VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

MODELEM ŘÍZENÝ VÝVOJ ANDROID APLIKACÍ

MODEL DRIVEN ANDROID APPLICATION DEVELOPMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. STANISLAV BĚLEHRÁDEK**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/20221/2016/xbeleh04

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Bělehrádek Stanislav, Bc.**
Obor: Informační systémy
Téma: **Modelem řízený vývoj Android aplikací**
Model Driven Development of Android Applications
Kategorie: Informační systémy

Pokyny:

1. Seznamte se s problematikou modelem řízeného vývoje (Model Driven Development, MDD) a s vývojem aplikací pro platformu Android. Prozkoumejte jazyky a nástroje pro podporu MDD, konkrétně se zaměřte na Executable UML.
2. Navrhněte způsob uplatnění MDD při vývoji Android aplikací. Navrhněte nový, či upravte stávající nástroj pro MDD tak, aby umožnil vývoj Android aplikací.
3. Po konzultaci s vedoucím navržený nástroj či jeho změnu implementujte. S využitím nástroje navrhněte a implementujte ukázkovou Android aplikaci. Výsledky zveřejněte jako open-source pod svobodnou licencí.
4. Proveďte zhodnocení dosažených výsledků a diskutujte další možný vývoj projektu.

Literatura:

- Olivier Le Goaer, Franck Barbier, Eric Cariou, Samson Pierre. Android Executable Modeling: Beyond Android Programming. In Proceedings of the 2014 International Conference on Future Internet of Things and Cloud (FICLOUD '14). IEEE Computer Society, Washington, DC, USA. [http://olegoaer.perso.univ-pau.fr/works/MobiApps2014.pdf]
- Chris Raistrick, Paul Francis, John Wright, Colin Carter, Ian Wilkie. Model Driven Architecture with Executable UML. Cambridge University Press New York, 2004. ISBN 978-0521537711
- Stephen J. Mellor, Marc J. Balcer. Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley Professional, 2002. ISBN 978-0201748048 [http://www.executableumlbook.com/]
- Ľuboslav Lacko. Vývoj aplikací pro Android. Computer Press, Brno, 2015. ISBN 978-80-251-4347-6

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2, započaté řešení bodu 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Rychlý Marek, RNDr., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato diplomová práce se věnuje návrhu a implementaci nástroje pro tvorbu Android aplikací, který je založený na modelem řízeném vývoji software (Model Driven Software Development). Nejprve je popsán obecně vývoj softwaru, potom konkrétně se zaměřením na MDD a executable UML. V další části je představena platforma Android, způsoby tvorby aplikací na tuto platformu a existující nástroje s podporou MDD. Následně je ukázán návrh nového MDD nástroje pro tvorbu Android aplikací. Navrhovaný nástroj je realizován jako Gradle plugin a samostatné vývojové prostředí využívající tento plugin. Nástroj k modelování aplikací využívá fUML a jazyk ALF. Funkce a možnosti vyvíjeného nástroje jsou demonstrovány při tvorbě vzorové aplikace.

Abstract

This thesis deals with the design and implementation of Android application development tool based on model driven software development. The first part of the thesis is focused on general software development and next part on software development based on model driven development and executable UML. In next part Android platform, methods of Android application development and existing MDD tools are described. This thesis continues with the design of my own MDD tool for the creation of Android applications. The designed tool is realized like Gradle plugin and independent development environment using this plugin. The designed tool is based on fUML and ALF language. The features and options of development tool are demonstrated by creation of example application.

Klíčová slova

fUML, ALF, Android, vývoj software, modelem řízený vývoj, generování kódu, Gradle

Keywords

fUML, ALF, Android, software development, model driven development, code generation, Gradle

Citace

BĚLEHRÁDEK, Stanislav. *Modelem řízený vývoj Android aplikací.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Modelem řízený vývoj Android aplikací

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doktora Marka Rychlého. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Stanislav Bělehrádek
24. května 2017

Poděkování

Chci poděkovat vedoucímu mé diplomové práce panu doktorovi Marku Rychlému za metodické vedení a rady, které mi při tvorbě této práce dal.

Obsah

Kapitola 1

Úvod

Software a počítače jsou téměř všude, najdeme je například v mobilních telefonech, pračkách i letadlech. Veliký rozmach nastal v oblasti internetu věcí (Internet of thinks), který vnáší počítače do spousty zařízení a připojuje je k internetu. Na vývoj software jsou kladeny čím dál větší nároky. Velikost a komplexnost systémů neustále roste včetně požadavků na kvalitu, spolehlivost i bezpečnost. Vývoj takových systémů musí být precizní a musí být řízen přesně definovanou metodikou. Nesmí být podceněna fáze analýzy a návrhu systému.

Právě ve fázi analýzy a návrhu ukázalo svou důležitou roli modelování. Kvůli složitosti systémů není možné systém zachytit celý detailně a přitom přehledně, proto jsou k popisu systému používány modely jako abstrakce reality. Takové modely zachycují to, co chceme zdůraznit bez zbytečných detailů. Je nutné si také uvědomit, že forma modelu může být různá, může to být například papírový model budovy, počítačový 3D model turbíny, UML diagram nebo třeba zdrojový kód.

Zpočátku byly tyto modely používány jako vzor při implementaci. Programátorovi mohl být předán model k implementaci a ten podle něj vytvářel zdrojový kód. Později byly z těchto modelů vytvářeny části zdrojových kódů automaticky. Byly vytvářeny například kostry tříd bez implementace metod. Chybějící kód, který nebyl vygenerován automaticky, musel být doplněn ručně. Poté nastaly snahy o generování celého kódu z modelů, což by vedlo k výraznému urychlení vývoje aplikací.

Moderní mobilní telefony jsou spíše než telefony počítače, co umějí mimo jiné telefonovat. Mobilní aplikace jsou nedílnou součástí dnešního života. Existují aplikace téměř na vše, na sportování, poslech hudby, čtení knih, hraní her a spoustu dalšího. Možnosti vývojářů při vývoji aplikací se rozšiřují díky rychle se zlepšujícímu hardware mobilních telefonů a zvyšující se kvalitě API mobilních operačních systémů. Jsou k dispozici skvělá vývojová prostředí a spousta kvalitních komponent a frameworků, to vše usnadňuje a urychluje vývoj.

MDD vývoj se v programování mobilních aplikací zatím výrazněji neprosadil. Dle mého názoru je to způsobeno nedostatečnou podporou tohoto paradigmatu ve vývojových prostředích.

Proto jsem se rozhodl vytvořit nástroj pro vývoj Android aplikací podle principů MDD, speciálně podle fUML (Semantics of a Foundational Subset for Executable UML Models) a ALF (Action Language For Foundational UML). V tomto nástroji bude možné aplikace modelovat pomocí UML, modely spouštět přímo ve vývojovém prostředí i generovat kód Android aplikací. Nástroj bude implementován jako Gradle plugin a bude jej tak možné použít z příkazového řádku. Pro pohodlnější používání nástroje bude vytvořeno i vývojové prostředí používající vytvořený Gradle plugin. Modely budou popisovány textově v jazyce ALF, ale bude možné zobrazit i jejich grafickou UML reprezentaci. Nástroj bude primárně

určen na generování kódu pro Android, ale bude umožňovat vytvořit generátory kódu i na jiné platformy.

Kapitola Softwarové inženýrství 2 popisuje historii a současnost vývoje software.

V kapitole Modelování softwarových systémů 3 se věnuji modelovacím technikám v oblasti vývoje software.

Kapitola Platforma Android 4 je věnována popisu struktury a jednotlivých částí této platformy včetně nastínění problematiky vývoje aplikací na tuto platformu.

Kapitola Existující nástroje s podporou MDD 5 obsahuje popis a zhodnocení několika vybraných nástrojů s funkcemi pro vývoj aplikací pomocí MDD.

V kapitole Návrh vlastního MDD nástroje pro vývoj Android aplikací 6 je popsán návrh nového nástroje.

Kapitola Implementace vlastního MDD nástroje 7 popisuje implementaci všech částí navrženého nástroje.

V kapitole Tvorba vzorové Android aplikace 8 je popsán postup implementace vzorové aplikace, v postupu jsou prezentovány klíčové funkce nového nástroje.

V kapitole Zhodnocení vytvořeného nástroje a možnosti dalšího vývoje 9 je zhodnocen implementovaný nástroj a jsou nastíněny možnosti dalšího vývoje nástroje.

Teoretická část diplomové práce a návrh vlastního nástroje byl řešen již v rámci semestrálního projektu. Tyto kapitoly jsou v diplomové práci rozšířeny.

Kapitola 2

Softwarové inženýrství

Tato kapitola se věnuje softwarovému inženýrství. Podle definice je softwarové inženýrství systematický přístup k vývoji, nasazení a údržbě softwaru [5].

Důvodem vzniku softwarového inženýrství byl růst vyvíjených systémů a zvyšující se nároky na kvalitu, spolehlivost a bezpečnost systémů. Pomocí tehdejších technik vývoje se cílů nedařilo dosáhnout.

Softwarové inženýrství zahrnuje několik oblastí jako je management, analýza, návrh, implementace, testování a údržba softwarových projektů.

2.1 Historie softwarového inženýrství

Pojem softwarové inženýrství vznikl v šedesátých letech během softwarové krize, tehdejší techniky vývoje software nebyly vhodné pro vývoj rozsáhlejších systémů. To vedlo ke zhoršování kvality, prodražení a náročnější údržbě výsledného softwaru [5].

V 70. letech vznikaly modulární programovací jazyky jako je například Pascal nebo jazyk C. Byl zkoumán životní cyklus vývoje softwaru a tzv. dobré praktiky (Good practises), což je soubor pravidel, který se při vývoji SW osvědčil. Začalo se používat systematické testování a strukturovaná analýza a návrh.

V 80. letech vznikala první vývojová prostředí (Integrated development Environment IDE) a CASE (Computer Aided Software Engineering) nástroje. Byly vyvinuty první funkcionální, logické, objektově orientované a paralelní programovací jazyky. Také vznikly první verzovací systémy pro správu verzí zdrojových kódů.

V 90. letech se začaly používat prototypy SW a svou roli začala hrát znovupoužitelnost a komponenty. Vznikl objektový návrh a návrhové vzory.

Později bylo vyvinuto UML (Unified Modeling Language) jako grafický jazyk pro vizualizaci softwarových systémů, UML je velice oblíbené a hojně používané. Vznikly formální metody pro simulaci, analýzu a verifikaci softwarových systémů.

2.2 Metodiky vývoje software

Během historie softwarového inženýrství vzniklo několik různých metodik vývoje softwaru. Metodika je definovaný proces pokrývající celý životní cyklus vytvářeného software. Metodika slouží k dosažení určitého cíle a obsahuje fáze, aktivity, role, artefakty a milníky [5].

Životní cyklus vývoje softwaru se skládá z etap, každá etapa definuje své vstupy, výstupy a činnosti, které v ní probíhají. Typické etapy vývoje software jsou analýza a specifikace požadavků, architektonický návrh, podrobný návrh, implementace a testování součástí, integrace a testování systému, akceptační testování a instalace, provoz a údržba.

V této podkapitole budou nejprve popsány fáze životního cyklu, poté některé vybrané modely životního cyklu software a nakonec vybrané metodiky používané při vývoji software.

Etapy vývoje software

Vývoj software se typicky skládá z několika etap. V každé etapě vývoje je řešen nějaký klíčový problém.

První etapou je analýza a specifikace požadavků. Cílem analýzy a specifikace požadavků je získat, analyzovat a definovat požadavky na vyvíjený software. Požadavky jsou získávány od zákazníka. Požadavky je nutné formálně sepsat. Získat relevantní požadavky bývá někdy problém, protože někdy zákazník v této fázi vývoje ještě nemá přesnou představu, jaké funkce systému bude potřebovat. V této etapě se vůbec neuvažuje nad tím, jakým způsobem budou požadavky realizovány. Součástí této etapy je studie vhodnosti, při které se určí, zda má smysl projekt vůbec realizovat. Během této etapy by měla proběhnout i analýza rizik a měl by vzniknout plán akceptačního testování. Požadavky se dělí na funkční, to jsou požadované funkce systému, a nefunkční, to jsou například požadavky na výkon apod.

Etapa návrhu navazuje na analýzu a specifikaci požadavků. Tato etapa se dělí na fázi architektonického návrhu a podrobného návrhu. Během architektonického návrhu dochází k ujasnění koncepce systému a k dekompozici, vymezení funkcionalit jednotlivých subsystémů a definování vztahů mezi nimi. Během fáze architektonického návrhu by měly vzniknout plány testování celého systému a integračního testování. Podrobný návrh se věnuje specifikaci všech subsystémů navržených v předchozí fázi. Dochází k výběru datových struktur, algoritmů a návrhových vzorů realizujících požadované funkce. Také by měl vzniknout návrh testování jednotlivých součástí tzv. jednotkové testy včetně testovacích dat.

Implementace a testování se také může dělit na dvě části. První část je implementace a testování součástí, druhá část je integrace a testování celého systému. Implementace a testování součástí zahrnuje samotnou implementaci subsystémů, vytvoření jejich dokumentace a otestování součástí pomocí jednotkových testů. Po implementaci a otestování součástí je nutné součásti spojit do jednoho celku. Po propojení nastává testování systému jako celku a nalezené chyby jsou opravovány.

Během etapy instalace, provoz a údržba probíhá akceptační testování, při kterém je rozhodnuto, zda byly splněny požadavky na systém. Pokud je zákazník spokojen s dodaným systémem, dochází k nasazení systému u zákazníka, může také nastat proškolování uživatelů systému.

Poté nastává nejdelší část životního cyklu softwaru, a to je provoz a údržba. To zahrnuje průběžné řešení problémů, které při používání systému nastanou.

Modely životního cyklu vývoje software

Modelem životního cyklu vývoje software je například vodopádový model, jedná se o základní model, je také nejstarší. Ve vodopádovém modelu jsou jednotlivé etapy seřazeny za sebou a následující etapa začíná až po skončení předchozí etapy. Hlavním problémem tohoto modelu je, že zákazník dostane spustitelnou verzi SW až během závěrečných fází vývoje. A pokud zákazník nebyl schopen přesně specifikovat požadavky během první fáze,

bude nemožné dodatečné požadavky zapracovat. Jediná možnost je vrátit se do první fáze vývoje.

Nedostatky vodopádového modelu řeší iterativní model. Vývoj je rozdělen do iterací, každá iterace obsahuje vodopádový model. Takže po každé iteraci má zákazník k dispozici spustitelnou verzi softwaru. Doplněné požadavky jsou zpracovávány v následující iteraci, to však může vést k horší struktuře systému.

Po iterativním modelu vznikl model inkrementální, který je jeho modifikací. V tomto modelu jsou na počátku definovány klíčové požadavky a vývoj je rozdělen do několika částí, které lze vyvíjet odděleně. Každá část je doručena klientovi v okamžiku dokončení. Při použití inkrementálního modelu, bývá dosaženo lepší struktury systému než u iterativního modelu, protože ke klíčovému určení struktury systému dochází na začátku a ta by již neměla být zásadně upravována.

Ve spirálovém modelu hrají klíčovou roli prototypy softwaru. Prototyp se od verze SW s omezenou funkcionalitou liší tím, že je po použití zahozen a v další fázi se vytváří znovu. Je zde kladen velký důraz na analýzu rizik. Jednotlivé etapy se opakují podobně jako u iterativního nebo inkrementálního modelu ale na vyšším stupni zvládnuté problematiky.

Agilní metodiky

Hlavní hodnoty agilních metodik je individualita, doručování funkčního software, spolupráce se zákazníkem, reakce na změny. Hodnoty, principy a praktiky agilních metod jsou popsány v agilním manifestu [1].

Nejvyšší prioritou je uspokojit zákazníka pomocí brzkého a průběžného doručování funkčního software. Metodika vítá změnu požadavků, dokonce i v pozdějších fázích vývoje. Vývojáři i vedoucí zaměstnanci musejí spolupracovat na projektu denně. Dokumentace nemá primární význam. Projekty jsou stavěny kolem motivovaných individualit, které by měly být podporovány a měly mít důvěru. Nejefektivnější výměna informací je tváří v tvář [1].

Metodiky Unified Process (UP)

Unified Process (UP) je generická metodika vývoje software, před použitím musí být nejprve adaptována na konkrétní organizaci a projekt. Musí být definovány firemní standardy, šablony dokumentů, používané nástroje atd. UP je řízeno požadavky a riziky, důraz je kladen na kvalitní a robustní architekturu systému. UP je iterativní a inkrementální. V UP je životní cyklus SW rozdělen na čtyři fáze a to zahájení (inception), příprava (elaboration), konstrukce (construction) a předávání (transition). Každá fáze může být realizována několika iteracemi.

Rational Unified Process (RUP) je nejznámější a dobře dokumentovanou verzí UP od společnosti Rational Software, která je divizí IBM.

[1] Agilní manifest: http://agilemanifesto.org/

Kapitola 3

Modelování softwarových systémů

V této kapitole je popsán význam modelování při vývoji a metody vývoje využívající modelování. Modely jsou obecně abstrakcí reality, umožňují nám zachytit požadované vlastnosti bez nepodstatných detailů. Usnadňují nám pochopení komplexních systémů. Se vzrůstající složitostí a rozsahem systémů, není možné popsat detailně, a přitom přehledně celý systém.

Modely mohou mít různou formu, může se jednat například o matematický model popsaný rovnicemi, model letadla z papíru, 3D počítačový model nebo třeba UML diagram. Všechny tyto modely jsou nějakou formou abstrakce reality, která ukazuje jen ty vlastnosti, které chceme.

Jednu entitu můžeme popsat několika modely a na každém modelovat jiné vlastnosti, to z pravidla vede ke snazšímu pochopení modelů. Například v UML můžeme popsat statickou strukturu systému diagramem tříd a dynamiku systému diagramem aktivit, díky tomu nemícháme různé vlastnosti systému do jednoho modelu a každý z modelů je přehledný.

V různých metodikách vývoje software je modelování používáno různě. Existuje několik způsobů využití modelů při vývoji.

- **Model as a sketch up:** Model má význam pouze předlohy pro implementaci. Z modelu není nic generováno.
- **Model as blueprint:** Zde je implementace částečně generována z modelu, ne však kompletně a je třeba práce programátora k doplnění implementace.
- **Model as programming language:** Model zde má největší význam, po vymodelování systému máme k dispozici funkční systém, který je možné spustit. Není nutná žádná další implementace.

Hlavní využití modelování je při analýze a specifikaci požadavků. V těchto fázích je nutné zachytit požadované funkce a vlastnosti vytvářeného systému.

V některých metodikách hraje model významnější roli. Například v MDD model neslouží pouze jako předloha k implementaci, ale částečně plní i roli implementace a je hlavním artefaktem při vývoji.

3.1 Unified Modeling Language

Nejpoužívanějším jazykem pro grafické modelování softwarových systémů je UML (Unified Modeling Language). UML je grafický jazyk, který poskytuje nástroj systémovým architektům, softwarovým inženýrům a vývojářům pro analýzu, návrh a implementaci softwarových

systémů i modelování byznys procesů [14]. Ke grafickému zobrazení jazyk UML využívá diagramy. UML diagramy jsou popsány v následujících podkapitolách. UML podporuje objektově orientovaný přístup k analýze a návrhu softwaru. UML je standardizované organizací Object Management Group (OMG).

UML bylo vytvořeno za účelem abstraktního znázornění vlastností a chování systému, nikoli za účelem detailního a úplného popisu. UML obsahuje určité neúplnosti a nejednoznačnosti, proto není UML přímo vhodné pro úplný popis systémů a generování kódu.

UML modely jsou ve specifikaci definovány pomocí metamodelů. Metamodely přesně definují strukturu modelů.

Diagramy v UML

Cílem této podkapitoly je pouze nastínit diagramy UML, nikoliv detailně popsat konkrétní diagramy. Přesný popis diagramů je možné nalézt v mnoha knihách nebo například ve specifikaci UML viz [14].

UML diagramy se skládají z entit a vazeb mezi nimi. Různé diagramy používají různé entity s různou grafickou reprezentací a významem.

UML obsahuje řadu diagramů, ty je možné rozdělit do třech kategorií podle toho, jaké vlastnosti systému modelují.

Jsou to strukturální diagramy, které modelují strukturu systému neměnící se v čase. Do této skupiny patří například diagram tříd, objektů, nasazení, komponent nebo balíčků.

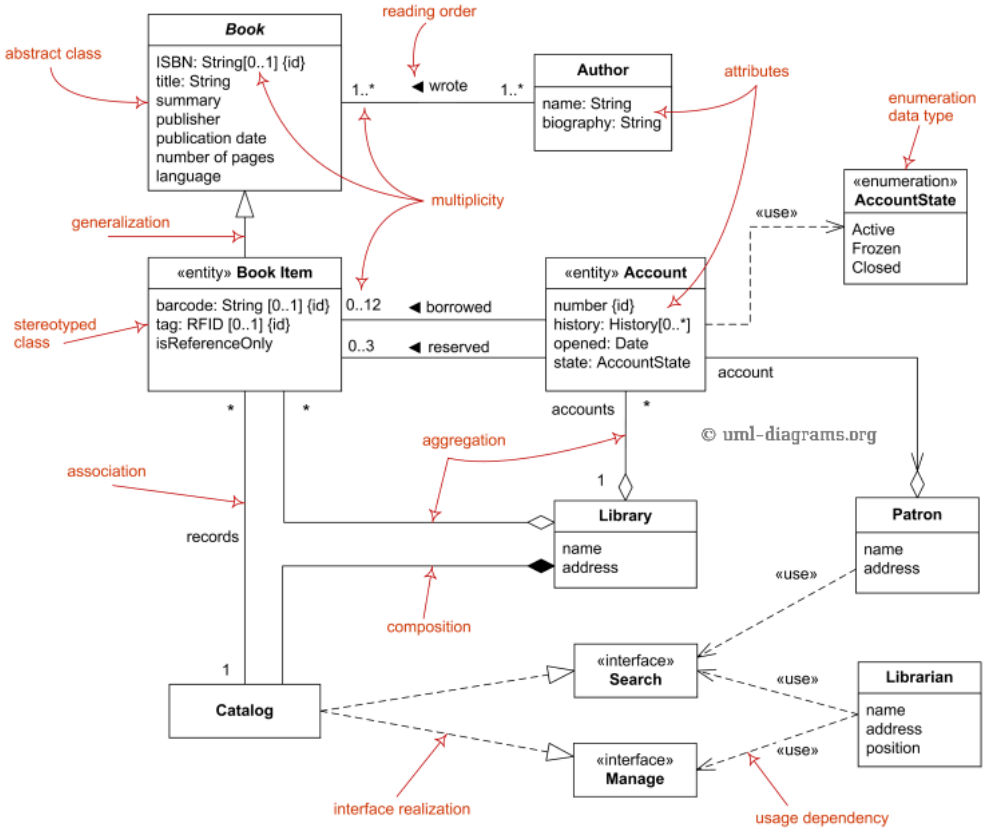
Dále to jsou diagramy chování, ty modelují dynamické chování systémů. Do této skupiny patří například diagram případů užití, diagram aktivit nebo stavový diagram.

Diagramy interakce jsou podskupinou diagramů chování a kladou větší důraz na spolupráci mezi entitami. Diagramy interakce jsou například diagram komunikace, časování, interakce nebo sekvenční diagram.

V následujících dvou odstavcích jsou popsány dva UML diagramy relevantní k této práci. Diagram tříd je používán v navrhovaném nástroji pro modelování statické struktury a diagram aktivit k modelování dynamického chování systému.

Diagram tříd Diagram tříd slouží pro modelování tříd, rozhraní a jejich závislostí. Příklad diagramu tříd je na obrázku 3.1. Rozhraní je modelováno stejně jako třída ale navíc obsahuje stereotyp `«interface»`. Třídy obsahují vlastnosti, konkrétně to jsou atributy a operace. Atribut má dané jméno, viditelnost, datový typ, násobnost a volitelně omezení. Datový typ může být primitivní typ, rozhraní nebo třída. Z hlediska viditelnosti může být atribut veřejný, chráněný, soukromý nebo viditelný v rámci balíčku. Násobnost atributu určuje, kolik instancí daného typu může atribut obsahovat, násobnost má spodní a horní hranici. Hranice může být nula, kladné celé číslo nebo neomezená. Operace mají jméno, viditelnost, libovolný počet parametrů a mohou mít návratový typ. Parametr se skládá ze jména a typu.

Základním vztahem tříd je asociace, ta říká že asociované třídy se mezi sebou nějakým způsobem využívají, asociaci je možné pojmenovat. Asociace může být jednosměrná, obousměrná nebo nespecifikovaná, to udává, ze které třídy je asociace dostupná. Pojmenované mohou být i oba konce asociace. Asociace může obsahovat na obou koncích definici násobnost. Asociace může být klasicky binární, taková propojuje dvě entity. Vícenásobné asociace propojují větší počet uzlů.



Obrázek 3.1: Příklad diagramu tříd s popisem (Převzato z http://www.uml-diagrams.org/class-diagrams-overview.html)

Speciálním typem asociace je agregace a kompozice. Agregace je slabší formou oproti Kompozici. Například pokud je zdrojová třída kompozice smazána, je mazána i cílová třída kompozice, u agregace toto neplatí.

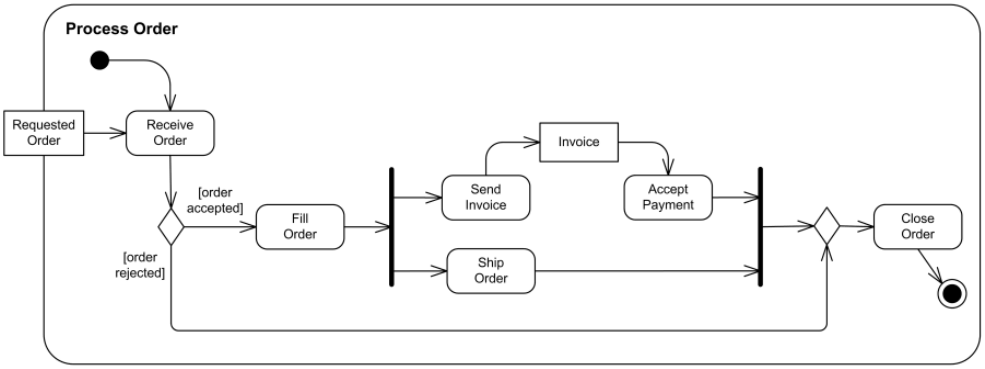
Dalším vztahem mezi třídami je generalizace, ta definuje dědičnost v diagramu tříd. Třída může mít libovolný počet potomků i rodičů. Stejně jako generalizace je modelována i realizace rozhraní.

Vazby mezi třídami obsahují násobnost, ta říká, kolik objektů dané třídy do vazby vstupuje. Násobnost může být definována přímo číslem, hvězdičkou udávající libovolný počet nebo rozsahem hodnot. Rozsah hodnot může být například 1..*, což udává násobnost větší nebo rovno jedné.

Diagram aktivit Diagram aktivit je UML diagram modelující chování. Diagram aktivit znázorňuje tok řízení (control flow) a tok objektů nebo dat (object flow). Uzly používané v diagramu aktivit jsou aktivity, akce, objekty a řídící uzly. Příklad diagramu aktivit je na obrázku 3.2.

Aktivita definuje parametrizované chování. Aktivita obsahuje vstupní a výstupní parametry a tok akcí, který definují samotné chování. Aby mohla být aktivita nebo akce vykonána musí na jejich vstupech být řídící i datové toky. To neplatí u uzlu spojovacího alternativy, tam postačí tok z jedné alternativy.

Akce definuje jeden atomický krok v aktivitě a není ji možné dělit.



Obrázek 3.2: Příklad diagramu aktivit (Převzato z http://www.uml-diagrams.org/shopping-process-order-uml-activity-diagram-example.html)

Existují akce pracující s objekty, konkrétně to je vytvoření objektu, zrušení objektu, test identity, načtení aktuálního objektu.

Pro práci s proměnnými existují akce pro jejich čtení a zápis. Pokud je proměnná pole, je možné použít akce pro přidání a vyjmutí prvku. Stejné akce existují pro práci s položkami struktury nebo třídy. Existují také akce pro spouštění jiných aktivit a chování. Objekty je možné propojovat asociacemi, které je možné akcemi přidávat, odstraňovat a číst. Diagram aktivit také poskytuje akce pro zasílání a přijímání signálů a událostí.

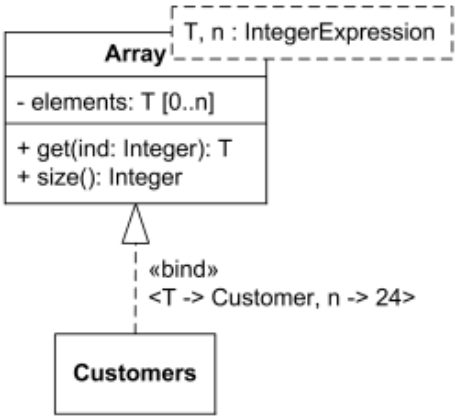
Pro řízení toku mezi uzly jsou použity uzly řízení. Začátek toku definuje inicializační uzel (initial node). Inicializačních uzlů může být i více, potom tok začíná v několika uzlech současně. Inicializační uzel v aktivitě být nemusí. Ukončovací uzel toku (flow final node) ukončuje konkrétní tok. Ukončovací uzel aktivity (activity fianl node) ukončí všechny toky v aktivitě. Tok je možné rozvětvit větvícím uzlem (fork node). Vstupní tok se rozdělí do výstupních uzlů. Naopak více toků je možné sloučit slučovacím uzlem (join node). Výstupní tok může pokračovat po doběhnutí všech vstupních toků. Větvící uzel se slučovacím je možné kombinovat, například může mít uzel tři vstupy a dva výstupy. Rozhodovací uzel (decision node) na základě hodnoty na rozhodovacím vstupu vybírá jednu z alternativ. Všechny alternativy a vstup jsou buď tok řízení nebo datový tok. Spojení alternativních toků z rozhodovacího uzlu je řešeno spojovacím uzlem (merge node).

Cyklů je možné v diagramu aktivit dosáhnou pomocí podmínek a cyklickým propojením uzlů.

Další možnosti v UML

UML umožňuje definovat šablony (template) pro parametrizaci tříd, balíčeků nebo operací. Parametrem může být klasifikátor nebo i hodnota. Parametry jsou naznačeny v rohu elementu. Na parametrizované entity mohou být v UML navázány konkrétní hodnoty, které v těle nahradí parametry. Modelováno je to generalizací se stereotypem `«bind»` jak je vidět na obrázku 3.3.

Pomocí profilů UML je možné definovat vlastní stereotypy. UML profil je definován pomocí UML balíčku označeným stereotypem `«profile»`. Balíček profilu obsahuje samotné stereotypy modelované jako třídy se stereotypem `«stereotype»`. Každý stereotyp se vztahuje k některé metatřídě, to definuje na které uzly je stereotyp aplikovatelný. Aby mohly být vytvořené stereotypy v balíčku používány, musí být nejprve na balíček aplikován konkrétní profil. Vytvoření i aplikování profilu je naznačeno na obrázku 3.4.



Obrázek 3.3: Příklad UML šablony (Převzato z http://www.uml-diagrams.org/template.html)

Stereotyp může obsahovat i parametry libovolných typů, jejichž hodnoty jsou zadány při aplikaci stereotypu, tento koncept se nazývá označené hodnoty (tagged value).

V UML je možné definovat omezení, tato omezení jsou výrazy, které je možné vyhodnotit, zda jsou splněny nebo ne. Omezení mohou být v UML použita na mnoha místech a samotný element určuje kdy je omezení vyhodnocováno. Například operace může mít precondition a post-condition podmínky. Podmínky musejí být v korektně navrženém systému vždy splněny. Podmínky mohou být definovány a vyhodnocovány různými způsoby. Mohou být definovány v nějakém programovacím jazyce, v přirozeném jazyce nebo například v OCL.

Object Management Group (OMG)

Object Management Group [1] (OMG) je mezinárodní neziskové konsorcium vytvářející technologické standardy týkající se objektově orientovaného vývoje software, OMG bylo založeno v roce 1989 a nyní zahrnuje stovky organizací [9]. OMG vytvořilo standardy například pro Unified Modeling Language (UML), Model Driven Architecture (MDA), Business Process Model & Notation (BPMN) a spoustu dalších.

V následujících podkapitolách jsou popsány OMG standardy relevantní k této práci.

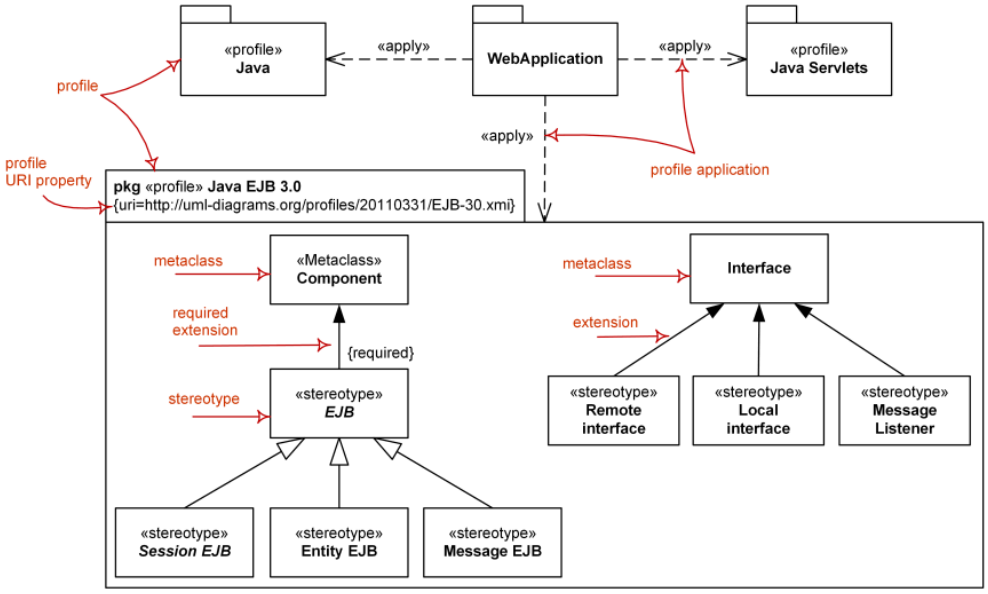
MetaObject Facility

MetaObject Facility (MOF) je čtyřvrstvá architektura pro definici modelů. V nejvyšší vrstvě M3 je meta-meta model, ten MOF používá pro popis metamodelů ve vrstvě M2. Příkladem metamodelu M2 je například UML metamodel, což je model popisující elementy UML. Ve vrstvě M1 jsou samotné modely, to může být například konkrétní UML diagram tříd. Poslední vrstva M0 nebo také datová vrstva reprezentuje objekty reálného světa [13]. MOF vrstvy jsou znázorněny na obrázku 3.5.

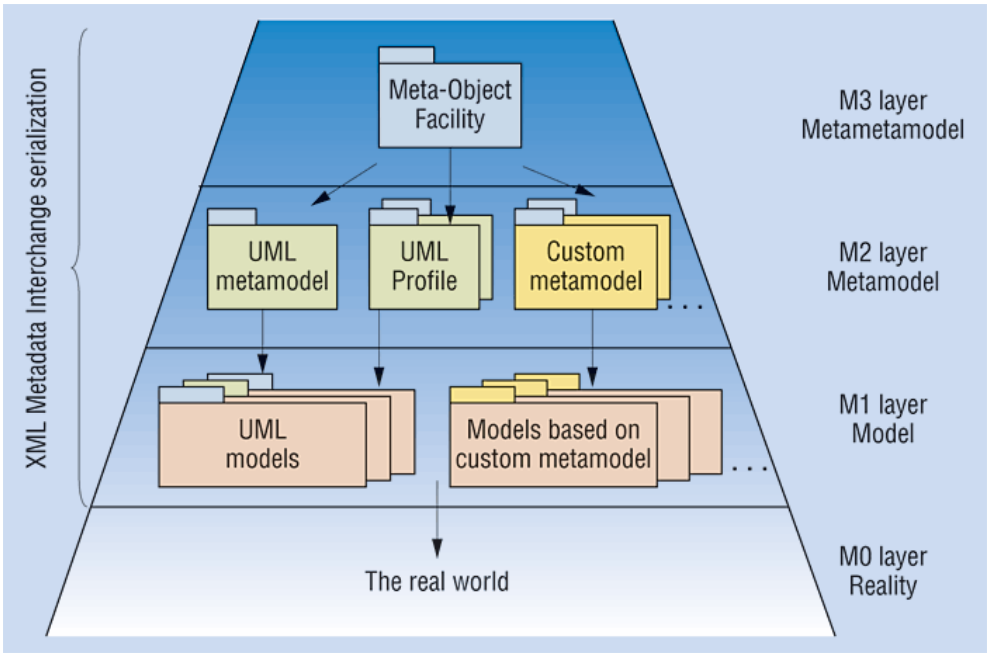
XML Metadata Interchange

XML Metadata Interchange (XMI) je standard používaný pro výměnu MOF modelů [15]. XMI umožňuje popisovat objektovou strukturu v XML souborech pomocí elementů a atri-

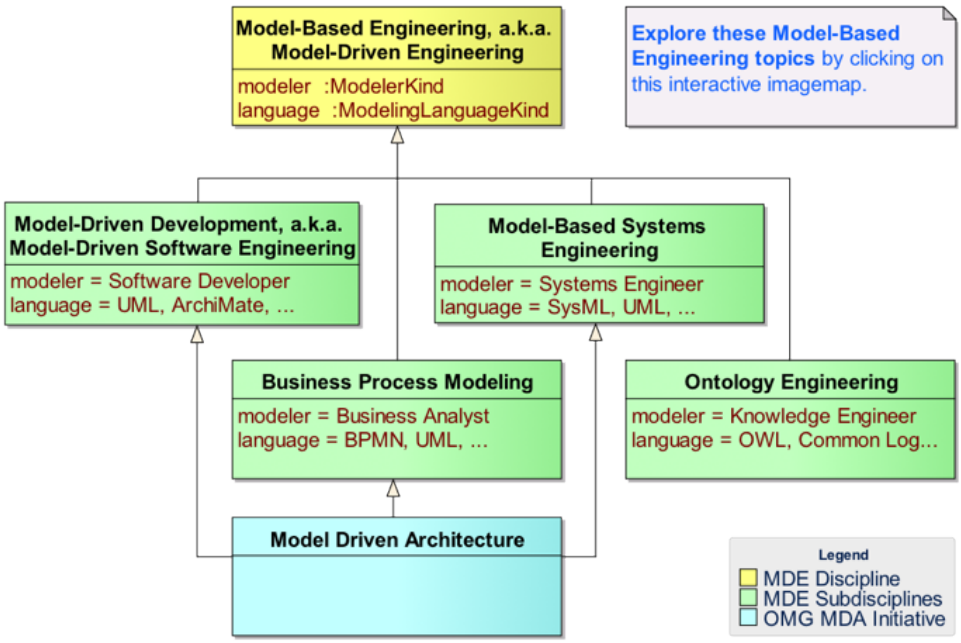
[1] Webové stránky OMG: http://www.omg.org/



Obrázek 3.4: Příklad UML stereotypu (Převzato z http://www.uml-diagrams.org/template.html)



Obrázek 3.5: Vrstvy MOF (Převzato z http://www.jot.fm/issues/issue_2006_11/article4/)



Obrázek 3.6: Vztahy mezi modelem řízenými paradigmaty (Převzato z http://modelbasedengineering.com/)

butů. V XMI souborech je možné z objektů odkazovat na jiné objekty v souboru nebo i na objekty do jiného souboru. Elementy jsou identifikovány vlastností ID nebo UUID. ID by mělo být unikátní v rámci jednoho dokumentu, na takové id pak odkazují reference. UUID by měl být globálně jednoznačný identifikátor. Je možného ho vytvořit například použitím URI, takto vypadá například UUID pro metamodel diagramu případů užití: http://www.omg.org/spec/UML/20200901/uml.xml#UseCase.

Object Constraint Language

Object Constraint Language (OCL) je textový formální jazyk rozšiřující UML, je založeno na predikátové logice prvního řádu, ale používá syntaxi podobnou programovacím jazykům [12]. OCL slouží pro definici omezení a dotazů nad entitami UML. Je deklarativní a má přesnou sémantiku. Je použitelný pro všechny MOF modely.

3.2 Model-driven a Model-based Engineering

Model Based Engineering (MBE) nebo také Model Driven Engineering (MDE) je zastřešující pojem pro několik disciplín, na obrázku 3.6 jsou znázorněny jednotlivé disciplíny a jejich vztahy, ve všech těchto disciplínách hraje významnou roli modelování, avšak význam a uplatnění modelů se liší [3]. Obecně důležitost modelů je u Model-Driven paradigmat větší než u Model-Based.

Business Process Model and Notation

Primárním cílem Business Process Model and Notation (BPMN) je poskytnout jednotnou notaci pro popis podnikových procesů pomocí diagramů [10].

Notace BPMN je velice podobná diagramu aktivit z UML, obsahuje například události, aktivity a propojení. Oproti diagramu aktivit obsahuje navíc artefakty, pomocí nichž znázorňuje, jaké artefakty aktivita používá a jaké vytváří.

Ontology Engineering

Ontology Engineering je subdisciplína MBE, věnuje se metodám a metodologiím pro vytváření ontologií. Ontologie je formální popis pojmů a jejich vztahů. Ontologie mohou být popsány jak graficky, tak i textově v různých jazycích [2].

Model based system engineering

Model based system engineering (MBSE) je metodologie systémového inženýrství zaměřená na vytváření a využívání modelů jako primárních prostředků pro výměnu informací mezi inženýry místo dokumentů [4].

Model driven development tzv. Model driven software engeneering

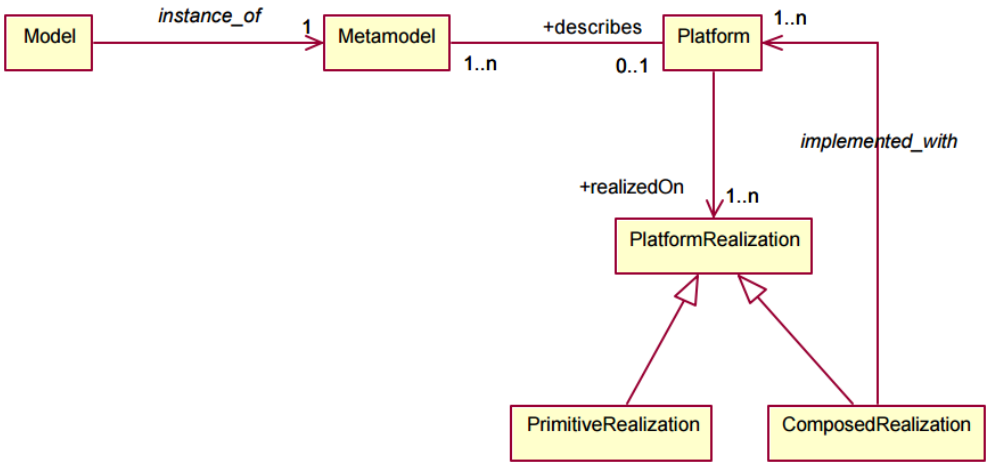
Model driven development (MDD) se oproti MDE zaměřují konkrétněji na vývoj softwarových systémů. Hlavní myšlenkou je přesunutí vývoje softwaru na vyšší úroveň abstrakce použitím modelů jako základních artefaktů a pomocí transformací modelů generovat zdrojový kód a jiné artefakty [17].

MDD považuje modely za primární artefakty během celého životního cyklu vývoje software. Cílem je automatizovat vývoj, eliminovat práci při programování a částečně ji přenést na modelování.

V MDD je věnována primární pozornost modelům a nikoliv počítačovým programům, hlavní výhodou je, že modely nejsou závislé na implementační technologii a jsou blíže doméně problému než populární programovací jazyky, to umožňuje modely snadněji specifikovat, pochopit a udržovat, v některých případech dokonce může systém vytvářet doménový expert namísto technologického specialisty [19].

Tak jako každé paradigma má i Model driven development své výhody a nevýhody. Výhody a nevýhody v následujícím seznamu jsou převzaty z [17].

+ Vyšší produktivita vývojářů způsobena generování kódu a jiných artefaktů.
+ Snazší udržovatelnost a přesun na jinou platformu díky abstraktním platformě nezávislým modelům.
+ Konzistence artefaktů díky jejich automatickému generování.
+ Usnadňuje komunikaci mezi zúčastněnými stranami. Modely neobsahují implementační detaily, které nejsou nutné pro porozumění logickému chování systému. Modely jsou srozumitelné pro více zainteresovaných osob.
− Zvýšená úroveň abstrakce může vést ke snížení komplexnosti artefaktů používaných vývojáři. Vždy existuje kompromis mezi zjednodušením pomocí zvýšením úrovně abstrakce a přílišným zjednodušením, kde chybí podstatné detaily pro transformace na cílový kód.



Obrázek 3.7: MDA závislosti (Převzato z [20])

Model driven architecture

Model driven architecture (MDA) je standard vytvořený organizací OMG. Je to konkrétní implementace MDD. MDD může používat jakékoliv modely, MDA však používá konkrétně UML.

Hnací silou za MDA je fakt, že softwarový systém může být eventuálně nasazen na různé platformy, pracující odděleně nebo společně, navíc se platformy s časem obvykle vyvíjejí, proto je výhodné použíta PIM a ty následně transformovat na PSM [20].

Na obrázku 3.7 jsou naznačeny závislosti mezi MDA modely, metamodely a realizacemi modelů na platfomách.

Jsou definovány tři druhy modelů z hlediska závislosti na cílové platformě. Modely jsou transformovány od CIM přes PIM až na PSM, PSM může být například i zdrojový kód.

- **Coputation independent model (CIM):** Jsou to primární modely, reprezentují systém z abstraktního hlediska. Struktura a konkrétní zpracování jsou v těchto modelech skryté.
- **Platform independent model (PIM):** Platformě nezávislé modely jsou obecné pro různé platformy. Neobsahují platformě specifické informace.
- **Platform specific model (PSM):** Tyto modely jsou závislé na konkrétní platformě není možné model použít pro jinou platformu.

3.3 Executable UML

Executable UML (spustitelné UML) jsou takové modifikace klasického UML, aby bylo možné systém popsat kompletně včetně implementačních detailů a takovéto modely bez dalších úprav spouštět nebo z nich generovat kompletní implementaci. Standardní UML nebylo vytvořeno za tímto účelem, proto vznikly různé modifikace UML. Je možné vytvořit různé spustitelné UML (executable UML), například založené na případech užití, workflow, metodách, stavových automatech a jejich kombinacích [16].

Výhodami spustitelné specifikace je brzká zpětná vazba pro vývojáře a dřívější dostupnost spustitelného programu, nevýhodou je tendence vývojářů modelovat vlastnosti cílové platformy [18].

Executable UML (xUML)

Executable UML (xUML) je další vyšší vrstva abstrakce při vývoji software [8].

xUML používá pro popis systémů tři prostředky, a to diagram tříd, stavový diagram a action language (AL). Diagram tříd je použit pro modelování struktury věcí reálného světa. Stavový diagram slouží pro popis objektů, které mají nějaké stavy nebo životní cyklus. Action language (AL) slouží pro popis výpočtů a algoritmů.

Foundational Subset for Executable UML Models (fUML)

Foundational Subset for Executable UML Models (fUML) obsahuje podmnožinu elementů UML 2. fUML navíc precizně definuje sémantiku modelů, nutnou pro jejich spouštění a kompletní generování kódu.

S představením OMG fUML standardu definujícím funkční sémantiku podmnožině UML a odpovídajícího virtuálního stroje mohou být UML modely použity ne pouze pro návrh systémů ale také pro kompletní vytváření spustitelných systémů, to byl důležitý krok pro naplnění potenciálu UML. [7]

fUML je platformě nezávislé, proto je vhodné pro modelování abstraktních modelů, ze kterých je možné generovat kód na různé platformy.

fUML používá pro definici statické struktury systému diagram tříd, diagram tříd je obecně popsán v odstavci 3.1.

Pro popis úplného chování je používán diagram aktivit, diagram aktivit je obecně popsán v odstavci 3.1. Diagram aktivit umožňuje detailní popis chování, umožňuje modelovat podmínky, cykly, volání funkcí, posílání signálů atd. Diagramem aktivit je popisováno celé chování systému a také chování metod tříd.

Modelovat chování graficky nemusí být vždy vhodné, proto vnikl jazyk ALF pro textový popis fUML modelů, jazyk ALF je popsaný v následující kapitole 3.3.

Action Language For Foundational UML

Action Language For Foundational UML (ALF) je jazyk používaný pro textový popis fUML elementů. Sémantika spouštění ALF jazyka je dána mapováním kódu na elementy fUML, výsledek spuštění ALF kódu je totožný se spuštěním fUML diagramu vygenerovaného ze stejného ALF kódu [11].

Primárním cílem ALF je poskytnout textový popis pro spustitelné chování modelů. Toto chování je nutné jinak definovat graficky, což není vždy vhodné a přehledné. Typicky při definování těl metod nebo chování přechodů u stavových automatů. ALF také poskytuje notaci pro popis strukturálních modelovacích elementů. fUML podmnožinu UML je možné kompletně modelovat s použitím pouze ALF. Proto je možné bez problémů převádět ALF na fUML a naopak.

ALF používá notaci podobnou jazyku C nebo Java, protože tato notace je dobře známá pro programátorům. ALF poskytuje jmenný systém založený na UML pro odkazování na elementy mimo aktivitu a současně poskytuje lokální jména pro odkazování uvnitř aktivit. ALF používá implicitní typový systém, který umožňuje, avšak nevyžaduje explicitní deklaraci typů. Typový systém zajišťuje statickou typovou kontrolu.

Specifika udává tři úrovně podpory syntaxe jazyka ALF v modelovacích nástrojích. První úroveň obsahuje pouze podmnožinu jazyka popisující výrazy a příkazy bez strukturálních elementů, jako jsou třídy, aktivity atd. V druhé úrovni je podporována kompletně syntaxe jazyka pro popis výrazů a příkazů bez strukturálních elementů. V třetí úrovni je kompletně podporována syntaxe jazyka ALF včetně popisu strukturálních elementů.

Existují tři způsoby, jak může být implementováno spouštění chování ALF kódu. První způsob je přeložení ALF kódu do nějaké spustitelné reprezentace na cílovou platformu, například překlad do jazyku Python, a následná interpretace nebo spuštění na cílové platformě. Druhý způsob je překlad ALF kódu na fUML modely a spuštění těchto modelů podle sémantiky fUML. Třetí možností je přímá interpretace nebo spouštění ALF kódu.

Kapitola 4

Platforma Android

V této kapitole je stručně popsána struktura platformy Android, konkrétně Operační systém Android, způsoby vývoje Android aplikací, možnosti testování a možnosti ladění.

Platformu Android spravuje konsorcium Open Headset Aliance, které se skládá z desítek společností včetně společnosti Google, Intel, Samsung, Qualcom atd. Android je veřejně publikován pod open-source licencí Apache/MIT, což mělo nepochybně pozitivní vliv na jeho rozšíření [6].

Android vede na trhu s chytrými telefony, koncem roku 2016 měl tržní podíl 87,5% [1], také počet vytvořených aplikací je na tuto platformu největší.

Klíčovou vlastností Androidu je jeho multiplatformnost, díky které je možné Android používat na různých zařízeních a architekturách, proto je Android nasazován na zařízení od různých výrobců. Tato výhoda s sebou nese nevýhodu v podobě horší optimalizace. Potřebný výkon pro běh systému i aplikací je vyšší, než by bylo nutné s optimalizovaným nativním kódem, jako je tomu například u mobilního operačního systému iOS od firmy Apple.

Výrobci mají možnost Android rozšiřovat o své nadstavby a systém si tak upravit podle svých představ.

Platforma Android byla navržena pro mobilní zařízení jako jsou chytré mobilní telefony a tablety. Postupem času se však tato platforma dostává do dalších zařízení jako jsou například chytré hodinky, chytré televize, chytré brýle nebo třeba auta.

Při návrhu platformy byly brány v úvahu vlastnosti cílových zařízení jako omezený výkon, malá kapacita baterie, malé rozměry displejů i malá paměť.

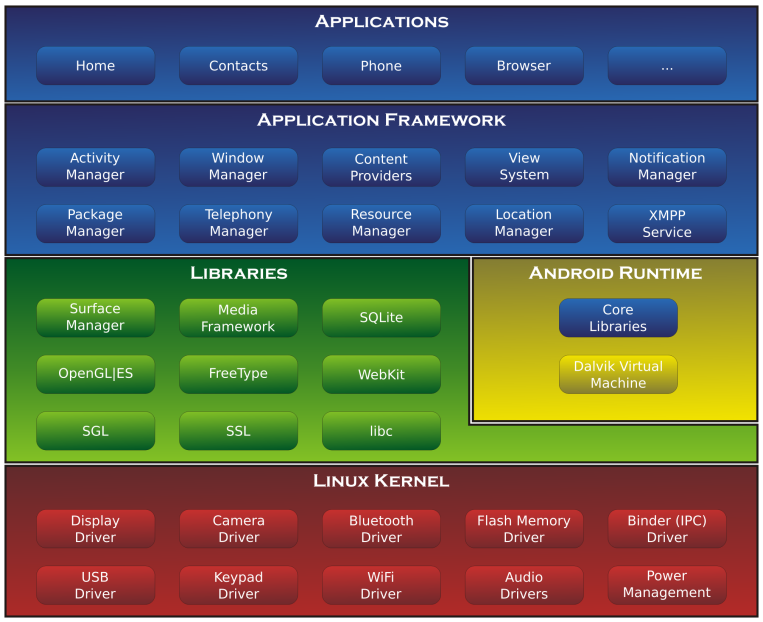
4.1 Operační systém Android

V této kapitole je popsána architektura operačního systému Android a její části. Struktura platformy je znázorněna na obrázku 4.1, jednotlivé části jsou popsány dále v textu.

Jako základ operačního systému Android slouží linuxové jádro. Jádro je částečně upraveno pro možnosti mobilních zařízení a ořezáno o nepotřebné funkce. Jádro slouží k interakci s hardwarem, čímž zajišťuje abstrakci vyšších vrstev od hardwaru. Jádro řídí správu paměti, přidělování procesoru, práci se sítí i ovladače.

Nad jádrem je vrstva nativních knihoven napsaných v C/C++. Tyto knihovny poskytují přístup ke komponentám systému Android.

[1] Převzato z https://www.letemsvetemapplem.eu/2016/11/03/android-stale-vede-podilu-trhu/



Obrázek 4.1: Architektura operačního systému Android (Převzato z [6])

Android běhové prostředí (runtime) zajišťuje běh jednotlivých aplikací. Pro každou aplikaci je vytvořen nový proces s virtuálním strojem Dalvik, ten zajišťuje spuštění a běh souborů DEX. Tyto soubory vznikají kompilací klasických CLASS souborů nástrojem DX. Soubory DEX a virtuální stroj Dalvik jsou optimalizovány pro mobilní zařízení.

Aplikační framework je napsaný v jazyce Java a poskytuje aplikacím vyšší funkce systému. Vývojáři převážně využívají komponenty této vrstvy, které poskytují základní služby systému.

Na nejvyšší úrovni architektury jsou samotné aplikace.

4.2 Vývoj Android aplikací

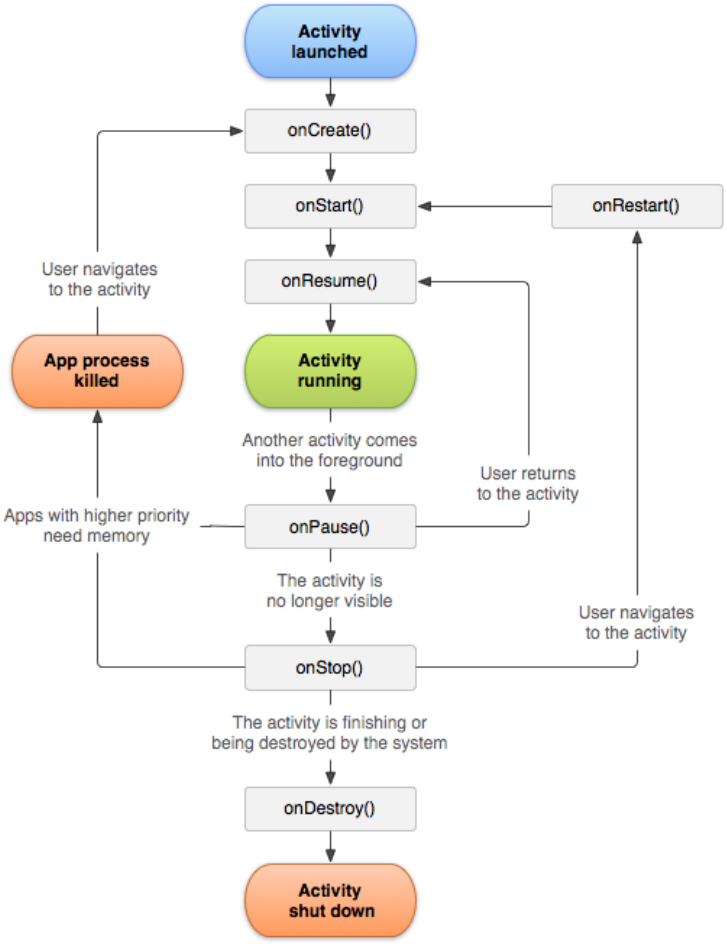
Android aplikace se skládá ze zdrojových kódů v jazyce Java, které jsou překládány do DEX souborů.

Android aplikace obsahuje XML soubor nazvaný AndroidManifest. Tento soubor obsahuje informace o aplikaci jako například požadovaná práva, verze SDK, podporované obrazovky, definici aktivit, definici Služeb apod.

Základním prvkem aplikací jsou aktivity, tyto aktivity definují GUI obrazovky a její chování. Každá aktivita má svůj životní cyklus, někdy vzniká, někdy je pozastavena a někdy zaniká. Životní cyklus Android aktivit je znázorněn na obrázku 4.2. Na tyto změny stavu je možné reagovat. Fragmenty mají také své GUI a životní cyklus ale oproti aktivitám je lze vkládat do layoutu aktivit.

GUI android aplikace lze vytvářet použitím grafických prvků operačního systému jako stavebních kamenů. Děděním z tříd existujících prvků lze vytvářet libovolné vlastní grafické prvky.

Postupem času vznikají nové verze operačního systému Android, avšak některá zařízení se nedočkají aktualizací na nové verze. Proto musejí vývojáři vytvářet aplikace na různé verze Android s rozdílnými API, pokud chce zachovat zpětnou kompatibilitu.



Obrázek 4.2: Životní cyklus Android aktivity (Převzato z [6])

ORM frameworky

Primárním uložištěm pro uložení dat je v systému Android kromě souborového systému SQLite databáze. Zajímavými nástroji pro usnadnění práce s SQLite databází jsou Object Relation Mapping (ORM) frameworky. Tyto frameworky poskytují abstraktní vrstvu nad SQLite databází. Díky tomu není nutné psát SQL dotazy, ale je možné s persistentními objekty pracovat jako s klasickými objekty. Volně dostupný je například Sugar ORM [2].

Android anotace

Existují frameworky pro Android, které eliminují často opakovaný kód použitím anotací. Takový kód se rychleji píše a je přehlednější. Například existuje framework AndroidAnotations [3].

Android studio - IntelliJ

Android Studio [4] je rozšířením vývojového prostředí IntelliJ [5] Community Edition.

Android Studio obsahuje WYSIWYG (what you see is what you get) editor Android layoutů. Je možné přetažením komponent z lišty nástrojů vkládat do GUI nové prvky. Druhou možností je modifikovat přímo XML kód, ve kterém je layout ukládán.

Ladění aplikací

Android studio poskytuje kromě programování aplikací také možnosti ladění. Po zkompilování je možné aplikaci spustit.

Ke spuštění není nutné hardwarové zařízení. Platforma android obsahuje emulátor hardwarových zařízení. Virtuálnímu zařízení je možné nastavit parametry jako velikost operační paměti, velikost displeje atd.

Po připojení hardwarového a instalaci ovladače je možné na zařízení nahrát vyvíjenou aplikaci a spustit.

K logování používá Android nástroj logcat, přímo v okně Android Studia jsou zobrazovány logované zprávy ze zařízení. Tyto zprávy je možné filtrovat.

Pokud je aplikace spuštěna v režimu ladění, je možné běh aplikace krokovat, včetně zobrazení aktuálních hodnot proměnných.

[2] Webové stránky Sugar ORM frameworku: http://satyan.github.io/sugar/
[3] Webové stránky AndroidAnotations frameworku: http://androidannotations.org/
[4] Webové stránky Android Studia: https://developer.android.com/studio/index.html
[5] Webové stránky IntelliJ: https://www.jetbrains.com/idea/

Kapitola 5

Existující nástroje s podporou MDD

V této kapitole jsou popsány existující nástroje podporující MDD (Model Driven Development). Nejprve je popsán kvalitní komerční nástroj Enterprise Architect, poté následuje popis open source nástrojů.

Enterprise Architect

Enterprise Architect (EA) [1] je nástroj pro modelování systémů od společnosti Sparx Systems. Podporuje modelování byznys i IT systémů včetně modelování real-time a vestavěných systémů. Také transformace mezi modely a generování kódu je v EA možné včetně generování dokumentace. Je předpřipraveno spoustu šablon pro generování a transformace, je možné je také upravovat nebo vytvářet vlastní. EA podporuje import a export modelů ve formátů XMI. V EA je možné simulovat stavové diagramy, diagramy aktivit, diagramy interakce a BPMN procesy modelovaných systémů.

Simulace podporuje následující funkce:

- Dynamické spouštění modelů chování
- Efekty psané ve standardním Javascriptu
- Definování a spouštění trigrů v běžící simulaci
- Definování a používání různých množin trigrů pro simulování různého pořadí událostí
- Automatické spouštění trigrů pro simulaci komplexní historie bez uživatelské interakce
- Zobrazování a nastavování hodnot proměnných při běhu simulace
- Vytváření a volání COM objektů během simulace pro rozšíření dosahu simulace poskytnutí vstup a výstupu
- Definování proměnných, konstant a funkcí skriptem před spuštěním

Sparx Systems nabízí pouze placené verze Enterprise Architectu nebo třicetidenní trial verzi.

[1] Webové stránky Enterprise Architectu: http://www.sparxsystems.com/products/ea/

Eclipse Modeling Framework

Eclipse Modeling Project [2] je skupina rozšířitelných nástrojů a frameworků pro modelování systémů v Eclipse. Jeho jádrem je Eclipse Modeling Framework (EMF). Eclipse Modeling Project nabízí mnoho nástrojů pro práci s modely všeho druhu, například obsahuje nástroje pro grafické nebo textové modelování, transformaci a analýzu modelů.

Eclipse Papyrus a Moka plugin

Papyrus [3] je modelovací prostředí, vytvořené jako rozšíření vývojového prostředí Eclipse.

Do Papyru je možné přidat modul Moka [4], který umožňuje spouštění fUML modelů. Moka je integrován do debugovacího nástroje Eclipse pro krokování spouštěného modelu. Spouštěný model je vizualizován včetně animací.

Chování aktivity v diagramu aktivit lze v Papyru specifikovat textově v jazyku ALF. ALF editor zvýrazňuje syntaxi i provádí její kontrolu. Po napsání ALF kódu aktivity je ALF kód převeden na příslušnou reprezentaci v fUML. Papyrus je dostupný pod open source licencí EPL včetně pluginu Moka.

Moliz

Moliz je projekt pro spouštění modelů založených na standardu fUML naprogramovaný v jazyce Java. Moliz používá referenční implementace fUML popsanou níže. Moliz je implementován jako plugin do Eclipse vývojového prostředí a vyžaduje nainstalovaný framework Eclipse Modeling Framework.

MIT App Inventor

MIT App Inventor [5] je grafický nástroj pro tvorbu Android aplikací. Celé vývojové prostředí běží ve webovém prohlížeči. Obsahuje nástroj pro tvorbu vzhledu aktivit, kde je možné do aktivity přidávat grafické prvky a nastavovat těmto prvkům vlastnosti. Dále obsahuje editor bloků, kde je definováno chování aktivit skládáním a propojováním funkčních bloků. Výsledná aplikace je kompilována na serveru a na klientský počítač je odesílán APK soubor aplikace.

[2] Eclipse Modeling Project: http://www.eclipse.org/modeling/
[3] Eclipse Papyrus: https://eclipse.org/papyrus/
[4] Moka modul: https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution
[5] Webové stránky MIT App Inventoru: http://appinventor.mit.edu/

Kapitola 6

Návrh vlastního MDD nástroje pro vývoj Android aplikací

V této kapitole je popsán návrh nástroje pro vývoj Android aplikací pomocí MDD, modelování v nástroji je založeno na technologiích fUML a ALF. Vytvářený nástroj je navržen jako nové vývojové prostředí, které je pojmenováno fUmlStudio. Veškeré části vyvíjeného nástroje budou volně dostupné jako otevřený software. Nástroj bude možné spouštět na desktopových platformách s podporou virtuálního stroje Java.

Uživatel bude primárně přímo používat hlavní část nástroje a to fUmlStudio. Samotné studio bude k překladu aplikace, generování kódu a dalším klíčovým činnostem používat Gradle plugin pojmenovaný fUmlGradlePlugin. Funkce systému Gradle se spouští přes tzv. tasky, které představují něco jako příkazy nebo úkoly. Případy užití fUmlStudia a fUmlGradlePluginu jsou v diagramu 6.1. Tento Gradle plugin bude možné používat z příkazové řádky a nebude tak nutné k vývoji aplikací používat fUmlStudio.

Tasky Gradle pluginu budou spustitelné z menu vývojového prostředí. Základní nastavení gradle pluginu bude také možné provést přes fUmlStudio.

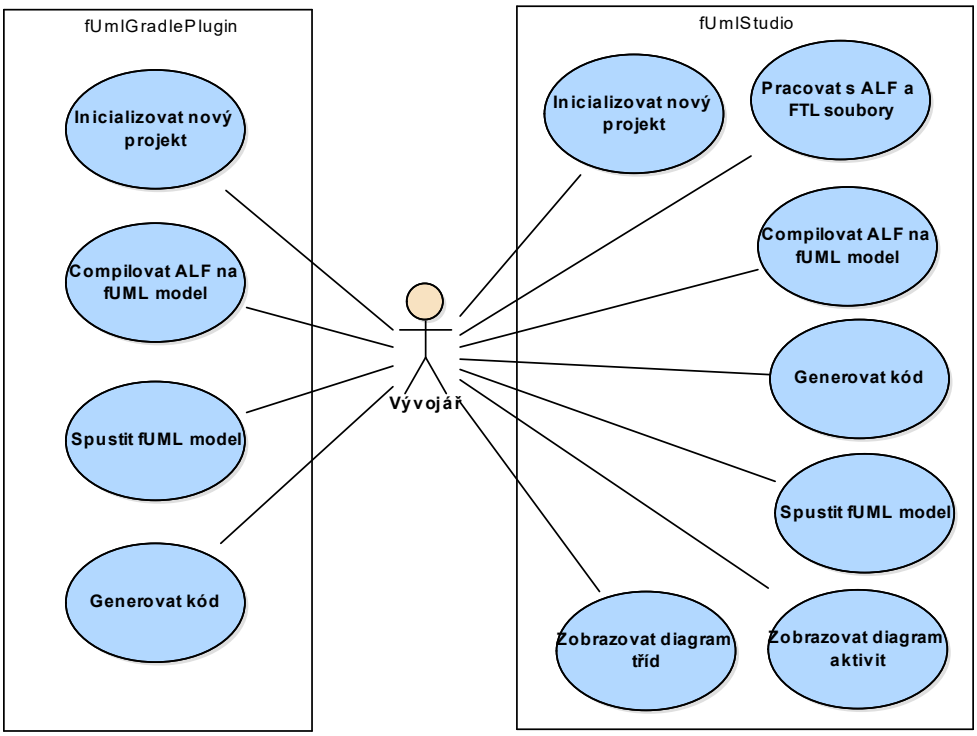
Důležitou částí fUmlStudia bude editor zdrojových kódů. Modely budou primárně vytvářeny textově v jazyce ALF. Vytvořené modely bude však možné zobrazovat v fUmlStudiu i graficky. Editace modelů v grafickém editoru nebude možná z důvodu náročné implementace této funkcionality.

V fUml studiu bude možné vymodelované aplikace přímo spouštět, výstup aplikace bude při běhu ve vývojovém prostředí ihned zobrazován. Pro spuštění modelu bude nejprve nutné spustit mapování ALF kódu na fUML model, tento model je následně možné spustit.

fUmlStudio respektive fUmlGradlePlugin bude umožňovat generování kódu z modelů pro platformu Android. Tento generátor bude transformovat modely na jiné artefakty jako jsou zdrojové kódy, konfigurační soubory atd. Způsob transformace je definován pomocí transformačních šablo v jazyce FTL (FreeMarker Template Language). Bude možné jednoduše vytvářet nové a modifikovat stávající transformační šablony. Při generování kódu v fUmlStudiu bude možné vybírat, která sada transformačních šablon bude použita.

6.1 Návrh vývojového prostředí

Okno fUmlStudia bude obsahovat v horní části hlavní menu pro práci s projekty a soubory. fUmlStudio bude umožňovat správu projektů, jako je vytváření nových a otevírání stávají-



Obrázek 6.1: Diagram případů užití navrhovaného nástroje

cích. Přes menu bude také možné vytvářet nové ALF a FTL soubory a editované soubory ukládat.

Pod hlavním menu bude menu nástrojů pro spouštění Gradle tasků pluginu a jejich nastavování.

Struktura otevřeného projektu bude přehledně zobrazena ve stromové hierarchii v levé části aplikace.

fUmlStudio bude umožňovat editovat několik souborů současně, otevřené editory budou rozděleny v záložkách a bude možné pohodlně mezi nimi přepínat a zavírat je. Tento správce záložek bude umístěn v hlavní části okna aplikace.

ALF soubory budou editovány v editoru zdrojových kódů. Ostatní textové soubory například výstupní Java soubory budou zobrazovány v editoru textu. V grafickém UML editoru bude zobrazován obecný diagram tříd a diagram Android aktivit. Ve stromové struktuře projektu budou dvě virtuální položky, jedna pro zobrazení diagramu tříd vyvíjené aplikace a druhá pro zobrazení digramu přechodů mezi aktivitami.

UML grafický editor a editor kódu

Klíčovou součástí vyvíjeného nástroje je grafický UML editor. V tomto editoru bude zobrazován diagram tříd a diagramy přechodu aktivit. V grafickém editoru bude možné pouze upravovat pozice elementů a jejich propojení, nikoliv samotné elementy.

Jako základ grafického editoru slouží knihovna Graph Editor [1], jedná se o obecnou knihovnu pro tvorbu diagramů. Bude nutné doimplementovat potřebné grafické prvky pro zobrazování UML diagramů. Je nutné vytvořit grafický uzel pro zobrazení tříd a jejich

[1] Graph Editor knihovna: https://github.com/tesis-dynaware/graph-editor

dědičností v diagramu tříd. Diagram přechodů aktivit bude zobrazovat aktivity jako třídy, přechod na další aktivitu bude znázorněn šipkou v diagramu.

Při prvním otevření diagramu nebude dostupné rozmístění uzlů, budou pouze načteny třídy z fUML modelu. Grafický editor chybějící uzly vytvoří na určenou pozici. Každý uzel obsahuje několik konektorů, na které se napojují propojení. Při zobrazování dědičnosti u poprvé otevřeného diagramu jsou pro automatické vytvoření propojení vybrány konektory s nejmenším počtem již existujících spojení. Při uložení a opětovném otevření diagramu se již zobrazí uzly na posledně uložených pozicích. Pokud byly vytvořeny nové třídy, tak jsou opět uzly vytvořeny na definovanou pozici. Pokud naopak byla třída odstraněna je jí odpovídající uzel z grafu odstraněn.

Spouštění modelů

Nástroj bude umožňovat vytvářený model přímo spouštět bez nutnosti generování kódu. Bude tak možné částečné ladění modelu. Po spuštění modelu bude zobrazen jeho výstup v logovacím panelu fUmlStudia. Pro ladění modelu je nutné na vhodná místa modelu vložit výpis ladícího textu.

6.2 Návrh Gradle pluginu

Gradle plugin fUmlGradlePlugin bude umožňovat kompletní vývoj aplikace bez použití fUmlStudia. Gradle plugin bude obsahovat task pro inicializaci nového projektu `gardle fUmlInstall`, pro smazání souborů vniklých při mapování a generování kódu `gradle fUmlClean`. Pro mapování ALF kódu na fUML bude task `gradle fUmlCompile`, pro spuštění fUML modelu bude sloužit task `gradle fUmlRun` a pro generování kódu bude určen task `gradle fUmlCodeGenerate`.

Gradle plugin bude možné integrovat do jiného gradle projektu. Gradle umožňuje modifikovat stávací pořadí a závisloti úloh. Je tak možné definovat, že úloha textttfUmlCodeGenerate fUmlGradlePluginu bude spuštěna před kompilací samotné Android aplikace.

Spouštění úlohy vyžaduje určité informace jako je například cesta k ALF knihovnám, umístění vytvářených modelů nebo třeba cílová složka generování kódu. Tyto parametry bude možné nastavovat v projektu v souboru `settings.gradle`, pokud nebudou parametry nastaveny, budou použity defaultní hodnoty.

Gradle plugin bude pro základní operace s ALF a fUML používat referenční implementace těchto technologií. Pro generování kódu bude nutné vytvořit další aplikaci nebo knihovnu. Návrh generátoru kódu je popsán v následující kapitole.

6.3 Návrh generátoru kódu

Generátor kódu transformuje platformě nezávislé modely (PIM) na platformě závislé (PSM), v tomto případě zdrojový kód.

Generování kódu bude definováno transformačními šablonami. Šablony budou textově popisovat struktury výstupních souborů. Budou umožňovat iterovat přes kolekce například přes všechny třídy v modelu. Přes tečkovou notaci budou dostupné další atributy objektu například jméno třídy. V šablonách by mělo být možné získat maximální množství informací o modelu a tak generovat jakýkoliv artefakt.

Šablony by mělo být možné do sebe vkládat a rozdělit tak definici generování do více souborů a tím je zpřehlednit.

Generátor kódu by měl umožňovat v šabloně definovat umístění a jméno generovaného souboru. Mělo by být možné iterovat například přes třídy a pro každou vytvořit soubor v definovaném umístění a s definovaným jménem.

Bylo by dobré generovat i kód pro těla metod. To bude náročnější než generová koster tříd. K tomu pravděpodobně nebudou šablony vhodné a samotná implementace bude náročná. V šabloně bude kód metody dostupný jako parametr objektu metody přes tečkovou notaci.

Použití vlastního Java kódu

Ruční modifikace vygenerovaného kódu není vhodným řešením, při novém generování kódu by došlo ke ztrátě těchto úprav. Ideální je modifikovat model, ze kterého je kód generován nebo transformační pravidla generátoru kódu. Modifikace transformačních pravidel však musí být obecná a funkční pro všechny případy, ne pouze pro konkrétní případ.

Pokud je však nutné vkládat vlastní Java kód nebo modifikovat generovaný zdrojový kód, jeví se jako nejvhodnější možnost ruční vytvoření nové třídy dědící z třídy generované a v této třídě potřebné úpravy provést přepsáním metod. V modelu může být operace označena jako abstraktní a její implementace bude muset být nutně doplněna ručně ve vytvořené třídě. Při označení operace jako abstraktní není pak ale možné vytvářet instance příslušné třídy, to znemožní spouštění modelu, proto je lepší operaci neoznačovat jako abstraktní ale nechat ji pouze bez implementace.

Generování kódu pro Android

Při modelování Android aplikací bude možné označit některé třídy jako Android aktivity nebo persistentní třídy. Ty třídy musí být nějakým způsobem odlišeny od ostatních tříd, aby generátor kódu mohl pro tyto třídy generovat jiný kód. Tohoto odlišení je například možné dosáhnout použitím stereotypů. Další možností je použití dědičnosti, například persistentní třídy budou dědit z knihovní třídy `PersistentLibrary.Persistent`. Při použití dědičnosti je možné i definovat operace knihovní třídě a ty pak používat v persistentních tříd například operace `Save`.

Dále musí být možné v modelu definovat přechody mezi Android aktivitami. Android aktivita bude modelována jako třída. Přechod na další aktivitu může být modelován jako operace. Operace musí být nějakým způsobem odlišitelná od ostatních operací. To je možné provést také stereotypem. Nebo operace může mít speciální jméno nebo parametry. V operaci musí být definována třída cílové aktivity a mělo by být možné do cílové aktivity předat parametry. V cílové aktivitě bude metoda volaná při jejím startu, ve které budou předané parametry dostupné.

Použití tříd systému Android nebude z ALF kódu možné. Bude nutné model nachystat tak, aby kód používající třídy Androidu byl oddělen v jiné operaci, která bude v modelu bez implementace. Při dědění z této třídy bude kód používající třídy Androidu doplněn v Javě v Android Studiu.

Kapitola 7

Implementace vlastního MDD nástroje

V této kapitole je popsána implementace vlastního MDD nástroje, který je předmětem této práce. Celý nástroje se skládá z několika částí, které budou jednotlivě popsány v následujících podkapitolách. Závislosti mezi částmi nástroje jsou ilustrovány v diagramu 7.1.

Hlavní část tvoří plugin do automatického sestavovacího nástroje Gradle. Základní vlastnosti Gradlu a postup implementace vlastního pluginu je popsána v sekci Implementace Gradle pluginu 7.2. Gradle plugin je implementován tak, aby celý vývoj aplikace mohl probíhat bez vývojového prostředí.

Gradle plugin využívá vytvořený generátor kódu, který generuje kód na základě transformačních šablon, implementace generátoru a tvorba transformačních šablon je popsána v podkapitole Implementace Generátoru kódu 7.1.

Gradle plugin je možné plnohodnotně využívat z příkazové řádky, a tak v extrémním případě není ani nutné pro vývoj aplikací v navrženém nástroji používat grafické uživatelské rozhraní. Ale pro pohodlnější a intuitivnější vytváření aplikací je také implementováno vývojové prostředí. Implementace vývojového prostředí je popsána v podkapitole 7.3.

Kompletní zdrojové kódy implementovaného nástroje jsou dostupné na GitHubu, fUmlCodeGenerator je dostupný zde [1], fUmlGradlePlugin zde [2] a fUmlStudio zde [3]. Zdrojové kódy i zkompilované výstupy implementovaných nástrojů jsou také uloženy na přiloženém DVD.

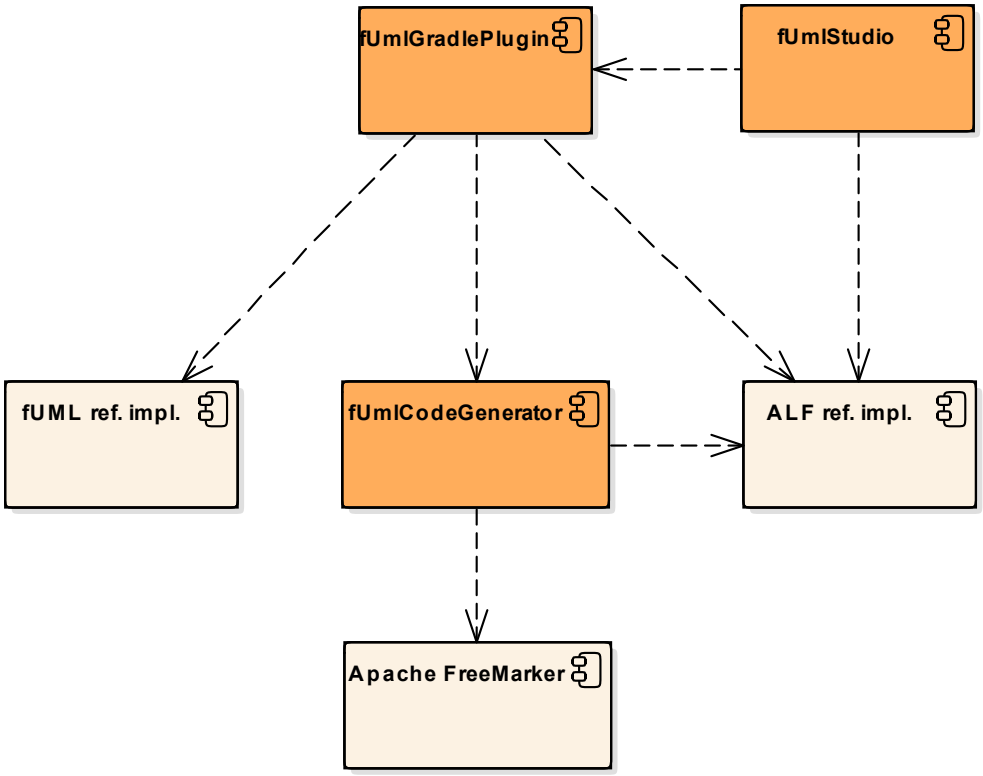
Referenční implementace fUML a ALF

Po vytvoření standardů fUML a ALF společností OMG, vznikly volně dostupné referenční implementace těchto standardů, které jsou naprogramované v jazyce Java.

fUML referenční implementace [4] je dostupná pod licencí Academic Free License version 3.0.

ALF referenční implementace [5] je dostupná pod licencí GNU General Public License (GPL) version 3.

[1] GitHub fUmlCodeGenerator: https://github.com/Standa631/fUmlCodeGenerator
[2] GitHub fUmlGradlePlugin: https://github.com/Standa631/fUmlGradlePlugin
[3] GitHub fUmlStudio: https://github.com/Standa631/fUmlStudio
[4] Webové stránky fUML referenční implementace: https://github.com/ModelDriven/fUML-Reference-Implementation
[5] Webové stránky ALF referenční implementace: https://github.com/ModelDriven/Alf-Reference-Implementation



Obrázek 7.1: Závislosti mezi částmi vyvíjeného nástroje

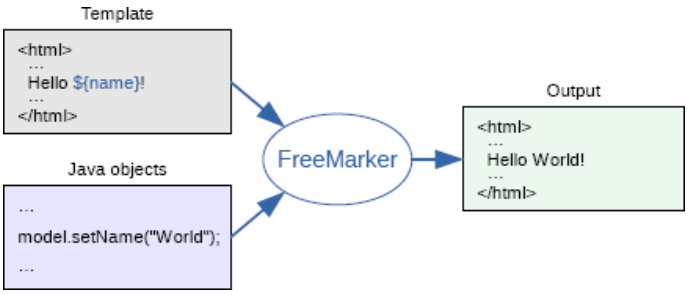
Obě tyto licence jsou open source a umožňují využití těchto implementací ve vlastním projektu, pokud bude projekt také pod licencí GPL. Proto implementované nástroje využívající ALF referenční implementaci jsou licencovány také pod GPL licencí.

fUML implementace vyžaduje na vstupu fUML model ve formátu XMI a zadání jedné nebo více aktivit modelu, které mají být spuštěny. Model je spouštěn ve viruálním stroji. Virtuální stroj pro spouštění fUML modelů nanaplňuje plný potenciál ze spustitelnosti modelů, nepodporuje analýzu běhu ani možnost běhu nějakým způsobem řídit [7]. V implementovaném nástroji je možné použít výpis do příkazové řádky pro ladění běhu modelu.

Referenční implementace ALF vyžaduje na vstupu cestu ke složce s ALF soubory, které jsou převáděny na fUML diagramy. Tyto výsledné diagramy jsou ve formátu XMI a je možné je spustit pomocí fUML referenční implementací.

ALF referenční implementace je využívána také v generátoru kódu pro namapování ALF kódu na fUML model a následné generování kódu z tohoto modelu. Při načítání fUML modelu pro zobrazení v grafickém editoru je také využita ALF referenční implementace pro načtení modelu do vnitřní reprezentace.

Tyto implementace jsou používány ve vyvíjeném nástroji k překladu ALF kódu na fUML model i ke spouštění fUML modelu. Část ALF kompilátoru načítající ALF kód a mapující jej na fUML model je použita pro načtení struktury modelů v generátoru kódu i při zobrazování diagramu tříd a aktivit v fUmlStudiu.



Obrázek 7.2: Apache FreeMarker schéma (Převzato z http://freemarker.org/)

7.1 Implementace generátoru kódu

Generátor kódu je implementován jako konzolová Java aplikace. Samotné generování je specifikováno parametry při spouštění generátoru.

Generátor má následující spouštěcí parametry:

```
-l <path>    - cesta k fUML(ALF) knihovnám
-m <path>    - složka s modely
-n <string>  - jméno překládané jednotky
-o <path>    - cíl generování kódu
-p <string>  - prefix pro namespace
-t <path>    - hlavní freemarker šablona
-u <path>    - složka pro generování fUML mapování
```

Generátor pro načtení modelů používá referenční implementaci ALF, která ALF soubory namapuje na fUML model. fUML model ve vnitřní reprezentaci je předán obalovací třídě pro FreeMarker. Generování těl metod neprobíhá přes FreeMarker šablony, ale je implementováno přímo v generátoru, generování probíhá z vnitřní reprezentace fUML modelu.

Šablonovací systém

Pro definici generování kódu je použit šablonovací engine Apache FreeMarker [6]. FreeMarker je Java knihovna pro generování textových výstupů na základě šablon zapsaných v jazyce FreeMarker Template Language (FTL). Schéma funkce FreeMarkeru je naznačeno na obrázku 7.2. FTL je velice mocný šablonovací jazyk. FTL umožňuje definovat podmínky, cykly, proměnné, funkce definované v šabloně nebo v Javě a spoustu dalšího. Pro generování výstupu je nutné mít definované šablony a dodaný objekt poskytující data.

FreeMarker podporuje dvě skupiny typů objektů v modelu. První jsou skalární typy, konkrétně Boolean, Number, String a Date. Druhou jsou kontejnery a to Hash, Sekvence a Kolekce. Každý objekt používaný v šabloně musí být jedním z těchto typů.

Pro použití jiných objektů v modelu šablony je vytvořena obalovací třída, která implementuje rozhraní `TemplateHashModel` FreeMarkeru a jakýkoliv objekt zabalí pro potřeby FreeMarkeru, k vlastnostem objektu se přistupuje přes tečkovou notaci. Obalovací třída na základě obaleného typu a požadovaného atributu dodá příslušná data opět zabalená do obalovací třídy jeli to nutné. Základní Java typy jako String, Integer atd. jsou zabaleny do příslušného skalárního typu FreeMarkeru. Java seznamy a hashovací tabulky jsou zabaleny do příslušného kontejneru FreeMarkeru.

[6] Apache FreeMarker: http://freemarker.org/

Kód 7.1: Použití FreeMarker direktivy `createFile`

```
<@createFile
    filePath="src\main\java\${class.packagePath}"
    fileName="${class.name}.java">
        <Obsah souboru>
</@createFile>
```

Díky této obalovací třídě je možné v šablonách využívat libovolné objekty. V našem případě objekty vnitřní reprezentace fUML modelu.

Dostupné objekty ve FreeMarker šablonách jsou dokumentovány v příloze B, tabulka popisuje název objektu, jeho atributy a typy atributů. Symbol `[]` značí, že se jedná o kolekci.

FreeMarker má jako vstup šablonu a vstupní řetězec, jeho výstupem je také textový řetězec. Avšak pro účely generování kódu je nutné generovat i souborovou strukturu výsledných zdrojových kódů. Pro tyto potřeby byla implementována vlastní direktiva `createFile`, tato direktiva svůj obsah uloží do souboru v libovolné složce na disk. Direktiva `createFile` vyžaduje dva parametry a to `filePath` pro definici cílové složky a `fileName` pro definici názvu souboru. Příklad použití direktivy je vidět v úryvku kódu 7.1.

Transformační šablony jsou dostupné ve složce se zdrojovými soubory vzorového projektu. Vytvořené transformační šablony generují kód pro Android. Android aplikace jsou programovány v Javě. Díky tomu, že je kód generován z diagramu tříd a Java je objektově orientovaný jazyk, nepřináší tvorba šablon zásadní problém. Avšak rozdíly mezi možnostmi UML diagramu tříd a Java kódu existují. Například UML diagram tříd umožňuje více násobnou dědičnost oproti Javě. Proto nástroj nepodporuje generování vícenásobnou dědičnost v modelu.

Generování chování aktivit

Chování metod je v fUML definováno aktivitami. Chování metod, definované fUML aktivitami, není generováno pomocí FreeMarker šablon ale je implementováno přímo v generátoru kódu. Použití šablon pro generování aktivit by bylo implementačně náročné a nepřinášelo by pravděpodobně zásadní výhody.

Na každý uzel diagramu aktivit je nutné namapovat Java kód realizující stejnou funkci. Před generováním samotného uzlu aktivity je nutné mít vygenerovaný kód jeho vstupů. Generování Java kódu aktivit probíhá rekurzivně, než proběhne generování kódu daného uzlu, je generován rekurzivně kód vstupů do uzlu. Je to proto, aby generované řádky Java kódu byly ve správném pořadí. Nejprve musí být vyhodnoceny vstupy uzlu a až následně samotný uzel.

Pro výstupní parametry a nestrukturované uzly (nemají parametry ale tok je na ně připojen přímo) je definovaná v Javě proměnná s vygenerovaným názvem například `_tmp_12`. Typ proměnné je získán z typu parametru případně z typu vstupu do větvícího uzlu (fork node).

Třídy jsou do kódu generovány jménem včetně jmenného prostoru, není pak nutné řeši importy tříd z jiných balíčků. Při použití importů by bylo nutné řešit další problém s možným konfliktem jmen tříd.

Větvící uzel (fork node) s objektovými toky je v Jave implementován jako přiřazení proměnné vstupu do proměnných výstupu například takto `Integer _tmp_12 = _tmp_10;`.

Kód 7.2: ALF kód demontsrující použití podmínek

```
Integer a = 10;
if (a < 50) {
    WriteLine("if: IF");
} else if (a < 100) {
    WriteLine("if: ELSE IF");
} else {
    WriteLine("if: ELSE");
}
```

Větvící uzel s řídícím tokem popisuje větve prováděné paralelně. Generátor kódu kód paralelních větví generuje sériově za sebe v náhodném pořadí větví.

Uzel parametru (ActivityParameterNode) definuje vstupní a výstupní parametry aktivity, návratová hodnota je v aktivitě modelována také jako výstupní parametr. Pro parametry není generován žádný kód. Kde jsou parametry použity je použito jejich jméno. Při přiřazení návratové hodnoty je generován Java kód `return _tmp_12;`.

Uzel testující identitu (TestIdentityAction) má dva vstupy a jeden výstup typu boolean. Uzel testuje, zda vstupní parametry jsou identické objekty. Pro tento uzel je generován kód `Boolean _tmp_12 = _tmp_10 == _tmp_11`.

Uzel volající akci (CallAction) modeluje volání aktivit nebo operací. Uzel má návratovou hodnotu, jméno volané operace nebo aktivity a vstupní parametry. Pokud jsou volány základní operace jako například plus, je generován kód pro infixové volání `Integer _tmp_12 = _tmp_10 + _tmp_11;`. Pokud se jedná o volání operace objektu je genrován například kód `_tmp_10.operace(_tmp_11);`. Volání operace `WriteLine(_tmp_10);` je pro Android generováno jako logování funkcí `Log.d("fUml", _tmp_10);`.

Uzel podmínky (ConditionalNode) v diagramu aktivit definuje podmíněný tok s libovolným množstvím aletrnativ. Tento uzel má libovolný počet klauzulí, každá klauzule má na vstupu datový tok typu boolean, který udává, zda je klauzule splněna u bude prováděno její tělo. Kód pro první alternativu je generován například takto `if (_tmp_11)`, další alternativy jsou generovány jako `else if (_tmp_12)`. Za kód podmínky je generován kód těla klauzule. Příklad ALF kódu a odpovídajícího vygenerovaného Java kódu je ukázán na částech kódu zde 7.2 a zde 7.3.

Uzel cyklu (LoopNode) v fUML aktivitě řeší chování cyklů. Tento uzel obsahuje vstup podmínky, inicializační část a prováděné tělo. Tento uzel obsahuje příznak, zda prvnímu spuštění těla předchází vyhodnocení podmínky. Jsou definovány proměnné uzlu, které musejí být po provedení těla aktualizovány na nové hodnoty. Tělo uzlu je prováděno cyklicky dokud podmínka platí. Na tento uzel jsou převáděny všechny druhy cyklu z jazyka ALF `while`, `do while` i `for`.

Z tohoto uzlu je generován v Javě while cyklus s podínkou nastavenou na true. Ukončení cyklu je řešeno příkazem break. Je to z důvodu zjednodušení generování kódu. Inicializační část je vložena před generovaný while cyklus. Vyhodnocování podmínky je vloženo na začátek těla generovaného cyklu a podmínka je vyhodnocena podmínkou, pokud neplatí je volán příkaz break. Za podmínku je generováno tělo cyklu. Pokud se jedná a cyklus s vyhodnocením na konci, je pouze přesunuta ukončovací podmínka na konec těla cyklu. Na konci těla cyklu musí být přiřazení nových hodnot do proměnných cyklu. Generátor kódu podporuje ALF cykly while a do-while, cyklus for není v generátoru implementován. Pří-

Kód 7.3: Generovaný Java kód podmínek odpovídajících podmínek

```
Integer _tmp_55 = 10;
Boolean _tmp_56 = _tmp_55 < 50;
Boolean _tmp_57 = _tmp_55 < 100;
if (_tmp_56) {
    Log.d("fUml", "if: IF");
}
else if (_tmp_57) {
    Log.d("fUml", "if: ELSE IF");
}
else if (true) {
    Log.d("fUml", "if: ELSE");
}
```

Kód 7.4: ALF kód demontsrující použití for cyklu

```
Integer i = 0;
while(i < 10) {
    WriteLine("While: " + IntegerFunctions::ToString(i));
    i = i + 1;
}
```

klad generování kódu z cyklu je demonstrován na příkladu, původní ALF kód 7.4 generuje Java kód 7.5.

Uzel čtení ze struktury (ReadStructuralFeatureAction) slouží pro načtení vnitřní položky z objektu. Do jazyku Java je tato funkce generována jako tečková notace pro přístup k atributům objektu. Může to být například takový kód `Integer _tmp_2 = _tmp_1.cislo;`.

Uzel pro zapsání (AddStructuralFeatureValueAction) slouží naopak pro zápis do vnitřní položky objektu, vygenerovaný kód může vypadat například následovně `_tmp_2.cislo = _tmp_1;`

Mapování z ALF na fUML generuje mnoho nadbytečných uzlů do diagramu aktivit. Proto výsledný Java kód obsahuje také mnoho nadbytečného kódu a není příliš čitelný. Jména proměnných, kromě parametrů, jsou ztracena a v kódu jsou pouze nic neříkající generované názvy proměnných.

Generátor kódu podporuje jen podmnožinu možností ALF a fUML nezbytnou ke generování kódu Android aplikací. Generátor kódu například vůbec nepodporuje signály a receptory, kterými je možné modelovat asynchronní volání. Také generické typy nejdou podporovány. Aktivní třídy jsou generovány jako klasické, jejich aktivita není generována. Také nejsou podporovány enumerátory a asociace, asociace je možné nahradit pomocí atributů a asociačních tříd. Dále nejsou podporovány operátory redukce.

Persistentní třídy

Persistentní třídy jsou odlišeny od ostatních tříd tím, že dědí z knihovní třídy `PersistentLibrary.Persistent`. V generačních šablonách je možné detekovat, zda se

Kód 7.5: Odpovídající generovaný Java kód cyklu

```
//inicializace proměnných
Integer _tmp_7 = 0;
Integer _tmp_8 = _tmp_7;
while (true) {
    //vyhodnocení podmínky
    Integer _tmp_9 = _tmp_8;
    Boolean _tmp_10 = _tmp_9 < 10;
    if (!(_tmp_10)) break;

    //tělo cyklu
    String _tmp_11 = String.valueOf(_tmp_9);
    String _tmp_12 = "While: " + _tmp_11;
    Log.d("fUml", _tmp_12);
    Integer _tmp_13 = _tmp_9 + 1;
    Integer _tmp_14 = _tmp_13;

    //aktualizace proměnných cyklu pro další iteraci
    _tmp_8 = _tmp_14;
}
```

jedná o třídu dědící z této knihovní třídy a tomu generování přizpůsobit. Použití dědičnosti také umožňuje používat metody rodičovské třídy.

Odlišení tříd bylo možné řešit pomocí stereotypů, v tomto řešení se ale objevil problém, mapování ALF kódu na fUML nemapuje stereotypy. Proto není možné detekovat jaký stereotyp byl u třídy použit.

Persistentní třídy jsou implementovány jako Sugar ORM tabulky, Sugra ORM ukládá data do SQLite databáze vestavěné v systému Android. Generační šablony negeneruje přímo kód SQL dotazů u persistentních tříd pro práci s vestavěnou SQLite databází. Nástroj využívá pro práci s databází Objektově relační mapování (ORM) konkrétně SugarORM [7]. Při použití SugarORM není nutné psát SQL dotazy, stačí pouze dědit z generické třídy `SugarRecord` případně vhodně anotovat atributy. Pak se používají metody samotného SugarORM pro práci s daty databáze.

Pro vložení nového záznamu je nutné konstruktorem vytvořit nový objekt, nastavit hodnoty atributů a zavolat metodu `Save` zděděnou ze třídy `SugarRecord`. Tím vznikne v databázové tabulce nový záznam. Pro načtení všech záznamů z tabulky do seznamu slouží metoda `listAll`. Smazání načtené položky se provádí metodou `delete`. Sugar ORM umožňuje načítaná data vyhledávat podle zadaných parametrů metodou `find` nebo jejich id metodou `findById`.

Příkladem je modelovaná persistentní třída v jazyce ALF 7.6 a odpovídající generovaný Java kód 7.7.

[7] Webové stránky Sugar ORM frameworku: http://satyan.github.io/sugar/

Kód 7.6: Příklad modelování persistentní třídy v ALF

```
namespace todo;

class Item specializes PersistentLibrary::Persistent {
    public label: String;
}
```

Kód 7.7: Vygenerovaný Java kód persistentní třídy

```
package net.belehradek.generated.todo;

import com.orm.SugarRecord;

public class Item extends SugarRecord<Item> {
        public String label;
}
```

Třídy aktivit

Třídy Android aktivit jsou odlišeny od ostatních tříd tím, že dědí z knihovní třídy `ActivityLibrary.Activity`. Toto je možné v generačních šablonách detekovat a generovat tak kód pro třídu Android aktivity.

Dále je nutné definovat jakým způsobem budou modelovány přechody mezi aktivitami. Přechod na další aktivitu bude modelován operací ve spouštěcí aktivitě. Odlišena bude od ostatních operací definovaným složením parametrů operace. První parametr bude typu podle cílové aktivity a násobnost parametru bude `[0]`, parametr neslouží k přenosu objektu ale pouze k definování třídy cílové aktivity. Druhým parametrem je pole textových řetězců pro přes parametrů do spouštěné aktivity. V těle této metody je v ALF kódu volána metoda pro spuštění nové aktivity kvůli přímému spouštění modelu. Ve výsledném Android kódu spouštění nové Android aktivity řešeno vygenerovaným kódem.

Pře přechod na aktivitu je operace přechodu zavolána s prvním parametrem `null` a ve druhém jsou předány parametry spouštění.

Příklad generovaného kódu aktivity 7.9 je ukázán na příkladu a zdrojový kód ALF modelu je naznačen zde 7.8.

7.2 Implementace Gradle pluginu

Gradle je nástroj na automatizaci téměř libovolných úkolů. Jeho hlavní výhoda je v jeho univerzálnosti, je založen na jazyce Groovy, tím je možné specifikovat jakýkoliv automatizovaný úkol (task). Nejčastěji je Gradle využíván na sestavování softwarových projektů.

Gradle umožňuje vytvářet a používat vlastní pluginy. Pluginy je možné programovat v jakémkoliv jazyce podporovaném v JVM. Pro implementaci fUmlGradlePluginu byl zvolen jazyk Groovy. Projekt Gradle pluginu je také sestavován Gradlem, pro kompilaci pluginu a nahrání do repozitáře slouží task `gradle uploadArchives`. Při použití pluginu je nutné definovat repozitář, ve kterém se plugin nachází.

Kód 7.8: Příklad modelování třídy aktivity v ALF

```
namespace todo;

class HelloActivity specializes ActivityLibrary::Activity {
    //metoda přechodu na jinou aktivitu
    public goToListActivity(in toStart: ListActivity[0],
            in params: String[*]) {
        this.startActivity(new ListActivity(), params);
    }

    //metoda volaná při startu aktivity
    public onStart(in params: String[*]) {
        WriteLine("HelloActivity start!");
        this.goToListActivity(null, null);
    }
}
```

Dva základní soubory v Gradle projektu jsou `build.gradle` a `settings.gradle`. Soubor `build.gradle` obsahuje definici Gradle buildu. Obsahuje například použité pluginy, repozitáře, závislosti a další nastavení. Konfigurační soubor `setting.gradle` obsahuje nastavení proměnných jako například název projektu.

Pro vývoj Gradle pluginu je nutné nejprve inicializovat nový projekt podle návodu [8].

Úkony vytvářeného gradle pluginu využadují ke své funkci externí jar soubory generátoru kódu a referenčních implementací ALF a fUML, tyto jar soubory jsou umístěny ve složce [9] pluginu. Task [10] jar soubory extrahuje to složky projektu tak, aby mohly být z gradle pluginu spouštěny.

Gradle plugin využívá konfigurovatelné hodnoty ze settings.gradle, aby byla hodnota korektně načte, musí být její hodnota předána do pluginu v souboru build.gradle. Nastavitelné parametry gradle pluginu:

```
gradle.ext.FumlSettingsLibPath = '<path>'
      - cesta k fUML(ALF) knihovnám
gradle.ext.FumlSettingsToolsPath = '<path>'
      - cesta k nástrojům (jar soubory)
gradle.ext.FumlSettingsUnitName = '<string>'
      - jméno hlavní jednotky
gradle.ext.FumlSettingsTransformationFile = '<path>'
      - hlavní freemarker template
gradle.ext.FumlSettingsNamespacePrefix = '<string>'
      - prefix pro namespace
```

Implementovaný gradle plugin poskytuje funkce, které poskytují vše potřebné pro vývoj aplikací. V následující části jsou popsány jednotlivé funkce pluginu.

[8] Gradle Writing Custom Plugins https://docs.gradle.org/current/userguide/custom_plugins.html
[9] resources
[10] fUmlInstall

Kód 7.9: Vygenerovaný Java kód třídy aktivity

```
package net.belehradek.generated.todo;

import ...;

public class HelloActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(getLayoutId());

        //načtení parametru a volání metody onStart
        Intent intent = getIntent();
        String[] params =
            intent.getStringArrayExtra("fUmlActivityParams");
        onStart(params);
    }

    //metoda pro načtení zobrazeného layoutu
    protected int getLayoutId() {
        int id = getResources().getIdentifier("_hello_activity",
            "layout", getPackageName());
        return id;
    }

    //metoda přechodu na jinou aktivitu
    public void goToListActivity (ListActivity toStart,
            String[] params) {
        Intent intent = new Intent(this, ListActivity.class);
        intent.putExtra("fUmlActivityParams", params);
        startActivity(intent);
    }

    //metoda volaná při startu aktivity
    public void onStart(String[] params) {
        String[] _tmp_0 = params;
        Log.d("fUml", "HelloActivity start!");
        goToListActivity(null, null);
    }
}
```

`gardle fUmlInstall`

Extrahuje do složky projektu soubory potřebné pro správnou funkci fUmlGradlePluginu. Do složky fUmlLibrary je nakopírována základní knihovna funkcí ALF/fUML. Do složky fUmlTools jsou nakopírovány potřebné jar soubory referenční implementace a jar soubor generátoru kódu.

`gradle fUmlClean`

Tento task maže ze složky projektu složky build, out a uml. Tyto složky obsahují soubory vznikající při mapování ALF kódu na fUML nebo při generování kódu.

`gradle fUmlCodeGenerate`

Task fUmlCodeGenerate spouští generování kódu. Spouští soubor fUmlCodeGenerator.jar s defaultnímy parametry nebo s parametry převzatými ze souboru settings.gradle. Vygenerované soubory jsou defaultně umístěny ve složce out.

`gradle fUmlCompile`

Spouští kompilaci ALF souborů. Spouští soubor alf-eclipse.jar, výstupem je fUML model ve formátu XMI defaultně ve složce uml.

`gradle fUmlRun`

Tento task slouží pro spuštění vygenerovaného fUML modelu taskem fUmlCompile. Samotné spuštění modelu zajišťuje referenční implementace souborem fuml-eclipse.jar. Vstup a výstup spuštěné aplikace je realizován přes standardní vstup a výstup příkazové řádky.

Příklad konkrétního použití pluginu je popsán v kapitole Tvorba vzorové Android aplikace 8.

7.3 Implementace vývojového prostředí

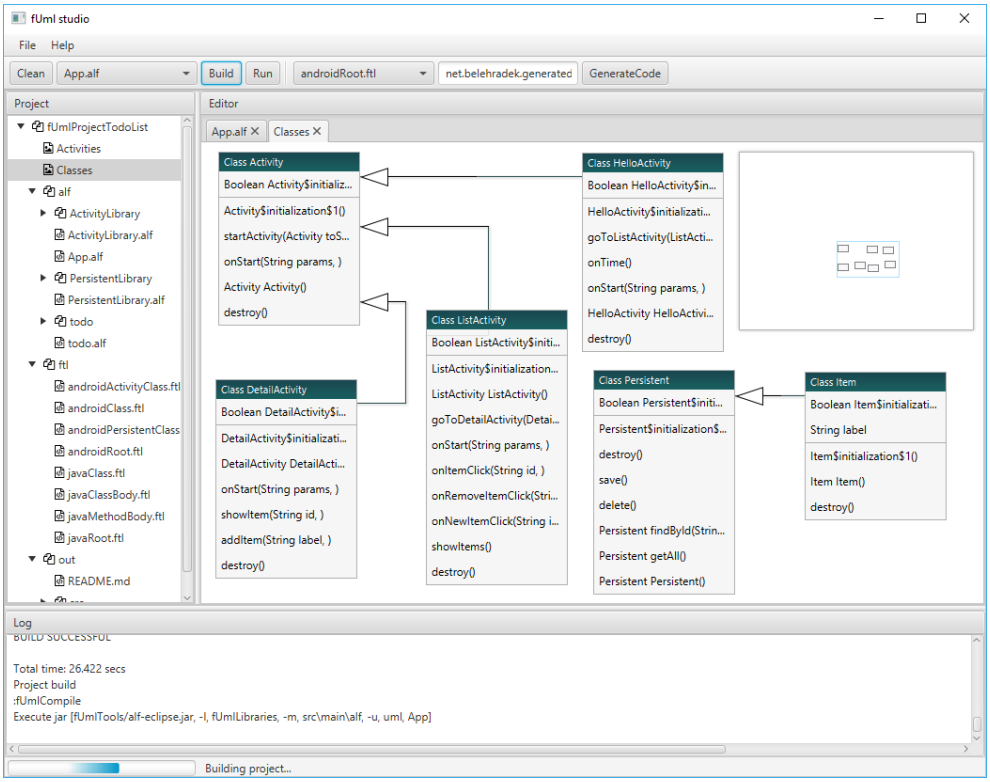
Pro usnadnění vývoje je vytvořeno i vývojové prostředí. Vytvářené vývojové prostředí je pojmenováno fUmlStudio. Vývojové prostředí je implementováno jako desktopová JavaFX aplikace. Díky implementaci pro Java Virtual Machine (JVM) je možné ji spouštět na různých platformách podporujících JVM.

Grafické uživatelské rozhraní vývojového prostředí se skládá z několika částí. GUI je vidět na obrázku 7.3. V horní části je menu pro správu projektů a souborů. Lišta s ovládacími prvky pro Gradle plugin je umístěna pod menu, pomocí této lišty lze zcela ovládat gradle fUmlPlugin. V levé části se nachází průzkumník projektu, zobrazující strukturu právě otevřeného projektu ve formě stromu. V pravé části je okno záložek pro zobrazení otevřených souborů. Soubory jsou zobrazeny v textovém či grafickém editoru podle typu. Ve spodní části se nachází panel logu, do kterého je přesměrováno veškeré logováni. Pod panelem logu se nachází stavová lišta zobrazující popisek právě probíhajícího úkolu a jeho stav.

Ke spouštění Gradle tasků z fUmlStudia je použito

Vývojové prostředí využívá Gradle plugin pro čištění složek projektu, kompilaci, spouštění ALF kódu a generování kódu. Gradle poskytuje tzv. Gradle Tooling API [11] pro použí-

[11] Gradle Tooling API: https://docs.gradle.org/current/userguide/embedding.html



Obrázek 7.3: fUmlStudio GUI

vání spouštění Gradle tasků v Java aplikacích. Pomocí roletového menu v liště nástrojů je možné vybrat z existujících ALF souborů, který bude defaultní a bude použit při kompilaci, spouštění a generování kódu. Druhé roletové menu poskytuje možnost výběru defaultní transformační šablony pro generování kódu. Zadávací textové pole umožňuje definovat prefix prostoru jmen při generování kódu. Při kompilaci ALF kódu je defaultní namespace `Model`, tato volba umožňuje defaultní prefix prostoru jmen změnit. Při změně nějaké z uvedených voleb je nové nastavení uloženo do souboru `settings.gradle`, kde jsou tato nastavení definována.

Při vytvoření nového projektu nebo při otevření existujícího je do správce projektu načtena struktura projektu. fUmlStudio prohledá příslušné složky a zobrazí soubory ve strumu. Kořen stromu je pojmenován podle jména projektu a obsahuje virtuální položku pro zobrazení kompletního diagramu tříd a aktivit. Dále obsahuje položku pro soubory ALF a položku pro soubory FTL. Také je zobrazována položka pro zobrazení vygenerovaných souborů. Průzkumník projektu neumožňuje mazat nebo přejmenovávat soubory, tato akce je nutné dělat jiným způsobem.

Při kliknutí na soubor v průzkumníku projektu je příslušný soubor otevřen v odpovídajícím editoru a editor je zobrazen jako nová záložka v okně záložek. Pokud byl již soubor otevřen je pouze příslušná záložka aktivována. Záložky je možné zavírat kliknutím na křížek u záložky.

Pro úplný přehled o spouštění Gradle tasků a jiných akcí v fUmlStudiu je ve spodní části umístěn panel logu zobrazující výstup ze spouštěných Gradle tasků i logovacích informací fUmlStudia.

Některé spouštěné úkoly v fUmlStudiu jsou výpočetně náročné a jejich provedení trvá delší dobu. Aby celé vývojové prostředí při spuštění těchto úkolů nezamrzlo, jsou tyto úkoly spouštěny ve zvláštním vlákně. Tyto úkoly spouštěné na pozadí jsou implemntovány děděním z abstraktní třídy `IdeWorkerTask`, task je definován popiskem a implementací abstraktní metody `work`, které je spouštěna. `IdeWorker` slouží jako správce těchto tasků, samotné tasky spouští a také informuje o jejich spuštění a dokončení. `IdeWorker` je implementován jako návrhový vzor singleton. Na tyto události reaguje fUmlStudio zobrazením průběhu úkoly a jeho názvu ve stavové liště.

Implementace textového a grafického editoru

Vývojové prostředí fUmlStudio poskytuje podporu editace několika typů souborů. Při kliknutí na soubor v průzkumníku projektu, je podle typu souboru vybrán příslušný editor.

Pokud se jedná o soubor s příponou `.alf` nebo `.ftl`, je soubor otevřen v editoru kódu. Pokud se jedná o jiný textový soubor, je otevřen v textovém editoru.

Pro zobrazení diagramu tříd a diagramu přechodů aktivit je používán grafický editor.

Textový editor je implementovaný jako JavaFX `TextArea`, které je nastaveno monospace písmo. Při otevření souboru je obsah souboru načten do řetězce a ten je vložen do editoru textu. Při ukládání je naopak textovým řetězcem přepsán obsah souboru.

Jako editor kódu je použit existující editor z Eclipse JFace UI toolkitu [12] `SourceViewer`.

Vývojové prostředí používá grafický editor pro zobrazení diagramu tříd a diagramu Android aktivit. Grafický editor je založen na knihovně Graph Editor [13]. Graph Editor je univerzální knihovna pro tvorbu diagramů v prostředí JavaFX, umožňuje tvorbu grafických uzlů, jejich konektorů a propojení mezi konektory.

Knihovna Graph Editor nemá žádnou podporu UML diagramů. Bylo nutné implementovat vzhled pro elementy používané v UML a styl jejich propojení. Defaultní skiny v knihovně Graph Editor mají nastavitelnou velikost uživatelem nikoliv automatickou podle obsahu uzlu. Proto je vytvořen nový typ uzlu, který má automatickou velikost a zachovává ostatní původní vlastnosti uzlu, jako například přesouvání pomocí myši. Byl vytvořen skin pro zobrazení třídy, umožňující do skinu vkládat atributy a operace třídy. Dále byl vytvořen skin pro zobrazení balíčku obsahující jen v titulku jméno balíčku. Pro zobrazení dědičnosti v diagramu tříd bylo nutné implementovat skin zobrazující propojení uzlů se šipkou.

Pro zobrazení diagramů je nutné načíst fUML model modelované aplikace. Model je načítán pomocí ALF referenční implementace do její vnitřní reprezentace. Jsou načteny všechny třídy kromě knihovních a pokud pro ně neexistují uzly jsou automaticky vytvořen v defaultním umístění. Pokud existuje v grafu uzel a již pro něj neexistuje třída je uzel odstraněn. Pokud třída dědí z jíné třídy, je tato rodičovská třída načtena a je vytvořeno propojení zobrazující dědičnost. Při zobrazování diagramu aktivit jsou načteny jako aktivity pouze třídy dědící z knihovní třídy `ActivityLibrary.Activity`. Zobrazení šipek mezi aktivitami, znázorňující přechod aktivit, je načteno ze tříd aktivit. Metody mající první parametr typu dědícího z `ActivityLibrary.Activity` a násobnosti `0..0` a druhý parametr typu `String[*]` jsou přechody aktivit. Podle typu prvního parametru je definována cílová aktivita.

Vlastnosti diagramu, jako pozice uzlů a jejich propojení, jsou ukládány v souboru .graph. Pokud je poprvé otevřen náhled tříd aplikace, je vytvořen prázdný .graph soubor, následně jsou automaticky vytvořeny uzly pro existující třídy a vygenerováno jejich propojení. Uzly

[12] Eclipse JFace: https://wiki.eclipse.org/JFace
[13] Graph Editor: https://github.com/tesis-dynaware/graph-editor

jsou jednoznačně identifikovány kvalifikovaným jménem třídy, kterou reprezentují. Při opětovném otevření souboru tak možné namapovat existující uzly na třídy a detekovat případné chybějící nebo nadbytečné uzly. Nadbytečné jsou odstraněny a chybějící vytvořeny. Následně je možné upravovat pozice uzlů a propojovacích čar a poté je uložit. Při následném diagramu tříd jsou již načteny pozice uzlů. Pokud je v diagramu nová třída, je generován nový uzel na defaultní pozici.

Kapitola 8

Tvorba vzorové Android aplikace

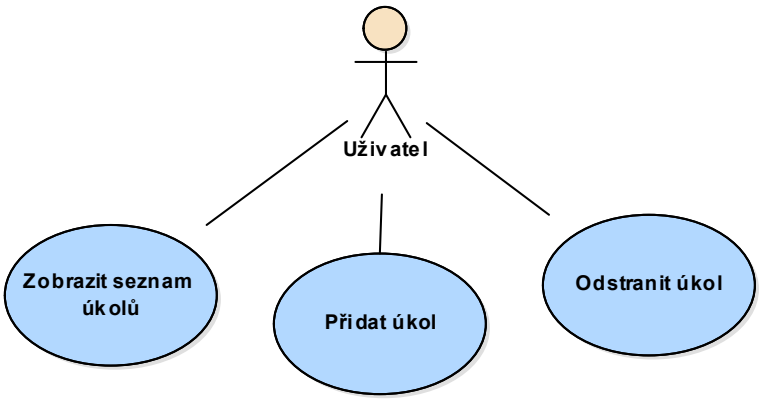
Pro prezentaci funkcí a možností vyvíjeného nástroje je v této kapitole popsána tvorba vzorové aplikace. Jako vzorová aplikace je zvolen jednoduchý správce úkolů neboli anglicky Todo list. Smyslem této vzorové aplikace je představit možnosti vyvíjeného nástroje a nikoliv vytvořit dokonalou aplikaci pro správu úkolů.

Kompletní zdrojové kódy implementované vzorové aplikace jsou dostupné na GitHubu, fUML projekt je dostupný zde [1] a Android projekt zde [2]. Zdrojové kódy i zkompilovaná vzorová aplikace TodoList je uložena na přiloženém DVD.

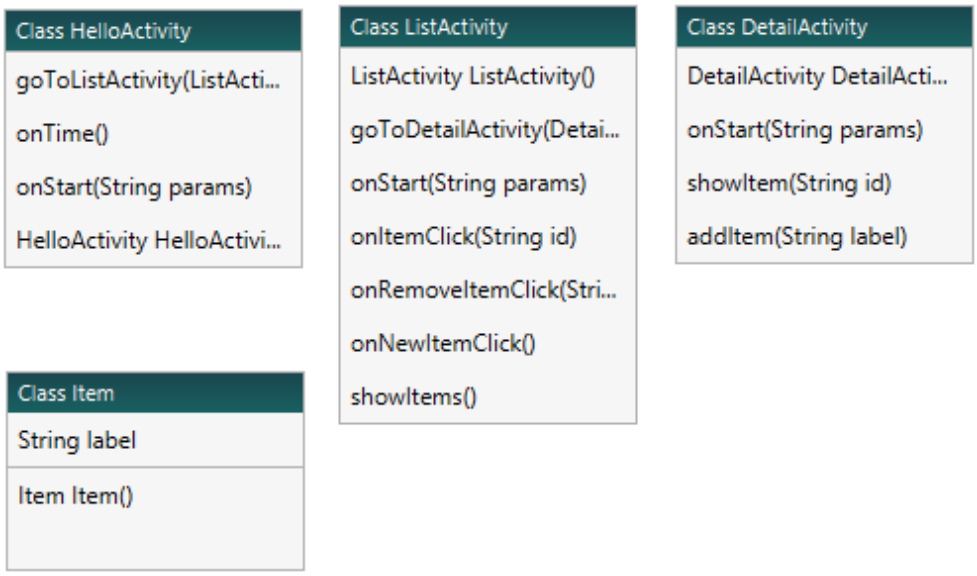
Správce úkolů bude umožňovat zobrazit úkoly, úkol přidat nebo existující odstranit. Případy užití aplikace jsou znázorněny v diagramu případů užití 8.1.

Správce úkolů se bude skládat ze tří aktivit. Po spuštění aplikace se zobrazí uvítací aktivita `HelloActivity` zobrazující základní informace o aplikaci jejím smyslu. Po definovaném časovém intervalu se přejde na aktivitu `ListActivity` se seznamem úkolů, na které budou zobrazeny všechny nedokončené úkoly jako seznam. Každý úkol v seznamu bude obsahovat tlačítko, pro označení, že je úkol dokončen. Ve spodní části bude plovoucí tlačítko se symbolem `+`, po kliknutí na toto tlačítko se zobrazí aktivita pro zadání nového úkolu `DetailActivity`. DetailActivity bude obsahovat textové pole pro zadání popisku úkolu a tlačítko pro potvrzení úkolu. Zrušení vytváření úkolu bude možné tlačítkem zpět v prostředí Android.

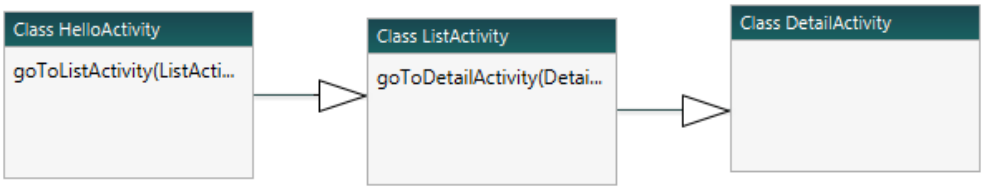
[1] GitHub fUmlProjectTodoList: https://github.com/Standa631/fUmlProjectTodoList
[2] GitHub fUmlAndroidTodoList: https://github.com/Standa631/fUmlAndroidTodoList



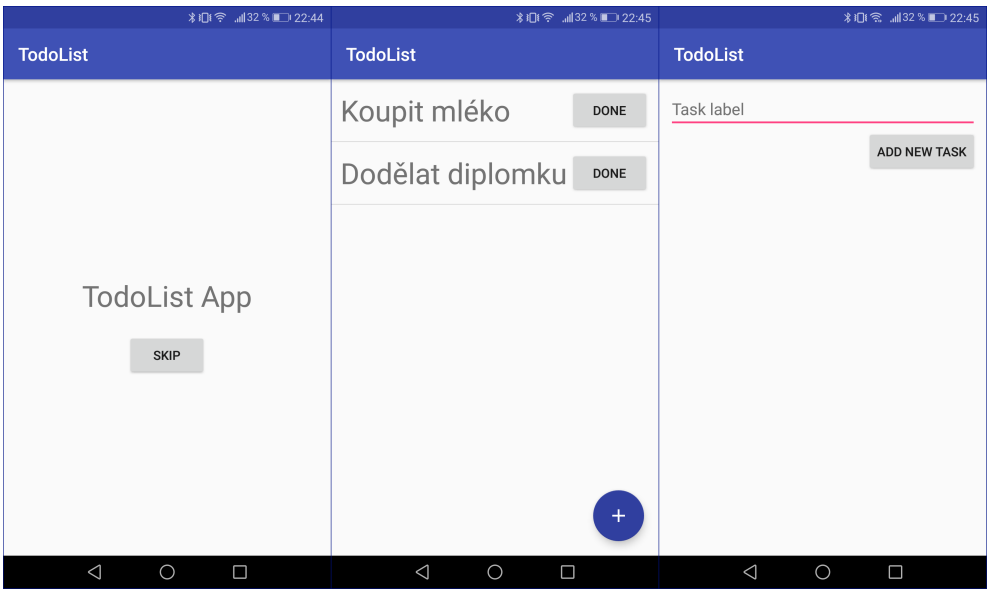
Obrázek 8.1: Případy užití vzorové aplikace TodoList



Obrázek 8.2: Diagram tříd vzorové aplikace TodoList



Obrázek 8.3: Diagram aktivit vzorové aplikace TodoList



Obrázek 8.4: Výsledné obrazovky vzorové aplikace TodoList

Kód 8.1: Příklad použití fUmlGradlePluginu v projektovém souboru build.gradle

```
apply plugin: 'FumlPlugin'

buildscript {
    repositories {
        maven {
            url uri('../repo') //loklani repozitar
        }
    }
    dependencies {
        classpath group: 'net.belehradek.fuml.gradle',
            name: 'FumlPlugin', version: '0.1'
    }
}
```

Kód 8.2: Příklad konfigurace fUmlGradlePluginu v souboru settings.gradle

```
gradle.ext.FumlSettingsLibPath = 'fUmlLibraries'
gradle.ext.FumlSettingsToolsPath = 'fUmlTools'
gradle.ext.FumlSettingsUnitName = 'App'
gradle.ext.FumlSettingsTransformationFile = 'root.ftl'
gradle.ext.FumlSettingsNamespacePrefix = 'generated'
```

Příprava projektu

Pro používání gradle pluginu je nutné jej zkompilovat do lokálního repositáře příkazem `gradle uploadArchives`, v souboru `build.gradle` je specifikovaná cesta k lokálnímu repositáři.

Nejprve je nutné založit nový projekt pro novou aplikaci. Založit nový projekt je možné ručně s pomocí Gradle pluginu nebo automatizovaně v fUmlStudiu. V fUmlStudiu stačí kliknout na položku menu `File->New->New` project a následně v dialogovém okně vyplnit název a umístění nového projektu. Pro vytvoření nového projektu bez použití fUmlStudia je nutné spustit `gradle init` pro vytvoření základní gradle struktury projektu. Následně je nutné do souboru build.gradle doplnit lokální repozitář, závislost na fUmlGradlePluginu a plugin aplikovat, což je vidět na kódu zde 8.1.

Nastavitelné parametry je možné nastavit v settings.gralde viz 8.2. Aby byly parametry aplikovány do pluginu je nutné je načíst i v souboru gradle.build a předat pluginu jak je vidět zde 8.3.

Po inicializaci projektu je možné vytvářet ALF zdrojové soubory i transformační FreeMarker šablony. Tyto soubory je možné vytvořit ve správci souborů nebo z fUmlStudia přes menu `New-New source file` nebo `New-New transformation file`.

Tvorba ALF modelu

Pro správné vytvoření třídy umístěné v balíčku je nutné udělat několik kroků. Zaprvé vytvořit soubor pojmenovaný názvem třídy s příponou `.alf`, soubor umístit do složek podle

Kód 8.3: Příklad načtení konfigurace fUmlGradlePluginu v souboru build.gradle

```
fUmlSettings {
    libPath = gradle.ext.FumlSettingsLibPath
    toolsPath = gradle.ext.FumlSettingsToolsPath
    unitName = gradle.ext.FumlSettingsUnitName
    transFile = gradle.ext.FumlSettingsTransformationFile
    namespacePrefix = gradle.ext.FumlSettingsNamespacePrefix
}
```

Kód 8.4: Příklad ALF souboru třídy

```
namespace JmenoBalicku;
class TridaVBalicku {
}
```

jmenného prostoru třídy. Soubor třídy může například vypadat takto 8.4. Například soubor třídy v balíčku jménem `JmenoBalicku` musí být umístěn ve složce se stejným jménem umístěné ve složce se zdrojovými soubory. Balíčky je možné hierarchicky uspořádávat. Třídu je možné definovat přímo v souboru balíčku a není tak nutné vytvářet další soubor pro třídu.

Pro každý balíček musí existovat soubor se jménem balíčku popisující strukturu balíčku s příponou `.alf`. Takový soubor může vypadat například takto 8.5.

Pro použití třídy z jiného balíčku, než je aktuální namespace, je nutné definovat v souboru import třídy. Například je to možné takto public `import JmenoBalicku::TridaVBalicku;`. Při importu je možné použít místo názvu třídy hvězdičku, pak jsou importovány všechny třídy balíčku.

Základní platformě nezávislý kód je implementován ALF kódem. Jsou to tři aktivity a přechody mezi nimi.

Modelování aktivit

Aktivitu spouštěnou při startu aplikace je nutné vytvořit a spustit v aktivní třídě App v souboru `App.alf` jak je vidět zde 8.6. Díky tomu je aktivita spuštěna při startu simulace a je možné tak částečně testovat vyvíjenou aplikaci bez nutnosti generování kódu. Ladící informace je možné tisknout na standardní výstup případně do panelu logu v fUmlStudiu příkazem `WriteLine("App start");`.

V jazyce ALF jsou definovány tři aktivity. Z HelloActivity je definován přechod na ListActivitu a z ListActivity na DetailActivitu. Třídy Android aktivit jsou od ostatních tříd odlišeny dědičností z knihovní třídy `ActivityLibrary.Activity`. Přechody jsou definovány metodami v ALF kódu. Jméno metody je libovolné ale typ prvního parametru musí být

Kód 8.5: Příklad ALF souboru balíčku

```
package JmenoBalicku {
    public class TridaVBalicku;
}
```

Kód 8.6: Přidání startovací aktivity do ALF kódu

```
public import ActivityLibrary::Activity;
public import todo::HelloActivity;

active class App {

} do {
    WriteLine("App start");
    //načtení a spuštění první aktivity
    Activity ac = new Activity();
    ac.startActivity(new HelloActivity(), null);
    WriteLine("App end");
}
```

typ spouštěné aktivity s násobností `0..0`, tento parametr pouze definuje typ cílové aktivity, není použit pro přenos objektu, proto tato násobnost. Druhým parametrem metody je pole typu String pro přenos parametrů do spouštěné aktivity. V aktivitách je možné přepsat metodu `onStart(in params: String[*])`, která je volána při startu aktivity a v parametru obsahuje seznam textových parametrů, předaný při volání přechodu na aktivitu.

Modelování persistentních tříd

Úkoly jsou v aplikaci ukládány persistentně, proto třída `Item` dědí z knihovní třídy `PersistentLibrary::P`
Třída `Item` obsahuje jeden atribut `label` typu `String` reprezentující jméno úkolu. Persistentní třída poskytuje metody pro načtení konkrétního záznamu podle jeho číselného identifikátoru metodou `findById`. Pro načtení všech záznamů v tabulce slouží metoda `listAll`. Načtený záznam je možné smazat zavoláním metody `delete`. Nový objekt se vkládá metodou `Save`.

Přenos do Android projektu

Do build.gradle je nutné přidat závislost na Sugar ORM
`compile 'com.github.satyan:sugar:1.3'`, které je využíváno u persistentních tříd.

Pokud není generování kódu integrováno do buildcyklu Android aplikace je nutné po naprogramování aplikace v ALF kódu spustit generování kódu taskem `fUmlCodeGenerate` fUmlGradlePluginu nebo ve vývojovém prostředí tlačítkem `Generate code` v liště nástrojů. Pokud nebyl kód generován přímo do složky Android projektu je nutné vygenerované soubory překopírovat.

Startovací aktivitu je nutné definovat v cílovém Android projektu v souboru `AndroidManifest.xml`, příklad vložené aktivity to manifestu je zde 8.7.

Do manifestu je také nutné přidat všechny používané aktivity
`<activity android:name="net.belehradek.generated.todo.HelloActivity"/>`.

Některé metody jsou určeny pro přepsání v nativní kódu, tyto metody by bylo vhodné označit v ALFu jako abstraktní ale pak by nebylo možné vytvářet instance těchto tříd a ALF kód spouštět. Proto tyto metody nejsou abstraktní, ale neobsahují implementaci. Přechod na ListActivity je inicializován v metodě `onTime`. ALF neobsahuje časové zpoždění, proto je zpoždění implementováno v Javě přepsáním metody `onTime` jak je vidět zde 8.8.

Kód 8.7: Přidání startovací aktivity do manifestu

```
<activity android:name=".HelloActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER"
        />
    </intent-filter>
</activity>
```

Kód 8.8: Přepsání metody `onTime`

```
@Override
public void onTime() {
    final Handler handler = new Handler();
    handler.postDelayed(new Runnable() {
        @Override
        public void run() {
            super.onTime();
        }
    }, 5000);
}
```

Pro změnu chování vygenerovaného kódu je nutné z vygenerované třídy dědit a v potomkovi metody určené ke změně přepsat (Override). Je možné používat původní vygenerovanou metodu, v přepisující i jiné metodě použitím klíčového slova `super`.

Automaticky vygenerovaný layout aktivity obsahují pouze popisek s názvem aktivity. Pro zobrazení jiné než automaticky vygenerovaného layoutu, je možné vygenerovanou metodu `getLayoutId` přepsat a vrátit v ní identifikátor nového layoutu, jak vidět zde 8.9.

Po naprogramování platformě závislého kódu Android aplikace je možné aplikaci v Android studiu přeložit a spustit.

Kód 8.9: Přepsání metody pro získání zobrazovaného layoutu

```
@Override
protected int getLayoutId() {
    return R.layout.hello_activity;
}
```

Kapitola 9

Zhodnocení vytvořeného nástroje a možnosti dalšího vývoje

V této kapitole je nejprve zhodnocen vytvořený nástroj a jsou diskutovány jeho výhody a nevýhody. V druhé části této kapitoly jsou uvedeny možnosti dalšího vývoje a jsou navržena vylepšení nástroje.

Zhodnocení vytvořeného nástroje

Implementovaný nástroj je funkční a v něm vymodelované modely je možné přímo spouštět. Z modelů je možné generovat funkční kód Android aplikace. Implementovaný nástroj se skládá z několika částí, které společně tvoří komplexní nástroj. Vývoj nástroje byl časově náročný. Z důvodu nedostatku času není vyladěno generování Java kódu těl metod a vývojové prostředí neumožňuje takové funkce jako dostupná vývojová prostředí.

Výhodou nástroje proti klasickému vývoji Android aplikace v Android Studiu je grafické zobrazení modelů, které je v některých případech přehlednější. Přechody mezi Android aktivitami je také možné zobrazit graficky v diagramu. Další výhodou je modelování vyšší úrovni abstrakce a snadnější přechod aplikace na jinou platformu.

fUmlStudio obsahuje jen nezbytné funkce pro správu projektů a souborů, pro pohodlnější vývoj aplikací by bylo užitečné množství funkcí rozšířit. Například ve stromovou strukturu rozšířit o kontextové menu pro další akce se soubory například mazání a přejmenovávání souborů.

Značnou výhodou je implementace veškeré funkcionality v fUmlGradlePluginu, což přináší spoustu dalších možností využití nástroje. fUmlGradlePlugin je plně nezávislý na fUmlStudiu. Projekt je tak možné modelovat, kompilovat, spouštět i generovat kód bez fUmlStudia. Gradle plugin lze používat z příkazové řádky. A jednou z největších výhod Gradle pluginu je možnost integrace do jiných Gradle projektů. Je tak možné automatizovat generování kódu z modelu před kompilací samotné Android aplikace. Kvůli prodloužení času kompilace projektu to nemusí být vždy vhodné.

Generátor kódu nepodporuje generování kódu pro kompletní fUML ale jen pro jeho podmnožinu. Generátor generuje ne jen kód koster tříd, ale i těla metod z diagramu aktivit, toto generování není vyladěné ale demostruje možnosti nástroje. Výhodou generování kódu z abstraktního modeluje je relativně jednoduchý přechod na jinou platformu vytvořením nových transformačnách šablon.

Existující MDD vývojová prostředí jsou propracovanější a mají více funkcí oproti vytvořenému vývojovému prostředí. Generátory kódu z UML existují, dokonce spouštění fUML

je dostupné přes modul Moka v Eclipse. Nenašel jsem však vývojové prostředí umožňující generování kódu Android aplikací z fUML modelů. Nepodařilo se mi ani najít nástroj, který by umožňoval generovat Java kód z fUML diagramu aktivit.

Možnosti dalšího vývoje nástroje

Jak je zmíněno v kapitole o implementaci generátoru kódu, generátor kódu podporuje jen podmnožinu funkcí z fUML modelů. Prvním krokem při dalším pokračování vývoje by mělo být rozšíření generátoru kódu o kompletní podporu fUML respektive ALF.

Dobrým rozšířením vývojového prostředí by byla možnost krokování při spouštění modelu a zobrazoví aktuálních hodnot proměnných.

Pokud chce vývojář použít nějakou platformě závislou funkcinalitu musí to řešit až při psaní kódu na cílovou platformu, není to možné při psaní ALF kódu, což je v pořádku ALF model má být platformě nezávislý a obecný.

Vhodné by ale bylo například implementovat abstraktní knihovnu pro vývoj mobilních aplikací, která by poskytovala funkcionalitu mobilních zařízení. K této knihovně by existovaly konkrétní implementace na různé mobilní platformy. To by eliminovalo množství kódu, které je nutné implemnetovat mimo fUML model.

Aktuálně generování chování metod je implementováno přímo v kódu fUmlCodeGeneratoru. Zajímavé by bylo generování kódu chování metod také přes FreeMarker šablony. Pro změnu generování by stačilo pouze upravovat generační šablony tak, jak je tomu při generování koster tříd. Otázkou je, zda by tyto šablony nebyly příliš složité a nepřehledné, pak je vhodnější ponechat generování přímo v kódu.

Nástroj je navržen tak aby mohla být jednoduše vytvořena další transformační schémata. Díky tomu by mělo nemělo být příliš složitě vytvořit transformační schémata pro generování kódu na jinou platformu například pro iOS nebo web. Problém je však v generování kódu chování metod, to by bylo nutné pro každou platformu implementovat v generátoru kódu.

Vhodným rozšířením by byla možnost editovat třídy přímo v diagramu tříd. Provedené změny by se okamžitě promítaly to ALF kódu. Modelování by urychlila funkce přechodu z diagramu na příslušné místo v kódu po kliknutí na grafický element. V některých případech by mohlo být užitečné grafické zobrazení diagramu aktivit pro těla metod.

Pro pohodlnější programování v ALFu by bylo dobré v editoru kódu zvýrazňovat syntaxi. Také by bylo přínosné zobrazovat syntaktické chyby přímo při psaní kódu.

Kapitola 10

Závěr

Ve své diplomové práci jsem se zabýval návrhem vlastního nástroje pro modelem řízený vývoj (Model Driven Development, MDD) Android aplikací.

Před návrhem nástroje bylo nutné nastudovat problematiku MDD a obecného modelování systémů, zjištěné poznatky jsou uvedeny v kapitole 3.

Vytvářený nástroj slouží pro vývoj Android aplikací, proto byla zkoumána platforma Android a možnosti vývoje na tuto platformu v kapitole 4.

Nástroje podporující MDD samozřejmě existují, některé vybrané byly vyzkoušeny a jsou popsány v kapitole 5.

Po nastudování potřebných teoretických základů a vyzkoušení existujích nástrojů, byl navržen vlastní MDD nástroj pro tvorbu Android aplikací. Návrh je popsán v kapitole 6. V navrženém nástroji je k modelování aplikací použito fUML a jazyk ALF pro textový popis modelů. V nástroji jsou modelovány Android aktivity a jejich přechody. Modely jsou v nástroji přímo spustitelné. Z modelů je na základě transformačních šablon generován kód Android aplikace.

V kapitole 7 je popsána implementace nástroje. Hlavní částí implementovaného nástroje je Gradle plugin, který provádí veškeré operace s modely. Pro pohodlnější používání nástroje bylo implementováno i grafické vývojové prostředí. Toto vývojové prostředí umožňuje spravovat soubory projektu, ovládat Gradle plugin i zobrazovat grafickou reprezentaci modelovaných tříd a aktivit.

Funkce a možnosti vytvořeného nástroje jsou demonstrovány na tvorbě ukázkové aplikace pro správu úkolů. Postup tvorby aplikace je předveden v kapitole 8. Část aplikace je modelována v implementovaném nástroji a platformě závislý kód je doprogramován v Android Studiu.

Veškerý implementovaný kód je zveřejněn na serveru GitHub jako otevřený software. Implementovaný nástroj je zhodnocen v kapitole 9. Implementovaný nástroj je funkční a v něm vytvořené modely je možné přímo spouštět. Z modelů je možné generovat funkční kód Android aplikace. Generovanou aplikaci je možné modifikovat a doplňovat o platformě závislý kód děděním z generovaných tříd.

Užitečné by bylo do nástroje doplnit možnost editace diagramů, nyní grafický editor slouží pouze pro zobrazení diagramů. Generování kódu nepodporuje kompletně všechny funkce fUML respektive ALF, ty by bylo vhodné do generátoru doplnit. Další navrhovaná rozšíření nástroje jsou popsána v kapitole 9.

Vypracováním diplomové práce jsem splnil vytyčené cíle uvedené v zadání práce.

Literatura

[1] Beck, K.; Cockburn, A.; Mellor, S.; aj.: *Manifesto for Agile Software Development*. [Online; navštíveno 15.12.2016].
URL http://agilemanifesto.org/

[2] Corp., P. T.: *Ontology Engineering*. [Online; navštíveno 20.12.2016].
URL http://www.modelbasedengineering.com/subdisciplines/ontology/

[3] Corp., P. T.: *What is Model-Based Engineering (MBE)?* [Online; navštíveno 20.12.2016].
URL http://modelbasedengineering.com/faq/index.html#What_is_Model-Based_Engineering

[4] Gianni, D.; D'Ambrogio, A.; Tolk, A.: *Modeling and Simulation-Based Systems Engineering Handbook*. CRC Press, 2015, ISBN 978-1-4665-7145-7.

[5] Křena, B.; Kočí, R.: *Úvod do softwarového inženýrství*. FIT VUT v Brně, 2010.

[6] Luboslav Lacko: *Vývoj aplikací pro Android*. Computer Press, 2015, ISBN 978-80-251-4347-6.

[7] Mayerhofer, T.; Langer, P.; Kappel, G.: A runtime model for fUML. In *Proceedings of the 7th Workshop on Models runtime*, ACM, 2012, s. 53–58.

[8] Mellor, S. J.; Balcer, M. J.: *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, 2002, ISBN 0-201-74804-5.

[9] (OMG), O. M. G.: *About OMG*. [Online; navštíveno 14.12.2016].
URL http://www.omg.org/gettingstarted/gettingstartedindex.htm

[10] (OMG), O. M. G.: *Business Process Model And Notation (BPMN), Version 2.0*. OMG Document Number formal/2011-01-03 (http://www.omg.org/spec/BPMN/2.0/PDF), 2011.

[11] (OMG), O. M. G.: *For A UML Action Language: Action Language For Foundational UML (ALF), Version 1.0.1*. OMG Document Number formal/2013-09-01 (http://www.omg.org/spec/ALF/1.0.1/PDF), 2013.

[12] (OMG), O. M. G.: *Object Constraint Language, Version 2.4*. OMG Document Number formal/2014-02-03 (http://www.omg.org/spec/OCL/2.4/PDF), 2014.

[13] (OMG), O. M. G.: *OMG Meta Object Facility (MOF) Core Specification, Version 2.5*. OMG Document Number formal/2015-06-05 (http://www.omg.org/spec/MOF/2.5/PDF), 2015.

[14] (OMG), O. M. G.: *OMG Unified Modeling Language (OMG UML), Version 2.5.* OMG Document Number formal/15-03-01 (http://www.omg.org/spec/UML/2.5/PDF), 2015.

[15] (OMG), O. M. G.: *XML Metadata Interchange (XMI) Specification, Version 2.5.1.* OMG Document Number formal/2015-06-07 (http://www.omg.org/spec/XMI/2.5.1/PDF), 2015.

[16] (OMG), O. M. G.: *Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.2.1.* OMG Document Number formal/2016-01-05 (http://www.omg.org/spec/FUML/1.2.1/), 2016.

[17] Picek, R.; Strahonja, V.: *Model Driven Development – Future or Failure of Software Development?* . 2008.

[18] Rumpe, B.: Executable Modeling with UML. A Vision or a Nightmare? *arXiv preprint arXiv:1409.6597*, 2014.

[19] Selic, B.: The pragmatics of model-driven development. *IEEE software*, ročník 20, č. 5, 2003: s. 19–25.

[20] Soley, R.; aj.: Model driven architecture. *OMG white paper*, ročník 308, č. 308, 2000: str. 5.

Přílohy

Příloha A

Obsah elektronického nosiče

Struktura elektronického nosiče:

- **/technicka_zprava.pdf**
  technická zpráva ve formátu PDF
- **/technicka_zprava-src**
  zdrojové soubory technické zprávy
- **/src/fUmlAndroidTodoList/**
  Android projekt vzorové aplikace
- **/src/fUmlCodeGenerator/**
  zdrojové soubory generátoru kódu
- **/src/fUmlGradlePlugin/**
  zdrojové soubory Gradle pluginu
- **/src/fUmlProjectTodoList/**
  fUML projekt vzorové aplikace
- **/src/fUmlStudio/**
  zdrojové soubory vývojového prostředí
- **/bin/TodoList.apk**
  instalační soubor vzorové aplikace TodoList
- **/bin/fUmlCodeGenerator.jar**
  spustitelný jar soubor generátoru kódu
- **/bin/fUmlStudio.jar**
  spustitelný jar soubor vývojového prostředí fUmlStudio
- **/bin/fUmlStudio.bat**
  skript pro spuštění fUmlStudia
- **/bin/repo/**
  maven repozitář s fUmlGradlePluginem

Příloha B

Objekty použitelné ve FreeMarker templatech

root

- allClasses - Class[]
- allActivityClasses - Class[]

Class

- name - String
- qualifiedName - String
- nameUnder - String
- packageName - String
- visibility - String
- imports - Import[]
- generals - Class[]
- isActivity - Boolean
- isPersistent - Boolean
- isLibrary - Boolean
- isTemplate - Boolean
- isAbstract - Boolean
- attributes - Attribute[]
- operations - Operation[]

Attribute

- name - String
- type - String
- visibility - String

Operation

- name - String
- type - String
- visibility - String
- parameters - Parameter[]
- activity - Activity
- isStartActivity - Boolean
- getStartActivity - Class
- isConstructor - Boolean

Activity

- name - String
- type - String
- visibility - String
- parameters - Parameter[]
- body - String
- code - String

Parameter

- name - String
- type - String

Import

- qualifiedName - String