Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava 4

Textový editor obohatený o grafické prvky

Tímový projekt

Tím: UFOPAK (č. 5.)

Vedúci projektu: Ing. Peter Drahoš

Autori:

Andrej Fogelton
Ondrej Kallo
Peter Ondruška
Martin Palo
Jakub Ukrop

Akademický rok: 2009/2010

Obsah

Kapitola 1

Úvod

Vytvorený dokument slúži pre projekt *Textový editor obohatený o grafické prvky* v rámci predmetu Tvorba softvérového/informačného systému v tíme. Keďže téma projektu stále poskytuje možnosti na ďalší výskum, výsledok nášho snaženia bude veľmi pravdepodobne vyvíjaný aj po uplynutí tohto predmetu. Cieľom vzniknutej dokumentácie je popísať nami poňatý návrh a implementáciu riešenia pre danú oblasť.

Dokument je rozdelený na jednotlivé časti opisujúce špecifikáciu, analýzu a návrh. Kapitola špecifikácie uvádza požiadavky na nami vytváraný editor. Časť analýza poskytuje pohľad na už existujúce aplikácie a rozoberá prostredia, a technológie, pomocou ktorých bude daný projekt vytvorený. V návrhu dokument oboznamuje s našimi predstavami o funkcionalite a realizácii projektu. Kapitola prototypu popisuje implemenované časti prototypu riešenia počas zimného semestra. Kapitola produkt sa zaoberá výsledkom implementácie počas letného semestra. V prílohách sa nachádzajú dokumenty súvisiace s projektom ako rozšírený abstrakt pre IIT.SRC 2010 alebo konfiguračný súbor pre zvýrazňovanie syntaxe, či príklad gramatiky pre XML.

Autorstvo kapitol

Tabuľka 1.1 obsahuje autorov a dátumy vytvorenia jednotlivých častí tohto dokumentu.

Tabuľka 1.1: Autori technickej dokumentácie

Názov	**Oblasť**	**Výtvorené**	**Autor**
1 Úvod	–	3.11.2009	Adamíková
2.1 Existujúce riešenia	Analýza	2.11.2009	Adamíková
2.2 Implementačné technológie	Analýza	2.11.2009	Palo
2.3.1 Lua	Analýza	28.10.2009	Ukrop
2.3.2 Rich Text Format	Analýza	28.10.2009	Ondruška
2.4 Zhrnutie analýzy	Analýza	3.11.2009	Palo, Ukrop
3 Špecifikácia	Špecifikácia	1.11.2009	Ondruška
4.1 Základné východiská	Návrh	3.11.2009	Kallo
4.2 Alternatívy jadra	Návrh	3.11.2009	Kallo
4.3 Práca s blokmi	Návrh	6.12.2009	Fogelton, Palo
4.4 Syntaktický analyzátor	Návrh	30.10.2009	Ukrop
4.5 Modul pre využitie značiek RTF	Návrh	2.11.2009	Ondruška
4.6 Architektúra systému	Návrh	3.11.2009	Kallo
5.1 Editačný komponent	Implementácia	7.12.2009	Kallo
5.2 Modul syntaktického analyzátora	Implementácia	6.12.2009	Ukrop
5.3 Modul pre literate programming	Implementácia	6.12.2009	Ondruška
6.1 Editácia blokov	Návrh	8.5.2010	Ukrop
6.2 Zvýrazňovanie syntaxe	Návrh	9.5.2010	Kallo
6.3 Dokumentačné bloky	Návrh	5.5.2010	Fogelton
6.4 Tlač do PDF	Návrh	5.5.2010	Fogelton
7.1 Editácia blokov	Implementácia	8.5.2010	Ukrop
7.2 Zvýrazňovanie syntaxe	Implementácia	9.5.2010	Kallo
7.3 Dokumentačné bloky	Implementácia	5.5.2010	Fogelton
7.4 Tlač do PDF	Implementácia	6.5.2010	Ondruška
8 Záver	–	3.11.2009	Ukrop
8.1 Testovanie	–	8.5.2010	Palo
8.2 Nápady na ďalšie vylepšenie	–	8.5.2010	Palo
8.3 Nadobudnuté skúsenosti/vedomosti	–	8.5.2010	Palo
Používateľská príručka	–	10.5.2010	Kallo

Kapitola 2

Špecifikácia

V tejto kapitole sa venujeme požiadavkám na vytváraný editor. Kapitola je rozdelená na dve časti. V prvej časti sa zaoberáme jednotlivými funkionálnymi požiadavkami, ktoré uľahčia a zefektívnia prácu používateľa nášho editoru. V druhej časti si predstavíme nefunkcionálne požiadavky na náš editor, ktoré tvoria tiež jeho neoddeliteľnú súčasť.

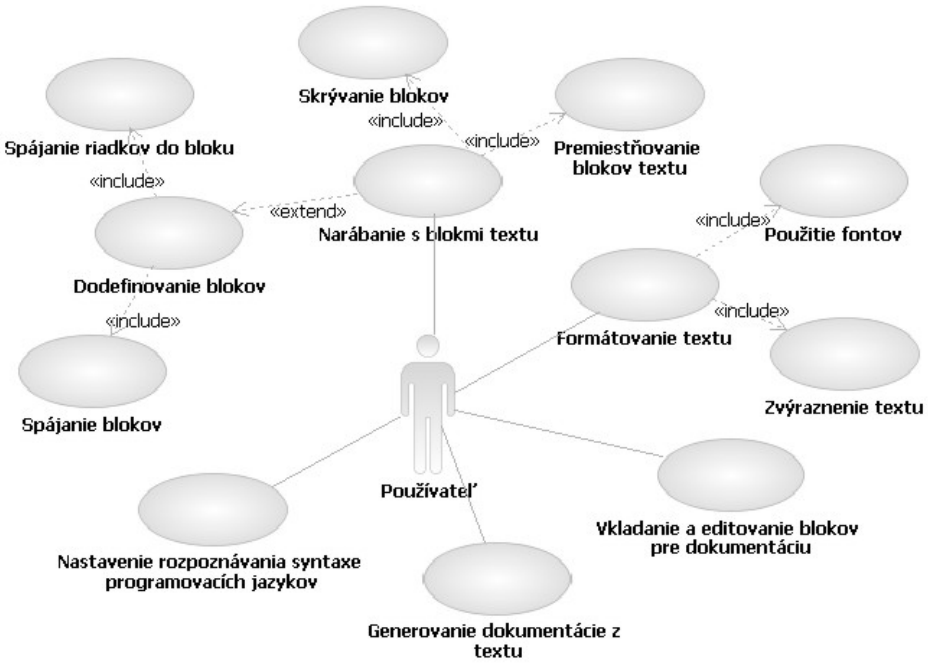
2.1 Funkcionálne požiadavky

Funkcionálne požiadavky vidíme na diagramoch prípadov použití (use-case) uvedených nižšie (Obrázky 2.1, 2.2).

- Rozšírenie zvýraznenia textu základného editora o použitie fontov, farieb a grafických elementov sprehľadní celkovú čitateľnosť kódu a zefektívni prácu programátora pri editácii zdrojových súborov.
- Automatické rozpoznávanie syntaxe daného jazyka počas editácie textu umožní programátorovi ľahko odhaliť prípadné preklepy vzniknuté pri písaní tak, ako sme na to zvyknutí pri dnes používaných vývojových prostrediach.
- Okrem automatického rozpoznávania bežných kľúčových slov programovacích jazykov bude editor rozpoznávať aj celé najčastejšie používané bloky kódu, ako sú napríklad funkcie, cykly, podmienky a graficky ich zvýrazní pomocou farebného bloku, čo

zvýši celkovú logickú čitateľnosť kódu.

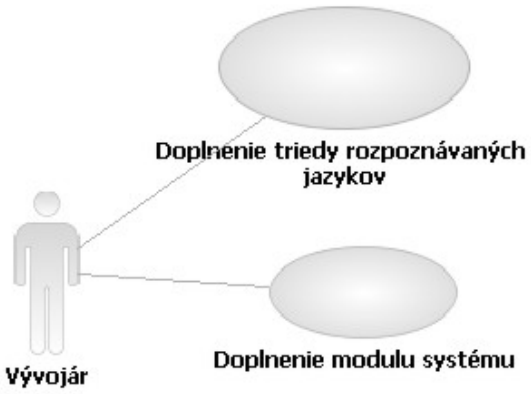
- Takto rozpoznané bloky kódu sa budú dať jednoducho zmenšiť na jeden riadok, čím zvýšime prehľadnosť veľmi dlhých a štruktúrovaných programov.
- Používateľ bude mať možnosť pracovať s celými blokmi kódu, a nielen so samotnými riadkami. Pre jednotlivé bloky môže programátor veľmi jednoducho využiť systém drag-and-drop a presunúť blok na inú pozíciu.
- Používateľ bude môcť vložiť do kódu špeciálne bloky slúžiace na dokumentovanie. V týchto blokoch sa bude písať dokumentácia priamo k zdrojovým kódom, a tá následne vygenerovať do dokumentu v požadovanej forme.



Obr. 2.1: Prípady použitia pre používateľa

2.2 Nefunkcionálne požiadavky

- Parser je nevyhnutný pre lexikálno-syntaktickú analýzu zdrojového kódu. Bude implementovaný ako skript a jeho úlohou bude rozpoznávať nielen kľúčové slová, ale aj



Obr. 2.2: Prípady použitia pre vývojára

celé funkčné bloky.

- Modulárnosť — systém musí byť dostatočne modulárny, aby umožňoval jednoduché doplnenie triedy rozpoznávaných jazykov o nový jazyk. Zavádzanie nových jazykov bude tvorené použitím skriptov jazyka Lua.
- Multi-platformovosť je základnou požiadavkou nášho editora. Pre jej zaistenie sa celý editor bude implementovať pomocou Qt toolkitu.
- Výstupom editora bude zdrojový kód obohatený o dokumentáciu. Aby bolo možné zdrojový kód skompilovať, všetka dokumentácia bude automaticky umiestnená v komentároch príslušného jazyka.

Kapitola 3

Analýza

V tejto kapitole najskôr opíšeme existujúce editory zdrojových kódov, potom bližšie preskúmame technológie, ktoré plánujeme využiť pri návrhu a implementácii požiadaviek.

3.1 Existujúce riešenia

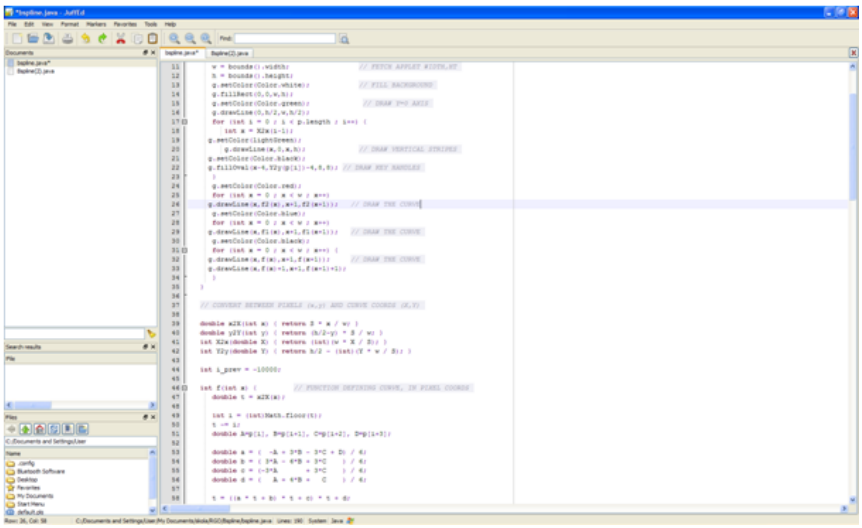
Táto časť je určená krátkemu a stručnému prehľadu niekoľkých editorov. Stručne popisuje ich základné vlastnosti. Menší prieskum vznikol na základe výberu témy pre tímový projekt ako pohľad na niečo už existujúce, no stále ešte doplniteľné. Medzi editormi sa nájdu jednoduchšie, s použitím len základných funkcií, ktoré sú kladené ako hlavné požiadavky potom sa nájdu aj projekty, ktoré sú väčšieho rozmeru a ponúkajú tak rozličné množstvo vlastností a možnej nadstavby.

JuffEd

Práca s JuffEd editorom je intuitívna. Obsahuje zvýrazňovanie syntaxe pre viac ako dvadsať populárnych programovacích jazykov, podporuje vyhľadávanie a nahradzovanie pomocou regulárnych výrazov, výber blokov, doplňovanie kódu, označovanie riadkov, viac pohľadov na dokument a rôzne znakové sady.

JuffEd je možné rozšírovať zásuvnými modulmi. Moduly môžu pridávať body a hlavné

kontextové menu, panely nástrojov. K jeho výhodám patrí aj voľná dostupnosť na trhu a viacplatformovosť, takže nezáleží na tom či používateľ má Windows, Linux, FreeBSD. Je alternatívou k takým editorom ako PsPad či Notepad++. JuffEd používa Qt4 knižnicu a edtičanú komponentu Scintilla.



Obr. 3.1: JuffEd

qPEditor

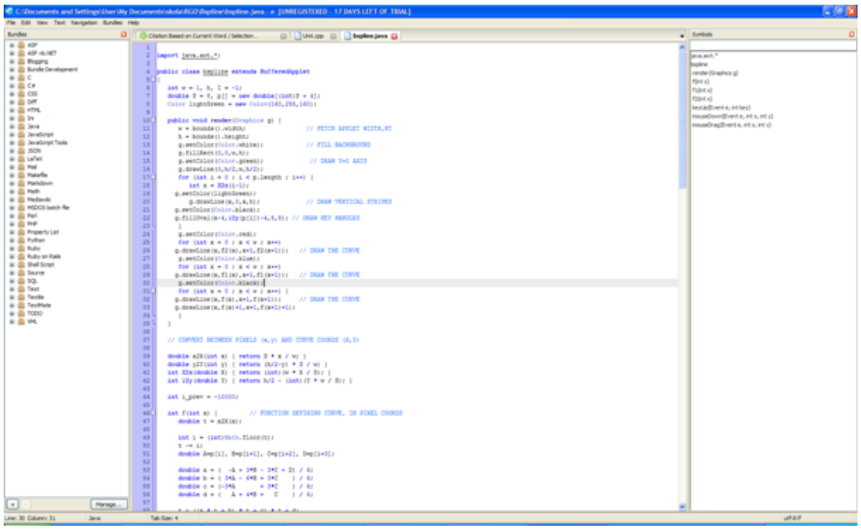
Editor pre programátorov. Má dostupný kód pod GPL licenciou. Je viacplatformový, napísaný je úplne celý v Qt, takže minimálnou požiadavkou je Qt 4.2.x. Úprava štýlov alokácie pre rôzne programovacie jazyky, číslovanie riadkov, podpora undo/redo, má panel záložiek. Použitie v Qt Designer-i ako zásuvný modul (plug-in).

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 trošku pomalé otváranie súborov.



Obr. 3.2: eTextEditor

SciTE

Editor založený na Scintille. V SciTE [8] 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 klá-

vesy 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. Šikovná je aj funkcia pre prechod medzi časťami slov, na rozdiel od Ctrl+šípky zohľadňuje aj podčiarovník a zmeny veľkosti písmen v slove či odseku (bloky textu oddelené prázdnym riadkom).

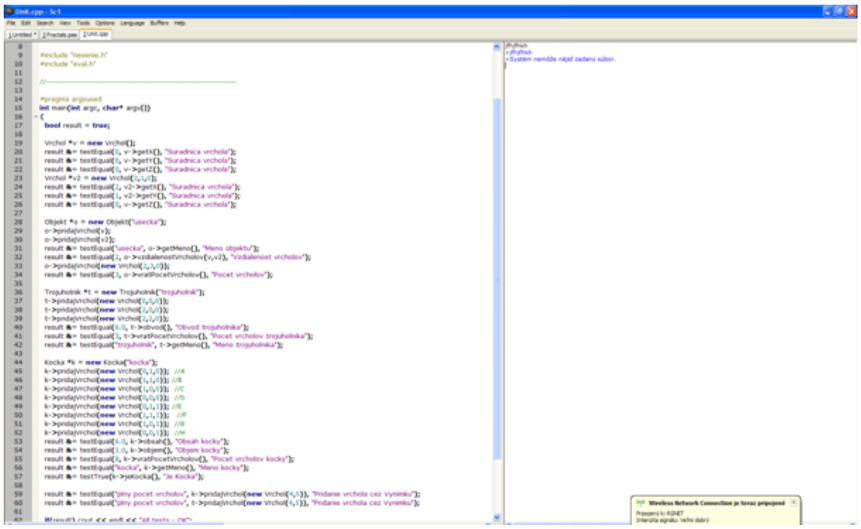
Kompilácia Skontrolovanie syntaxe a prenesenie na riadok, kde sa daná chyba nachádza.

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

Makrá Funkčnosti 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.

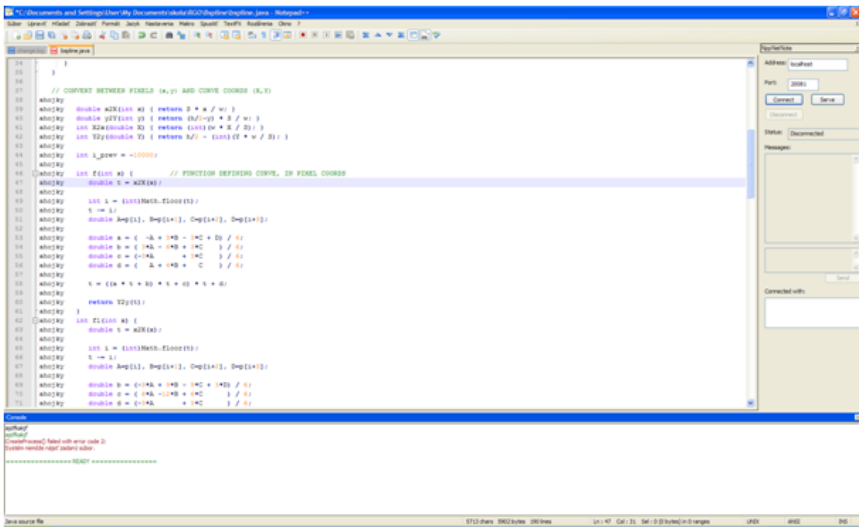


Obr. 3.3: SciTE

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.



Obr. 3.4: Notepad++

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: oddeľovač 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, kórejčinu 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 nájdenia bloku.

Prehľad funkcií editorov SciTE a Notepad++ vznikol na základe väčšieho záujmu o tieto editory z dôvodu ich všeobecnej používateľnosti a oblasti záujmu pre tímový projekt.

3.2 Implementačné technológie

V tejto časti si predstavíme dva implementačné nástroje, ktoré prichádzajú do úvahy pri vývoji editora. Sú nimi nástroj *Qt toolkit 4.0* a *Scintilla*, respektíve *QScintilla*. Najskôr predstavíme jednotlivé technológie, pričom sa zameriame najmä na funkcie, techniky a metódy, ktoré by sme mohli využiť pri implementácii. Pri tejto analýze sa budeme venovať najmä nástrojom pre spracovanie textu a prácu s grafikou, keďže tieto oblasti budú kľúčové pri vývoji aplikácie.

Predbežne by sme chceli použiť nástroj Qt toolkit, ktorý by v prípade nedostatočnej funkcionality interných tried pre prácu s textom mohol byť doplnený o funkcie nástroja Scintilla, pomocou portu QScintilla. Podľa získaných poznatkov z analýzy teda určíme, či použijeme pre vývoj editora len nástroj Qt alebo obidva nástroje skombinujeme.

3.2.1 Qt toolkit

Qt toolkit [6] 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 (MP3 prehrávače), bežiace pod operačnými systémami Windows, Linux, MAC OS, Symbian. Multiplatformovosť je práve jedna z rozhodujúcich výhod, kvôli ktorým chceme implementovať editor pomocou tohto nástroja. Qt toolkit je v súčasnosti dostupný pod týmito licenciami:

- Komerčná
- Qt GNU LGPL v 2.1
- Qt GNU GPL v 3.0

Pre potreby školského projektu nám postačuje licencia GNU GPL v 3.0 (General Public License). Táto licencia umožňuje tvoriť aplikácie s otvoreným kódom (open-source), mimo komerčného využitia.

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

- Qt modul pre vývojové prostredie Eclipse
- Qt modul pre vývojové prostredie Visual Studio
- knižnice Qt a zdrojové kódy v nejakom inom vývojovom prostredí
- integrované vývojové prostredie Qt Creator

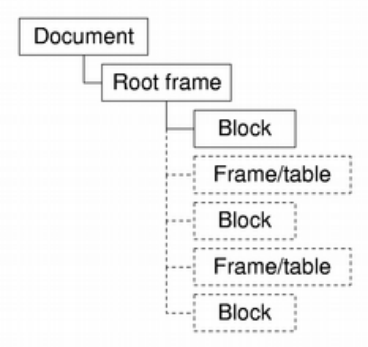
Rozhodli sme sa pre použitie prostredia Qt Creator.

Práca s obohateným textom

Qt toolkit poskytuje množstvo užitočných tried pre prácu s textom. Na oficiálnej stránke Qt [6] je práca s obohateným textom (rich text processing) uvedená ako kľúčová technológia. Týmto je daný predpoklad, že spomenutá funkcionalita bude v Qt bohato zastúpená. Qt obsahuje triedy pre čítanie a manipuláciu s dokumentami so štruktúrovaným obohateným textom. Údaje v rámci dokumentu môžu byť sprístupnené pomocou dvoch rozhraní:

- **Cursor-based interface** používa sa pre upravovanie textu, štruktúra dokumentu je pritom zachovaná.
- **Read-only hierarchical interface** poskytuje celkový pohľad na štruktúru dokumentu a umožňuje operácie nad textom spojené s vyhľadávaním, ukladaním alebo tlačením dokumentu.

Štruktúra dokumentu s obohateným textom poskytuje pohľad na to, aké elementy spoluvytvárajú samotný dokument a akým spôsobom sú tieto elementy usporiadané v dokumente. Základný model štruktúry dokumentu je načrtnutý na Obrázku 3.5.



Obr. 3.5: Základná štruktúra dokumentu s obohateným textom

Každý dokument má základný element `Root frame`, ktorý obsahuje najmenej jeden textový blok. Elementy `Frame` (rámy) a `Table` (tabuľky) sú vždy oddelené elementom `Block` (blok textu), aj keď ten nemusí obsahovať žiadne informácie. To umožňuje vkladanie nových elementov medzi už existujúce elementy.

Rozsiahle nástroje pre prácu s textom využiteľné v editore poskytujú najmä nasledovné triedy:

`QTextEdit` poskytuje funkcie pre zobrazovanie a editáciu obyčajného textu (plain text) a obohateného textu (rich text). Samotný text môže byť vkladaný použitím triedy `QTextCursor` alebo použitím funkcií triedy `QTextEdit` (napríklad `insertHtml()`, `insertPlainText()`, `append()` alebo `paste()`). Pomocou triedy `QTextCursor` je možné napríklad vkladať objekty ako tabuľky, zoznamy alebo obrázky do dokumentu, vyhľadávať a označovať text.

`QTextDocument` predstavuje kontajner pre dokumenty s formátovaným obohateným textom. Je to základná trieda pre prácu s obohateným textom.

`QSyntaxHighlighter` umožňuje definovať pravidlá pre zvýrazňovanie textu. Zvýrazňovanie textu je užitočné najmä pri práci so zdrojovými kódmi, čo je aj prípad nášho editora. Zlepšuje čitateľnosť kódu a pomáha jednoduchšie odhaliť syntaktické chyby v kóde.

Práca s grafikou

Pri implementácii editora bude dôležité vyriešiť problémy ohľadom vykresľovania grafických elementov v aplikácii. Keďže cieľom je vytvoriť editor obohatený o grafické prvky, bude táto oblasť jedna z najdôležitejších. Riešených bude hneď niekoľko problémov, ako napríklad reprezentovanie blokov kódu pomocou grafických elementov (napr. obdĺžniky), presúvanie celých blokov (drag-and-drop) alebo skrývanie blokov (folding). Qt toolkit poskytuje pre vývojárov niekoľko tried pre prácu s grafikou. Teraz stručne predstavíme tie, ktoré by sme mohli využiť.

Základnými triedami v Qt toolkit pre vykresľovanie geometrických útvarov a prácu s nimi sú `QPainter`, `QPaintDevice`, `QPaintEngine`, `QGraphicsScene`, `QGraphicsView` a `QGraphicsTextItem`.

`QPainter` poskytuje optimalizované funkcie pre vykresľovanie v GUI aplikáciach od kreslenia čiar, až po komplexné tvary ako diagramy a grafy.

`QPaintDevice` predstavuje podklad, na ktorý je možné pomocou triedy `QPainter` kresliť. Takýmto podkladom môže byť napríklad inštancia triedy `Widget` (základný grafický element každej GUI aplikácie v QT – okno) alebo `QImage` (reprezentuje obrázok).

`QPaintEngine` je abstraktná trieda, ktorá poskytuje definíciu toho, ako môžeme pomocou triedy `QPainter` kresliť na určité zariadenie na určitej platforme. Qt 4.0 poskytuje niekoľko implementácií tejto triedy pre OpenGL či PostScript. Programátor si však môže vytvoriť aj svoju vlastnú implementáciu, napríklad pre PDF.

`QGraphicsScene, QGraphicsView a QGraphicsTextItem` Tieto tri triedy poskytujú funkcionalitu, ktorá by mohla byť využiteľná pre grafickú reprezentáciu blokov textu. Trieda `QGraphicsScene` poskytuje možnosti pre riadenie a dohľad nad veľkým množstvom 2D grafických prkov. Slúži ako scéna pre grafické prvky. Tie môžu byť vytvorené, buď volaním jednej z metód tejto triedy (`addEllipse()`, `addLine()`, ...), alebo pridaním inštancie triedy `QGraphicsItem` do scény. Spolu s triedou `QGraphicsView`

sa `QGraphicsScene` používa napríklad aj pre vizualizáciu a riadenie textových prvkov.

V `QGraphicsScene` môžeme určovať napríklad pozíciu jednotlivých prvkov, ich viditeľnosť, priehľadnosť, poradie na pozadí/popredí a ošetrovať udalosti (event handling). Hlavná sila tejto triedy spočíva v tom, že dokáže veľmi rýchlo lokalizovať objekt aj na scéne s obrovským množstvom objektov (rádovo milióny). Táto trieda však nemá žiadne vizualizačné schopnosti.

Pre vizualizáciu objektov slúži trieda `QGraphicsView`. Tá zobrazuje grafické objekty v rolovacom okne a jej funkcionalita spočíva napríklad aj v selektovaní, priblížení objektu alebo v rolovaní scény. Inštancia triedy `QGraphicsTextItem` predstavuje textový formátovateľný objekt, ktorý je možné pridať do scény (na inštanciu `QGraphicsScene`).

Drag–and–drop

Technológia drag–and–drop (potiahni a pusť) bude použitá napríklad pri premiestňovaní blokov kódu v editore, ale aj na presun textu medzi oknami, respektíve medzi editorom a inými aplikáciami. Funkcie pre túto technológiu poskytuje už základná trieda všetkých aplikácií `QAplication` (nastavenie času držania tlačidla myši, po ktorom sa inicializuje ťahanie, atď.), avšak primárnymi nástrojmi pre túto technológiu v Qt sú trieda `QDrag` a funkcie triedy `QWidget`.

3.2.2 Scintilla

Scintilla [8] je nástroj, založený na jazyku C++, vyvinutý Neilom Hodgsonom. Je určená pre vývoj aplikácií, ktoré sa zaoberajú editáciou textu, a preto je vhodná aj pre implementáciu editora zdrojových kódov.

V rámci práce so zdrojovými kódmi poskytuje Scintilla možnosti pre editáciu a odstraňovanie chýb (debugging) kódu. V tom je zahrnutá podpora pre formátovanie syntaxe, indikácia chýb, dopĺňanie kódu a automatického pomocníka (editor zobrazí tipy na volanie

funkcie). Okrem toho umožňuje použiť značky pre označenie aktuálneho riadku alebo bod prerušenia (breakpoint). Umožňuje tiež zvýraznenie textu pomocou tučného písma, kurzívy, nastavenia pozadia alebo fontu písma, čo u väčšiny editorov zdrojových kódov nebýva možné.

Nevýhodou tohto nástroja je, že na rozdiel od Qt nepodporuje v takej miere multiplatformovosť a nasadenie aplikácií vytvorených pomocou nej na rôznych platformách si vyžaduje viac práce vývojára. V náš prospech však hrá fakt, že v súčasnosti existuje vstupná brána (port) Scintilly pre Qt — QScintilla (popísaná v ďalšej časti), takže je možné použiť podstatnú časť jej funkcionality v aplikáciách Qt.

Pre demonštráciu možností nástroja Scintilla bol vytvorený editor zdrojových kódov SciTE. Funkcionalita tohto editora je bližšie popísaná pri analýze existujúcich editorov v časti 3.1.

3.2.3 QScintilla

Nástroj QScintilla [5] poskytuje široké možnosti v oblasti práce s textom, najmä so zdrojovými kódmi. Chceme zanalyzovať jej možnosti a zistiť, či by nebolo vhodné skĺbiť funkcie nástrojov QScintilla a Qt v našom editore.

QScintilla ponúka pre vývojárov rovnaké typy licencií ako Qt, je teda prístupná aj pod licenciou GNU GPL a vo verzii 2 je kompatibilná s Qt v3 aj Qt v4.

QScintilla obsahuje viac ako 30 tried — tzv. lexerov, ktoré umožňujú lexikálnu analýzu jednotlivých programovacích jazykov. Pre každý jazyk je vytvorená samostatná trieda, napríklad `QsciLexerJava`. QScintilla teda značne uľahčuje vývoj editorov, ktoré podporujú lexikálnu analýzu veľkého množstva jazykov.

Okrem lexerov poskytuje QScintilla aj niektoré triedy implementujúce prostriedky pre tvorbu GUI a komplexnejšiu prácu s textom (napríklad triedy `QSciDocument` — reprezentácia dokumentu, `QsciScintilla` – API pre Scintillu na báze Qt, `QsciPrinter` — podtrieda Qt triedy `QPrinter` pre tlač dokumentov). Tie majú však menšiu funkcionalitu ako paralelné nástroje v Qt.

3.3 Doplnkové technológie

3.3.1 Lua

Lua [9] 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 funkcionalitu a (meta)mechanizmy ako dodefinovať 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 [2]. 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 860K aj s dokumentáciou), vďaka ktorej nie je problém pripojiť ju celú k aplikácii, ktorá to potrebuje.

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 Wireshark alebo VLC media player.

LPeg

LPeg [3] 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*), formalizme podobnom bezkontextovým gramatikám. Na roz-

diel od bežných gramatík, PEG nedefinuje jazyk, ale algoritmus na jeho rozpoznanie.

LPeg poskytuje dva moduly s rozličným spôsobom práce. Stručne popíšeme rozdiely medzi nimi a uvedieme ukážky syntaxe. Kompletnú syntax uvádza domovská stránka [1]. V prvom module `re` (skratka z *regex*) sú vzory popisované reťazcami so syntaxou odvodenou z regulárnych výrazov. Ako príklad uveďme funkciu `match(text, pattern)`, ktorá sa pokúša nájsť vzor v prefixe reťazca a vráti index prvého nezhodujúceho sa znaku[1].

```
print(re.match("more words", "[a-z]+"))
--> 5, hľadáme 1 a viac znakov od ‘a‘ po ‘z’
```

Vzory môžeme opísať aj gramatikou, nasledujúci kód rozoberá aritmetický výraz:

```
p = [[
  expression <- <factor> ([+-] <factor>)*
  factor     <- <term> ([*/] <term>)*
  term       <- <number> / "(" <expression> ")"
  numer      <- [0-9]+
]]
print(re.match("13+(22-15)", p))   --> 11
```

Druhý modul `lpeg` pracuje so vzormi ako s premennými vlastného dátového typu a obsahuje viac spôsobov na ich vytváranie a spájanie. Kód pre spracovanie aritmetického výrazu bude teraz nasledovný:

```
  p = lpeg.P{ "expression",
  expression = lpeg.V("factor") * (lpeg.S("+-") * lpeg.V("factor"))^0,
  factor     = lpeg.V("term") * (lpeg.S("*/") * lpeg.V("term"))^0,
  term       = lpeg.V("number") + "(" * lpeg.V("expression") * ")",
  number     = lpeg.R("09")^1
}
print(lpeg.match(p, "13+(22-15)")) --> 11
```

[1] Lua indexuje od 1.

Neterminálne symboly, rozsahy a množiny znakov sú definované pomocou samostatných funkcií. Tento zápis je pre jednoduché gramatiky príliš komplikovaný, ale dovoľuje znovupoužitie už definovaných vzorov, rovnako ako využitie všetkej funkcionality jazyka Lua.

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 [3].

3.3.2 Rich Text Format

Rich Text Format(RTF) je metóda slúžiaca na zakódovanie formátovaného textu a obrázkov v textovom dokumente. RTF bolo vyvinuté pre prenášanie dokumentov medzi rôznymi platformami bez straty formátovania.

Syntax RTF

Každý RTF súbor obsahuje neformátovaný text, riadiace slová, riadiace symboly a grupy. Pre zjednodušenie prenositeľnosti štandardný RTF dokument obsahuje 7-bitové znaky. *Riadiace slovo* je špeciálne formátovaný príkaz, ktorý sa používa na označenie riadiaceho kódu a informácií používaných pri manažovaní zobrazenia dokumentov. Riadiace slovo má maximálnu dĺžku 32 znakov a jeho forma je:

```
\LetterSequence<Delimiter>
```

Každé riadiace slovo začína spätným lomítkom (backslash). Nasleduje postupnosť písmen (LetterSequence) tvorených malými písmenami v rozsahu „a“ až „z“ vrátane. RTF je citlivý na veľkosť písmen a každé riadiace slovo musí byť tvorené malými písmenami. Nakoniec nasleduje oddeľovač (Delimiter), ktorý označuje koniec riadiaceho slova. Môže ho tvoriť:

- medzera — v tomto prípade je medzera súčasťou riadiaceho slova
- číslica alebo pomlčka — ktoré indikujú následnosť číselného parametru.

Následná postupnosť čísel je oddelená medzerou alebo ľubovolným znakom rôznym od písmena alebo číslice. Parameter môže byť pozitívne alebo negatívne číslo v rozsahu -32767 až 32767. Ak nasleduje číselný parameter bezprostredne za riadiacim slovom, stáva sa jeho súčasťou. Riadiace slovo je potom oddelené medzerou alebo nepísmenovým či nečíselným znakom rovnakým spôsobom, ako je to u ostatných riadiacich slov.

Pokiaľ je medzera oddeľovačom riadiaceho slova, neobjaví sa v dokumente. Každý nasledujúci znak za oddeľovačom vrátane medzier sa v dokumente objaví. Z tohto dôvodu môžeme používať medzery len tam, kde to je naozaj potrebné, v žiadnom prípade nie na rozdeľovanie RFT kódu.

Riadiaci symbol pozostáva zo spätného lomítka a jedného nepísmenkového znaku. Napríklad `\~` reprezentuje nedeliteľnú medzeru. Riadiace symboly neobsahujú žiaden oddeľovač.

Grupy pozostávajú z textu a riadiacich slov alebo symbolov uzatvorených v zátvorkách (`{}`). Otváracia zátvorka udáva začiatok grupy a uzatváracia zátvorka udáva koniec grupy. Každá grupa špecifikuje text ňou ovplyvnený s rôznymi atribútmi textu. RTF súbor môže obsahovať grupy pre fonty, štýly, farbu, komentáre a podobne.

Vlastnosti niektorých riadiacich slov (tučné, kurzíva, ...) majú len dva stavy. Pokiaľ má riadiace slovo nenulový parameter, rozpoznáva sa, že dané riadiace slovo vlastnosť zapína. Keď má riadiace slovo parameter `0`, rozpoznáva sa, že riadiace slovo vypína danú vlastnosť. Napríklad `\b` zapína tučné písmo a `\b0` ho vypína.

Riadiace slová, symboly a zátvorky tvoria riadiace informácie. Všetky ostatné znaky súboru sú chápané ako čistý text. V nasledujúcom príklade je príklad čistého textu, ktorý sa nenachádza v grupe:

```
{\rtf\ansi\deff0
  {\fonttbl
    {\f0\froman Tms Rmn;}
    {\f1\fdecor Symbol;}
    {\f2\fswiss Helv;}
```

```
  }
  {\colortbl;\red0\green0\blue0;\red0\green0\blue255;
    \red0\green255\blue255;\red0\green255\blue0;
    \red255\green0\blue255;\red255\green0\blue0;
    \red255\green255\blue0;\red255\green255\blue255;}
  {\stylesheet
    {\fs20 \snext0Normal;}
  }
  {\info
    {\author John Doe}
    {\creatim\yr1990\mo7\dy30\hr10\min48}
    {\version1}{\edmins0}
    {\nofpages1}{\nofwords0}{\nofchars0}{\vern8351}
  }
  \widoctrl\ftnbj
  \sectd\linex0\endnhere \pard\plain
  \fs20 This is plain text.\par
}
```

Fráza „This is the plain text.“ nie je súčasťou grupy a chápe sa ako text dokumentu. Ako sme už skôr spomenuli, spätné lomítko a zátvorky majú pre RTF špeciálny význam. Ak ich chceme použiť v texte, musíme pred ne napísať spätné lomítko.

Komentáre v RTF

Komentáre v RTF sa skladajú z dvoch častí:

- ID autora — uvádza sa riadiacim slovom `\atnid`
- Text komentáru — uvádza sa riadiacim slovom `\annotation`

Presnú syntax komentáru si môžeme pozrieť v špecifikácii [7]. Spôsob použitia komentáru si môžeme pozrieť v nasledujúcom príklade:

```
{\insrsid8729657 An example of a paradigm might be Darwinian biology.}
{\cs15\v\fs16\insrsid8729657
  {\*\atnid JD}
  {\*\atnauthor John Doe}\chatn
  {\*\annotation
    {\*\atndate 1180187342}
    \pard\plain \s16\ql \li0\ri0\widctlpar\aspalpha\aspnum\faauto
    \adjustright\rin0\lin0\itap0 \fs20\lang1033\langfe1033\cgrid
    \langnp1033\langfenp1033
    {\cs15\fs16\insrsid8729657 \chatn }
    {\insrsid9244585 How about some examples?}
  }
}
```

V komentári sa môže nachádzať aj časová pečiatka (označená riadiacim slovom `\atntime`), dátumová pečiatka (`\atndate`) alebo ikona (`\atnicn`). Tieto údaje sú však voliteľné a komentár ich nemusí obsahovať.

3.4 Zhrnutie

Zistili sme, že existujúce editory síce disponujú veľkým množstvom rozličnej funkcionality, ale nevyužívajú, okrem farebného zvýraznenia textu, žiadne grafické prvky. Napriek tomu, že mnohé z nich sú rozšíriteľné pomocou zásuvných modulov, sme sa vzhľadom na povahu nášho riešenia rozhodli nevyužiť tieto možnosti.

Na základe bližšej analýzy nástrojov, ktoré prichádzajú do úvahy pre vývoj editora sme dospeli k niekoľkým záverom. V prvom rade sme sa rozhodli, že ako vývojový nástroj budeme používať Qt toolkit, pričom sa budeme snažiť využiť natívne triedy tohto nástroja v čo najväčšej možnej miere. V zálohe tiež budeme mať použitie nástroja QScintilla, najmä v oblasti lexikálnej analýzy a formátovania textu. Samotný nástroj Scintilla pri vývoji

nebudeme využívať, vzhľadom na zníženú multiplatformovosť.

Jazyk Lua v spojení s knižnicou LPeg spĺňa naše požiadavky na rýchlosť aj vyjadrovaciu silu (gramatiky) a umožní nám efektívne vytvoriť syntaktický analyzátor. Na ukladanie dokumentácie a jej formátovania do zdrojového kódu využijeme existujúce značky formátu RTF, ktoré postačujú naším potrebám.

Kapitola 4

Návrh

V tejto kapitole predstavujeme návrh riešenia vypracovaný na základe špecifikácie požiadaviek. Pri návrhu riešenia sme sa sústredili hlavne na návrh alternatív pre reprezentáciu blokov, návrh syntaktického analyzátora a návrh modulu pre využitie značiek Rich Text Format (RTF). Jednotlivé prvky návrhu sú rozdelené do samostatných podkapitol.

Keďže projekt, ktorý riešime, má experimentálny charakter a v praxi niečo podobné ešte neexistuje, našou úlohou je preskúmať možnosti vytvorenia textového editora s danou funkcionalitou. Vzhľadom na tieto skutočnosti návrh nie je príliš detailný, zato sa zaoberá rôznymi alternatívami, ktoré môžu viesť k dosiahnutiu cieľa – vytvoreniu funkčného prototypu.

Pretože ešte nie je jasné, ktorú z alternatív sa nám podarí úspešne použiť, o detailnej architektúre zatiaľ nemôžeme hovoriť. Namiesto identifikácie jednotlivých súčiastok a ich rozloženia sa preto podkapitola venovaná architektúre zaoberá všeobecnejším opisom modelu architektúry na najvyššej úrovni.

4.1 Základné východiská

Ako bolo v zhrnutí na konci analýzy naznačené, rozhodli sme sa navrhnúť a vytvoriť vlastný textový editor, ktorý bude obsahovať požadovanú funkcionalitu. Možnosť vytvoriť

zásuvný modul (plugin) do niektorého z editorov podporujúcich takýto spôsob rozšírenia funkcionality sme zavrhli, pretože by išlo o príliš hlboký zásah do jadra editora.

Vhodnejšie v prípade využitia už existujúceho editoru by bolo využiť ho len ako základ a modifikovať ho. Takto vytvorená nadstavba by potom bola naším finálnym produktom v rámci zadaného projektu. Túto alternatívu sme zamietli z dôvodu, že by to znamenalo nutnosť študovať zdrojový kód vytvorený niekým iným. Čas strávený štúdiom pritom môžeme rovnako využiť aj vývojom vlastného textového editora s úplne základnou použiteľnosťou, doplnenou o funkcionlitu uvedenú v špecifikácii. Svojmu kódu potom budeme lepšie rozumieť než cudziemu.

Pri vytváraní editora využijeme jednu zo základných tried, ktoré Qt tookit ponúka, triedu `QMainWindow`. Ide o hlavné okno aplikácie ako ho všetci poznáme, s priamou možnosťou využitia menu a panelov nástrojov. Jednotlivé položky menu a panelov nástrojov budú reprezentované pomocou triedy `QAction`, ktorá umožňuje aj priradenie ikony a klávesovej skratky. Čo sa ich funkcionality týka, vytvorenie ich prepojenia na príslušné obslužné funkcie je v Qt záležitosť jedného príkazu.

4.2 Alternatívy jadra aplikácie

Vychádzajúc zo špecifikácie, náš textový editor by mal ku zvýrazneniu textu používať aj fonty, farby a grafické elementy. Toto všetko sa dá spraviť využitím elementov priamo v rámci Qt, kde máme prvky s bohatou podporou práce s obohateným textom (rich text). Čo však robí tento projekt zaujímavým, netradičným a sťažuje situáciu je fakt, že nami vytvorený textový editor má umožniť používateľovi prácu na úrovni blokov.

Komponent, ktorým bude nakoniec reprezentovaný blok, musí nezávisle od konkrétnej implementácie spĺňať niekoľko podmienok:

- podpora práce s obohateným textom
- schopnosť pracovať na úrovni textu
- schopnosť pracovať na úrovni blokov

- možnosť zbaliť/rozbaliť blok
- upravený vzhľad

Úprava vzhľadu bude dôležitá bez ohľadu na to, ktorá alternatíva nakoniec povedie k cieľu. Na základe zvyšných podmienok sa nám ponúkajú nasledovné 4 možnosti ako pristúpiť k riešeniu:

Trieda `QGraphicsTextItem` ako základ

Trieda `QGraphicsTextItem` bola už spomínaná v rámci analýzy. Táto trieda je, čo sa týka práce s textom, veľmi podobná triede `QTextEdit`. Umožňuje spracovanie obohateného textu, taktiež aj prístup do dokumentu prostredníctvom triedy `QTextCursor`. Použitie tohoto komponentu na reprezentáciu bloku je teda logicky jednou z alternatív.

Každý dokument otvorený v našom textovom editore bude reprezentovaný scénou, ktorá je inštanciou triedy `QGraphicsScene`. Túto bude používateľ vidieť vďaka triede `QGraphicsView`, ktorá sa stará o vizualizáciu scény. Jednotlivé bloky budú potom do scény pridávané ako inštancie triedy odvodenej od `QGraphicsTextItem`. Scéna vie rozoznať objekty, ktoré jej boli pridané. Akékoľvek zmeny vo formátovaní, ktoré používateľ zvolí v editore, budú odoslané scéne a jej úlohou je, aby boli zmeny vykonané v príslušnom bloku.

Použitie triedy `QGraphicsTextItem` má z hľadiska želaného správania sa blokov výhodu v tom, že pôvodné reakcie triedy na zachytávanie udalostí stačí v niektorých prípadoch len trochu upraviť. Príkladom je presúvanie komponentu po scéne. V pôvodnej implementácii ním môžeme pohybovať ľubovoľne, stačí ho chytiť, presunúť a pustiť. Úprava bude v tomto prípade pozostávať z pridania obmedzení čo sa týka pohybov bloku po scéne. Zároveň treba upraviť správanie sa komponentu na kliknutie myšou, aby sme po kliknutí na text mohli tento editovať, no pri uchopení bloku s ním manipulovali ako s celkom.

Väčší problém nastane pri vytváraní hierarchie blokov. Keďže jednotlivé komponenty uložené v scéne sú pôvodne samostatné jednotky, je potrebné zabezpečiť synchronizáciu vnorených blokov. Pri presune vonkajšieho bloku sa vnorené bloky musia presunúť s ním, pri zbalení vonkajšieho bloku musia „zmiznúť“ spolu s ostatným textom. Synchronizovaný

presun by bolo možné zabezpečiť napríklad v metóde, ktorá sa stará o vykresľovanie bloku. Je potrebná synchronizácia vonkajšieho bloku s priamo vnorenými a pre šírenie po všetkých úrovniach. Skrývanie bloku sa dá zabezpečiť vypnutím zobrazovania bloku.

Ďalší problematický bod je presunutie bloku do, resp. z niektorého iného bloku. V tomto prípade je potrebné vytvoriť miesto pre príslušný blok a pripojiť ho k vonkajšiemu, aby spolu boli synchronizované. Pri odnímaní bloku platí opačný proces.

Možnosť zbaliť/rozbaliť blok možno pridať pripojením grafického komponentu, ktorý bude blok sprevádzať. V rámci tejto alternatívy by sme využili `QGraphicsPixmapItem`, ktorý by sme synchronizovali s príslušným blokom podobne ako je načrtnuté vyššie. Príslušné správanie bloku bude implementované v obsluhe kliknutia na príslušný komponent.

Trieda `QTextEdit` ako základ

Ako bolo uvednené v analýze (časť 3.2.1), `QTextEdit` je jednou zo základných tried integrovaných priamo v Qt toolkite, ktorá podporuje prácu s rich text prvkami. V rámci tejto alternatívy je blok reprezentovaný súčiastkou (widget), ktorá obsahuje tlačidlo na zbalenie/rozbalenie bloku a komponent odvodený od triedy `QTextEdit`. Takáto reprezentácia umožní veľmi ľahkú prácu s obohateným textom, rovnako ako aj kopírovanie, vkladanie obrázkov, vyhľadávanie v texte a pod. Pri takejto reprezentácii by aj problém so zbalením/rozbalením bloku bol triviálny, stačilo by nechať nezobrazovať/zobrazovať telo bloku.

Súčiastky ako také nie je možné do `QTextEdit`-u vkladať. Po vytvorení ich teda len necháme zobrazovať používateľovi, budeme odchytávať ich vykresľovanie a v rámci neho ich po normálnom vykreslení prekreslíme do obrázka. Do `QTextEdit`-u je totiž možné vkladať obrázky. Presúvanie blokov teda bude založené na presúvaní obrázkov. Keďže základom blokov bude komponent odvodený od `QTextEdit`-u, tento spôsob vkladania zároveň umožňuje zo svojej podstaty vytváranie hierarchie blokov. Keď v rámci hierarchie príde k zmenie v niektorom z blokov, v celej hierarchii smerom hore by sa vynútila aktualizácia, aby používateľ mohol danú zmenu vnímať.

V rámci tejto alternatívy sa črtá problém, či nám možnosti Qt umožňujú po kliknutí

na príslušný obrázok reprezentujúci blok zistiť, na ktorý blok sme klikli (na ktoré miesto v rámci daného bloku) a následne predať príslušnú udalosť danému bloku na spracovanie.

Založiť riešenie na použití QScintilly

V prípade, že nás neuspokoja možnosti, ktoré ponúka samotný Qt toolkit, zvážili sme aj možnosť využiť QScintillu (viď 3.2.3). QScintilla priamo ponúka veľa vymožeností z hľadiska podpory práce programátora, no na strane druhej by bolo potrebné pridať určité správanie, ktoré je vyžadované a vyššie spomínané komponenty ho zvládajú.

Vytvorenie vlastného komponentu

Toto je posledná alternatíva, ku ktorej sa uchýlime len v krajnej núdzi. Znamenalo by to totiž, že sami musíme implementovať všetky užitočné funkcie pre prácu s textom a podporu práce s obohateným textom, čiže veci, ktoré vyššie spomínané komponenty priamo ponúkajú.

4.3 Práca s blokmi

Nasleduje návrh niektorých aspektov práce s blokmi kódu.

Čo všetko je blok?

Pri syntaktickej analýze sme definovali čo je všetko budeme pokladať za blok. Napríklad, celý program bude hlavný blok (vrchol stromu) ten bude obsahovať podblok funkciu a funkcia sa skladá z hlavičky a tela. Hlavička sa skladá z návratovej hodnoty, názvu a parametrov. Parametre sú ďalší blok, ktorý sa skladá z menších podblokov už samostatných parametrov. Z tohto vyplýva, že parametre sú blokom ako celok, ale neuchovávajú žiadny text.

Presun blokov

Na základe syntaktickej analýzy sa vytvorí stromová hierarchia blokov daného zdrojového kódu. Používateľ bude môcť dané bloky presúvať systémom „drag-and-drop“. Kvôli tomu, aby bolo možné presúvať napríklad celú funkciu, vrátane všetkých podblokov, je nutná stromová hierarchia. Pri presúvaní akéhokoľvek bloku sa vždy budú zároveň presúvať aj všetky jeho podbloky. Pri tomto presúvaní bude nesmierne dôležité dovoliť presunutie bloku na miesta, kde sa naozaj môže dať, teda aby sa dané bloky napríklad neprekrývali.

Okrem toho treba zabezpečiť, aby bolo možné vložiť/presunúť bloky medzi už existujúce bloky. Pri presúvaní bloku medzi existujúce bloky sa medzi nimi vykreslí dočasná čiara, ktorá bude indikovať miesto, kde bude daný blok presunutý. Taktiež bude potrebné rozlíšiť, či používateľ chce blok presunúť medzi bloky alebo vložiť do bloku.

Z programátorského hľadiska to bude riešené tak, že ak sa kurzor myši nachádza medzi blokmi, vykreslí sa už spomínaná dočasná čiara indikujúca vloženie medzi bloky. Z toho vyplýva, že medzi blokmi bude musieť byť vynechaný určitý voľný priestor. Bloky, medzi ktoré sa daný blok vkladá sa „rozostúpia“ a vytvorí sa tak priestor pre vkladaný blok. Ak sa kurzor myši bude nachádzať v určitom bloku, bude to znamenať vloženie do bloku. Vtedy bude potrebné zabezpečiť zväčšenie bloku, do ktorého sa daný blok vkladá a posunutie jeho podblokov, aby sa tak vytvoril priestor pre vkladaný blok.

Skrývanie blokov

Kvôli zjednodušovaniu zdrojového kódu bude možné skryť jednotlivé bloky. Pri skrývaní sa opäť uplatňuje princíp hierarchizácie. Tým, že používateľ skryje funkcie, ostane zobrazený len prvý riadok bloku a zvyšok bloku spolu s ostatnými podblokmi sa tiež skryjú. Keďže funkcia ako blok neobsahuje vlastne žiaden text v tomto konkrétnom prípade ostanú zobrazené bloky na prvom riadku. Keď sa pred skrytím celej funkcie skryje iný podblok, tak po rozbalení (odokrytí) funkcie bude tento blok stále skrytý, pre zobrazenie je nutné ho potom samozrejme rozbaliť.

Veľkosť blokov

Veľkosť bloku je veľmi dôležitá, pretože blok obsahuje vo väčšine prípadov málo textu ale jeho veľkosť musí byť reálne väčšia pretože obsahuje aj iné podbloky, ktoré musí zahŕňať. Pri písaní textu v editore sa automaticky zväčšuje blok, v ktorom sa aktuálne píše, ale keď veľkosť podbloku presiahne veľkosť bloku v hierarchii nad ním automaticky sa začne zväčšovať aj daný blok.

4.4 Syntaktický analyzátor

Modul syntaktického analyzátora (a tiež lexikálneho, v prípade nevyužitia (Q)Scintilly) bude postavený na knižnici LPeg (viď časť 3.3.1). Spracúvanie externých skriptov vyžaduje interpret jazyka Lua, ktorý bude priamo súčasťou aplikácie.

Podsystém načíta gramatiku požadovaného jazyka z dodaného skriptu a počas behu bude podľa potrieb iných podsystémov generovať abstraktný syntaktický strom (AST). AST bude priamo využívaný vizualizačným modulom pri vykresľovaní grafických blokov. Počas práce používateľa sa bude musieť automaticky (respektíve manuálne) spúšťať opakované generovanie kompletného, prípadne len parciálneho AST.

Dôležitou časťou je ošetrovanie vzniku syntaktických chýb, ktoré by nemali ovplyvňovať celý AST, ale len jeho minimálnu podmnožinu. Toto dosiahneme sledovaním bloku, ktorý práve používateľ edituje a následnou analýzou obmedzenou len na tento blok a bloky v ňom vnorené.

AST bude reprezentovaný stromom pozostávajúcim z *textových* a *zložených* elementov (oboch reprezentovaných triedou `TElement`). Okrem spoločného atribútu `type` budú textové elementy používať atribút `text` (obsahujúci text) a zložený atribút `content` (obsahujúci zoznam ľubovoľných elementov) na ukladanie svojho obsahu. Každý zložený element reprezentuje jeden grafický blok, ktorého formátovanie určuje kľúč `type`.

Vzhľadom na možnosť pridávania nových jazykov máme tri možnosti definície formátovacích kľúčov:

1. Kľúče budú dané, jazykovo nezávislé s pevným formátovaním, napríklad `keyword`, `number`, `control_statement`, a podobne. Gramatika v skripte bude parsovať daný jazyk priamo do týchto všeobecných kategórií.
2. Kľúče budú definované spolu s príslušným formátovaním v skripte obsahujúcom gramatiku jazyka. Príklady pre C++ sú `include`, `while` alebo `function_parameter`. Vyžaduje použiť metajazyk na popis formátovania.
3. Kombinácia predošlých. Skript definuje jazykovo špecifické kľúče, ktoré následne namapuje na „vstavané“ základné kľúče (s pevným formátovaním). Zároveň má autor skriptu možnosť doplniť nové kategórie, prípadne modifikovať prednastavený spôsob formátovania.

Pri implementácii prototypu zvolíme pravdepodobne prvú, najjednoduchšiu cestu, ale neskôr zvážime aj možnosti niektorého z prispôsobivejších variantov.

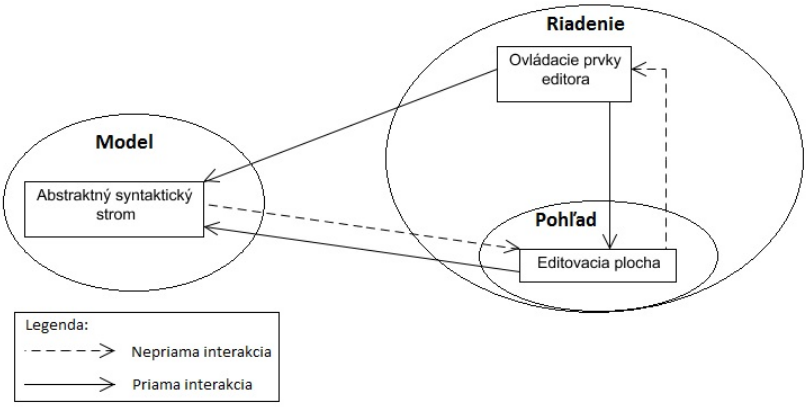
4.5 Modul pre využitie značiek RTF

Tento modul zabezpečí rozpoznávanie značiek RTF (viď 3.3.2) v našom editore. Značky, ktoré sa v dokumente budú využívať na označovanie blokov dokumentácie, ako aj ostatné formátovacie značky, budú rozpoznávané, ale v editore sa nebudú zobrazovať. Čo sa týka zakomponovania RTF do nášho projektu, z danej špecifikácie plánujeme využiť najmä komentáre.

4.6 Architektúra systému

Pri implementácii textového editora by sme chceli využiť štandardný návrhový vzor model-pohľad-riadenie (Model-View-Controller). V našom prípade bude modelom abstraktný syntaktický strom, ktorý vznikne syntaktickou analýzou textu. Čo sa pohľadu týka, dosť závisí od konečnej, použitej alternatívy. Vo všeobecnosti to však bude komponenta, ktorá má na starosti vizualizáciu textu a jeho štrukturovanie. Pohľad nám bude zobrazovať text usporiadaný do blokov v závislosti na tom, ako je tento uložený v syntaktickom strome (modeli).

Riadenie bude zahŕňať ovládacie prvky editora. Vzhľadom na to, že plocha editora slúži nielen na zobrazovanie, ale aj na editáciu textu, môžeme povedať, že sa tu čiastočne zlučuje riadenie a pohľad. Situáciu možno vidieť na Obrázku 4.1.



Obr. 4.1: Návrh architektúry

Kapitola 5

Prototyp

5.1 Editačný komponent

V rámci vytvárania prototypu sme vytvorili kostru textového editora. Väčšina položiek menu však nemá žiadnu funkcionalitu, zvyšok len obmedzenú. Taktiež tlačidlá na formátovanie v paneli nástrojov nie sú funkčné. To však ani nebolo cieľom, viac sme sa sústredili na naprogramovanie správania sa blokov a vývoj jednotlivých modulov.

Kostra mala slúžiť na poskytnutie rámca pre prototypovanie správania sa blokov, v rámci ktorého prebiehala pokusná integrácia s ostatnými modulmi. Taktiež nám poskytla možnosť vyskúšať si niektoré aspekty tvorby aplikácie v Qt, tvorbu menu a panelov nástrojov. Určité časti tejto kostry spolu s nadobudnutými skúsenosťami budeme môcť potom využiť pri tvorbe finálnej aplikácie. A v neposlednom rade pri tejto kostre sme mali možnosť zistiť ďalšie detaily, ktoré budú potom dôležité, čo všetko bude treba ešte zohľadniť, aby sa to správalo ako textový editor.

5.1.1 Práca s blokmi

Pri reprezentácii blokov sme sa rozhodli pre prvú alternatívu spomínanú v návrhu (viď časť 4.2). Dokument je teda reprezentovaný ako grafická scéna, do ktorej umiestňujeme jednotlivé bloky ako grafické textové prvky. V rámci prototypu ešte nie je implementované

želané finálne správanie sa blokov, sústredili sme sa na hlavné rysy. Rovnako vzhľad blokov je len ilustračný, slúži iba na vizualizáciu pre lepšiu predstavu.

Hierarchia blokov

Bloky môžu byť usporiadané do stromovej štruktúry. Každý blok pozná svojho priameho predka a má zoznam svojich priamych potomkov. Ak je potrebné vykonať niečo v rámci daného stromu, akcia sa jednoducho prešíri v takto udržiavanom strome smerom nadol. Bloky v rámcie hierarchie sa presúvajú spolu, taktiež ukrytie obsahu predka spôsobí skrytie všetkých potomkov v hierarchii.

Hierarchiu blokov je možné vytvárať dvojako:

- Vytvorením nového bloku v oblasti, ktorú už zaberá iný blok. V tomto prípade sa pridá novovytvorený blok medzi potomkov toho pôvodného a pôvodný blok sa stáva rodičom nového bloku.
- Presunutím bloku do oblasti, ktorú už zaberá iný blok. Pôvodný blok sa stáva predkom presunutého bloku. V prípade, že presunutý blok bol na nejakom inom mieste v rámci hierarchie či dokonca bol členom úplne iného stromu blokov, všetky pôvodné väzby s daným blokom sú automaticky zrušené.

Spoločné presúvanie blokov v rámci hierarchie je implementované v metóde reagujúcej na presun bloku myšou. Pozícia všetkých potomkov v hierarchii sa zmení o vektor posunutia rodičovského bloku. Je to teda zmena oproti návrhu, kde sme uvádzali, že možné by to napríklad bolo vo vykresľovacej metóde bloku. Zmena bola nutná, pretože potrebujeme zohľadniť skutočnú zmenu pozície potomkov, nielen prekresliť ich na nové miesto.

Skrývanie blokov

Každý blok je sprevádzaný grafickým elementom, ktorý reaguje na kliknutie. Na takéto kliknutie sa obsah bloku podľa aktuálneho stavu buď ukryje alebo znovu objaví. Pri ukrytí je ponechaný prvý riadok textu. Spolu s ukrytím obsahu sa skrývajú aj všetci potomci bloku v rámci hierarchie.

Pri skrývaní sa najprv uloží šírka skrývaného bloku. To je dôležité, pretože šírka bloku nemusí zodpovedaťšírke prvého riadku, ktorý ponechávame. Blok by v takom prípade zmenšil svoju šírku, čo je pre nás neprijateľné správanie. Ďalej prekopírujeme pôvodný obsah a nastavíme text bloku, aby obsahoval len prvý riadok. Všetkých potomkov v rámci hierarchie potom prestaneme zobrazovať. Odkrytie bloku je tiež priamočiary proces. Obsah bloku nastavíme na pôvodný obsah a potomkov v rámci hierarchie znovu necháme zobraziť.

Veľkosť blokov

Originálne grafické textové prvky mali veľkosť určovanú podľa toho, akú veľkú oblasť zaberal ich textový obsah. My sme toto správanie museli upraviť, aby sme zohľadnili niektoré špecifiká.

V prípade, že je blok ukrytý, obsahuje len prvý riadok, pričom jeho šírka môže byť menšia ako šírka bloku. Preto je pri skrývaní nutné uložiť si pôvodnú šírku bloku, aby sme v prípade, že je blok ukrytý, mohli napriek všetkému zachovať jeho pôvodnú šírku. Výška bloku sa automaticky prispôsobuje veľkosti písma v ponechanom riadku.

Inak šírku bloku určujeme nastavením jeho pravého okraja na zadanú x-ovú súradnicu. Táto súradnica je väčšou z hodnôt x-ovej súradnice oblasti, ktorú zaberá text bloku, a x-ovej súradnice najpravejšieho okraja niektorého z potomkov bloku. Pri výpočte šírky bloku sú preto potomkovia vždy aktuálne zoradení podľa pravého okraja, porovnajú sa dané hodnoty a na väčšiu z nich sa nastaví okraj bloku.

5.2 Modul syntaktického analyzátora

Tento modul má dve hlavné časti. Prvou je gramatika analyzovaného jazyka realizovaná ako Lua skript a druhou je abstraktný syntaktický strom (ďalej AST), ktorý sa pomocou tejto gramatiky generuje. Stručne popíšeme funkcionalitu a niektoré implementačné detaily oboch týchto častí.

5.2.1 Gramatika

Na špecifikáciu gramatiky v jazyku Lua sme použili knižnicu LPeg. Vytvorená bola gramatika pre jazyk C, pričom sme vychádzali 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:

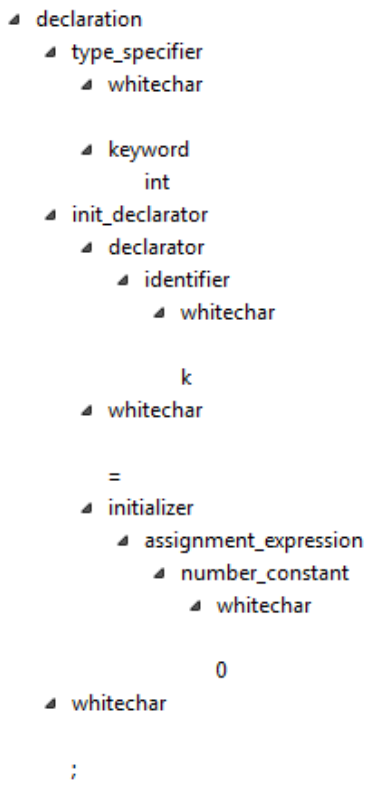
- aplikacia spustí skript s gramatikou a gramatika sa skompiluje
- aplikácia zavolá LPeg funkciu `match`, ktorej vstupom je gramatika a text (kód), ktorý chceme analyzovať
- vystupom je Lua tabuľka obsahujúca ďalšie tabuľky a tento systém tabuliek zodpovedá syntaktickému stromu
- výstup je umiestnený na zásobník z ktorého je postupne čítaný, z Lua tabuliek sa zrekonštruuje AST v C++

Gramatika využíva funkcie na zachytávanie častí 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.

5.2.2 Syntaktický strom

Implementácia AST sa nelíši od spôsobu uvedeného v návrhu (časť 4.4). Zmenou je zrušenie atribútu `text`, ktorý mal slúžiť pre textové uzly. Text v textových uzloch sa ukladá do atribútu `type`, ktorý pri netextových uzloch obsahuje typ uzla. Identifikácia textových uzlov je jednoduchá vzhľadom nato, že vždy ide o listy stromu (a zároveň sú všetky listy

textovými uzlami). Strom obsahuje okrem štandardných uzlov z BNF gramatiky aj uzly typu `keyword`, ktoré uľahčujú identifikáciu kľúčových slov. Obrázok 5.1 ukazuje AST vygenerovaný pre výraz `int k=0;`[1].



Obr. 5.1: Ukážka syntaktického stromu

Modul poskytuje aj možnosť dopĺňania stromu prostredníctvom analýzy jeho podstromov. Uzly, ktoré môžu figurovať ako korene podstromov (tzv. *vstupné body* pre analýzu) sú vymenované v skripte gramatiky v premennej `entrypoints`. V aktuálnej C gramatike sú to *deklarácia premennej*, *deklarácia funkcie*, *inštrukcia preprocesora* a *príkaz*. Vždy, keď sa obsah niektorého z textových uzlov zmení, nájdeme najbližšieho takého predka, ktorý je vstupným bodom, a podstrom reanalyzujeme . Dôležité je, že zvyšok AST je nezávislý od analyzovanej vetvy.

[1]Uzly `whitechar` sa momentálne nachádzajú aj tam, kde by nemali (obsahujú prázdny retazec).

5.3 Modul pre literate programming

Modul slúži na vytváranie dokumentácie priamo v zdrojom kóde. Jeho úlohou je zabezpečiť kompilovateľnosť programu pre kompilátor a vysokú čitateľnosť dokumentačnej časti pre programátora. Jeho funkcionalitu môžeme rozdeliť do dvoch častí.

5.3.1 Načítanie zdrojového kódu

Pri načítavaní zdrojového kódu je potrebné previesť dokumentačnú časť kódu do podoby ľahko čitateľnej pre programátora. Z načítaného kódu sa najprv odstránia značky pre komentár. Následne sa analyzuje načítaný text a načítané značky v RTF formáte sa prevedú na používateľské značky, ktoré používa programátor. Používateľské značky, ktoré boli implementované sú `<Author>` a `<Text>`. Po značke `<Author>` nasleduje identifikátor autora dokumentácie. Za značkou `<Text>` nasleduje samotný text dokumentácie.

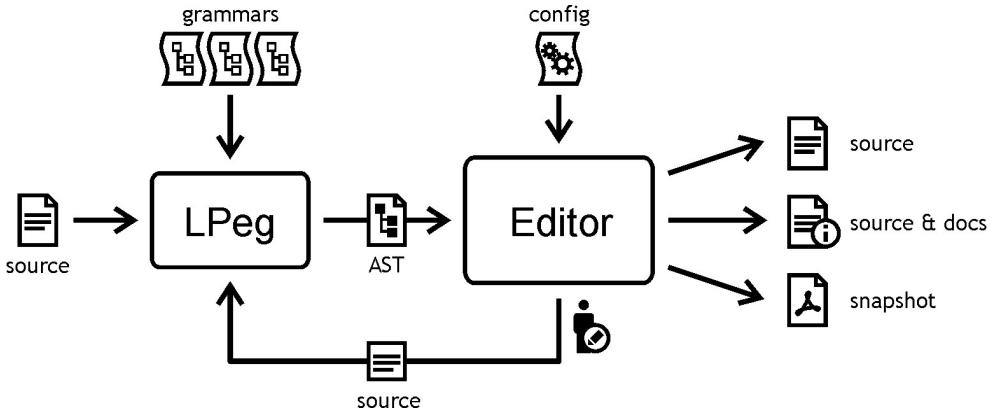
5.3.2 Uloženie dokumentácie v kóde

Pri ukladaní vytvoreného zdrojového súboru potrebujeme previesť používateľské značky spätne do RTF formátu. Takto vytvorený text je potom potrebné vložiť medzi značky komentáru, čím zabezpečíme kompilovateľnosť programu. Používateľská značka `<Author>` sa mení na RTF značku `\atnid`. Značka `<Text>` sa zmení na `\annotation`. Takto definované značky bude možné v budúcnosti ďalej spracovávať pre vygenerovanie dokumentácie.

Kapitola 6

Návrh v letnom semestri

V letnom semestri sa nám podarilo v značnej miere implementovať návrh zo zimného semestra. Počas implementácie sa objavili chyby v návrhu, na ktoré bolo nutné reagovať jeho zmenou. Do návrhu tiež boli doplnené nové myšlienky, ale i zakomponované riešenia, ktoré sme navrhli až po nadobudnutí dostatočných skúseností s knižnicou *Qt*. Pri plánovaní počas semestra sme vypracovali zoznam kľúčových vlastností a hlavné časti funkcionality editora, ktoré sme chceli implementovať. Nasledujúca schéma charakterizuje princípy editora:



Obr. 6.1: Principiálna schéma editora *TrollEdit*.

6.1 Editácia blokov

Návrh práce s blokmi v časti *4.3 Práca s blokmi* bol rozšírený. Je dôležité, aby sa text v blokoch pri editácii správal tak, ako je na to používateľ zvyknutý z bežných textových editorov. Musí správne fungovať zalamovanie riadkov, mazanie textu a pohyb kurzora. Pri odsadení prvého riadku bloku sa automaticky odsadí celý blok. Počas písania bude väčšinu zobrazovacej plochy zaberať len text a obrysy blokov sa budú zobrazovať len u aktuálne vybratého uzla a jeho rodičov (parent).

Po kliknutí do bloku sa označí daný blok (zmena farby pozadia), ale zároveň sa prehľadne zobrazia i jeho rodičia (iný odtieň farby pozadia). Takýmto spôsobom môžeme označiť aj netextový blok (blok, ktorý obsahuje len iné bloky). Pri stlačení a ťahaní ľavým tlačidlom vybraný blok presunieme na novú pozíciu. Skrývanie (folding) blokov bude realizované pomocou ikon, ktoré sa budú pre každý viacriadkový blok zobrazovať na lište v ľavej časti okna. Pri skrývaní bude mať používateľ možnosť označiť skrytý obsah vlastným pomenovaním.

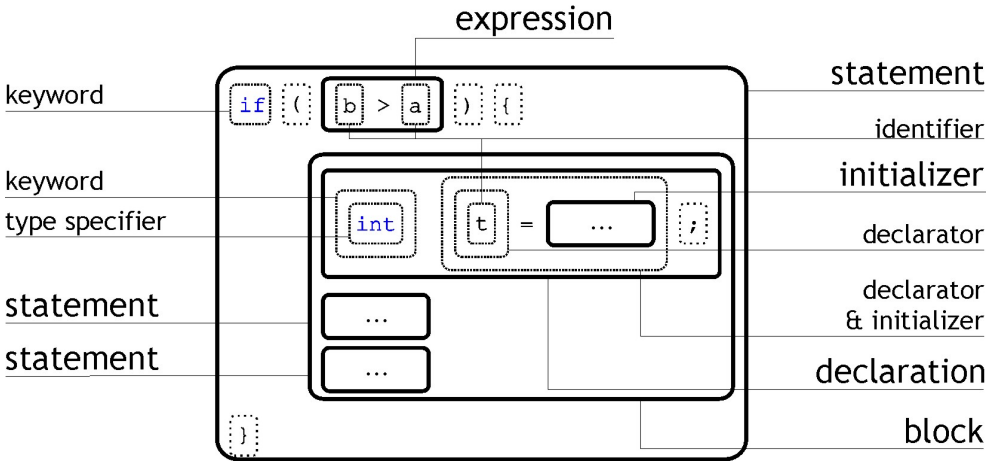
6.2 Syntaktický analyzátor

Program automaticky priraďuje gramatiku otváraným súborom, vykonáva analýzu textu a generuje grafickú reprezentáciu AST. Gramatika opísaná v časti *5.2.1 Gramatika* bola rozšírená o zoznam elementov, ktoré bude môcť používateľ označovať/presunúť, keďže nie je žiadúce, aby bolo možné označiť každý blok. Spôsoby akými gramatika spracúva biele znaky a syntakticky nekorektné časti kódu boli mierne modifikované.

6.2.1 Definícia bloku

6.2.2 Reanalýza blokovej hierarchie

Pri písaní zdrojového kódu bude nutné aktualizovať AST strom a zapracovať doň potrebné zmeny, aby vnútorná reprezentácia zodpovedala realite. Táto reanalýza sa bude vykonávať



Obr. 6.2: Analýza do blokov, pomenovanie typov blokov v názornej ukážke.

po prechode na ďalší riadok v zdrojovom kóde.

6.3 Zvýrazňovanie syntaxe

Editor má slúžiť ako nástroj pre programátorov, je dôležité, aby sa písaný zdrojový kód nezlieval do jednoliatej masy. Presne tomuto zabráni zvýrazňovanie syntaxe. Syntaktická analýza programu umožní identifikovať elementy, ktoré majú byť zvýraznené, ako napríklad kľúčové slová, komentáre, reťazce, a pod. Konfiguračný súbor editora bude obsahovať spôsob zvýrazňovania pre jednotlivé elementy – farbu textu a či má byť použité tučné písmo, kurzíva, podčiarknutie textu a iné. Vzhľadom na to, že náš editor umožňuje prácu s grafickými blokmi vytváranými na základe analýzy textu, je možné posunúť zvýrazňovanie prvkov jazyka až na úroveň blokov. Vybraté bloky budú napríklad podsvietené inou farbou podľa elementu AST, ktorý obsahujú.

Návrh v časti *4.4 Syntaktický analyzátor* predpokladá, že budú využité kľúče nezávislé na konkrétnom jazyku. To je veľmi výhodné z hľadiska zvýrazňovania textu, umožní to totiž mať rovnako naformátovaný prvok jazyka (napr. kľúčové slovo, komentár, atď.) bez ohľadu na konkrétny programovací jazyk a zápis daného prvku v ňom.

6.4 Dokumentačné bloky

Doterajší návrh modulu pre literate programming opísaný v kapitole *4.5 Modul pre využitie značiek RTF* sa zakladal na princípe využitia značiek RTF na formátovanie dokumentačných blokov. Tento princíp ostal zachovaný pre formátovanie obohateného textu (rich text) v dokumentačných blokoch, aby bolo možné previesť dokumentačné bloky do textovej podoby v komentároch daného programovacieho jazyka. Keďže správanie sa takéhoto textu sa značne líši od správania textu zdrojového kódu, bolo nevyhnutné vytvoriť novú triedu pre reprezentáciu dokumentačných blokov.

Veľkou výhodou nášho editora je i umiestňovanie obrázkov ako dokumentačných blokov. Tieto obrázky (napr. UML diagramy) bude možné zmenšovať a zväčšovať a umiestňovať ich napríklad i na pravú časť obrazovky (nie len pod seba ako sme zvyknutí pri príkazoch), kde je väčšinou iba prázdne miesto, čo prispeje k zvýšeniu prehľadnosti a k lepšiemu využívaniu pracovného miesta. Aj kvôli možnosti manuálnej zmeny veľkosti a polohy sme sa rozhodli pre vytvorenie už spomínanej novej triedy pre dokumentačné bloky. Väzbu medzi dokumentačným blokom a súvisiacim blokom zdrojového kódu zobrazíme použitím grafických elementov – šípok. Obrázky a diagramy so väčšinou vytvárané s veľkým rozlíšením a ich umiestnenie na pracovnej ploche editora by niekedy mohlo znamenať jej celé obsadenie, a tým zníženú prehľadnosť. Tomuto faktu chceme zabrániť možnosťou zmeny veľkosti orázku. Ako aj možnosť otvorenia obrázku v externom prednastavenom editore.

Medzi značné zlepšenie funkcionality editora považujeme možnosť vkladať na pracovnú plochu editora i súbory akéhokoľvek typu, napríklad zvukové záznamy alebo inú dokumentáciu v ľubovoľnom formáte (PDF, doc, atď). Po zanechaní súboru na ploche editora sa na danom mieste vytvorí odkaz, tento odkaz umožní spustiť daný súbor v externom editore prednastavenom na danom počítači. Takéto riešenie je veľmi jednoduché a prispeje k prehľadnejšej dokumentácii a udržiavaní všetkých potrebných súborov na jednom mieste. Samozrejme i pri týchto blokoch je možné, aby boli napojené graficky šípkou na rodiča, na ktorého sa viažu.

6.5 Tlač do PDF

Aby bola dokumentácia dobre čitateľná, používateľ môže jednotlivé dokumentačné bloky ľubovoľne usporiadať, a tým vytvoriť vizuálne a obsahovo súvislú dokumentáciu, ktorá bude vhodná aj pre vytvorenie dokumentácie vo forme PDF. Tlačenie do PDF formátu je na báze exportovania vymedzenej časti grafickej scény. Časť scény, ktorú je možné vytlačiť je možné zviditeľniť tlačidlom *Printable Area*, vtedy sa na pracovnej ploche editora objavia deliace čiary pre jednotlivé strany. Je to výhodé aj z toho dôvodu, aby si používateľ vedel sám učiť, ktoré časti zdrojového kódu chce a ktoré nepotrebuje (skryté bloky) vytlačiť. Takisto rozhoduje i o tom či dokumentačné bloky budú vytlačené alebo nie. Ak ich používateľ nebude chcieť vytlačiť, jednoducho ich presunie za hranicu okraja tlače alebo ich skryje.

Kapitola 7

Produkt

7.1 Editácia blokov

Kostru editora popisovanú v časti *5.1 Editačný komponent* sme rozšírili a pridali sme novú funkcionalitu. Implementovali sme editáciu textu (písanie, mazanie, zlomy riadkov) ako aj pohyb medzi blokmi pomocou klávesnice. Medzery a tabulátory sme sa rozhodli nezobrazovať a realizovať ich len ako prázdny priestor pred príslušným blokom. Podobne bol aj znak konca riadku nahradený nastavením príznaku v bloku. Blok obsahujúci viac riadkov má odsadenie spoločné pre všetky svoje podbloky.

Zrušili sme manuálne vkladanie nových blokov, ktoré sa budú vytvárať z napísaného textu automaticky pri jeho analýze. Pri pohybe blokov sa zobrazuje pomocná čiara, ktorá používateľovi signalizuje kam sa blok aktuálne vloží. Na určovanie veľkosti blokov sme použili vstavanú funkciu `childrenBoundingRect()`, ktorá automaticky spočíta obdĺžnik zahŕňajúci všetkých potomkov bloku. Zoraďovanie potomkov podľa pravého okraja spomínané v podsekcii *Veľkosť blokov* (v časti *5.1.1 Práca s blokmi*) už nie je potrebné.

7.1.1 Zvýrazňovanie syntaxe

Popis formátovania jednotlivých prvkov, ktoré majú byť zvýraznené, je obsiahnutý v konfiguračnom súbore, ktorý sprevádza editor. Tento je realizovaný ako Lua skript. Pri štarte

editora sa načítajú konfiguračné údaje, ktoré sú následne spracovávané. Prvou zložkou konfiguračného súbora je tabuľka obsahujúca kľúče všetkých tabuliek v poradí, v akom majú byť načítavané. Všetky ostatné tabuľky majú jednotnú štruktúru, aby bolo možné ich univerzálne spracovanie bez ohľadu na to, aké položky v skutočnosti obsahujú. Tabuľky môžu v súčasnej verzii obsahovať tieto položky:

- **target** - určuje, či tabuľka obsahuje formátovanie textu alebo opisuje vzhľad bloku. Môže nadobúdať hodnoty *text*, *block* alebo *both*. Hodnota *both* určuje, že tabuľka opisuje obe formátovania.
- **base_t** - určuje formátovanie textu, od ktorého aktuálne opisované formátovanie dedí nastavenia parametrov. Hodnotou je názov príslušného formátovania.
- **base_b** - určuje štýlovanie bloku, od ktorého aktuálne opisované štýlovanie dedí nastavenia parametrov. Hodnotou je názov príslušného štýlovania.
- **family** - typ písma.
- **size** - veľkosť písma (v bodoch).
- **color** - farba písma.
- **bold** - určuje, či má byť text tučný.
- **italic** - určuje, či má byť použitá kurzíva.
- **underline** - určuje, či má byť text podčiarknutý.
- **hovered** - farba pozadia bloku, keď na neho ukáže kurzor.
- **hovered_border** - farba okraja bloku, keď na neho ukáže kurzor.
- **selected** - farba pozadia bloku, keď je blok označený.
- **showing** - farba okrajov blokov, ktoré sú predchodcami označeného bloku v rámci hierarchie.

Ako je vidieť, konfiguračný súbor ponúka možnosť definovať si zopár základných štýlov, od ktorých ďalšie štýly jednoducho dedia nastavenia jednolivých paramterov. V prípade, že chceme niektorý zo zdedených parametrov zmeniť, stačí za základným štýlom uviesť názov položky a novú želanú hodnotu.

Metóda, ktorá spracúva formátovanie textu, dostane načítané údaje z konfiguračného súboru. Venuje sa len tým, kde položka *target* ukazuje, že to má zmysel. V prípade, že je definovaný základný formát, od ktorého sa dedia nastavenia parametrov, vytvoria sa kópie písma a textu definovaných v základnom formáte. Ak sú uvedené ďalšie parametre, príslušné hodnoty sú potom nastavené. Na rovnakom princípe pracuje aj metóda, ktorá má na starosti spracovanie nastavení štýlovania blokov.

Výsledkom spracovania textového formátovania je pár písmo + farba, ktorý určí konkrétne zvýraznenie. Takéto páry sú ukladané do hash tabuľky, kde kľúčom je typ elementu gamatiky, ktorý má byť daným štýlom naformátovaný. Spracovanie štýlovania blokov má už zložitejší výstup. Výsledkom je totiž hash tabuľka, kde kľúčom je typ elementu, ktorého sa štýlovanie týka, a hodnotou hash tabuľka dvojíc *názov parametra* → *hodnota parametra*.

Samotné zvýrazňovanie textu a štýlovanie si obstaráva každý blok sám. Má prístup k formátovacím údajom, odkiaľ si vie na základe typu obsahu vybrať potrebné informácie. Ako kľúč použije typ elementu AST, ktorý mu je priradený. Nastavenie príslušného písma pre textový blok zabezpečí formátovanie, štýlovanie sa zasa vykoná v metóde vykreslenia použitím nastavenej farby.

7.2 Modul analyzátora

7.2.1 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:

$$uzol1, uzol1.1, uzol1.1.1, \ldots, \ldots, uzol1.2, \ldots, uzol1.3, \ldots, \ldots$$

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.

7.2.2 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.

7.3 Grafická reprezentácia blokov

Na grafickú reprezentáciu dát uchovávaných hierarchiou objektov `TreeElement` slúži trieda `Block`, ktorá je podtriedou `QGraphicsRectItem` (jednoduchý grafický prvok v tvare obdĺžnika) a tiež triedy `QObject` (kvôli využívaniu signálov a slotov). Manipulácia s obsahom a štruktúrou `Block`-ov je automaticky reflektovaná aj v stave `TreeElement`-ov — obe hierarchie musia byť v každom okamihu konzistentné. Pri zmazaní bloku sa zmaže aj príslušný element (nie naopak).

Informácie o koreňovom bloku ako aj niektoré iné údaje a funkcie spoločné pre všetky bloky uchováva trieda `BlockGroup`. Túto triedu je v budúcnosti možné využiť aj na paralelné zobrazenie viacerých hierarchií (`BlockGroup`) na jednej scéne (`DocScene`).

7.3.1 Interakcia s blokmi

Používateľ má možnosť editovať text v blokoch a meniť ich štruktúru. Textové bloky obsahujú objekt triedy `TextItem` (podtrieda `QGraphicsTextItem`), ktorá okrem editovateľného poľa umožňuje vysielanie signálov pri stlačení kláves, ktoré môžu ovplyvňovať aj iné bloky (ako napríklad Enter, Del alebo smerové šípky). Tieto signály zachytáva trieda `BlockGroup`, ktorá následne sprostredkúva zmeny zasiahnutým blokom.

Presun blokov v strome je postavaný na Qt implementácii technológie drag-and-drop. Presúvateľné sú len tie bloky, ktoré sú v gramatike deklarované ako označiteľné (`selectable`). Pri každej významnej interakcii, pri ktorej dochádza k zmene veľkosti alebo polohy blokov, sú zmeny plynulo animované.

7.3.2 Plávajúce bloky

Bloky, ktoré sú v gramatike explicitne vymenované ako plávajúce (`floating`) nie sú zaradené do hierarchie, ale sú umiestnené mimo nej. Zároveň sú voľne presúvateľné. Takéto bloky sa aktuálne využívajú na prezentáciu komentárov, dokumentácie, obrázkov a hyperlinkov na externé súbory.

7.4 Dokumentačné bloky

Doterajšia implementácia je opísaná v kapitole *5.3 Modul pre literate programming*. Opísaný spôsob implementácie je už neaktuálny, používateľ nebude používať značky, vďaka ktorým bude formátovať text pri generovaní dokumentácie. Formátovanie textu bude princípom stláčania tlačidiel tučné, kurzíva, podčiarknutie a pravdepodobne aj rez písma. Tieto možnosti budú používateľovi poskytnuté, keď vytvorí dokumentačný blok použitím klávesy Ctrl a ľavého kliknutia myši. Dokumentačný blok tvorí samostatná trieda `DocBlock`, ktorá dedí od triedy `Block`. Bolo nutné vytvoriť novú triedu, pretože správanie sa dokumentačných blokov je odlišné od bežných blokov v zdrojovom kóde. Napríklad môžu obsahovať text vo viacerých riadkoch alebo obrázok, či dokonca odkaz na iný súbor. Pri vkladaní obrázka systémom drag-and-drop sa zistí aktuálne označený blok, vytvorí sa spojenie grafickým elementom – šípkou medzi označeným blokom (blok ku ktorému sa vzťahuje obrázok) a novým blokom, ktorý obsahuje obrázok. Obrázok sa vkladá v `TextItem` na nultú pozíciu a je mu zakázané obsahovať v tomto prípade text. Daný nový blok s obrázkom je vytvorený na mieste „pustenia“ obrázka na pracovnú plochu editora (grafická scéna).

Všetky komentáre v zdrojových súboroch sme sa rozhodli oddeliť od zdrojového kódu touto formou. Je to nový experimentálny spôsob dokumentovania zdrojového kódu, ktorý ako pevne veríme prispeje k vyššej efektivite práce a to tým, že komentáre nebudú prekážať medzi príkazmi v zdrojovom kóde. Pri načítaní zdrojového súboru sa všetky komentáre prevedú na dokumentačné bloky a umiestnia sa na pravú stranu pracovnej plochy editora. Zároveň sa vytvorí grafické spojenie medzi nimi a blokmi, ku ktorým sa vzťahujú. Dokumentačný blok obsahuje funkciu `addArrow`, ktorá má v sebe konštruktor na vytvorenie šípky, ktorej parametrom sú počiatočný a koncový blok. Trieda `Arrow` dedí od triedy `QGraphicsLineItem`, kedže sme nechceli bloky spájať čiarami, ale šípkami, tak táto trieda je oproti pôvodnej obohatená o vykreslenie malého trojuholníka na konci čiari.

Pri ukladaní súboru funguje opačný systém. Pri ukladaní dokumentačného textu sa tento text umiestni medzi komentárové značky spolu s informáciou o pozícii a tento komentár sa umiestni za danú časť zdrojového kódu ku ktorému bol pripojený šípkou. V

prípade obrázku sa namiesto textu uloží cesta k obrázku a v prípade iných typov súborov sa tam umiestni cesta k nim.

7.5 Generovanie dokumentácie

Všetky bloky sa v editore zobrazujú na grafickej scéne, ktorú si môžeme predstaviť ako plátno, na ktorom sú jednotlivé bloky rozmiestnené. Pri generovaní dokumentácie zo zdrojového kódu sa využíva rozmiestnenie jednotlivých blokov na scéne. Používateľ si sám zvolí, kde sa jednotlivé bloky budú nachádzať, a ako bude výsledná dokumentácia vyzerať. Pre lepšiu predstavu o tom, ako budú jednotlivé bloky v dokumentácii rozmiestnené na stranách si používateľ zapne zobrazenie pomocných čiar, ktoré ohraničujú tlačovú oblasť scény. Všetky bloky, ktoré sa budú nachádzať mimo vyznačenej tlačovej oblasti nebudú vytlačené. Pokiaľ používateľ nechce vytlačiť niektorý blok, tak ho jednoducho umiestni mimo tlačovú oblasť a vytlačená dokumentácia nebude blok obsahovať.

7.5.1 Tlačenie PDF

Pre vygenerovanie dokumentácie vo forme PDF sa používa metóda `printPdf()`. Dokument PDF sa generuje priamo zo scény editora ohraničenej oblasťou tlače, ktorú si môže používateľ zobraziť tlačidlom *Printable area*. Scéna editora je tvorená triedou `DocumentScene`, ktorá je zdedená od triedy `QGraphicsScene`.

Pred generovaním PDF dokumentu je potrebné nastaviť tlačiareň, ktorá je obsluhovaná triedou `QPrinter`. Pre tlačiareň sa nastavuje rozlíšenie tlače, veľkosť papiera, výstupný formát dokumentu a názov a umiestnenie súboru, do ktorého sa celý generovaný dokument uloží. Názov a umiestnenie súboru si používateľ zvolí v dialógovom okne, ktoré je vytvorené pomocou triedy `QFileDialog`. Pre nastavenú tlačiareň sa potom zavolá obsluhujúca trieda `QPainter`, ktorá vykoná samotné vykreslenie scény na strany dokumentu. Pokiaľ výška scény presahuje veľkosť strany formátu A4, tak sa pri tlačení rozdelí scéna na menšie úseky, ktoré sa postupne vytlačia do jedného dokumentu. Pre vytlačenie jednej strany sa

používa metóda `render()`.

Kapitola 8

Záver

Projekt textového editora obohateného o grafické prvky (pracovný názov *TrollEdit*) je svojím zameraním v súčastnosti jedinečný a poskytuje používateľom zaujímavé možnosti doteraz nie bežné pre editory. Vďaka poznatkom z analýzy sme vytvorili niekoľko alternatív pre finálnu integráciu uvažovaných technológii, najmä s ohľadom na požadovanú multiplatformovosť.

V prvom semestri práce na projekte sme sa zamerali na analýzu a výber vhodných technológií pre implementáciu editora. Taktiež sme vypracovali návrh, v ktorom sme predstavili plánovanú funkcionalitu a spôsob akým ju dosiahnuť. Následne sme vytvorili niekoľko funkčných prototypov, ktoré zodpovedali modulom editora. Počas skúškového obdobia sa nám podarilo integrovať modul pre syntaktickú analýzu s premiestňovaním blokov. Tým sme dosiahli najdôležitejšiu funkcionalitu, ktorá odlišuje náš editor od ostatných, a ktorá bola základom pre vývoj ostatných modulov a funkcií (zvýrazňovanie syntaxe, dokumentačné bloky, tlač do PDF). Návrh sme niekoľkokrát prehodnotili, prípadne doplnili, keďže sme postupom času prichádzali na nové možnosti funkcionality alebo jednoduchšie spôsoby implementácie.

8.1 Testovanie

Z dôvodu experimentálnosti projektu testovanie prebiehalo len v rámci tímu, keďže sa po skončení projektu nejedná o finálny produkt pre zákazníka. Charakter testovania bol počas semestra nezáväzný, to znamená, že keď niekto doplnil funkcionalitu editora a zdrojové kódy nahral do repozitára, väčšinou poprosil ostatných o spätnú väzbu alebo bola táto funkcionalita prezentovaná a testovaná na spoločných stretnutiach (vo väčšine prípadov). Pri testovaní sme teda nepoužívali nijaké špecializované nástroje ani sme testy podrobne nedokumentovali. Komplexnejšie testovanie prebehlo pred konferenciou IIT.SRC. Vtedy sme si určili niekoľko scenárov použitia, vytvorili testovacie súbory so zdrojovými kódmi a otestovali funkčnosť a správanie editora. Výsledky testovania neboli vždy uspokojivé, ako problémová sa napríklad prejavila i animácia pri presúvaní blokov, ktorá značne spomaľovala editor pri práci s väčšími súbormi. Ďalšie komplexné testovanie, ktoré bude zároveň i posledným, prebehne na konci semestra pred súťažou TPcup, keďže ešte teraz dopĺňame a vylepšujeme funkcionalitu jednotivých modulov.

8.2 Nápady na ďalšie vylepšenie

Počas práce na tomto projekte sme sa snažili implementovať základné funkcie editora spolu s tými, ktoré robia náš editor jedinečným. Navrhli sme niekoľko vylepšení, ktoré by prispeli k väčšiemu komfortu používateľa a zefektívnili jeho prácu, avšak ktorých implementovanie si vyžaduje viac času, ako sme tomu boli schopní venovať. Predpokladáme, že v budúcnosti bude v tomto projekte pokračovať ďalší tím, ktorý by mohol implementovať tieto vylepšenia:

- Detekcia pachov kódu.
- Možnosti „undo“/ „redo“.
- Možnosť rozšírených nastavení priamo v editore.

- Možnosť presúvať priradenie šípkou na iný blok a vytvárania šípok nie len pri vytváraní dokumentačných blokov.

8.3 Nadobudnuté skúsenosti/vedomosti

Práca na projekte zmenila v mnohom náš pohľad na prácu v tíme, každého zaväzovala zodpovednosť za výsledky celého tímu, nielen za svoje. Postupom času sme sa naučili, ako spraviť prácu v time efektívnejšou, čo nám môže pomôcť pri ďalších iných projektoch. Osvojili sme si používanie nástrojov pre manažment tímu a manažment zdrojových kódov. Okrem toho sme sa zúčastnili konferencie IIT.SRC, čo bola pre väčšinu z nás tiež nová a cenná skúsenosť. Čo sa týka programátorských vedomostí, naučili sme sa pracovať s frameworkom *Qt*, ktorý sa ukázal ako veľmi pružný nástroj pre implementáciu editora. Okrem toho každý člen tímu získal nové vedomosti a prax aj pri práci na module, ktorý vytváral (syntaktická analýza – *Lua*, *LPeg*; tlač do PDF; atď.)

Zoznam použitej literatúry

[1] R. Ierusalimschy. LPeg – Parsing Expression Grammars For Lua. `http://www.inf.puc-rio.br/~roberto/lpeg/lpeg.html`. [posledný prístup 27.10.2009].

[2] R. Ierusalimschy. *Programming in Lua*. Lua.org, 2003. ISBN 85-903798-1-7.

[3] R. Ierusalimschy. A text pattern-matching tool based on Parsing Expression Grammars. *Softw. Pract. Exper.*, 39(3):221–258, 2009.

[4] Notepad++. `http://notepad-plus.sourceforge.net/uk/site.htm`. [posledný prístup 1.11.2009].

[5] QScintilla: What is QScintilla. `http://www.riverbankcomputing.co.uk/software/qscintilla/intro`. [posledný prístup 31.10.2009].

[6] Qt 4.0: Qt Reference Documentation (Open Source Edition). `http://doc.trolltech.com/4.0/index.html`. [posledný prístup 30.10.2009].

[7] Rich Text Format (RTF) Version 1.5 Specification. `http://www.biblioscape.com/rtf15_spec.htm`. [posledný prístup 1.11.2009].

[8] Scintilla and SciTE. `http://www.scintilla.org/SciTE.html`. [posledný prístup 1.11.2009].

[9] The Programming Language Lua. `http://www.lua.org`. [posledný prístup 27.10.2009].

Dodatok A

Ukážka gramatiky jazyka XML pre LPeg

```
-- Simple XML grammar

-- TODO:
--   improve grammar for attributes
--   possible attribute encoding in header
--   opening and closing symbols of elements - for highlight
--   start and end tags can be different at the moment...

-- important fields for Analyzer class
extension = "xml"
full_grammar = "document"
other_grammars = {
markup_element="in_markup_element",
}
paired = {"<", ">", "start_element", "end_element"}
selectable = {"xml_header", "markup_element", "unknown"}
multi_text = {"unknown"}

require 'lpeg'

--patterns
local P, R, S, V = lpeg.P, lpeg.R, lpeg.S, lpeg.V
--captures
```

```
local C, Ct, Cc = lpeg.C, lpeg.Ct, lpeg.Cc

-- nonterminal, general node
function N(arg)
return Ct(
Cc(arg) *
V(arg)
)
end

-- nonterminal, ignored in tree
function NI(arg)
return
V(arg)
end

-- terminal, text node
function T(arg)
return
N'whites'^-1 *
TP(arg) *
N'nl'^0
end

-- terminal, plain node without comments or whitespaces
function TP(arg)
return
Ct(C(arg))
end

-- ***  GRAMMAR  ****
local grammar = {"S",

-- ENTRY POINTS
document =
Ct(
Cc("document") *
N'nl'^0 *
N'xml_header'^-1 * N'markup_element' *
N'unknown'^-1 *-1),

in_markup_element =
Ct(
```

```
N'nl'^0 *
(N'markup_element')^0 *
N'unknown'^-1 *-1),

-- NONTERMINALS
xml_header = T"<?xml" * T"version=" * N'version_number' * T"?>",
version_number = T"\"" * NI'number' * T'.' * NI'number' * T"\"",
markup_element =
N'comment' +
N'empty_element' +
N'start_element' * (N'word'^1 + N'markup_element'^0) * N'end_element',
start_element = T"<" * N'markup_tag' * N'attribute'^0 * T">",
end_element = T"</" * N'markup_tag' * T">",
empty_element = T"<" * N'markup_tag' * T"/>",
attribute = N'attribute_key' * N'attribute_value',

-- TERMINALS
markup_tag = T((NI'letter' + S("_:")) * NI'markup_tag_char'^0),
word = T(NI'char'^1),
number = T(NI'digit'^1),
attribute_key = NI'attribute_word',
attribute_value = T'="' * NI'attribute_word'^1 * T'"',
attribute_word = T((P(1) - S("=\"<>/"))^1),
comment = T(P"<!--" * (1 - P"-->")^0 * P"-->"),

-- LITERALS
unknown = TP(P(1)^1), -- anything
whites = TP(S(" \t")^1),-- spaces and tabs
nl = S(" \t")^0 * TP(P"\r"^-1*P"\n"), -- single newline, preceding spaces
--are ignored

letter = R("az", "AZ"),
digit = R("09"),
char = P(1) - S("<>/"),
markup_tag_char = NI'letter' + NI'digit' + S(".-_:"),
}
-- ***  END OF GRAMMAR  ****

-- *** POSSIBLE GRAMMARS (ENTRY POINTS) ****
grammar[1] = "document"
document = P(grammar)
grammar[1] = "in_markup_element"
in_markup_element = P(grammar)
```

```
--**********************************************************************
-- TESTING - this script cannot be used by Analyzer.cpp when these lines
-- are uncommented !!!
-- dofile('default_grammar.lua')
-- test("../../input/input.xml", document)
```

Dodatok B

Ukážka konfiguračného súboru pre zvýrazňovanie syntaxe

```
cfg_keys = {"text_style", "comment_style", "keyword", "line_comment",
 "multi_comment", "string_constant", "character_constant", "number_constant",
 "label", "header_file", "funct_call", "assignment_operator",
 "binary_operator", "prefix_operator", "postfix_operator", "markup_tag",
 "attribute_key", "attribute_value", "block_style", "if_statement",
  "unknown", "preprocessor", "while_statement", "for_statement",
 "switch_statement", "funct_param", "block", "funct_definition"}

text_style = {target="text", family="courier", size="12", color="black",
bold="false", italic="false", underline="false"}
comment_style = {target="text", base_t="text_style", family="verdana",
 italic="true"}

keyword = {target="text", base_t="text_style", color="blue"}
line_comment = {target="text", base_t="comment_style", color="gray"}
multi_comment = {target="text", base_t="comment_style", color="gray"}
doc_comment = {target="text", base_t="comment_style", color="gray"}
string_constant = {target="text", base_t="text_style", color="darkmagenta"}
character_constant = {target="text", base_t="text_style", color="darkmagenta"}
number_constant = {target="text", base_t="text_style", color="orangered"}
label = {target="text", base_t="text_style", color="maroon"}
header_file = {target="text", base_t="text_style", color="lightpink"}
funct_call = {target="text", base_t="text_style"}
```

```
assignment_operator = {target="text", base_t="text_style"}
binary_operator = {target="text", base_t="text_style"}
prefix_operator = {target="text", base_t="text_style"}
postfix_operator = {target="text", base_t="text_style"}
markup_tag = {target="text", base_t="text_style", color="brown"}
attribute_key = {target="text", base_t="text_style", color="red"}
attribute_value = {target="text", base_t="text_style", color="blue"}

block_style = {target="block", hovered="white", hovered_border="lightblue",
 selected="blue", showing="blue"}

if_statement = {target="block", base_b="block_style", hovered="yellow",
hovered_border="red"}
unknown = {target="block", base_b="block_style", selected="red",
showing="red", hovered="red", hovered_border="red"}
preprocessor = {target="block", base_b="block_style"}
while_statement = {target="block", base_b="block_style", hovered="blue",
hovered_border="blue"}
for_statement = {target="block", base_b="block_style"}
switch_statement = {target="block", base_b="block_style"}
funct_param = {target="block", base_b="block_style"}
block = {target="block", base_b="block_style"}

funct_definition = {target="both", base_b="block_style", base_t="text_style",
 bold="true"}

-- TODO: struct, union...?
```

Dodatok C

Rozšírený abstrakt z IIT.SRC 2010

TrollEdit - New Way of Source Code Editing

Andrej FOGELTON*, Ondrej KALLO*, Peter ONDRUŠKA*
Martin PALO†, Jakub UKROP*

Slovak University of Technology
Faculty of Informatics and Information Technologies
Ilkovičova 3, 842 16 Bratislava, Slovakia
ufopak@googlegroups.com

Editing a source code is asignificant part of a programmer's job. Most of the time programmers modify and debug existing code rather than create new code; this refactoring of source code always takes a lot of time. *TrollEdit* accelerates the casual way of source code editing by providing users with graphic view of the inner structure and logic of programs. Additionally, main principles of literate programming are supported, meaning that source code documents can contain fully formatted documentation.

Block hierarchies and their visualization are the main ideas underlining this project. Source code written in any of textual programming language consists of number of lexical, syntactic and semantic units – logical "chunks of code" or blocks. Enriched language grammars implemented using the *Lua* based *LPeg* [1] pattern matching library are used to parse the source code into an abstract syntactic tree (AST). Support of any programming language or any formal language is limited only by availability of its grammar, which can be easily transformed into the *LPeg* syntax. Every node in the AST represents what we call a block.

By using block hierarchies based on AST rather than simple syntactic rules of a language, our editor can offer features not common in most source code editors. These affect refactoring tasks such as fast selection of blocks without need to select blocks by cursor, enhanced drag-and-drop within a program by snapping to available spots or customized code folding by replacing folded blocks by desired summary information etc.

Since everything within a program is a block, some may end up containing too many sub-blocks or too small to be useful e.g. a single semicolon. Therefore, it is essential for users to be able to control and shift the level on which the current editing is going

---

* Master study programme in field: * Software Engineering,† Information Systems
Supervisor: Peter Drahoš, Institute of Applied Informatics, Faculty of Informatics and Information Technologies STU in Bratislava

IIT.SRC 2010, Bratislava, April 21, 2010, pp. 1–2.

on according to their momentary needs. Figure 1 illustrates the concept of the block hierarchy on a sample C code fragment.

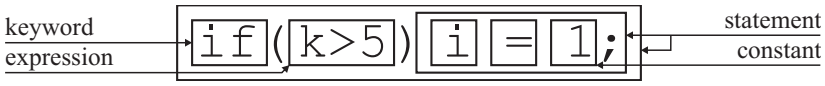


Figure 1. Example of block recognition

TrollEdit is based on the graphics view framework provided by *Qt* toolkit. Blocks can contain graphic items which can be have different shapes, font or even images, rich text, UML diagrams etc. Every block supports layout, movement, drag-and-drop and text editing features. By providing constraints on where certain items can be dropped we can maintain a structure of the document, since it is not desirable to be able to drop anything anywhere. Blocks have to be moved with their descendants and resized according to size and layout of their descendant blocks. A programmer can fold blocks, which are naturally bound (functions, cycles, etc.) or define own blocks using menu items.

By relying on the former mentioned display and layout capabilities of the block layout system, *TrollEdit* can be used as a document editor for writing documentation directly into the code. Similarly to standard language comments these blocks will be omitted from the executable code, so code and documentation can coexist without any side effects. This was inspired by the literate programming idea conceived by Donald E. Knuth [2]. As opposed to the original approach no special tags or macros are necessary to separate the source code from documentation. To achieve better readability ”documentation blocks” support full text formatting (font size, bold, italics, colour, etc.). Moreover, the user can choose to generate different outputs consisting only of the selected code and documentation blocks.

TrollEdit is an aspiring *OpenSource* project with the specific aim to create a multi-platform editor based on the *Qt* toolkit. It is designed with extensibility and flexibility in mind, for example new languages can be added at any time by providing their *LPeg* grammars. When combined with a debugger blocks can also hold various helpful data, for example block execution count. Potential visual effects in code blocks are not limited to the standard syntax highlighting as more visual feedback can be achieved e.g. colour coding algorithm cycles according to depth or complexity.

References

[1] Ierusalimschy, R.: A text pattern-matching tool based on Parsing Expression Grammars. *Softw. Pract. Exper.*, 2009, vol. 39, no. 3, pp. 221–258.

[2] Knuth, D.E.: *Literate Programming*, 1992, Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

Dodatok D

Poster z IIT.SRC 2010

TrollEdit - New Way of Source Code Editing

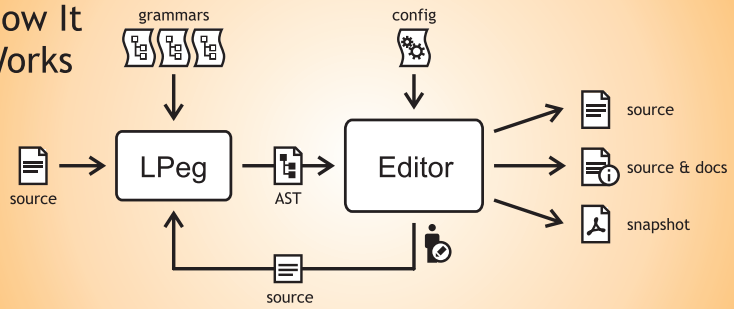
Andrej Fogelton, Ondrej Kallo, Peter Ondruška, Martin Palo, Jakub Ukrop
Supervisor: Peter Drahoš

Features

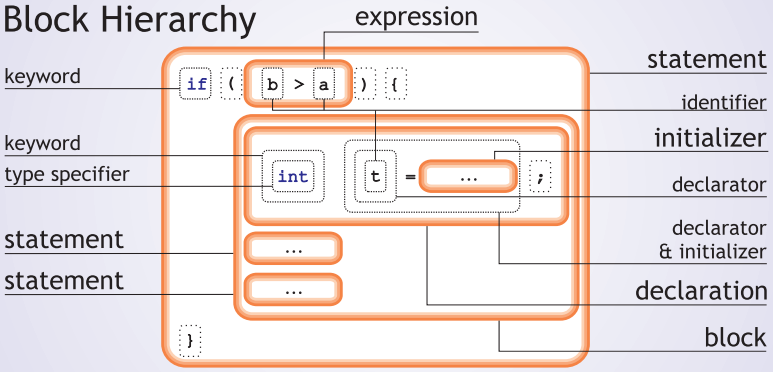
- code structure visualization
- support of any formal language (programming, markup, ...)
- configurable syntax highlighting
- drag & drop for fast refactoring
- images and formatted text in comments
- custom folding captions
- various outputs

- technologies:
  Qt – multi-platform application framework
  LPeg – pattern-matching library for Lua

How It Works

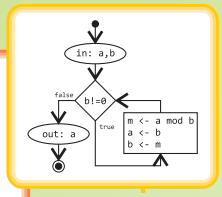


Block Hierarchy



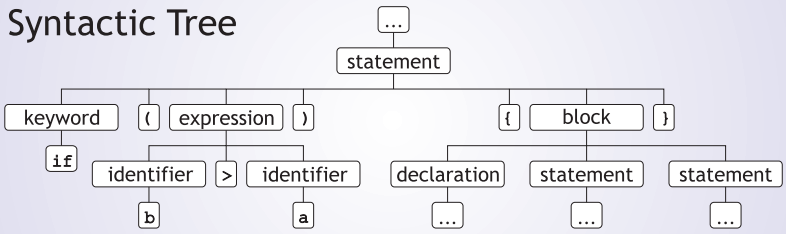
Versatile Comments

```
int gcd(int a, int b) {
    if (b > a) {
        int t = a;
        a = b;
        b = t;
    }
    while (b != 0) {
        int m = a % b;
        a = b;
        b = m;
    }
    return a;
}
```



TODO: make recursive function

Syntactic Tree



Customizable Folding

```
int gcd(int a, int b) {
    switch values if b > a...
    while (b != 0) {
        int m = a % b;
        a = b;
        b = m;
    }
    return a;
}
```

Grammar

statement = 'if' '(' expression ')' statement ['else' statement] |
'{' block '}' | ...
block = { declaration | statement }

Selection & Movement

```
int gcd(int a, int b) {
    if (b > a) {
        int t = a;
        a = b;
        b = t;
    }
    while (b != 0) {
        int m = a % b;
        a = b;
        b = m;
    }
    return a;
}
```

```
int gcd(int a, int b) {
    while (b != 0) {
        int m = a % b;
        a = b;
        b = m;
    }
    return a;
}
```

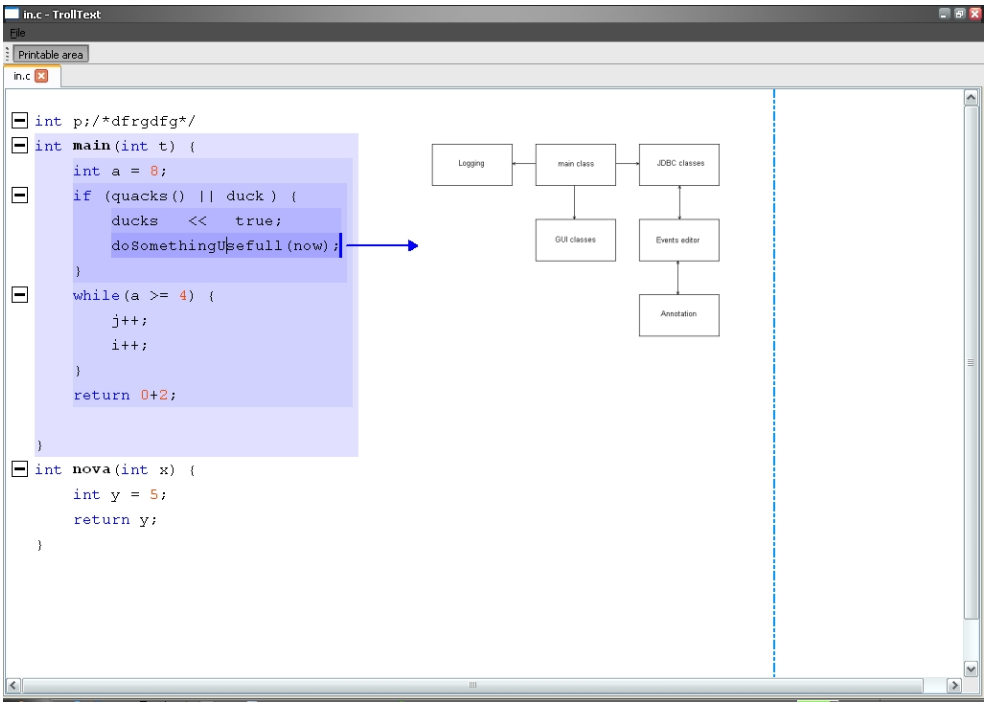
```
int gcd(int a, int b) {
    while (b != 0) {
        if (b > a) {
            int t = a;
            a = b;
            b = t;
        }
        int m = a % b;
        a = b;
        b = m;
    }
    return a;
}
```

Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies, Slovak University of Technology
ufopak@googlegroups.com
Ilkovičova 3, 842 16 Bratislava, Slovakia

Dodatok E

Ukážka tlače

Editor má natívnu podporu tvorby dokumentácie. Obsahuje možnosť vytlačiť viditeľnú časť zdrojového kódu a príslušných dokumentačných blokov (nie sú práve skryté), ktoré sa nachádzajú v ohraničeniach oblasti tlače.

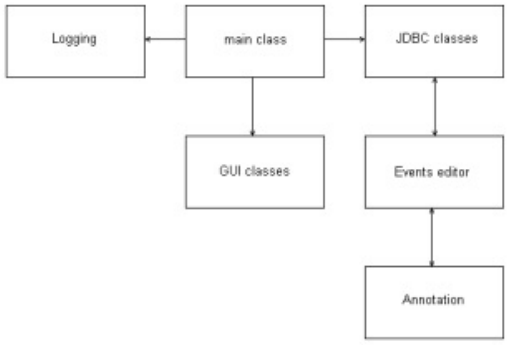


Obr. E.1: Ukážka tlače v editore.

Ukážka vytlačeného PDF sa nachádza na ďalšej strane.

```
int p;/*dfrgdfg*/
int main(int t) {
    int a = 8;
    if (quacks() || duck ) {
        ducks   <<   true;
        doSomethingUsefull(now);
    }
    while(a >= 4) {
        j++;
        i++;
    }
    return 0+2;

}
int nova(int x) {
    int y = 5;
    return y;
}
```



Dodatok F

Používateľská príručka

Táto časť obsahuje potrebné informácie pre inštaláciu a používanie editora *TrollEdit*.

F.1 Inštalácia

F.1.1 Windows

Pre spustenie inštalácie je potrebné spustiť súbor `Setup.exe`, ktorý je možné stiahnuť si zo stránky projektu. Potom stačí nasledovať jednotlivé kroky v rámci inštalátora. Po úspešnom ukončení bude *TrollEdit* nainštalovaný a pripravený na použitie.

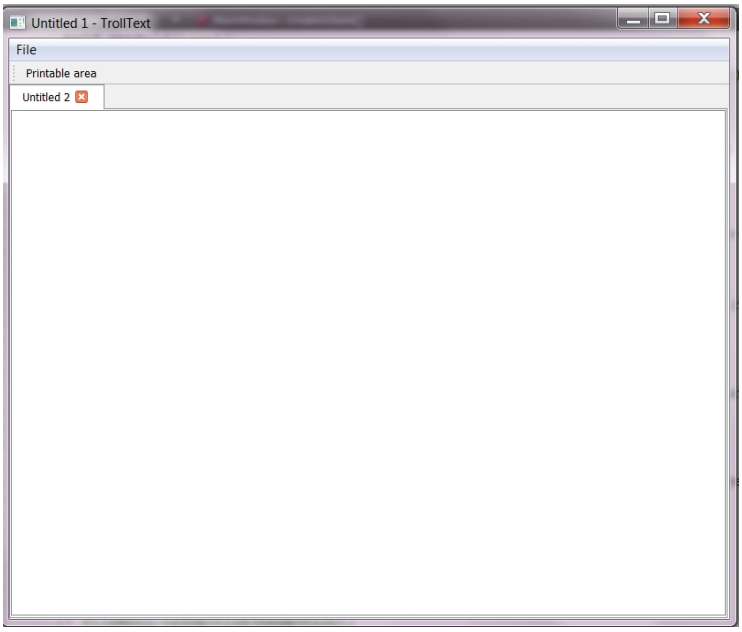
F.1.2 Linux

Používateľ si zo stránky projektu stiahne súbor `TrollEdit.tar.gz`. Príkazom `tar xvzf TrollEdit.tar.gz` rozbalí jeho obsah do podadresára. Potom musí zmeniť aktuálny adresár na daný podadresár a spustiť príkaz `make`, ktorý zabezpečí vytvorenie vykonateľného súboru.

F.2 Práca s editorom

F.2.1 Nový dokument

Po otvorení editora je automaticky pripravený nový dokument. V prípade, že používateľ chce potom začať prácu na novom dokumente, treba otvoriť menu *File* → *New*, prípadne použiť klávesovú skratku `Ctrl+N`.



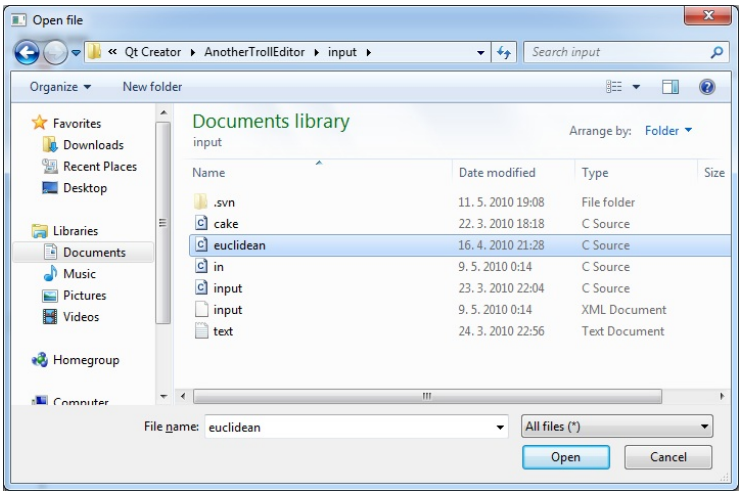
Obr. F.1: Nový súbor otvorený v editore.

F.2.2 Ukladanie a otváranie

Na otvorenie existjúceho dokumentu, treba otvoriť menu *File* → *Open... (Ctrl+O)*. Otvorí sa dialóg pre výber súboru (obr. F.2). Pomocou neho si používateľ nájde a zvolí želaný súbor a kliknutím na tlačidlo *Open* ho otvorí do editora.

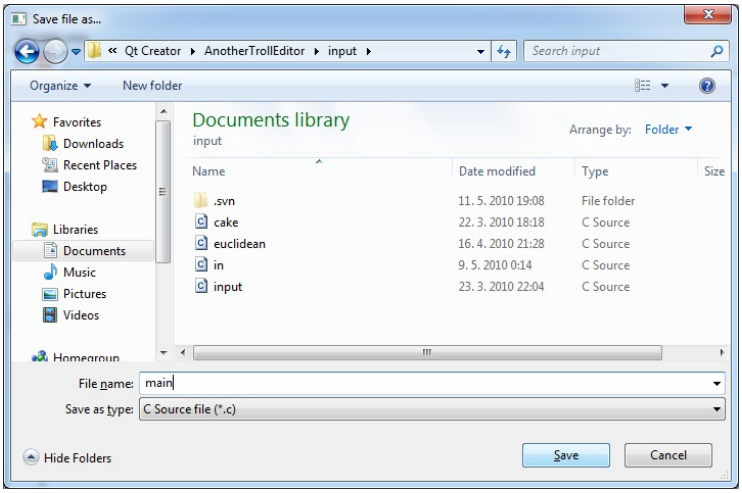
V prípade ukladania môžu nastať dve situácie:

1. Používateľ otvorí menu *File*→ *Save* (Ctrl+S). V prípade, že je otvorený existujúci dokument, ktorý nebol modifikovaný, jeho obsah sa uloží. Inak sa postupuje, akoby používateľ zvolil akciu *Save as...*



Obr. F.2: Dialog Otvoriť súbor.

2. Používateľ z menu vyberie akciu *File*→ *Save as...*. Otvorí sa dialóg výberu (obr. F.3). Používateľ si vyberie miesto, kam súbor chce uložiť, a taktiež zvolí aj želaný názov. Po kliknutí na *Save* je súbor s daným menom uložený.

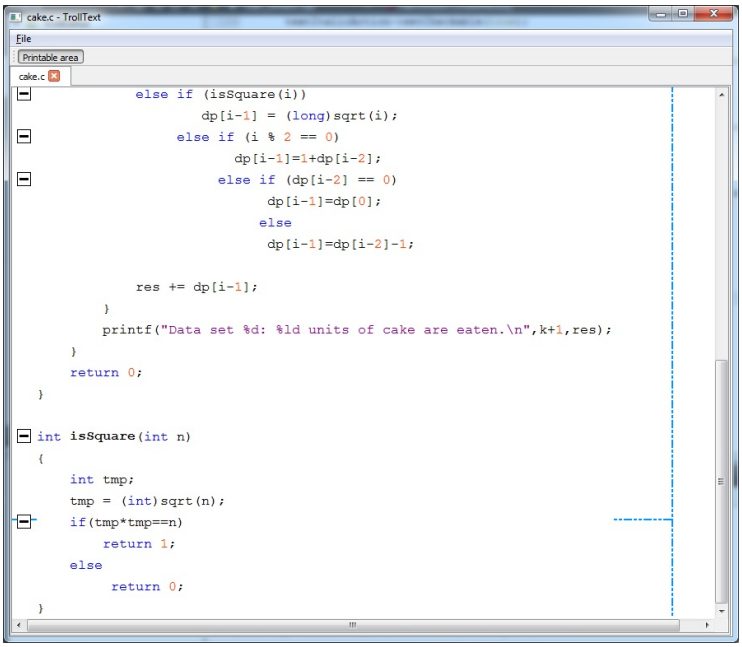


Obr. F.3: Dialog Uložiť súbor.

F.2.3 Tlačenie dokumentu

TrollEdit v súčasnej dobe podporuje export obsahu scény do PDF dokumentu. Aby používateľ mal možnosť získať náhľad, ako bude scéna rozdelená, aby sa zmestila na jednotlivé

strany rozmerov A4, je na paneli nástrojov umiestnené tlačidlo *Printable area*. Po jeho stlačení sa na scéne objavia čiary, ktoré ju rozdeľujú na oblasti umiestňované na jednotlivé strany generovaného PDF. Ako to vyzerá v praxi je možné vidieť na obrázku F.4.

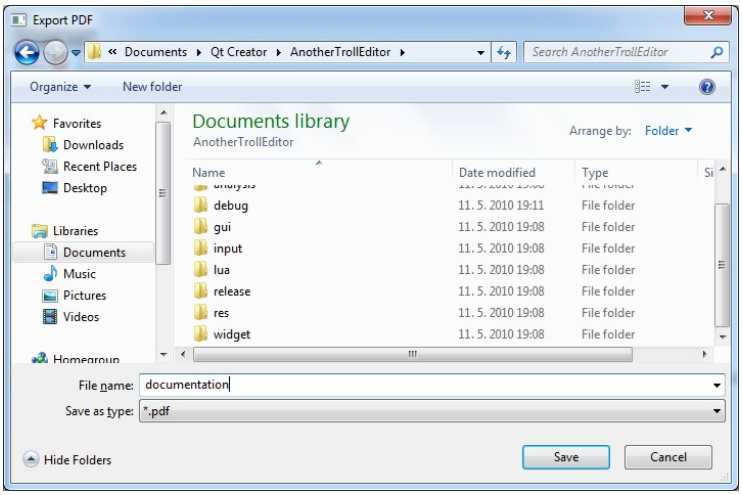


Obr. F.4: Zobrazené pomocné čiary na scéne editora.

Pre samotný export musí používateľ zvoliť akciu menu *File*→ *Print PDF* (Ctrl+P). Otvorí sa dialóg (obr. F.5), pomocou ktorého si používateľ zvolí kam chce vygenerovaný dokument uložiť a aký názov mu má byť pridelený. Po stlačení tlačidla *Save* bude príslušné PDF vygenerované.

F.2.4 Presun blokov

Ak chce používateľ presunúť blok, musí nad neho presunúť kurzor, stlačiť ľavé tlačidlo myši a držať ho. Kurzor potom treba presunúť nad želané miesto, kam má byť presúvaný blok umiestnený. Po pustení ľavého tlačidla myši je daný blok presunutý na vybrané miesto.



Obr. F.5: Dialog Vytlačiť súbor.

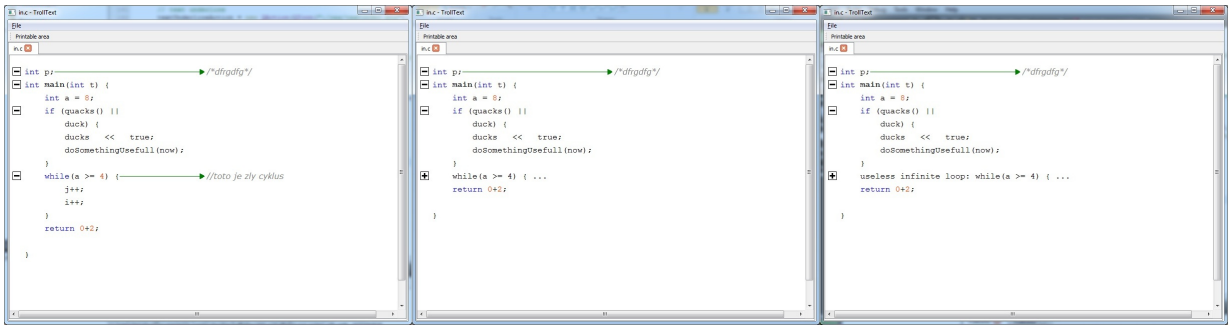
F.2.5 Plávajúce komentáre

Ak používateľ chytí obrázok, potiahne ho nad scénu editora a tam ho pustí, umiestni sa tento do samostatného dokumentačného bloku. Dokumentačné bloky je možné voľne presúvať po scéne, preto názov *plávajúce komentáre*. Aby však bolo jasné, k čomu sa taký komentár viaže, je tu šípka ukazujúca na blok, ku ktorému patrí. Táto šípka sa priradí automaticky v prípade, že používateľ má označený nejaký blok v momente vytvorenia dokumentačného bloku. Obrázky bude možné zmenšovať a zväčšovať intuitívne kurzorom myši, ktorý sa zmení na obojstrannú šípku pri prechode cez okraj dokumentačného bloku. Ľavým dvojklikom na obrázok je možné tento obrázok otvoriť v externom prednastavenom editore.

Kvôli zjednodušeniu práce a udržiavania všetkých súborov na jednom mieste existuje možnosť pretiahnuť na pracovnú plochu i súbory iného typu ako obrázku, v takomto prípade sa vytvorí ikonka s názvom daného súboru na ploche editora. Dvojklikom na túto ikonku sa otvorí daný súbor v externom editore.

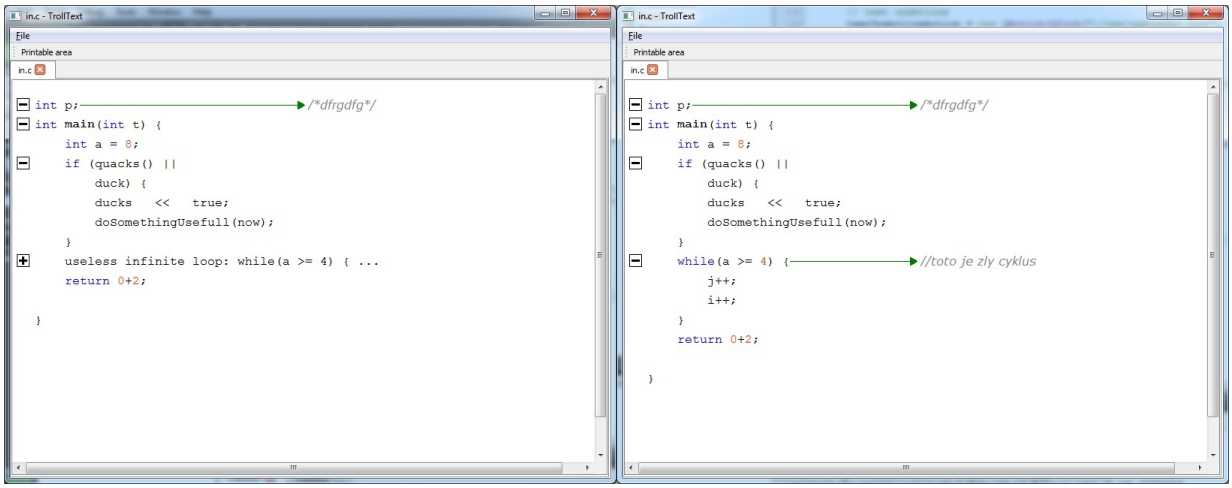
F.2.6 Skrývanie blokov

Po kliknutí na ikonu so symbolom „-“, ktorá je na ľavom okraji vedľa bloku, sa obsah bloku skryje, pričom viditeľný zostane len prvý riadok. Keďže nie vždy je tento prvý riadok nápomocný v určovaní, čo sa v danom bloku deje, je možné prepísať text tohoto riadku a pridať si tak ľubovoľnú používateľovi zrozumiteľnú poznámku. Ako to vyzerá je možné vidieť na obr. F.6



Obr. F.6: Ukážka funkcionality skrývania blokov s nahradením vlastným textom.

V prípade, že niektorý blok je skrytý, ikona vedľa neho zmení symbol na „+“. Po kliknutí na tento symbol blok opäť rozbalí svoj obsah (obr. F.7)



Obr. F.7: Rozbalenie skrytého bloku s vlastným textom.

F.2.7 Programovanie v editore

V tejto funkcionalite sa *TrollEdit* veľmi neodlišuje od ostatných editorov, písanie v ňom funguje ako v inom editore. Základnou odlišnosťou je bloková reprezentácia a možnosť označovania týchto blokov. Dôležitou skutočnosťou pre programátora je fakt, kedy sa vykonáva opätovná analýza zdrojového kódu po jeho zmene. Táto analýza je časovo náročnejšia a preto sa spúšťa iba v čase prechodu písania na ďalší riadok.