Mendelova univerzita v Brně
Provozně ekonomická fakulta

Návrh a implementace nástroje pro konfiguraci a spouštění robotického frameworku

Bakalářská práce

Vedoucí práce:
Ing. František Ostřížek

Petr Kalas

Brno 2016

Rád bych poděkoval panu Ing. Františkovi Ostřížkovi za vedení práce a také panu Ing. Janu Kolomazníkovi, Ph.D. za cenné rady při konzultacích. Dále bych rád poděkoval rodině za velkou podporu při studiu a dokončování práce. Zvláštní poděkování potom patří kamarádům a komunitě Stack Overflow za vždy rychlou a dostupnou programátorskou výpomoc.

Čestné prohlášení

Prohlašuji, že jsem tuto práci: **Návrh a implementace nástroje pro konfiguraci a spouštění robotického frameworku**
vypracoval samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*
Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.
Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

21. května 2016 .....

Abstract

Kalas, P. Draft and implementation of a tool for configuration and startup robotic framework. Bakalářská práce. Brno, 2016.

The main goal of the thesis is to create a desktop application for working with configuration files. In the solving process several data transformations were used and for the presentation layer was chosen Java FX. The application is an editor with data represantion in the form of an interactive graph. The main result is to facilitate the work with robotic framework of the team Aistorm at Mendel university.

Keywords: desktop application, parsing, graph, reflection, Java, JavaFX, XML

Abstrakt

Kalas, P. Návrh a implementace nástroje pro konfiguraci a spouštění robotického frameworku. Bakalářská práce. Brno, 2016.

Cílem bakalářské práce je vytvoření desktopové aplikace pro práci s konfiguračními soubory. Při řešení bylo použito několik transformací dat a pro prezentační vrstvu byla zvolena JavaFX. Vytvořená aplikace je editorem s reprezentací dat v podobě interaktivního grafu. Hlavním výsledkem je usnadnění práce s robotickým frameworkem týmu Aistorm na Mendelově univerzitě.

Klíčová slova: desktopová aplikace, syntaktická analýza, graf, reflexe, Java, JavaFX, XML

Obsah

Seznam obrázků

Seznam ukázek kódu

1 Úvod a cíl práce

1.1 Úvod

V rámci robotického týmu Mendelovy univerzity v Brně vznikl projekt Buggyman, jehož cílem bylo vytvoření autonomního robota schopného se pohybovat po zpevněných cestách. Původní motivací projektu byla účast v několika robotických soutěžích.

Nynější robotovo prostředí je areál arboreta Mendlovy univerzity, kde by mohl zastupovat roli průvodce po areálu a následně tak například dovést návštěvníka arboreta na místo zvolené prostřednictvím webové aplikace. Takové využití by mohlo přispět k propagaci skupiny Aistorm, univerzitního arboreta a tím k propagaci celé Mendelovy univerzity.

Robot se od své první verze postavené na základě modelářského dálkově ovládaného auta dostal k nynějšímu robustnějšímu podvozku s pohonem stejnosměrným elektromotorem. Řídící program robota je naprogramován v jazyce Java. V tomto jazyce vytvořený framework, jenž tvoří BuggyManovu výbavu, obstarává jeho hardwarové komponenty a komunikaci mezi nimi. Nastavení modulů, které jsou reprezentací hardwarových robotem využívaných komponent, je vyjádřeno jejich výčtem, atributy a závislostmi v konfiguračních souborech. Na vzniklou potřebu snadnější a efektivnější tvorby konfiguračních souborů se stal odpovědí návrh a vývoj uživatelsky přívětivé aplikace, která by úpravu a tvorbu konfigurací výrazně usnadnila.

1.2 Cíl práce

Cílem bakalářské práce je navrhnout a implementovat aplikaci, která nabídne možnost přečíst konfigurační soubory pro mobilního robota ve formátu značkovacího jazyka XML a následně vytvořit grafickou reprezentaci přečtených dat. Grafická podoba konfiguračních souborů bude mít formu interaktivního grafu.

Uzly v tomto grafu jsou myšleny jednotlivé robotické moduly. Zobrazované prvky budou generovány na základně atributů a závislostí dané komponenty na dalších komponentách. Tyto závislosti by měli tvořit hrany grafu. Nejpodstatnější část aplikace nebude jenom grafická reprezentace existujících dat, ale možnost upravování a vytváření nových konfiguračních souborů. Prostřednictvím nabídky, která má za úkol reflektovat všechny použitelné komponenty v robotickém frameworku, bude možné do nové či upravované práce přidávat instance komponent a následně zadat požadované atributy a navázat požadované závislosti. Kromě možnosti načítání a upravování existujících a vytváření nových konfiguračních souborů aplikace uživateli poskytne kontrolu nad správností. Tato kontrola bude ve formě kontrolní funkce vytvářející hlášení a také pomocí grafických prvků upozorňujících na nedostatky. Aplikace by také měla umožnit ze svého prostředí robotický framework spustit.

Plánem je navržený koncept implementovat pomocí programovacího jazyku Java, několika nástrojů spjatých s Java platformou a za pomocí platformy JavaFX

umožňující rychlý vývoj moderních multiplatformních GUI aplikací. Grafické uživatelské prostředí by mělo uživateli při práci poskytnout dostatek informací a ergonomie.

Práce bude rozvržena na část teoretickou a praktickou. Teoretickou částí se stane seznámení se současným stavem projektu BuggyMan a popis technologií, ze kterých vychází pozdější návrh a vývoj aplikace. Částí praktickou pak vytvoří samotný postup navržení a implementace aplikace.

2 Použité technologie

Tato kapitola poskytuje základní popis technologií, na kterých budou vývoj a implementace postaveny. Volba jazyka Java, stejně jako použití nástroje Maven a značkovacího jazyka XML, je dána tím, že předmětem vývoje se stává rozšíření existující práce. Volbou jiných technologií než těch, na kterých je postavena rozšiřovaná práce, by bylo velice neefektivní ne-li nemožné. Dostatečný prostor pro vlastní volbu technologie nicméně nabízí například výběr prostředků pro samotnou grafickou reprezentaci a další.

2.1 Jazyk a platforma Java

Počátky programovacího jazyka Java sahají až do první poloviny devadesátých let. Jazyk s pracovním názvem Oak, jehož tvůrci byli ze společnosti Sun Microsystems, vycházel z jazyka C++. Roku 1995 byl Oak přejmenován a byla uvolněna první vývojová sada s názvem Java Development Kit. Java se od té doby vypracovala do pozice, kdy už není chápána jen jako programovací jazyk, ale i jako samostatná platforma. Jedná se softwarovou platformu, která je hardwarově nezávislá, což je také jednou z její největších výhod. Mezi další předenosti patří vysoký výkon, distribuovanost, dynamičnost a vysoká míra zabezpečení. Platforma Java se vyznačuje také kvalitní dokumentací API na technologii JavaDoc, velkém množství knihoven a nástrojů nejrůznějšího druhu. V současné době Java spadá pod křídla společnosti Oracle. (Schildt, 2012)

Java je jazyk objektově orientovaný, ne však striktně. Existují v ní i primitivní datové typy. Jedná se o jazyk kompilovaný a princip práce se dá zjednodušeně popsat následujícím způsobem. Veškerý zdrojový kód je ukládán v textových souborech s příponou .java, které pak kompilátor nazvaný *javac* překládá do souborů *.class*. Tyto soubory obsahují strojový kód tzn. *bytecode*. Ten je spouštěn pomocí JVM - Java Virtual Machine, jejíž instance se při spuštění java aplikace inicializuje a následně se stará o běh programu. Java je také rozdělena do edic následujícím způsobem:

- **Java SE (Standard Edition)**: jedná se o edici zaměřenou pro vývoj na stolní počítače.
- **Java ME (Micro Edition)**: tato edice se učena pro zařízení s omezenými prostředky jako mobilní zařízení.
- **Java EE (Enterprise Edition)**: uplatnění této edice se nachází na poli serverových a webových aplikací.

(Schildt, 2012)

2.2 Apache Maven

Open-source nástroj Maven od společnosti Apache je určen pro správu, automatizaci a řízení sestavení aplikace v různých programovacích jazycích. Jeho parketou je ale především programovací jazyk Java. Maven poskytuje programátorovi spoustu výhod. Zajišťuje popis projektu pomocí xml souboru, v kterém jsou obsaženy informace o projektu a především závislosti na externích knihovnách. O tyto závislosti se Maven sám stará, umožňuje jejich vyhledávání přímo z prostředí IDE (integrace existuje pro všechny populární vývojové prostředí) a tím výrazně usnadňuje programátorovi práci. Součástí práce tohoto správce závislosti je vytváření lokálního repozitáře s knihovnami jazyka Java. (Pinkas, 2006)

2.3 XML

XML je zkratka pro eXtensible Markup Language, tj. rozšiřitelný značkovací jazyk. Formát je konsorciem W3C definován jako formát pro přenos obecných dokumentů a dat. Vychází z obecného standardu SGML (Standard Generalized Markup Language), stejně jako například formát dokumentů HTML. Na rozdíl od HTML, které má pevnou sadu značek, značky v XML pevné nejsou a mohou být libovolně definovány na základě konkrétních potřeb. Samotná definice značek může být součástí XML dokumentu nebo na ni může pouze vést odkaz. Možností definice sady značek může být ale i dopředná domluva. (Holubová, Pokorný, 2008)

2.4 JavaFX

JavaFX je jedna z technologií patřící do ekosystému Javy. Jedná se o framework určený pro vytváření vizuálně propracovaných aplikací, které jsou určeny pro všechny platformy, na nichž se můžeme setkat s Javou. Jedná se o nástupce zastaralé technologie Swing. JavaFX se vyznačuje podporou velkého množství technologií pro vizualizaci, a to vše takovou cestou, aby byl vývoj co nejjednodušší. První veřejné představení JavaFX bylo na konferenci JavaOne v květnu 2007 a v současné době je aktuální verze JavaFX 8. Označení verze číslem osm je z důvodu integrace celé JavaFX platformy do základní distribuce programovacího jazyka Java. Existují 3 způsoby nasazení.

- Plugin do Java appletů v prohlížečích
- Standardní desktopové aplikace
- Mobilní aplikace

Důvodů volby JavaFX 8 z výběru dalších grafických frameworků, jako jsou AWT, SWT nebo Swing, bylo hned několik. Jedná se především o celkovou modernost tohoto řešení a z ní vyplývajících výhod. Je to například možnost stylování pomocí kaskádových stylů, podpora stavby aplikací na návrhovém vzoru (respek-

tive architektuře) MVC (Model-Control-Viewer) a s ní související technologie FXML (podrobněji popsána v samostatné sekci) a také nástroj JavaFX Scene Builder.

2.4.1 Architektura platformy JavaFX

Následující rozdělení popisuje architekturu technologie grafického frameworku JavaFX.

- **Stage**: jedná se o kontejner nejvyšší úrovně, který je zobrazen jako okno se svým záhlavím. Stage je vytvářen platformou a právě na zmiňovaném záhlaví lze snadno rozpoznat na jaké platormě je JavaFX aplikace spuštěna.
- **Scene**: scéna je kontejner udržující veškerý obsah aktuálně zobrazených objektů. Graf objektů, který scéna udržuje, musí být specifikován kořenovým objektem, ke kterému se následně další objekty typu Node přidávají. Scéna je obvykle kontejneru typu Stage předána při inicializaci aplikace a následně je možné operovat mezi dalšími scénami v závislosti na konkrétní potřebě.
- **Node**: node neboli uzel je nadtřída všech zobrazitelných komponent, které následně tvoří zmiňovanou grafovou strukturu udržovanou scénou a udržují si svůj stav. Každý uzel může mít více potomků, ale vždy jen jednoho určitého předka. Kromě velkému výčtu standardních formulářových prvků může Node představovat vykreslené 2D grafické prvky jako čáry, kruhy, cesty, ale i 3D prvky.
- **Transition**: jedná se o veškeré změny stavů prvků typu Node, které se dějí dynamicky v čase. Konkrétně jde o různé rotace, postupné vykreslování a mizení prvků a podobně.
- **Effect**: efekty se v kontextu JavaFX myslí veškeré možnosti dekorace prvků Node.
- **Render engine**: obstarává veškeré prvky typu Node, Transition a Effect. Úlohou je vyrenderovat tyto v jazyce Java konkrétní objekty na scénu v jeden celistvý finální obraz.

(Oracle, 2015)

2.4.2 FXML

Důležitou částí GUI frameworku JavaFX je deklarativní jazyk založený na XML zvaný FXML. V kontextu MVC architektury se dá říci, že představuje společně s kaskádovými styly prvky prezentačí vrstvy. Účelem jazyka je vytváření uživatelského rozhraní odděleného od jakékoliv programové logiky a tedy bez nutnosti psaní nativního kódu Java. FXML se zapisuje do vlastních dokumentů, které jsou následně provázány s prvky typu Node nebo Scene. Pro provázání událostí prvků se definují *handlery*. Dokumenty FXML se mohou kromě klasického způsobu edito-

vání v jakémkoli XML editoru také dynamicky generovat pomocí nástroje JavaFX Scene Builer.

2.5 Návrhové vzory

Návrhové vzory jsou programátorskou obdobou matematických vzorců. Jedná se o sadu určitých univerzálních a časem ověřených řešení. Jejich použití nebývá v žádném případě jedinou volbou, jedná se ale zpravidla o tu nejefektivnější volbu z hlediska objektového návrhu. Výhody návrhových vzorů se dají shrnout do třech bodů.

- Řešení lze získat rychleji bez zbytečného přemýšlení.
- Výrazné zmenšení pravděpodobnosti chyby.
- S použitím návrhových vzorů přichází určitá terminologie usnadňující komunikaci.

Posledním bodem se myslí například situace, kdy při sdělení jinému programátorovi, že řešení je na bázi návrhového vzoru *dekorátor*, dochází k plnému pochopení bez dalšího vysvětlování. Analogii s matematickými vzorci lze ještě využít dále. Pokud se v nich dosazují čísla, tak u návrhových vzorů lze mluvit o dosazování tříd a objektů. (Pecinovsky, 2010)

Takové přirovnání napovídá o dalším důležitém faktu týkajícího se návrhových vzorů. Jedná se o záležitost týkající se objektově orientovaného programování. Výhodou pak je, že návrhové vzory představují řešení natolik obecné, že jejich implementace je možná ve všech objektově orientovaných jazycích.

Obecně se návrhový vzor skládá z následujících čtyř základních prvků. (Pagejones, 2001)

- **Název vzoru**: skládá se zpravidla z jednoho nebo dvou slov. Zvětšuje návrhovou slovní zásobu a usnadňuje tak komunikaci.
- **Problém**: jedná se o popis problému, při kterém se má vzor použít. Může se jednat o popis specifických návrhových problémů, nebo také o seznam podmínek, které se musí splnit, aby použití vzoru dávalo smysl.
- **Řešení**: popisuje prvky, z nichž se návrh skládá - jejich vztahy, povinnosti a spolupráci. Jedná se o abstraktní popis způsobu použití a uspořádání prvků k vyřešení problému.
- **Důsledky**: vyjadřují výsledky a kompromisy použití vzoru. Zabývají se často prostorovými a časovými kompromisy. Rovněž k nim patří vliv na tvárnost, rozšířitelnost a přenositelnost systému.

Existuje také určité dělení vzorů do třech základních kategorií podle jejich účelu. Každou z těchto kategorií pak zastupuje velké množství konkrétních vzorů.

- **Tvořivé vzory** řeší vytváření a skládání objektů v návrhu.
- **Strukturální vzory** jsou pro uspořádání a třídění objektů v návrhu.
- **Vzory chování** charakterizují způsoby, podle nichž se v objektovém návrhu o řídí komunikace a rozdělování povinností.

(Ivo, 2013)

2.5.1 Model-Controller-Viewer

Tento návrhový vzor je často chápan jako softwarová architektura. Rozděluje datový model aplikace, uživatelské rozhraní a řídící logiku do třech nezávislých komponent. Díky tomu změny v jedné z nich mají pouze minimální vliv na ostatní. Tok událostí v MVC aplikaci lze popsat následovně. (Margai, 2013)

- Uživatel interaguje s uživatelským rozhraním - View.
- Akce je zachycená řadičem, který také rozhodne jak reagovat - Controller.
- V komponentě model typicky pak dochází k určité modifikaci dat - Model.
- Změny jsou zobrazeny uživateli pomocí View.

Dalším způsobem, jak rozdělení popsat, je rozdělení na vrstvy. Můžeme tedy mluvit o následujících třech vrstvách.

- **Prezentační vrstva**: zajišťuje komunikaci s uživatelem prezentováním dat v přívětivé a čitelné formě.
- **Vrstva logiky**: data, která přenáší mezi prezentační a datovou vrstvou zde prochází rozhodovací logikou a transformacemi jako výpočty a agregace.
- **Datová vrstva**: představuje uložiště dat.

Tento návrhový vzor je široce používán ve webových projektech. V tomto odvětví existují celé frameworky, jejichž úkolem je vnést tento vzor do implementace. S MVC se lze setkat ale i v mnoha dalších odvětvích. Příkladem je právě softwarová platforma pro tvorbu okenních aplikací JavaFX.

2.5.2 Builder

Úkolem tohoto návrhového vzoru je tvorba různých (a zpravidla složitých) instancí dané třídy podobným konstrukčním procesem. Největším přínosem se tak stává použití opakované používání jednoho procesu s mírnými změnami - tedy v projektu ubývá opakujícího se kódu. Vzor se řadí do *tvořivých návrhových vzorů*.

Základem je interakce následujících třech objektů.

- **Product**: vytvářený objekt.

- **Director**: objekt, který řídí vytváření *produktu*. Tedy určuje pořadí vytváření jednotlivých částí, stará se o validaci jeho stavu a případně hlášení chyb.
- **Builder**: zde probíhá samotná realizace vytváření. Udržuje logiku konkrétních kroků konstrukce *produktu*. Tento objekt vrací *produkt*.

(Pecinovsky, 2013)

2.6 Teorie grafů

V matematice se grafem rozumí nejčastěji grafické znázornění funkční závislosti. V teorii grafů se grafem rozumí objekty popsané množinou vrcholů a množinou hran. (Šeda, 2003)

Graf je definován jako uspořádaná trojice G = (U, H, f), kde U představuje neprázdnou množinu uzlů, H je konečná množina hran a f je zobrazení incidence. To přiřazuje dvojici uzlů právě jednu hranu. (Foltýnek, 2011)

Za pomoci grafů lze reprezentovat různé struktury z mnoha oborů. Za použití ohodnocení hran nebo vrcholů se grafy stávají vhodným nástrojem pro reprezentaci reálných sítí a lze je pak používat k analýze dopravy, počítačových sítí nebo v elektrotechnických schémat. Existuje mnoho typů grafových struktur a také algoritmů pro práci s nimi.

2.7 UML modelování

UML, neboli unifikovaný modelovací jazyk, je univerzální jazyk pro vizuální modelování systému. Má široké pole působiště. Nejčastěji je jeho využití spojováno s modelováním objektově orientovaných systémů. Navržení tohoto jazyka bylo důsledkem potřeby spojení existujících modelovacích technik a softwarového inženýrství. (Arlow, 2007)

Existuje více způsobů jak se UML ve vývoji softwaru používá. Steve Mellor a Martin Fowler nezávisle na sobě definovali s charakteristikou tří způsobů.

- **UML pro tvorbu náčrtku**: jedná se nejčastější použití UML. Účelem se stává především usnadnění komunikace. Náčrtky lze použít pro dopředné inženýrství - vytváření kódu na základě diagramu, ale také pro zpětné inženýrství. V takovém případě se diagram vytváří na základě již existujícího kódu za účelem jeho pochopení.
- **UML pro podrobný návrh**: tento způsob je stojí především na kompletnosti. Předpokladem je možnost tento návrh použít jako detailní návrh pro programátory, od kterého se pak očekává práce s nutností minima přemýšlení a nulové vlastní invence. V popisovaném scénáři je autorem diagramu zpravidla zkušenější vývojář.
- **UML jako programovací jazyk** Při velmi pokročilém použití UML nástrojů se lze dostat do stavu, kdy se samotné programování stává mechanickou záleži-

tostí natolik, že se stává předmětem automatizace. Modelování je pak schopno zcela zastoupit programování a diagramy lze překládat přímo do spustitelného kódu.

(Fowler, 2009)

2.8 Testování

Obecně testování slouží k zlepšování kvality výrobků před jejich uvedením na trh. Analogicky je tomu u testování software. Cílem je ladění kódu, ověřování požadované funkčnosti a správné distribuci chyb k vývojářovi.

Testování není konečný proces. Jakýkoliv systém nelze otestovat komplexně a tedy nelze prohlásit, že by byl systém stoprocentně funkční. Důvody jsou velké množství vstupů, velké množství výstupů a především, že specifikace aplikace je subjektivní. Testování je tedy proces založený na riziku. Úkolem je vytváření optimálního počtu testů a rozvrhnout je tak, aby pokrývaly stěžejní funkcionalitu aplikace. (Jandák, 2012)

Existuje množství přístupů k testování software a stejně tak způsobů dělení. Jedním z nich je například dělení testování na základě znalostí o testovaném systému, kdy vymezujeme přístup *black-box*, při kterém je účelem analyzovat chování softwaru vzhledem k očekávaným vlastnostem tak, jak ho vidí uživatel. Druhým přístupem dle zmíněného dělení je *white-box*, při kterém má tester přístup ke zdrojovému kódu a na základě něj testy provádí. Jiným dělením, které je v kontextu této práce dělení důležité, je dělení na testy *automatické* a testy *manuální* podle toho, zda jsou prováděny člověkem nebo softwarem.

Stupně testování specifikují v jakém časovém horizontu a na jaké úrovni se testování provádí:

- **Testování programátorem**: neformální testování programátorem na principu zkoušení právě napsaného kódu.
- **Testování jednotek**: známe pod pojmem *unit-testing*. Proces testování co nejmenších částí kódu. Jedná se o automatické, opakovatelné a snadno zpracovatelné testy.
- **Integrační testování**: slouží pro testování větších celků, resp. komponent a komunikaci mezi nimi. Mohou být také automatické, ale i manuální.
- **Systémové testování**: jedná se komplexní testy systému v nejpozdějších částech jeho vývoje. Testuje se dle scénářů použití aplikace a předmětem testování bývá i vykonnostní a bezpečností záležitosti.
- **Akceptační testování**: tyto testy bývají prováděny ze strany zákazníka. Principiálně se jedna o velmi podobné testování jako systémové testování. Úkolem je zjistit, zda je testovaný systém připraven na ostré nasazení do provozu.

(Borovcová, 2008)

S testováním souvisí pojem *mockování*. Jedná se v rámci testování o práci s tzv. *mock objekty*. Zjednodušeně se jedná o objekty, které pro účely testování slouží jako náhrady za objekty s reálnou implementací. Výhody přístupu testování s použitím mock objektů:

- Zrychlení vykonávání testů díky zjednodušení kódu, který by se musel při plné implementaci mockovaných objektů vykonávat.
- Možnost simulace takových scénářů při testování, které by byly jinak obtížné vytvořit.
- Průzkum takových stavů a interakce objektů, které by bez mockování byly nenasimulovatelné nebo jen velmi obtížně.

(Koskela, 2013)

2.9 Použité nástroje

Určitou podmnožinou volby technologií je volba vhodných nástrojů pro práci s nimi. V kontextu práce se jedná především o volbu vývojového prostředí - IDE. Přestože rozhodování v tomto směru bývá spíše zatíženo osobními zkušenostmi a preferencemi vývojáře, existují mezi vývojovými prostředími i takové rozdíly, které mohou být při volbě pro specifický projekt rozhodující. Konkrétně tedy bylo velmi vhodné volit takové IDE, které nabídne určitou integraci s nástroji na jiné části práce - UML modelování a vytváření grafického návrhu.

2.9.1 IntelliJ IDEA

Z vývojových prostředí, které zmíněné kritéria splňovala, bylo zvoleno prostředí IntelliJ IDEA. Právě integrace s jinými nástroji dosahuje u tohoto IDE velmi vysoké úrovně. Pro práci s nástroji JavaFX Scene Builder a Visual Paradigm není nutné prostředí IDEA ani opouštět. Samozřejmostí je pak u takové spolupráce reakce na provedené změny skrze všechny zmíněné nástroje.

Další důvody pro volbu tohoto prostředí tvoří pak záležitosti týkající se osobních preferencí autora. Jedná se o velice dobře zvládnuté refaktorování kódu a inteligentní systém doplňování a našeptávání. Velkými výhodami tohoto prostředí je také velká míra přizpůsobivosti a dobře vymyšlený způsob klávesových zkratek a navigace.

2.9.2 JavaFX Scene Builder 2.0

Jedná se o nástroj, který nabízí tvorbu grafických uživatelských prostředí bez nutnosti psaní kódu. Práce je založena na intuitivním skládání komponent na pracovní desce v celistvý obraz. Na pozadí se generuje FXML kód do daného FXML souboru. Nástroj je možné používat samostatně, ale výhodou je jeho možnost integrace k IDE,

který pak pro FXML soubory volí Scene Builder jako defaultní editor a změny v něm provedené se následně v reálném čase projevují i v editoru.

Samotný JavaFX Scene Builder je velice dobrou ukázkou možností a síly JavaFX platformy, jelikož je v něm sám vytvořen.

Nástroj samotný se skládá z menu, levého panelu rozděleného na Library panel a Document panel, pracovní plochy, kde je vyobrazen skládaný návrh, a pravého panelu rozděleného na části: Properties, Layout a Code. Poslední tři zmiňované části se vztahují vždy k aktuálně vybrané komponentě na pracovní ploše.

- **Library Panel**: obsahem je několik tématicky pojmenovaných záložek, z nichž každá skrývá odpovídající prvky. Prvky je odsud možné pomocí drag-and-drop tahů umísťovat na pracovní plochu.
- **Document Panel**: přidáváním prvků na pracovní plochy vznikla v tomto panelu jejich stromová struktura. Z tohoto místa je umožněna rychlá orientace a výběr prvků z pracovní plochy.
- **Properties Panel**: zde je vyobrazen seznam všech vlastností vybrané komponenty v několika podkategoriích. Jedná se o množinu vlastností pro komponentu specifických, a také o množinu vlastností, které jsou pro komponenty společné.
- **Layout Panel**: tato část pravého panelu je zaměřena na umístění vybrané komponenty v jejím aktuálním prostředí. Umístění je zde možno volit relativní či absolutní v souřadnicovém systému. Kromě umístění je zde také řešena například rotace.
- **Code Panel**: pod poslední částí s názvem Code se skrývá sada možností pro propojení s funkční logikou. V architektuře MVC tedy propojení pohledu z prezentační vrstvy (View) s odpovídajícím ovladačem (Controller).

2.9.3 Visual Paradigm for UML

Tento nástroj spadá do kategorie CASE nástrojů. Pro ty se uvádí následující obecná definice:

Case nástroj je softwarový produkt určený k podpoře jedné nebo více inženýrských aktivit v rámci procesu vývoje softwaru. (Brown, 1994)

Visual Paradigm podporuje všechny druhy diagramů jazyku UML 2. Nástroj umožňuje velice snadno vytvářet kód na základě diagramů, ale také zpětně generovat diagramy na základě existujícího kódu. Výhodou tohoto nástroje, která především vedla k jeho volbě pro tuto práci, je jeho přímá integrace do prostředí IDE IntelliJ IDEA.

2.10 Použité Java knihovny

Použití následujících knihoven by teoreticky nemuselo být pro vypracování řešení nezbytné, nicméně práce s nimi některé činnosti výrazně zjednodušuje.

2.10.1 Jsoup

Jsoup je knihovna poskytující velice intuitivní API k extrakci dat a manipulaci s dokumenty založené na DOM (Document Object Model) modelu. Přestože je podle její samotné dokumentace určena pro HTML, je možné ji velice efektivně využít i k práci s XML soubory. Knihovna umožňuje převést celý validní HTML dokument do své vlastní struktury, ve které se pak pomocí několika vlastních datových typů a velkému výběru vhodných funkcí lze snadno pohybovat. Výběr může být realizován pomocí atributů, tříd, značek, vztahů mezi elementy a velice efektivním typem výběru je použití selektorů - způsobem, který se využívá v CSS nebo JQuery.

Tato knihovna se ukázala jako velice užitečná pro zpracování konfiguračních souborů robotického frameworku Aistorm.

2.10.2 Reflections

V kontextu programovacích jazyků se *reflexí* (nebo také *introspekcí*) myslí schopnost jazyka získat informace o vlastní struktuře. Pojednává se tedy o určitém typu metadat. Pro reflexi je v Javě připraven balík java.lang.reflection. Obsahem toho balíku jsou například třídy, které vyjadřují konstrukce jazyka:

- Annotation
- Method
- Constructor

Knihovna Reflections pak umožňuje jednoduše provádět pokročilejší reflexivní operace jako získání všech tříd dědící danou třídu nebo selekci všech parametrů označených danou anotací.

2.10.3 JUnit

JUnit je testovací framework pro jazyk Java. Díky své popularitě a rozšířenost v komunitě se v podstatě jedná o standard pro testování Java kódu. Samozřejmostí je tak podpora ve všech vývojových prostředích. Junit je určen pro vytváření automatických testů a je důležitou součástí *test-driven-development* přístupu. Hlavními částmi frameworku jsou:

- **Fixtures**: představují *fixní* množinu objektů, které jsou vytvořeny pro účely konkrétního testu. Pracuje se s nimi pomoci metod volaných před a po testu *setUp()* a *tearDown()*.
- **Test suites**: slouží pro svázání více testů tak, aby bylo zajištěno, že se provedou společně. Pro tyto účely jsou k dispozici anotace *RunWith* a *Suite*.
- **Test runners**: obstarávají samotný průběh testování - vyhledání testů a jejich spuštění.

- **JUnit classes**: jsou třídy udržující logiku testovacích scénářů.

2.10.4 Mockito

Jedná se o open-source framework určený pro snadnější vytváření mock objektů. Mockito je velmi dobře použitelné s testovacím frameworkem JUnit. Nabízí svůj vlastní *Test runner*, který při použití počítá s použitím anotací, kterými se definují mock objekty.

3 Projekt BuggyMan

Základní stavba robota BuggyMan se skládá z modelářského podvozku, který nese řídící počítač.

3.1 Hardwarová stavba

Podvozek, na kterém je robot usazen, je upravený typ Lossy 8eight. Jedná se o robustní buggy kit v měřítku 1:8, který má poháněná všechny 4 kola a to nízkootáčkovým stejnosměrným elektromotorem s převodovkou. Další výbavou je GPS modul, kamera, kompas a sada senzorů. Jedná se o jeden ultrazvukový a tři infračervené senzory vzdálenosti. Samotné řízení probíhá v počítači HP Revolve 810 společně s deskou Arduino Mega 2050. Napájení je v robotovi řešeno ve dvou okruzích, a to první pro podvozek a senzory za využití dvou NiMh baterií a druhý jako samostatný akumulátor počítače. (Ondoušek, 2015)

Jak robot BuggyMan aktuálně vypadá lze vidět na obrázku 1.

Obrázek 1: Robot BuggyMan s podvozkem Lossy 8eight
(Ondoušek, 2015)

3.2 Programové vybavení

Pro komunikaci s hardwarem se tým Aistorm rozhodl vytvořit řídící program robota jako framework, který je založen na modulovém přístupu. Hlavní motivací pro toto rozvržení je velice dobrá možnost týmové spolupráce. Framework je složen z tzv. *frozen spots*, udávající architekturu a pravidla pro komponenty a jejich vztahy, a tzv. *hot spots*. Ty naopak zastupují části kódu, jenž mohou být zaměnitelné. Jednoduchým příkladem takového rozdělení je v objektově orientovaném programování abstraktní třída, tedy bez implementace, představující frozen-spot, zatímco třída doplňující konkrétní implementaci lze považovat právě za hot-spot. Tento přístup se tedy stává velice výhodným pro použití k robotu, jehož i fyzická stavba se skládá z hardwarových modulů (senzory, podvozek a další). Hot-spoty se tedy v tomto případě zamýšlí programové jednotky udržující konkrétní implementaci - tedy moduly. Pro tvorbu frameworku zvolil tým Aistorm konkrétní jazyk Java ve verzi 1.6 v prostředí eclipse.

Nedílnou součástí frameworku je systém logování, který poskytuje možnost sledování záznamů provedených akcí a analýzu chyb. Logování je vyřešeno pomocí systému slf4j, jehož doménou je vytvoření kompatibility mezi více logovacími systémy. (Ostřížek, 2015)

4 Metodika

Pro dosažení stanoveného cíle je nutno definovat posloupnost kroků vedoucích k řešení.

1. Analýza současného stavu
    - Definice výchozího stavu a zorientování se ve frameworku Aistorm
    - Analýza konfiguračních souborů
    - Sběr požadavků na aplikaci
2. Návrh řešení
    - Vytvoření diagramu případu použití
    - Definování workflow aplikace - vytvoření diagramu aktivit
    - Vytvoření grafického návrhu řešení - drátěný model
    - Definice entit a vztahů mezi nimi
3. Implementace řešení
    - Nalezení vhodných řešení pro transformaci dat
    - Tvorba grafických reprezentací entit a vlastního programu
    - Implementace v jazyku Java a testování

5 Analýza

Po seznámení se s technologiemi, nutnými pro porozumění stávajícímu stavu problematiky a také s technologiemi nezbytnými pro vypracování plánovaného řešení, byla úspěšně vymezena metodika. Její první částí se stala analýza. Pro pozdější návrh a implementaci je nutné podrobně analyzovat především body již hotového řešení (framework Aistorm), na které se bude vytvářená aplikace vázat. Výstup vytvářené aplikace na tvorbu konfiguračních souborů je také bod, na který se bude dále navazováno dalším zpracováním, proto je potřeba tento řetězec pečlivě promyslet a tyto body návaznosti správně analyzovat.

5.1 Aistorm framework

Framework je tvořen, jak již bylo zmíněno, moduly. Podrobnější pohled odhaluje ,že se jedná se o samotatné Java projekty založené na správci závislostí Maven. Pokud modul potřebuje komunikovat s jiným modulem, stačí ho zaevidovat v *Project Object Model* souboru (POM.xml) projektu, který závislost vyžaduje.

Moduly mohou obsahovat mnoho tříd, z pohledu konfiguračních souborů je ale důležitá jen určitá podmnožina z nich. Jedná se o třídy představující čtyři entity, které budeme dále nazývat *komponenty*.

- **Aspect**: jedná se o zařízení starající se o volání všech veřejných metod, jejichž volání a návratovou hodnotu zaznamenává do logového souboru.
- **Logger**: zařízení starající se o zapisování zpráv ze všech ostatních zařízení na jedno určené místo.
- **Device**: jedná o zařízení, které je určeno pro specifickou úlohu. Zařízení může představovat senzor, podvozek, výpočetní jednotku nebo jakoukoli jinou věc či funkci.
- **View**: tato komponenta je vázaná na komponentu typu Device. Po spuštění samotného programu frameworku dostává prostor pro prezentaci dat, které zpracovává dané zařízení.

Zatímco Aspect a Logger není nutno implementovat v rámci frameworku vícekrát, tak právě komponenty Device jsou obsahem většiny modulů. Volitelně takový Device modul obsahuje právě komponentu View k ní vázanou. Device se tedy stává určitým základním stavebním kamenem frameworku a i zde mu bude věnována zásadní pozornost. Jedná se totiž o třídy pro tuto práci zcela esenciální, právě s nimi se bude dále pracovat.

Device vzniká z Java třídy ve chvíli, kdy je mu přidělena rodičovská třída ADevice. Jedná se o třídu abstraktní definující základní množinu atributů a metod, které jsou žádoucí u každé komponenty typu Device. Právě s potomky ADevice má za úkol vytvářená grafická aplikace pracovat - jsou jimi totiž myšleny uzly grafové struktury. Jejich atributy jsou tvořeny různými datovými typy. Mohou to být primitivní

datové typy, řetězce a nebo, a to je velice důležité, jiní potomci ADevice - tedy zařízení z jiných modulů. Taková reference na jiný modul pak tvoří systém mezimodulového provázání. Pro grafovou strukturu představují reference hrany grafu. Důležité je také zmínit, že reference, jejichž vkládání je řešeno pomocí metody typu setter, by neměla očekávat konkrétní třídu jiného Device. Očekávaná třída u takové metody by měla být interface (tedy rozhraní) vytvořené společně s každou novou komponentou. Tato podmínka, která ale není frameworkem vynucena, není vždy splněna. Fungujícím, nikoli ale zcela správným postupem, je ve zmiňovaných setterech přijímat abstraktní třídu ADevice.

5.1.1 Struktura konfiguračních souborů

Jedná se o soubory s příponou xrc, jenž uchovávají ve formě zápisu jazyka XML nezbytné nastavení jednotlivých komponent frameworku pro jeho spuštění. Cesta ke konfiguračnímu soubor je při spouštění frameworku zadávána přepínačem *-f*.

Pokud se podíváme na strukturu konfiguračního souboru, zjistíme, že obsahuje hlavičku s informacemi o verzi xml a použitém kódování, následně uvnitř párové značky robot a párové značky device seznam použitých komponent čtyř typů. Tyto typy korespondují s výše definovanými komponenty. Podrobněji k jejich párovým značkám:

- **aspect**: jedná se o nedílnou součást každého konfiguračního souboru a vyskytuje se vždy právě v jedné instanci. Jedinou jeho vlastností je reference na komponentu typu logger.
- **logger**: logger má v konfiguračním souboru totožné postavení jako aspect. Jeho jediná instance ale nemá vůbec žádnou značku property, věcí navíc je atribut path určující cestu pro ukládání logů.
- **device**: nejčastěji zastoupená značka reprezentující komponentu Device. Uvnitř této značky se nachází množina vlastností reflektující nastavitelné Java atributy odpovídající třídy - množina nepárových značek property, které mají vždy dvojici atributů, z nichž první name a druhá je jedna z dvojice - value nebo ref. Name označuje název určené proměnné, kterou device má a je odrazem reálného atributu ve třídě, kterou device představuje. Pokud je hodnota vlastnosti zařízení textového nebo jakéhokoliv číselného datového typu, tak je název atributu udržující hodnotu pod slovem value. Pokud tomu tak není, jedná se o odkaz na jiné zařízení, potom je název tohoto odkazovaného zařízení pod atributem s názvem ref. Součástí atributů je vždy reference na logger. Výskyt této závislosti odpovídá logice dědění tříd představující komponenty Device z rodičovské třídy ADvice.
- **view**: tyto značky jsou, vzhledem k povaze představované komponenty, částí konfiguračních souborů nepovinné. Některé konfigurační soubory je nemají vůbec, některé pro každý device. Obsahem view značky je, stejně jako u značky

device, vždy odkaz na logger. Další povinnou součástí je odkaz na navázaný Device. Mohou se zde objevovat ale i další reference.

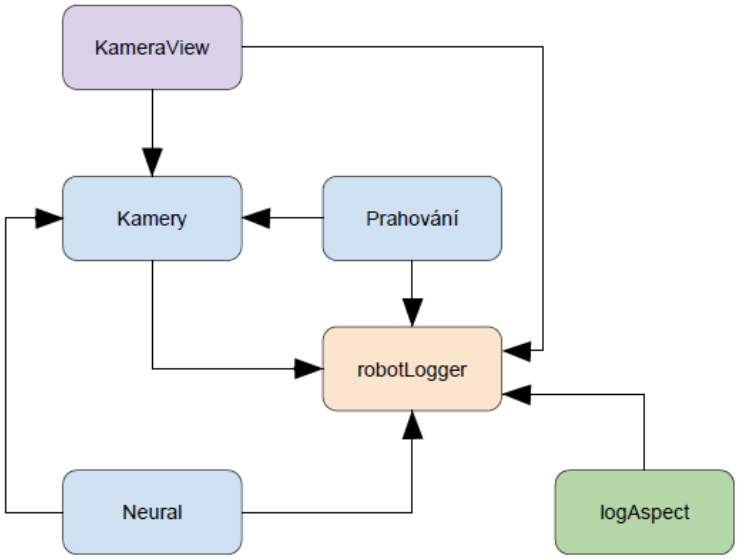
Zápis těchto značek vždy uchovává určitý stav konkrétní instance konkrétního typu komponenty. Tento stav je definovaný atributy, resp. jejich hodnotami.

Pro všechny zmiňované značky existují dva společné atributy. Jedná o *class* - tedy cestu k představované Java třídě v kanonickém tvaru a *name* - tedy jméno. Důležité je zmínit, že *name* představuje unikátní identifikátor v rámci konfiguračního souboru.

5.1.2 Příklad konfiguračního souboru a jeho grafické reprezentace

Pro názornost je na výpisu 1 uveden reprezentativní vzorek z množiny již existujících a používaných konfiguračních souborů, jež se distribuují společně s frameworkem.

Již takový jednoduchý příklad konfiguračního souboru je vhodný ke znázornění ve formě grafu tak, jak bylo již v cíli práce zamýšleno. Jedná se tedy o jednoduchý orientovaný graf, v němž tvoří komponenty uzly a závislosti hrany grafu. Vyobrazen je na obrázku 2.



Obrázek 2: Grafová reprezentace názorného konfiguračního souboru

Listing 1: Příklad konfiguračního souboru

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- framework bezici offline bez realneho robota -->
<robot>
  <devices>

    <aspect name="logAspect"
      class="cz.mendelu.aistorm.log.AspectLog">
      <property name="logger" ref="robotLogger" />
    </aspect>

    <logger
      name="robotLogger"
      class="cz.mendelu.aistorm.log.RobotLogger"
      path="C:\Bakalarka\logs\">
    </logger>

    <device
      name="Kamery"
      class="cz.mendelu.aistorm.device.KameryDummy">
      <property name="resource" value="C:\Bakalarka\Fotky" />
      <property name="leva" value="0" />
      <property name="formatIndex" value="4" />
      <property name="interval" value="100" />
      <property name="logger" ref="robotLogger" />
    </device>

    <view
      name="KameraView"
      class="cz.mendelu.aistorm.view.KameryView">
      <property name="device" ref="Kamery"/>
      <property name="logger" ref="robotLogger" />
    </view>

    <device
      name="Prahovani"
      class="cz.mendelu.aistorm.prahovani.PrahovaniDeviceRefactor">
      <property name="logger" ref="robotLogger" />
      <property name="heightOfLine" value="100" />
      <property name="heightFromLine" value="20" />
      <property name="kamery" ref="Kamery" />
      <property name="neighbors" value="5" />
    </device>

    <device
      name="Neural"
      class="cz.mendelu.aistorm.neural.NeuralDeviceRefactor">
      <property name="heightOfLine" value="80" />
      <property name="heightFromLine" value="34" />
      <property name="kamery" ref="Kamery" />
      <property name="Neighbors" value="1" />
      <property name="interval" value="1000" />
      <property name="logger" ref="robotLogger" />
      <property name="path" value="C:\Bakalarka\vzorky2\" />
      <property name="vahy" value="C:\Bakalarka\neuronkaVahy3.txt" />
    </device>

  </devices>
</robot>
```

5.2 Uživatelské požadavky na aplikaci

Tato sekce shrnuje požadavky na aplikaci z uživatelského hlediska. Požadavky byly diskutovány s tvůrcem i uživatelem robotického frameworku. Bodově byly vymezeny následující požadavky:

- Aplikace by měla dynamicky reagovat na změny ve frameworku. Není vhodné vytvořit jednorázový obraz frameworku, se kterým by aplikace dále pracovala. Samotný framework podstupuje vývoj a vzhledem k této skutečnosti je nejvhodnější vymyslet takové řešení, které zajistí, že se bude moci uživatel při práci s aplikací spolehnout na konzistenci mezi frameworkem a aplikací pro tvorbu konfiguračních souborů. Nežádoucí je také nutnost takovou konzistenci obnovovat ručně. Vhodné by také bylo řešení automatizované do takové míry, aby její zajištění pro uživatele nebylo obtížné nebo zdlouhavé.
- Požadována je vysoká míra intuitivnosti, jednoduchosti a flexibility při použití aplikace. Maximální využití výhod grafické reprezentace, jako je okamžité znázorňování správnosti prováděné akce například pomocí zvýrazňování, je cestou pro zmíněné požadované vlastnosti. Jako další možnosti, jak vytvořit větší uživatelskou přívětivost, je nadefinování klávesových zkratek nebo vytvoření a zabudování nápovědy k použití. Usnadnění práce s návrhem by mělo jít až do té míry, kdy budou uživateli nabízeny vhodné následující možnosti jako reakce na jeho provedené akce.
- Důležitým požadavkem je určitá míra kontroly akcí vykonávaných uživatelem. Aplikace by neměla umožnit nadefinovat a především vyexportovat návrh (resp. konfigurační soubor), jehož složení nedává v rámci politik definovaných frameworkem sémanticky smysl. Aplikace by, tedy nejlépe již při návrhu grafové struktury, měla uživateli nabízet takové možnosti, které problémovému stavu předchází.

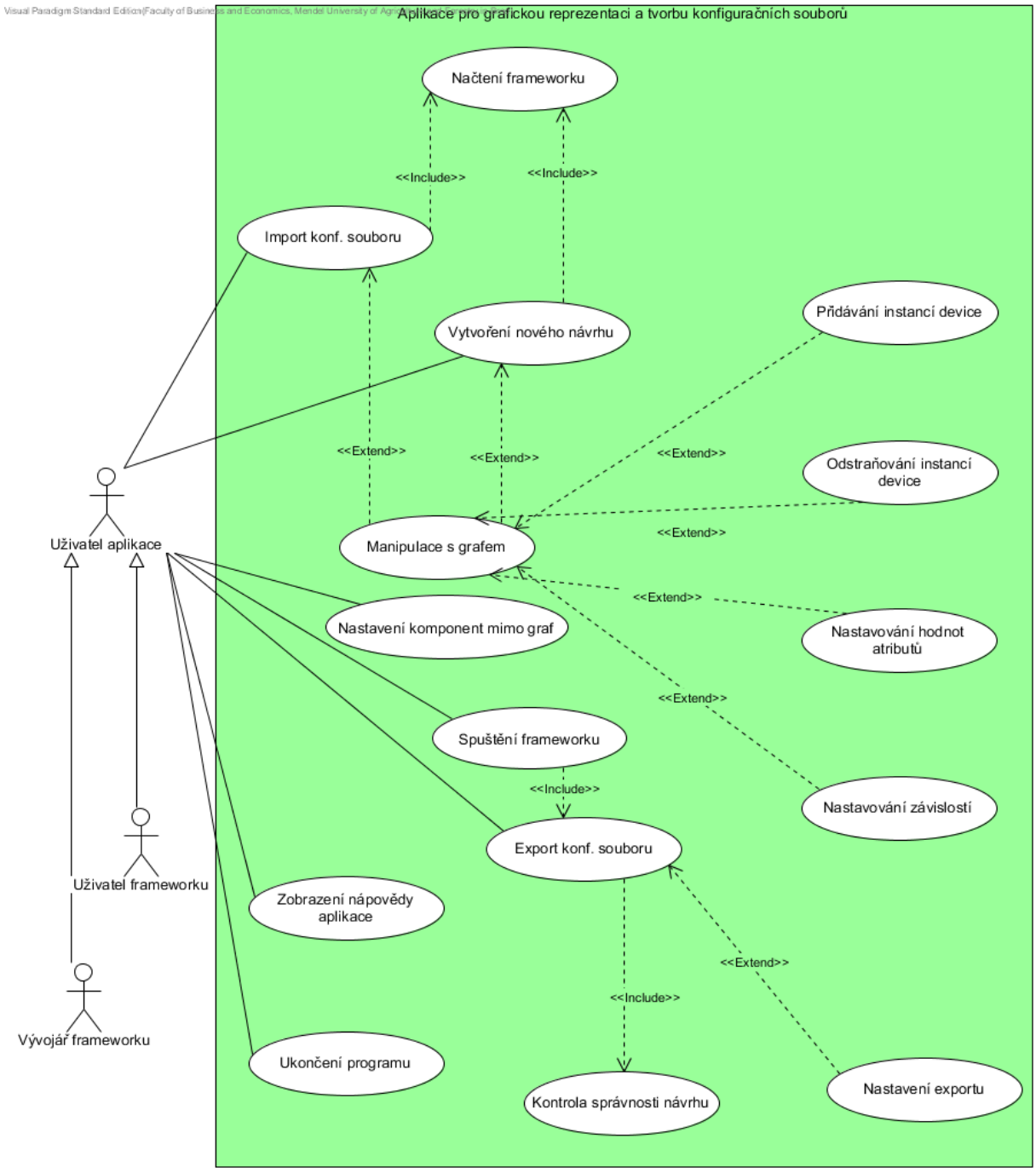
6 Návrh a implementace

Provedená analýza umožňuje přistoupit ke konkrétnímu návrhu a následné implementaci. Požadovanou funkci aplikace definovanou v cíli práce je nutno dekomponovat na dílčí složky. Jako nástroj pro znázornění tohoto procesu je velice vhodné UML modelování.

6.1 Diagram případů použití

Diagram případů použití nabízí velice dobré znázornění interakce uživatele s vytvářenou aplikací. Vytvořením tohoto diagramu se podařilo definovat právě zmiňované dílčí složky funkcionality aplikace. Tedy samotné poměrné vágní *vytváření návrhu* se zde tříští na již konkrétnější akce jako přidávání instancí, nastavování závislostí a další.

Aktorem se zde stává jak osoba, jenž pouze potřebuje nadefinovat či upravit konfigurační soubor pro spuštění frameworku, tak vývojář frameworku. Tomu může aplikace například nabídnout rychlý pohled na to, jak jím vyvíjený nový modul bude vypadat při konečné konfiguraci nebo jednoduše rychlé otestování, zda nové zařízení očekává správné atributy. Samotné případy použití se ale mezi těmito aktory nijak neliší, a tak je lze generalizovat jako prostého uživatele aplikace.



Obrázek 3: Use Case diagram

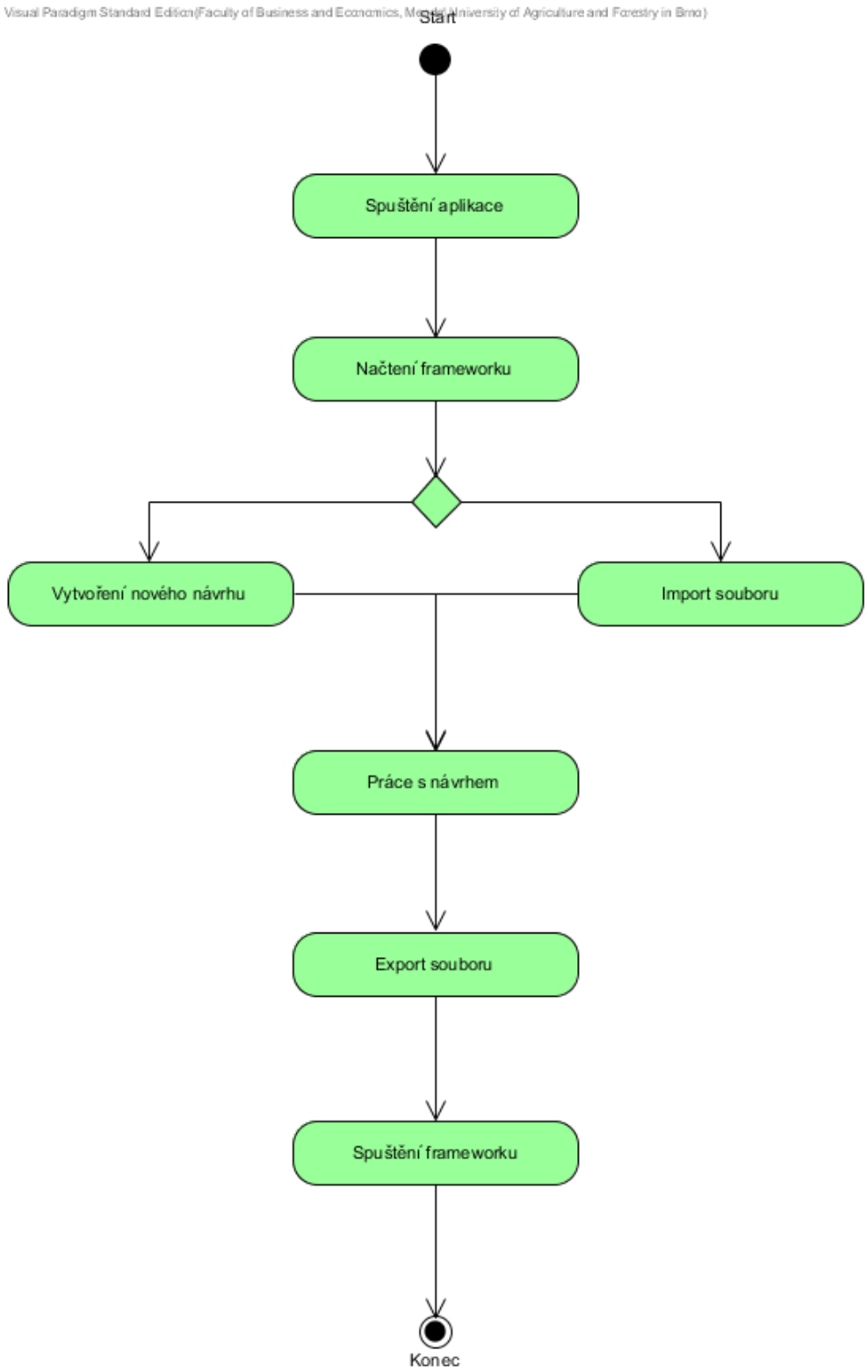
Jak je zřetelné z diagramu na obrázku 3, jednotlivé Use Case lze rozdělit do dvou kategorií. Jedná se o manipulaci s grafovou strukturou - tedy vytváření samotného obsahu plánovaného konfiguračního souboru. Druhou kategorií je pak práce s aplikací jako takovou. Tedy exportování, importovnání, zobrazení nápovědy k použití a

další. Takovéto rozdělení se již v této fázi návrhu dá předpokládat i na další aspekty vytvářené aplikace.

6.2 Diagram aktivit

Důležitou skutečností, která je na výše uvedeném diagramu případů použití (obrázek 3) znázorněna hned v jeho horní části, je nedílné zahrnutí načtení robotického frameworku k práci s aplikací. Touto problematikou se podrobněji zabývá sekce pojednávající o transformaci dat. V této části je pouze důležité si uvědomit, že načtení frameworku je nezbytnou prerekvizitou pro další funkce aplikace. Stává se tedy první aktivitou po spuštění aplikace. Až na ní navazuje další interakce uživatele se systémem. Postavení této akce je tedy takové, že by v pozdější implementaci bylo vhodné ji vyvolávat automaticky se spuštěním s možností provést načtení znovu již na popud uživatele.

Workflow aplikace zachycuje diagram aktivit na obrázku 4.



Obrázek 4: Diagram aktivit

Diagram zachycuje jednoduchý sled akcí, jak by mohly jít po sobě. Nejedná se o kompletní popis algoritmu - tedy nejsou zahrnuty možnosti vrácení se na předchozí kroky, které by měli být při implementaci zahrnuty. Konkrétně se jedná o opakované vytváření nového návrhu, importování souborů, exportování a další akce za jednoho běhu programu.

6.3 Transformace dat

Lze prohlásit, že v navrhované aplikaci je transformace dat tou naprosto nejpodstatnější částí. Právě tato záležitost řeší určité body napojení celé aplikace na již existující systém a to jak z pohledu vstupu tak i výstupu. Jedná se o esenciální část práce, která určuje fungování vytvářeného systému jako článek určitého *řetězce zpracování dat*.

Pokud podle výše vyobrazeného diagramu aktivit bude uvažován scénář, kdy uživatel importuje již nějaký existující konfigurační soubor, tak celkový počet transformací dat provedený aplikací je tři. Konkrétně jde o následující procesy.

- **Načtení frameworku**: transformace informací z robotického frameworku do vhodné reprezentace v aplikaci.
- **Import**: transformace konfiguračního souboru na bázi XML do vhodné reprezentace v aplikaci.
- **Export**: transformace návrhu v aplikaci do souboru na bázi XML.

Ze zmíněného diagramu je nutné pro problematiku transformace dat ještě podrobněji speficikovat akci *práce s návrhem*. Kromě vytváření grafové struktury a nastavování atributů lze jako vnořenou aktivitu této akce označit také možnost určitého nastavení hodnot, které mají při spuštění programu určité defaultní hodnoty (zapsané přímo v kódu), ale je možné je změnit. Pro tento účel aplikace operuje s objektem typu asociativního pole, který ukládá do souboru a při spuštění se ho snaží přečíst. Za určitou další doplňující transformaci dat lze považovat tedy **serializaci a deserializaci** tohoto objektu, který v kontextu aplikace představuje *Nastavení aplikace*.

6.3.1 Reflexe

Pomocí reflexe se řeší načtení frameworku. Prerekvizitu pro provedení této činnosti tvoří zkompilované všechny moduly robotického frameworku v lokálním repozitáři počítače, na kterém je v úmyslu aplikaci spouštět.

Konkrétní aspekty Java kódu frameworku, které jsou z pohledu aplikace pro tvorbu konfiguračních souborů významné, můžeme shrnout bodově.

- Abstraktní třída ADevice
- Potomci třídy ADevice

- Atributy takových potomků - resp. jejich metody typu setter

Reflexe umožňuje snadným způsobem třídu ADevice získat a následně i všechny její potomky v rámci zvoleného balíčku. Pro každého takového potomka lze pak iterovat jeho metody. Pokud se jedná o metody typu setter, dostáváme se tak k požadovaným atributům daných tříd. Zde se již předpokládá určité dodržování konvencí od vývojářů robotického frameworku. Vzhledem ke skutečnosti, že ve třídě procházené za pomocí reflexe se může objevit cokoliv, je nutné množinu atributů transformovaných do reprezentace pro aplikaci určitým způsobem omezit. Po analýze více vzorků konfiguračních souborů a konzultaci byly vymezeny následující podmínky, které musí atributy, resp. settery splňovat, aby se staly předmětem zpracování při procházení frameworku.

- Atribut musí mít veřejnou metodu typu setter, tedy z prefixem set
- Atribut musí mít datový typ z množiny datových typů určených pro konfigurovatelné atributy

Funkce procházející framework je v rámci celé aplikace důležitá, pseudokód je uveden na výpisu 2.

Jako vhodná reprezentace získaných informací bylo zvoleno asociativní pole, kde klíčem je nalezená třída, která je potomkem ADevice a tak představuje zařízení. Hodnotou v tomto poli je pak další asociativní pole, které pak pod klíčem - názvem atributu, uchovává datový typ atributu. Díky tomu, že pomocí reflexe se lze takto dostat ke třídám a uložit je jako objekty typu Class, je v další fázi aplikace možné s výhodou s těmito třídami pracovat. Příkladem je metoda isAssignableFrom nad Class objektem, díky které je možné zjistit, zda je určitá třída dosaditelná do jiné.

V prostředí programovacího jazyku Java je asociativní pole řešeno pomocí tříd implementující rozhraní Map z rodiny Collection. Konkrétně byla zvolena třída HashMap, tedy hashovací tabulka.

Výstupní strukturu transformace dat popisují následující zápis.

```
{Class, {String, Class}}

{Potomek ADevice, {Název atributu, Třída atributu}}
```

6.3.2 Import konfiguračních souborů

Způsobů, jakým se dá pomocí Javy přečíst XML soubor, je více. Přistupovat k takovému souboru jako z běžnému textovému souboru je jedna z možností, nicméně takový přístup by byl implementačně náročný. Mnohem vhodnější metodu představují hotová řešení pro orientaci v DOM struktuře XML souborů. Jedním takovým řešením se stává knihovna Jsoup popisovaná v kapitole použitých technologií.

Soubor načtený běžným způsobem v Javě, tedy do objektu File, Jsoup převádí do jeho vlastních objektových reprezentací Document a specifičtější Element.

Listing 2: Zjištění informací o frameworku - pseudokód

```
function projdi framework
   vytvor prazdnou strukturu pro konfiguraci frameworku
   ziskej tridy balicku cz.aistorm ktere jsou potomky tridy Adevice
   for trida
      pridej tridu ke strukture
      ziskej mnozinu vsech public setteru tridy
      for setter z mnoziny vsech setteru
         if setter nesplnuje pozadavky
            oznac setter k odebrani
         end
      end
      odeber z mnoziny vsech setteru vsechy oznacene settery
      for setter z mnoziny vyfiltrovanych setteru
         ziskej parametr setteru
         vloz nazev a tridu parametru ke strukture jako atribut iterovane tridy
      end
   end
   return konfigurace frameworku
end
```

Manipulace pak s prvky XML souboru představující komponenty probíhá velice intuitivním způsobem pomocí metod getElementsByTag. Stejně snadným způsobem jsou dostupné pak i atributy značek, resp. jejich hodnoty. Hodnoty ve stavu, v jakém jsou pomocí parsování dosažitelné, jsou datového typu String. Důležitou částí transformace dat do struktury, se kterou pracuje aplikace, je pozdější přetypování.

I v případě importování dat byl zvolen jako výstupní formát transformace asociativní pole typem hodnoty další asociativní pole. Takové rozvržení odpovídá charakteru zpracovávaných dat. Stejně jako u načítání frameworku se pojednává o načítání zařízení, kde v tomto případě z konfiguračního souboru namísto samotného objektu třídy je možné jako klíč uvést pouze kanonický název třídy. I takovou hodnotu však lze použít pro porovnávání s objekty třídy a určit tak tedy třídu jako objekt typu Class. V asociativním poli představující hodnotu pole prvního se pak stává klíčem název atributu dané třídy a hodnotou hodnota tohoto atributu.

Výstupní strukturu transformace dat popisují následující zápis.

```
{String, {String, String}}
```

```
{Kanonický název třídy, {Název atributu, Hodnota atributu}}
```

6.3.3 Export konfiguračních souborů

V okamžiku, kdy aplikace umí zpracovat informace z frameworku za pomocí reflexe a následně zpracovat konfigurační soubor, požadované transformace dat ve směru vstupu jsou vyřešeny. Výstupní tok dat představuje proces opačný k importu. Pro vytvoření XML souboru není knihovna Jsoup vhodná, a přestože existuje mnoho nástrojů usnadňující serializaci a obecně tvorbu inkriminovaných dokumentů, byl zvolen postup zcela nejjednoduššího principu. Jedná se o zápis do souboru bez použití externích knihoven na bázi výstupního proudu PrintStream.

Logickým předpokladem při implementaci exportu je dodržení takové struktury souborů, aby byly dále použitelné pro framework. Jednoduchým testem se stává kontrola exportovaného souboru oproti importovanému ve scénáři, kdy nejsou provedeny žádné změny charakteru editace návrhu.

Návrh aplikace ale v této záležitost naráží na následující skutečnost. Pro grafovou reprezentaci zařízení a závislostí je velmi vhodné, aby jednou vytvořené rozvržení v editoru bylo po opětovném importu znovu vyvoláno. Jde tedy o ukládání pozic uzlů grafu. Pokud by byl požadavek nezměnit strukturu konfiguračních souborů, nabízí se řešení vytvoření dalšího souboru, ve kterém by byla informace o pozicích uložena. Takový soubor by se ale musel distribuovat společně s XML a po zvážení byl tento způsob označen jako nevhodný. Diskuzí s autorem robotického frameworku se došlo k racionálnějšímu řešení, a to ukládání informace přímo do konfiguračního souboru. Pro pozdější zpracování frameworkem to netvoří žádný problém, přesto je nutné ve frameworku učinit jisté kroky, které s novou značkou počítají. Ukázka je na výpisu 3.

Listing 3: Příklad exportovaného device se značkou pozice

```
<device name="AkcniZasah_instanceb2c0" class="cz.mendelu.aistorm.regulator.AkcniZasah">
   <position x="244.11624541040425" y="342.46139596768523">
   <property name="logger" ref="robotLogger1">
</device>
```

Tato záležitost se stává tedy prvním bodem celé práce, která zasahuje do systému, ke kterému se aplikace připojuje. Přestože takovéto kroky nebyly v původním konceptu zcela zamýšleny, ukazuje se, že určité záležitosti jsou sice bez modifikace propojených systémů řešitelné, ale malé změny mohou v konečném důsledku vést k mnohem větší efektivnosti řešení celého řetězce zpracování dat.

Při procesu exportu se objevuje ještě další problém. Výsledkem získání informací o zařízeních z frameworku je množina konkrétních zařízení a ke každému z nich množina atributů. Přestože výběr atributů je zatížen určitým filtrováním, srovnáním s atributy u konkrétních zařízení z exemplárních konfiguračních souborů se zjistilo, že těch nastavovaných konfigurací je pouze podmnožina. Objevují se také rozdíly mezi množinou nastavených atributů u stejného zařízení v rámci odlišných XML souborů. Vhodným řešením této záležitosti se tak stává scénář, kdy jsou uživateli aplikace k nastavení hodnoty nabídnuty všechny možné atributy a následně jen ty, ke kterým přiřadil hodnotu (a tím se myslí i nastavení reference), se stávají předmětem exportu.

Shrnutí předmětů a zásad exportu konfiguračních souborů.

- Předmětem každého exportu je Logger a volitelně Aspect, všechny jejich atributy jsou v aplikaci nastavitelné.
- Stěžejní částí je zápis zařízeních typu Device, zápis atributů je řízen nastavením jejich hodnot. Jedná se o zapsaní grafové struktury.
- Identifikátorem komponent je jméno - name. Pokud nejsou jména unikátní, export neprobíhá.
- Zařízení typu View a případně další nejsou v této fázi vývoje uvažována.

6.4 Grafický návrh

Aplikace vytvářená v rámci této práce nezahrnuje pouze vyjádření zmiňované grafické reprezentace konfiguračních souborů. Neméně důležitou částí je prostředí, ve kterém se manipulovatelná grafová struktura má nacházet. Jedná se tedy o desktopovou aplikaci, u které je zamýšleno plnit funkci nástroje pro průchod dat s možností editace. U takového editoru se stává samozřejmostí, kromě určitého způsobu použití (možnost opakovaného exportování a importování bez nutnosti restartu aplikace), také rozložení ovládacích prvků.

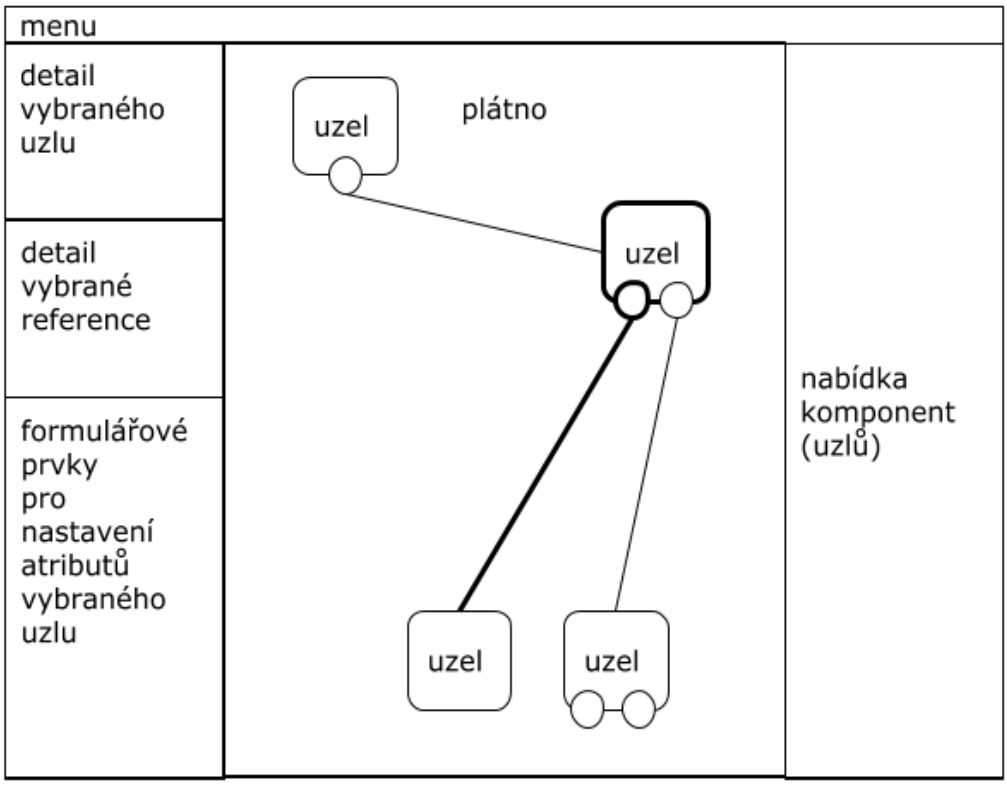
Vytvoření grafického návrhu aplikace se stává z několika na sebe navazujících kroků.

6.4.1 Drátěný model

Základem pro další návrh je vytvoření drátěného modelu aplikace. Jedná se o jednoduchý model navrhovaného rozložení bez použití konkrétní technologie, jehož úkolem je rozložení obsahu a funkcionalit. Sekundárním přínosem může být určitá pomoc při definování funkcí aplikace. Při vytváření drátěného modelu je možné si díky vizuálnímu projevu navrhované aplikace uvědomit absenci funkcionalit, které nemusely být při předchozích fázích návrhu zřejmé. Drátěný model aplikace představuje obrázek 5.

Z drátěného modelu vyplývá, že složení grafického prostředí aplikace se skládá z následujících částí:

- **Menu lišta** nabízející základní operace s programem - vytvoření nového návrhu, exportování a importování souboru, nastavení programu a další.
- **Levý panel** jehož úkolem je zobrazení informací k aktuálně vybranému prvku. Tento panel se skládá z tří podčástí.
  - **Detail vybraného uzlu** sloužící k zobrazení základních informací o uzlu (komponentě), které mají všechny uzly společné.
  - **Detail vybrané reference** nabízí informace o vybrané referenci (tedy odkazu resp. závislosti) na vybraném uzlu.
  - **Formulářové prvky pro atributy** je část se vstupními poli pro všechny prvky jiné než reference.
- **Plátno** je místem pro vykreslování grafu a manipulace s ním. Výběr prvku zde koresponduje s nabídkou levého panelu.
- **Pravý panel** je nabídkou komponent, které je odtud možné přidat na plátno nebo ji odtud nakonfigurovat. Záleží na charakteru komponenty.



Obrázek 5: Drátěný model aplikace

Drátěný model vyobrazuje hlavní scénu, resp. okno aplikace, ve kterém probíhá většina interakce uživatele s nástrojem. Uživatel se může setkat v průběhu práce s dalšími dvěma okny. Jedná se o scénu s možností změny nastavení určitých proměnných, do které se lze dostat přes navigaci z lišty menu. Druhým typem oken jsou informativní okna nesoucí určitou zprávu (chybového nebo pouze informativního charakteru), které se zobrazují v závislosti na aktuálně prováděných akcí. Struktura obou zmíněných oken je velmi jednoduchá a nevzniká tedy pro ně potřeba vytvoření drátěných modelů.

6.4.2 FXML prvky

Krokem grafického návrhu navazující na vytvoření vyhovujícího drátěného modelu je tvorba konkrétních prvků grafického uživatelského rozhraní. Nástrojem k tomu jsou dokumenty FXML. Potřeba vytvořit samostatné FXML dokumenty vznikla pro následující prvky:

- Okno editoru
- Okno nastavení
- Uzel grafu

- Reprezentace uzlu grafu v pravém panelu
- Reference (odkaz na jiné zařízení)

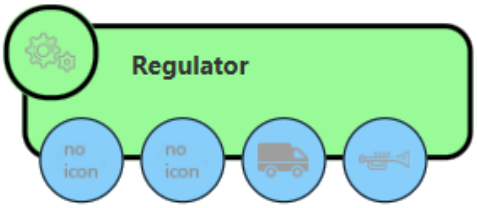
Pro zbylé prvky vytvářející vzhled aplikace, jako jsou informativní okna, hrany grafu (vystupující z prvku reference) nebo vstupní formulářové prvky pro atributy, nevznikla potřeba tvorby FXML dokumentů. Důvodem je jejich snadná tvorba a manipulace přímo z programového kódu a absence nutnosti složitějšího provazování akcí reagující na grafické prvky.

V souladu se vzneseným požadavkem na aplikaci týkající se intuitivnosti a přehlednosti při práci s návrhem byla do vytváření FXML prvků zahrnuta identifikace pomocí ikony. Každému zařízení tedy může být přidělena ikona, která je pak použita v aplikaci hned na několika místech. Nejdůležitějšími místy je ale právě její zobrazení na grafických prvcích pro uzel grafu, reprezentaci uzlu grafu v pravém panelu a referenci. Přítomnost ikony pro zařízení není nutná. Pokud chybí, je na její místo dosazena defaultní ikona.

6.4.3 Grafová struktura

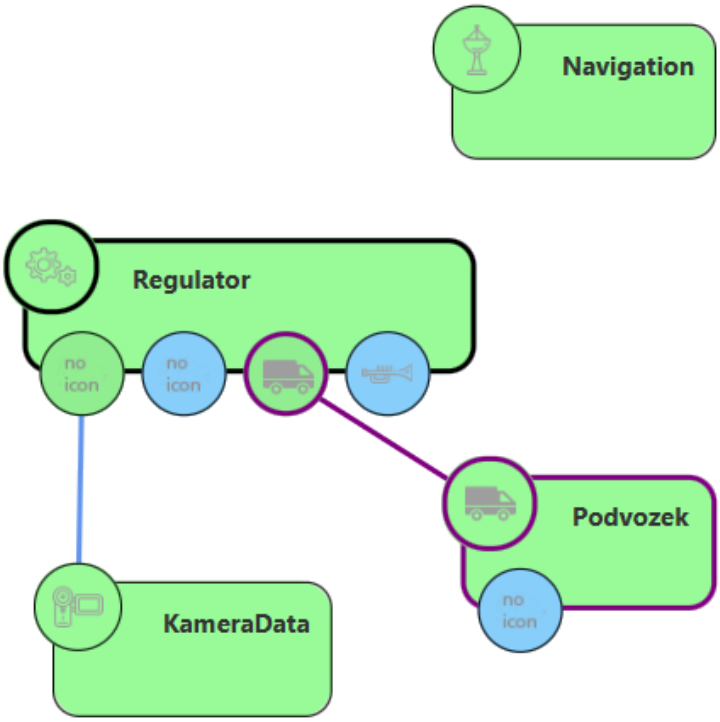
Při pohledu na grafovou reprezentaci konfiguračního souboru na obrázku 2 lze rozeznat jako uzly tři druhy komponent, které jsou odlišeny barvami. Vzhledem k povaze komponent Logger a Aspect bylo rozhodnuto, že v samotné grafové struktuře, se kterou má pracovat uživatel aplikace, tyto komponenty nemají být. Zatímco na Logger se totiž obecně každá instance jiného druhu komponenty odkazuje vždy, na Aspect naopak není odkazováno vůbec. Zařízení typu View se pro současný stav vývoje aplikace neuvažuje, jsou nepovinné a případné vhodné řešení by implikovalo k vytvoření samostatné vrstvy plátna vyhrazené pouze pro ně.

Důsledkem tohoto rozboru se stává skutečnost, že uzly, se kterými aplikace operuje, jsou pouze pro komponenty typu Device. Pro identifikaci uzlu slouží jméno (které je unikátní identifikátor) a ikona zařízení. Hranami grafu se z grafického hlediska zamýšlí určité spojnice mezi uzly. Pro přesnější rozlišení více závislostí stejného uzlu byl navrhnut systém vykreslování samostatných grafických prvků pro reference do těla uzlu. Více pochopitelná je tato skutečnost z obrázku 6. Ten představuje již konkrétní uzel grafu (resp. instance třídy Regulator - jedna z komponent typu Device). V levém horním rohu je vlastní ikona. V dolní části je pak vykreslena množina závislostí, která toto zařízení může mít. Jsou to prvky pro reference, které jsou graficky reprezentovány ikonou požadovaného zařízení k dosazení.



Obrázek 6: Uzel grafu

Hrana grafu vychází sice z uzlu, ale specifičtěji přímo z prvku reference a končí v odkazovaném uzlu. Názorné je propojení dvou uzlů na obrázku 7. Z tohoto obrázku jsou viditelné ještě další důležité charakteristiky grafu. Uzel Regulator je ve stavu *vybraný (focused)* a stejně tak jeho reference na Podvozek je v tomto stavu. Tento výběr má za následek viditelné zvýraznění vybraného uzlu, hrany a také odkazovaného uzlu. Díky tomuto zvýrazňování je možná rychlá orientace i ve složitějších strukturách, kdy se lze těžko vyhnout překrývání prvků. Důsledkem *výběru* uzlu a reference je také vyplnění levého panelu.

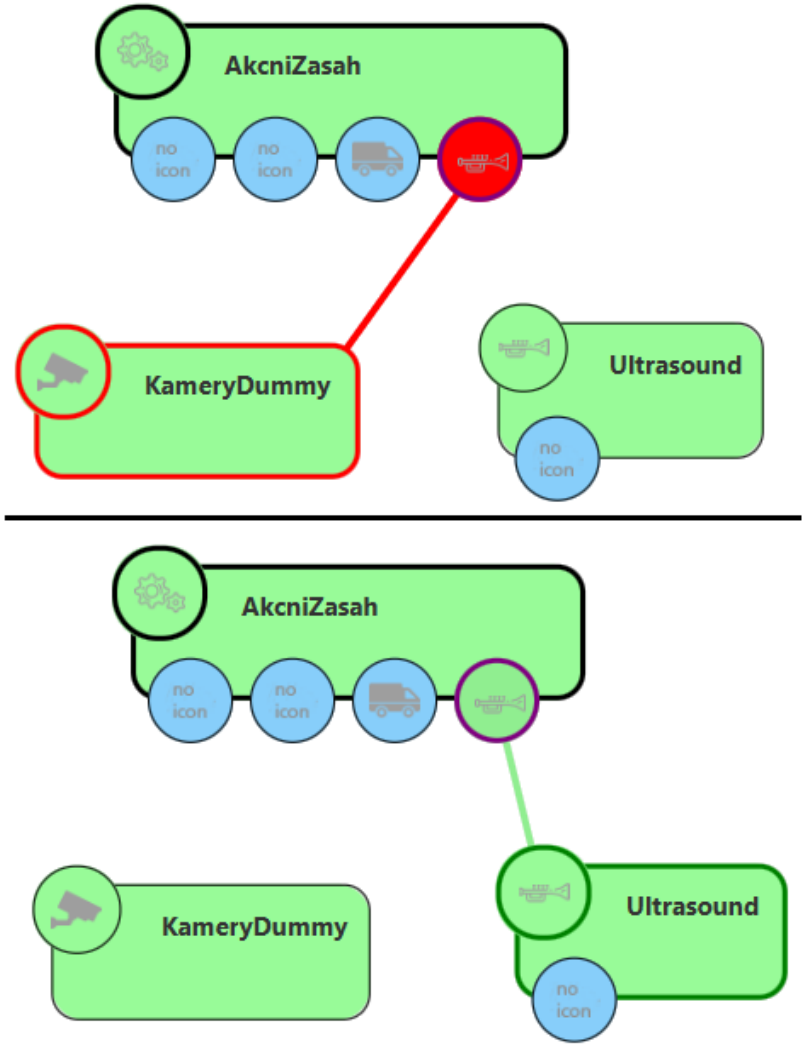


Obrázek 7: Propojení uzlů

Manipulace s uzly grafu probíhá pomocí Drag And Drop operací. Uchycení lze provést za jakoukoliv část těla uzlu kromě prvků pro reference. Drag And Drop akcí, která je započata kliknutím na referenci, je navazování vztahu mezi uzly, resp vytváření hrany. S tím úzce souvisí vizuální indikátory reprezentující stav reference (tedy navázáno / nenavázáno, lze vidět na obrázku 7) a také při samotném navazování také vizuální indikace správnosti chystaného provázání uzlů. Ukázka je na obrázku 8. Zatímco v situaci v horní polovině obrázku lze vidět, že reference nesedí, tak v situaci pod černou čarou je navazovaná reference správná - tedy Java třída reprezentující zařízení Ultrasound je dosaditelná pod třídu požadovanou referencí. V obou případech se nejedná o konečný stav navazování. Drag And Drop operace nebyla ukončena a jedná se tak o okamžik, kdy uživatel spojnici započatou v prvku reference protáhl nad cílový uzel a je mu vizuálně indikována správnost volby. Dosaditelnost je implementačně řešena způsobem podle ukázky kódu na výpisu 4.

Listing 4: Podmínka řešící dosaditelnost odkazovaného typu

```
if (reference.getReferencedDeviceClass().isAssignableFrom(node.getDeviceClass())) { ... }
```



Obrázek 8: Indikace správnosti propojování uzlů

6.4.4 Komunikace s uživatelem

Uživatelsky vstřícnou a příjemnou aplikaci vytváří především přehlednost a správná komunikace s uživatelem. Uživatele je potřeba o dění programu neustále informovat.

Pro informování o chybách, událostech nebo i pro výpis vyjímky slouží informační dialogové okno *(pop-up)*, které mění svůj vzhled a rozložení podle typu

informace. Uživatel se s tímto oknem může setkat, pokud se například pokusí vyexportovat do konfiguračního souboru návrh, jehož sémantická stránka není správná.

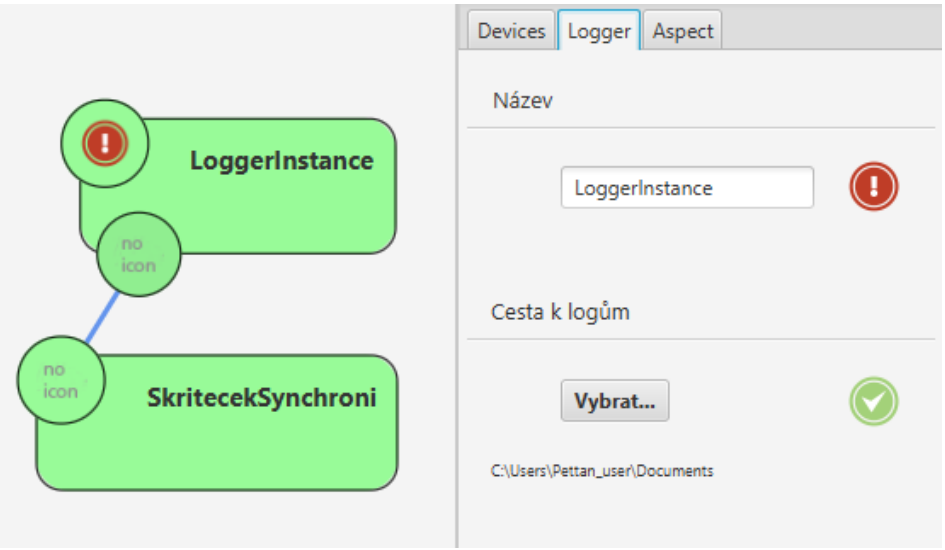
Příklad informačního a chybového hlášení:

Import souboru autonomousConfig.xml byl úspěšný: Bylo nalezeno a vykresleno 10 komponent Device.

Chyba při vytváření uzlu z konfiguračního souboru: V konfiguračním souboru se nachází třída cz.mendelu.aistorm.manualControl.Gamepad, která nebyla vyparsována z frameworku AIstorm.

Dalším implementovaným prvkem pro zvýšení komfortnosti použití aplikace je zavedení několika základních klávesových zkratek. Akce, které jsou jejich prostřednictvím proveditelné, je samozřejmě možné provést i bez použití klávesnice. Příkladem je akce *vymazání vybraného prvku* pomocí klávesy Delete nebo použití klávesy Escape pro akci *zrušit výběr prvku*.

O správnosti vytvářeného návrhu aplikace informuje ještě před kontrolou, která probíhá při exportu a jejím výsledkem jsou informativní dialogová okna. Průběžná kontrola a její vyhodnocení při práci je řešena pomocí umístěných grafických prvků. Typickým příkladem může být situace, kdy je zadáno pro více než jednu komponentu stejné jméno. To je v rámci konfiguračního souboru unikátní identifikátor a taková situace se tedy stává chybnou. U komponent, kterých se tato situace týká je aktivován indikátor. Příklad vyobrazuje obrázek 9, kde byl zadán stejný název pro Logger a pro jeden z instancí Device.



Obrázek 9: Indikátory chybového stavu

6.5 Spouštění frameworku

Jedním z požadavků, který je spojen s požadavkem na snadné použití aplikace, se stala možnost spuštění robotického frameworku Aistorm přímo z aplikace pro práci s konfiguračními soubory.

Pro spuštění frameworku je nutné zavolat třídu *cz.mendelu.aistorm.ui.Aistorm* s parametrem určující adresářovou cestu k vytvoření logů a parametrem představující cestu ke konfiguračnímu souboru.

Spuštění frameworku z grafického uživatelského prostředí vytvářené aplikace zahrnuje kontrolu a dočasné uložení rozpracovaného návrhu a předání požadovaných cest volané třídě pro spuštění. Framework se poté nad oknem editoru spustí a uvede do provozu.

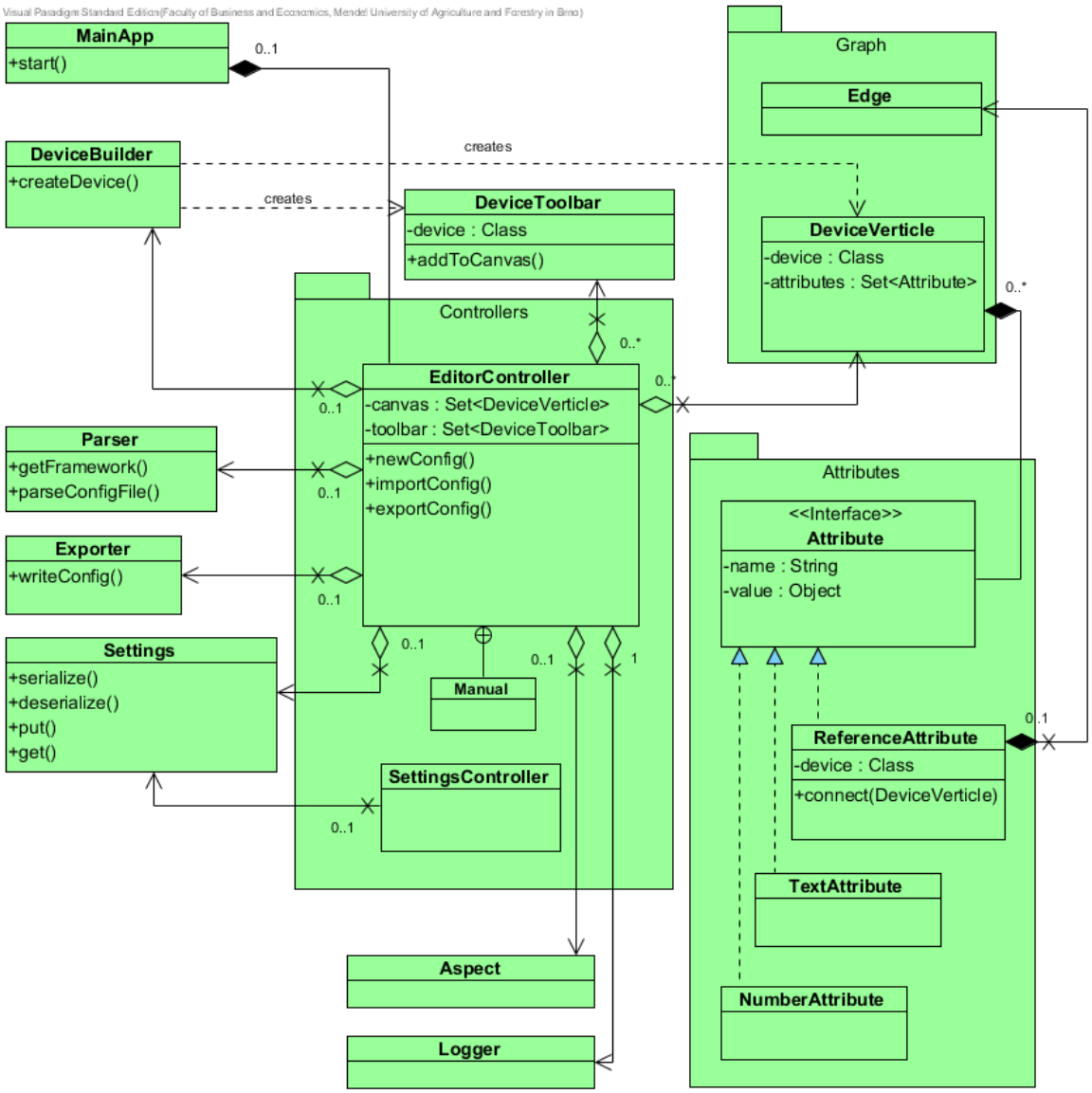
6.6 Diagram tříd

Jako shrnutí výše popsaných entit a procesů a pro možnost vytvoření uceleného pohledu na architekturu aplikace je na obrázku 10 vyobrazen diagram tříd. Jedná se o zjednodušený diagram tříd plnící účel vytvoření představy o komunikaci mezi objekty. Nejedná se o plný diagram a je vynechána většina atributů i metod jednotlivých tříd, které vznikly v průběhu implementace. Jejich množství vzhledem k impementaci související s technologií JavaFX je značné - zahrnuje množství definovaných formulářových prvků jako atributů a metod zvládající *eventhandlery*.

Návrhový vzor *Builder* je zde použit u tříd týkajících se vytváření komponent Device. Postavení *Director* tohoto návrhového vzoru zde zastává třída EditorController. Třída DeviceBuilder je *Builder* vytvářející produkty - DeviceVerticle a DeviceToolbar. Zatímco první jmenová třída z dvojice představující určitou komponentu typu Device je reprezentace uzlu grafu, tak DeviceToolbar je položka nabídky v pravém postranním panelu.

Vytvořené rozhraní Attribute má v návrhu tři konkrétní implemetující třídy, z nichž každá představuje určitý druh atributu. Číselné datové typy byly shrnuty pod jednu číselnou reprezentaci NumberAttribute, která udržuje další logiku rozdělení číselných datových typů v sobě. Aplikace je v případě dalšího rozšiřování připravena na přidání nových datových typů parametrů, stačí s novou třídou implementovat rozhraní Attribute.

Vnořená třída Manual představuje jednoduchý webový prohlížeč pro účel zobrazení manuálu aplikace, který je pro snažší editaci vytvořen a uložen v aplikaci jako soubor html.



Obrázek 10: Zjednodušený diagram tříd

7 Testování

Pro testování aplikace byl zvolen přístup automatického testování a to konkrétně na stupni jednotkových testů. Použitými nástroji se staly frameworky určené pro testování JUnit a Mockito. Nebyl zvolen přístup *test-drive-development*, tedy testy řízeného vývoje, a testy tak nebyly vytvářeny před samotnou implementací. Testovací jednotky byly tvořeny po vytvoření určitého celku implementace typicky třídy.

Vytvořená sada automatických testů nepokrývá zcela všechny části implementace a soustředí se na programátorem vyhodnocené kritické části kódu a to takové, která udržuje programovou logiku. Důvodem je především využití technologie grafického uživatelského prostředí JavaFX, která vzhledem ke svému charakteru v některých testovacích scénářích vytváří obtížně řešitelné situace - jedná se o testování prezentační části programu. Velice dobrým prostředkem pro jejich částečné zvládnutí se stalo *mockování* pomocí frameworku Mockito.

Kromě vytváření jednotkových testů se stalo prostředkem testování použití konfiguračních souborů dodané s frameworkem. Při vývoji se tak stalo rutinní záležitostí importování těchto souborů a při exportování ověření správnosti jejich struktury. Tento druh testování se řadí do testovacího stupně *Testování programátorem*.

7.1 Příklad testovací třídy

Jako příklad je na výpisu 5 v části práce Přílohy uveden testovací kód pro třídu *Exporter*. Pro objekty, které by jinak bylo nutné složitě vytvářet, protože jsou produktem spuštění aplikace uživatelem a jeho tvorby návrhu, jsou zde jako mock objekty. Přesto bylo potřeba pomocí konstrukce *when* nasimulovat k těmto objektům ještě určité návratové hodnoty. Pro jedinou metodu, kterou tento objekt má, jsou vytvořeny tři testovací případy.

8 Závěr a diskuze

Cílem této bakalářské práce bylo vytvořit desktopovou aplikaci, která by umožnila vytvářet konfigurační soubory pomocí intuitivního a moderního grafického uživatelského prostředí. Definované požadavky by se daly shrnout jako pružné napojení na existující robotický framework, systém kontroly vytvářeného návrhu a jednoduchost použití.

Přestože stanovený cíl byl vytyčen dostatečně jasně, během implementace se z několika různých důvodů bylo nutno zamyslet nad stavem vývoje a určité postupy změnit. Největší problém tvořilo napojení na existujicí robotický framework. V této záležitosti se vyskytlo více problémů a to především se čtením poměrně pokročilé struktury frameworku. Záchranou se stalo použití knihovny Reflections. Určité problémy při implementaci nastaly při práci s JavaFX technologií. Jedná se ve srovnání s jinými GUI frameworky o poměrně mladou technologii, což s sebou přineslo i určité problémy. Komunita kolem této technologie není příliš veliká (důvodem ale může být i celkový úpadek vývoje desktopových aplikací) a řešení některých situací se tak stalo záležitostí obtížnějšího hledání, než by programátor předpokládal.

V průběhu implementace také došlo ke změně vývojového prostředí z důvodu nepohodlné práce s některými knihovnami a problémů se závislostmi týkající se spolupráce technologií Maven a JavaFX. Přechod proběhl bez větších problémů z Netbeans do prostředí IntelliJ IDEA.

Výsledná podoba aplikace naplňuje očekávání a výsledek se stává výrazným usnadněním pro vytváření konfiguračních souborů, jak bylo zamýšleno. Přesto výraz *výsledná* nelze použít zcela v pravém slova smyslu, protože ač nyní aplikace plní svůj předpokládaný úkol, již v této chvíli lze určit další vývoj.

8.1 Možná rozšíření

Pro aplikaci je zamýšlen další vývoj a rozšiřování hned v několika směrech. Jedná se o rozšíření z pohledu uživatelského hlediska, kde by bylo vhodné dodat některé prvky funkcionality známé z mnoha editorů i jiného typu, jako jsou například textové procesory nebo nástroje pro práci s grafikou. Konkrétně by mohl být vytvořen systém pro navigaci v provedených akcích uživatelem s možností vrácení provedených kroků. Dalšími prvky zpříjemňující práci uživatele by mohl být systém zoomování a automatického inteligentního rozvrhování uzlů grafu řešící některé neduhy s překrýváním vykreslených prvků. S překrýváním také souvisí možné rozšíření vykreslování hran grafů, které by mohlo být estetičtěji řešeno pomocí pravoúhlých lomených čar nepřerušující čáry jiné.

Další možné rozšíření se pak nachází na poli provázání s robotickým frameworkem. V implementované verzi aplikace pro tvorbu konfiguračních souborů se nepracuje s volitelnými zařízeními typu View. Jejich začlenění do editoru by vedlo pravděpodobně k vytvoření sekundárního plátna, které by bylo částečně propojené s plátnem primárním. Vytvářena by tak vzhledem k povaze komponent View byla další

grafová struktura. Současně s vývojem této aplikace probíhá i vývoj frameworku a vznikají tak další typy komponent na stejné úrovni jako ty, se kterými pracuje aplikace v současném stavu. I takové komponenty by se dle požadavků robotického týmu Aistorm mohly stát předmětem návrhu ve vytvořené aplikaci.

9 Reference

ARLOW, J. NEUSTADT I. *UML 2 a unifikovaný proces vývoje aplikací.* Brno: Computer Press, 2007. 565 s. ISBN 978-80-251-1503-9.

BLOCH, J. *JAVA efektivně: 57 zásad softwarového experta.* Praha: Grada Publishing, 2002. 230 s. ISBN 80-247-0416-1.

BOROVCOVÁ, A. *Testování webových aplikací.* Diplomová práce. Univerzita Karlova v Praze, Matematicko-fyzikální fakulta, 140 s..

BREIDENBACH, R. C. *Spring framework.* Greenwich: Manning Publications, 2010. 472 s. ISBN 1-933-98813-4.

BROWN, Alan W. *Principles of CASE tool integration.* New York: Oxford University Press, 1994. 270 s. ISBN 0195094786.

FOLTÝNEK, T. *Teoretické základy informatiky (sbírka úloh pro cvičení).* Brno: KONVOJ, 2011. 70 s. ISBN 978-80-7302-147-4.

FOWLER, M. *Destilované UML.* Praha: Grada, 2009. Knihovna programátora (Grada). 173 s. ISBN 978-80-247-2062-3.

HOLUBOVÁ, I. POKORNÝ J. *XML technologie: principy a aplikace v praxi* 1. vyd. Praha: Grada, 2008. 267 s. ISBN 978-80-247-2725-7.

Ivo, A. *Editor elektronických schémat s rozhraním v QT.* Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií 32 s..

JANDÁK, T. *Možnosti automatizovaného testování aplikací.* Bakalářská práce. Západočeská univerzita v Plzni, Fakulta elektrotechnická 37 s..

KOSKELA, Lasse. *Effective unit testing: a guide for Java developers.* Manning Publications Co. 2013. 224 s. ISBN 1935182579.

KREJČÍ, V. *Adobe Photoshop: design grafiky GUI.* 1. vyd. Praha: Grada, 2008. 199 s. ISBN 978-80-247-2011-1.

MACHÁČEK, M. *Vývoj aplikací na platformě JavaFX 8.* Bakalářská práce. Vysoká škola ekonomická v Praze, Fakulta informatiky a statistiky. 2015. 77 s.

MARGAI, L. *Využití návrhového vzoru Mode-View-Controller ve webových aplikacích.* Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií 46 s..

MARINILLI, M. *Professional Java User Interfaces.* New York, USA: John Wiley Sons Inc., 2006. 656 s. ISBN 0-471-48696-5.

ONDROUŠEK, V. *Mobilní robot Buggyman.* [online]. [cit. 2015-15-10]. Dostupné z. https://aistorm.mendelu.cz/cz/projekty/buggyman.

ORACLE CORPORATION *JavaFX 8 Javadoc.* [online]. [cit. 2016-16-2]. Dostupné z. https://docs.oracle.com/javase/8/javafx/api/.

OSTŘÍŽEK, F. *Návrh a implementace řídicího software mobilního robotu.* Diplomová práce. MENDELU Brno, 2015. 55 s.

PAGE-JONES, M. *Základy objektově orientovaného návrhu v UML.* Praha: Grada, 2001. Moderní programování. 386 s. ISBN 80-247-0210-X.

PECINOVSKÝ, R. *OOP: naučte se myslet a programovat objektově.* Brno: Computer Press, 2010. 576 s. ISBN 978-80-251-2126-9.

PECINOVSKÝ, R. *Návrhové vzory: 33 vzorových postupů pro objektové programování.* Brno: Computer Press, 2013. 526 s. ISBN 98025115824.

PINKAS, J. *Informace: Apache Maven.* [online].[cit. 2015-15-10] dostupné z http://www.java-skoleni.cz/info/maven.php.

SCHILDT, H. *Java 7. Výukový kurz.* Brno: Computer Press, 2012. 664 s. ISBN 978-80-251-3748-2.

ŠEDA, M. *Teorie grafů.* Skripta pro předmět Teorie grafů. Vysoké učení technické v Brně, Fakulta strojního inženýrství 89 s..

VOS, J. *JavaFX 8: a definitive guide to building desktop, mobile, and embedded Java clients* . Expert's voice in Java. 2014 616 s. ISBN 978-1-4302-6574-0.

10 Přílohy

10.1 CD se zdrojovými kódy

Součástí této práce je CD s IntelliJ IDEA projektem aplikace. V projektu se nachází několik exemplárních konfiguračních souborů.

10.2 Ukázky kódu

Listing 5: Příklad testovací třídy - ExporterTest.java

```
@RunWith(MockitoJUnitRunner.class)
public class ExporterTest {
    private Exporter exporter;
    @Mock
    private Settings settings;
    @Mock
    private CLogger logger;
    @Mock
    private CAspect aspect;
    @Mock
    private DeviceVerticle deviceVerticle;
    private List<DeviceVerticle> devices;
    private File file;
    /**
     * Vytvoreni prostredi pro test - simulace navrhu pomoci mockovanych objektu
     */
    @Before
    public void setUp() throws Exception {
        devices = new ArrayList<DeviceVerticle>() {{add(deviceVerticle);}};
        file = new File("testOutput.xml");
        exporter = new Exporter(settings,logger,aspect,devices);

        when(logger.getPath()).thenReturn(new File("/"));
        when(deviceVerticle.getXmlDescriptor()).thenReturn("name=\"null\" class=\"null\"");
    }
    /**
     * Test zda metoda vyhodi ocekavanou exception pri predani null parametru.
     */
    @Test(expected=FileNotFoundException.class)
    public void writeConfigFileNotFoundException() throws Exception {
        exporter.writeConfig(null);
    }
    /**
     * Test zda metoda skutecne vytvori soubor.
     */
    @Test
    public void writeConfigFileCreated() throws Exception {
        exporter.writeConfig(file);
        File file = new File("testOutput.xml");
        assertTrue(file.exists());
    }
    /**
     * Otestuje zda vytvoreny xml soubor obsahuje pouze povolene znacky a atributy.
         Otestuje take syntaxi xml.
     */
    @Test
    public void writeConfigFileValid() throws Exception {
        exporter.writeConfig(file);
        String xmlBody = new String(Files.readAllBytes(Paths.get("testOutput.xml")),
            StandardCharsets.UTF_8);
        Whitelist wl = new Whitelist();
        wl.addTags("robot","devices","logger","aspect","device","property","position");
        wl.addAttributes("aspect","name","class");
        wl.addAttributes("property","name","ref");
        wl.addAttributes("logger","name","class", "path");
        wl.addAttributes("device","name","class");
        wl.addAttributes("position","x","y");
        assertTrue(Jsoup.isValid(xmlBody,wl));
    }
}
```