FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Překladač vývojových schémat z Javy do C/C++
Student:	Marek Tornóci
Vedoucí:	doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Cílem práce je navrhnout a implementovat překladač (převodník) již navržených a implementovaných vybraných metod v Javě do jazyka C/C++. Metody představují zejména různé operace při zpracování obrazu.

1) Seznamte se s vývojovým prototypovacím prostředím, které umožňuje graficky (Drag & Drop) umisťovat na plochu bloky, ty propojovat do funkčních schémat, která jsou reprezentována pomocí datových struktur.
2) Navrhněte překladač (převodník), který umožní přeložit navržené schéma do jazyka C/C++.
3) Navržený překladač (převodník) implementujte v jazyce Java a ověřte jeho funkčnost na vzorových příkladech vytvořených ve vývojovém prototypovacím prostředí. Výsledný vytvořený kód v C/C++ musí být přeložitelný C/C++ překladačem do spustitelného kódu a musí fungovat stejně jako původní schéma navržené ve vývojovém prototypovacím prostředí.
4) Diskutujte možná omezení a možnosti navrženého a implementovaného překladače (převodníku).

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 2. února 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalárska práca

Překladač vývojových schémat z Javy do C/C++

Katedra softwarového inžinírství
Vedúci práce: doc. RNDr. Ing. Marcel Jiřina, Ph.D.

15. mája 2018

Poďakovanie

V prvom rade by som chel poďakovať vedúcemu práce, doc. RNDr. Ing. Marcelovi Jiřinovi, Ph.D. za konzultovanie bakalárskej práce a ochotu pri riešení problémov, ktoré sa počas práce vyskytli. Ďalej by som veľmi rád poďakoval rodine a priateľom, ktorí ma počas štúdia podporovali. Na záver by som chcel poďakovať Ing. Jakubovi Novákovi za pomoc a poskytnutie informácií ohľadom prostredia Surmon.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 15. mája 2018 ……

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na túto prácu

Tornóci, Marek. *Překladač vývojových schémat z Javy do C/C++*. Bakalárska práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Táto práca sa zaoberá návrhom a implementáciou aplikácie, ktorá umožní preložiť vybrané, už naimplementované bloky v jazyku Java do jazyka C++. Tieto vybrané bloky predstavujú rôzne operácie pri spracovaní obrazu, ktoré medzi sebou dokáže užívateľ ľubovoľne kombinovať v interaktívnom prostredí Surmon. Výsledkom samotnej práce je naimplementovaná aplikácia v jazyku Java integrovaná v danom prostredí.

Klíčová slova prekladač vývojových diagramov, Java, spracovanie obrazu, generátor C++ kódu, OpenCV

Abstract

This work deals with a design and implementation of the application, which will be able to translate chosen already implemented blocks in the Java programming language to the C++ programming language. Those methods represent various image processing operations which can a user freely combine in the interactive environment called Surmon. The result of the thesis is implemented application in the Java programming language integrated in the Surmon environment.

Keywords translator of flowcharts, Java, image processing, generator of C++ code, OpenCV

Obsah

Zoznam obrázkov

Úvod

V dnešnej dobe sa spracovanie obrazu dostáva viac a viac do popredia, kvôli čomu vznikli v tejto oblasti mnohé nástroje, ktoré túto prácu uľahčujú. Jedným z takýchto nástrojov je program s názvom Surmon. Jedná sa o grafické prototypovacie prostredie, ktorého úlohou je umožniť užívateľovi vytvárať algoritmy pomocou skladania schém formou drag and drop. Čo momentálne dané prostredie neumožňuje je generovanie ekvivalentného kódu napísaného v jazyku C++, ktorý by po skompilovaní poskytoval rovnaké výsledky ako samotná schéma. Tento kód by potom mohol užívateľ využiť napríklad nahratím do továrenskej kamery, ktorá by mu snímaním okolia poskytovala vstupný obraz.

Cieľom tejto bakalárskej práce je navrhnúť a implementovať takúto aplikáciu, ktorá dokáže k ľubovoľne vytvorenej schéme pozostávajúcej z určitej podmnožiny blokov prostredia Surmon generovať ekvivalentný C++ kód.

Na začiatku práce popisujem samotné prototypovacie prostredie Surmon, popisujem vytváranie jeho blokových schém a rozoberám štruktúru vstupného súboru pre moju aplikáciu, ktorý prostredie k jednotlivým schémam generuje.

V ďalšej časti s názvom Analýza a návrh analyzujem jednotlivé funkčné a nefunkčné požiadavky, ktoré musí aplikácia splňovať a podrobne vysvetľujem návrh aplikácie vhodne doplnený diagrammi pre lepšie pochopenie.

V časti implementácia najprv popisujem technológie, ktoré som sa rozhodol využiť pri implementácií samotnej aplikácie, popisujem štruktúru balíkov a potom rozoberám najdôležitejšie triedy aplikácie a ich jednotlivé metódy. Na koniec bolo nutné aplikáciu otestovať, či funguje správne a naozaj poskytuje kód, ktorý by po skompilovaní dával rovnaké výsledky ako jednotlivé schémy. V poslednej sekcii s názvom Testovanie opisujem postup pri testovaní aplikácie.

V poslednej kapitole rozoberám niektoré obmedzenia samotnej aplikácie.

Kapitola 1

Cieľ práce

Cieľom tejto bakalárskej práce je navrhnúť a implementovať aplikáciu, ktorá dokáže generovať ekvivalentný C++ kód k vytvorenej schéme pozostávajúcej z podmnožiny blokov prostredia Surmon. Tento výsledný kód musí byť skompilovateľný a poskytovať rovnaké výsledky ako navrhnutá schéma v prostredí Surmon. Aplikácia bude napísaná v jazyku Java a integrovaná v prostredí Surmon. Posledný cieľ sa týka rozšíriteľnosti aplikácie. Samotná aplikácia by mala byť v budúcnosti pomerne ľahko rozšíriteľná o preklad ďalších blokov.

Kapitola 2

Zoznámenie sa s prostredím Surmon

V tejto kapitole najprv vysvetlím prostredie Surmon, popíšem jeho blokové schémy a nakoniec vysvetlím štruktúru vstupného súboru pre moju aplikáciu, ktorý prostredie k jednotlivým schémam generuje.

2.1 Surmon

Jedná sa o vývojové prototypovacie prostredie napísané v jazyku Java pozostávajúce z mnohých modulov, ktorého úlohou je umožniť užívateľom vytvárať algoritmy pomocou vytvárania schém. Tieto schémy sa skladajú z blokov, ktoré sú medzi sebou prepojené hranami. Bloky poskytujú veľké množstvo rôznych funkcií, ale predovšetkým predstavujú rôzne operácie pri spracovaní obrazu. Schémy sú organizované do projektov, kde je v rámci jedného projektu možné vytvoriť ľubovoľný počet schém.

2.2 OpenCV

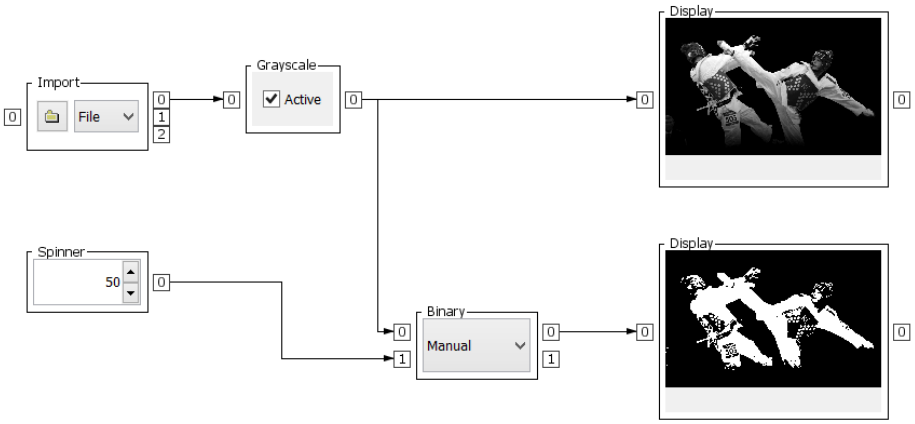
OpenCV je najpopulárnejšia voľne dostupá knižnica pre počítačové videnie a spracovanie obrazu. Táto knižnica je napísaná v jazyku C/C++ a beží na operačných systémoch Linux, Windows a Mac OS X. Jedným z cieľov OpenCV je poskytnúť jednoduchú infraštruktúru počítačového videnia, ktorá pomáha ľudom rýchlo vytvárať pomerne sofistikované aplikácie. [1] OpenCV obsahuje viac ako 500 naimplementovaných funkcií, ktoré slúžia pre spracovanie obrazu. Túto knižnicu využíva aj prostredie Surmon, ktoré k tomu využíva wrapper s názvom JavaCV.

2.3 Popis blokových schém

Blok predstavuje konkrétnu operáciu nad dátami, napríklad algoritmus z knižnice OpenCV, ktorý dáta transformuje, prípadne dáta načíta, zobrazí, alebo ich uloží v určitom formáte na disk.

Každý blok má nejaký počet konektorov. Konektory sa rozdelujú na vstupné a výstupné. Slúžia k prepojeniu blokov a poskytujú, respektíve príjmajú určité typy dát. Konektory, ktoré nie sú kompatibilné, inými slovami v sebe neuchovávajú rovnaký typ dát, nie je možné prepojiť. Z jedného výstupného konektoru môže viesť neobmedzený počet hrán do vstupných konektorov iných blokov. Do vstupného konektoru môže naopak viesť iba jedna konkrétna hrana z nejakého výstupného konektoru.

Vytvorenú schému je možné aktivovať, čím jednotlivé bloky danej schémy dáta najprv načítajú zo vstupných konektorov, spracujú a následne ich uložia na výstupné konektory, odkiaľ sa ďalším blokom pošlú na vstupné konektory, s ktorými sú prepojené. Samotný blok začne dáta transformovať až v momente, kedy dáta spracovali všetky bloky, na ktorých je závislí.



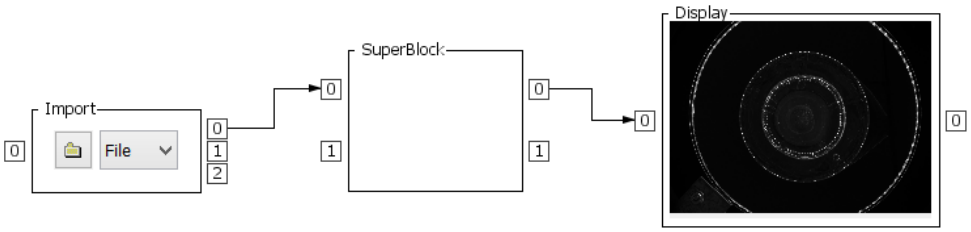
Obr. 2.1: Ukážka vytvorenej schémy po aktivacii.

Väčšina blokov má nastaviteľné vlastnosti, ktoré ovplivňujú spracovanie dát. Typicky sa jedná o argumenty pre funkcie z knižnice OpenCV, alebo rôzne operačné módy, podľa ktorých sa blok rozhoduje akú metódu použiť. Bloky môžu mať viacero vstupných, či výstupných dát, čo sa pozná podľa počtu vstupných, respektíve výstupných konektorov.

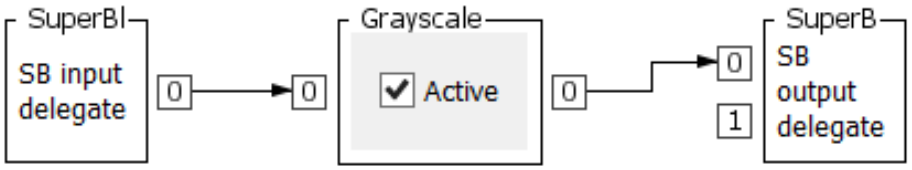
2.3.1 SuperBlock

Superblock je špeciálny blok, ktorý predstavuje jednu celú schému, ktorá je umiestnená v iných schémach v podobe bloku. Vďaka tomuto bloku nie je nutné mať všetky bloky v jednej schéme a veľmi často ho využíva blok s názvom Iterator. Každý takýto blok v sebe obsahuje paletu, ktorá sa zobrazí po jeho rozkliknutí. Táto paleta vždy začína blokom SuperBlockInputDelegate a končí blokom SuperBlockOutputDelegate.

- **SuperBlockInputDelegate** - Blok, ktorý sa vždy nachádza v schéme superbloku. Obsahuje iba výstupné konektory, na ktorých má rovnaké dáta, ako má na svojich vstupných konektoroch jeho superblok, ktoré následne deleguje ďalej.
- **SuperBlockOutputDelegate** - Blok, ktorý sa vždy nachádza v schéme superbloku. Obsahuje iba vstupné konektory, z ktorých dáta deleguje výstupným konektorom jeho superbloku.



Obr. 2.2: Ukážka schémy, ktorá obsahuje superblok.



Obr. 2.3: Ukážka vnútra superbloku z obrázku 2.2.

2.4 Dokument schémy

Ku každej schéme je zvlásť generovaný súbor vo formáte XML, ktorý je uložený v adresáry samotného projektu s názvom danej schémy. V tomto súbore je definovaná celková štruktúra schémy. Je v ňom uložený každý blok schémy spolu s jeho vlastnosťami, konektormi a celkovej štruktúry prepojenia týchto konektorov. Pri modifikácii schémy sa po uložení tieto zmeny prejavia aj v danom súbore.

Rozbor dôležitých častí súboru:

1. **Element <node></node>** – Element reprezentujúci konkrétny blok prostredia Surmon. Obsahuje jediný atribút s názvom `class`, v ktorom je uložený názov triedy, ktorá v prostredí Surmon blok implementuje. Obsah tohto atribútu je unikátny a udáva nám názov elementu, podľa ktorého vieme identifikovať, o aký blok sa jedná. Samotné telo elementu v sebe obsahuje iné elementy, ktorých obsah nejakým spôsobom udáva jeho vlastnosti a konektory.

2. **Element <pin></pin>** – Element reprezentujúci konektor. Obsahuje dôležitý atribút s názvom `direction`, ktorý nesie informáciu o tom, či sa jedná o vstupný alebo výstupný konektor. Tento element býva obsiahnutý v tele elementu `<node></node>`.

3. **Element <property></property>** – Element uchovávajúci nejakú vlastnosť bloku alebo konektoru. Obsahuje atribút s názvom `name`, ktorého hodnota predstavuje názov vlastnosti. Hodnota elementu býva často argument pre funckiu z OpenCV alebo nejaký mód, podľa ktorého sa volá konkrétna funkcia.

4. **Element <edge></edge>** – Ide o element reprezentujúci hranu. Nachádza sa v tele elementu reprezentujúceho výstupný konektor. Hodnota jeho atribútu `source` nám udáva `uuid` výstupného konektoru, z ktorého do neho vedie hrana.

Súbor začína počiatočným tagom `<obbb>` a končí koncovým tagom `</obbb>`, ktoré spolu vytvárajú element, ktorý obsahuje všetky elementy v súbore. Všetky elementy, ktoré reprezentujú bloky nachádzajúce sa na nejakej palete, sú uložené v tele elementu s názvom superblok.

```
<?xml version="1.0" encoding="UTF-8"?>
<obbb>
  <info>
    <user>Marek</user>
    <date>Tue Apr 24 14:27:48 CEST 2018</date>
  </info>
  <node class="org.obbb.blocks.SuperBlock">
    <node class="org.obbb.ImageProcessing.Blocks.Enhance.Morfology">
      <pin direction="input" id="in0" index="0" type="org.opencv.core.Mat">
        <property class="java.lang.String" name="tooltip">
          <![CDATA[<br>Input image]]>
        </property>
        <property class="java.util.UUID" name="uuid">
          <![CDATA[823882b1-1ee5-4032-8191-fd7b49af18b6]]>
        </property>
        <property class="java.lang.Class" name="blockClass">
          <![CDATA[class org.obbb.model.Pin]]>
        </property>
      </pin>
      <pin direction="input" id="in1" index="1" type="org.opencv.core.Mat">
        <property class="java.lang.String" name="tooltip">
          <![CDATA[<br>Structuring element (optional)]]>
        </property>
        <property class="java.util.UUID" name="uuid">
          <![CDATA[c82451c5-842b-441e-9574-7376fa70565c]]>
        </property>
        <property class="java.lang.Class" name="blockClass">
          <![CDATA[class org.obbb.model.Pin]]>
        </property>
      </pin>
      <pin direction="output" id="out0" index="0" type="org.opencv.core.Mat">
        <property class="java.lang.String" name="tooltip">
          <![CDATA[<br>Output image]]>
        </property>
        <property class="java.util.UUID" name="uuid">
          <![CDATA[47c34c16-885d-4db7-82fa-041b58504900]]>
        </property>
        <property class="java.lang.Class" name="blockClass">
          <![CDATA[class org.obbb.model.Pin]]>
        </property>
      </pin>
      <property class="java.util.UUID" name="uuid">
        <![CDATA[e659cc7d-cfed-49a1-a4b2-2824b8b5971f]]>
      </property>
      <property class="java.lang.Class" name="blockClass">
        <![CDATA[class org.obbb.ImageProcessing.Blocks.Enhance.Morfology]]>
      </property>
      <property class="org.obbb.ImageProcessing.Blocks.Enhance.Morfology.Operation"
                name="operation">
        <![CDATA[Gradient]]>
      </property>
      <property class="int" name="repetition"><![CDATA[2]]></property>
      <property class="int" name="elemSize"><![CDATA[4]]></property>
    </node>
  </node>
</obbb>
```

Z ukážky XML súboru je možné vyčítať, že v schéme, ktorej tento súbor odpovedá, sa nachádza jediný blok s názvom Morfology, ktorý obsahuje dva vstupné konektory spolu s jedným výstupným konektorom. Ďalej sú vidieť jeho vlastnosti `operation`, `repetition`, `elemSize`, ktoré nejakým spôsobom ovplivňujú spracovanie dát týmto blokom.

2.4.1 Súbor superbloku

Každý superblok predstavuje jednu celú schému, a preto je mu generovaný vlastný XML súbor, v ktorom je jeho štruktúra zachytená.

V prípade, že sa v súbore nachádza element `<node></node>`, ktorý predstavuje superblok prostredia Surmon, je v ňom uložený názov jeho XML súboru ako hodnota elementu `<property></property>`, ktorého absolútna cesta odpovedá absolútnej ceste projektu, v ktorom je uložený.

```
<?xml version="1.0" encoding="UTF-8"?>
<obbb>
   <node class="org.obbb.blocks.SuperBlock">
      <node class="org.obbb.blocks.SuperBlock">
         <property class="java.lang.String" name="file">
            <![CDATA[NestedBlock.o5b]]>
         </property>
      </node>
   </node>
</obbb>
```

Pre zjednodušenie sú z XML súboru vymazané všetky ostatné dáta, aby bolo vidieť iba `<property></property>` element, ktorý v sebe nesie názov XML súboru tohto superbloku.

Každý XML súbor takéhoto superbloku potom začína elementom, ktorý reprezentuje blok s názvom SuperBlockInputDelegate a končí elementom, ktorý reprezentuje blok s názvom SuperBlockOutputDelegate.

```
<?xml version="1.0" encoding="UTF-8"?>
<obbb>
   <node class="org.obbb.blocks.SuperBlock">
      <node class="org.obbb.blocks.SuperBlockInputDelegate">
         <pin direction="output" id="out0" index="0" type="java.lang.Object">
            <property class="java.util.UUID" name="uuid">
               <![CDATA[fa12cdd0-4a87-4af5-87b7-1416db655cf0]]>
            </property>
            <property class="java.lang.Class" name="blockClass">
               <![CDATA[class org.obbb.model.Pin]]>
            </property>
         </pin>
         <property class="java.util.UUID" name="uuid">
            <![CDATA[51e2936f-5e29-4e1b-b74e-bb697fdbae40]]>
         </property>
      </node>
      <node class="org.obbb.blocks.SuperBlockOutputDelegate">
         <pin direction="input" id="in0" index="0" type="java.lang.Object">
            <property class="java.util.UUID" name="uuid">
               <![CDATA[6eae5897-0ba6-4abd-b65d-a1bc20d1b20c]]>
            </property>
            <property class="java.lang.Class" name="blockClass">
               <![CDATA[class org.obbb.model.Pin]]>
            </property>
            <property class="boolean" name="duplicateOnConnect">
               <![CDATA[true]]>
            </property>
         </pin>
         <property class="java.util.UUID" name="uuid">
            <![CDATA[fe81ad22-192c-4cac-b0bb-91e28a0e39b9]]>
         </property>
      </node>
   </node>
</obbb>
```

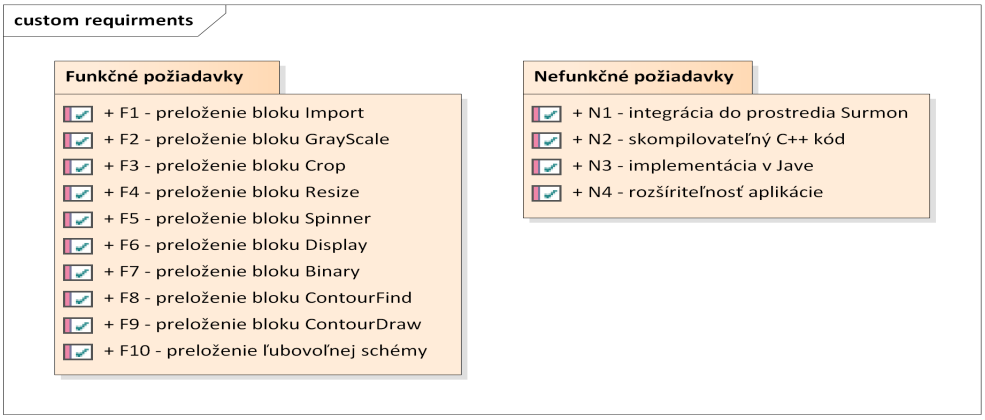
Kapitola 3

Analýza a návrh

V tejto kapitole najprv analyzujem funkčné a nefunkčné požiadavky, ktoré sú kladené na aplikáciu a potom sa dôkladne venujem návrhu samotnej aplikácie, kde rozoberám blokové triedy, popisujem vnútornú reprezentáciu, algoritmus, kontrolu vstupného súboru a cache.

3.1 Analýza požiadavok

V tejto sekcii sa venujem požiadavkom, ktoré musí aplikácia splňovať. Tieto požiadavky špecifikujú očakávané funkcionality a charakteristiky aplikácie. Požiadavky sa delia na funkčné a nefunkčné. Funkčné požiadavky sú tie, ktoré špecifikujú požadovanú funkčnosť aplikácie, teda to čo bude aplikácia vedieť robiť. Nefunkčné požiadavky nedefinujú, čo má daná aplikácia robiť, ale ako to má robiť. [2]



Obr. 3.1: Diagram funkčných a nefunkčných požiadavok.

Popis funkčných požiadavkov:

F1 Import je blok prostredia Surmon, ktorý slúži na načítanie vstupného obrazu. Z tohto bloku je možné načítať obraz priamo z disku alebo clipboard. Aplikácia musí vedieť preložiť tento blok v prípade, že sa jedná o načítanie z disku.

F2 GrayScale je blok, ktorý na svoj vstupný konektor príjma obraz a na výstupný konektor tento obraz posiela čiernobiely. Aplikácia musí vedieť preložiť tento blok aj s nastaviteľnou vlastnosťou s názvom `activate`, kde si užívateľ vybere, či tento prevod obrazu daný blok uskutoční.

F3 Crop je blok, ktorý slúži na vyrezanie obrazu pomocou zadaných súradníc. Aplikácia musí vedieť blok preložiť.

F4 Resize je blok, ktorý slúži na zmenu rozlíšenia vstupného obrazu. Aplikácia musí vedieť tento blok preložiť.

F5 Spinner slúži na posielanie zadaného čísla iným blokom. Aplikácia musí vedieť preložiť tento blok.

F6 Display slúži na zobrazovanie vstupného obrazu. Jednou z možností bloku je načítať a zobraziť obraz z disku, v prípade že sa na jeho vstupnom konektore nič nenachádza. Aplikácia musí vedieť preložiť tento blok celý.

F7 Binary blok mení vstupný obraz a musí sa mu nastaviť operačný mód. Aplikácia musí tento blok preložiť, až na operačný mód s názvom `Bimodal`.

F8 ContourFind slúži na nájdenie kontúr v obraze. Aplikácia musí preložiť tento blok so všetkými nastaviteľnými vlastnosťami, medzi ktoré patria `FindOnBorders, MaxLength, MinLength`.

F9 ContourDraw slúži na zobrazenie kontúr v obraze. Aplikácia by mala vedieť preložiť tento blok so všetkými nastaviteľnými vlastnosťami.

F10 Aplikácia musí vedieť preložiť ľubovoľne vytvorenú schému, ktorá vznikne kombináciou pozostávajúcou z vyššie uvedených blokov.

Popis nefunkčných požiadavkov:

N1 Aplikácia musí byť integrovaná v prostredí Surmon.

N2 Výsledný zdrojový C++ kód, ktorý vznikne preložením musí byť skompilovateľný a poskytovať rovnaké výsledky ako navrhnutá schéma.

N3 Výsledná aplikácia musí byť napísaná v jazyku Java.

N4 Aplikáciu by malo byť relatívne ľahké rozšíriť o preklad ďalších blokov.

3.2 Návrh tried reprezentujúcich bloky prostredia Surmon

Vzhľadom na to, že každý blok prostredia Surmon predstavuje nezávislú entitu, rozhodol som sa v aplikácii reprezentovať tieto bloky ako nezávislé triedy, ktoré nazývam blokovými triedami.

Každému bloku prostredia Surmon odpovedá jedna konkrétna bloková trieda, napríklad bloku Import odpovedá trieda s názvom Import. V ich návrhu je využité dedenie, ktoré patrí medzi základné techniky objektovo orientovaného programovania. Samotný strom dedičnosti neni nijak zložitý. Nad blokovými triedami je postavená abstraktná trieda s názvom Block, z ktorej tieto triedy dedia. Tento prístup je využitý predovšetkým z redundancie kódu, ktorý by sa opakoval v každej blokovej triede. Medzi ďalšiu výhodu patrí polymorfické volanie metódy uloženej v tejto abstraktnej triede, ktorú využíva samotný algoritmus opísaný v kapitole s názvom Algoritmus 3.4.

Bloky prostredia Surmon pracujú s knižnicou OpenCV, pomocou ktorej dáta spracovávajú. Túto knižnicu využívajú aj moje blokové triedy, ale nie vo fyzickej podobe. Bloková trieda v sebe obsahuje časť kódu napísanú v jazyku C++, uloženú ako textový reťazec, ktorá presne odpovedá kódu, napísanom v jayzku Java, pomocou ktorého odpovedajúci blok prostredia Surmon dáta spracuje. Zvyčajne ide o kód odpovedajúci nejakej funkcii knižnice OpenCV, alebo sa jedná o väčšiu časť, ktorá v sebe taktiež využíva funkcie z OpenCV. Podstatné je to, že táto preddefinovaná časť C++ kódu dáta spracuje rovnakým spôsobom, ako odpovedajúci blok prostredia Surmon. Preddefinovaný kód napĺňa každá instancia blokovej triedy potrebnými argumentami a nahráva ich do cache v správny okamžik. Instancie blokových tried taktiež nazývam blokovými instanciami.

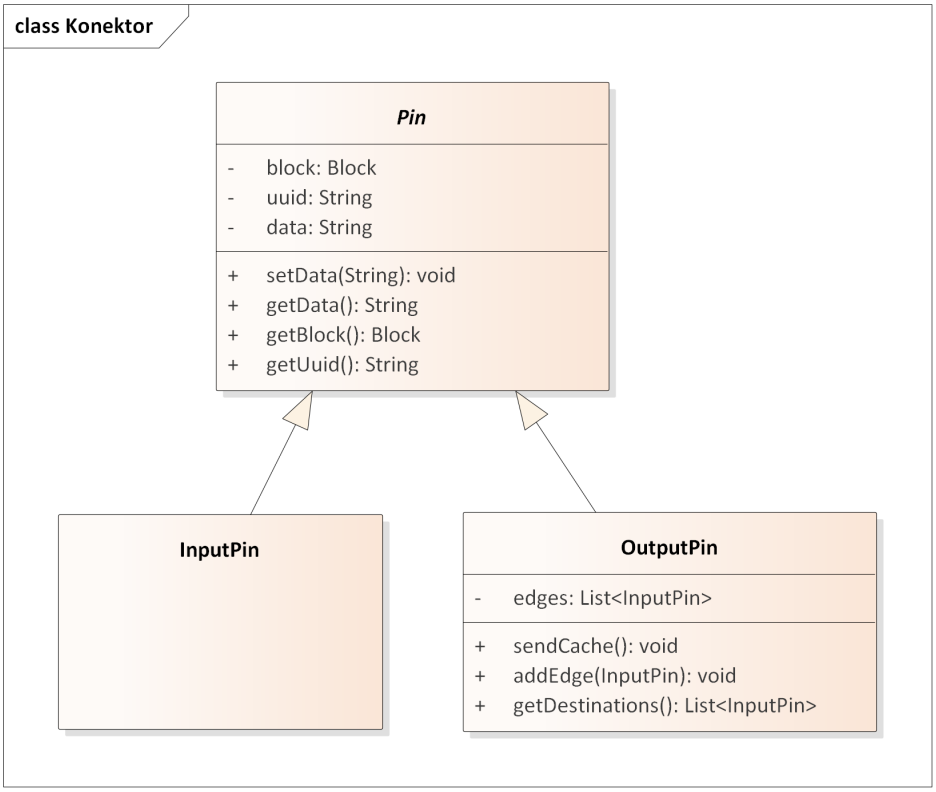
3.2.1 Mapovanie blokových vlastností

V 2.3 je opísané, že každý blok prostredia Surmon má svoje vlastnosti, ktoré sú ukladané ako hodnoty elementu `<property></property>` v XML súbore opísaného v 2.4, ktoré nejakým spôsobom ovplivňujú spracovanie jeho dát.

Každá instancia blokovej triedy si preto potrebuje tieto vlastnosti počas parsovania vstupného XML súboru ukladať. Ako najlepší spôsob ukladania sa javí abstraktný dátový typ (ADT) s názvom slovník, uložený priamo v abstraktnej triede `Block`. Jej kľúčom je hodnota atribútu `name` elementu `<property></property>` a hodnota pre túto štruktúru sa používa hodnota príslušného elementu. Podľa týchto vlastností sa bloková instancia napríklad rozhoduje, akú časť kódu použiť, alebo sa môže rovno jednať o argumenty pre časť preddefinovaného kódu, ktorá sa nimi vyplní, pred tým ako je blokovou instanciou zapísaná do cache.

3.2.2 Konektory

Rovnako ako bloky prostredia Surmon, aj moje blokové triedy obsahujú vstupné a výstupné konektory. Tieto konektory sú reprezentované ako triedy InputPin a OutputPin, nad ktorými je postavená abstraktná trieda s názvom Pin. Konektory si v sebe uchovávajú dáta ako textový reťazec. Slúžia na uchovávanie a posielanie dát medzi instanciami blokových tried. Trieda, ktorá reprezentuje výstupný konektor obsahuje dôležitú metódu `sendCache()`. Táto metóda slúži výstupnému konektoru na poslanie svojich dát všetkým vstupným konektorom, s ktorými je prepojený.



Obr. 3.2: Diagram tried konektorov.

3.2.3 Aktivačný proces

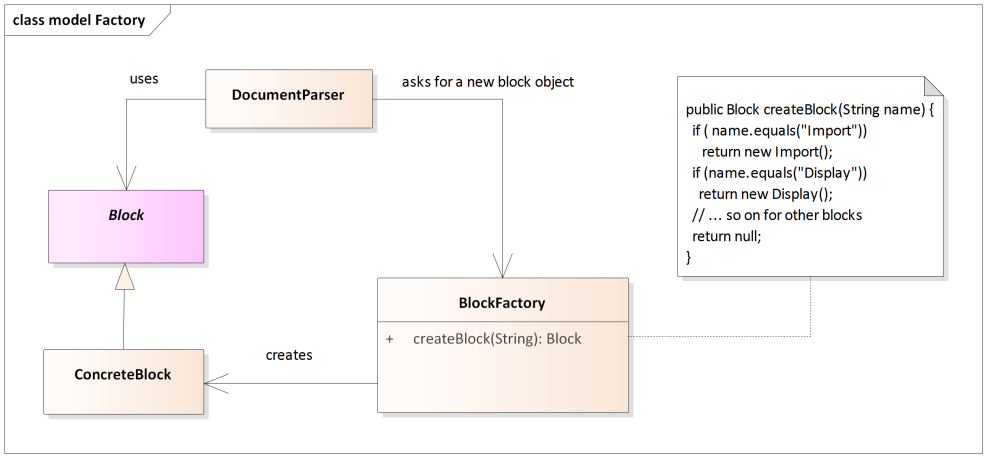
Aktivačný proces, inými slovami aktivácia je proces, ktorý je vyvolaný zavolaním metódy `activate()` blokovou instanciou. Počas tohto procesu si blok najprv načíta dáta zo vstupných konektorov, vykoná svoju logiku – u väčšiny blokov táto logika predstavuje vyplnenie preddefinovaného C++ kódu dátami, alebo vlastnosťami, tento kód zapíše do cache a nakoniec nastaví dáta na svoje odpovedajúce výstupné konektory.

3.2.4 Dôležité metódy abstraktnej triedy Block

- **Metóda activate()** – abstraktná metóda, ktorú musí implementovať každá bloková trieda. Zavolaním tejto metódy sa spustí aktivačný proces popísaný v 3.2.3. Túto metódu využíva samotný algoritmus, ktorý ju cez spoločné rozhranie polymorficky volá.
- **Metóda sendOutputPinCache()** – metóda, ktorá posiela signál každému výstupnému konektoru, aby rozoslal svoje dáta všetkým vstupným konektorom, s ktorými zdiela väzbu.
- **Metóda loadInputPinCache(int pinNum)** – metóda, ktorá bloku vráti dáta zo vstupného konektoru ležiaceho na zadanom indexe.
- **Metóda setOutputPinCache(int pinNum)** – metóda, ktorá nastaví dáta výstupnému konektoru, ktorý leží na zadanom indexe `pinNum`.

3.2.5 Vytváranie blokových instancií

Pri vytváraní blokových instancií je použitý návrhový vzor Factory, ktorý je jedným z najpoužívanejších návrhových vzorov. Tento typ návrhového vzoru spadá do kategórie tvorivých návrhových vzorov, ktoré slúžia na vytváranie objektov. Objekty sa v ňom vytvárajú bez odhalenia logiky klientovi a na novo vytvorené objekty sa odkazuje pomocou spoločného rozhrania, čím môže byť interface alebo abstraktná trieda. [3]



Obr. 3.3: Návrhový vzor Factory pre vytváranie blokových instancií.

Na obrázku 3.3 je klientom trieda DocumentParser, ktorá blokové instancie využíva.

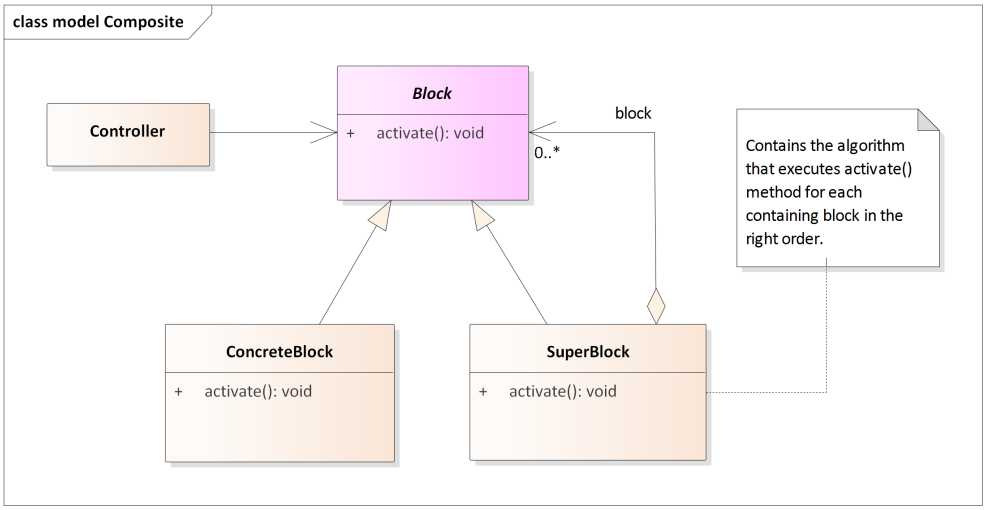
Medzi výhody tohto spôsobu vytvárania blokových instancií patrí:

- **loose coupling** - znamená redukovanie závislostí triedy, ktorá priamo používa inú triedu. Toto platí pre triedy DocumentParser a BlockFactory z obrázku 3.3,
- **rozšíriteľnosť** - programátor môže aplikáciu veľmi ľahko rozšíriť o preklad ďalších blokov, tým že k nim vytvorí a naimplementuje blokové triedy, ktoré následne pridá do Factory a samotná aplikácia bude vedieť poskytovať jej instancie cez spoločné rozhranie, ktorým je abstraktná trieda s názvom Block.

Vstupom pre metódu, ktorá poskytuje tieto blokové instancie cez spoločené rozhranie je hodnota atribútu s názvom `class` elementu `<node></node>`, podľa ktorého sa jednotlivé bloky rozoznávajú. Blokové instancie sú vytvárané počas čítania XML súboru, ktoré je podrobnejšie rozobraté v 4.4.

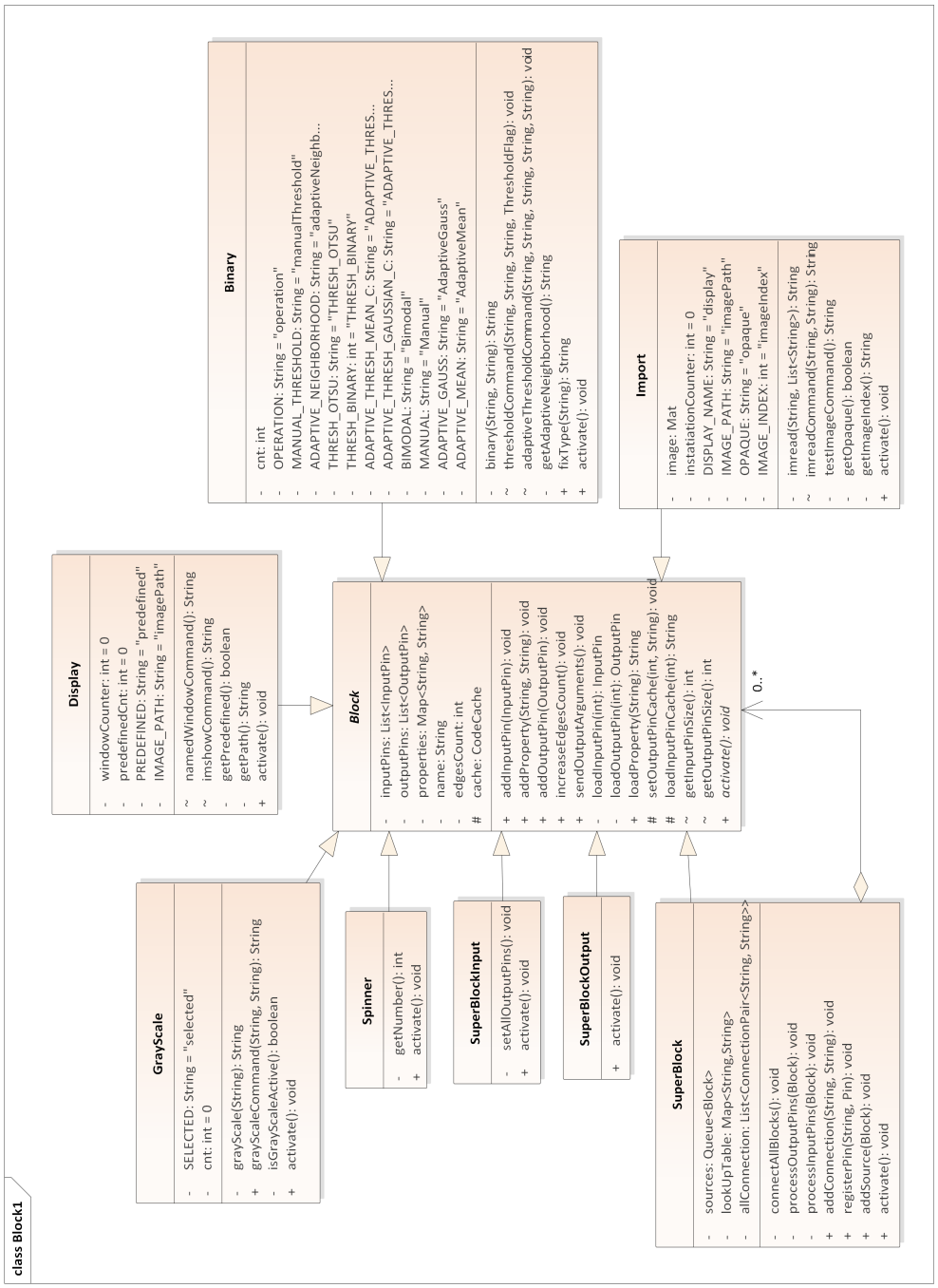
3.2.6 Bloková trieda SuperBlock

Aj keď sa preklad špeciálneho bloku s názvom SuperBlock 2.3.1 nenachádza vo funkčných požiadavkoch, je dobré ho do práce začleniť hlavne z návrhového hľadiska. Ide o špeciálny blok, ktorý je veľmi užitočný a v prípade, že by nebol počas návrhu braný do úvahy, by to mohlo znamenať, že v budúcnosti by samotné rozšírenie aplikácie o jeho preklad bolo problematické a znamenalo by to prepísanie veľkej časti zdrojového kódu. Pri návrhu triedy SuperBlock je využitý mierne upravený návrhový vzor Composite, ktorý skladá objekty do stromovej štruktúry, ktorá reprezentuje celok-časť hiearchiu. [4]



Obr. 3.4: Návrhový vzor Composite v aplikácii.

Ako je vidieť z obrázku 3.4 SuperBlock predstavuje akýsi celok, ktorý obsahuje ďalšie bloky, ktoré predstavujú časti. Medzi týmito blokmi sa znovu môže nachádzať SuperBlock, ktorý znovu obsahuje bloky. Tento spôsob reprezentácie umožní spustiť instancií triedy SuperBlock algoritmus na svoje bloky (časti). K vytváraniu tejto štruktúry dochádza počas parsovania vstupného XML súboru. Táto dátová štruktúra je podrobnejšie popísaná v 3.3.

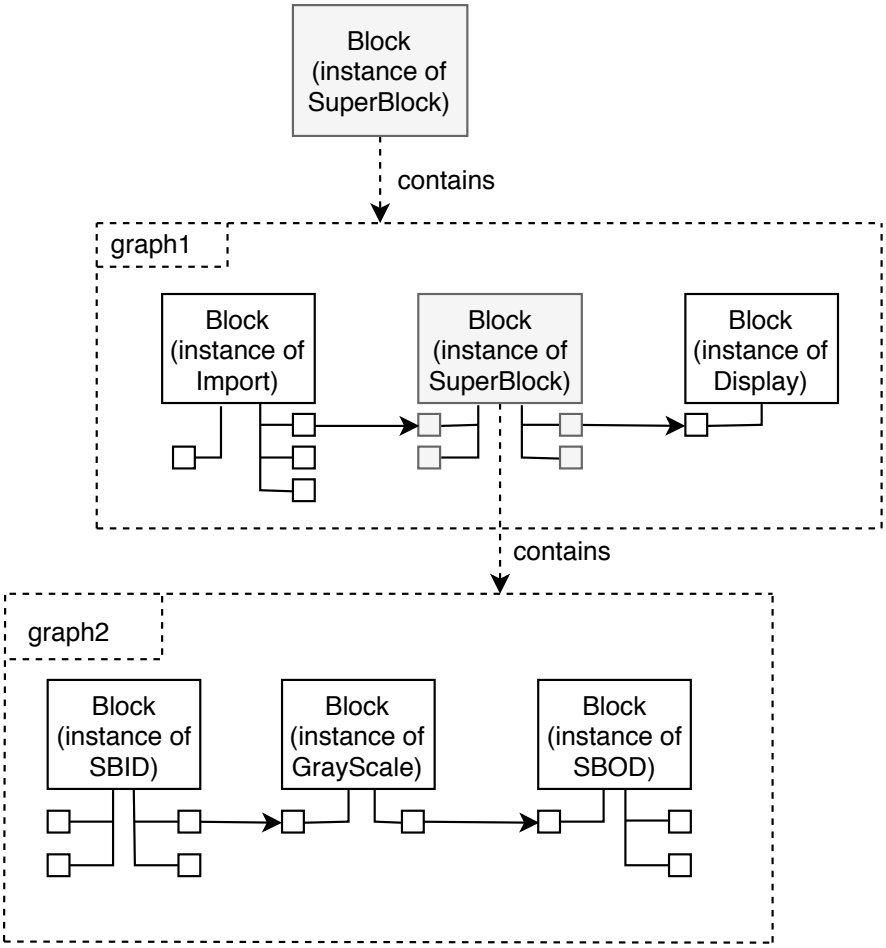


Obr. 3.5: Prvá časť diagramu tried blokov.

Obr. 3.6: Druhá časť diagramu tried blokov.

3.3 Vnútorná reprezentácia

Vzhľadom na to, že v schéme sú bloky na sebe závislé a táto závislosť určuje poradie vyhodnocovania jednotlivých blokov, tak si aplikácia potrebuje vytvárať dátovú štruktúru, ktorá sa skladá z acyklických orientovaných grafov. [5] Táto dátová štruktúra odpovedá návrhovému vzoru Composite 3.4, kde každý SuperBlock (celok) obsahuje bloky (časti). Tieto bloky vytvárajú acyklický orientovaný graf, v ktorom sú medzi sebou prepojené pomocou svojich konektorov 3.2. Jeden takýto acyklický graf presne odpovedá nejakej schéme a je vytvorený z jej XML súboru.



Obr. 3.7: Ukážka dátovej štruktúry vytvorenej k schéme 2.2 a 2.3.

Obrázok 3.7 sa skladá z 2 grafov, pretože pôvodná schéma v sebe obsahuje superblok, ktorý obsahuje ďalšiu schému. Tieto grafy sú rovnako prepojené ako schémy v prostredí Surmon.

3.4 Algoritmus

Pre samotné poskladanie funkčného C++ kódu je nutné navrhnúť algoritmus, ktorý musí zabezpečiť:

1. Posielanie dát z jedného bloku na blok druhý pomocou konektorov, ktoré uchovávajú určitý typ dát.

2. Docieliť, aby jednotlivé bloky nahrávali svoj preddefinovaný kód v správnom poradí, inými slovami bloky musia nahrať kód v momente, kedy nie sú závislé na iných blokoch.

Tieto dva problémy riešim využitím mierne upraveného algoritmu, ktorý slúži na nájdenie nejakého topologického usporiadania v orientovanom acyklickom grafe. [6]

Tento algoritmus je uložený v metóde `activate()` blokovej triedy SuperBlock, ktorá v sebe obsahuje acyklický orientovaný graf 3.7, ktorý reprezentuje štruktúru zapojenia nejakej schémy.

Vyvolaním metódy `activate()` z objektu typu SuperBlock sa spustí tento algoritmus, ktorý operuje na jeho orientovanom acyklickom grafe vytvorenom z blokov.

Nájsť topologické usporiadanie v grafe tvoreného týmito blokmi presne odpovedá poradiu, v ktorom majú bloky zavolať svoju metódu `activate()`, pretože ju vždy zavolajú až v momente, kedy sú zdrojmi tohto grafu, čo znamená, že nie sú závislé na iných blokoch, od ktorých príjmajú nejaké vstupné dáta.

Týmto spôsobom bloky vyvolávajú aktivačný proces v správnom poradí, ktorý zabezpečuje nahrávanie ich kódu v správnom poradí do cache a odosielanie dát svojim susedným blokom.

Medzi blokmi nejakého grafu sa môžu nachádzať ďalšie objekty typu SuperBlock, ktoré obsahujú svoj vlastný acyklický orientovaný graf.

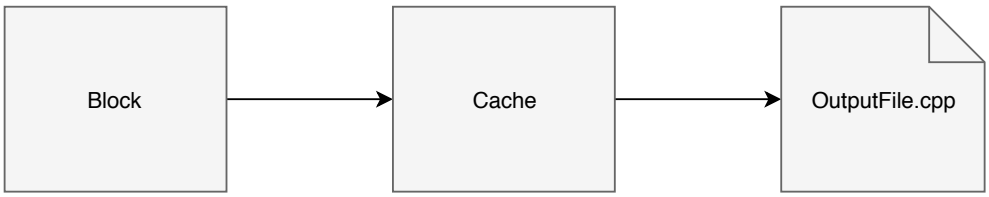
Pred tým ako je na tieto bloky zavolaná metóda `activate()`, ktorá spustí rovnaký algoritmus na ich vlastné grafy, je nutné, aby každý takýto blok najprv delegoval dáta zo svojich vstupných konektorov na výstupné konektory počiatočného bloku v jeho grafe s názvom SuperBlockInputDelegate. Na konci spracovania jeho grafu je taktiež nutné, aby koncový blok s názvom SuperBlockOutputDelegate delegoval dáta zo svojich vstupných konektorov na výstupné konektory jeho superbloku.

3.5 Kontrola vstupného súboru

Prostredie Surmon sa neustále vyvíja a stále vznikajú jeho nové a nové verzie, čoho dôsledkom bývajú modifikované jednotlivé bloky, či už pridaním alebo zmenením nejakej jeho vlastnosti. Všetky takéto modifikácie sa následne prejavujú aj v XML súbore elementu `<node></node>`, ktorý tento blok reprezentuje. Tieto zmeny môžu predstavovať problém, pretože aplikácia s týmito novými hodnotami nebude vedieť pracovať, čo môže spôsobiť chybné preloženie bloku a vo výsledku špatný zdrojový kód. Ďalšou vecou, ktorú je treba kontrolovať je, či aplikácia dokáže preložiť každý blok, ktorý sa v súbore nachádza. Z týchto dôvodov sa v aplikácii niektore časti XML súboru pri jeho spracovaní kontrolujú. Podrobnejšia štruktúra je rozobratá v 4.3.

3.6 Cache

Cache je v aplikácii medzičlánok medzi blokom a výstupným súborom, ktorý na konci algoritmu obsahuje zdrojový výsledný C++ kód. Blok nenahráva zdrojový kód priamo do súboru, ale najprv sa nahrá do cache, ktorá sa skladá z viacerých častí a slúži pre lepšiu organizáciu tohto zdrojového kódu. Z cache je po skončení algoritmu nahratý výsledný C++ kód do súboru.



Obr. 3.8: Ukážka nahrávania kódu blokom.

3.6.1 Výhody cache

Hlavnou výhodou tohto medzičlánku je možnosť nahratý zdrojový kód nejakým zmysluplným spôsobom usporiadať pred tým ako je zapísaný do výstupného súboru.

- cache umožnuje blokom definovať svoje vlastné funkcie pred telo funkcie `main()`, v ktorej môže tieto funkcie volať,
- umožnuje presunúť všetky blokom definované C++ premenné na samotný začiatok funkcie `main()`,
- umožnuje jednotlivým blokom definovať knižnice na samotný začiatok výstupného súboru.

Kapitola 4

Realizácia

V tejto kapitole popisujem realizáciu samotnej aplikácie, kde sa zameriavam na časti, ktoré sa týkali samotnej implementácie. Táto kapitola začína popísaním použitých technológií, ďalej popisujem štruktúru môjho projektu a následne sa zameriavam na popis dôležitých tried a ich metód, ktoré predstavujú najdôležitejšie časti aplikácie. Táto kapitola je zakončená testovaním, kde popisujem postup pri testovaní aplikácie.

4.1 Technológie

Java – je jedným z najpopulárnejších programovacích jayzkov v súčasnej dobe. Ide o objektovo orientovaný jazyk, ktorý vyniká kvôli jednoduchosti, automatickej správy pamäte, množstvu knižníc a výbornej dokumentácií. Okrem spomenutých výhod je tento programovací jazyk vybratý, aby bolo možné aplikáciu ľahko integrovať do prostredia Surmon, ktorý je v Jave napísaný.

C++ – je staticky typovaný programovací jazyk, ktorý podporuje procedurálne, objektovo-orientované a generické programovanie. Považuje sa za jazyk strednej úrovne, pretože zahŕňa kombináciu jazykových funkcií vysokej a nízkej úrovne. [7]

JDOM – predstavuje jednoduchú Java reprezentáciu XML dokumentu a poskytuje spôsob ako tento dokument reprezentovať pre jednoduché a efektívne čítanie, manipuláciu a zapisovanie. JDOM obsahuje jednoduché a priamočiare API, je ľahký na použitie, rýchly a optimalizovaný pre Java programátora. Jedná sa o alternatívu k DOM a SAX, aj keď sa s nimi veľmi dobre integruje. [8] Tento framework využívam, pretože sa s ním naozaj ľahko pracuje, je prehľadný a umožňuje mi rekurzívne parsovanie viacerých XML súborov, čo je napríklad so SAX problematické.

4.2 Štruktúra balíkov

Samotná aplikácia obsahuje okolo 30 tried a z toho dôvodu je nutné tieto triedy nejakým logickým spôsobom usporiadať kvôli prehľadnosti. Presne pre túto vec existujú v Jave balíky, ktoré združujú triedy do logických celkov, kde potom jednotlivé balíky využívajú balíky iné. Aplikácia obsahuje 7 hlavých balíkov, ktoré sú zastrešené hlavným balíkom s názvom org.surmon.schematranslator

- **blocks** – v tomto balíku sa nachádzajú všetky blokové triedy. Môžu sa tu nachádzať aj vnorené balíky, ktoré bližšie zoskupujú blokové triedy do užších skupín,
- **definedfunctions** – balík, v ktorom sa nachádzajú triedy, ktoré v sebe obsahujú funkcie, ktoré môžu využívať všetky blokové triedy. Triedy v tomto balíku predstavujú preddefinované C++ funkcie, ktoré sú blokmi nahrávané pred samotné telo funkcie `main`,
- **documentparser** – balík, v ktorom sa nachádzajú triedy, ktoré slúžia na parsovanie vstupného XML súboru. Nachádza sa v ňom rozhranie DocumentParser, ktoré musia tieto triedy implementovať. Tento balík slúži na pridávanie tried, ktoré parsujú vstupný súbor,
- **factory** – v tomto balíku sa nachádzajú triedy, ktoré poskytujú nejaký určitý typ objektu a sú implementované pomocou návrhového vzoru Factory,
- **simplecodegenerator** – v tomto balíku sa nachádzajú triedy, ktoré slúžia na generovanie jednoduchého C++ kódu (podmienky, cykly, funkcie.). Triedy v tomto balíku zbavujú programátora do určitej miery hardcodingu,
- **utilities** – balík obsahuje triedy, ktoré slúžia ako utility, ktoré využívajú blokové triedy. Väčsinou sa jedná o statické triedy,
- **wrappers** – tento balík obsahuje triedy, ktoré predstavujú C++ objekty ako napríklad `vector`, ktoré tatiež zbavujú programátora hardcodingu pri implementácií blokových tried.

4.3 Trieda DataValidator

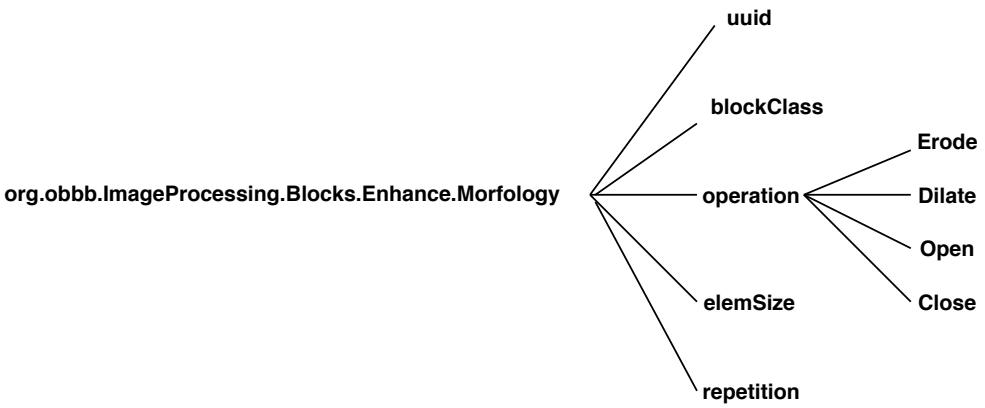
Táto trieda zabezpečuje validáciu určitých častí vstupného XML súboru. Obsahuje dátovú štruktúru, ktorá slúži ako databáza, podľa ktorej sa pri čítaní XML súboru jeho určité časti kontrolujú. Táto trieda obsahuje 2 hlavné metódy `validateBlockName()` a `validateBlockProperties()`. V prípade, že nejaká časť XML súboru neprejde validáciou, tak je táto chyba zapísaná do výstupného súboru.

Popis metód:

- Metóda `validateBlockName()` – metóda, ktorá príjma z XML súboru názov bloku a kontroluje, či sa tento blok nachádza v databáze,
- Metóda `validateBlockProperties()` – metóda, ktorá slúži na kontrolu blokových vlastností.

4.3.1 Štruktúra databázy

Pre každý naimplementovaný blok je v databáze uložená hodnota `class` atribútu elementu `<node></node>` (podľa nej poskytuje BlockFactory jednotlivé bloky), ktorá obsahuje odkazy na všetky povolené hodnoty atribútu `name` elementu `<property></property>`, ktoré následne obsahujú odkazy na všetky možné hodnoty tohto elementu, v prípade, že to má zmysel kontrolovať a hodnota tohto elementu neni napríklad číslo.



Obr. 4.1: Ukážka záznamu pre blokovú triedu Morfology

Ako je vidieť na obrázku 4.2, pre naimplementovaný blok `Morfology` je v aplikacii uložený názov org.obbb.ImageProcessing.Blocks.Enhance.Morfology, ktorý má odkazy na jeho povolené vlastnosti `uuid`, `blockClass`, `operation`, `elemSize`, `repetition` a `operation` má odkazy na `Erode`, `Dilate`, `Open`, `Close`, pretože ich má zmysel kontrolovať. Takýto záznam je vytvorený v aplikacii pre každú jednu naimplementovanú triedu.

4.4 Trieda XMLParserJDOM

Táto trieda slúži na parsovanie XML súboru pomocou metódy `parse()`, počas ktorého dochádza k vytváraniu dátovej štruktúry popísanej v 3.3 a kontrolovanie častí tohto súboru pomocou triedy DataValidator, na ktorú táto trieda obsahuje referenciu.

4.4.1 Metóda parse()

Metóda príjma 2 argumenty, ktorými sú absolútna cesta XML súboru a referencia na objekt typu SuperBlock, do ktorého sa ukladajú vytvorené bloky. Funkcia obsahuje cyklus, kde sa iteruje cez všetky blokové elementy, ktoré sa nachádzajú v elemente superblok. Z každého takéhoto elementu sa vyparsuje hodnota jeho atribútu `class` a poskytne sa metóde `getBlock()`, ktorá vytvorí a vráti určitý blok. V metóde `processBlockPins()` sa vytvárajú a priraďujú bloku konektory, registrujú sa medzi blokovými konektormi spojenia a ukladajú sa v objekte `superblock`. Metóda `processBlockProperties()` slúži na mapovanie blokových vlastností opísaných v 3.2.1, kde je slovník realizovaný ako hash mapa. V prípade, že vytvorený blok metódou `getBlock()` odpovedá superbloku, tak sa metóda rekurzívne zavolá s cestou na jeho XML súbor a referenciou na tento blok, čím sa znovu spustí rovnaký proces. Na konci funkcie je vrátená referencia na objekt typu `Block`, z ktorého sa spúšta celkový algoritmus zavolaním jeho metódy `activate()`.

```
1  public Block parse(String path, SuperBlock superblock) throws Exception
2    File inputFile = new File(path);
3    SAXBuilder saxBuilder = new SAXBuilder();
4    Document document = saxBuilder.build(inputFile);
5
6    Element root = document.getRootElement();
7    List<Element> blocks = root.getChildren("node").get(0).getChildren();
8
9    for (Element block : blocks) {
10     String blockName = block.getAttributeValue("class");
11     dataValidator.validateBlockName(blockName);
12
13     Block blockInstance = factory.getBlock(blockName);
14     blockInstance.setName(blockName);
15
16     processBlockPins(block, blockInstance, superblock);
17     processBlockProperties(block, blockInstance);
18
19     if (blockName.equals(BlockType.SUPERBLOCK.getType())) {
20       String xmlFileName = blockInstance.loadProperty("file");
21       String xmlFilePath = path.substring(0, path.lastIndexOf('\\') + 1);
22       parse(xmlFilePath + xmlFileName, (SuperBlock)blockInstance);
23     }
24   }
25   return superblock;
26 }
```

4.5 Trieda SuperBlock

Ako je opísané v 3.2.6 trieda `SuperBlock` predstavuje celok, ktorý v sebe uchováva časti (bloky). K uchovávaniu týchto blokov využíva táto trieda frontu, v ktorej má uložené bloky, ktoré sú zdrojmi jeho acyklického orientovaného grafu.

Algoritmus opísaný v 3.4 má táto trieda naimplementovaný v metóde s názvom `activate()`.

4.5.1 Metóda activate()

Pre nájdenie nejakého topologického usporiadania v grafe je využitá mierne upravená verzia Kahnovho algoritmu, ktorá využíva frontu. V tejto fronte sa nachádzajú zdroje grafu, ktoré sa postupe vyberajú a volá sa ich metóda `activate()`. Blok sa stáva zdrojom a je algoritmom pridaný do fronty v momente, kedy do jeho vstupných konektorov nevedie žiadna hrana.

```
1  public void activate() {
2
3    connectAllBlocks();
4
5    while(! sources.isEmpty()) {
6      Block block = sources.poll();
7      block.setCache(cache);
8      processInputPins(block);
9      block.activate();
10     processOutputPins(block);
11     List<OutputPin> outputPins = block.getOutputPins();
12     for (OutputPin outputPin : outputPins ) {
13       List<InputPin> destinations = outputPin.getDestinations();
14       for (InputPin inputPin : destinations) {
15         Block destinationBlock = inputPin.getBlock();
16         destinationBlock.decreaseEdgesCount();
17         if (destinationBlock.getEdgesCount() == 0) {
18           sources.add(destinationBlock);
19         }
20       }
21     }
22   }
23
24   sendOutputArguments();
25 }
```

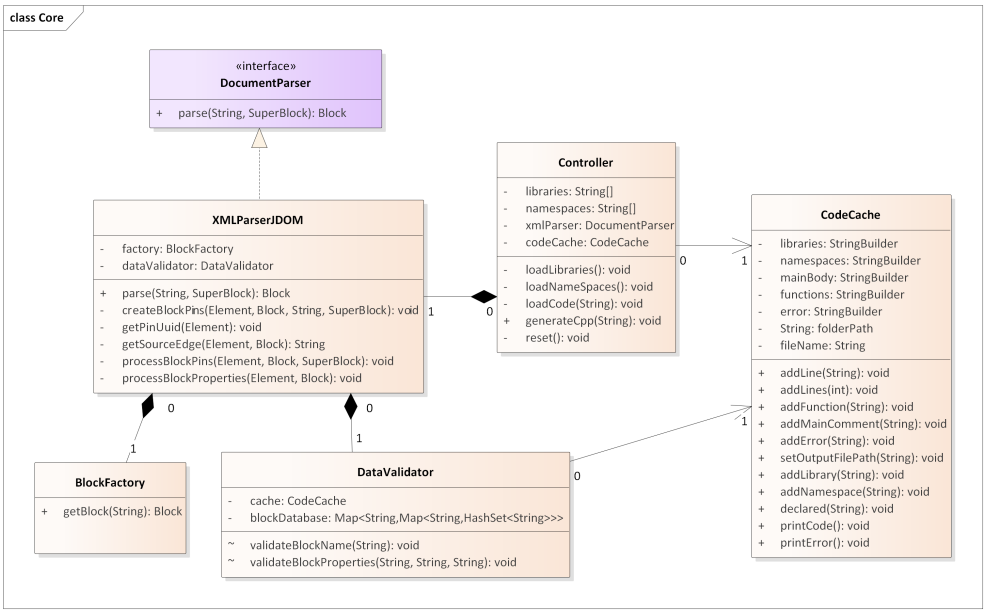
Metóda `processInputPins()` slúži na delegovanie dát na výstupné konektory bloku SuperBlockInputDelegate. Metóda `processOutputPins()` slúži na delegovanie dát zo vstupných konektorov bloku SuperBlockOutputDelegate na výstupné konektory superbloku, ktorý metódu `activate()` momentálne vykonáva.

Na konci metódy musí byť zavolaná metóda `sendOutputPinCache()`, ktorá je opísana 3.2.4, pretože daný superblok môže byť súčasťou iného grafu.

4.6 Trieda Controller

Trieda `Controller` predstavuje štartovací bod aplikácie. Táto trieda obsahuje metódu `generateCpp()`, ktorá príjma z prostredia Surmon absolútnu cestu vstupného XML súboru, ktorý následne deleguje triede XMLParserJDOM.

Úlohou tejto triedy je nahrať do cache všetky potrebné C++ knižnice, namespace, začiatok tela `main` a spustiť samotný algoritmus vyvolaním metódy `activate()` zo štartovacieho bloku, na ktorý mu poskytne referenciu metóda `parse()` triedy XMLParserJDOM. Na konci algoritmu trieda nahrá koniec tela funkcie `main()`, čím dôjde k vytvoreniu výsledného kódu C++ kódu.



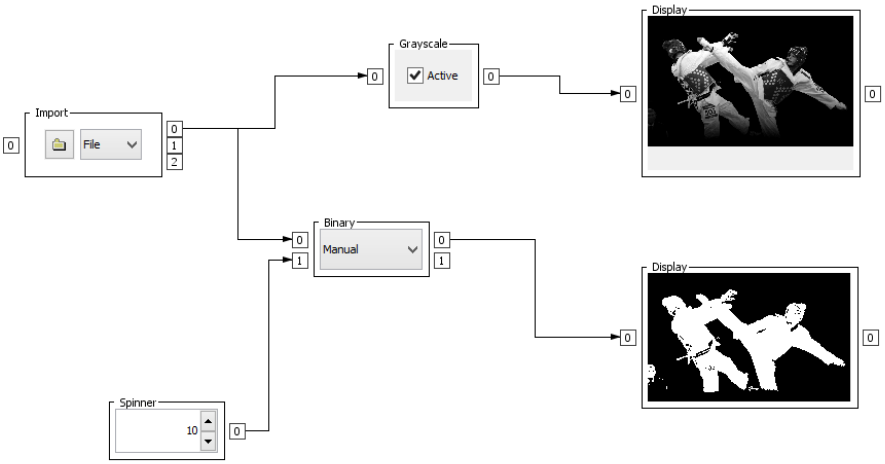
Obr. 4.2: Diagram tried dôležitých častí aplikácie.

4.7 Testovanie

Aplikácia bola testovaná manuálne na schémach, ktoré boli vytvorené v prostredí Surmon z blokov, ktoré aplikácia dokáže preložiť. Pri testovaní sa manuálne porovnávali výstupy zo skompilovaného C++ kódu s výstupmi zo schém, vytvorených v prostredí Surmon. Pri testovaní bola využitá knižnica OpenCV verzie 3.1. Aplikácia bola testovaná približne na 60 schémach, kde som sa snažil, aby jednotlivé schémy boli unikátne.

Počas testovania som nenarazil na žiadne chyby, ktoré by súviseli so samotným algoritmom, ktorý zabezpečuje skladanie kódu. Väčšina chýb, ktoré sa počas testovania vyskytli spočívali v chybne napísanom preddefinovanom kóde, ktorý bol umiestnený v nejakej blokovej triede.

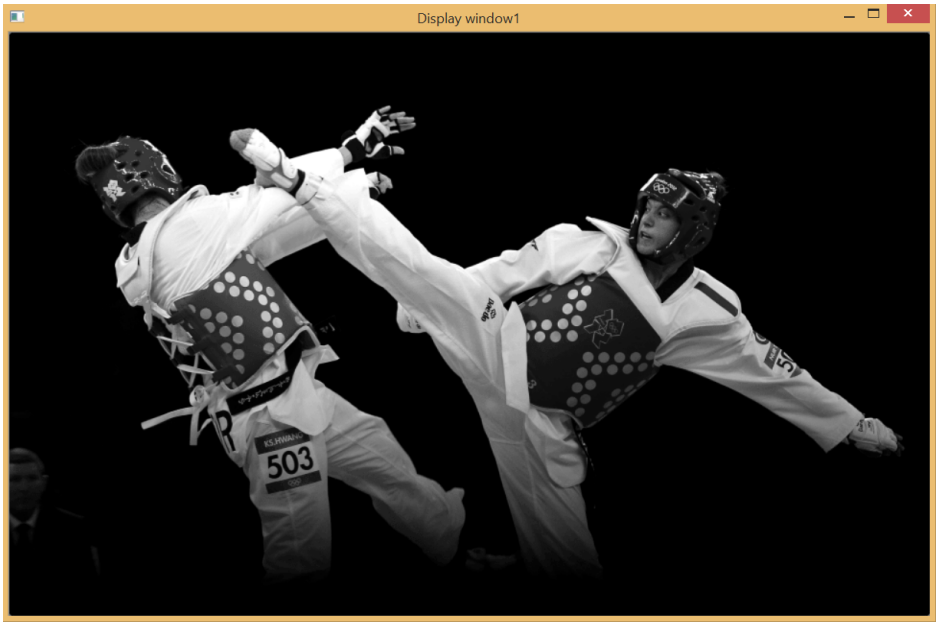
Vybrané testovacie schémy spolu s výstupnými kódmi a obrázkami sú uložené na médiu, ktoré je priložené k tejto práci.



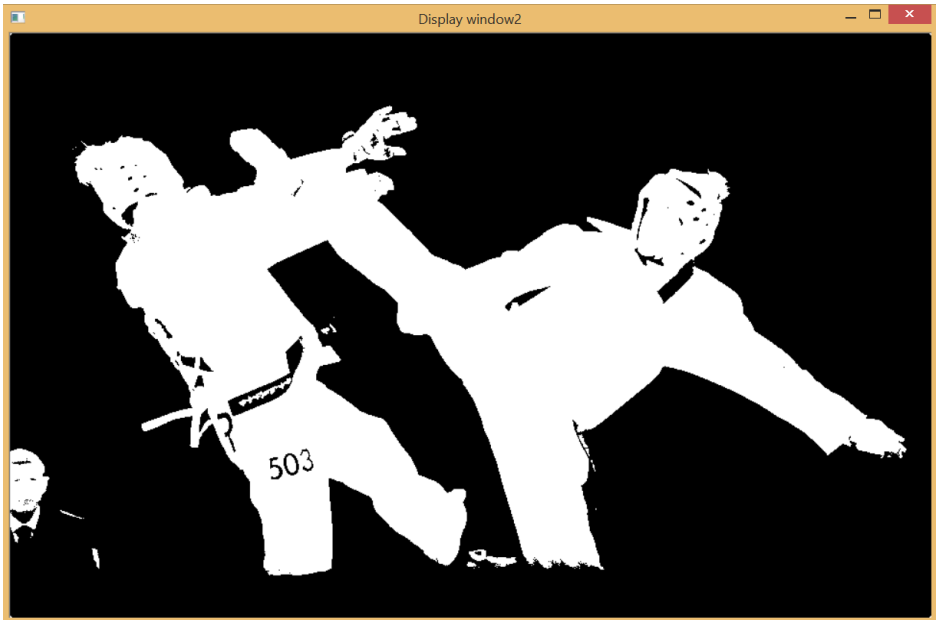
Obr. 4.3: Ukážka testovacej schémy.

Ukážka kódu vygenerovanému ku schéme 4.3:

```
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <iostream>
#include <vector>

using namespace cv;
using namespace std;

int main() {
  //SUPERBLOCK STARTS
  vector<string> vec1;
  vec1.push_back("C:/Users/Marek/Desktop/1.png");
  vec1.push_back("C:/Users/Marek/Desktop/2.png");

  Mat mat1;
  mat1 = imread(vec1[0], -1);

  if (! mat1.data) {
    cout << "Could not open or find the image" << endl;
    return -1;
  }

  Mat gray1;
  cvtColor(mat1, gray1, CV_BGR2GRAY);

  if (mat1.type() != CV_8UC1) {
    cvtColor(mat1, mat1, CV_BGR2GRAY);
  }

  Mat binary1;
  threshold(mat1, binary1, 10, 255, THRESH_BINARY);

  namedWindow("Display window1", WINDOW_AUTOSIZE);
  imshow("Display window1", gray1);

  namedWindow("Display window2", WINDOW_AUTOSIZE);
  imshow("Display window2", binary1);
  //SUPERBLOCK ENDS

  waitKey(0);
  return 0;
}
```



Obr. 4.4: Prvý zobrazený displej po spustení skompilovaného kódu vygenerovanému ku schéme 4.3.



Obr. 4.5: Druhý zobrazený displej po spustení skompilovaného kódu vygenerovanému ku schéme 4.3.

Kapitola 5

Obmedzenia aplikácie

V tejto kapitole by som sa rád vyjadril k určitým obmedzeniam samotnej aplikácie.

5.1 Správa pamäte

Aj keď to mojou úlohou nebolo riešiť, tak medzi najväčšie obmedzenie v doterajšom stave výsledného zdrojového C++ kódu, ktorý aplikácia generuje k jednotlivým schémam považujem slabšiu prácu s pamäťou. V zdrojovom kóde sú minimálne uvolňované objekty napríklad typu `Mat`, ktoré bloky generujú. Z ukážky zdrojového kódu zo sekcie 4.7 je tento nedostatok vidieť, kedy za funkciou `threshold()` mohla byť vygenerovaná funkcia `mat1.release()`, ktorá by tento už nepotrebný objekt uvolnila. Tento problém uvolňovania pamäte sa dá v aplikácii riešiť a vedie to na problém hladania najdlhšej cesty v acyklickom orientovanom grafe z blokov, ktoré takéto objekty generujú.

5.2 Preddefinovaný kód

Momentálne sa v aplikácii nachádzajú nejaké časti väčšieho preddefinovaného kódu, ktoré bloky nahrávajú pred telo funkcie `main()` v triedach, ktoré sa nachádzajú v balíku `definedfunctions`. Tento kód v týchto triedach je generovaný pomocou primitívneho generátoru, ktorý zbavuje do určitej časti hardcodingu pri ich písaní. Lepším riešením by bolo mať tieto kódy uložené v súboroch, ktoré by boli logicky usporiadané a z nich by jednotlivé bloky tento kód čítali a zapisovali do cache.

Záver

Cieľom tejto bakalárskej práce bolo navrhnúť a implementovať aplikáciu, ktorá umožní preložiť schémy pozostávajúce z predom určenej podmnožiny blokov prostredia Surmon.

Aplikáciu sa mi podarilo navrhnúť, implementovať a taktiež je integrovaná v prostredí Surmon. Výsledný C++ kód, ktorý vznikne prekladom takejto schémy je skompilovateľný a poskytuje rovnaké výsledky ako navrhnutá schéma. V súčasnej podobe dokáže aplikácia pracovať aj so schémou, ktorá obsahuje špeciálny blok s názvom superblok, ktorý je v samotnom prostredí Surmon do určitej miery nefunkčný.

Samotná práca mi prišla viac menej implementačná, kde po vymyslení hlavnej myšlienky, ktorá spočíva v skladaní kódu je rozšírenie aplikácie o ďalšie blokové triedy podobné, založené na predpripravenom C++ kóde, ktorý je uložený v blokových triedach s ich naimplementovanou logikou. Týmto považujem aj cieľ v relatívne ľahkej rozšíriteľnosti aplikácie za splnený.

Literatúra

[1] BRADSKI, Gary, KAEHLER, Adrian. *Learning OpenCV*. United States of America: O'Reilly Media, Inc., 2008. 555. 978-0-596-51613-0.

[2] CHUNG, Lawrence, et al. *Non-functional requirements in software engineering*. Springer Science & Business Media, 2012.

[3] *Tutorials Point* [online]. Tutorials Point. [cit. 2018-04-26]. Dostupné z: `https://www.tutorialspoint.com/design_pattern/factory_pattern.htm`.

[4] GAMMA, Erich, et al. *Design patterns: elements of reusable object-oriented software*. 2. vydanie. United States at Courier Westford in Westford, Massachusetts: Addison-Wesley, 1995 Pearson Education India, 1995. 395. 0-201-63361-2.

[5] RUOHONEN, Keijo. *Graph Theory*. In: *Keijo Ruohonen* [online]. Tampere University of Technology. 2013. [cit. 2018-05-01]. Dostupné z: `http://math.tut.fi/~ruohonen/GT_English.pdf`.

[6] CORMEN, Thomas H., et al. *Introduction to Algorithms*. 3. vydanie. The MIT Press, 2009. 1292. 978-0-262-03384-8.

[7] Tutorials Point (I) Pvt. Ltd. *Learn C++ programming language* [online]. 2014. [cit. 2018-05-10]. Dostupné z: `http://cds.iisc.ac.in/wp-content/uploads/DS286.AUG2016.Lab2_.cpp_tutorial.pdf`.

[8] *Frequently Asked Questions* [online]. Jason Hunter, Rolf Lear. [cit. 2018-05-10]. Dostupné z: `http://www.jdom.org/docs/faq.html#a0000`.

Dodatok **A**

Zoznam použitých skratiek

GUI Graphical user interface

XML Extensible markup language

SAX Simple API for XML

DOM Document Object Model

ADT Abstract Data Type

Dodatok B

Obsah priloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresár so spustiteľnou formou implementácie
- `tests` ..... ukážky z testovania aplikácie
- `src`
  - `impl` ..... zdrojové kódy implementácie
  - `thesis` ..... zdrojová forma práce vo formáte LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce vo formáte PDF