy editora	
Rozhranie		dokument	ID testu 03
Účel testu		Overenie funkcionality 2 modov editora	ID UC 14
Vstupne podmienky		Otvorený dokument obsahujúci text (zdrojový kód)	
Výstupné podmienky		Prepnutý mód editovania dokumentu	
Krok	Akcia	Očakávaná reakcia	Skutočná reakcia
1.	Označ dokument	Dokument je označený (zvýraznený okraj).	

2.	Stlač alt + ľavé tlačidlo myšky	Nastala zmena módu editovania dokumentu. Editovaný text je bez zmeny.	
4.	Dopíš text do dokumentu	V dokumente pribudol text.	
5.	Stlač alt + ľavé tlačidlo myšky	Editovanie sa preplo na pôvodný mód. V dokumente sú všetky zmeny.	
Úroveň splnenia testu		Musí – Mal by – Mohol by	
Poznámka		-	

7 Zhodnotenie a návrhy do ďalšej fázy riešenia

V projekte sme analyzovali existujúce riešenia podobných systémov a tiež vhodné technológie. Navrhli sme viaceré spôsoby ako implementovať jednotlivé funkcionality pre systém, pre ktoré sme vykonali viacero experimentov. Do prototypu sa nám podarilo implementovať časť funkcionality dynamického spracovania abstraktného syntaktického stromu, funkcionality dvoch módov a natívne undo/redo v druhom móde editovania. Keďže tento projekt vyvíjame iteratívnym spôsobom vývoja, táto fáza projektu slúžila hlavne na dôkladnú analýzu a návrh jednotlivých častí systému. V ďalšej fáze projektu sa preto budeme venovať predovšetkým implementovaniu navrhnutých funkcionalít a dôkladnému testovaniu.

7.1 Návrhy pre optimalizáciu riešenia

7.1.1 Optimalizácia paralelizmu

V ďalšej fáze bude nutné implementovať paralelizmus aj na strane Lua jazyka, keďže ide o skriptovací jazyk, kedy ak by sa pracovalo len s jedným *lua_state* nad ktorým bude spúšťaná analýza a medzitým prídu aj ďalšie aktualizácie, tak budú ďalšie príkazy čakať, kým nespracuje skôr zavolanú analýzu. Je to dané spôsobom práce skriptovacích jazykov, kedy kým neskoční jeden proces, tak nezačne ďalší.

Tento spôsob nám umožní prestať uvažovať o vytvorení nového *lua_state* na strane C++ ale rovno zavolať v Lua analýzu a v rámci nej nechať paralelne spracovať text. Bol by menší problém so zamenením analýzy a obsahu AST stromu avšak bolo by nutné ošetriť spracovanie iných úkonov, ktoré sú na strane Lua.

7.1.2 Optimalizácia spracovania AST

Najoptimálnejším spracovania abstraktného syntaktického stromu je jeho priame spracovanie a namapovanie na C-štruktúru pomocou LUA FFI priamo pri spracovaní knižnicou LPeg. Takéto riešenie však vyžaduje rozsiahli zásah do tejto knižnice pri zachovaní jej funkcionality. Pre aktuálne potreby by malo byť postačujúce dokončenie riešenia s LUA C API.