Jihočeská univerzita v Českých Budějovicích

Přírodovědecká fakulta

Nástroje strojového zpracování obrazu v ekologii

Diplomová práce

Bc. Daniel Domin

Vedoucí práce: Mgr. Miloš Prokýšek, Ph.D

České Budějovice 2015

Bibliografické údaje

Domin, D., 2015: Nástroje strojového zpracování obrazu v ekologii

[Tools for image processing in ecology. Mgr. Thesis, in Czech.] – 74 p., Faculty of Science, The University of South Bohemia, České Budějovice, Czech Republic.

Anotace:

This thesis deals with automated image processing in biology and ecology. Based on the analysis of image processing steps and most common cases of application in biology and ecology, the main outcome of this thesis is a modular application framework prototype. It allows modeling of the whole process of image analysis which can be applied on a large batch of input data. Moreover, it can be extended by modules with new features so it can be used generally for any sequential data processing.

Prohlašuji, že svoji diplomovou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své diplomové práce, a to **v** nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích 20.4.2015

Poděkování

Děkuji Mgr. Miloši Prokýškovi, Ph.D za odborné vedení mé práce a Mgr. Miloslavu Thonovi za cenné rady a připomínky.

Obsah

1 ÚVOD 1

1.1 Cíle práce 2

1.2 Metodika 2

2 ZPRACOVÁNÍ OBRAZU 4

2.1 Snímání obrazu 4

2.2 Předzpracování obrazu 4

2.2.1 Vylepšení obrazu 5

2.2.2 Restaurace obrazu 5

2.2.3 Úprava barev 5

2.2.4 Transformace signálu 5

2.2.5 Matematická morfologie 6

2.3 Analýza obrazu 6

2.3.1 Segmentace obrazu 6

2.3.2 Popis nalezených objektů 7

2.3.3 Klasifikace 7

3 VYUŽITÍ ZPRACOVÁNÍ OBRAZU 8

3.1 Úprava obrazu 8

3.2 Určení počtu objektů 9

3.3 Zjištění kvantitativních parametrů objektu 9

3.4 Analýza pohybu 10

3.5 Rozpoznání objektu 10

4 DOSTUPNÉ NÁSTROJE 12

4.1 Jednoúčelové 12

4.2 Rozšíření 13

4.3 Modulární aplikace 14

4.4 Orchestrace více nástrojů 16

4.5 Knihovny a frameworky 16

5 ANALÝZA 18

5.1 Požadavky na aplikaci 19

5.1.1 Nefunkční požadavky 19

5.1.2 Funkční požadavky 19

5.2 Případy užití 20

6 NÁVRH APLIKACE 21

6.1 MODULÁRNÍ ARCHITEKTURA 21

6.2 ZPRACOVÁNÍ DAT 23

6.3 SCHÉMA APLIKACE 25

6.4 POUŽITÉ TECHNOLOGIE 27

6.4.1 Microsoft .NET Framework 27

6.4.2 NShape Framework 27

6.4.3 Emgu CV 27

7 IMPLEMENTACE 29

7.1 STRUKTURA PROJEKTU 29

7.2 JÁDRO 30

7.3 PROCES ZPRACOVÁNÍ 33

7.3.1 Roury 33

7.3.2 Graf filtrů 34

7.4 FILTR 38

7.4.1 Abstraktní třída Filter 40

7.4.2 Implementované filtry 43

7.4.3 Tvorba vlastních filtrů 46

7.5 GUI 49

7.5.1 Grafická reprezentace filtru 51

7.5.2 Vkládání a odebírání filtrů 53

7.5.3 Změna parametrů 54

7.5.4 Spojování filtrů 55

7.5.5 Ukládání projektu 55

7.5.6 Spuštění procesu zpracování 56

8 TESTOVÁNÍ 57

9 ZÁVĚR 58

SEZNAM POUŽITÉ LITERATURY 59

PŘÍLOHA A – TESTOVACÍ SCÉNÁŘE 64

1 Úvod

V současné době nachází zpracování obrazu stále více uplatnění v mnoha různých oborech, ať už se jedná o automatizaci výroby, kde jsou roboty schopny kontrolovat vlastní práci [1] nebo v oblasti domácí zábavy při snímání hráče, který ovládá hru gesty a pohyby svého těla [2]. S nárůstem výkonu mobilních zařízení je možné provádět zpracování obrazu i v terénu. Například nová verze překladače Google Translator umí text snímaný fotoaparátem nejen přeložit, ale také přeložený text vložit zpět do původního obrázku při zachování barev a fontů [3].

Své využití má zpracování obrazu i v biologii. Již pouhá úprava pořízeného snímku, odstranění šumu, oříznutí či změna velikosti je prvním krokem v procesu zpracování. Velkou skupinou řešených úloh je zpracování obrazů pořízených digitálními mikroskopy. Jedná se především o úpravu a rekonstrukci rozostřených či jinak deformovaných obrazů, detekci buněk a sledování jejich pohybu, kvantitativní analýza vlastností a v neposlední řadě vizualizace a zvýraznění sledovaných znaků [4]. Uplatnění nachází i při vyhodnocování výsledů elektroforézy [5], počítání a měření rostlin či živočichů a při řešení mnoha dalších úloh [6].

Každá úloha, byť ze stejného oboru, ale vyžaduje specifický způsob řešení. Výsledkem je vznik řady úzce specializovaných nástrojů, vhodných pouze k řešení dané úlohy. Navíc tyto softwarové nástroje bývají velmi často dodávány s konkrétním hardware a jedná se o proprietární řešení, které znemožňuje interoperabilitu s jinými programy [7]. Nejen při využití ve výzkumu je nutné vědět, které algoritmy byly programem aplikovány a jaké bylo nastavení jejich parametrů. Některé aplikace však fungují jako černá skříňka a není možné tyto údaje zjistit. Jen u velmi malého množství dostupných nástrojů byla jejich funkčnost ověřena nějakou vědeckou studií [4].

Výše uvedené problémy jsou hlavní motivací této práce. Cílem je vytvoření jednotného programového základu, který bude možné využít k řešení jednoduchých i složitějších úloh zpracování obrazu bez nutnosti pro každý projekt vytvářet nové jednoúčelové aplikace. Díky modulárnímu konceptu bude možné sadu základních operací rozšířit, případně zcela nahradit vlastními moduly, které mohou obsahovat libovolnou dílčí funkci aplikovatelnou v procesu zpracování obrazu. Na uživatelské úrovni pak půjde pouze o sestavování vhodné posloupnosti těchto modulů a nastavování

jejich parametrů. Nebudou tedy vyžadovány žádné programátorské znalosti pro využití dostupných operací.

1.1 Cíle práce

Hlavním cílem této práce je návrh a implementace aplikačního frameworku, který umožní modelovat a řídit procesy zpracování obrazu včetně možnosti využití grafického návrháře. Bude tvořit programový základ využitelný pro řešení libovolného sekvenčního zpracování dat bez omezení doménou řešeného problému. Součástí práce bude také programová dokumentace včetně návodu pro tvorbu vlastních modulů.

Tento hlavní cíl je dále rozpracován na systém dílčích cílů:

- Specifikovat požadavky na aplikační framework,
- Vytvořit návrh aplikace dle definovaných požadavků,
- Ověřit funkčnost výsledného řešení.

Pro dosažení cílů práce byly stanoveny tyto úkoly:

- Vytýčit metody zpracování obrazu relevantní pro oblast biologického výzkumu,
- Analyzovat nástroje zpracování obrazu referované v oborových publikacích,
- Navrhnout a implementovat aplikační framework pro analýzu obrazu,
- Navrhnout a implementovat funkce zpracování obrazu na základě předchozí analýzy,
- Otestovat framework z hlediska chybovosti a výkonu.

1.2 Metodika

Pro analýzu principů procesu zpracování obrazu bude využito studia primárních a sekundárních pramenů. Dále bude provedena retrospektivní rešerše využití zpracování obrazu v biologii a komparativní analýza dostupných nástrojů strojového zpracování obrazu. Na základě sběru požadavků a jejich následné analýzy bude vypracován návrh aplikace. Vývoj aplikace bude veden pomocí agilní metodiky extrémního programování [8]. Tato metodika je vhodná pro tvorbu prototypových řešení, kdy jsou implementovány pouze nejnutnější části a při vývoji dochází často k velmi zásadním

změnám ve zdrojovém kódu. Funkčnost výsledného prototypu pak bude otestována na několika konkrétních úlohách.

Jako vývojová platforma byl zvolen Microsoft .NET Framework 4.5 a programovací jazyk C# ve vývojovém prostředí Microsoft Visual Studio 2013. Díky plánovanému rozšíření podpory .NET frameworku pro operační systémy Linux a OS X [9] bude výsledné řešení multiplatformní. Dále bude použit framework NShape[1] pro práci s diagramy, který má otevřený zdrojový kód psaný taktéž v C# a pro nekomerční využití je k dispozici zdarma. Implementaci algoritmů zpracování obrazu budou zajišťovat knihovny projektu EmguCV[2]. Jedná se o .NET wrapper nad nejčastěji používanou opensource knihovnou OpenCV, ve které je implementováno velké množství algoritmů pro zpracování obrazu.

[1] Web projektu http://www.dataweb.de/en/products/diagramming.html
[2] Web projektu http://www.emgu.com/wiki/index.php/Main_Page

2 Zpracování obrazu

Pokud hovoříme o zpracování obrazu, je tím myšlena jakákoliv forma zpracování informací, kdy na vstup přichází obraz a výstupem může být zpracovaný obraz nebo sada vlastností vstupního obrazu. Obraz je v tomto případě možné definovat jako funkci f(x, y), kde x a y jsou souřadnice v rovině a funkční hodnota je hladinou jasu v daném bodě. Pokud jsou hodnoty x, y a f konečné a diskrétní jedná se o obraz digitální.

Samotné zpracování se zpravidla provádí v několika krocích. Jejich pořadí ani provedení však není pevně dané a závisí čistě na konkrétní aplikaci. Každý z níže uvedených kroků se může skládat z několika dalších dílčích operací, případně se mohou i jednotlivé kroky libovolně v různých modifikacích během zpracování opakovat [10] [11].

2.1 Snímání obrazu

Snímáním jsou převáděny vstupní optické veličiny na spojitý elektrický signál. Následným krokem potřebným pro další zpracování obrazu je jeho digitalizace, jelikož počítač není schopen pracovat s nekonečným množstvím dat, které tento signál obsahuje. Digitalizací je tedy převeden spojitý signál do diskrétního tvaru za použití vzorkování a kvantování. Vzorkováním je signál rozdělen na předem daný počet intervalů a jsou vybrány pouze funkční hodnoty v daných bodech. Při kvantování je obor hodnot obrazové funkce rozdělen na intervaly a získané funkční hodnoty jsou pak obvykle nahrazeny průměrnou hodnotou intervalu, do kterého spadají. Výsledkem je matice MxN bodů v závislosti na použitém vzorkování s prvky nabývajících K hodnot daných zvoleným kvantováním [10]. Pro naše potřeby se předpokládá, že vstupem je již pořízený digitální obraz a problematika jeho pořízení nebude v práci řešena.

2.2 Předzpracování obrazu

Digitalizovaný obraz je vhodnými metodami upravován tak, aby bylo možné následně vyhodnocovat informaci obsaženou v obraze ať už pro další analýzu obsahu, identifikaci objektů nebo jen zvýraznění sledovaných rysů pro pozorování člověkem. Jedná se o soubor základních postupů, které lze zařadit do několika skupin.

2.2.1 Vylepšení obrazu

Jde o takový proces úpravy obrazu, aby výsledný obraz byl vhodnější pro další zpracování. Tento krok je možné si představit jako běžnou úpravu digitálních fotografií. Například vyčištění od šumu a zkreslení, úprava jasu, histogramu a kontrastu, zaostření obrazu, změna velikosti, oříznutí atd.

2.2.2 Restaurace obrazu

Při restauraci obrazu jsou odstraňovány chyby vzniklé například při rozostření objektivu nebo snímáním pohybujícího se objektu při dlouhých expozičních časech. Cílem je využít apriorní znalosti matematického modelu poruchy a vyřešení inverzní úlohy [12]. Od vylepšení obrazu se liší tím, že je odstraněna původní chyba a výsledná data jsou realistická. Obraz není upraven pouze tak, aby pro pozorovatele vypadal lépe.

2.2.3 Úprava barev

Změna barev v obrazu je využita ke zvýraznění a získání hledaných rysů. Může jít o pouhé zvýšení kontrastu pro snadnější zkoumání člověkem, nebo pro lepší prezentaci zobrazených dat. Patří sem také převody mezi barevnými modely nebo paletami barev. Využívány jsou nejvíce různé formy prahování tj. převodu obrazu do binární formy, kde hodnoty jasu nabývají pouze dvou hodnot 0 a 1. V tomto případě je určena hranice jasu ve zdrojovém obrazu. Pixelům s hodnotou jasu nad hranicí je nastavena ve výsledném obrazu hodnota jasu 1 a ostatním 0. Opačným postupem lze naopak z monochromatického obrazu vytvořit obraz pseudobarevný [13] , kde výsledná barva pixelu odpovídá barvě přiřazené k dané hladině jasu. Toho se využívá při prezentaci obrazů, kde jsou rozdíly jasů velmi malé, nebo například při zobrazování třetího rozměru či teploty.

2.2.4 Transformace signálu

Transformace signálu má velmi široké využití a přesahuje i do ostatních kroků zpracování obrazu. Umožňuje nám detekovat hrany, objekty, restaurovat obraz, odstranit šum, extrahovat příznaky, sestavit pyramidovou reprezentaci obrazu (obraz je uchován v několika úrovních rozlišení) nebo jeho kompresi [11]. Komprese může být samostatným krokem a slouží ke zmenšení objemu dat potřebného k uložení, zpracování nebo přenosu obrazové informace. Nejčastěji se používá Fourierova

transformace. Jedná se o integrální transformaci pro převod signálů z časové oblasti do oblasti frekvenční. V případě diskrétního obrazu je používána diskrétní Fourierova transformace, která je však časově náročná $O(N^2)$ kvůli velkému počtu násobení a proto byla vyvinuta rychlá Fourierova transformace s časovou náročností O(N log N) [14]. Pro signály závislé na poloze počátku časové osy, kde nestačí použít Fourierovu transformaci, se používá vlnková transformace [15]. Můžeme na ni nahlížet jako na konvoluci vhodně zvolených vlnek (časově omezené funkce) s analyzovaným signálem. V případě diskrétního obrazu je opět využita diskrétní vlnková transformace.

2.2.5 Matematická morfologie

Matematický nástroj vycházející z teorie množin pro extrakci obrazových komponent, které jsou užitečné pro reprezentaci a popis tvaru oblasti. Využívá se zde relace obrazu a strukturního elementu, což je předem definovaný tvar. Pro každý pixel se porovnává strukturní element s okolím vybraného pixelu. Vstupem je binární obraz nebo obraz v odstínech šedi [16]. Cílem morfologických operací jsou kvantitativní analýza obrazu (počet, rozměry a orientace objektů) a zpracování obrazu potlačením šumu, vyhlazením hran, eliminací defektů, hledáním obrysů, skeletů apod. Základními operacemi jsou dilatace a eroze. Dilatace se používá k zaplnění malých děr a úzkých zálivů v objektech, výsledný objekt je zvětšený. Duální transformací k dilataci je eroze. Ta objekty zmenší, složitější rozdělí na jednodušší části a objekty menší než strukturní element odstraní. Pokud odečteme erodovaný obraz od původního, získáme hrany objektů [17].

2.3 Analýza obrazu

2.3.1 Segmentace obrazu

Segmentace obrazu je nejobtížnější částí v procesu zpracování. Při segmentaci se snažíme obraz rozdělit na nepřekrývající se dílčí části, kterými mohou být celé oblasti nebo konkrétní objekty zájmu. Pokud je výsledkem segmentace soubor oblastí, které odpovídají objektům ve vstupním obraze, jedná se o segmentaci kompletní. Kompletní segmentace vyžaduje zpracování se znalostí řešeného problému tak, abychom byli schopni přesně definovat hledané objekty. V opačném případě, kdy nalezené oblasti objektům přesně neodpovídají, nazývá se tato segmentace částečnou. K částečné

segmentaci postačuje například využití homogenity obrazových vlastností (např. jas, barva, textura) a lze ji úspěšně použít ke zjištění počtu objektů v obraze. Při segmentaci jsou aplikovány metody předchozích kroků zpracování, mezi nejčastěji používané je možné zařadit například prahování, detekci hran a hledání oblastí [10].

2.3.2 Popis nalezených objektů

Popis objektů lze provádět dvěma základními způsoby. Prvním je kvantitativní způsob, ten popisuje objekt pomocí souboru číselných charakteristik (např. plocha objektu, velikost). Druhou možností je způsob kvalitativní, ve kterém jsou popsány vztahy mezi jednotlivými objekty a jejich tvarové vlastnosti. Volba metody popisu je zvolena vždy na základě předpokládaného využití získaných informací. Ve většině případů jsou získaná data využita v posledním kroku zpracování ke klasifikaci objektů [10].

2.3.3 Klasifikace

Klasifikace je posledním krokem zpracování, při kterém se objekty nalezené v obraze zařazují na základě daného pravidla – klasifikátoru do jednotlivých tříd. V tomto kroku se tedy pokoušíme zjistit, co je na obrázku za objekty, případně zda se hledaný objekt v obraze nachází. Metody klasifikace můžeme rozdělit do dvou skupin v závislosti na tom, který způsob popisu objektu využívají. Příznakové metody pracují s vektorem číselných charakteristik získaných při popisu objektu kvantitativním způsobem. Použitý klasifikátor lze nastavit pomocí učení s učitelem. To vyžaduje trénovací množinu, což je soubor prvků s předem známou příslušností k třídě. Pokud nemáme k dispozici trénovací množinu, nebo předem neznáme počet tříd, je možné využít některou z metod shlukové analýzy [18]. Naproti tomu strukturální rozpoznávání zpracovává kvalitativní popis objektů. Ke klasifikaci jsou pak použity předem známé gramatiky, které kontrolují syntaxi nalezených objektů. Objekty jsou popsány pomocí slov poskládaných z jednotlivých primitiv definujících vlastnosti objektu. Primitiva tak tvoří abecedu, nad kterou gramatika svými pravidly definuje jazyk. Díky tomu je možné rozhodnout o příslušnosti slova (objektu) k jazyku (třídě) [10].

3 Využití zpracování obrazu

Zpracování obrazu má širokou škálu uplatnění i v oboru biologie [4] [6] [19]. Je důležité mít na paměti, že každý řešený problém je zcela unikátní. Ačkoliv existuje velké množství vyzkoušených a doporučených postupů jejich aplikace pro řešení problému vždy vyžaduje specifické úpravy. To je dáno především vlastnostmi vstupních dat, jejich rozdílnou kvalitou a požadovaným výstupem. Jako příklad můžeme uvést detekci zvěře v záznamu z fotopasti. Postup, který bude umět identifikovat zachycený živočišný druh za ideálních světelných podmínek, přestane fungovat v momentě, kdy se setmí, nebo se bude objekt pohybovat příliš rychle atp. V některých případech bude postačovat úprava parametrů jednotlivých kroků zpracování, jiné si však mohou vyžádat využití odlišných postupů a metod předzpracování (např. zaostření rozmazaného objektu, odstranění šumu při nízké světelnosti). I přes to se můžeme pokusit rozdělit řešené úlohy do skupin dle výsledku, kterého chceme aplikací zpracování obrazu docílit. Některé z uvedených skupin mohou a také tvoří dílčí kroky náročnějších úloh.

3.1 Úprava obrazu

Zpracování obrazu začíná již v momentě, kdy chceme pořízený obraz nějak upravit. Může se jednat o oříznutí, změnu velikosti případně formátu nebo kompresi, tak aby byl na výsledném obraze viditelný pouze požadovaný objekt, nebo bylo možné tento obraz poslat elektronickou poštou, či vystavit na web.

Náročnější úpravy mohou zahrnovat odstraňování chyb vzniklých při pořizování snímku. Především snímky pořízené v terénu jsou velmi často rozmazané, zrnité nebo mají nevyvážený histogram. Je to dáno tím, že okolnosti ne vždy umožňují pořízení kvalitního snímku. Světelné podmínky lze ovlivnit jen velmi těžko, některé jevy nebo živočichy lze pozorovat pouze v konkrétní denní dobu, stejně tak snímání vzdáleného rychle se pohybujícího objektu má za následek nekvalitní fotografii. V některých případech je nutné se vypořádat s důsledky nevhodně zvolené metodiky pořízení, kdy například snímání objektu umístěného na čtvercové síti, která měla původně usnadnit měření objektu, nyní ztěžuje jeho další zpracování. Všechny tyto chyby lze do jisté míry použitím vhodných metod předzpracování potlačit, nebo odstranit úplně.

Poslední skupinou jsou úpravy, které mají za cíl získat z obrazových dat novou informaci. Může to být taková úprava, která usnadní pozorování požadovaného jevu,

nebo zvýrazní sledované rysy, například ve snímcích z digitálních mikroskopů. K tomu lze s úspěchem využít především změny jasu, kontrastu, histogramu nebo barevného schématu. Dále pak lze například údaje o teplotách ze snímků pořízených infrakamerou prezentovat pomocí pseudobarev, kdy je určitému rozsahu teplot přiřazena jiná barva. Tento postup je aplikovatelný i na zvýraznění málo zřetelných jasových přechodů v rentgenových snímcích [20].

3.2 Určení počtu objektů

Velmi častým úkolem bývá zjištění počtu objektů v pořízeném snímku. Jako příklad lze uvést mikroskopický snímek buněčné struktury, bakterií [21] nebo fotografie listů napadených mšicemi [19]. Takovéto úlohy lze řešit použitím vhodného prahování a následnou částečnou segmentací. Hlavním předpokladem je však, že jsme prahováním schopni oddělit sledované objekty od pozadí a tyto objekty nepotřebujeme rozlišovat na základě nějakých kritérií. V opačném případě je nutné využít vyšších metod zpracování, detekci objektů a jejich rozpoznání na základě některé z klasifikačních metod. Určité úlohy nevyžadují znát přesný počet objektů, ale postačuje například jen procentuální zastoupení plochy v nasnímané scéně, jako je tomu například při detekci virózy obilnin [6]. V takovém případě je využito pouze prahování a z výsledného binárního obrazu je možné poměrem černých a bílých pixelů určit kolik procent z celkové plochy zabírá objekt zájmu. Takto lze zjišťovat poměr zastoupení více ploch. Pokud máme pro každou sledovanou plochu odpovídající binární obraz, můžeme s nimi provádět množinové operace a určit celkovou plochu nebo jejich průnik.

3.3 Zjištění kvantitativních parametrů objektu

V některých případech nás při zkoumání objektů na pořízených snímcích mohou zajímat jejich kvantitativní vlastnosti, jako jsou například rozměry, velikost, míra zakřivení atd. Příkladem mohou být nejen měření mikroskopovaných buněk, cizopasných hub nebo fytoparazitických háďátek [19] ale i hmyzu a rostlin apod. Abychom však mohli nad získanými obrazovými daty provádět nějaká měření v běžně používaných jednotkách SI, potřebujeme nutně určit velikost jednoho pixelu. Kalibraci lze provést například umístěním objektu známé velikosti do obrázku. Následným vyznačením této vzdálenosti v obrazové matici jsme schopni na základě podílu reálné

vzdálenosti a počtu pixelů zjistit velikost, kterou jeden pixel představuje. V tento okamžik je možné všechny údaje naměřené v pixelech převádět do reálných jednotek. Měření můžeme provádět ručně, obdobným způsobem jakým jsme postupovali při kalibraci, tedy vyznačením krajních bodů resp. oblasti. Některé úlohy lze více automatizovat. Například v předzpracovaném obraze, kdy postupujeme obdobně jako při zjišťování počtu objektů, můžeme určit obsah nebo poloměr detekovaných objektů. Pokud bychom ale chtěli měřit konkrétní část objektu, museli bychom být nejdříve schopni tuto co nejpřesněji detekovat, což je úloha vyžadující užití vyšších metod zpracování obrazu.

3.4 Analýza pohybu

Analýza pohybu zahrnuje dva typy úloh. V obou případech je zpracováváno více vstupních obrazů, může se jednat například o snímky pořízené videokamerou. Do první skupiny můžeme zahrnout prostou detekci pohybu. Takto je možné třeba analyzovat dlouhý filmový záznam z kamery monitorující výskyt zvěře a vyhledat pasáže, kdy byl ve scéně detekován pohyb. Rozdílem hodnot jasu ve všech bodech obrazové matice dvou po sobě jdoucích snímků a vhodným prahováním výsledku k odstranění šumu velmi snadno určíme, kde došlo ke změnám. Kromě informace o tom, že k pohybu došlo resp., že se zachycená scéna v průběhu snímání změnila, však další údaje nezískáme.

Druhou skupinou jsou právě úlohy, ve kterých nás zajímá směr a rychlost pohybu. Rozšířením výše zmíněného postupu o detekci významných bodů jsme schopni určit vektor jejich rychlosti [22]. V případě, kdy potřebujeme sledovat pohyb jednotlivých objektů jako například při analýze pohybu mikroskopovaných buněk [4], musíme je nejprve detekovat. Následně je třeba zajistit, aby ten samý objekt byl správně rozpoznán v každém následujícím snímku. K tomu se využívá zejména aktivních kontur [23].

3.5 Rozpoznání objektu

Se zvětšujícím se množstvím pořízených dat je stále větší snaha analýzu jejich obsahu co nejvíce automatizovat. Jedná se především o vyhledávání objektu, který odpovídá předem známé šabloně a náročnější úlohu rozpoznání objektu ve scéně a jeho klasifikaci. Jako příklad můžeme zmínit rozpoznávání druhů pylových zrnek, které je

velmi obtížné i pro odborníky v oboru [24], nebo zařazení živočišného druhu z fotografií zachycených fotopastí. Problémem však je, že zatím neexistuje žádný universální algoritmus, který by dokázal rozpoznat libovolný objekt v jakékoliv scéně. Každou úlohu je tak nutné řešit specifickým postupem včetně vhodně zvolené metodiky pořízení vstupních dat. Momentálně jsou nejefektivnější algoritmy založeny na konvolučních neuronových sítích [25]. Další metody využitelné pro vyhledání nebo klasifikaci objektů lze najít v [26].

4 Dostupné nástroje

Softwarových produktů, které se zabývají zpracováním obrazu, je velké množství. To je také jedním z hlavních problémů, jelikož je velmi obtížné se v nich zorientovat a zvolit ten nejvhodnější k řešení dané úlohy. Využívání různých aplikací k řešení odlišných úloh zas klade velké nároky na uživatele. Vyžaduje znalost práce s každým jedním nástrojem, což může být velmi náročné, jelikož je filosofie jednotlivých aplikací mnohdy velmi odlišná. Jak již bylo zmíněno v úvodu, existuje mnoho velmi pokročilých proprietárních řešení, o kterých ale nevíme, jak fungují uvnitř. Využívají vlastní formát, který znemožňuje další zpracování výsledků, a mohou být svázány s konkrétním hardware. Na druhé straně pak stojí knihovny s otevřeným kódem, který můžeme detailně prozkoumat a využít k tvorbě řešení přímo na míru. To ale vyžaduje vyšší programátorské znalosti a vývoj příslušného software je časově náročný nehledě na potřebu jeho otestování a odladění. Takovýto postup tedy není vhodný, pokud nemáme dostatek času a potřebujeme zpracování provádět ihned. Existující nástroje zpracování obrazu se můžeme pokusit rozdělit do několika skupin dle jejich povahy.

4.1 Jednoúčelové

Jednoúčelovými nástroji nejsou myšleny pouze aplikace, které řeší jeden konkrétní úkol jako například program Peacock, který zpracovává rentgenové snímky [20]. Patří sem všechny aplikace, které lze využít k řešení omezené skupiny úkolů, především proto, že jejich funkčnost není možné nijak rozšířit.

- **AxioVision**[3]

  AxioVision je komerční software od firmy ZEISS určený primárně pro práci s hardwarovým vybavením této firmy. Je zaměřený na zpracování obrazů pořízených digitálními mikroskopy. Zajišťuje celý proces zpracování od ovládání periferií při pořizování snímků až po ukládání výsledků zpracování a jejich správu ve vlastním proprietárním formátu ZVI (Zeiss Vision Image). Aplikace je spustitelná pouze v prostředí operačního systému Windows, k dispozici je zkušební verze s omezenými funkcemi. Aplikaci je možné rozšiřovat množstvím modulů, které lze k základním funkcím dokoupit. Jedná se

[3] Web projektu http://www.zeiss.com/microscopy/en_de/products/microscope-software/axiovision-for-biology.html

především o podporu periferií nebo nové metody zpracování. Plně integrován je VBA (Visual Basic for Applications), což umožňuje modifikovat aplikaci s využitím stávajících funkcí programu [27].

- **QuickPHOTO**[4]

  QuickPHOTO je komerční produkt firmy PROMICRA. Umožňuje pořizovat snímky z podporovaných digitálních mikroskopů a fotoaparátů a provádět ruční měření velikostí objektů na pořízených snímcích. Program je dostupný ve třech verzích, které se liší nabízenými funkcemi (např. možnost měření úhlů, ploch a počítání objektů), a je možné ho provozovat pouze na operačním systému Windows. Rozšíření aplikace je možné třemi moduly pro záznam videosekvence, vytváření snímků s větší hloubkou ostrosti a skládání snímků pořízených fluorescenčním mikroskopem.

4.2 Rozšíření

Některé nástroje byly vyvinuty jako rozšíření existujících aplikací, které tak zprostředkovávají přístup k různým výpočetním nástrojům, grafickému rozhraní a pokročilejším formám skriptování. Jedná se především o aplikace RapidMiner[5] a MATLAB[6]. K využití těchto rozšíření je nutné vlastnit licenci pro základní software. Může jít o vhodnou alternativu v případě, že takovou licenci již vlastníme.

- **IMMI - Image Mining Extension for RapidMiner**[7]

  Rozšíření pro aplikaci RapidMiner s otevřeným kódem vytvořené skupinou SPLab, která je součástí Vysokého učení technického v Brně. Distribuováno je pod licencí AGPL stejně jako RapidMiner samotný. Rozšíření je dostupné zdarma a je zaměřeno především na dolování dat. Uživatel modeluje postup zpracování v grafickém rozhraní spojováním dostupných modulů, které provádějí základní operace a u nichž lze modifikovat jejich parametry. RapidMiner je komerční software, ale nabízí komunitní verzi volně ke stažení. Program je napsán v jazyce Java a ke svému běhu vyžaduje JRE (Java Runtime

[4] Web projektu http://www.promicra.cz/produkty-quickphoto-micro.php
[5] Web projektu https://rapidminer.com/
[6] Web projektu http://www.mathworks.com/products/matlab/index.html
[7] Web projektu http://splab.cz/en/research/data-mining/articles

Environment) verzi 7 a vyšší. Díky tomu možné ho spustit na nejrozšířenějších operačních systémech jako Windows, Linux a OS X.

- **Image Processing Toolbox**[8]

  Obdobným způsobem je řešeno rozšíření programu MATLAB od firmy MathWorks, které je nabízeno jako standardní modul výše zmíněné aplikace. MATLAB využívá vlastní programovací jazyk. Uživatel sestavuje algoritmus zpracování z dostupných funkcí, které jsou modulem rozšířeny o ty určené ke zpracování obrazu. Modul je primárně zaměřen na úpravy obrazu, jeho předzpracování, segmentaci a detekci objektů. Jak základní software, tak i modul jsou komerční produkty a nemají žádnou bezplatnou verzi. Zdrojový kód není k dispozici. Aplikaci je možné provozovat na všech běžně rozšířených operačních systémech Windows, Linux a OS X. Dalším podobným rozšířením je ještě Computer Vision System Toolbox[9]. To se na rozdíl od Image Processing Toolboxu specializuje na počítačové vidění, především rozpoznávání a klasifikaci objektů, sledování jejich pohybu a zpracování videosekvencí.

4.3 Modulární aplikace

Další kategorií jsou samostatně fungující aplikace, které mají vlastní uživatelské rozhraní a obsahují již v základu sadu operací k využití. Důležitým aspektem je však možnost jejich rozšiřování o uživatelem vytvořené moduly. Ty mohou implementovat nové operace a algoritmy zpracování obrazu, ale i zcela jinou funkčnost včetně úprav uživatelského rozhraní. Některé aplikace uvedené v předchozích kapitolách jsou z principu modulární. Do této kategorie ale nespadají buď proto, že lze aplikaci rozšířit pouze o moduly nabízené výrobcem, nebo nové funkce a úpravy lze provádět pouze s využitím již existujících funkcí programu. Tato kategorie tedy zahrnuje aplikace, většinou s otevřeným kódem, které lze programově, nebo s využitím definovaného rozhraní rozšířit o libovolnou funkci.

[8] Web projektu http://www.mathworks.com/products/image/index.html
[9] Web projektu http://www.mathworks.com/products/computer-vision/index.html

- **ImageJ**[10]

  Velmi často využívaný nástroj je ImageJ. Experimentální systém vyvíjený National Institutes of Health, jedním z oddělení ministerstva zdravotnictví ve Spojených státech amerických. Jedná se o volné dílo šiřitelné volně včetně zdrojových kódů. Program je psán v jazyce Java a k běhu vyžaduje verzi 1.6 a vyšší. Je možné ho spustit na operačních systémech Windows, Linux a OS X nebo ve webovém prohlížeči jako applet. Aplikace umožňuje prováděné akce nahrávat jako makro. Takto vytvořené makro lze pak spouštět nad velkým množstvím vstupních obrazů. Aplikaci je možné rozšířit vlastními moduly napsanými v jazyce Java. Existuje také více než 150 již hotových modulů, které řeší specializované úlohy zpracování obrazu, nebo poskytují rozšíření grafického rozhraní například v podobě nástroje pro grafické modelování procesu zpracování.

- **Cameléon**[11]

  Zajímavým projektem, který lze také zařadit do této kategorie je Cameléon. Cílem skupiny vývojářů s názvem shinoe je vytvořit nástroj, který umožní rychlé vytváření prototypů a testování algoritmů jejich skládáním a propojováním v intuitivním grafickém rozhraní. Ačkoliv je Cameléon navržen obecně pro jakékoliv zpracování, jsou k dispozici již hotové nástroje pro práci s obrazovými daty. K tomu využívají vlastní knihovnu algoritmů Population[12]. Program je s otevřeným kódem, volně šiřitelný pod licencí MIT. Spustit jde momentálně na operačním systému Linux. Aplikaci lze rozšiřovat o vlastní knihovny psané v jazyce C++. K tomu slouží SDK (Software development kit), díky tomu je možné vytvářet prototypy libovolných uživatelských aplikací. Nevýhodou celého projektu je velmi pomalý vývoj, nekompletní dokumentace a nedostatek dalších informací. Na rok 2015 je plánována verze 2.1.0, která možná tyto nedostatky odstraní.

[10] Web projektu http://imagej.nih.gov/ij/index.html
[11] Web projektu http://www.shinoe.org/cameleon/
[12] Web projektu http://www.shinoe.org/population/

4.4 Orchestrace více nástrojů

Zvláštní skupinou jsou nástroje, které přímo neprovádějí zpracování obrazu, ale jsou určené k propojení několika různých aplikací nebo systémů, kdy jsou výstupy jednotlivých aplikací předávány jako vstupy jiných. Tímto způsobem je možné zjednodušit a automatizovat procesy zpracování, které zahrnují využití několika různých softwarových nástrojů.

- **Taverna**[13]

  Taverna je robustní systém na správu workflow, vyvíjený týmem pod vedením profesorky Carole Goble z Manchesterské university. Jeho hlavním cílem je umožnit vytváření pracovních postupů, kdy je ke zpracování dat nutné využít více aplikací nebo veřejně dostupných datových zdrojů a dalších služeb. Můžeme tedy nastavit v jednom software pořízení snímku z digitálního mikroskopu, předat jej k předzpracování v dalším a výstup následně porovnat například s daty v databázi BioMart[14]. K dispozici je grafický návrhář a celé workflow pak lze odeslat a spouštět ze serverové části aplikace. Taverna umí pracovat s webovými službami, aplikacemi, které implementují požadované rozhraní nebo spouštět skripty na cílovém systému. Aplikace je s otevřeným kódem pod licencí LGPL [28], psaná v jazyce Java pro běh na operačních systémech Windows, Linux a OS X.

4.5 Knihovny a frameworky

Tato kategorie obsahuje nástroje na úrovni sad algoritmů zpracování obrazu a programových celků, které lze využít při tvorbě vlastních aplikací. K jejich použití je nutná znalost použitého programovacího jazyka a vývoje aplikací obecně.

- **OpenCV**[15]

  Nejznámější knihovnou používanou v oboru zpracování obrazu je OpenCV (Open Source Computer Vision Library). Je to knihovna s otevřeným kódem obsahující přes 500 funkcí určených především pro využití při zpracování obrazu a počítačovém vidění. Je psaná v jazyce C/C++ s důrazem na vysoký

[13] Web projektu http://www.taverna.org.uk/
[14] Web projektu http://www.biomart.org/
[15] Web projektu http://opencv.org/

výkon a podporu vícejádrových procesorů. Momentálně dostupná stabilní verze je 2.4.10. Knihovna je k dispozici pod licencí BSD, lze ji tedy využít i v komerčních projektech. Nabízí rozhraní pro jazyky C++, C, Python a Java a podporuje operační systémy Windows, Linux, Mac OS, včetně mobilních iOS a Android. Existují také dva projekty OpenCVSharp[16] a Emgu CV, které mají za cíl umožnit využití knihovny OpenCV v aplikacích postavených na .NET Frameworku.

- **BoofCV**[17]

  Alternativou pro OpenCV je v jazyce Java BoofCV. Při porovnání rychlostí s OpenCV dosahuje BoofCV lepších výsledků u vyšších úrovní zpracování díky lepší implementaci algoritmů, naopak u nízkoúrovňového zpracování zaostává při aritmetických operacích s poli, což je dáno použitým programovacím jazykem [29]. Použité algoritmy jsou validovány oproti původním implementacím z vědeckých prací, kde byly představeny a je prováděno jejich regresní testování [30]. Knihovna je s otevřeným kódem, distribuována pod licencí Apache 2.0 k použití i v komerčních projektech.

- **AForge.NET**[18]

  AForge.NET je framework, který obsahuje několik knihoven s různým zaměřením od zpracování obrazu, počítačové vidění přes neuronové sítě až po podporu robotických souprav. Je napsán v jazyce C#, s otevřeným kódem a distribuovaný pod licencí LGPL. Za zmínku stojí nástroj Image Processing Prototyper (IPPrototyper), který je součástí frameworku a slouží k rychlému vytváření prototypů algoritmů zpracování obrazu. Tato aplikace umožňuje sestavit pořadí jednotlivých kroků zpracování a aplikovat je na skupinu vybraných vstupních obrazů. Dostupné metody lze rozšiřovat o vlastní implementace algoritmů. Aplikace obsahuje systém správy rozšíření. Všechny metody jsou implementovány jako samostatné knihovny. Ty jsou při spuštění načteny a v aplikaci zpřístupněny.

[16] Web projektu https://github.com/shimat/opencvsharp
[17] Web projektu http://boofcv.org/index.php?title=Main_Page
[18] Web projektu http://www.aforgenet.com/framework/

5 Analýza

Dle popisu zpracování obrazu v kapitole 2 je patrné, že vstupní obraz je postupně upravován různými operacemi, kde výstup předchozí úpravy je vstupem úpravy následující. U každé operace lze nastavit hodnoty parametrů, které ovlivňují použitý algoritmus. Pořadí ani počet těchto operací není nijak limitován, je tedy možné, aby se některá z úprav v procesu vyskytovala vícekrát a to třeba i se stejným nastavením parametrů. Dále je nutné mít možnost přidávat další operace nad rámec stávajících operací, především pokud chceme při zpracování využít nový nebo upravený algoritmus, který nebyl dosud implementován. Tato funkčnost navíc zajistí, že bude aplikaci možné využít na libovolné zpracování dat sekvenčním způsobem a nebude omezena pouze na zpracování obrazu.

Jelikož vstupem může být jeden i více obrázků (např. videozáznam), musí být aplikace schopna celou dávku zpracovat dle uživatelem sestaveného procesu. V ideálním případě tak, aby všechny operace pracovaly paralelně a využívaly při svém běhu více vláken. Předpokládá se, že zpracování každého vstupu bude bezstavové, tudíž nebude nijak ovlivněno zpracování vstupu následujícího. Jádro aplikace by mělo být nezávislé na použitém grafickém rozhraní, což umožní případné spuštění vytvořeného procesu v prostředí serverového OS. Proces zpracování tak může být odladěn na běžné pracovní stanici pro jeden vzorek, následně uložen a spuštěn na výkonném serveru nad větším množstvím vstupních dat.

S tím souvisí volba formátu datového souboru popisujícího sestavený proces. Musí obsahovat použité operace, hodnoty jejich parametrů a vzájemné vazby, kterými jsou propojeny. Zápis dat by měl být v člověku srozumitelném formátu a musí být možné ho upravovat i mimo aplikaci například v textovém editoru. Pro uživatelsky přívětivější a intuitivnější tvorbu procesu je vhodné nabídnout uživateli grafické rozhraní. Ovládání by mělo využívat především operace drag and drop, tedy jednoduchým přetahováním vybraných objektů a jejich spojováním sestavit požadovaný postup s možností okamžitého náhledu na jeho výsledek.

Celé řešení by mělo být platformně nezávislé a podporovat alespoň tři nejrozšířenější operační systémy Windows, Linux a Mac OS.

5.1 Požadavky na aplikaci

Níže uvedené požadavky na výsledný program plynou z provedené analýzy.

5.1.1 Nefunkční požadavky

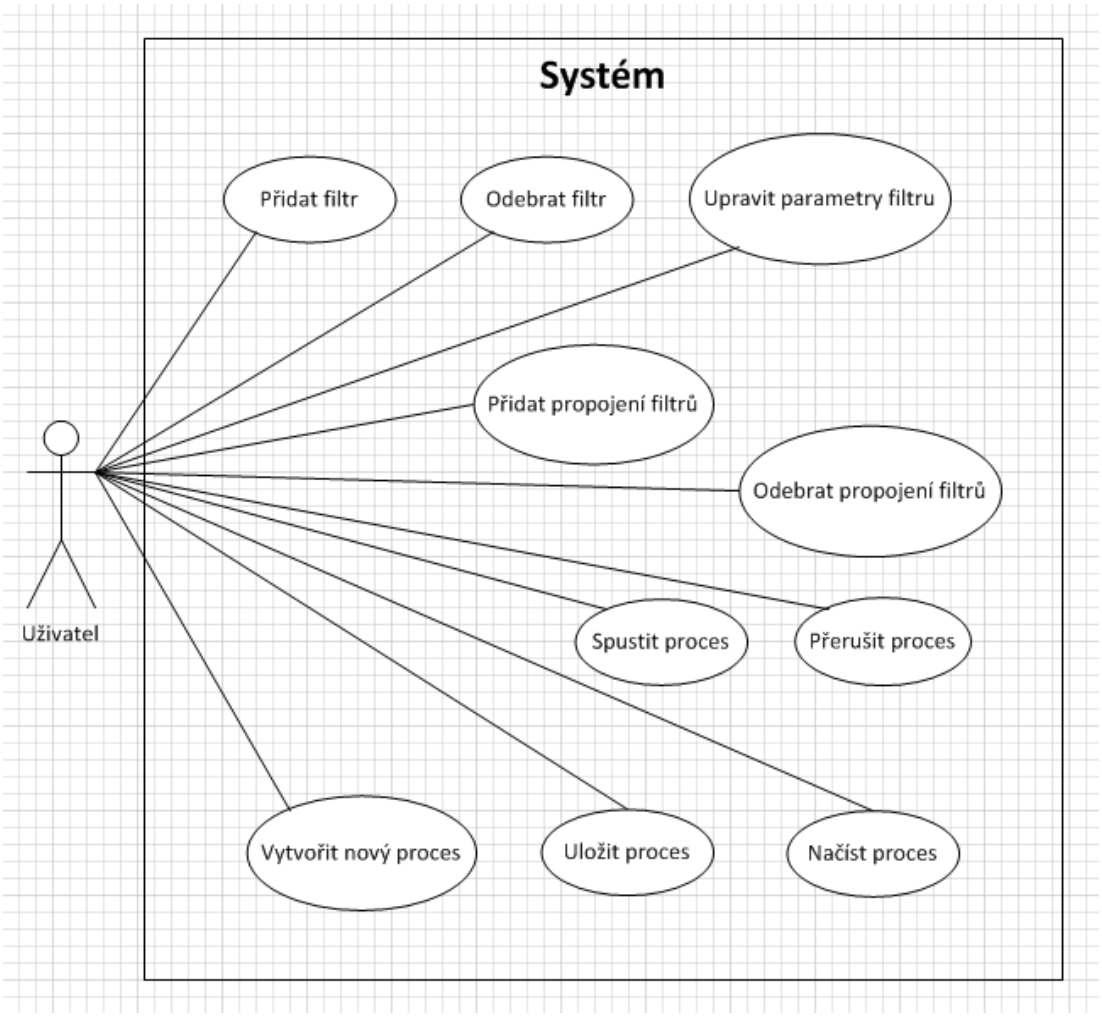
- Multiplatformní řešení (Windows, Linux, Mac OS)
- Vícevláknové zpracování
- Modulární – možnost přidávat další filtry
- Část zpracování nezávislá na GUI
- GUI, které umožní modelovat proces zpracování
- Proces zpracování lze modelovat i mimo aplikaci
- Datový soubor modelovaného procesu je člověkem čitelný
- Zdrojový kód je srozumitelný a dokumentovaný

5.1.2 Funkční požadavky

- Aplikace umožňuje import a export modelovaného procesu
- Aplikace umožňuje přidávat a odebírat filtry v procesu zpracování
- Aplikace umožňuje propojovat vstupy a výstupy filtrů
- Aplikace zobrazuje náhled výsledku zpracování
- Aplikace umožňuje filtrům nastavovat parametry
- Aplikace umožňuje dávkové zpracování vstupních dat
- Aplikace validuje vytvořený proces zpracování
- Aplikace umožňuje přerušit dávkové zpracování

5.2 Případy užití

Na základě seznamu funkčních požadavků byl vypracován diagram případů užití, které budou v aplikaci implementovány.



Obrázek 1: Use Case diagram

6 Návrh aplikace

6.1 Modulární architektura

Modulární architektura je jedním z možných způsobů návrhu softwaru, který je úzce spojen s provázaností entit. Hlavním cílem je rozdělení funkčnosti programu na nezávislé, zaměnitelné moduly, z nichž každý obsahuje vše nezbytné pro jediný aspekt požadované funkcionality [31]. Tento proces se nazývá oddělení zodpovědností (Separation of Concerns). Snahou je rozdělit program na takové části, které se ve své funkci budou co nejméně překrývat. To znamená, aby určitou funkcionalitu vykonávala pouze část programu k tomu určená [32]. Žádná z dalších částí programu by pak neměla tu samou funkcionalitu kopírovat, ale využít již existující část, která ji implementuje. Správným postupem při návrhu a vývoji je tak možné zvýšit přehlednost zdrojového kódu, jeho robustnost, spolehlivost, znovupoužitelnost a další rozšiřitelnost.

Modularity software lze dosáhnout na několika úrovních, obecně však platí, že jednotlivé moduly (části programu) definují rozhraní, které popisuje prvky modulem vyžadované a ty, jenž daný modul poskytuje ostatním k využití. Implementační část pak obsahuje výkonný kód, který odpovídá deklarovanému rozhraní.

Na úrovni zdrojového kódu jde především o jeho strukturu v závislosti na programovacím paradigmatu. Při strukturovaném programování je implementovaný algoritmus rozdělen na dílčí úlohy (procedury či funkce), které jsou spojovány v jeden celek. Využito je zároveň řídících struktur sekvence (vykonání sledu příkazů nebo procedur), výběru (vykonání příkazů na základě stavu programu) a opakování (příkazy jsou opakovány, dokud není dosaženo nějakého definovaného stavu programu). Objektově orientované programování přináší koncept objektů, což jsou datové struktury obsahující data ve formě polí (atributy) a procedury s výkonným kódem (metody). Objekty si udržují svůj vnitřní stav a poskytují metody, které mohou přistupovat k atributům objektu a měnit jejich hodnoty. Základní principy OOP přispívají k větší modularitě kódu. Zapouzdření skrývá vnitřní logiku a data objektu tak, aby k nim nebylo možné přistupovat přímo, ale pouze voláním metod objektu. Dědičnost využívá již existující funkčnost nějakého objektu, kterou je možné v jeho podtřídách dále

rozšířit. Polymorfismus pak umožnuje nahrazovat odkazované objekty pokud jsou podtřídou deklarovaného objektu, nebo implementují stejné rozhraní [33].

Aplikaci lze dělit i horizontálně na více vrstev. Sousední vrstvy spolu komunikují přes definovaná rozhraní, díky tomu mohou být zaměňovány, aniž by bylo nutné celou aplikaci přepracovávat. Přenos dat mezi vrstvami je součástí architektury. Bývá založen na standardních protokolech a technologiích, jako jsou CORBA, Java RMI, .NET Remoting, sokety, UDP nebo webové služby. Nejznámějším příkladem může být třívrstvá architektura, která se skládá z vrstvy prezentační (GUI), aplikační (logika aplikace) a datové (uchovávání a zpřístupnění dat, např. databáze) [33].

Z pohledu další rozšiřitelnosti systému je modularita jeho zásadním aspektem. Rozšiřitelností je míněna schopnost systému přidat novou funkcionalitu s minimálním nebo žádným dopadem na jeho stávající vnitřní strukturu a tok dat. Především pro změnu chování systému není nutné měnit zdrojový kód ani ho překompilovat. Metody rozšíření je možné rozdělit dle způsobu provedení [34].

- **Bílá skříňka**

  V tomto případě jsou změny prováděny ve zdrojovém kódu systému. V závislosti na tom, zda je měněn přímo původní zdrojový kód nebo je pouze rozšířen (např. dědičností) hovoříme o otevřené skříňce nebo o skříňce skleněné.

- **Černá skříňka**

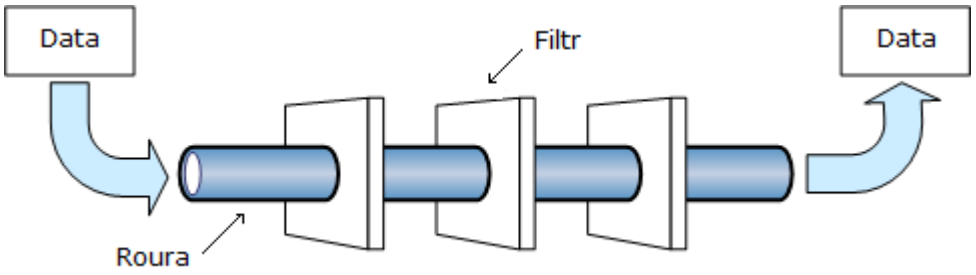
  Pokud nemáme k dispozici zdrojové kódy ani nevíme, jak systém vnitřně funguje, představuje pro nás černou skříňku, kterou je možné rozšiřovat například změnou konfiguračních souborů nebo vlastními skripty (pokud takovou možnost systém podporuje). Lze také případně využít rozhraní systému, ale bez hlubší znalosti jejich implementace.

- **Šedá skříňka**

  Šedá skříňka je kompromisem mezi výše uvedenými variantami. V tomto případě není nutné mít k dispozici zdrojový kód celého systému, ale pouze popis relevantních rozhraní sloužících k jeho rozšíření a případně návod k jejich využití.

6.2 Zpracování dat

Zpracování vstupních dat je definováno uživatelem jako sekvence jednotlivých operací, které se mají se vstupem provést. Vstupem každého následujícího kroku je výstup kroku předchozího. Tomu odpovídá návrhový vzor roury a filtry [35]. Filtry představují atomické a vzájemně nezávislé operace, které jsou propojeny rourami. Roury zajišťují přenos dat mezi filtry. Schéma návrhového vzoru je znázorněno na Obrázek 2.

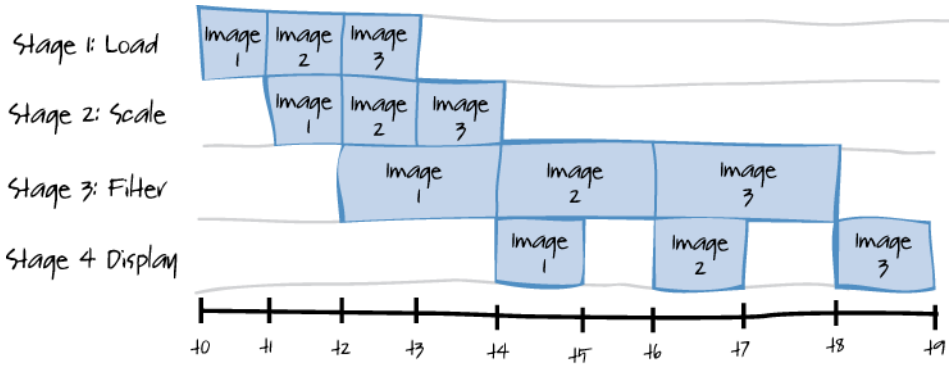


Obrázek 2: Filtry a roury[19]

V našem případě tak vstupní obrázek prochází postupně jednotlivými transformacemi a je postupně upravován, případně jsou prováděny další operace, které filtr implementuje. Ačkoliv je z pohledu uživatele proces zpracování sekvenční, v souvislosti s požadavkem na dávkové zpracování a maximální možné využití paralelismu je možné tento návrh upravit tak, aby byly dostupné systémové prostředky využity co nejefektivněji. Pokud bychom aplikovali tento proces na každý obrázek zvlášť, pracoval by vždy právě jeden filtr a následující obrázek by bylo možné začít zpracovávat až po dokončení předchozího.

Jakou vhodnou variantu původního návrhu je možné využít návrhového vzoru potrubí [36]. Oproti předchozímu návrhovému vzoru je každý filtr spouštěn jako samostatná úloha. Tím je docíleno toho, že všechny kroky zpracování mohou běžet zároveň a lze tak lépe využít dostupný hardware (např. vícejádrové procesory). Vstupní dávka postupně prochází soustavou filtrů a všechny filtry zpracovávají určitou část vstupní dávky, v ideálním případě zpracovává každý filtr jiný obrázek.

[19] Upraveno z https://bigballofmud.wordpress.com/2009/04/18/pipelines/

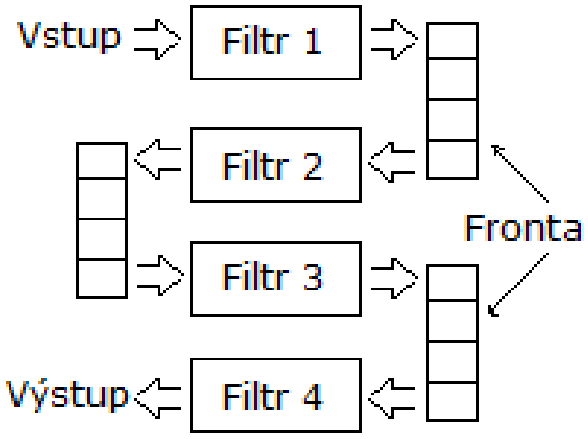


Obrázek 3: Postup zpracování vstupní dávky[20]

Analogii můžeme najít u pipeliningu v mikroprocesorech kdy je cyklus zpracování instrukcí rozdělen na několik části. V každém taktu může pak být zpracováváno několik instrukcí zároveň (každá se nachází v jiné části cyklu zpracování).

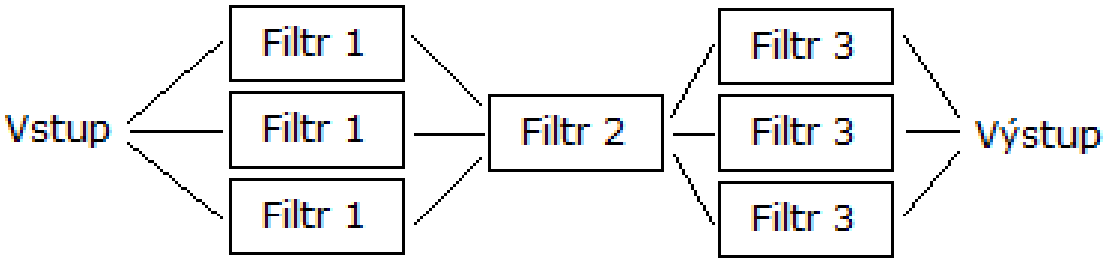
Na rozdíl od zpracování instrukce se doba činnosti jednotlivých filtrů vzájemně velmi odlišuje v závislosti na jejich výpočetní náročnosti a dalších faktorech. K tomu abychom docílili stavu na Obrázek 3, samotný paralelní běh zpracování nestačí. Rychlejší filtry by musely čekat na dokončení činnosti filtru následujícího. V tomto případě by druhý filtr čekal s hotovými výsledky zpracování druhého obrázku a mohl ho předat třetímu filtru až v momentě, kdy ten dokončí zpracování obrázku prvního, tedy o jeden časový úsek později. Součástí návrhového vzoru je i úprava roury a to tak, aby byla schopna fungovat jako fronta a umožnila vyrovnávat rozdílné rychlosti jednotlivých filtrů (viz Obrázek 4). K tomu lze využít blokující fronty. Blokující fronty řeší typický problém producenta a konzumenta, kdy dva procesy přistupují ke stejné vyrovnávací paměti. V takovém případě je nutné zajistit, aby se producent nesnažil přidávat prvky do již zaplněné fronty a konzument nepožadoval data z prázdné fronty. Pokud je dosaženo maximální kapacity vyrovnávací paměti, zajistí blokující fronta uspání procesu producenta do doby, než je nějaký prvek odebrán. Analogická je i situace na straně konzumenta, ten je blokován do doby, než je do prázdné fronty vložen další prvek [37].

[20] Převzato z https://msdn.microsoft.com/en-us/library/ff963548.aspx



Obrázek 4: Pipelines

Stále se však může stát úzkým hrdlem celého zpracování dlouho trvající operace, která způsobí zastavení všech filtrů předchozích, jelikož dojde k zaplnění blokujících kolekcí, a všech následujících, které zas budou čekat na vstupní data od pomalého filtru. Ve výsledku tak bude celý proces zpracování stejně rychlý jako nejpomalejší filtr v posloupnosti. Řešením by bylo vytvořit více instancí problematického filtru. Ty by si rozdělily vstupní data čekající v blokující frontě a zpracované výsledky vkládaly do kolekce výstupní (viz Obrázek 5).



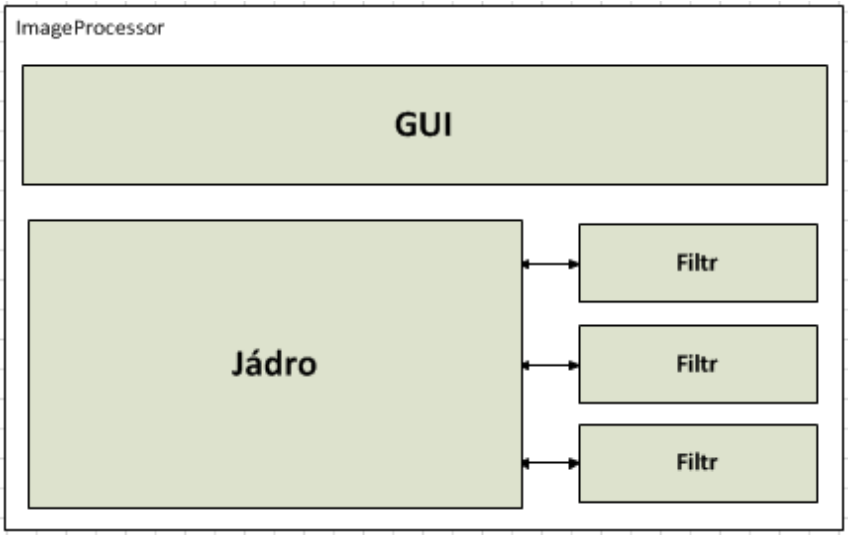
Obrázek 5: Loadbalancing filtrů

Tato varianta je ale implementačně velmi náročná především proto, že by program musel být schopen za běhu takovéto úzké hrdlo detekovat a nalézt optimální řešení. Řešení takového problému jde však za rámec této práce.

6.3 Schéma aplikace

Při vypracování návrhu je třeba nejprve vycházet z požadavku na modularitu řešení. Ta má zásadní vliv na celkovou stavbu aplikace a její komponenty. Dalším požadavkem je oddělení grafického rozhraní od výkonné části aplikace tak, aby byla odstraněna

jakákoliv závislost procesu zpracování na použitém rozhraní. Aplikaci tedy můžeme rozdělit na tři základní části, viz Obrázek 6.



Obrázek 6: Blokové schéma aplikace

- **Jádro**

  Jádro je výkonnou částí aplikace, která zajišťuje načtení a inicializaci všech modulů. Vytváří a uchovává modelovaný proces zpracování a provádí zpracování vstupních dat dle zadané posloupnosti operací. Je samostatnou komponentou, kterou lze využít i v jiných projektech.

- **Modul**

  Modul v našem případě představuje jednu atomickou operaci, kterou lze v procesu zpracování využít. Pro naše účely bude označena jako filtr, jelikož provede určenou operaci nad vstupními daty a výstup předává dále.

- **GUI/UI – Uživatelské rozhraní**

  Využívá rozhraní nabízené jádrem a slouží ke zprostředkování interakce mezi uživatelem a výkonnou částí aplikace. Uživatelské rozhraní může být realizováno jako grafické, nebo může mít formu textového terminálu příkazové řádky. Jádro není nijak závislé na implementaci UI.

6.4 Použité technologie

Zde je uveden krátký výčet softwarových technologií, které byly použity při implementaci aplikace. Programovací jazyk C# byl zvolen, jelikož autor práce s ním má dlouholeté zkušenosti při řešení firemních projektů.

6.4.1 Microsoft .NET Framework

Je framework od firmy Microsoft určený primárně pro vývoj aplikací pod operačním systémem Windows momentálně ve verzi 4.5.2. Framework se skládá ze dvou hlavních částí. První je Framework Class Library (FCL). Je to rozsáhlá knihovna, která zprostředkovává přístup k systémovým nástrojům, GUI, připojení k databázím, síťové komunikaci atd. Druhou částí je Common Language Runtime (CLR), což je virtuální stroj, který zajišťuje vykonání programového kódu. Díky tomu je framework teoreticky platformně nezávislý a umožňuje využití jakéhokoliv z podporovaných programovacích jazyků (např. C#, VB.NET) včetně možnosti použít v jednom programu současně více různých jazyků. Zdrojový kód je totiž zkompilován do sady instrukcí Common Intermediate Language (CIL), která je nezávislá na použité platformě a až při spuštění jsou tyto instrukce převedeny do strojového kódu v závislosti na CLR. K vývoji pod .NET Frameworkem nabízí Microsoft vlastní vývojové prostředí Visual Studio, aktuálně ve verzi 2013 [38].

6.4.2 NShape Framework

Jedná se o framework s otevřeným zdrojovým kódem a velkým množstvím funkcí pro tvorbu a práci s diagramy. Je k dispozici zdarma pro nekomerční projekty, aktuálně ve verzi 2.2.0, Využívá .NET Framework 2.0 a je psán v jazyce C#. Umožňuje dokonce propojení diagramu s objektovým modelem aplikační vrstvy a ukládání projektu do různých formátů včetně SQL databáze.

6.4.3 Emgu CV

Emgu je rekurzivní akronym pro Emgu is the Most General Unifier. Jedná se o multiplatformní wrapper nad knihovnou OpenCV. Celý je psaný pod .NET Frameworkem v jazyce C#. Díky tomu je možné využít funkcí OpenCV i v aplikacích vyvíjených například v C#. Navíc je možné takový program zkompilovat v projektu

Mono[21], což je open source implementace Microsoft .NET Frameworku. Ten umožňuje běh takto zkompilovaného kódu na operačních systémech jako iOS, Android, Windows Phone, Mac OS X a Linux.

[21] Web projektu http://www.mono-project.com/

7 Implementace

7.1 Struktura projektu

Celé řešení je rozděleno na několik dílčích projektů tak, aby byly zachovány principy modularity software zmiňované v kapitole 6.1. Projekty odpovídají navrženému schématu aplikace z kapitoly 6.3. Každý projekt řeší určitou část funkcionality aplikace.

- **`HJD.ImageProcessor.BaseFilters`**

  Tento projekt obsahuje implementaci sady základních filtrů, které zároveň mohou sloužit jako vzory pro implementaci vlastních filtrů.

- **`HJD.ImageProcessor.Core`**

  Jádro aplikace řeší načítání knihoven s filtry a jejich inicializaci. Součástí jádra je také vytváření grafu filtrů a vykonání zpracování samotného.

- **`HJD.ImageProcessor.Filters`**

  V tomto projektu je obsaženo vše, co se týká filtrů. Rozhraní, použité datové typy, parametry a roury. Tuto knihovnu musí odkazovat projekt, ve kterém chceme implementovat nové filtry.

- **`HJD.ImageProcessor.GUI`**

  Tento projekt implementuje grafické rozhraní, které využívá knihovnu NShape pro tvorbu diagramů a manipulaci s nimi. Jsou volány metody jádra, které zajišťuje vykonání požadovaných příkazů.

- **`HJD.ImageProcessor.GUI.FilterShapes`**

  Zde jsou definovány nové tvary objektů pro reprezentaci filtrů v diagramech.

Adresářová struktura odpovídá názvům projektů. Navíc jsou zde adresáře:

- **`/lib`** – obsahuje odkazované knihovny ve výše uvedených projektech,
- **`/NShape`** – obsahuje zdrojové kódy projektu NShape (při implementaci bylo nutné provést úpravy).

7.2 Jádro

Z pohledu aplikace je každý filtr modulem, který rozšiřuje funkčnost aplikace. Moduly se vzájemně nijak neovlivňují, ani na sobě nejsou nikterak závislé. Jak bylo zmíněno v kapitole 6.1, je důležité, aby přídání nové funkce bylo možné bez zásahu do apikace samotné. Bylo tedy zvoleno rozšiřování funkčnosti pomocí dynamických knihoven, které jsou při spuštění detekovány a v nich obsažené filtry jsou následně přidány do kolekce dostupných filtrů. O detekci a inicializaci filtrů se stará jádro aplikace resp. třída `ImageProcessorEngine`[22]. K implementaci načítání dynamických knihoven za běhu lze použít v prostředí Microsoft .NET Frameworku reflexi [39]. Implementace je ale relativně náročná a navíc od verze 4.0 je součástí .NET frameworku MEF [40] (Managed extensibility framework[23]). Ten je určen právě pro tvorbu rozšiřitelných aplikací a obstarává veškeré úkony spojené s načítáním externích komponent. Jeho implementace je velmi jednoduchá a navíc přináší několik užitečných funkcí, které budou zmíněny dále.

Každý modul musí implementovat rozhraní `IFilter` a atribut `FilterAttribute`[24]. V následujícím kódu vidíme, že tento atribut přidává k třídě filtru údaje o jeho názvu, počtu vstupů a výstupů. To nám umožnuje získat tyto informace aniž bychom museli vytvářet instanci dané třídy.

```
[MetadataAttribute]
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
public class FilterAttribute : ExportAttribute, IFilterAttribute
{
        public FilterAttribute(string filterName, int inputs, int outputs)
                : base(typeof(IFilter))
        {
                FilterName = filterName;
                Inputs = inputs;
                Outputs = outputs;
        }
        public string FilterName { get; private set; }
        public int Inputs { get; private set; }
        public int Outputs { get; private set; }
}
```

[22] Namespace HJD.ImageProcessor.Core
[23] Namespace System.ComponentModel.Composition
[24] Obojí v namespace HJD.ImageProcessor.Filters

Jádro aplikace ihned po spuštění prohledá výchozí adresář s moduly a všechny v něm obsažené dynamické knihovny. Umístění adresáře je uloženo v nastavení aplikace v souboru `app.config`[25].

```
<setting name="FiltersDirectory" serializeAs="String">
    <value>\filters</value>
</setting>
```

Všechny třídy z nalezených knihoven, které implementují rozhraní `IFilter` jsou přidány do kolekce. S použitím MEF je tento proces velmi jednoduchý. Základním principem MEF je práce s komponenty. Ty se v případě MEF nazývají části. Každá část deklaruje své závislosti (importy) a nabízené vlastnosti (exporty). Jakmile je nějaká část vytvořena, kompoziční engine MEF se pokusí nalézt takové exporty z dostupných částí, které odpovídají požadavkům importu. Jelikož části své schopnosti specifikují deklarativně, je možné s nimi pracovat i při běhu programu. To umožňuje vyhledání dostupných částí a zpracování metadat, které obsahují, aniž by bylo nutné vytvářet jejich instance nebo nahrávat odpovídající knihovny. Kolekce uchovávající dostupné filtry je tedy označena atributem `ImportMany`. Ta informuje MEF, že má být naplněna všemi dostupnými objekty typu `IFilter`. Za pozornost stojí použitá třída `ExportFactory`. Kolekce `Filters` neobsahuje přímo instance filtrů ale tovární třídu `ExportFactory`. Její dva generické parametry určují, jakého typu budou vytvářené objekty a zda mají být načtena i jejich metadata.

```
[ImportMany(typeof(IFilter), AllowRecomposition = true)]
private List<ExportFactory<IFilter, IFilterAttribute>> Filters {get; set;}
```

Dále je třeba určit, odkud se mají části načíst. K tomu slouží katalog. Jelikož filtry jsou uloženy v knihovnách ve výše zmíněném adresáři, je použit `DirectoryCatalog`. Ten zajistí načtení všech částí ze souborů v zadaném umístění, které odpovídají zvolenému filtru. V našem případě se jedná o dynamické knihovny.

```
AggregateCatalog catalog = new AggregateCatalog();
catalog.Catalogs.Add(new DirectoryCatalog(System.IO.Path.GetDirectoryName(
        System.Reflection.Assembly.GetExecutingAssembly().Location) +
        Properties.Settings.Default.FiltersDirectory, "*.dll"));
```

[25] Projekt HJD.ImageProcessor.Core

Posledním krokem je kompozice nalezených částí. Ta je zcela v režii MEF a provádí se následujícím příkazem.

```
CompositionContainer cc = new CompositionContainer(catalog);
cc.ComposeParts(this);
```

Po jeho vykonání je kolekce `Filters` naplněna všemi filtry, které byly v adresáři s moduly nalezeny. Pro práci s načtenými filtry jsou k dispozici dvě důležité metody.

Metoda `GetAvailableFilters` vrací seznam dostupných filtrů v podobě kolekce tříd `FilterMetadata`[26], což je pouze transportní objekt pro strukturovaná data popisující filtr. Ty je možné získat přes vlastnost `Metadata` třídy `ExportFactory`. Jedná se právě o ta metadata přidaná k filtru pomocí atributu `FilterAttribute`.

```
foreach (var filter in Filters)
{
        filterMetadata.Add(new FilterMetadata(
                filter.Metadata.FilterName,
                filter.Metadata.Inputs,
                filter.Metadata.Outputs));
}
```

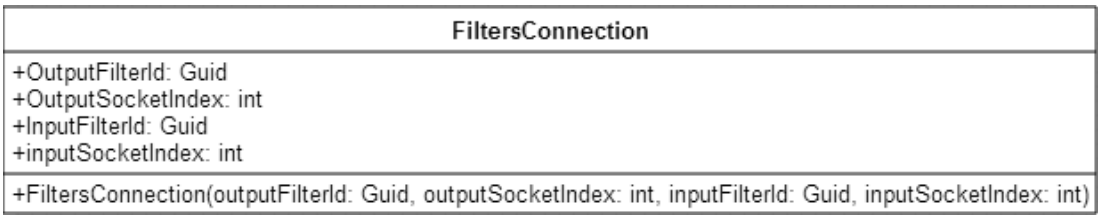
Metoda `CreateNewFilterInstance` slouží k vytváření instancí filtru na základě jeho jména. Zadaný parametr je porovnán se jmény uvedenými v metadatech všech dostupných filtrů a pokud je odpovídající filtr nalezen, metodou `CreateExport` třídy `ExportFactory` je vytvořena jeho instance

```
public IFilter CreateNewFilterInstance(string filterName)
{
        foreach (var filter in Filters)
        {
                if (filter.Metadata.FilterName == filterName)
                {
                        return filter.CreateExport().Value;
                }
        }
        return null;
}
```

[26] Namespace HJD.ImageProcessor.Filters

7.3 Proces zpracování

Součástí jádra jsou také nástroje pro modelování procesu zpracování. Datová struktura uchovávající reprezentaci vytvářeného grafu filtrů je obsažena ve třídě `FiltersChain`[27]. Graf filtrů vychází z návrhu zpracování dat popsaného v kapitole 6.2. Pro naše potřeby bylo třeba tento návrhový vzor mírně upravit, jelikož filtry mohou mít více vstupů i výstupů a nejsou tedy propojeny pouze jednou rourou. Vazba filtrů je reprezentována třídou `FiltersConnection`[28]. Ta obsahuje identifikátory instancí filtrů a indexy výstupu a vstupu, které mají být propojeny. Směr toku dat je jasně určen rozlišením filtru produkujícího a konzumujícího data.

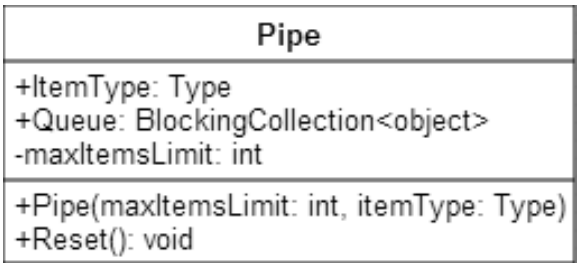


Obrázek 7: Třída FiltersConnection

7.3.1 Roury

V prostředí Microsoft .NET Frameworku je možné pro implementaci roury využít třídy `BlockingCollection<T>` [41]. Jedná se o blokující frontu, která implementuje všechny potřebné funkce. V případě zaplnění své kapacity pozastaví proces producenta, který prvky generuje, nebo naopak, pokud je prázdná, blokuje proces konzumenta. Zároveň řeší konkurenční přístup k obsaženým prvkům. Je možné, aby prvky z kolekce odebíralo více konzumentů nebo vkládalo více producentů. Blokující kolekce zajistí, že dva konzumenti nemohou vyzvednout stejný prvek. Pokud má být proces zpracování ukončen, volá producent nad blokující kolekcí metodu `CompleteAdding`. Ta předává postupně všem dalším kolekcím informaci, že již žádné další prvky vkládány nebudou a je možné procesy producentů a konzumentů ukončit. Proces zpracování lze přerušit i v jeho průběhu. Při vytváření blokujících kolekcí je všem předán společný objekt `CancellationTokenSource`. Zavoláním jeho metody `Cancel` je pak přerušeno čekání blokujících kolekcí a všechny procesy jsou následně ukončeny. Implementace roury v aplikaci se nachází ve třídě `Pipe`[29].

[27] Namespace HJD.ImageProcessor.Core
[28] Namespace HJD.ImageProcessor.Core
[29] Namespace HJD.ImageProcessor.Filters



Obrázek 8: Třída Pipe

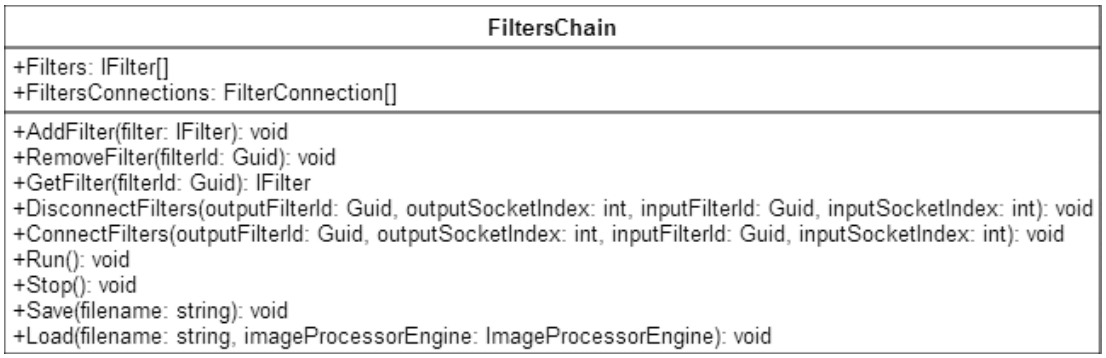
Třída obsahuje blokující kolekci prvků typu `object`. Tento datový typ byl zvolen, aby bylo možné rouru použít univerzálně na přenos libovolných objektů. Při konkrétním spojení však musí rourou procházet pouze taková data, která je schopen konzument zpracovat. Pro tyto účely je roura označena vlastností `ItemType`. Ten informuje o tom, jaký datový typ bude rourou procházet a je nastaven na výstupní datový typ připojeného producenta. Následně může být provedena kontrola na straně konzumenta, zda odpovídá požadovanému datovému typu, případně je možné objekty získané z kolekce přetypovat.

Přenášený datový typ a maximální počet uchovávaných prvků v kolekci je zadán při vytváření instance třídy `Pipe`. V případě, že je dosaženo maximální kapacity, začne kolekce blokovat producenta. Tato hodnota je určena pro všechny roury stejně, konstantou `PIPE_LENGTH` ve třídě `FiltersChain`. Metoda `Reset` slouží k nové inicializaci roury po té, co proces zpracování doběhl, nebo byl přerušen. V takovém případě není možné kolekci znovu použít a metodou `Reset` je vytvořena její nová instance.

7.3.2 Graf filtrů

Graf filtrů, který určuje postup zpracování dat je uchováván ve třídě `FiltersChain` (viz Obrázek 9). Tato třída obsahuje kromě popisu grafu, také metody pro jeho vytváření a vykonání procesu zpracování. Jednotlivé instance filtrů a jejich vazby jsou uloženy v následujících kolekcích. V případě spojů je použit slovník, který udržuje ke každému spoji příslušnou rouru.

```
private List<IFilter> filters;
private Dictionary<FiltersConnection, Pipe> pipes;
```



Obrázek 9: Třída FiltersChain

Pro přidávání nových filtrů do grafu slouží metoda `AddFilter`. Jejím parametrem je již vytvořená instance filtru, kterou je možné získat například metodou `CreateNewFilterInstance` třídy `ImageProcessorEngine`. Metoda kontroluje podle `Id` filtru, zda již tato instance není v grafu vložena. Takto přidané filtry lze následně spojovat. Metoda `ConnectFilters` vytvoří spojení filtrů a příslušnou rouru. Vstupní parametry určují producenta a konzumenta a jejich příslušné výstupy a vstupy, které mají být propojeny. Při sestavování grafu filtrů je každá akce ověřována a v případě nevalidního úkonu je vyvolána výjimka. Jedná se především o pokus spojit neexistující filtry, vzájemné propojení dvou vstupů či výstupů, připojení více vazeb na jeden vstup atd. Metoda `DisconnectFilters` odstraňuje vytvořené spojení, pokud existuje. Při odstranění filtru z grafu metodou `RemoveFilter` jsou odebrány i všechny jeho vazby na ostatní filtry.

Celý graf lze uložit do zvoleného souboru ve formátu XML metodou `Save`. Formát uložených dat je následující:

```
<FiltersChain>
  <Filters>
    <Filter Name="SmoothFilter" Id="293250ee-aee7-4d7a-bf82-0fb7ca16d3b3">
      <Parameters>
        <Parameter Name="SmoothType" Assembly="Emgu.CV"
         Type="Emgu.CV.CvEnum.SMOOTH_TYPE">CV_GAUSSIAN</Parameter>
      </Parameters>
    </Filter>
        ...
  </Filters>
  <Connections>
    <Connection OutputFilterId="293250ee-aee7-4d7a-bf82-
0fb7ca16d3b3" OutputSocketIndex="0" InputFilterId="5ab96b41-d2ed-406b-ad01-
8e97ef8cd912" InputSocketIndex="0" />
        ...
  </Connections>
</FiltersChain>
```

Jeden graf filtrů je reprezentován elementem `FiltersChain`. Ten obsahuje v elementu `Filters` elementy jednotlivých filtrů. Filtr je identifikován dvojicí atributů. `Name` určuje jméno filtru uvedené v metadatech modulu a `Id` pak jeho konkrétní instanci v rámci grafu. Všechny parametry filtru jsou uloženy v elementu `Parameters`. Element `Parameter` určuje atributem `Name`, o který parametr se jedná a musí odpovídat názvu parametru uvedenému v třídě filtru. Jelikož hodnoty parametru mohou být libovolných datových typů, které nejsou součástí .NET frameworku nebo aplikace samotné, je v atributu `Assembly` uveden název dynamické knihovny, ve které se nachází příslušný datový typ `Type`. Hodnota elementu `Parameter` je pak již hodnotou parametru v textové podobě.

Jednotlivá spojení jsou pak umístěna v elementu `Connections`. Každé spojení je určeno atributy, které odpovídají identifikátoru spojovaných filtrů a indexu jejich výstupů.

Analogickým způsobem je možné metodou `Load` ze zadaného souboru graf filtrů načíst. Samozřejmým předpokladem je, že všechny typy použitých filtrů jsou k dispozici a byly při spuštění aplikace načteny. Další nutnou podmínkou je, aby knihovny s použitými nestandardními datovými typy byly spolu s moduly v příslušném adresáři. Při načítání jsou pak vytvářeny instance filtrů a jejich parametry se nastavují následujícím kódem:

```
Type type = Type.GetType(parameter.Attribute("Type").Value);
if (type == null)
{
        type = Assembly.LoadFrom(Path.GetDirectoryName(
                Assembly.GetExecutingAssembly().Location) +
                Properties.Settings.Default.FiltersDirectory + @"\" +
                parameter.Attribute("Assembly").Value + ".dll")
                .GetType(parameter.Attribute("Type").Value);
}
if (type.IsEnum)
{
        value = Enum.Parse(type, parameter.Value);
}
else
{
        value = Convert.ChangeType(parameter.Value, type);
}
filterInstance.SetParameter(parameter.Attribute("Name").Value, value);
```

Pokud určený datový typ parametru není aplikaci znám, pokusí se načíst uvedenou knihovnu z adresáře modulů. Následně je textová hodnota parametru převedena na odpovídající datový typ.

K ukládání a načítání dat je využito LINQ to XML [42]. Díky tomu je možné při práci s XML dokumentem použít LINQ dotazy, které výrazně zjednoduší a zpřehlední zápis zdrojového kódu. Následujícím kódem je vytvořen element `Connections` a naplněn elementy `Connection`, které odpovídají spojům získaným z kolekce `FiltersConnections` včetně jejich atributů.

```
new XElement("Connections",
from connection in this.FiltersConnections
        select new XElement("Connection",
        new XAttribute("OutputFilterId", connection.OutputFilterId),
                ...
        )
)
```

K práci s XML soubory by také bylo možné použít i třídy `XmlReader` a `XmlWriter` nebo `XmlDocument`.

Zbývající metody `Run` a `Stop` slouží ke spuštění resp. přerušení procesu zpracování dle sestaveného grafu filtrů. Metoda `Run` prochází všechny instance filtrů přidané do grafu a pokouší se u každého z nich spustit jejich metodu `Process` jako samostatnou úlohu. Využito je přitom anonymní metody zapsané pomocí lambda výrazu [43]. Ta nemá žádné vstupní parametry a volá pouze metodu `Process` daného filtru. Jako parametry vstupují do této metody pole vstupních a výstupních rour, které jsou určeny vytvořeným grafem filtrů. Pro vytváření úloh je použita třída `TaskFactory`. Je nastavena tak, aby vytvářela dlouho běžící úlohy. Tím odpadá opakované nastavování parametrů u každé nové úlohy.

```
CancellationTokenSource token = new CancellationTokenSource();
var taskFactory = new TaskFactory(TaskCreationOptions.LongRunning,
                        TaskContinuationOptions.None);
var tasks = new List<Task>();

foreach (IFilter filter in filters)
{
        tasks.Add(taskFactory.StartNew(
        () => filter.Process(GetInputPipes(filter),GetOutputPipes(filter),
        token)));
}
Task.WaitAll(tasks.ToArray());
```

Poslední část kódu - `Task.WaitAll()` čeká na dokončení všech vytvořených úloh. Pokud dojde v jednotlivých úlohách k výjimce, jsou tyto sloučeny do jediné výjimky typu `AggregateException`, která obsahuje všechny dílčí výjimky. Její zpracování je blíže popsáno v kapitole 7.5.6.

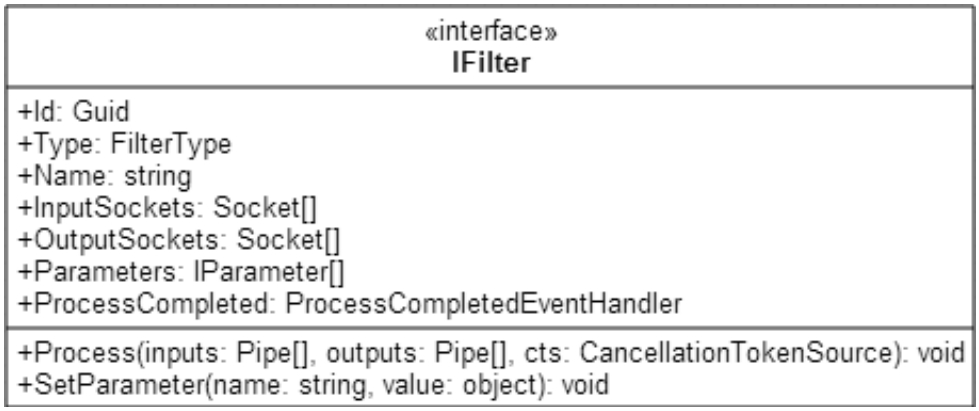
Použitý `CancellationTokenSource`, který je předáván každému filtru společně se vstupními a výstupními rourami, slouží k přerušení jejich činnosti. Metoda `Stop` je implementována pouze voláním následujícího kódu:

```
token.Cancel();
```

Využití tohoto tokenu na úrovni jednotlivých filtrů je detailně popsáno v následující kapitole 7.4.1.

7.4 Filtr

Filtr je základní operací, kterou lze v rámci celého zpracování provádět. Třídy jednotlivých filtrů musí implementovat rozhraní `IFilter` (viz Obrázek 10) a jsou k aplikaci přidávány jako dynamické knihovny. Každá knihovna může obsahovat jeden či více filtrů.



Obrázek 10: Rozhraní IFilter

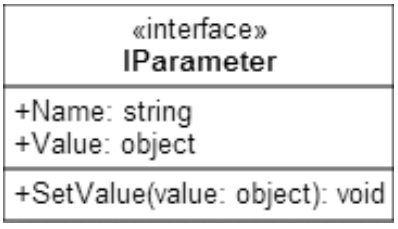
Rozhraní `IFilter` definuje metadata filtru, metodu vykonávající zpracování vstupů a metodu pro úpravu parametrů zpracování.

- `Id` jednoznačně identifikuje instanci filtru.
- Výčtová hodnota `Type` určuje pozici filtru v grafu a způsob, jakým s ním bude pracováno. Tato hodnota by teoreticky šla odvodit od počtu vstupů a výstupů, ale nemusí však ve všech případech zcela odpovídat.

    - `Head` - Počáteční filtr generující vstupy, které jsou následně dále zpracovány. Jeho metoda `Process` je volána pouze jednou ale může produkovat větší množství výstupů.
    - `Link` – Jedná se o filtr, který zpracuje přidělené vstupy a svůj výstup posílá dalšímu filtru.
    - `End` – Konečný filtr provádějící operace, které nemají žádné další výstupy.
- Vlastnost `Name` uchovává název filtru.
- Pole `InputSockets` a `OutputSockets` deklarují počet vstupů a výstupů společně s jejich datovým typem.
- Pole `Parameters` obsahuje seznam nastavitelných parametrů a jejich přípustné hodnoty. Hodnotu parametru lze změnit voláním metody `SetParameter`.
- Metoda `Process` je volána pro každou dávku vstupů. Implementace této metody obsahuje algoritmus zpracování, který aplikuje na obdržené vstupy a výsledek odesílá do výstupní fronty.
- Událost `ProcessCompleted` je vyvolána po každém zpracování vstupů, je možné ji využít třeba k aktualizaci grafického rozhraní, jelikož předává i pole zpracovaných výstupů.

Použitá třída `Socket` obsahuje pouze určení datového typu. Z polí `InputSockets` a `OutputSockets` je tedy možné zjistit nejen počet vstupů a výstupů, ale i datový typ objektů, které jsou příslušným vstupem očekávány, resp. vychází daným výstupem. Tuto informaci lze využít při validaci spojení filtrů nebo případně zobrazit uživateli.

Parametry filtru jsou uloženy v poli `Parameters` a mají definované rozhraní `IParameter`.



Obrázek 11: Rozhraní IParameter

Jméno parametru `Name` musí být v rámci jednoho filtru unikátní, jelikož přímo identifikuje konkrétní parametr. Vlastnost `Value` obsahuje hodnotu parametru. Její změna se provádí metodou `SetValue`. Ta nejdříve validuje vstupní hodnotu a následně nahradí hodnotu původní. Momentálně jsou implementovány tři typy parametrů.

- **`DirectoryParameter`**

  Tento parametr uchovává cestu v adresářové struktuře. Využit je například při určování adresáře pro výstup exportního filtru. Celá cesta je uložena jako textový řetězec.

- **`EnumParameter`**

  Reprezentuje parametr, který může nabývat pouze hodnot z předem definované množiny. Tyto hodnoty jsou uloženy v poli `PossibleValues`, které je naplněno při vytváření instance parametru.

  ```
  EnumParameter(string name, object value, object[] possibleValues)
  ```

  Metoda `SetValue` kontroluje, zda je zadaná hodnota obsažena v poli přípustných hodnot

- **`RangeParameter`**

  Posledním typem parametru je číselná hodnota ze zadaného rozsahu. Hodnota parametru může být desetinné číslo a je reprezentována datovým typem `double`. Přípustné hodnoty jsou omezeny minimem a maximem při vytváření instance parametru.

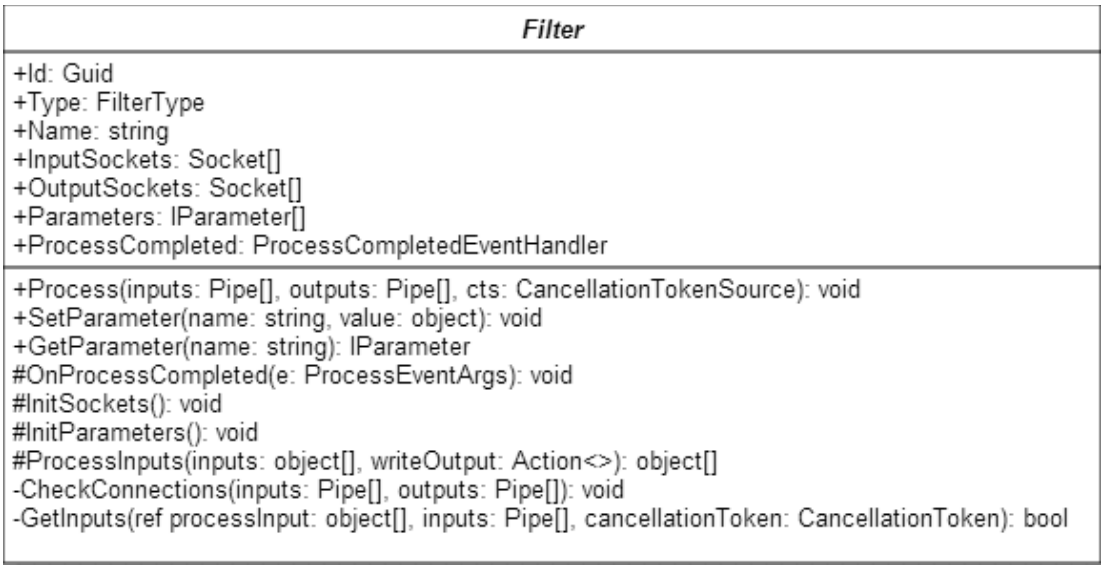
  ```
  RangeParameter(string name, object value,
          double minimum, double maximum)
  ```

  Metoda `SetValue` kontroluje, zda se jedná o datový typ `double` a hodnota parametru je v rozsahu definovaném minimální a maximální možnou hodnotou.

7.4.1 Abstraktní třída Filter

Během vytváření jednotlivých filtrů bylo identifikováno několik identických částí kódu, které byly opakovaně implementovány pro každý filtr. Jedná se především načítání vstupních dat z přidělených rour. S ohledem na zjednodušení a zpřehlednění

tvorby dalších filtrů byla vytvořena abstraktní třída `Filter`, která implementuje rozhraní `IFilter`.



Obrázek 12: Abstraktní třída Filter

- Metody `InitSockets` a `InitParameters` slouží k nastavení typu a počtu vstupů resp. výstupů a parametrů, které má filtr k dispozici. Obě metody jsou volány v konstruktoru třídy `Filter`.
- Metoda `SetParameter` nastavuje zadanému parametru předanou hodnotu. Pokud není dle uvedeného jména žádný parametr nalezen, je vyvolána výjimka.
- Metoda `GetParameter` slouží k získávání hodnot parametrů, využita je především při zpracování v metodě `ProcessInputs`. Opět v případě, že není zadaný parametr nalezen, je vyvolána výjimka.
- Privátní metoda `CheckConnections` provádí kontrolu, zda počet a typ přidělených vstupů a výstupů odpovídá požadovaným. Kontrola je prováděna oproti definovaným polím `InputSockets` a `OutputSockets` před samotným zpracováním dat v metodě `Process`.
- Abstraktní metoda `ProcessInputs` musí být implementována ve všech dceřiných třídách. Její implementace pak obsahuje algoritmus zpracování jednoho kroku (s výjimkou filtrů generujících data). Parametry jsou pole vstupních dat a delegát, který slouží k zápisu dat výstupních. Ty jsou také předávány i jako návratová hodnota metody. Použití dvou různých metod předání výstupních hodnot může být matoucí, liší se však svým účelem. Ačkoliv je každý způsob použit k jiným účelům, měla by být předávaná data shodná.

Návratová hodnota metody je předávána jako parametr události `ProcessCompleted`. Díky tomu je možné při sledování příslušné události získat i data, která byla v daném kroku vyprodukována. Naproti tomu delegát předávaný jako parametr slouží k zápisu výstupních dat do roury a jejich odeslání následujícímu filtru. Tento způsob je použit kvůli filtrům generující vstup, jelikož jejich metoda `ProcessInputs` je volána pouze jednou a nebylo by tak možné předávat generovaná data dalším filtrům.

- Metoda `GetInputs` načítá data ze vstupních rour. Jak bylo zmíněno v kapitole 7.3, je oproti standardnímu návrhovému vzoru v našem případě použito více vstupních a výstupních rour. Kvůli tomu je nutné zajistit, aby byla vstupní data předána ke zpracování až v momentě, kdy máme k dispozici všechny požadované výstupy předchozích filtrů.

```
private bool GetInputs(ref object[] processInput, Pipe[] inputs,
          CancellationToken cancellationToken)
{
          int i = 0;
          foreach (var input in inputs)
          {
                    if (!input.Queue.TryTake(
                        out processInput[i], -1, cancellationToken))
                    { return false; }
                    i++;
          }
          return true;
}
```

Uvedený kód postupně prochází vstupní roury a pokouší se pomocí metody `TryTake`, volané na blokující kolekci, získat další prvek, který obsahuje. Číselný parametr určuje maximální dobu, po kterou má metoda na případný prvek čekat. Uvedená záporná hodnota -1 znamená, že nemá být aplikován žádný časový limit na dobu čekání. Předávaný objekt `CancellationToken` zajistí přerušení čekání v případě, že bude v aplikaci zavolána jeho metoda `Cancel`, například při ukončení procesu zpracování na pokyn uživatele nebo vyvoláním výjimky. Získaná data jsou ukládána do předaného pole `processInput`. Pokud již není k dispozici žádný prvek, tj. všechna vstupní data byla již zpracována (tato informace je v blokující kolekci signalizována voláním metody `CompleteAdding`), vrací metoda `TryTake` hodnotu `false` stejně jako celá metoda `GetInputs`.

- Implementace metody `Process` obsahuje kontrolu vstupních a výstupních rour pomocí metody `CheckConnections`. Následně je spuštěna nekonečná smyčka:

```
if (Type != FilterType.Head)
{
          if (!GetInputs(ref processInput, inputs, cts.Token))
                    break;
}
else
{
          processInput = new object[] { cts.Token };
}

processOutput = ProcessInputs(processInput,
          (index, item) => { if (outputs[index] != null)
          outputs[index].Queue.Add(item, cts.Token); });
OnProcessCompleted(new ProcessEventArgs(processOutput));

if (Type == FilterType.Head) { break; }
```

  Pokud se jedná o filtr generující data, tedy typu `Head`, je předán jako jediný vstup metodě `ProcessInputs` objekt `CancellationToken`. Ten umožnuje implementované dlouho běžící operaci zjistit požadavek na přerušení a vhodnými nástroji svou činnost ukončit. Všechny ostatní typy filtrů využívají metodu `GetInputs` k získaní vstupních dat z příslušných rour. Metodě `ProcessInputs` je kromě získaných vstupních dat předán i delegát v podobě anonymní funkce, která přidává prvek do výstupní roury s daným indexem. Nekonečná smyčka je přerušena, pokud nejsou již k dispozici žádná vstupní data nebo v případě ukončení činnosti filtru generujícího data. Pokud během této smyčky dojde k nějaké chybě, je zavolána metoda `Cancel` objektu `CancellationToken`, která zajistí ukončení celého zpracování. Na konci zpracování je pomocí metody `CompleteAdding` předána všem výstupním blokujícím kolekcím informace, že již žádné další prvky nebudou přidávány.

7.4.2 Implementované filtry

Součástí aplikace je sada několika vzorových filtrů. Pro implementaci algoritmů strojového zpracování obrazu byla použita knihovna Emgu CV. Tím byla zároveň ověřena možnost využití externích nástrojů a knihoven třetích stran pro implementaci algoritmů zpracování. Všechny tyto filtry dědí od abstraktní třídy `Filter` a nachází se

v projektu `HJD.ImageProcessor.BaseFilters`. Je možné je rozdělit do několika skupin. První z nich jsou filtry, které generují data.

- **ImageLoader**

  `ImageLoader` slouží k načítání obrazových dat ze souborů uložených v adresáři. Cesta k adresáři je určena parametrem typu `DirectoryParameter`. Metoda `ProcessInputs` prochází všechny soubory v daném umístění a vytváří z nich objekty typu `Image<Bgr, Byte>`, což je datový typ používaný v knihovně Emgu CV. Druhým výstupem je název souboru, který lze využít v dalším zpracování, například v exportním filtru, kdy může být zpracovaný obrázek uložen pod stejným jménem, jako měl zdroj. Zde si můžeme všimnout využití objektu `CancellationToken` získaného ze vstupu.

  ```
  CancellationToken ct = (CancellationToken)inputs[0];
  ```

  Před každým načtením souboru z adresáře je kontrolováno, zda nebyl vyvolán požadavek na ukončení zpracování.

  ```
  if (ct.IsCancellationRequested) break;
  ```

  Pokud ano je tento proces ukončen. Návratová hodnota je prázdné pole, jelikož u tohoto filtru je událost `ProcessCompleted` vyvolána až po té, co jsou všechny soubory zpracovány.

Další skupinou filtrů jsou ty, které se podílejí na zpracování obrazu. Všechny používají datové typy knihovny Emgu CV (ať už jako vstupní nebo výstupní data či parametry) a využívají knihovnou nabízené algoritmy. Jejich možné pořadí a vzájemné propojení je dáno jak omezením datovými typy uvedenými v polích `InputSockets` a `OutputSockets` tak logikou a konkrétní aplikací požadovaného algoritmu zpracování obrazu.

- **SmoothFilter**

  Filtr provádí vyhlazení vstupního obrazu dle metody zvolené výčtovým parametrem. Výstupem je vyhlazený obraz v odstínech šedi ve formátu `Image<Gray, Byte>`.

- **GrayFilter**

  Tento filtr převádí vstupní obraz ve formátu `Image<Bgr, Byte>` do obrazu v odstínech šedi `Image<Gray, Byte>`.

- **ThresholdFilter**

  Metoda použitého prahování v tomto filtru je určena výčtovým parametrem. Navíc je možné upravovat hodnoty prahů dalšími číselnými parametry z definovaných rozsahů. Výstupem je binární obraz ve formátu `Image<Gray, Byte>`.

- **CountourFinder**

  Tento filtr hledá hrany objektů v zadaném binárním obrazu. Metoda detekce a druh detekovaných hran jsou určeny výčtovými parametry. Výstupem jsou úsečky kopírující hrany. Definované jsou svými krajními body a celá struktura je uložena ve výstupním objektu `Contour<Point>`.

- **ContourDrawer**

  Filtr, který vykresluje hrany do obrazu. Vstupem je objekt reprezentující strukturu hran `Contour<Point>` a obraz typu `Image<Bgr, Byte>`, do kterého mají být hrany vykresleny. Parametry lze určit tloušťka, typ čar a hrany, které mají být vykresleny.

Poslední skupina zahrnuje filtry, které byly vytvořeny k účelům, ne zcela souvisejícím s transformací obrazových dat. Jedná se spíše prvky umisťované do procesu zpracování, které umožňují specifické úkony v rámci modelovaného procesu například jeho větvení a monitorování.

- **Multiplier**

  Výstupem multiplikátoru jsou dvě kopie vstupního obrazu. Využívá se především, pokud chceme obraz zpracovat ale zároveň jeho původní podobu použít v další části zpracování. Například při hledání hran a jejich následném vykreslení do původního obrazu, nebo při náhledu na dílčí výsledky zpracování.

- **BitmapViewer**

  Vstupem tohoto prvku je libovolný obraz, který dědí od `IImage`. Při zpracování je tento převeden do bitmapy `System.Drawing.Bitmap` a výstup je pak zobrazován v grafickém rozhraní. Pro tento druh filtru je použit odlišný ovládací prvek, který umožňuje vykreslení převedené bitmapy.

- **Exporter**

  `Exporter` má obdobnou funkci jako `BitmapViewer`. V tomto případě ale dochází k ukládání vstupního obrazu do parametrem zadaného adresáře. Název souboru může být určen druhým vstupem, který lze získat například z `ImageLoader`. Pokud tento vstup k dispozici není, je pro název souboru generován náhodný řetězec.

7.4.3 Tvorba vlastních filtrů

Tato kapitola obsahuje základní přehled postupu vytváření nových filtrů do stávající aplikace. Detailnější popis celého systému a jeho jednotlivých částí je obsažen v předchozích kapitolách. Uveden je pouze postup pro vytváření standardních filtrů. Jako vzorové příklady mohou sloužit již implementované filtry v projektu `HJD.ImageProcessor.BaseFilters`.

Prvním krokem je založení nového projektu, který bude kompilován jako dynamická knihovna. Tento projekt musí odkazovat na knihovnu `HJD.ImageProcessor.Filters`. Zde jsou umístěna všechna potřebná rozhraní a třídy. Následně vytvoříme třídu, která bude reprezentovat námi implementovaný filtr. Abychom mohli používat třídy z výše uvedeného jmenného prostoru, musíme přidat direktivu using:

```
using HJD.ImageProcessor.Filters;
```

Třída samotná musí být označena atributem `FilterAttribute`. Ten určuje jméno, počet vstupů a výstupů v tomto pořadí. Jméno musí být unikátní v rámci celé aplikace, slouží jako identifikátor filtru. V našem případě tedy vytváříme filtr s názvem `NovyFiltr` a jedním vstupem a výstupem:

```
[FilterAttribute("NovyFiltr", 1, 1)]
public class NovyFiltr : Filter
```

Třída filtru musí implementovat rozhraní `IFilter`. V takovém případě je však nutné provést jeho kompletní implementaci včetně získávání vstupních dat. Z tohoto důvodu byla vytvořena abstraktní třída `Filter`, která rozhraní `IFilter` implementuje a vyžaduje pouze vlastní algoritmus zpracování dat. Pro standardní filtry je doporučeno použít třídu `Filter`. Rozhraní `IFilter` je vhodné implementovat v případě, že chceme vytvořit filtr s odlišným chováním. Dále je třeba nastavit metadata filtru pomocí následujících vlastností:

```
public override string Name { get { return "NovyFiltr"; } }
```

Pod tímto jménem vystupují v aplikaci jednotlivé instance filtru (např. při zobrazení vlastností nebo logování chyb). Na rozdíl od jména uvedeného v atributu filtru nemusí být unikátní, neslouží k přímé identifikaci filtru. Je však vhodné udržovat obě tato jména shodná.

```
public override FilterType Type {get { return FilterType.Link; } }
```

Typ filtru určuje, kde v procesu zpracování se nachází. Důležité je označit filtr generující data typem `Head`, jelikož se s ním pak pracuje odlišně. Ostatní typy `Link` a `End` jsou víceméně shodné. Informují pouze, zda filtr produkuje další data ke zpracování či nikoliv.

Posledními metadaty jsou informace o vstupech, výstupech a jejich datových typech a možné parametry zpracování, které lze nastavovat.

```
protected override void InitSockets()
{
        inputSockets = new Socket[] { new Socket(typeof(string)) };
        outputSockets = new Socket[] { new Socket(typeof(string)) };
}
```

V metodě `InitSockets` nastavíme pole vstupních a výstupních bodů. Ty určují počet vyžadovaných vstupů a jejich datový typ resp. výstupy filtru. Jejich počet musí odpovídat počtu uvedenému v atributu filtru. Pokud filtr nemá žádné vstupy nebo výstupy je vloženo prázdné pole.

```
protected override void InitParameters()
{
        parameters = new IParameter[]
        {
                new RangeParameter("Par", 1d, -255, 255)
        };
}
```

Parametry filtru jsou nastaveny obdobným způsobem v metodě `InitParameters`. Název parametru musí být v rámci třídy filtru unikátní, opět slouží k jeho jednoznačné identifikaci. Jednotlivé druhy parametrů a jejich použití jsou rozepsány v předchozí kapitole 7.4.

Poslední a nejdůležitější části je samotná implementace funkce filtru v metodě `ProcessInputs`. Parametry této metody jsou vstupní data a funkce pro zápis dat výstupních:

```
protected override object[] ProcessInputs(object[] inputs,
        Action<int, object> writeOutput)
```

Pořadí vstupů odpovídá pořadí, v jakém byly deklarovány v metodě `InitSockets`. Pouze v případě filtru typu `Head` je do vstupního pole předán jediný objekt `CancellationToken`, který se využívá ke sledování požadavku na přerušení činnosti, jelikož metoda `ProcessInputs` je v tomto případě spuštěna pouze jednou a předpokládá se, že bude generovat data. Je nutné algoritmus zpracování ve filtru generujícím data implementovat tak, aby na tento požadavek byl ihned ukončen. Příklad takové implementace je možné nalézt ve filtru `ImageLoader`. U ostatních filtrů je metoda `ProcessInputs` volána opakovaně pro každý prvek, který filtrem prochází. Výsledek zpracování je třeba zapsat do výstupní roury. K tomu slouží metoda `writeOutput`, předávaná jako parametr. Jejím vstupem je index výstupní roury (odpovídá pořadí určenému pořadím prvků v `outputSockets`) a objekt, který do ní má být vložen.

```
        writeOutput(0, vystupniObjekt);
```

Současně by všechny výstupy zpracování měly být předány jako návratová hodnota metody `ProcessInputs`. Toto neplatí v případě filtru typu `Head`, zde se vrací pouze prázdné pole. Více je tato problematika popsána v kapitole 7.4.1.

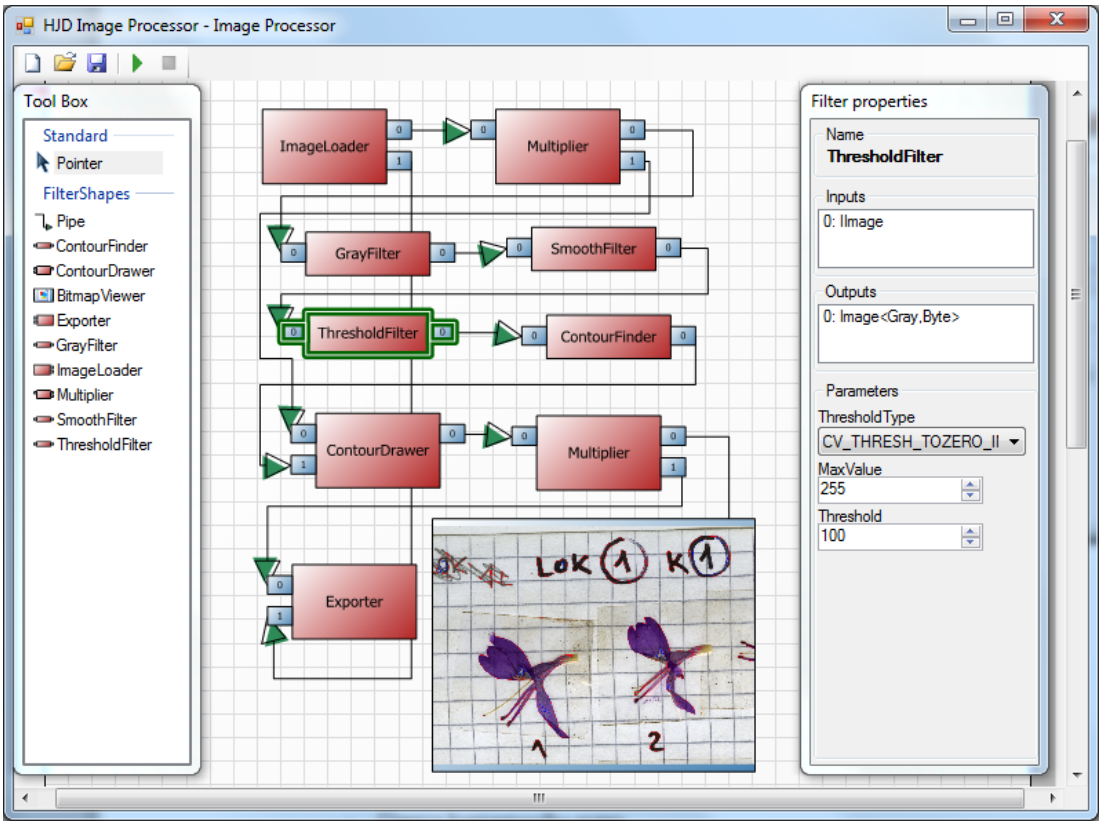
```
return new object[] { vystupniObjekt };
```

Výslednou knihovnu včetně všech použitých knihoven třetích stran umístíme do adresáře pro moduly a při spuštění aplikace budou v ní obsažené filtry k dispozici.

7.5 GUI

Grafické rozhraní využívá metod jádra a třídy pro správu procesu zpracování. Umožňuje tak sestavovat graf filtrů rychlým a uživatelsky přívětivým způsobem. Díky oddělení těchto dvou části je možné GUI nahradit jakýmkoliv jiným bez nutnosti zásahu do výkonné části aplikace, případně lze aplikaci využít i bez GUI. Výhodou grafického rozhraní je možnost zobrazit náhled na výstup zpracování a mít tak okamžitou odezvu na provedené změny v procesu či parametrech filtrů.

Grafické rozhraní je standardní formulářová aplikace umístěna v projektu `HJD.ImageProcessor.GUI`. Při její implementaci byl použit framework pro práci s diagramy NShape. Ten se skládá z několika vrstev, které jsou tvořeny odpovídajícími komponentami v podobě ovládacích prvků a jsou umisťovány do okna formuláře. Základní komponentou je `Project`. Ta uchovává datovou reprezentaci diagramu a umožňuje manipulaci s jeho jednotlivými entitami. K projektu se připojuje repozitář, který zajišťuje ukládání a načítání dat z příslušného úložiště. Dle volby repozitáře jím může být například XML soubor, SQL databáze apod. O grafickou prezentaci dat projektu se stará komponenta `Display`.



Obrázek 13: Grafické rozhraní aplikace

Obrazovka hlavního formuláře se skládá z několika částí. Horní lišta obsahuje základní ovládací prvky pro vytváření nového projektu, načtení existujícího a uložení rozpracovaného. Další dvě tlačítka slouží ke spuštění vytvořeného procesu zpracování resp. jeho přerušení.

V levém plovoucím okně *Tool Box* jsou zobrazeny všechny dostupné filtry včetně roury, kterou je možné spojovat jejich výstupy a vstupy. Vytváření grafu filtrů je prováděno přetahováním odpovídajících prvků z nabídky do prostoru čtverečkovaného papíru. Dle takto vytvořeného procesu je prováděno zpracování.

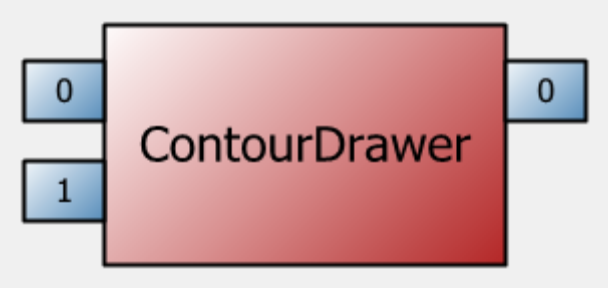
Okno *Filter Properties* v pravé části zobrazuje vlastnosti vybraného filtru. Jeho název, jednotlivé vstupy a výstupy včetně jejich datových typů. Zde je také možné měnit hodnoty parametrů daného filtru. Ihned po změně parametru je provedeno opakované spuštění procesu zpracování. Výsledek změny je okamžitě vidět.

Inicializace aplikace je prováděna v metodě `MainForm_Load` hlavního formuláře `MainForm`. Zde je vytvořena instance jádra `ImageProcessorEngine` a grafu filtrů `FiltersChain`. Následně je přidáno sledování událostí komponent NShape. Jedná se o přidání a odebrání prvku z diagramu a jejich propojení resp. odstranění spoje. Tyto události jsou pak přeposílány odpovídajícím metodám grafu filtru. Použitá třída

`BackgroundWorker` slouží ke spouštění procesu zpracování v odlišném vlákně, než běží grafické rozhraní. Díky tomu je možné během zpracování dat aplikaci ovládat a aktualizovat obsah formuláře. Metoda `InitTools` pomocné třídy `NShapeHelper` vytváří entity reprezentující jednotlivé filtry. Pro tento účel byla implementována tovární třída `TemplateFactory`, která v případě standardních filtrů sestavuje jejich grafickou podobu z dílčích tvarů na základě počtu vstupů a výstupů. K tomu využívá informace z atributů filtrů, které jsou k dispozici ještě před vytvořením jejich instancí. Framework NShape pak zajistí jejich zobrazení v okně *Tool Box*.

7.5.1 Grafická reprezentace filtru

Framework NShape pracuje s objekty, které reprezentují jednotlivé entity použité v diagramu. Každý takový objekt má svou grafickou podobu. Pro potřeby modelování procesu bylo však třeba vytvořit vlastní entity, jelikož základní objekty nabízené knihovnou nenabízely potřebnou funkcionalitu. Všechny tyto objekty jsou umístěny v projektu `HJD.ImageProcessor.GUI.FilterShapes`. Hlavním problémem při vytváření grafické reprezentace filtru je variabilní počet jeho vstupů a výstupů. Kvůli tomu je nutné tento tvar generovat dynamicky na základě vlastností filtru.



Obrázek 14: Grafická reprezentace filtru

Třída `FilterShape` reprezentuje filtry zobrazené na Obrázek 14. Tyto filtry jsou tvořeny základním obdélníkem s názvem filtru a malými čtverci, které odpovídají jednotlivým vstupům a výstupům. Zobrazen je příslušný index daného přípojného bodu. Tyto čtverce také zajišťují vzájemné propojení filtrů pomocí roury. Třída `FilterShape` byla rozšířena o potřebné údaje jako je identifikátor instance filtru, který reprezentuje, počet vstupů a výstupů a rozměry jednotlivých částí. Aby bylo možné s těmito hodnotami pracovat při ukládání a načítání celého projektu, je nutné upravit následující metody:

```
new public static IEnumerable<EntityPropertyDefinition>
                GetPropertyDefinitions(int version)
{
        foreach (EntityPropertyDefinition pi in
                        Box.GetPropertyDefinitions(version))
                yield return pi;
        yield return new EntityFieldDefinition("Id", typeof(string));
        ...
}
```

Metoda `GetPropertyDefinitions` vrací seznam všech vlastností, které daná entita obsahuje. Zde je třeba vrátit všechny vlastnosti nadřazeného objektu, v našem případě `Box`, a přidat ty, o které je třída filtru rozšířena. Určuje se název vlastnosti a její datový typ. Je možné všimnout si parametru `version`. Ten lze použít, pokud existuje více verzí datového souboru, kdy například projekty uložené ve starší verzi některé vlastnosti neobsahuje. Samotné ukládání a získávání hodnot z uloženého projektu je implementováno v těchto metodách:

```
protected override void SaveFieldsCore(IRepositoryWriter writer,
        int version)
{
        base.SaveFieldsCore(writer, version);
        writer.WriteString(FilterId.ToString());
        ...
}
```

Opět je možné rozlišovat verze datových souborů a přizpůsobit tomu ukládané vlastnosti. Nejprve jsou uloženy vlastnosti mateřského objektu a pak vlastnosti nově přidané.

```
protected override void LoadFieldsCore(IRepositoryReader reader,
        int version)
{
        base.LoadFieldsCore(reader, version);
        FilterId = new Guid(reader.ReadString());
        ...
}
```

Metoda načítání funguje analogicky. Důležité je zachovat stejné pořadí jednotlivých vlastností jaké bylo určeno v metodě `GetPropertyDefinitions` a to jak při ukládání tak i při načítání.

Každý tvar má přiděleny kontrolní body, které umožňují jeho přesunování, změnu velikosti a připojování jiných tvarů. Tyto body jsou generovány v metodě `CalcControlPoints` a jsou umístěny na středy vnějších stran čtverců

představujících vstupy a výstupy filtru. Metoda `HasControlPointCapability` určuje, které operace daný kontrolní bod umožňuje. V případě filtru je povoleno pouze jeho přesouvání. Změna velikosti a rotace jsou zakázány. Připojování je povolenou pouze k bodům umístěným na bočních čtvercích. Toto omezení je implementováno v metodě `GetControlPointIds`. Ta zároveň kontroluje, zda již k danému bodu není jiný tvar připojen, v takovém případě již nelze další připojit.

Zvláštním případem filtru je `BitmapViewer`. Ten je v grafickém rozhraní zobrazován odlišným způsobem oproti ostatním filtrům a je reprezentován třídou `PreviewShape`. Vychází z již existujícího objektu `Picture`. Ten slouží k zobrazování souborů bitmap. Pro potřeby náhledu na výstup filtrů však musel být upraven tak, aby dokázal zobrazovat všechny obrázky během zpracování. Tuto funkci implementuje metoda `RedrawMethod`, která předanou bitmapu vykreslí na plochu objektu. Na rozdíl od ostatních filtrů je tento možné otáčet a měnit jeho velikost. Má jediný vstup, což odpovídá filtru `BitmapViewer`, a jeho připojení je možné v jakékoliv části plochy objektu.

Všechny objekty filtrů implementují rozhraní `IFilterShape`. To obsahuje identifikaci filtru, který reprezentují, a počet vstupů a výstupů. Díky tomu je možné zobrazovat vlastnosti filtru bez ohledu na to, která třída se stará o jeho implementaci v grafickém rozhraní.

Poslední použitou entitou je orientovaná čára `PipeShape`, která znázorňuje směr toku dat a vytváří spojení mezi filtry. Dědí od standardní lomené čáry `RectangularLine`. Zvláštní třída je použita pro lepší identifikaci tohoto objektu v grafu.

Nové třídy je nutné při spuštění aplikace do frameworku NShape registrovat, aby s nimi mohl pracovat. Tato registrace je provedena ve třídě `FilterShapes` statickou metodou `Initialize`.

7.5.2 Vkládání a odebírání filtrů

Přetahováním filtrů z okna *Tool Box* a jejich umisťováním na plochu objektu `Diagram` je vyvolána událost repozitáře `ShapesInserted`. Na základě této události je přidána do kolekce filtrů v objektu `filtersChain` nová instance příslušného filtru. Speciálním případem je filtr `BitmapPreview`, jehož grafická reprezentace

sleduje událost filtru `ProcessCompleted` a výslednou bitmapu zobrazuje. To je provedeno následujícím kódem:

```
if (shape.Type == nshapeProject.ShapeTypes["PreviewShape"])
{
        filter.ProcessCompleted += (IFilter f, ProcessEventArgs eventArgs)
                        new object[] { (Bitmap)eventArgs.Outputs[0] });
                };
}
```

Metoda `Redraw` překresluje obsah entity `PreviewShape` zadanou bitmapou, která je výstupem filtru `BitmapPreview`. Jelikož proces zpracování běží v odlišném vlákně, než grafické rozhraní, je nutné použít metodu `Invoke`. Ta provede vykonání určené metody ve stejném vlákně, ve kterém se nachází daný ovládací prvek, v našem případě tedy objekt `Display`. Analogicky jsou instance filtrů na základě události `ShapesDeleted` odstraněny.

7.5.3 Změna parametrů

Po označení entity filtru kurzorem myši je vyvolána událost `ShapeClick`. Na základě identifikátoru filtru jsou z `filtersChain` získána jeho metadata a zobrazena v okně *Filter properties*. Kromě názvu druhu filtru a datových typů jednotlivých vstupů a výstupů jsou zobrazeny i jeho parametry. Pro každý typ parametru je vytvořen ovládací prvek, který dědí od společného `ParameterControl`. Tyto ovládací prvky slouží nejen k zobrazení aktuální hodnoty parametru, ale i k jeho úpravě. Při změně hodnoty je vyvolaná událost `ParameterChanged`, která nastaví příslušnému parametru novou hodnotu a spustí proces zpracování. Implementované parametry:

- **`RangeParameterControl`**

  Reprezentuje parametr typu `RangeParameter`. Použitý ovládací prvek `NumericUpDown` zobrazuje aktuální číselnou hodnotu parametru. Zároveň kontroluje, zda je zadaná hodnota v rozsahu určené vlastnostmi `Minimum` a `Maximum`.

- **`EnumParameterControl`**

  Reprezentuje parametr typu `EnumParameter`. Pro zobrazení a volbu hodnoty je použit ovládací prvek `ComboBox`, který je naplněn přípustnými hodnotami z vlastnosti `PossibleValues`.

- **`DirectoryParameterControl`**

  Cesta k adresáři z parametru typu `DirectoryParameter` je zobrazena v textové podobě. Její změna je provedena dvojklikem na `TextBox` a výběrem za adresáře pomoci standardního dialogového okna `FolderBrowserDialog`.

7.5.4 Spojování filtrů

Validace spojení na úrovni frameworku NShape není ve smyslu logiky propojení filtrů a rour možná. Framework umožňuje spojení jakýchkoliv bodů, které mají definovanou vlastnost přípojného bodu. Navíc je možné připojit jen jednu část čáry, což není u roury přípustné. Proto byla vytvořena třída `ConnectionManager`. Kdykoliv je přidáno nebo odstraněno spojení dvou entit při události repozitáře `ConnectionInserted` resp. `ConnectionDeleted`, uloží se informace o této části spojení do datové struktury `ConnectionManager`. Jakmile jedna roura spojuje dva filtry, je volána metoda grafu filtrů `filtersChain ConnectFilters`. Pokud tato metoda vyhodnotí spojení jako nevalidní, je předchozí operace vrácena a spojení i na straně grafického rozhraní odstraněna. Obdobným způsobem je již při odstranění jedné části spoje volána metoda `DisconnectFilters`, která filtry na úrovni grafu filtrů rozpojí.

7.5.5 Ukládání projektu

Datový soubor formátu XML, který je použit pro ukládání diagramu, je odlišný od souboru, který uchovává graf filtrů. Jedná se pouze o data diagramu a vlastnosti jednotlivých entit jako velikost pozice atd. Každý projekt je tak tvořen dvěma soubory. Při načítání projektu je nutné naplnit `ConnectionManager`, strukturu uchovávající spojení filtrů, odpovídajícími hodnotami z načteného diagramu. K tomu slouží metoda `Load`. Pokud se v diagramu vyskytují náhledy výstupu `PreviewShape`, je nutné znovu přidat sledování události `ProcessCompleted` příslušného filtru `BitmapPreview`. Tento úkon provádí metoda `ReconnectPreviewShapes` pomocné třídy `NShapeHelper`.

7.5.6 Spuštění procesu zpracování

Aplikace se pokouší zahájit proces zpracování po kliknutí na zelenou ikonku přehrávání v horní liště formuláře. Stejně tak při úpravě parametru filtru je stávající proces zastaven, pokud běží, a spuštěn znovu. Proces zpracování je spouštěn v samostatném vlákně s použitím třídy `BackgroundWorker`. Tím je zajištěno, že grafické rozhraní bude responsivní i v průběhu zpracování. Při vyvolání události `DoWork` třídy `BackgroundWorker` je volána metoda `Run` grafu filtrů `filtersChain` a zachycena případná výjimka `AggregateException`. V tomto případě se jedná o výjimku složenou ze všech výjimek, které byly v průběhu zpracování vyvolány. Následujícím kódem jsou tyto zpracovány a zobrazeny v jediné chybové hlášce aplikace.

```
catch (AggregateException ae)
{
        StringBuilder sb = new StringBuilder();
        ae.Handle((ex) =>
                {
                        sb.AppendLine(ex.Message);
                        return true;
                });
        ShowError(sb.ToString());
}
```

8 Testování

Testování aplikace bylo prováděno ve dvou úrovních [44]:

- **Developer testing**

  Na této úrovni jde především o kontrolu zdrojového kódu vývojářem a ověření, zda jde program sestavit a spustit.

- **System testing**

  V rámci systémového testování je ověřována funkčnost aplikace jako celku. Během testování jsou simulovány různé případy užití, které mohou v praxi nastat. Tyto testy se používají v pozdější fázi vývoje. Jsou prováděny odpovědnou osobou dle předem připravených testovacích scénářů. Testovací scénář obsahuje zpravidla tyto údaje [45]:

  - Identifikátor
  - Popis testovacího scénáře
  - Počáteční podmínky
  - Specifikace kroků a vstupů
  - Očekávané výsledky
  - Výsledek testování

  Testování obvykle probíhá v několika iteracích, zjištěné chyby jsou opraveny a v následujícím testování znovu otestovány. Součástí této úrovně jsou funkční i nefunkční testy. Použité testovací scénáře jsou uvedeny v Příloha A – Testovací scénáře.

9 Závěr

Cílem práce bylo vytvořit modulární aplikační framework vhodný ke zpracování vstupních obrazových dat dle uživatelem zvolené posloupnosti úprav s možností změny jejich parametrů. Hlavní důraz byl kladen na maximální možný paralelismus při zpracování a rozšiřitelnost frameworku o další operace a algoritmy úprav.

Pro tyto účely byly analyzovány metody a postup zpracování obrazu. Byla provedena rešerše využití zpracování obrazu v biologii a průzkum dostupných softwarových nástrojů. Výsledkem analýzy je návrh aplikačního frameworku, na jehož základu bylo implementováno prototypové řešení. To splňuje všechny definované požadavky a jeho funkčnost byla úspěšně ověřena na úrovních developer a system testingu. Navíc je toto řešení natolik obecné, že není omezeno pouze na zpracování obrazu a umožnuje jakékoliv sekvenční zpracování libovolných dat v závislosti na použitých modulech.

Další rozvoj aplikace je možný v několika směrech. Jedná se především o vývoj nových modulů využitelných nejen v oblasti zpracování obrazu. Dále pak úpravu grafického rozhraní, jeho uživatelské přívětivosti a designu, případně lze celou aplikaci koncipovat jako webové řešení. To by však vyžadovalo větší zásah do stávajícího programového kódu a vytvoření nového grafického rozhraní. Uvedené návrhy tak mohou sloužit jako inspirace k dalšímu využití výsledků této práce.

Seznam použité literatury

[1] ODBORNÉ ČASOPISY. *Automatizace provozu továren* [online]. 2002 [cit. 2015-02-10]. Dostupné z: http://www.odbornecasopisy.cz/index.php?id_document=28501

[2] MICROSOFT. *Kinect for Windows* [online]. © 2015 [cit. 2015-02-10]. Dostupné z: https://www.microsoft.com/en-us/kinectforwindows/develop/default.aspx

[3] TUROVSKY, Barak. Hallo, hola, olá to the new, more powerful Google Translate app. In: *Inside Search* [online]. 1/14/15 [cit. 2015-02-10]. Dostupné z: http://insidesearch.blogspot.cz/2015/01/hallo-hola-ola-to-new-more-powerful.html

[4] MEIJERING, Erik a Gert VAN CAPPELLEN. *Biological Image Analysis Primer*. [online]. 2006 [cit. 2015-02-15]. Dostupné z: http://www.imagescience.org/meijering/publications/download/biap2006.pdf

[5] KAŇKA, Jiří. *Analýza obrazů gelové elektroforézy*. Praha, 2011. Diplomová práce. České vysoké učení technické v Praze, Fakulta elektrotechnická, Katedra kybernetiky.

[6] LUKÁŠ, Jan. *Využití obrazové analýzy v rostlinolékařské praxi*. Praha: Výzkumný ústav rostlinné výroby, 2008. ISBN 978-80-87011-69-0. Dostupné také z: http://www.vurv.cz/files/Publications/ISBN978-80-87011-69-0.pdf

[7] DIETZ, Ch., Martin HORN a Michael BERTHOLD. Integrative Open-Source Software for Image Analysis in Biology. In: *Leica Science Lab* [online]. November 28, 2012 [cit. 2015-02-11]. Dostupné z: http://www.leica-microsystems.com/science-lab/integrative-open-source-software-for-image-analysis-in-biology/

[8] POPELKA, Vladimír. *Srovnávací analýza metodik vývoje software*. Praha, 2009. Bakalářská práce. Vysoká škola ekonomická v Praze, Fakulta informatiky a statistiky Vyšší odborná škola informačních služeb.

[9] GUTHRIE, Scott. Announcing Open Source of .NET Core Framework, .NET Core Distribution for Linux/OSX, and Free Visual Studio Community Edition. In: *ScottGu's blog* [online]. Wednesday, November 12, 2014 [cit. 2015-02-12]. Dostupné z: https://weblogs.asp.net/scottgu/announcing-open-source-of-net-core-framework-net-core-distribution-for-linux-osx-and-free-visual-studio-community-edition

[10] FIŘT, Jaroslav a Radek HOLOTA. *Digitalizace a zpracování obrazu* [online]. 2002 [cit. 2015-02-10]. Dostupné z: http://home.zcu.cz/~holota5/publ/DigZprO.pdf

[11] HAMARSHEH, Qadri. *Digital image processing (750474) Lecture 2* [online]. 2002 [cit. 2015-02-10]. Dostupné z: http://www.philadelphia.edu.jo/academics/qhamarsheh/uploads/Lecture_2_Fundamental_Steps_in_Digital_Image_Processing.pdf

[12] HLAVÁČ, Václav. *RESTAURACE (OBNOVENÍ) OBRAZU PŘI ZNÁMÉ DEGRADACI* [online]. 21. 12. 2004, 0:22:18 [cit. 2015-02-12]. Dostupné z: http://cmp.felk.cvut.cz/~hlavac/Public/TeachingLectures/RestauraceObrazu.pdf

[13] CENTRE FOR REMOTE IMAGING SENSING & PROCESSING. Interpretation of Optical Images [online]. © 2001 [cit. 2015-02-13]. Dostupné z: http://www.crisp.nus.edu.sg/~research/tutorial/opt_int.htm

[14] HUŠEK, M. a P. PYRIH. FOURIEROVA TRANSFORMACE [online]. © 2000-3000 [cit. 2015-02-14]. Dostupné z: http://matematika.cuni.cz/dl/analyza/37-fou/lekce37-fou-pmax.pdf

[15] *Elektrorevue* [online]. 2013 15(4) [cit. 2015-02-14]. ISSN 1213-1539. Dostupné z: http://www.elektrorevue.cz/cz/download/zpracovani-obrazu-pomoci-vlnkove-transformace--image-processing-using-the-wavelet-transform-

[16] HLAVÁČ, Václav. *Matematická morfologie* [online]. 5. 2. 2015, 4:20:12 [cit. 2015-02-15]. Dostupné z: http://cmp.felk.cvut.cz/~hlavac/TeachPresCz/11DigZprObr/71-3MatMorpholBinCz.pdf

[17] HORÁK, Karel. *Matematická morfologie* [online]. 19. 12. 2014, 15:17:10 [cit. 2015-02-15]. Dostupné z: http://midas.uamt.feec.vutbr.cz/POV/lectures-pdf/08_Matematicka_morfologie.pdf

[18] ADAMEC, Václav. *Zpracování a rozpoznávání obrazu*. Olomouc, 2011. Bakalářská práce. Přírodovědecká fakulta Univerzity Palackého v Olomouci, Katedra informatiky.

[19] SAINIS, J., R. RASTOGI a V. CHADDA. *Applications of image processing in biology and agriculture*. [online]. 1. 3. 2004, 8:58:12 [cit. 2015-02-17]. Dostupné z: http://www.barc.gov.in/publications/nl/1999/199905-01.pdf

[20] BOLEČEK, Libor. *Zobrazování černobílých snímků v nepravých barvách*. Brno, 2010. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav radioelektroniky.

[21] *Aquatic Microbial Ecology* [online]. 2000 (22) [cit. 2015-02-19]. ISSN 0948-3055. Dostupné z: http://www.int-res.com/articles/ame/22/a022p103.pdf

[22] ZÍTKA, Michal. *Detekce pohybu v obraze*. Brno, 2008. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizační a měřící techniky.
[23] ALY, Ashraf, Safaai BIN DERIS a Nazar ZAKI. A NEW ALGORITHM FOR CELL TRACKING TECHNIQUE. *Advanced Computing: An International Journal* [online]. 2011 2(6) [cit. 2015-02-20]. ISSN 2229-6727. Dostupné z: http://airccse.org/journal/acij/papers/1111acij02.pdf

[24] RAZANTO, M. *Automatic Recognition of Biological Particles in Microscopic Images* [online]. 14. 9. 2006, 11:04:32 [cit. 2015-02-22]. Dostupné z: http://yann.lecun.com/exdb/publis/pdf/ranzato-07a.pdf

[25] The Revolutionary Technique That Quietly Changed Machine Vision Forever. In: *MIT Technology review* [online]. September 9, 2014 [cit. 2015-02-22]. Dostupné z: http://www.technologyreview.com/view/530561/the-revolutionary-technique-that-quietly-changed-machine-vision-forever/

[26] UCHIDA, S. Image processing and recognition for biological images. *Development, Growth & Differentiation* [online]. 2013 55(4) [cit. 2015-02-22]. ISSN 1440-169X. Dostupné z: http://onlinelibrary.wiley.com/doi/10.1111/dgd.12054/full

[27] CARL ZEISS. *AxioVision Perform to Perfection* [online]. 19. 1. 2009, 17:06:12 [cit. 2015-03-01]. Dostupné z: http://www.kemet.co.uk/_uploads/downloads/AxioVision.pdf

[28] THE GNU OPERATING SYSTEM AND THE FREE SOFTWARE MOVEMENT. GNU Lesser General Public License [online]. 2014/11/08 15:04:00 [cit. 2015-03-02]. Dostupné z: http://www.gnu.org/licenses/lgpl.html

[29] BOOFCV. Performance:OpenCV:BoofCV [online]. last modified on 21 February 2013 [cit. 2015-03-03]. Dostupné z: http://boofcv.org/index.php?title=Performance:OpenCV:BoofCV

[30] BOOFCV. Validation:Algorithms [online]. last modified on 11 June 2014 [cit. 2015-03-03]. Dostupné z: http://boofcv.org/index.php?title=Validation:Algorithms

[31] JANSSEN, Cory. Modular Programming. In: *Techopedia* [online]. © 2010 – 2015 [cit. 2015-03-04]. Dostupné z: http://www.techopedia.com/definition/25972/modular-programming

[32] MAKABEE, Hayim. Separation of Concerns. In: *Effective Software Design* [online]. February 5, 2012 [cit. 2015-03-06]. Dostupné z: http://effectivesoftwaredesign.com/2012/02/05/separation-of-concerns/

[33] NIROSH, L.W.C.. Introduction to Object Oriented Programming Concepts (OOP) and More. In: *CodeProject* [online]. 5 Feb 2015 [cit. 2015-03-06]. Dostupné z: http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep

[34] NAHARI, Hadi a Ronald L. KRUTZ. *Web Commerce Security Design and Development*. Indianapolis: Wiley Pub., © 2011. ISBN 978-0-470-62446-3.

[35] MICROSOFT. Pipes and Filters Pattern [online]. © 2015 [cit. 2015-03-08]. Dostupné z: https://msdn.microsoft.com/en-us/library/dn568100.aspx

[36] MICROSOFT. Pipelines [online]. © 2015 [cit. 2015-03-08]. Dostupné z: https://msdn.microsoft.com/en-us/library/ff963548.aspx

[37] GRAY, Watson. Producer Consumer Thread Race Conditions. In: *50 Shades of Gray Watson* [online]. [cit. 2015-03-09]. Dostupné z: http://256.com/gray/docs/misc/producer_consumer_race_conditions/

[38] .NET Framework. In: *Wikipedia: the free encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, last modified on 17 April 2015 [cit. 2015-04-20]. Dostupné z: http://en.wikipedia.org/wiki/.NET_Framework

[39] JORDAN, Jacob. Using Reflection to load unreferenced assemblies at runtime in C#. In: *CodeProject* [online]. 24 Jan 2009 [cit. 2015-03-10]. Dostupné z: http://www.codeproject.com/Articles/32828/Using-Reflection-to-load-unreferenced-assemblies-a

[40] MICROSOFT. Managed Extensibility Framework (MEF) [online]. © 2015 [cit. 2015-03-10]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/dd460648%28v=vs.110%29.aspx

[41] MICROSOFT. BlockingCollection Overview [online]. © 2015 [cit. 2015-03-12]. Dostupné z: https://msdn.microsoft.com/en-us/library/dd997371%28v=vs.110%29.aspx

[42] MICROSOFT. LINQ to XML [online]. © 2015 [cit. 2015-03-14]. Dostupné z: https://msdn.microsoft.com/en-us/library/bb387098.aspx

[43] MICROSOFT. Lambda Expressions (C# Programming Guide) [online]. © 2015 [cit. 2015-03-14]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/bb397687.aspx

[44] Hlava, Tomáš. Fáze a úrovně provádění testů. In: *Testování softwaru* [online]. 21.8.2011 [cit. 2015-03-21]. Dostupné z: http://testovanisoftwaru.cz/tag/systemove-testovani/

[45] PAGE, Alan a Ken JOHNSTON. *Jak testuje software Microsoft.* Brno: Computer Press. 2009. ISBN 978-80-251-2869-5.

Příloha A – Testovací scénáře

Název případu užití	Přidání filtru
Identifikátor případu užití	IP-01
Účel případu užití	Vložení nového filtru zvoleného typu
Primární aktéři • Uživatel	
Základní scénář 1. Uživatel označí myší požadovaný typ filtru v panelu *Tool Box* a přetažením na plochu diagramu ho umístí do diagramu. 2. Aplikace vytvoří novou instanci filtru určeného typu. 3. Na ploše diagramu je zobrazena grafická reprezentace instance filtru s odpovídajícím počtem vstupů a výstupů. 4. UC končí.	

Název případu užití	Odebrání filtru
Identifikátor případu užití	IP-02
Účel případu užití	Odstranění vybraného filtru
Primární aktéři • Uživatel	
Základní scénář 1. Uživatel označí myší požadovaný filtr na ploše diagramu. 2. Klávesou *delete* na nebo volbou *Delete* z kontextové nabídky zahájí jeho odstranění. 3. Aplikace ukončí běžící proces zpracování. (viz **IP-04 Přerušení procesu zpracování**) 4. Instance filtru a její grafická reprezentace je odstraněna. 5. Jsou odstraněna všechna existující spojení s tímto filtrem. 6. UC končí.	

Název případu užití	Spuštění procesu zpracování
Identifikátor případu užití	IP-03
Účel případu užití	Spuštění vytvořeného procesu zpracování
Primární aktéři • Uživatel	
Základní scénář 1. Uživatel kliknutím na tlačítko *Start processing* zahájí proces zpracování. 2. Tlačítko *Start processing* je zneaktivněno a tlačítko *Stop processing* je zaktivněno. 3. Aplikace spustí dle vytvořeného grafu filtrů proces zpracování. 4. Po dokončení procesu zpracování je tlačítko *Start processing* zaktivněno a tlačítko *Stop processing* je zanektivněno. 5. UC končí.	
Alternativní scénáře **3.a** – Při procesu zpracování dojde k chybě 1. Všechny chybové hlášky jsou uživateli zobrazeny v okně.	

Název případu užití	Přerušení procesu zpracování
Identifikátor případu užití	IP-04
Účel případu užití	Přerušení již běžícího procesu zpracování
Primární aktéři • Uživatel	
Vstupní podmínky Proces zpracování je spuštěn	
Základní scénář 1. Uživatel kliknutím na tlačítko *Stop processing* zahájí ukončení procesu zpracování. 2. Aplikace přeruší běžící proces zpracování. 3. Tlačítko *Start processing* je zaktivněno a tlačítko *Stop processing* je zneaktivněno. 4. UC končí.	
Alternativní scénáře **2.a** – Při ukončování procesu zpracování dojde k chybě 1. Všechny chybové hlášky jsou uživateli zobrazeny v okně.	

Název případu užití	Spojení filtrů
Identifikátor případu užití	IP-05
Účel případu užití	Propojení výstupu a vstupu dvou instancí filtru

Primární aktéři
• Uživatel

Základní scénář
1. Uživatel označí myší v panelu *Tool Box* objekt *Pipe*
2. Nástrojem tužky propojí v panelu diagramu zobrazené kruhové body požadovaného výstupu a vstupu filtrů v tomto pořadí.
3. Aplikace vytvoří propojení filtrů.
4. Takto vytvořené propojení je graficky zobrazeno na ploše diagramu.
5. UC končí.

Alternativní scénáře
3.a – Vytvořené spojení není validní
1. Chybová hláška je uživateli zobrazena v okně.

Název případu užití	Odstranění spojení filtrů
Identifikátor případu užití	IP-06
Účel případu užití	Odstranění vytvořeného propojení dvou instancí filtru

Primární aktéři
• Uživatel

Základní scénář
1. Uživatel označí myší požadované spojení na ploše diagramu.
2. Klávesou *delete* nebo volbou *Delete* z kontextové nabídky zahájí jeho odstranění.
3. Aplikace ukončí běžící proces zpracování. (viz **IP-04 Přerušení procesu zpracování**)
4. Vybrané spojení a jeho grafická reprezentace je odstraněna.
5. UC končí.

Název případu užití	Změna parametru filtru
Identifikátor případu užití	IP-07
Účel případu užití	Nastavení hodnoty parametru zvoleného filtru

Primární aktéři
• Uživatel

Základní scénář
1. Uživatel označí myší požadovaný filtr na ploše diagramu.
2. V panelu *Filter Properties* se zobrazí dostupné informace a parametry vybraného filtru.
3. Uživatel provede změnu hodnoty požadovaného parametru.
4. Aplikace ukončí běžící proces zpracování. (viz **IP-04 Přerušení procesu zpracování**)
5. Aplikace nastaví parametru filtru uživatelem zadanou hodnotu.
6. Aplikace spustí proces zpracování. (viz **IP-03 Spuštění procesu zpracování**)
7. UC končí.

Alternativní scénáře
4.a – Při ukončování procesu zpracování dojde k chybě
1. Všechny chybové hlášky jsou uživateli zobrazeny v okně.
5.a – Zadaný parametr nebo jeho hodnota není validní
1. Chybová hláška je uživateli zobrazena v okně.
6.a – Při procesu zpracování dojde k chybě
1. Všechny chybové hlášky jsou uživateli zobrazeny v okně.

Název případu užití	Nový projekt
Identifikátor případu užití	IP-08
Účel případu užití	Založení nového projektu

Primární aktéři
• Uživatel

Základní scénář
1. Uživatel klikne na tlačítko *New*.
2. Uživatel potvrdí v dialogovém okně, zda chce opravdu nový projekt vytvořit.
3. Aplikace ukončí běžící proces zpracování. (viz **IP-04 Přerušení procesu zpracování**)
5. Aplikace vytvoří nový projekt.
6. UC končí.

Alternativní scénáře
3.a – Při ukončování procesu zpracování dojde k chybě
1. Všechny chybové hlášky jsou uživateli zobrazeny v okně.

Název případu užití	Uložení projektu
Identifikátor případu užití	IP-09
Účel případu užití	Uloží rozpracovaný proces zpracování
Primární aktéři • Uživatel	
Základní scénář 1. Uživatel klikne na tlačítko *Save*. 2. Uživatel určí v dialogovém okně název souboru a potvrdí tlačítkem *Uložit*. 3. Aplikace uloží graf filtrů a jeho grafickou reprezentaci do dvou souborů s uživatelem určeným jménem. 4. UC končí.	
Alternativní scénáře **3.a** – Při ukládání souborů dojde k chybě 1. Chybová hláška je uživateli zobrazena v okně.	

Název případu užití	Načtení projektu
Identifikátor případu užití	IP-10
Účel případu užití	Načte uložený proces zpracování
Primární aktéři • Uživatel	
Základní scénář 1. Uživatel klikne na tlačítko *Open*. 2. Uživatel vybere v dialogovém okně požadovaný soubor a potvrdí tlačítkem *Otevřít*. 3. Aplikace načte graf filtrů a zobrazí jeho grafickou reprezentaci. 4. UC končí.	
Alternativní scénáře **3.a** – Při načítání souborů dojde k chybě (chybná data, chybějící soubor) 1. Chybová hláška je uživateli zobrazena v okně.	