FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Analýza metod Model-Driven Development pro vytváření webových informačních systémů
Student:	Olga Zotkina
Vedoucí:	Ing. Marek Suchánek
Studijní program:	Informatika
Obor / specializace:	Informační systémy a management
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

- Seznamte se s oblastí Model-Driven Development, popište klíčové pojmy a proveďte stručnou rešerši existujících přístupů ke generování webových informačních systémů z konceptuálních modelů.
- Analyzujte open-source redakční systém Wordpress a vytvořte konceptuální model jádra tohoto systému s využitím dostupné dokumentace i reverzního inženýrství zdrojových kódů. Zaměřte se především na strukturu, ale zohledněte také důležité procesy a workflow.
- Vyberte a použijte tři různé přístupy pro vygenerování zdrojových kódu aplikace z vytvořeného modelu nebo jeho upravené formy. Popište postup, výslednou aplikaci a potenciální další vývoj včetně změn v modelu.
- Porovnejte přístupy z hlediska snadnosti použití, kompletnosti vygenerované aplikace a dalších přínosů. Zohledněte možnosti budoucího rozvoje a udržitelnost aplikace.
- Zhodnoťte přínosy analyzovaných metod ve smyslu úspor prostředků během vývoje, na provoz i pro následnou údržbu a rozvoj systému.

Elektronicky schválil/a Ing. David Buchtela, Ph.D. dne 10. září 2020 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Analýza metod Model-Driven Development pro vytváření webových informačních systémů

Olga Zotkina

Katedra softwarového inženýrství
Vedoucí práce: Ing. Marek Suchánek

12. května 2021

Prohlášení

Prohlašuji, že jsem předloženou práci vypracovala samostatně a že jsem uvedla veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 12. května 2021 . . . . . .

Odkaz na tuto práci

Zotkina, Olga. *Analýza metod Model-Driven Development pro vytváření webových informačních systémů.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato bakalářská práce řeší efektivitu použití vývoje řízeného modely pro automatizaci procesu vytváření zdrojového kódu. Cílem bylo zhodnotit přínosy analyzované metody během navrhování, implementace, tvorby dokumentace a rozvojů informačního systému. Vyhodnocení probíhalo na základě tří nástrojů GenMyModel, Umbrello a Enterprise Architect pro automatické generování zdrojového kódu z modelu open source redakčního systému WordPress a výsledky ukazují, že dojde ke zrychlení vývoje a usnadnění dalšího rozvoje v budoucnu. Na základě analýzy v této práci je stanoveno doporučení použít modelem řízený vývoj pro rozsáhlé informační systémy, kde ušetří vývojářům podstatné množství času. Pro menší aplikace se ukázal vhodnější tradiční vývoj.

Klíčová slova Informační systém, vývoj řízený modely, CASE-nástroj, modelování, generování zdrojového kódu

Abstract

This bachelor thesis addresses the effectiveness of using development driven models to automate the process of creating source code. The aim was to evaluate the benefits of the analyzed method during the design, implementation, creation of documentation and development of the information system. The evaluation was carried out on the basis of three tools GenMyModel, Umbrello and Enterprise Architect for automatic generation of source code from the open source model of the WordPress editorial system, and the results show that it will accelerate development and facilitate further development in the future. Based on the analysis in this work, it is recommended to use Model-Driven development for large-scale information systems, where it will save developers a significant amount of time. For smaller applications, traditional developments have proved more suitable.

Keywords Information system, Model-Driven Development, CASE-tool, modeling, source code generation

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Vzhledem k rostoucí potřebě informačních systémů a zvýšení jejich složitosti byl vytvořen nový přístup k vývoji informačních systémů řízený modely. Tento přistup umožnil výrazně zjednodušit vývoj, zvýšit úspor prostředků a urychlit vstup produktu na trh. Hlavními zásadami pro vývoj řízený modely jsou abstrakce, která pomáhá se zaměřením na určité detaily systému a odstraněním nepodstatných častí systémů, a automatizace, která přispěla k urychlení vývoje a zlepšení kvality výsledného kódu.

V současné době existuje mnoho nástrojů vývoje řízeného modely, které automatizují určité funkce, které jsou součástí procesu vývoje. Z množství nástrojů je však obtížné vybrat aplikaci vhodnou pro vývoj konkretního informačního systému, která splňuje specifické požadavky.

Cílem bakalářské práce je využití třech metod vývoje k generování zdrojového kódu aplikace pro vymezení přínosů a nevýhod jednotlivých metod a následného porovnání těchto přístupů. Generování zdrojového kódu probíhá na základě modelů získaných reverzním inženýrstvím již existujícího open source redakčního systému WordPress.

WordPress může být považován za vhodný systém pro provedení procesu reverzního inženýrství. WordPress má rozsáhlou dokumentaci, která vyžaduje značné množství času pro její prozkoumaní a porozumění. Tento systém má obrovské množství funkcí a možností, které mohou ztěžovat pochopení práce s WordPressem jako jednotným systémem. Abstraktní modely, které ukazují hlavní podstaty systému a základní procesy v něm, mohou odrážet systém jako celek a pomoct při zvládnutí WordPress v počátečních fázích. Dílčím úkolem je vytvoření UML diagramů WordPress, které jsou postačující pro generování kódu výsledné aplikace.

K dosažení stanovených cílů jsou v teoretické části práce vymezené základní vlastnosti vývoje řízeného modely, stanovená nejpoužívanější iniciativa vývoje řízeného modely a zohledněné její standardy. Navíc byly popsané diagramy, které se používají pro modelování architektury systému. Ke zpracování odborných témat, která se týkají struktury WordPress, byly použité doku-

mentace WopdPress a informativní web založený jedním z autorů WordPress. K vygenerování zdrojového kódu jsou použité dokumentace zvolených přístupů vývoje řízeného modely.

Kapitola 1

Cíl práce

Cílem je zhodnotit přínosy použití nástrojů pro vývoj řízený modelem metod ve smyslu úspor prostředků během vývoje, na provoz i pro následnou údržbu a rozvoj systému na základě třech různých odůvodně vybraných nástrojů pro vygenerování zdrojových kódů aplikace z modelu. Porovnání přístupu je realizováno z hlediska snadnosti použití, kompletnosti vygenerované aplikace a možnosti budoucího rozvoje a udržitelnosti aplikace.

Dílčím cílem je vymezit základní entity a vztahy open source redakčního systému WordPress, vytvořit strukturální model jádra tohoto systému a zohlednit základní procesy tohoto systému pomocí vhodných diagramů s využitím dostupné dokumentace i reverzního inženýrství zdrojových kódů. Následně na základě vytvořeného modelu vygenerovat zdrojový kód aplikace.

Hlavní přínos práce spočívá v analýze třech používaných v praxi metod vývoje řízeného modelem a vyhodnocení jejich přínosu pro vývoj. Tuto analýzu lze použít pro hodnocení oprávněnosti vývoje systému pomocí vývoje řízeného modelem a případné zvolení vhodnějšího nástrojů pro modelování a generací zdrojového kódu aplikací ze seznamu analyzovaných nástrojů uvedených v bakalářské práci.

Kapitola 2

Vývoj řízený modely

2.1 Základní principy MDD

Architektura Model-Driven Development (MDD) byla vytvořena s cílem pomoct řídit složité systémy a vzájemné závislosti komplexních systémů prostřednictvím modelů. V rámci MDD je systém chápán jako určitý soubor částí a vztahů mezi těmito částmi, které jsou kombinovány za účelem získání konkrétního výsledku. Vývoj řízený modely je způsob přístupu k vývoji softwaru, při kterém se modely, nikoli programy, stávají primárním základem vývoje, ze kterých se generuje kód a další artefakty [4].

Model v kontextu MDD je abstraktní popis softwaru, který poskytuje informaci o některých jeho aspektech ve formě sady prvků a jejich vztahů. Model abstraktně popisuje objekty, jejich vlastnosti, metody a vztahy mezi nimi. Model není vázán na konkrétní jazyky nebo programovací prostředí. V MDD jsou modely objektově orientované [5].

Před vytvořením architektury MDD byly modely používány hlavně jako náčrtky, které pomáhaly pochopit obecný obrys systému a zjednodušit jeho chápání jako celku nebo jako výkresy představující návrh systému. V architektuře MDD se vyvíjený systém velmi shoduje s modelem, protože model se používá jako dokumentace a specifikace. Specifikace (specification) – je deklarativní popis zařízení nebo pracovního procesu. Model je součástí definice systému [6].

V této architektuře hrají hlavní roli modely, protože právě z modelu lze pomocí transformace generovat nejen zdrojový kód softwarového systému, ale také další artefakty - nespustitelné soubory, testy a dokumentace. Technologie modelování a vývojové metody MDD jsou primárně založené na kombinaci abstrakce a automatizace. MDD nastavuje na model dvě podmínky: musí existovat v elektronické formě a byt strojově čitelným. I když graf nakreslený na papíře splňuje kritéria modelu, pokud není poskytnut automatický přístup k jeho obsahu, nemůže se podílet na vývoji řízeném modely.

Abstrakce (abstraction) je proces ignorování irelevantních detailů a zamě-

řování na podstatné aspekty s cílem systému porozumět obecně. Použití abstrakce neznamená nedostatek přesnosti. Při použití abstrakce je zachována přesnost v těch detailech, na které je soustředěna pozornost, a zbytečné konkrétní podrobnosti jsou vynechané.

V MDD se abstrakce používá k ignorování podrobností implementaci a zaměření pozornosti na logické zobrazení aplikace. Abstrakce se navíc uplatňuje na různých úrovních a z různých pohledů. Abstrakce nepopírá důležitost jakýchkoliv detailů pro systém, ale pomáhá oddělit aspekty, které jsou pro konkrétní hledisko irelevantní. Díky abstrakcí v přístupu MDD lze vynechat podrobnosti implementaci a zaměřit se na popis architektury a návrhu systému.

Automatizace se stala hlavní výhodou MDD oproti jiným přístupům k modelování. Architektura MDD položila základ pro myšlenku standardizované automatizace vývoje. Takový návrh umožnil snížit dobu nezbytně nutnou k tvorbě systému a vyhnout se značnému počtu manuálních chyb, které dělají i zkušení programátoři. Model v MDD má dostatek informací pro následné generování implementačních artefaktů z něj a nabízí šablony pro transformaci. Architektura MDD implikuje kompletní změnu standardu vývoje systému, nejen konkrétní vylepšení.

2.2 MDA

Jednou z nejznámějších iniciativ MDD je přístup Model Driven Architecture (MDA). Architektura MDA byla novým přístupem k vytváření aplikací pro více platforem. Před vytvořením architektury MDA neexistoval žádný jednotný integrační mechanismus [7].

Jelikož byl každý informační systém vytvořen na určité platformě (softwarové platformy jako Microsoft .NET, CORBA, Java atd.), byl zatížen svými velmi objemnými pravidly a omezeními. K vytvoření, vývoji a rozvoji systému na nové platformě a školení zaměstnanců společnosti, kteří s ní budou v budoucnu pracovat, byly nutné značné finanční investice. Pokud cílem bylo vytvořit stejný systém, ale na jiné platformě, vše se opakovalo znovu. Navíc v případě používání různých platforem, jejichž potřeba často vyvstává, platformy musí být kompatibilní a pracovníci společnosti musí být schopni pracovat se všemi platformami současně. Tato okolnost vedla k vysokým finančním a časovým nákladům [8].

Bylo nutné vytvořit optimální řešení, které by bylo efektivnější: snížilo počet chyb nevyhnutelných během manuálního tradičního vývoje, software udělalo přenosným a umožnilo opětovného použití na různých platformách. Podle tvůrců byla architektura MDA novým kolem ve vývoji programovacích technologií, protože popisovala vývojový proces jako celek a umožňovala automatizaci vytváření aplikací [9]. Základem pro vytvoření a vývoj architektury

MDA byly standardy a specifikace OMA a Common Object Request Broker Architecture (CORBA), které se již v praxi osvědčily [10].

Je třeba poznamenat, že CORBA, který byl vytvořený před MDA, byl sice známý, ale umožňoval pouze interakci mezi systémy v různých operačních systémech, programovacích jazycích a výpočetních zařízeních. Jeden z vývojářů CORBA Michi Henning věřil, že ve standardu CORBA je tolik technických závad, že již bylo obtížné něco opravit nebo přidat, aniž by se něco nepokazilo [11]. Proto bylo zapotřebí nové architektury, ve které by vývojáři vzali v úvahu nedostatky předchozích technologií. Základem MDA staly OOP technologie a standardy Common Warehouse Metamodel (CWM), Unified Modeling Language (UML), XML Metadata Interchange (XMI) a Meta Object Facility (MOF).

MDA je založeno na použití nástrojů nebo kombinací řetězců nástrojů poskytovaných dodavateli MDA. Seznam dodavatelů MDA a produktů, které poskytují, jsou veřejně dostupné na webových stránkách OMG [12]. Nástroje zvyšují úroveň automatizace v MDA, integrují modelování, pomáhají procházet mezi úrovněmi abstrakce a modely v MDA a nabízí další přínosy. Softwarové nástroje odpovědné za architektonický návrh softwaru se nazývají CASE-nástroje [8].

2.2.1 Standardy MDA

Na počátku 90. let byly softwarové aplikace vytvářeny bez dodržování jakýchkoli standardů, takže byly výjimečné a zcela odlišné. Vývoj průmyslu však pokračoval velkými kroky, které vyžadovaly složitější obchodní aplikace se širokou funkčností. Bohužel bylo často téměř nemožné vytvořit univerzální systém, který splňuje všechny požadavky. Ve výsledku vzniklo mnoho vysoce kvalitních programů zaměřených na řešení jednoho problému, ale vytvoření takového, který splňuje všechny požadavky, bylo problematické. Myšlenka vytvořit malou sadu souborů, které popisují systém, a poté z nich vygenerovat samotný systém, vznikla už dávno, ale adaptace konceptu vývoje založeného na modelu byla poměrně pomalá [13].

Hlavní myšlenkou MDA je umožnit vytváření softwarových systémů založených na vytvořených modelech, to znamená převést modely do konkrétních implementací. K vytvoření celého řetězce nezbytných nástrojů k tomu jsou však zapotřebí standardy. Existuje mnoho možných řešení, jak vytvořit model a postavit z něj systém, ale pokud si každá společnost vytvoří svůj vlastní způsob, bude čelit problému kompatibility systému. Navíc neexistuje žádná záruka, že tato nebo jiná metoda bude mít všechno, co je nezbytné k vytvoření vývojových konceptů řízených modelem, proto jsou jediné standardy důležité [14].

Vývojáři se samozřejmě pokusili ještě před přijetím standardů vytvořit sadu CASE-nástrojů, ale výsledkem bylo, že pokusy o jejich implementaci nebyly zakončené úspěchem ve velkém počtu projektů. Lidé nechtěli záviset na

jediném prodejci, jehož produkt byl jedinečný. Klienti ztratili možnost zakoupit další nástroje od jiného prodejce, protože byly nekompatibilní, nemohly se navzájem nahrazovat ani doplňovat. To vedlo k potřebě koupit kompletní sadu nástrojů od jediného vývojáře, která ještě nebyla testována v praxi dříve. Zákazníci s takovými projekty zacházeli opatrně, protože se obávali, že zbytečně přijdou o své finanční prostředky, ale nakonec nedostanou vyžadovaný produkt, který vyřeší všechny zadané úkoly.

Z tohoto důvodu bylo nutné vypracovat seznam jednotných specifikací, které by umožnily softwarovým produktům různých vývojářů bez problémů vzájemně integrovat. Nelze popřít, že taková omezení vedou k dalším nákladům a obtížím během vývoje, takže zpočátku tuto myšlenku nepodporovali všichni. Ale standardy jsou nyní nedílnou součástí vývoje softwarových systémů.

Hlavní činnost v oblasti standardizace MDD provádí neziskové konsorcium Object Management Group (OMG). OMG je mezinárodní neziskové softwarové konsorcium, které bylo založeno v roce 1989. OMG se od samého začátku prohlásila za demokratickou organizaci a standardy, nad kterými pracovala, byly bezplatné a otevřené k dodatkům a změnám. Společnost OMG nazvala svým posláním standardizovat proces a poskytnout vývojářům informace, že existují standardy, které jim umožní implementovat řetězec nástrojů pro vývoj řízený modely a vytvořit plnohodnotné spustitelné simulační prostředí. OMG vyvíjejí standardy CORBA, UML, MOF, CWM a XMI [15].

Na začátku OMG měla pouze třináct členů. Aktuálně její členové jsou stovky organizací, nejen výrobci softwaru, ale i koncové uživatele software, přičemž každá členská organizace má stejně důležitý hlas v hlasovacím procesu. Zpočátku nebyl princip fungování OMG ideální a měl řadu nedostatků, proto byl kritizován nejen zvenčí, ale také členy konsorcia. Některé specifikace byly vydány technicky nezralé: nemohly být částečně nebo plně implementovány v praxi kvůli vadám, byly objemné a složité k porozumění. Řada dalších specifikací, které byly implementovatelné, byla prakticky k ničemu, protože jejich implementace vyžadovala nepřijatelnou režii, ale nepřinesla významné výhody. Bylo mnoho takových případů, které potenciální spotřebitele odrazovaly.

Důvodem pro výše uvedené bylo, že někteří členové OMG nebyli odborníky v oblasti informačních technologií, ale pouze koncovými uživateli, kteří často hlasovali "pro"standard, který teoreticky potřebovali, ale nerozuměli implementaci standardu v praxi. Účastníci-experti zase mohli hlasovat "pro", navzdory technickým vadám standardu, protože chtěli uzavřít s potenciálním klientem v budoucnu smlouvu. Kromě toho, někdy nechtěli standardy přijmout, protože jednotlivý systém mohl ztratit jedinečnost, co působí zvýšeni konkurence, protože by jiné společnosti byly schopny nabízet stejné služby. Na jedné straně standardy technologii zjednodušily, ale na druhé straně někdy přinutily předělat již hotový produkt [16].

Výsledkem je, že se spotřebitelé i přes standardizaci stále ocitli vázáni

na konkrétní produkt vyvinutý konkrétní společností na konkrétní platformě. Členové OMG však analyzovali své chyby a postupem času dokázali většinu z nich napravit.

Pro vytvoření nových standardů vyvinula společnost OMG postup založený na konceptu Request For Proposal (RFP). OMG má speciální oddělení Task Force, které vytváří RFP a odpovídá na požadavky nejen členů OMG, ale i jiných společností a jednotlivců. Task Force poté odesílá návrh RFP všem členům OMG s podrobným popisem návrhu konkrétního standardu. Hlavním požadavkem na RFP je skutečná potřeba existenci standardu pro navrhovaný nebo existující produkt. Členové OMG dostávají tři týdny na to, aby si to promysleli, jestli uvazuji za vhodné ho zavedení, poté návrh RFP je projednán, následně je vypracován harmonogram vydání nového standardu. Po vytvoření nových specifikací prostřednictvím návrhů a hlasování, až dojde ke shodě na konečné specifikaci, stane se návrh přijatým standardem [10].

Základem politiky konsorcia, který je klíčem k úspěchu, je "no shelf-ware rule". Pravidlo spočívá v tom, že než přijmou nový standard, zkontrolují, zda ho lze implementovat a skutečně použít. Cílem však není jen vytvořit nový standard, ale i vytvářet odborníky, které budou umět pracovat se standardem, takže společnost poskytuje školení, knihy a webináře [17].

IT průmysl se vyvíjí, proto společnost OMG se snaží udržovat své standardní použitelné. Všechny standardy, které byly přijaty, jsou neustále aktualizovaný, doplňovány a revizovány. Stojí za zmínku, že rychlý vývoj těchto standardů je vyvolán velkým a neustále se zvyšujícím počtem společností, které používají kompatibilní produkty. Jak jejich počet roste, zakládá se více RFP a tím se rychleji standardy vyvíjí [18, 15].

2.2.2 MOF

MOF (Meta Object Facility) je obecný abstraktní jazyk pro popis metamodelů, který definuje obecná rozhraní a sémantiku pro interakci metamodelů [19]. MOF byl původně navržen tak, aby poskytoval typový systém pro CORBA a rozhraní s cílem použití k vytvoření a manipulaci s těmito typy.

Metadata jsou obecný termín pro data, která popisují původní data. Hlavní rozdíl mezi daty a metadaty spočívá v tom, že metadata jsou vždy informativní, protože ukazují na další data, a zpracované. Zatímco původní data mohou být pouze popisem něčeho, pozorováním nebo měřením a mohou být neinformativní a nezpracovaná. Lze říct, že metadata popisují základní informace o datech, odrážející pouze podstatu dat, aniž by zacházely do detailu. Cílem používaní metadat je usnadnit hledání a práci s konkrétními instancemi dat. Je třeba ujasnit, že v kontextu MOF má pojem „model“ poněkud odlišný význam a označuje jakoukoliv sadu metadat, která mají společnou abstraktní syntaxi a sémantickou strukturu.

Standard MOF je schopen pracovat s jakýmikoli metadaty, která lze popsat pomocí metod objektivního modelování. MOF navíc neomezuje úroveň

jejich detailů ani toho, co mají popisovat, vždy záleží pouze na individuálních požadavcích na metadata v konkrétním systému. MOF byl navržen tak, aby různé společnosti nebyly nuceny používat stejné modelovací jazyky. MOF tedy nezakazuje použití behaviorálních a datových modelů jiných než UML, ale musí splňovat podmínku, že jsou založeny na principech MOF.

Pro MOF klíčový význam má meta-metamodel. Meta-metamodel je metamodel, který popisuje další metamodel. Protože metadata samotná jsou informace, i když nutně splňující určitá kritéria, lze je taky popsat jinými metadaty. Typický systém má mnoho druhů metadat a metamodelů. Metamodel MOF spojuje tyto metamodely tak, že pro ně definuje společnou abstraktní syntaxi a stává se pro ně meta-metamodelem. Struktura metadat MOF má obvykle čtyřvrstvou architekturu, ačkoli specifikace umožňuje více či méně vrstev než čtyři.

Horní vrstva M3 odpovídá meta-metamodelu - jazyku pro vytváření metamodelů. Meta-metamodel definuje jazykový model na nejvyšší úrovni abstrakce a je nejkompaktnějším popisem modelu.

Třetí úroveň M2 odpovídá metamodelu a definuje jazyk pro specifikaci modelů. Druhá úroveň odpovídá metamodelu UML, který popisuje samotný UML, a metamodelu, který popisuje IDL (Interface Definition Language) - jazyk definice rozhraní. Tato úroveň má rozvinutější sémantiku základních konceptů.

Úroveň M1 označuje modely, které popisují informace o konkrétní oblasti předmětu - názvy tříd, atributy, akce. Mohou to být například modely napsané v UML a rozhraní jazyka IDL.

Nejnižší úroveň M0 odráží instance modelovaných objektů, kde koncept modelu je vytvořen na úrovni objektu. Tato vrstva obsahuje konkrétní informace a slouží k popisu objektů v reálném světě [19].

2.2.3 CWM

Většina organizací trpí přebytkem nadbytečných a nekonzistentních údajů, které je obtížné efektivně spravovat, vyhledávat, zpracovávat a analyzovat. Je zřejmé, že vývojáři i koncoví uživatelé potřebují snadno pochopit, jaké informace mají v datovém úložišti, ohodnotit cennost informace z obchodního hlediska a určit původní zdroje informace. Kromě toho, jsou zapotřebí nástroje, které pomáhají spravovat informace a generovat zprávy automaticky a pravidelně [3].

Jakýkoli informační a analytický systém má současně mnoho různých produktů, které mají svá vlastní metadata uložená v příslušném úložišti nebo datovém slovníku ve speciálních formátech. Všechna tato metadata se ale liší strukturou a syntaxí, i když jsou sémanticky vzájemně propojená a pro konzistentní a správné fungování systému jako celku je nutné je sdílet a přenášet z jednoho prostředku do druhého. Sémantika je definice pravidel pro přisuzování významu výrazům jazyka. Na oplátku syntaxe určuje pravidla pro konstruo-

vání výrazů jazyka. Jedním z hlavních problémů ve správě metadat je výměna dat mezi různými produkty, úložišti a databázemi.

CWM je obecný standard pro popis informačních interakcí mezi datovými úložišti. Je založen na přístupu výměny metadat modelů, přičemž modely, které odpovídají metadatům specifickým pro daný produkt, jsou vytvořené na základě syntaktických a sémantických specifikací společných metamodelů. Systémy, které „rozumějí“ metamodelu CWM, si vyměňují data ve formátech, které jsou konzistentní s tímto modelem. CWM poskytuje rámec pro reprezentaci metadat o zdrojích dat, účelech dat, transformacích, analýze, procesech a operacích, které vytvářejí a spravují datové úložišti. Sbírka metamodelů CWM představuje nejčastěji používaná metadata v databázích a nástrojích pro obchodní analýzu.

Model CWM byl založen na třech již existujících objektových technologiích: UML, XMI, MOF. Model CWM používá grafickou interpretaci sémantiky UML k reprezentaci metamodelů a datových modelů a standardní modely UML k popisu sémantiky analýzy a návrhu modelu. Každý metamodel je implementován jako balíček obsahující sadu základních tříd definovaných v UML. XMI se používá k výměně metadat ve formě streamů nebo souborů ve standardním formátu XML. Formální popisy metamodelu CWM jsou postaveny na základě MOF a využívají rozhraní CORBA [20].

Metamodel CWM se skládá z řady submetamodelů, které jsou strukturovány a distribuovány do čtyř vrstev (viz příloha C.1). Nejspodnější vrstva, „základ“ (Foundation), se skládá z metamodelů, které podporují specifikaci základních strukturálních prvků, jako jsou výrazy a datové typy. Druhá vrstva, „zdroje“ (Resource), obsahuje metamodely používané ke upřesnění zdrojů dat a cílových databází.

Třetí vrstva, „analýza“ (Analysis), obsahuje metamodely, které zpracovávají transformaci procesů, vizualizaci a šíření dat, interaktivní analytické zpracování (OLAP), dolování dat (data mining) a tzv. Business Nomenclature, která obsahuje třídy a asociace pro reprezentaci business metadat a snadný přístup k těmto metadatům.

Čtvrtá vrstva „Management“ (Management) se skládá z metamodelů, které jsou zodpovědné za provoz úložišti. Tyto nástroje umožňují simulovat postupy pro správu úložišti, stanovit pravidla pro jejich implementaci, určovat kontrolní a protokolovací procesy pro načítání informací a provádění úprav dat v úložišti [3]. CWM v MDA se používá ke standardizaci úplné metamodely za účelem provedení těžby databáze a mapování MDA na databázová schémata v datovém prostoru [20].

2.2.4 XMI

Standard XML Metadata Interchange (XMI) se používá k výměně metadat pomocí XML a popisuje mapování modelů MOF a UML na standard XML. Standard XMI je nezávislý na technologii middlewaru, takže jakékoli úložiště

metadat nebo nástroj, který dokáže kódovat a dekódovat toky XMI, si může vyměňovat metadata s jinými úložišti nebo nástroji. Kromě toho, XMI nevyžaduje implementaci rozhraní CORBA a metamodely nemusí být založeny na MOF. Metadata jsou vyměňována s úložišti, jejichž modely nejsou založeny na MOF, prostřednictvím vlastních mapování mezi dokumentem XMI a metamodelem úložiště. Je možné vyměňovat si metadata s nástroji, které nerozumí metadatům, nebo metadaty v diferenciální formě. XMI také podporuje kódování jednotlivých zvláštních fragmentů modelu.

Standard XMI je založen na dvou hlavních pravidlech:

1. The XML DTD Production Rules - pravidla XML Document Type Definition (DTD) určuje generování typu dokumentu XML pro metadata zakódovaná v XMI, která specifikuje syntaxi pro dokumenty XMI, což umožňuje použití obecných dokumentů XML pro mapování a ověřování dokumentů XML.

2. The XML Document Production Rules - pravidla tvorby dokumentů XML pro kódování metadat do formátu kompatibilního s XML, která lze také použít opačně k dekódování dokumentů XMI a obnovení metadat.

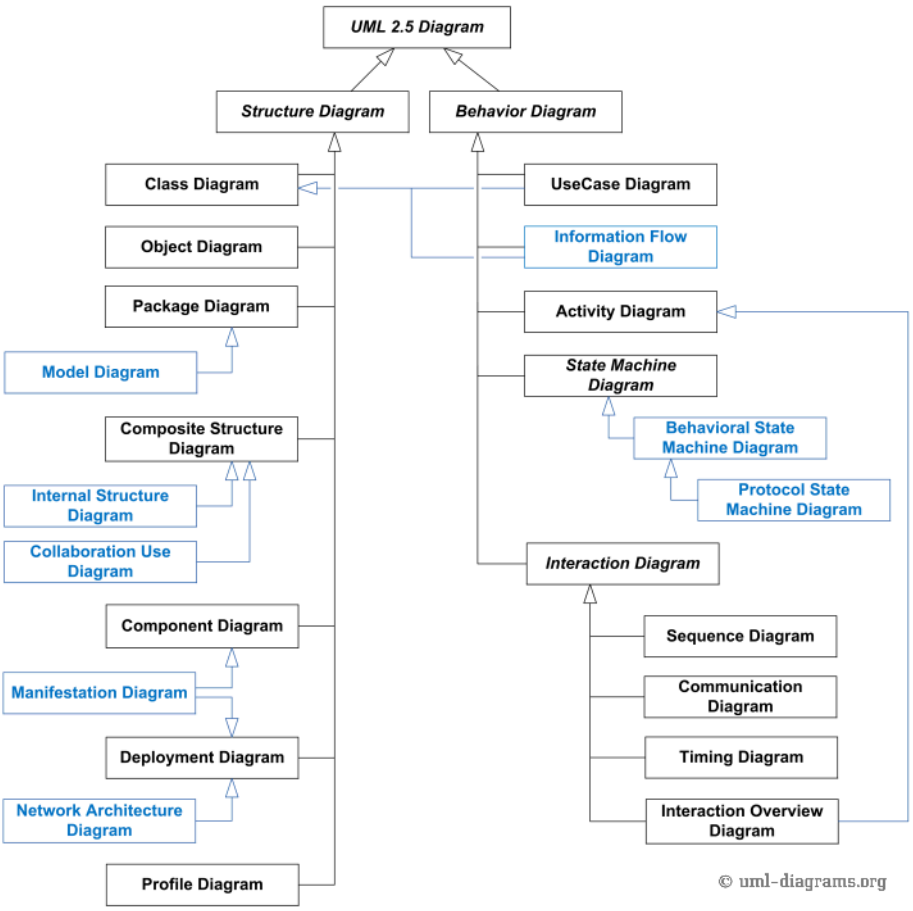
Výsledkem je, že metamodely jsou převedené na strukturu dokumentu DTD a modely jsou převedené na tělo dokumentu XML. Tím se vytvoří jediný dokument, který kombinuje model a jeho metamodel, který obsahuje data spolu s informací potřebné k jejich interpretaci [21].

2.3 UML

UML je grafický deskriptivní modelovací jazyk, který byl původně vytvořen pro standardizaci vizualizace návrhu systému. UML se nyní sestává z integrované sady diagramů a nabízí řadu různých nástrojů pro definování, vizualizaci, konstrukci a dokumentaci artefaktů softwarového systému a modelování obchodních procesů. Vzhledem k tomu, že UML má dvanáct diagramů, jak je ukázáno na obrázku 2.1, pro UML byla zavedena vlastní sekce v rámci teto práci, ačkoliv se jedna o standard, spadající do MDA.

Artefakt (artifact) je fyzická implementační jednotka, která je vytvořena nebo použita během vývoje softwaru a je odvozena od elementu modelu. Jedná se o informační prvky na úrovni implementační platformy, které jsou součástí softwarového systému a jsou nějakým způsobem používány během provozu systému. Artefakty implementují třídy a metody, ale nemají atributy a operace. Stojí za zmínku, že UML lze použít k modelování nejen softwarových systémů, ale také jiných než softwarových systému, například technologických procesů v podniku.

UML vznikl v 90. letech jako objektově orientovaný modelovací jazyk a jako výsledek procesu sjednocení různých již existujících objektově orientovaných



Obrázek 2.1: UML diagrams 2.5. Poznámka: položky v modré barvě nejsou součástí oficiální taxonomie diagramů UML 2.5. [1]

modelovacích jazyků. Jazyky grafického modelování byly vytvořeny s cílem usnadnit proces návrhu, protože již existující programovací jazyky nemohly poskytnout dostatečnou úroveň abstrakce [22]. UML lze nazvat univerzálním, protože je vhodný pro modelování projektů různých velikostí a aplikačních oblastí, mohou jej používat odborníci s různými úrovněmi kompetencí v organizacích různých typů.

První verze UML 1.1 byla vydána v prosinci 1997 a byla založena na třech technikách objektově orientovaného modelování: Booch, OMT a OOSE, které jsou kombinací osvědčených postupů objektově orientovaného programování, jazyků pro návrhové modelování a jazyků architektonického popisu [23].

Další verze UML 1.2 vylepšila notaci a rozšířila sémantiku jazyka. UML 1.2 měl přesnější definice pravidel abstraktní syntaxe a sémantiky a vylepšené možnosti modelování umožnily navrhnout velké systémy [24].

Poslední verze UML 2.5 byla představená v červnu 2015. V této verzi

byla specifikace jazyka zjednodušena a několik jeho různých popisných dokumentů bylo sloučeno do jedné specifikace. Dříve existovaly úrovně shody s jazykem (compliance levels) L1–L4 , které byly v této verzi vyloučeny, protože byly v praxi považovány za zbytečné. Místo toho existují dvě možnosti: bud nástroj odpovídá celému UML nebo neodpovídá UML. Je možná částečná shoda nástrojů s UML, který implementuje podmnožinu jeho metamodelu, notace a sémantiky, ale má být jasně uvedeno, o kterou podmnožinu se jedná. Tato úprava změnila pravidla pro určování, jestli nástroj odpovídá UML. Bez ohledu na zlepšení se od první verze principy používání jazyka nezměnily [25].

Jazyk UML má určitá sémantická pravidla, která umožňují určit jména prvků jazyka, rozsah, kde tyto jména platí, viditelnost jména - oblast kde mohou je používat jiné prvky, integritu modelu a implementace nějakého dynamického modelu navrženého systému.

Model UML pro implementovatelný systém určuje jeho vlastnosti a chování. Hlavními modelovací prvky modelu jsou entity (things), které jsou abstrakce a instance meta-tříd metamodelu. Entity jsou propojeny různými vztahy a diagramy seskupují sbírky určitých entit.

Model UML se skládá ze tří hlavních kategorií prvků modelu:

1. klasifikátory (classifiers) obecně popisují strukturálních a behaviorálních vlastností libovolné sady prvků systému. Hlavními typy klasifikátorů jsou třídy, rozhraní a datové typy, ale mohou také byt uzly, komponenty, signály a další. Klasifikátory mají nejen atributy a operace, ale obrovské množství rozšířených vlastností [26].

2. události (events) popisují soubor možných jevů. Událost nějakým způsobem mění systém.

3. chování (behaviors) popisuje soubor možných činnosti. Činnost je nějaký druh akce, ke které dochází po určitou dobu a která může generovat události a nějak na ně reagovat.

Modely UML neobsahují samotné objekty, jevy nebo činnosti; obsahují pouze specifikace pro modelování objektů, jevů a činnosti v konkrétním kontextu.

Ačkoli UML je grafický jazyk, nikoli programovací jazyk, jeho nástroje lze použít ke generování kódu v různých jazycích pomocí diagramů UML. Diagram UML graficky představuje jakoukoli část navrženého nebo existujícího systému, obvykle znázorněnou jako připojený graf s vrcholy (entity) a hranami (vztahy).

Vztah (relationship) je abstraktní prvek a je obvykle zobrazen jako čára spojující související prvky. UML 2.5 definuje dvě podtřídy vztahů: asociace a řízené vztahy [25].

Asociace (association) - vztah mezi klasifikátory, který označuje, že instance klasifikátorů spolu souvisejí nebo jsou logicky nebo fyzicky kombinovány do sady. Asociace se používá v různých typech diagramů UML: binární

asociace mezi případem užití a aktérem v případě užití, asociace mezi artefakty nebo dvěma cíli nasazení v diagramu.

Řízený vztah (directed relationship) je abstraktní vztah mezi množinou zdrojových prvků a množinou cílových prvků.

Podtřídy směrovaného vztahu jsou: zobecnění, závislost, vazba šablony. Zahrnuje také řízené vztahy, které se používají pouze v diagramu případů užití jako variace vztahu závislostí mezi prvky případu užití tohoto diagramu, a nazývají se «include» a «extend» [27].

V případě dědičnosti (generalizace) jedna ze dvou souvisejících entit je zvláštním případem druhé a nazývá se class-child. Dítě zdědí strukturu a chování svého class-parent. Graficky je vztah dědičnosti zobrazen jako čára s otevřenou šipkou, která ukazuje na rodiče.

Závislost (dependency) je potřebná k vyjádření toho, že UML prvek závisí na dalších prvcích modelu a využívá určitou část jiné entity pro upřesnění své implementace nebo specifikace. V těchto vztazích mezi dvěma entitami se při změně nezávislé entity může změnit sémantika jiné závislé entity. Závislost je graficky znázorněna jako přímá přerušovaná čára.

Vazba šablony (template binding) je vztah mezi prvkem a šablonou podpisu (template signature) cílového šablonu - prvkem, který definuje uspořádanou sadu formálních parametrů pro cílovou šablonu. Tento vztah obsahuje a definuje sadu skutečných parametrů pro nahrazení místo formálních parametrů šablony. Vazba šablony je znázorněna jako přerušovaná čára s klíčovým slovem «bind» a informacemi o vazbě, s otevřenou šipkou, která ukazuje na šablonu [28].

Vztah «include» mezi dvěma případy užití ukazuje, že chování vloženého případu užití je součástí chování základního případu užití, přičemž bez vloženého případu užití základní případ užití není úplný. Tento typ vztahu je označen čárkovanou čarou, označenou nahoře klíčovým slovem «include», s otevřenou špičkou sahající od základního případu užití k vloženému případu užití.

Vztah «extend» určuje vztah nezávislého základního případu užití s jiným případem užití, které rozšiřuje základní případ užiti o své funkcionality pouze za určitých podmínek. Tento vztah je zobrazen jako přerušovaná čára s otevřenou šipkou, nad kterou je napsáno klíčové slovo «extend». Šipka ukazuje z rozšiřujícího se případu užití na základní případ užití.

Grafické symboly použité v diagramu určují jeho druh. Specifikace UML umožňuje kombinaci různých typů diagramů do jednoho, ale některé nástroje UML omezují použití sady možných grafických prvků v konkrétním typu diagramu.

Diagramy pomáhají vizualizovat systém z různých úhlů pohledu a také poskytují představu o prvcích, které tvoří systém. Stejný prvek lze navíc použít ve všech diagramech, v několika, co se stává nejčastěji, nebo ve vzácných případech v žádném diagramu. Specifikace UML definuje dva hlavní typy diagramů UML: strukturní diagramy a diagramy chování [25].

Strukturní diagramy vyjadřují statickou strukturu softwaru a jeho částí a také ukazují různé úrovně abstrakce, implementace systému a vzájemný vztah jednotlivých částí systému.

Strukturální diagram představuje různé objekty v systému. Prvky strukturálního diagramu jsou důležité části systému, odrážející abstraktní, reálné pojmy světa a implementaci. Strukturální diagramy neobsahují žádné podrobnosti o dynamickém chování, nepoužívají pojmy týkající se času [28].

2.3.1 Diagram tříd (Class diagram)

Třída (class) - kategorie entit, které mají společné atributy, operace, vztahy a sémantiku. Třída je graficky znázorněna jako obdélník, kde se zpravidla zapisuje její název, atributy a operace. Třída implementuje jedno nebo více rozhraní.

Interface (Interface) je klasifikátor, který představuje soubor činností, které se používají k definování služeb třídy nebo komponenty. Interface popisuje chování prvku třídy nebo komponenty zcela nebo zčásti. Nedefinuje implementaci operací, pouze určuje operace. Interface proto nemají atributy a stavy, ale mají operace a přijaté signály. Můžeme říci, že rozhraní odpovídá abstraktní třídě, která nemá žádné atributy ani metody, ale pouze abstraktní operace. Mezi rozhraními může existovat vztah typu zobecnění (dědičnost), v důsledku čehož class-child obsahuje všechny operace a signály svých předků, včetně svých vlastních, pokud existují.

Diagram tříd je nejběžnější ve vývoji systému, protože je hlavním způsobem popisu a zobrazení logické a fyzické struktury systému, jeho části nebo součásti ve formě tříd a zvláštních případů tříd. Aplikace se často generují z diagramu tříd. Třídy v tomto diagramu jsou zobrazeny jako související bloky. Diagram tříd také odráží omezení tříd a jejich vzájemné vztahy. Mezi hlavní typy vztahů, které se mezi nimi vytvářejí, patří zobecnění, asociace a závislost [27].

2.3.2 Diagram objektů (Object diagram)

Objekt (object) - instance třídy, nebo spíše klasifikátoru, což je konkrétní materializace abstrakce. Hodnoty atributů objektu jednoznačně identifikují objekt a určují jeho stav v určitém časovém okamžiku.

Objektový diagram odráží množinu instancí tříd a vztahy mezi nimi, které jsou instancemi asociací, v určitém okamžiku. Jedná se o speciální případ diagramů tříd, které mají spíše pomocnou povahu a poskytují přesnější představu o tom, jaké jsou systémové objekty a jejich vzájemné vztahy v určitém konkrétním bodě fungování systému. To pomáhá odhalit nedostatky v návrhu a poté je opravit [25].

2.3.3 Diagram komponent (Component diagram)

Komponenta (component) je fyzicky existující součást systému, s přesně definovanou sadou požadovaných a poskytovaných interface, která zajišťuje implementaci této sady interface. Komponentou může být spustitelný kód samostatného modulu, dávkové soubory, zdrojový kód programů a zcela jakékoli artefakty, které se používají k provozu aplikace. Artefakty mohou implementovat komponenty. Komponenty existují, když je program spuštěn. Graficky je komponenta reprezentována jako obdélník, uvnitř kterého musí být uveden explicitní název komponenty, se dvěma malými obdélníky vloženými vlevo [25].

Použití komponent je podmíněno tím, že už vytvořené komponenty lze znovu použít, a jelikož určitá komponenta podporuje konkretní sadu interface, lze ji bez poškození funkcionalit systému nahradit jinou komponentou, která podporuje stejnou sadu interface.

Diagram komponent je tvořen množinou komponent, které sestavují modelovaný systém, a vztahů mezi nimi. V tomto diagramu komponenty hrají roli spustitelných modulů nebo rozhraní pro určení vztahu mezi komponentami a objekty jsou součástí komponent.

Diagramy komponent se používají k vizualizaci a ke zjednodušení reprezentace rozsáhlého systému prostřednictvím jeho rozdělením na menší komponenty. Mezi prvky tohoto diagramu mohou existovat takové typy vztahů: komponenta implementuje rozhraní, komponenta používá rozhraní nebo objekt je součástí komponenty [27].

2.3.4 Diagram složených struktur (Composite structure diagram)

Tento diagram je podobný diagramu tříd, ale konkretizuje určité detaily tím, ze popisuje je podrobněji. Diagram složených struktur lze použít k zobrazení vnitřní struktury klasifikátoru a schématu interakcí klasifikátoru s prostředím prostřednictvím portů. Takový diagram se nazývá vnitřní strukturální diagram. Podmnožinou diagramu složených struktur UML je diagram kolaborace (collaboration diagram). Takový diagram se používá k demonstraci rolí a interakcí různých tříd v dané kolaboraci [26].

Kolaborace (Collaboration) je sbírka rolí a dalších prvků, které fungují společně, čímž vytvářejí nějaký druh kooperativního efektu, který je něčím větším než souhrnem jejich jednotlivých efektů. Prvky kolaborace spolupracují za účelem dosažení konkrétního cíle. Stejná třída se může účastnit několika kolaborací. Graficky je kolaborace znázorněna jako elipsa, ohraničená přerušovanou čarou, kde uvnitř je zapsán název kolaborace [25].

2.3.5 Diagram nasazení (Deployment diagram)

Diagram nasazení určuje fyzický hardware, na kterém bude softwarový systém spuštěn, a mapuje generovanou softwarovou architekturu na architekturu fy-

zického systému, který jej provozuje. Výsledkem je, že diagram ukazuje nejen složení a připojení systémových prvků, ale také jejich fyzické umístění na výpočetních prostředcích za běhu systému.

Artefakt může být také nasazen k provedení na výpočetním zdroji, který se nazývá uzel. Umístění nasazeného artefaktu se nazývá cíl nasazení. Uzel (node) je hardwarové zařízení nebo nějaký druh softwarového modulu runtime, který může vykonávat jeden nebo více artefaktů. Starší verze UML 1.x nasazovaly komponenty do uzlů, ale aktuální verze UML nasazuje artefakty do uzlů. Komponenty jsou nasazeny do uzlů bezprostředně prostřednictvím artefaktů [23, 24].

Mezi uzly může existovat asociační vztah, což ukazuje, že uzly jsou fyzicky propojeny za běhu systému. Graficky je uzel zobrazen jako rámeček, uvnitř kterého je uveden název artefaktu, který je nasazen na tomto uzlu [27].

2.3.6 Diagram balíčků (Package Diagram)

Balíček (package) je univerzální mechanismus pro uspořádání velkého počtu prvků modelu do skupin, přičemž prvky mohou být jakéhokoliv typy entit, včetně samotných balíčků [25]. Balíčky jsou výhradně koncepční a existují pouze během vývoje systemu. Vytvoření balíčků v UML umožňuje seskupit různé samostatné prvky do bloků, takže s nimi lze později manipulovat jako s jednotným prvkem. Stojí za zmínku, že správně vytvořený balíček by měl kombinovat funkčně a somaticky podobné prvky, které by bylo možné během vývoje systému společně změnit. Graficky balíček je zobrazen jako obdélníková složka se záložkou, ve kterou uvnitř je zapsán název balíčku. Obsah balíčku může být uložen do obdélníku složky.

Diagram balíčku se používá k vylepšení řízení složitosti samotného modelu a ukazuje vztah mezi velkými komponentami v rozsáhlých systémech. V něm se zobrazuji závislosti mezi balíčky, které tvoří model [26].

2.3.7 Diagram profilů (Profile diagram)

Diagram profilů umožňuje přidat nové vlastnosti a sémantiku pro diagramy UML, ale neposkytuje možnost odebírat žádná omezení vztahující se na metamodel. Profily (profile) pomáhají přizpůsobit metamodel UML pro různé konkrétní platformy a domény. K definování omezení specifických pro platformu nebo doménu se používají uživatelské stereotypy, hodnoty značek a omezení. Profily lze kombinovat a používat současně pro jeden konkrétní model.

Diagramy chování ukazují dynamické chování objektů v systému, které lze popsat jako posloupnost změn v systému v průběhu času. Tyto diagramy představují funkčnost systému a popisují, co by se v systému mělo dít [25].

2.3.8 Stavový diagram (State Machine Diagram)

Stav (state) je určitá doba v životním cyklu objektu, během které splňuje nějakou podmínku, vykonává činnost nebo čeká na událost. Stav je určen hodnotami některých atributů objektu a přítomností nebo nepřítomností vztahů s jinými objekty [1].

Stavové diagramy se liší od ostatních diagramů tím, že se zaměřují pouze na jednu instanci konkrétní třídy a popisují proces změny jejího stavu. Cílem použití tohoto diagramu je lepší znázornění toho, jak jsou sladěný a chovají se složité objekty, proto tento diagram ukazuje, jak objekt přechází z jednoho stavu do druhého. Diagram simuluje životní cyklus objektu a odráží změnu objektu v závislosti na vnitřních a vnějších faktorech.

Přechod (transition) v UML je vztah mezi dvěma stavy, který zobrazuje, že objekt v prvním stavu musí provést určené akce a přejít do druhého stavu. K přechodu obvykle dochází po skončení výkonu činnosti, přijetím objektem zprávy nebo signálu [27].

Stavový diagram používá standardizované podmíněné označení. Stavy objektu během jeho životního cyklu jsou zobrazeny jako zaoblené obdélníky.

Přechody mezi stavy jsou zobrazeny šipkami. Pokud existuje událost, která způsobila určitý přechod, je vedle šipky uveden její název. Za název události lze přidat ochrannou podmínku v hranatých závorkách a akci, která se při přechodu provádí. Ochranná podmínka (guard condition) je logický výraz který musí být pravdivý aby došlo k přechodu.

Plně vyplněný kruh označuje počáteční stav. Objekt je v počátečním stavu ihned po vytvoření. Kruh s malým vyplněným kruhem uvnitř označuje konečný stav objektu. Pokud objekt přešel do konečného stavu, nebude již moct svůj stav změnit [29].

2.3.9 Diagram aktivit (Activity diagram)

Aktivita (activity) - je spustitelné chování které specifikováno jak sériové nebo paralelní provádění samostatných činností (action) - některých nedělitelných pracovní prvků spojené tokem řízení, kde toky přicházející z výstupy jednoho uzlu na vstupy jiného uzlu. Činnost je atomární operace, jejíž provedení nelze přerušit. Činnosti se zobrazuji na diagramu jako obdélníky se zaoblením. Řídicí uzel (control node) je abstraktní uzel činnosti, který koordinuje řídicí toky.

Diagram aktivity je zobrazen jako blokové schéma a slouží k modelování toku řízení z jedné činnosti do druhé. Tok řízení (control flow) je postupné provádění příkazů v nějaké aktivitě a zobrazuje v aktivitě cestu mezi uzly pomoci šipek. Diagram aktivity ukazuje nejen tok řízení v rámci jednoho systému, ale také z jednoho systému do druhého, pokud má aplikace více než jeden systém.

V tomto diagramu musí byt uveden konkrétní jednotlivý začátek aktivity - počáteční uzel (activity initial node) a alespoň jeden konec - koncový uzel aktivity (activity final node), které ukončuje všechny toky řízení. V koncovém

uzlu toku (flow final node) se ukončuje konkretní tok, co nemá vliv na ostatní toky v diagramu. Uzel začátku toku aktivity vypadá na diagramu jako černý kroužek, uzel konce aktivity jako menší černý kroužek ve vetším kroužku, uzel konce toku jako kroužek s křížkem uvnitř.

Tok může být rozdělen na několik paralelních toku v uzlu rozvětvení (fork node), do kterého jde přesně jeden tok, ale z něho vychází více než jeden tok. Paralelní toky lze sloučit v uzlu spojeni (join node), přičemž se toky synchronizují, to znamená, že každý z toku musí počkat, dokud všichni ostatní nedosáhnou uzlu spojeni, po kterém proces pokračuje ve jednotlivém toku. Pro zobrazeni těchto uzlu na diagramu se používá synchronizační čára, do který vede jedna šipka a vychází několik šipek případně naopak [26].

V uzlu sloučení (merge node) se spojuji několik alternativních toku, navíc je zvolen jeden z těchto toku, který bude pokračovat dále. Při použiti rozhodovacího uzlu (decision node) lze zvolit podmínku na základě kterou se vybere tok, který bude pokračovat. Teto uzly na diagramu vypadají jako kosočtverec, do kterého vedou šipky s tím rozdílem, ze v případě uzlů sloučení z kosočtverce vždy vychází jedna šipka a v případě rozhodovacího uzlu vychází alespoň dvě nad kterými v hranatých závorkách je napsána podmínka pro přechod.

Můžeme říci, že diagram aktivit je popisem algoritmu pro provoz systému, kde by měl být algoritmus chápán jako určitá řada konkrétních činnosti nebo operací, které následují v určitém pořadí, jejichž provedení vede k jasně definovanému výsledku.

2.3.10 Diagramy případů užiti (Use Case Diagram)

Diagram případů užití poskytuje přehled funkčnosti systému a odráží to, co systém dělá ve vnějším světě. Tento diagram popisuje, co systém dělá, ale neuvádí, jak na to. S jeho pomocí se v prvních fázích návrhu systému stanoví obecné hranice tematické oblasti a připravuje se úvodní dokumentace, která je nezbytná pro lepší komunikace mezi vývojáři systému a zákazníkem. Diagram případu užití lze označit jako počáteční koncepční model systému, který následně usnadňuje detailizaci v podobě fyzických a logických modelů [30].

Hlavní prvky diagramu případů užití jsou dva druhy entit: případy užití a aktéři, mezi nimiž se obvykle vytvářejí vztahy, jako jsou závislosti mezi případy užití, zobecnění mezi případy užití nebo aktéry a asociace mezi případem užití a aktérem [25].

Kromě toho lze v diagramu případů užití použít speciální grafický komentář, který slouží k označení hranice systému, pokud to z nějakého důvodu není zřejmé. Aktéři se nachází za hranicemi systému a případy užití uvnitř.

Aktér (actor) je vnější entita, což znamená ze tato entita bezprostředně interaguje se systémem, ale není její části. Aktér může to být jakýkoli objekt, předmět, technické zařízení, program nebo jiný systém, který interaguje zvenčí se simulovaným systémem a ovlivňuje tento systém způsobem, který nastavuje vývojář systému. Každý aktér definuje sadu rolí, které uživatel může

mít v procesu interakce se systémem. V určitém okamžiku hraje určitý uživatel pouze jednu roli z této sady. Role (role) je funkce, kterou vykonává entita. Lze uvažovat každého aktéra jako samostatnou roli ve vztahu ke konkrétnímu případu užití.

Standardní grafické označení aktéra v diagramu případu užití je "človíček", pod kterým je uvedeno jeho jméno. Jméno aktéru by nemělo být vlastním jménem nebo konkrétním jménem, protože taková osoba může hrát několik rolí najednou. Takže požadavkem na název je vysoce sémantický informační obsah [22].

Případ užití (use case) popisuje postupně prováděné systémem činnosti, který systém vykonává po požadavků aktéra, a navíc řada těchto činností musí být konečná. Po zpracování požadavku se systém vrací do původního stavu a může provést následující požadavky.

Případ užití specifikuje obecné charakteristiky chování nebo fungování systému, ale nezohledňuje vnitřní strukturu systému. Každý případ užití závazně určuje posloupnost činností, které musí systém provést jako reakci na komunikaci s aktérem, ale činnosti nepopisuje. Jinými slovy, nic se neříká o tom, jak bude realizovaná implementace případů užití nebo interakce aktérů se systémem [26].

Případ užití je graficky znázorněn jako elipsa s jeho jménem uvnitř. S cílem vysvětlení sémantiky činnosti při provádění konkrétního případu užití lze doplnit další text, který se nazývá scénář. Diagram případů užití by měl definovat všechna možná chování systému prostřednictvím konečné množiny případů užití, které lze pro snadnější použití seskupit do samostatného balíčku [31].

2.3.11 Interakční diagramy

Interakční diagramy se používají k popisu toku zpráv v systému, interakce mezi objekty a k zachycení dynamického chování systému. Interakční diagramy zahrnují několik různých typů diagramů:

- diagram časování,
- sekvenční diagram,
- diagram komunikace,
- diagram přehledu interakcí.

2.3.12 Diagram časování (Timing diagram)

Diagram časování existuje od verze UML 2.0 a používá se, když je čas v projektovaném systému důležitý. Tento diagram odráží změnu stavů objektů v důsledku interakce v průběhu času, přičemž je třeba věnovat pozornost době

trvání událostí. Diagram časování nevysvětluje, jak objekty interagují, ale popisuje změny, ke kterým dochází v závislosti na časových omezeních.

Diagramy časování kombinují stavové a sekvenční diagramy. Pokud je v diagramu časování mnoho stavů, jsou změny stavu indikovány křížením vodorovných čar. Pokud počet stavů je malý, změny se zobrazí jako přechod z jedné vodorovné čáry do druhé. Limity časování jsou uvedeny ve složených závorkách [25].

2.3.13 Sekvenční diagram (Sequence diagram)

Sekvenční diagram zobrazuje výměnu zpráv mezi více čárami života a modeluje posloupnost zpráv mezi objekty v průběhu času v konkrétním případě užití převzatém z diagramů případů užití.

Čára života (lifeline) označuje dobu, po kterou existuje objekt v systému a asociuje se s jedním konkrétním objektem v sekvenční diagram. Čára života je zobrazena jako přerušovaná svislá čára, která se táhne směrem dolů od objektu. Objekt má tvar obdélníku se svým názvem uvnitř. K označení konce existence objektu v procesu interakce používá tučný šikmý kříž na místě ukončení čáry života.

V tomto kontextu interakce (interaction) označuje takové chování objektu, v důsledku čehož si vyměňuje zprávy s jiným objektem pro dosažení konkrétního cíle.

Zpráva (messange) definuje požadavky na výměnu dat mezi objekty. V důsledku výměny zprávy jsou přenášeny některé informace, jejichž cílem je vyvolat jako odpověď reakci, která povede k určité činnosti. Zpráva je zobrazena jako šipka se špičkou, kde šipka přechází od odesílatele k příjemci [26].

Sekvenční diagram zobrazuje pouze ty objekty, které jsou obsaženy v analyzovaném případu užití, a zbývající objekty se skrývají. Rovněž jsou vyloučené jakékoliv jiné vztahy, kromě těch, které jsou bezprostředně nutné pro přenos dané sekvence zpráv [25].

2.3.14 Diagram komunikace (Communication diagram)

Diagram komunikace je sémanticky ekvivalentní sekvenčnímu diagramu a také popisuje sekvenční výměnu zpráv mezi množstvím objektů. Pomocí nástrojů lze snadno převést sekvenční diagram na diagram komunikace. Ale diagram komunikace se zaměřuje na vztah mezi objekty, které se účastní interakce, což zobrazuje strukturální organizaci objektů vyměňujících si zprávy, zatímco sekvenční diagram se zaměřuje na pořadí interakcí v čase. Na diagramu komunikace se čas nezohledňuje, ale projevuje se pouze ve formě chronologicky očísloványch zprav. Šipce zprávy je přiřazeno pořadové číslo, které pomáhá udržovat správné pořadí zpráv.

Kromě běžných prvků, jako jsou instance zpráv, aktérů a objektů, diagram také zobrazuje nepřímé asociace mezi účastníky interakce. Nad těmito vztahy jsou uvedené zprávy, které účastníci vysílají [26].

2.3.15 Diagram přehledu interakcí (Interaction Overview Diagram)

Diagram přehledu interakcí je forma interakčního diagramu v modelovacím jazyce UML. Tento diagram sjednocuje tok řízení mezi uzly, které jsou obsaženy v diagramech aktivit, a posloupnost zpráv mezi čárami života sekvenčních diagramů. Diagram přehledu interakcí je založen na diagramu aktivit, kde uzly jsou nahrazeny sekvenčními diagramy, protože samotné sekvenční diagramy jsou nevhodné, aby odrážely složité chování větvení [7].

V diagramu přehledu interakce nejsou zohledněny některé jednotlivé aspekty fungování systému, ale naopak je analyzována obecná povaha interakcí ve vytvářeném systému [4].

Většina označení v diagram přehledu interakcí je stejná jako v diagramech aktivit. Notace jsou převzaté ze sekvenčního diagramu. Vlastní nové prvky tohoto diagramu jsou interakční prvky (interaction elements) a interakční případy (interaction occurrences), které jsou odkazy na existující diagramy aktivit. Interakční prvky navíc zobrazují obsah diagramu aktivit vestavěný v obdélníkovém rámečku.

2.4 Modely MDA

Architektura MDA je založena na strukturovaném procesu vývoje systému krok za krokem, vytvořením a použitím několika typů modelů v různých fázích. Každý model je postupně transformován z předchozích a doplněn o nové detaily, to znamená, že původní model je postupně transformován do jiného v rámci stejného systému. Přičemž dochází k sekvenčnímu přechodu od abstraktních informačních modelů, které neobsahují žádné informace o implementaci na konkrétní technologické platformě, k konkrétním modelům, které obsahují právě tyto informace, s následným generováním programového kódu systému [14].

MDA definuje tři modely systému, které odpovídají modelové pohledům MDA. MDA odděluje obchodní zájmy od technologických. Rozdělení na několik modelů umožňuje flexibilní správu změn se zaměřením na konkrétní aspekt systému a snižování vzájemné závislosti prvků. Při vývoji systému, který je vytvořen v architektuře MDA, vývojáři mají k dispozici tři (výchozí pohledy) modelové pohledy modelu.

Pohled (viewpoint) je technika abstrakce, která potlačuje nevýznamné detaily, aby se mohla zaměřit na konkrétní sadu problémů v systému, a lze ji znázornit pomocí modelů.

Computation Independent Viewpoint (CIV) se zaměřuje na analýzu systémových požadavků a prostředí systému s přihlédnutím k softwarovým, hardwarovým a dalším technologickým omezením. Tento pohled nebere v úvahu žádné podrobnosti o jeho struktuře a procesech [8].

Platform Independent Viewpoint (PIV) skrývá implementační podrobnosti systému specifické pro platformu a zaměřuje se na konkrétní funkční prvky systému, které jsou stejné, když systém interaguje s jakoukoli platformou. Části úplné specifikace systému lze z této platformy abstrahovat.

Platforma (platform) je sestava subsystémů a technologií, které představují jednu sadu funkcí založenou na rozhraních a šablonách použití používaných jakoukoli aplikací bez zadání podrobností implementace. Příklady platforem jsou programovací jazyky, operační systémy, databáze atd. Platformy lze popsat jako modely MDA, které pak používají jiné platformy. Model se nazývá nezávislý na platformě, pokud je nezávislá na vlastnostech jakékoli platformy [14].

Platforma hraje důležitou roli při implementaci technologie, protože podporuje aplikaci. Aplikace je funkční součástí systému, která je popsána pomocí modelu. Spolu s aplikací tvoří platforma systém.

Platform Specific Viewpoint (PSV) definuje podrobnosti implementace a procesy, které závisí na konkrétních použitých platformách a programovacích jazycích. PSV koreluje nezávislou na platformě logiku s dalšími funkcemi, které vznikají s konkrétní platformou nebo systémem [7].

2.4.1 Model nezávislý na počítačovém zpracování (CIM)

Proces vývoje začíná vytvořením modelu nezávislém na počítačovém zpracování (CIM). Nazývá se také obchodní model, protože představuje pouze přesně to, co by měl systém dělat, bez ohledu na to, jak bude systém implementován.

V současné době MDA nepoužívá model nezávislý na počítačovém zpracování deklarovaný ve specifikacích přístupu, obdobu technické úlohy. Existují samostatná řešení, která implementují tento přístup pro databázové aplikace (AndroMda, Bold for Delphi) a webové služby a portály.

Modely CIM definují obecné systémové požadavky a zahrnují obecné doménové slovníky. V těchto modelech není popsáno nic technického charakteru, aby byla zajištěna maximální nezávislost systému na způsobu implementace.

Model CIM není pro proces vývoje aplikace povinný, i když je žádoucí pro zmenšení počtu chyb. Implementace programového kódu pouze na základě požadavků zákazníka je možná pouze u malých projektů. Vzhledem k tomu, že zákazník není odborníkem v této oblasti, nemusí rozumět potenciálním problémům a některé procesy považuje za zřejmé a automatizované. Může si být vědom toho, co bude v systému dělat ručně, ale nemusí být schopen podrobně pochopit, co bude dělat samotný systém. Nakonec může být projekt mnohokrát přepracován.

Všechny součásti modelu CIM by měly být dobře analyzovány. Jejich existence a nutnost musí být jasně odůvodněna. Model by neměl mít zbytečné prvky, pouze té prvky, které budou implementovány v dalších fázích. Požadavky uvedené v modelu CIM musí být vysledovány na modely PIM a PSM [14].

V této fázi lze k vytvoření CIM použít jakékoli prostředky. Podmínkou pro vytvoření modelu CIM podle standardu MDA je požadavek, aby byl CIM postaven tak, aby mohl být transformován do modelu nezávislého na platformě PIM.

Pro kompatibilitu s následujícími kroky je však velmi žádoucí mít popis modelu v jazyce UML. UML modelům navíc obvykle rozumí nejen vývojář, ale také zákazník, což může zjednodušit komunikaci mezi nimi za účelem upřesnění detailů během procesu vývoje.

Ke zvýšení efektivity práce na této fázi je možné použít různé softwarové nástroje pro ontologické modelování a CASE-nástroje (například IBM Rational Rose atd.), které umožňují automatizovat opětovné použití a transformaci modelů [32].

2.4.2 Model nezávislý na platformě (PIM)

Ve druhé fázi je vyvinut model nezávislý na platformě (PIM). V této fázi probíhá převod CIM na PIM na základě popisu v jazyce UML, vytvořeného v první fázi, s možným upřesněním některých detailů. Při absenci modelu CIM je vyvíjen od začátku.

Během transformace se provádí automatická formalizace modelu domény, vztahy, příčiny a následky modelu CIM se transformují do pravidel. V důsledku tohoto procesu jsou koncepty CIM mapovány na šablony faktů a prvky logických pravidel.

PIM je popsán v UML, což dává smysl, protože UML byl původně navržen jako jazyk nezávislý na platformě. UML ve svých raných verzích nebyl dostatečně zdokonalen a umožňoval pouze popsat strukturu a chování systému, aniž by definoval jakékoli algoritmy nebo fungování. Ale počínaje standardem UML 2.0. zahrnoval prostředky, kterými bylo možné popsat fungování systému. UML se stal spustitelným jazykem, ale úroveň podrobností modelů UML se nezvýšil [8].

Model PIM popisuje obecnou strukturu systému, jeho funkčnost a prvky, požadavky na uživatelské rozhraní. Model PIM má představovat algoritmus, přičemž nezohledňuje funkce softwaru, který bude použit pro implementaci, protože design je maximálně oddělen od implementace. Množství informací o modelu není omezeno, ale nemělo by být relevantní pro implementaci systému na konkrétní platformě.

PIM je dostatečně nezávislý na to, aby jej bylo možné v budoucnu mapovat na jednu nebo více platforem.

S pomocí modelu PIM je možné testovat systém na úrovni systémových požadavků a technických specifikací, a to ještě před zahájením praktické implementace, od prvních fází. Je spustitelný a má k tomu dostatečnou úroveň podrobností. Tato funkce je významnou výhodou, protože vám umožňuje identifikovat chyby v raných fázích návrhu, které by bylo obtížné opravit po implementaci prototypu systému.

PIM pomáhá implementovat jednu z důležitých vlastností MDA - future proofing. Se vznikem nových technologií a platforem v budoucnu je možné znovu implementovat a upgradovat systém na nich založený bez větších obtíží nebo nákladů, pomocí PIM a odpovídajícího mapování - převedením PIM na konkrétní PSM.

V souladu s přístupem MDA je po skončení této fáze výsledkem popis PIM v jazyce UML, který zcela popisuje systém bez uvedení jakýchkoli podrobností o implementačních technologiích [14].

2.4.3 Model specifický pro konkrétní platformu (PSM)

Ve třetí fázi jsou vytvořeny modely specifické pro konkrétní platformu (PSM), jejichž počet odpovídá počtu platforem, na kterých by aplikace měla fungovat. V případě potřeby může systém pracovat na několika platformách současně.

Také v MDA existuje možnost vytvoření heterogenního systému. Jeho části mohou vzájemně interagovat, i když fungují na různých platformách middlewaru. PSM je vytvořen transformací PIM tak, aby odpovídal požadavkům modelu platformy.

Konverze z PIM na PSM je standardizována pomocí speciálně navržených standardních mapování specifických pro každou platformu middlewaru. Během tohoto procesu jsou do modelu přidány informace o podrobnostech praktické implementace na vybrané platformě. Pokud PSM neobsahuje takové informace, které jsou nezbytné k implementaci systému na dané platformě, považuje se to za abstraktní a spoléhá se na jiné explicitní nebo implicitní modely, které obsahují tyto podrobnosti. Konstrukce heterogenních systémů má vysokou úroveň automatizace díky popisu interakce prvků systému, který je obsažen v PIM.

Vývoj je založen na modelu systému, který obsahuje informace o fungování, struktuře a interakci částí systému, což umožňuje rozdělit systém na několik částí při zobrazení na PSM a velmi zjednodušit proces integrace heterogenních technologií. Pro každý fragment systému je vytvořen jeho vlastní model PSM. Sada nástrojů, která provádí toto mapování, je navíc schopna analyzovat interakci systémových částí na základě modelu PSM a vytvářet popisy rozhraní, mostů, mediátorů, které jsou potřebné pro jejich interakci mezi platformami.

Mapování z PIM přes PSM na podporované platformy MDA se provádí pomocí nástrojů, které tento úkol usnadňují. Transformace mohou také zjednodušit a automatizovat nástroje, jako jsou analyzátory a deskriptory modelů.

Převod z PIM na PSM v MDA je nejnáročnějším krokem, protože na teto fáze obecný popis systému v jazyku UML se stává vhodným pro vytváření aplikace na konkrétní platformě [14].

Transformace zahrnuje tři po sobě následující fáze:

- Vývoj mapovacího schématu (mapping)
- Značení (marking)
- Samotná transformace (transformation)

Pokud je však PIM založen na virtuálním stroji, PIM virtuálního stroje je převeden na PSM pro konkrétní platformu a poskytuje všechny informace potřebné k vytvoření systému. V tomto případě není nutná žádná jina transformace.

Prvním přístupem k mapování je mapování typů modelů (Model Type Mappings). Mapování typů modelů určuje, jak jsou modely vytvořené pomocí typů PIM převedeny na typy PSM. Při vytváření modelu PIM se navíc prvky pro konstrukci vybírají podle požadavků aplikace. Mapování poskytuje pravidla a algoritmy pro převod instancí typů PIM na instance typů PSM.

Nevýhodou tohoto průchodu však lze nazvat vždy stejný výsledek, protože aplikace se bude vždy spoléhat pouze na model PIM. Zobrazení nezávisí na žádných dalších faktorech, výsledkem je, že se stejnými vstupními daty se vždy získají stejná výchozí data, to znamená, že zobrazení je deterministické.

Přístup mapovaní instance modelu (Model Instance Mappings) je založen na potřebě převést určité prvky PIM na prvky PSM konkrétním způsobem, v závislosti na konkrétní platformě. Vyvinuté transformační schéma přímo závisí na schopnostech platformy a ovlivňuje kromě obsahu modelu i samotný model, včetně jeho metamodelu a typů v něm použitých. V tomto schématu odpovídají typy modelů, vlastnosti a prvky metamodelů PIM typům modelů, vlastnostem a prvkům metamodelů PSM.

Obvykle k provedení konverze nestačí jednoduché porovnávání typů. Proto je nutné označit určité typy v PIM speciálními značkami, ktere ukazuji, že konkrétní prvek musí být převeden. Informace o tom, zda má být daná položka označena, nelze získat od PIM. Proto se provádí značení - proces vytváření značek. Tento proces používá informace o platformě, které jsou obsaženy v modelu platformy. Model platformy je obvykle prezentován jako soubor technických konceptů, které popisují technické vlastnosti, rozhraní a funkce platformy, požadavky na připojení částí platformy a aplikace k platformě. V architektuře MDA musí být popis takového modelu implementován v jazyce UML.

Značky jsou samostatné datové struktury, které patří do transformačních schémat a obsahují informace o vytvořených odkazech. Lze je kombinovat do tematických šablon pro použití v různých schématech transformace. Značky jsou specifické pro každou platformu. Značky i PIM jsou nezbytné pro proces transformace vedoucí k PSM.

Při umisťování značek, aby srovnání mělo smysl, musí architekt dodržovat určitá pravidla. Každé mapování modelu má implicitní omezení typu a pro mapování každé položky modelu v PIM lze použít specifické štítky. Kromě toho existují sady štítků, které při použití na stejný prvek modelu vytvářejí vzájemně se vylučující mapování.

Není nutné označovat prvek, se kterým budete muset provést transformaci, lze popsat požadavek na prvek s kterým se provádí proces mapovaní a poté během procesu transformace vybírají se takové prvky, které tento požadavek splňují. Tyto dva typy mapování lze kombinovat a vytvořit tak sofistikovanější a přesnější transformaci PSM na PIM.

Párování lze provést pomocí šablon. Šablony jsou parametrizované modely, které mohou definovat určená mapování, která obsahují informace pro vedení transformace. K označení prvků, které mají být porovnány podle vzoru, můžete použít značky spojené se vzorem [14].

Popis mapování jednoho modelu na jiný je uveden v přirozeném, speciálním zobrazovacím jazyce nebo v jazyce akce. Doporučuje se, aby byl jazyk přenosný, aby jej bylo možné použít s různými nástroji. V rámci MDA se doporučuje použít sadu specializovaných transformačních jazyků QVT (Query / View / Transformation) [33].

Dotaz (Query) je nějaký výraz aplikovaný na původní model, jehož provedení má za následek jednu nebo více instancí typů v tomto modelu. Obvyklým účelem použití dotazů v QVT je definování filtrů. Dotazy lze také vytvářet pomocí operační sémantiky.

Pohled (View) je model, který je odvozen od základního modelu, který popisuje, jak by měl vypadat cílový model. Lze jej nazvat zvláštním případem transformace, kde základní a cílové modely spolu úzce souvisejí. Nelze jej změnit samostatně, a pokud je provedena změna původního modelu, odpovídající změny se projeví v pohledu.

Obvykle je takový model jen pro čtení, ale pokud je pro zobrazení k dispozici úprava, všechny provedené změny se projeví v původním modelu. Aby bylo možné úpravy provádět, musí být transformace obousměrná, to znamená, že je povolena modifikace pro zdrojový i cílový model. Navíc, pokud se pokusíte provést změny ve dvou modelech současně, mohou nastat konfliktní situace. Pokud je transformace jednosměrná, je zobrazení jen pro čtení. Reprezentace může být založena na úplném původním modelu nebo na jeho určité podmnožině [33].

Transformace (transformation) je proces automatické generování, který vám umožňuje získat cílový nezávislý nebo závislý model z originálu v souladu s pravidly transformace. Pravidlo obsahuje popis, který určuje mapování mezi prvky zdrojového a cílového modelu, nebo implementaci, která určuje akce, kterými jsou prvky cílového modelu vytvořeny z původního modelu. Pravidlo může obsahovat popis a implementaci současně. Pravidla se používají k transformaci konkrétní podmnožiny prvků zdrojového modelu na odpovídající prvky cílového modelu [34]. Generovaný nezávislý model nemá nic společ-

ného s původním modelem, zatímco závislý model je v důsledku transformace přidružen k základnímu modelu.

Po procesu vývoje schématu převodu probíhá proces značení. Při použiti mapovaní typu instance modelu musí architekt označit položky PIM, které chce zobrazit v PSM. V nejjednodušších případech je jeden prvek PIM označen jedním prvkem PSM. Ve složitějších případech může mít jeden prvek PIM několik značek z různých schémat transformace. Ve výsledku se tagované prvky modelu a metamodely PIM přenesou do modelu a metamodelu PSM. Položka PIM může mít více než jeden štítek, což znamená, že se účastní více mapování a bude transformována podle každého z nich.

V mapovaní typů modelu se popis mapování cílového modelu automaticky generuje pomocí pravidel a algoritmů určených pro konkrétní typy modelů. Od uživatele mohou být požadovány další informace, které nelze získat z původního modelu. Schematům mapování modelu lze uložit a použít pro další transformace po provedení nezbytných změn v schématu [7].

Dalším krokem je převod označeného PIM a jeho mapování na PSM ručně, automaticky nebo pomoci počítačových nástrojů. Stojí za zmínku, že pomocí nástroje MDA můžete okamžitě generovat kód bez nutnosti generovat model PSM.

Proces transformace je také dokumentován ve formě mapy přenosu prvků modelu a metamodelů. Tato mapa ukazuje, jak spolu souvisí prvky PIM a PSM, to znamená, že se ukazuje, ze kterého prvku PIM byl prvek PSM transformován a jaké mapování bylo použito. Je možné provést změny v PIM a PSM a poté synchronizovat modely pomocí nástrojů pro modelování MDA, které vytvářejí mapu transformace. Dokumentace MDA popisuje čtyři hlavní přístupy k převodu PIM na PSM:

- Ruční
- Používání profilů
- S přizpůsobeným schématem převodu pomocí šablon a značek
- Automaticky

Ruční přístup se používá, když je třeba transformaci zvážit v kontextu konkrétního projektu. Při použití MDA má manuální přístup dvě významné výhody: schopnost jasně vidět rozdíly mezi modely PIM a PSM a možnost získat mapu transformace po její dokončení.

Druhý způsob jak PIM lze převést na PSM pomocí profilu UML specifického pro konkrétní platformu, který obsahuje mapování PIM pomocí značek specifických pro platformu a pravidel převodu určených pomocí operací definovaných v profilu UML.

Mezitím zobrazení lze provést pomocí vzoru a značek odpovídajících prvkům tohoto vzoru. Je možné kombinovat několik vzorů a naznačit nové

značky. V přístupu převodu typu modelu budou všechny položky v PIM, které odpovídají určitému vzoru, převedeny na položky v jiném vzoru, aby se vygeneroval PSM. Štítky spojují parametry vzorů položek PIM s položkami generovanými z nich v PSM. Při přístupu mapovaní instance modelu značky které jsou obsažené ve vzoru používají k vytvoření schématu transformace pomocí procesu značkování PIM a následného generování modelu PSM.

Poslední způsob je automaticky. Nyní pomocí nástrojů MDA automatizováno 50–70% převodu PIM na PSM a automatizace převodu PSM na kód je téměř 100% [35].

V některých případech je díky nástrojům MDA generování kódu možné přímo z PIM, který obsahuje všechny informace potřebné pro implementaci. V takových případech poskytuje middleware kompletní sadu požadovaných služeb. Takovými architektonickými řešeními jsou vytvořené šablony, další nástroje, generátory kódu a knihovny programů. Ve výsledku je vytvořen výpočetně úplný model PIM, který zahrnuje kompletní popis struktury a klasifikací a chování modelu je určeno jazykem akce.

Je třeba poznamenat, že kterýkoli z uvedených přístupů transformace MDA lze použít nejen pro transformaci modelu PIM na model PSM, ale také pro obecné transformace z modelu na model. Po dokončení fáze transformace obsahuje výsledný dokončený model závislý na platformě všechny technické informace nezbytné pro vygenerování systémového kódu. Zahrnuje také podpůrný kód a popisy, které umožňují použití zvolené platformy a technologií. Výsledný PSM může také fungovat jako PIM, který má být převeden na ještě podrobnější PSM, který bude přímou implementací.

Posledním krokem v procesu převodu je skutečné vygenerování implementačního kódu i artefaktů. Generování kódu může být částečným nebo úplným v závislosti na užitečnosti a kvalitě sady nástrojů MDA [8].

Kapitola 3

Analýza WordPress

3.1 Popis systému WordPress

WordPress (WP) je nejpoužívanější open source redakční systém pro vytváření a administrace webových stránek. Pro analýzu WP byla využita dokumentace [36], zdrojový kód WP používaný k instalaci [37], popis databáze WP [2], databáze WP vlastní instalace, oficiální web WP [38], kniha o WP [39] a informativní web založený jedním z autorů WP [40].

Redakční systém je program, který je nainstalován na hosting a usnadňuje správu webu. Hosting je úložiště, na kterém jsou umístěny soubory webu. Základem WP jsou sada souborů a databáze.

Kořenový adresář WP obsahuje tři složky: `wp-content`, `wp-includes` a `wp-admin` a sadu různých PHP souborů, které realizují základní operace WP. Složka `wp-content` obsahuje všechna nahraná uživatelská data. V složce `wp-includes` jsou uložené všechny potřebné soubory pro spuštění aplikace WP prostřednictvím uživatelského rozhraní. Složka `wp-admin` obsahuje soubory CSS, JavaScript a PHP, které realizují funkčnost konzole a administrativní části webu.

Databáze je úložiště, který obsahuje všechny dynamické informace: obsah, komentáře, uživatelé, rubriky, štítky a vlastně všechny informace, které uživatel může změnit.

WP byl vytvářen pro běžné uživatele, aby mohli bez znalosti programování ovládat vytvořené pro vlastní potřeby webové stránky. Proto WP má zjednodušenou logiku interakce s uživatelem a umožňuje intuitivně ovládání webových stránek bez čtení manuálů. WP poskytuje zdarma širokou řadu funkcionalit, které jsou pravidelně aktualizované, přičemž si během svého vývoje WP zachovává základní principy správy obsahu.

Základními elementy, ze kterých se skládá webová stránka vytvořená pomocí WP, jsou stránky (pages) a příspěvky (posts), komentáře (comments), pluginy (plugins), témata (themes), widgety (widgets) a média (media). Přestože záznamy a stránky mají podobné uživatelské rozhraní, poskytují různé

možnosti publikování obsahu na webu.

Příspěvky jsou dynamické stránky webu, které jsou uspořádané chronologicky, přičemž nové záznamy jsou vždy raženy výše než staré. Pro příspěvky lze vytvářet rubriky a štítky a pak je seskupit do kategorií. Stránky jsou statickým obsahem webu, takže nejsou zobrazeny chronologicky. Stránky se používají k reprezentaci obecných informací, jako jsou kontakty, o webu, pravidla webu. Stránky nemají rubriky a štítky.

Základní funkce WP lze rozšířit instalací pluginů – zásuvných modulů. Na webové stránky WP lze nainstalovat plugin poskytující libovolnou funkcionalitu nebo vytvořit vlastní plugin, když takový neexistuje.

Téma je sada souborů, která spravuje vizuální část webové stránky. Téma určuje design webu, tedy všechny obrázky mimo obrázků v tele stránky nebo příspěvku, pohyblivé prvky, formy komentování a podobně. Stejně jako pluginy je témata potřeba nainstalovat a je možné vytvořit vlastní.

Widgety jsou malé pohyblivé bloky kódu, které lze umístit na libovolnou stránku webu WP a tím přidat určitou funkcionalitu pro tuto stránku. Rozdíl mezi widgetem a pluginem je v tom, že widget vždy realizuje funkcionalitu, kterou vidí koncový uživatel, ale plugin může dělat něco, co se nezobrazuje na stránkách webu.

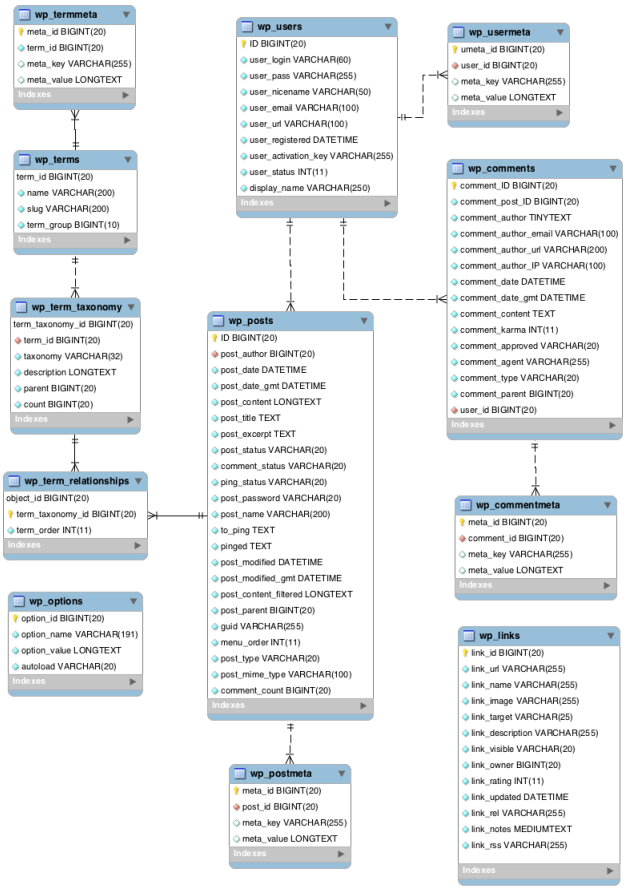
Vlastnosti WP je nezávislost elementů kontextu od designu webu. To znamená, že WP umožňuje pozměnit vizuální vzhled webové stránky, ale ponechat obsah webu nezměněným.

3.2 Strukturální model jádra CMS WordPress

Vzhledem k tomu, že WP je používán především jako systém pro správu obsahu webových stránek, databáze, kde je uložen obsah webu, má hlavní význam. V databázi WP všechna uživatelská data jsou uložena do složek, obsahující určitý typ informace z databáze. Je možné přidat nové složky, ale složky vytvořené při instalaci systému jsou vždy obsažené v databázi WP.

Databáze WP obsahuje 12 tabulek [2], jak je zobrazeno na obrázku 3.1:

- V tabulce `wp_posts` jsou uložené stránky, příspěvky a informace o médiích.
- WP používá tabulku `wp_postmeta` pro ukládání informací o stránkách, příspěvcích a média. Nejčastěji táta tabulka je používána pluginy.
- Tabulka `wp_users` ukládá informaci o registrovaných uživatelích.
- Tabulka `wp_usermeta` obsahuje doplňkovou informace o registrovaných uživatelích.
- Tabulka `wp_comments` ukládá všechny komentáře k příspěvkům na webu, včetně potvrzených, čekajících na moderace, spamů, pings a trackbacks.



Obrázek 3.1: Schéma databáze WordPress [2]

Ping je upozornění jiného webu, že na původním webu je publikován článek, který odkazuje na ten samý web. Trackback je přijetí ping od jiného webu.

- Tabulka `wp_options` je zodpovědná za všechna globální nastavení WP.
- Tabulka `wp_terms` obsahuje informace o všech termínech (terms) – rubrikách (category) a štítkách (tag), které byly vytvořeny uživatelem, přičemž doplňující informace o rubrikách a štítkách je uložena do tabulky `wp_termmeta`.
- Tabulka `wp_term_taxonomy` je určena pro ukládání informace o taxonomie (taxonomy). Taxonomie v WP určuje princip umístění rubrik a štítků.
- Pro propojení příspěvků s rubrikou se používá tabulka

`wp_term_relationship`, která uskutečňuje standardní přístup pro realizace vztahu many-to-many v relačních databázích.

- Tabulka `wp_links` se obvykle používá k ukládání blogroll - seznamu odkazů na jiné stránky nebo blogy.

Důležité si všimnout, že v původním zdrojovém kódu neexistuje databáze. Databáze WP se generuje během instalace WP. Databáze musí vytvořit uživatel samostatně a při instalace propojit založenou WP stránku s databází nebo nejvíce hostings umožňují provést to automatické. Následně WP pouze vytvoří správné tabulky v té databázi a ve zdrojovém kódu se zobrazí odpovídající složky.

Model ER databáze WP je prezentován na oficiálním webu WP. Je třeba poznamenat, že tento model byl naposledy aktualizován pro verzi 4.4. Je také relevantní pro některé předchozí verze, ale není specifikováno, zda platí pro novější verze, včetně aktuální nejnovější verze 5.7.1. Byla provedeno ověření generováním při instalaci databáze WP verze 5.7.1 a žádné změny od verze 4.4 se netýkaly databáze, což znamená, že v tomto modelu nebylo potřeba provedení jakýchkoliv změn.

ER model databáze WP byl použit jako podklad pro vytvoření strukturálního modelu CMS WP prostřednictvím Class Diagram, v kterém byly vynechány některé entity obsahované v původním modelu. Vynechané entity nebo atributy entit nebyly považovány za realizující základní funkcionalitu WP.

Všechny entity, které obsahují metadata (`wp_postmeta`, `wp_termsmeta` a t.d), pouze upřesňují nebo rozšiřují informace o už existujících odpovídajících jim entitách a proto objekty těchto entit vždy jsou spojené s nějakým jiným objektem. Kvůli tomu, že entity s metadaty netvoří vlastní samostatné objekty, tento entity nejsou obsažené v Class Diagram.

Entita `wp_links` realizuje vypnutou funkcionalitu starší verze WP a aktuálně se využívá jenom v případě nutnosti pomocí pluginu, který ji znovu zapíná. Běžný uživatel samostatně není schopen žádným způsobem provést jakékoliv změny s objekty této entity.

Data, která jsou obsažena v entitě `wp_options`, ovlivňují fungování webů na WP jako celku, ale nejsou žádném způsobem spojené s její jednotlivými elementy.

Komentáře a možnost registrace uživatelů lze vypnout a proto entity `wp_comments` a `wp_users` lze považovat za nepovinné a vyloučit. Navíc přesto že v ER modelu jsou entity `wp_users` a `wp_comments` jsou propojené, ve skutečnosti WP neukládá data o takových vazbách a v Class Diagram by této třídy nebyly propojené s žádnou jinou třídou.

Hlavním účelem entity `wp_term_relationships` je umožnit aplikaci WP určit, jestli termín je rubrikou, odkazem nebo štítkem. Kvůli tomu, že v ER modelu vztah many-to-many musí být rozdělen na dva vztahy one-to-many, `wp_term-relationships` realizuje vztah many-to-many mezi `wp_terms` nebo

wp_links a wp_posts. V UML lze entitu wp_term-relationships nahradit vztahem asociace many-to-many.

Přesto že technicky tabulka wp_term_taxonomy s verzí 4.2 není nutná, protože logika ukládání dat se změnila a atributy třídy wp_term_taxonomy je možné umístit v tabulce wp_terms, je tato tabulka je použitá v poslední verze WP. Vyloučení této tabulky z databáze WP mohlo působit chyby provozu WP vzhledem k problému kompatibility WP s existujícími pluginy, widgety a tématy. Protože cílem je modelování reálného systému, tato entita byla použita v Class Diagram i přesto že šlo ji vynechat.

V důsledku Class Diagram obsahuje devět tříd celkem a tři entity jsou převzaté z původního ER modelu databáze WP s vynecháním zbytných atributů. Výsledný Class Diagram je ukázán na obrazcích 4.1, 4.4 a 4.7.

Atribut term_group entity wp_terms zařazuje termín do nějaké skupiny, co je umožněno pouze pomocí pluginu, a proto nespadá do základních funkcionalit WP.

Atribut term_id entity wp_term_taxonomy už není nutný kvůli zavedení vztahu agregace.

Atributy post_date_gmt a post_modified_gmt duplikují čas vytváření a poslední modifikace wp_posts, ale v GMT.

Atributy comment_status a comment_count jsou zbytečné, protože Class Diagram nemá třídu wp_comments. Post_content_filtered používá se pouze pluginy a jeho obsah se maže po každé aktulizaci verze WP.

Atributy ping_status, to_ping a pinged WP doplňuje automatické podle nastavení webu. Automatické upozornění jiných webů lze zapnout nebo zakázat ostatním webům posílat upozornění, a proto této atributy byly vynechány.

Atribut post_type určuje typ instalaci třídy wp_posts a byl nahrazen vztahy generalizace. Instalaci třídy wp_posts jsou spojené vztahy dědičnosti s instalacemi třídy post, page a attachment pojmenovaných podle typu, které atribut má v databáze WP. Pro typ revision byla vytvořena vlastní třída s cílem ukázat a následné realizace jeho vztahu s třídou wp_posts.

Třída page má vlastní atributy: menu_order obsahuje pořadové číslo zobrazení stránky v seznamu a post_parent se používá k určení nadřazenou stránky, když taková existuje.

Třída attachment obsahuje kromě atributů převzatých u wp_posts: post_mime_type, kde je uložen typ souboru attachment ('image/jpeg' nebo 'application/pdf') a post_parent, který se používá k ukládání informací o vazbě attachment s post nebo page.

Třída post nemá vlastní atributy, ale vždy má přidělenou category nebo tag, proto třída post je spojena vztahem agregace s třídou wp_terms. Když uživatel neudělí žádnou category nebo tag instalaci třídy post, WP automatické zařadí tento post do automatické vygenerovanou category „Bez rubriky“. Smazání category nesmaže posts v dané category. Jeden post může spadat do více categories nebo tags.

Každá instalace třídy `wp_posts` může mít `revision`. `Revision` se vytváří automaticky, protože WP zaznamuje každý uložený objekt třídy `wp_posts` nebo publikovanou aktualizaci. Entita `revision` má atributy: `revision_ID`, `revision_name`, `revision_status` a `revision_parent`, který obsahuje ID verzovánou instalaci třídy. Atribut `revision_status` je `read-only`, protože vždy nastaven WP během generování verze na `inherit` a není možné ho změnit. Vzhledem k tomu, že `revision` pouze verzuje nějakou instalaci třídy `wp_posts`, obsahuje taky atribut `revision_parent`, v kterém je uložena instalace třídy `wp_posts`, pro kterou nejsou povoleny změny.

V Class Diagram instalace třídy `wp_posts` a `revision` jsou spojené vztahem kompozice, protože instalace třídy `revision` nemůže existovat bez instalace třídy `wp_posts`.

Atribut `taxonomy` třídy `wp_term_taxonomy` byl nahrazen vztahy dědičnosti s instalacemi tříd `category` a `tag` pojmenovaných podle typů, které atribut může mít v databáze WP. Typ `link_category` byl vynechán, protože entita `wp_links` nebyla realizována v Class Diagram.

Atributy ID všech tříd jsou `read-only`, protože se vytváří WP automatické a uživatel má zakázáno provádět s nimi změny.

Všechny operace tříd jsou funkcemi definovanými v dokumentací, které realizuje základní funkčnost WP.

Třídy `wp_terms`, `wp_term_taxonomy` a `wp_posts` mají operace `get`, která žádá na vstupu ID nějaké instalace třídy a následně vrací objekt instalaci třídy, když ten existuje. Operace `get`, které vracejí konkrétní jednotlivé atributy jsou triviální a můžou být realizovány na základě `get_term`, `get_term_taxonomy` případně `get_post`, a proto nebyly uvedeny na diagramu. Kromě toho každá z těchto tříd obsahuje operace pro vytváření, upravování a odstranění určité instalaci třídy.

Operace `insert` má jako vstupní parametry všechny atributy jednotlivé třídy uložené do datového kontejneru vektor a vždy vrací ID nově vytvořené instalací třídy. U třídy `wp_term` je navíc taky vrácen `taxonomy_term_id`, který je uložen do pole spolu s `term_id`.

Operace `update` na vstupu vyžaduje ID instalace třídy, s kterou se provádí změna. Ostatní atributy, které nejsou uvedené ve vstupu, budou nastavené jako implicitní. Následné operace vrací ID aktualizované instalaci třídy.

Operace `delete` smaže instalace třídy a informace o všech vztazích, kterými byla spojena s jinými třídami. Vstupním parametrem je ID instalace třídy, výchozím parametrem je `true`, když se podařilo element smazat, nebo `false`, když se odstranění nepovedlo.

Třída `wp_terms` má navíc operace `wp_get_term_taxonomy_parent_id()`, která vrací ID nadřazeného prvku taxonomie k zadanému ID termínů.

Třída `wp_term_taxonomy` obsahuje operace `get_terms()`, která vrací vektor objektu termínu odpovídajících taxonomie, která je zadána podle ID.

Třída `wp_posts` má ještě dvě operace. Operace `wp_push_post` publikuje instalace třídy `wp_posts` a nastavuje hodnotu atributu `post_status` na `publish`.

Operace wp_trash_post přesouvá instalace třídy `wp_posts` do koše a nastavuje hodnotu atributu `post_status` na `draft`. Vstupním parametrem je ID objektu `wp_posts`. Této operace nic nevracejí.

Třída `post` má tři operace. Operace `wp_set_post_terms()` nastavuje termíny pro zadaný příspěvek. Seznam termínů je uložen do vektoru. Vstupními parametry jsou ID příspěvku, vektor nových termínů, název taxonomie termínů, ke kterou se připojuje příspěvek, a logická proměnná, která určuje, jestli přidat nové rubriky k už existujícím (`true`) nebo nahradit termíny (`false`). Operace vrací vektor, který obsahuje ID přidaných termínů. Operace `wp_get_post_tags()` a `wp_get_post_categories()` vyžadují na vstupu ID příspěvků a následně vrací pro něho všechny ID štítku případně rubrik ve formě vektoru.

Třída `revision` má operace `wp_get_post_ revision`, která vrací vektor všech redakce uvedeného příspěvku. Pomocí operací `get_revision` podle ID redakce lze získat objekt této třídy. Operace `wp_delete_post_revision` smaže určitou verze podle ID instalaci třídy `revision` nebo všechny `revision` spojené
s instalací třídy `wp_posts`, když na vstupu je objekt třídy `wp_posts`.

Je třeba poznamenat, že třídy `wp_posts` a `wp_term_taxonomy` byly označené za abstraktní s cílem zakázat generování instalaci tříd v některých jazycích programování, které to umožňují. Takové její operace jako `get`, `insert`, `update` a `delete` byly poznamenány jako abstraktní taky.

3.3 Procesní modely

3.3.1 State Machine Diagram Page

State Machine Diagram popisuje process změny stavů v důsledku vnějšího vlivu a je vhodný pro popis základních činností běžného uživatele, které určitým způsobem ovlivňují kontent webů na WP. Jak bylo vymezeno, základní elementy, ze kterých se skládá web v redakčním systému WP jsou page, post a attachment. Page a post mají stejné stavy, ale kvůli tomu, že web musí mít alespoň hlavní stránku, ale může nemít žádný post, byl vytvořen State Machine Diagram Page. State Machine Diagram Post je zbytečný, protože bude mít ne jen stejné stavy, ale i přechody. Attachment může být pouze vytvořen nebo zmazán a proto State Machine Diagram Attachment bude mít dva stavy, což je triviální.

První stav, který vždy má jakákoliv stránka na WP je `Added`. Po vytvoření stránky uživatel může ihned publikovat stránku, uložit jako koncept, nastavit čas automatické publikace stránky nebo nechat čekat publikace stránky na schválení od jiných uživatelů, kteří mají povoleno spravovat stránky. Tím stránka přejde do odpovídajícího stavu `Published`, `Draft Scheduled` nebo `Pending review`. Po provedeni jakékoliv z těchto činností, kromě vytvoření stránky, lze stránku odstranit. Po odstranění stránka je přesunuta do koše

a je ve stavu `Draft`. Z koše ji lze obnovit nebo smazat navždy a tím působit přechod ze stavu `Deleted permanently` do finálního stavu.

Skoro všechny přechody diagramu jsou oboustranné, co znamená, že existuje činnost, která působí přechod ze stavu A do B, a činnost, která působí přechod ze stavu B do A. Nelze vrátit stránku do stavu `Added`, po publikace nechat čekat na schválení a po odstranění stránky působit jakýkoliv jiný přechod, kromě do stavu `Draft`. Po stavu `Deleted permanently` stránka už neexistuje a není možné s ní provést žádnou činnost.

Stojí za zmínku, že v databáze page může mít stav `auto-draft`, protože WP automatické ukládá page během její redakce uživatelem každé třicet vteřin. Taková kopie existuje jenom jedna a nahrazuje sama sebe. V diagramu takový stav není, protože k němu není možné udělat žádný přechod z důvodu neexistence události, která působí takový přechod.

Stránka má tři stavy viditelnosti: veřejná, soukromá a chráněna heslem, ale této stavy byly považovány za atributy třídy `page`, protože jsou nastavitelné jakkoliv během doby existence stránky.

Výsledný State Machine Diagram již namodelovaný v konkretních metodách pro MDD je ukázán na obrazcích 4.2, 4.5 a 4.8.

3.3.2 Proces vykreslení WordPress stránky

Proces načítání jednotlivé stránky je jeden z nejdůležitějších a nejčastěji prováděných v WP.

Na webové stránce založené jedním ze zakladatelů WP je článek [41], kde je postupně popsán celý proces načítání jednotlivé stránky a spouštění souborů WP. Popis je realizován pomocí seznamu a schématu. Kvůli tomu, že seznam obsahuje třicet bodů, takový popis je velmi obsáhlý a může zmást začínajícího uživatele WP. Schéma není nijak strukturováno, obsahuje velké množství textu v seznamech. Prakticky tvoří jen stylizovaný zanořený seznam, proto není dobře pochopitelný.

Activity Diagram procesu vykreslení WP stránky je modelován s cílem strukturovat a kompaktně představit tento rozsáhlý proces založený na spouštění různých souborů WP. Výsledný Activity Diagram již namodelovaný v konkretních metodách je ukázán na obrazcích 4.3, 4.6 a 4.9.

Celý proces začíná tím, že uživatel zadává adresu webové stránky WP, kterou chce načíst do adresního řádku prohlížeče. Následně prohlížeč po získání IP-adresy webu se obrací pomocí této adresy na hostitelský serveru a žádá o stažení webu.

Jako odpověď na žádost začíná načtení a příprava nastavení webu WP. Nejprve dochází ke stažení hlavního konfiguračního souboru `wp-config.php` v kořenovém adresáři WP. Odtud jsou extrahované globální proměnné, hodnoty standardních konstant pro WP stránky a informace pro připojení k databázi. V případě, že na webu již existuje soubor obsahující informace o me-

zipaměti (cache) stránek webu nebo takový generuje plugin, je stažen soubor `advanced-cache.php`.

Pomocí informace o databázi získanou ze souboru `wp-config.php` WP se snaží připojit k určité databázi. Když se to nepovede, tak posílá signál ze nastala chyba, aby prohlížeč ukázal zprávu "Error establishing database connection".

Po této chybě proces se ukončuje a je potřeba upravit problém konfigurace WP s databází před novým pokusem načítání WP stránky.

Když takový problém není, CMS WP vybere odpovídající databáze na serveru MySQL a pokračuje načítáním souborů `object-cache.php`, kde jsou uloženy všechna data, které byly získány mnohokrát v procesu generování kódu, nebo data získaná v důsledku provedení složitých operací.

Jestli stránka je multiwebem, spouští se režim Multisite, který stahuje soubor `wp-content/sunrise.php` a pluginy multiwebu. Multiweb (multisite) je pracovní režim WP, který umožňuje používat existující soubory jádra a existující databázi pro vytváření z webových stránek WP společně sítí. V multiwebu každá stránka má své vlastní nastavení, ale témata, pluginy a uživatelé jsou sdíleny.

Následně se provádí načítání všech ostatních pluginu, přičemž se první stahují povinně pluginy. Kromě pluginu v této fáze je stažen soubor `pluggable.php`, v kterém jsou uložené funkce, které byly přepsané pomocí pluginů WP. Potom jsou načteny Rewrite Rules – pravidla pro přepsání odkazů URL do struktury pravidel WP.

Spuštění funkce `setup_theme` připravuje stránku WP k procesu načtení šablony stránky, který začíná stažením `functions.php` souboru nadřízené šablony. Soubor `functions.php` obsahuje sadu funkcí a instrukcí pro určitou šablonu. Po stažení `functions.php` z podřízeného tématu bude načten `functions.php` hlavní rodičovské šablony. V případě že se nadřízená šablona nepoužívá, je stažen `functions.php` hlavní aktivní šablony.

Po inicializaci aktivní šablony a nahrávání její do souboru `functions.php` dochází k provedení funkce `after_setup_theme`, která nastavuje základní funkcionalita šablony. Pak WP nahrává objekt aktuálního uživatele a určuje jeho role pro tento web. Jakmile WP je kompletně nahrán, ale předtím, než všechny hlavičky soubory jsou odeslány, funkce `init_widget` registruje widgety a provádí kód potřebný pro jejich provoz.

Následně WP volá funkce `wp()` ze souboru `wp-includes/functions.php`, která určuje hlavní požadavek do databáze – prostředí WP. V této fáze se načítají filtry, stránky, příspěvky a ostatní elementy z databáze, a v důsledku všechny objekty jsou určené, ale nejsou poslané do prohlížeče.

Jako poslední se načítají témata po spuštění funkce `template_redirect`, která umožňuje přesměrovat vytvořené objekty do témata. Potom WP začíná stahovat soubory aktivního tématu podle jeho hierarchie.

Po spuštění funkce `shutdown` je proces vykreslení stránky WP ukončen a prohlížeč ji zobrazuje uživateli.

Kapitola 4

Vývoj pomocí MDD

Vzhledem k tomu, že cílem bakalářské práce je použít přístupy MDD pro vygenerování zdrojových kódů aplikace z vytvořeného modelu, hlavním kritériem při výběru CASE-nástroje byla možnost generování zdrojových kódy. Kromě toho v předchozí kapitole byly vymodelovány tři diagramy: Class Diagram, State Machnine Diagram a Activity Diagram, z čeho vyplývá, že zvolené metody musí umožňovat modelovat této diagramy a generovat kód z těchto diagramů.

Zvolení podobných nástrojů nebude mít přínosné výsledný srovnání, protože, jak se předpokládalo, nebudou lišit, vzhledem k tomu, že fungují na stejných principech a vlastně jsou složeny z nějakého množství modulů, každý z nich splňuje určitou funkcionalitu. Pro vhodné srovnání bylo rozhodnuto zvolit komerční nástroj, nástroj poskytující licenci GPL a online nástroj bez nutnosti instalace.

Kvůli tomu, že komerční nástroj je obecně používán v praxi pro modelování architektury rozsáhlých systémů a aplikací, předpokládá se, že má být složitější na použití, ale mít více funkcionalit včetně těch, které usnadňují vývoj. Nástroj s licencí GPL poskytuje možnost vývojáři přidat svoje vlastní funkcionality a ušetřit na nákladech za nástroj. Online nástroje se pouze začaly rozvíjet a běžná taková vývojová metoda umožňuje jenom modelování a základní funkce. Byly zvoleny tři přístupy pro vývoj v vývoje řízeného modelem: Enterprise Architect (EA), Umbrello a GenMyModel.

EA byl zvolen jako příklad CASE-nástroje, který patří do kategorie komerčních řešení s širokou škálou funkcí pro komplexní modelování rozsáhlé architektury, umožňuje generování kódu velkého množství jazyků.

Umbrello byl zvolen jako příklad CASE-nástroje, který poskytuje licence GPL a zároveň má generátor zdrojového kódu.

Nebyl nalezen žádný jiný CASE-nastroj, kromě GenMyModel, který lze používat online bez nutnosti instalace a který zároveň umožňuje generování zdrojového kódu z modelu.

Očekávalo se, že v případě, jestli CASE-nástroj to bude umožňovat, všechny

potřebné diagramy budou vygenerovány z původního. Proto se alespoň jedna metoda musela použít k modelování původního diagramu, z kterého se vygeneruje zdrojový kód v ostatních metodách.

Nástroj GenMyModel je popisován jako jednoduchý a intuitivní na ovládání a vzhledem k tomu, že autorka bakalářské práci nikdy nepracovala s takovým typem nástrojů, bylo rozhodnuto udělat původní diagramy pomocí tohoto přístupu
s cílem ohodnotit, jestli lze bez potíží modelovat diagramy v něm bez žádné předchozí zkušenosti a případně vymezit, jaké problémy nastávají při prvním použití nástroje hned v praxi. Funkcionalita GenMyModel umožňuje exportovat diagramy ve formátu XMI souboru, který je typickým formátem výměny UML modelu, z čeho vyplývá, že tento CASE nástroj odpovídá požadavkům.

4.1 GenMyModel

4.1.1 Popis

GenMyModel [42] je bezplatný online UML nástroj pro vývojáře a softwarové architekty, který umožňuje generovat kód z modelu. Na rozdíl od známých alternativ není závislý na webovém prohlížeči nebo operačním systému (Windows, Linux, Mac OS). GenMyModel má tří druhů edice: Solo, Team, Enterprise.

- Edice Solo je zdarma a nabízí modelování v UML EMF, RDS, uložení modelů a verzování.
- Edice Team rozšiřuje možnosti tarifů Solo pro práci v týmu a podporuje modelování jednoho modelu s použitím několika jazyků. Cena takové edice 25 euro měsíčně nebo 275 euro ročně pro jednoho uživatele.
- Edice Enterprise umožňuje přidávat read-only uživatele a vytvářet hlášení a je určena pro zlepšení komunikace mezi zákazníkem a dodavatelem systému.

Byla zvolená edice Solo, protože je zadarmo a nabízí všechny funkcionality, které jsou potřebné v rámci provedení analýzy.

4.1.2 Modelování

Kvůli tomu, že GenMyModel je umožněno používat pouze online, prvním krokem byla registrace na oficiálním webu pomocí GitHub, Google Account nebo emailové adresy. Druhým krokem bylo zvolit edici, která se bude používat při vývoji.

Po zvolení tarifu byl založen projekt, ve kterém byly vytvořeny ručně Class Diagram, State Machine Diagram a Activity diagram, jak je ukázáno na obrazcích 4.1, 4.2 a 4.3.

Elementy, ze kterých se skládají diagramy jsou zobrazeny graficky i slovně ve vertikálním panelu nástrojů, což pomáhá zrychlené vyhledat nutný element z nabízených. Po výběru a umístění prvků na diagramu lze přidat jeho vlastnosti.

Navíc při modelování určitého diagramu jsou ukázány pouze ty elementy a vztahy, které můžou být použity v tomto diagramu a ostatní jsou skryty, což zmenšuje počet takových chyb, jako je například použití nevhodného elementů nebo vztahů. Je umožněna jednoduchá navigace po elementům systému pomocí levého menu, kde jsou vypsány všechny diagramy, datové typy a třídy, které byly použité v projektu. Pro modelování stačilo intuitivní ovládání GenMyModel a nenastala nutnost číst tutoriál GenMyModel.

4.1.3 Generování kódu

Tlačítko generování kódu je umístěno do levého vertikálního menu a není potřeba ho hledat. Po otevření okénka generování kódu je nabízeno zvolit buď vlastní generátor nebo jeden z "Sample Generator" pro Database, Ecore, JPA, UML a Utils. Pro UML lze generovat zdrojový kód jazyků C#, C++, HTML, Java, Lua, PHP, Python, Ruby, SQL nebo TypeScript. Pro jazyk C++, který byl zvolen, lze vygenerovat .cpp a `main` soubory nebo `Makefile`.

V případě generování .cpp souboru se otvírá okno s kódem, ve kterém se používají šablony GenMyModel, které generují zdrojový kód zvláštních elementů diagramu a jejich vlastnosti. Lze ihned spustit tento kód nebo udělat uživatelské změny, jestli jsou takové nutné pro vyloučení nějakých chyb.

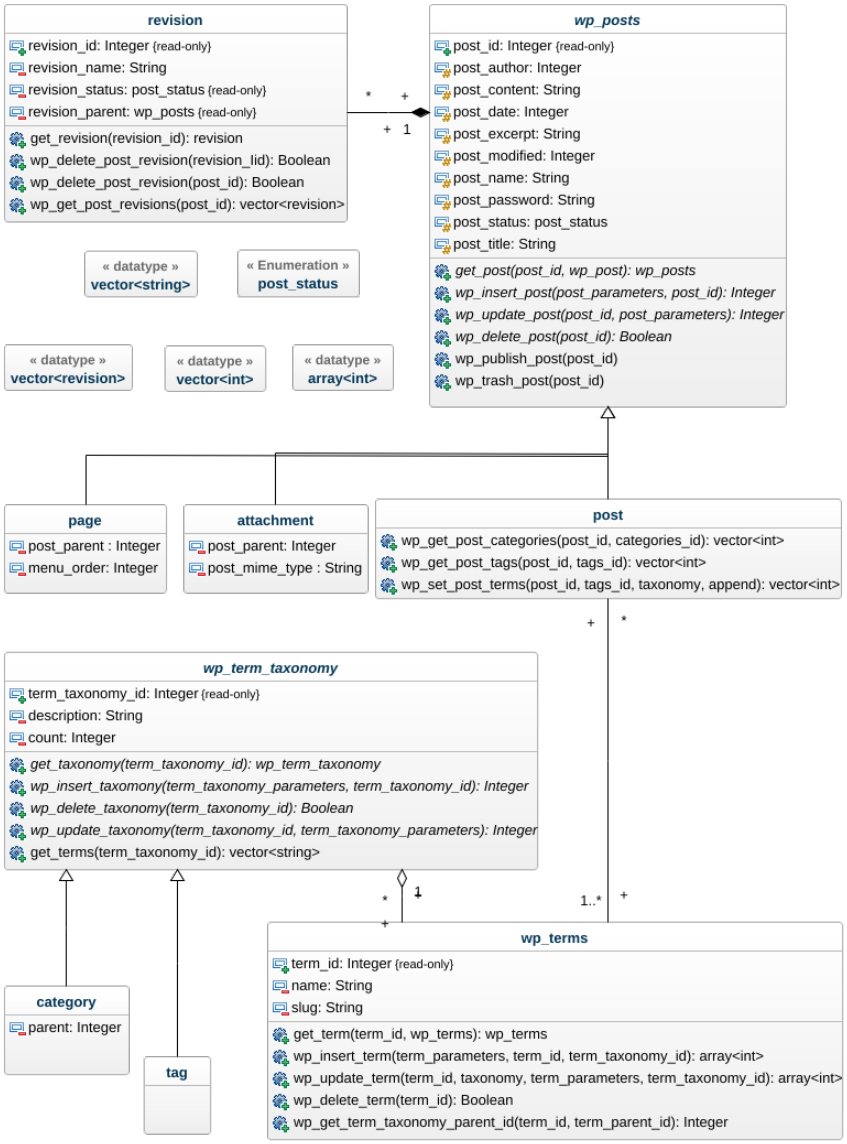
Výsledkem generování je sada pár souborů .cpp a .hpp každého diagramu a každé třídy Class Diagram. Lze nahrát výsledek generování na GitLab nebo jednoduše stáhnout všechny soubory archivované do ZIP souboru. Pomocí "Prewiew" je umožněno podívat se online na kód, které obsahují jednotlivé vygenerované soubory. Ale bohužel není vhodná navigace mezi soubory. Ve vodorovném menu je umístěno tolik souborů, kolik se vedle podle velikosti okénka. Pro prohlédnutí dalšího souboru je potřeba zavřít nějaký soubor už otevřeny soubor.

4.1.4 Výsledný kód a dokumentace

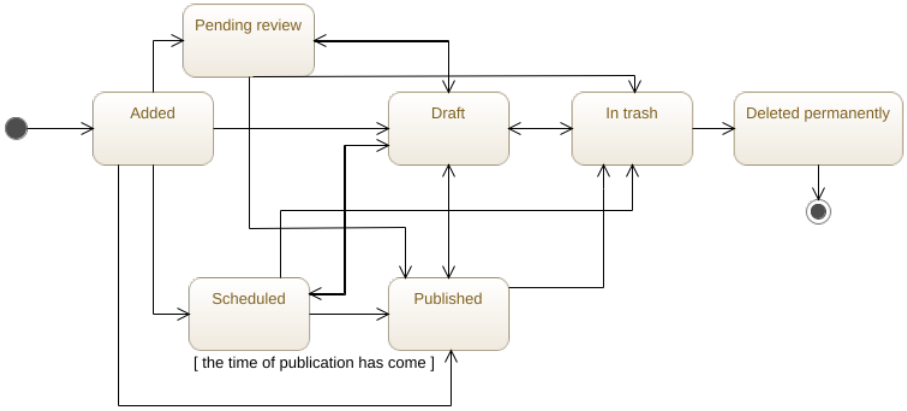
Všechny soubory .cpp měly pouze jednu řádku:

```
1
2 /* Generated from GenMyModel */
3
4 #include "wp_posts.hpp"
```

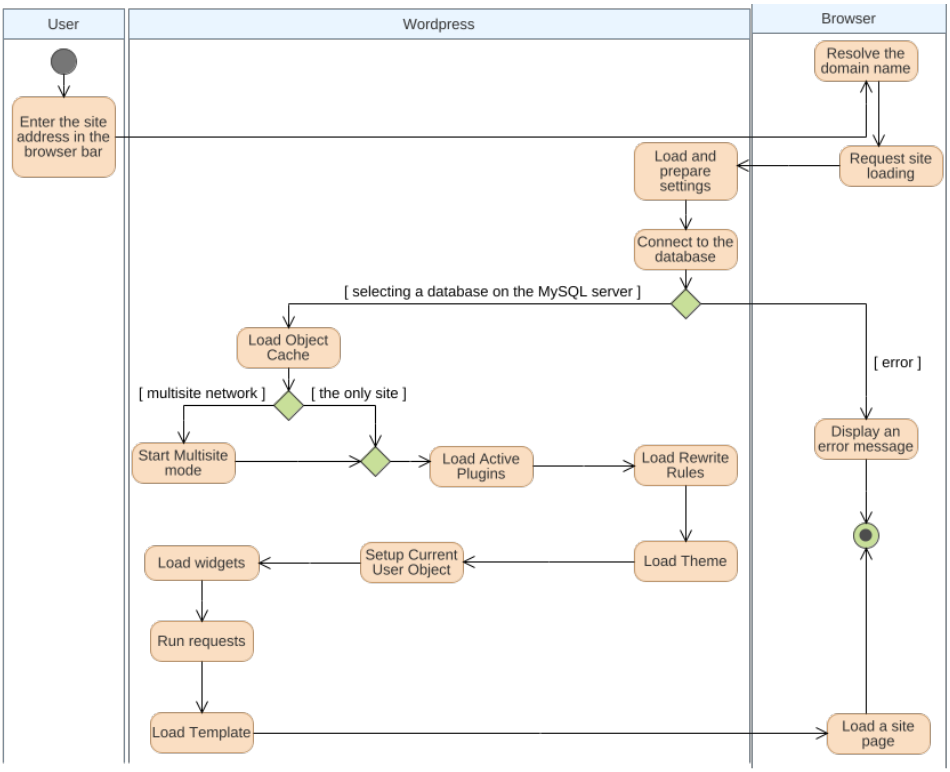
Hlavičkové soubory .hpp diagramů obsahovaly pouze deklarace tříd pojmenovaných podle názvu diagramu. Soubory .hpp tříd obsahovaly název třídy a atributy s příslušnými datovými typy, jak je vidět v kódu níže.



Obrázek 4.1: Class Diagram GenMyModel



Obrázek 4.2: State Machine Diagram GenMyModel



Obrázek 4.3: Activity Diagram GenMyModel

```
1
2  /* Generated from GenMyModel */
3
4  #ifndef DEF_WP_POSTS
5  #define DEF_WP_POSTS
6
7  #include <iostream>
8
9  class wp_posts
10 {
11   public :
12     int post_id;
13
14   protected :
15     int post_author;
16     std::string post_content;
17     int post_date;
18     std::string posr_excerpt;
19     int post_modified;
20     std::string post_name;
21     std::string post_password;
22     post_status post_status;
23     std::string post_title;
24
25
26 };
27
28
29 #endif
```

Vygenerovaný zdrojový kód měl výrazné vady a poztrácel část informace zachycenou v modelu. Všechny vztahy mezi třídy kromě dědičnosti nebyly vhodné realizované v vygenerovaném zdrojovém kódu.

Nebylo realizováno omezení vztahů kompozice mezi třídami `wp_posts` a `revision`, který určuje, že instalace třídy `revision` nemůže existovat bez instalací třídy `wp_posts` a vždy musí být nějakým způsobem propojen s právě jednou. Očekávalo se, že generátor vytvoří nějaké atribut třídy `wp_post`, které bude použit jako datový kontejner pro objekty `revision` a operace `insert` bude umožňovat přidání objektu `revision` pouze do tohoto kontejnerů, čím zakáže generování objektu `revision` bez spojení s objektem `wp_posts`.

Nebyl uskutečněn vztah asociace tříd `wp_terms` a post. Vztah many-to-many mezi timito třídami bylo vhodné realizovat přidáním do atributů každé z tříd datového kontejneru, kde jsou uloženy ukazatele na objekty, s kterými je propojen původní objekt.

Následně vztah agregace mezi třídami `wp_terms` a `wp_term_taxonomy` nebyl znázorněn v zdrojovém kódu. Byla potřeba přidat atribut do třídy `wp_terms`, který odkazuje na taxonomie pod kterou spadá nebo obsahuje ID taxonomie, a atribut do třídy wp_term_taxonomy, který je schopen odkázat na seznam všech termínů taxonomie.

Vytvořený a použity v Class Diagram Enumerator `post_status` nebyl žádným způsobem generován do zdrojového kódu jako například zvláštní třída. Místo toho bylo ponecháno pouze jeho jméno a není nějak označeno že je Enumerator. Kromě toho stejná situace nastala s uživatelem vytvořenými elementy DataType.

Přičemž ze v modelu jsou definovány operace tříd včetně vstupních, výstupních parametrů a datového návratové hodnoty, v zdrojovém kódu třídy nemají operace.

Takže atributy třídy ztratily informace o implicitní hodnotě a vlastnostech určené v diagramu: `read-only`, `unique`, `ID` a další, které nebyly použité při modelování.

Soubor `main` neobsahoval žádný kód přidaný generátorem. Soubor `Makefile` automatizoval spuštění dohromady všech ostatních souboru a byl kompletní.

Žádné změny v modelu nepomůžou nějakým způsobem zdokonalit nebo rozšířit výsledný kód. Šablony, realizované v GenMyModel pro jazyk C++ umí generovat správně pouze třídy a atributy. Pro zvětšení počtu elementů a vztahu modelů které budou vhodné generovaný do zdrojového kódu je potřeba napsání vlastních šablon, které to budou umožňovat.

GenMyModel umožňuje generování dokumentace ve formátech Word a PDF. Dokumentace vytvořeného modelu obsahovala očíslovaný seznam všech elementů modelu a poznámku v jakém diagramu se nachází každý element. Třídy navíc měly typické obdélníky rozdělené na sekce „Attributes“, „Operations“ a „Associations“.

Jednotlivý atribut třídy měl název, datový typ a jeho násobnost (multiplicity). Podobně jako v zdrojovém kódy nebyly upřesněny implicitní hodnota nebo vlastnosti atributu. Operace měla definované název, datový typ návratové hodnoty, vstupní a výstupní parametry včetně datových typů, ale bez určení toho jaké parametry jsou vstupní a jaké jsou výstupní. Vztah obsahoval odkaz na třídu v dokumentaci, s kterou je původní třída spojena.

4.1.5 Dokončení aplikace

Bez ohledu na realizace grafické strany aplikace modelování a následná generování zdrojového kódu skoro žádným způsobem nezrychlily vývoj systému, protože metoda GenMyModel není schopna udělat nic kromě generování souboru a uložení do nich jednotlivých tříd s atributy a generování dokumentace která obsahuje pouze základní data, přičemž pouze pro Class Diagram. Výsledná aplikace nebyla funkční a je potřeba její naprogramovat celou včetně ručního přepsání z modelu nebo kopírování z vygenerované dokumentace operace tříd s provedením oprav.

4.1.6 Identifikované problémy

- Při exportu souboru .xmi, tento soubor je prázdny.
- Není možné udělat popis přechodů v State Machine Diagram a Activity Diagram

4.2 Umbrello

Vzhledem k tomu, že každý model v Umbrello je uložen ve formátu .xmi, přesto že tato metoda neumožňuje import souborů .xmi, šlo přímo otevřít takový soubor, exportovaný z GenMyModel. Ale následně bylo zjištěno, že bohužel model je prázdný, což znamená, že metoda GenMyModel nemá korektně realizovanou funkcionalitu importu. Stojí za zmínku, že GenMyModel umožňovalo dva druhu importu: import as XMI a import as XMI with Diagrams, ale výsledkem otevření v Umbrello každého z vygenerovaných souborů byl prázdný model.

Kvůli tomu, že Umbrello nabízí import kódů, jako základ pro model byl využit zdrojový kód, vygenerovaný z modelu GenMyModel, se kterým byly následně provedené nezbytně nutné opravy. Vzhledem k tomu, že daný kód obsahuje pouze třídy a její atributy, takový postup ušetří ruční práce, ale nebude mít výrazný dopad na následné hodnocení metody.

4.2.1 Popis metody

Umbrello UML Modeller je program pro modelování diagramů UML založený na technologii KDE. Tato aplikace je svobodný software, a proto lze jí rozšířit vlastními funkcionality a opravit kód podle svých požadavků. I když Umbrello je navržen tak, aby vytvářel UML diagramy na platformě Unix, je možné aplikaci instalovat na Windows a Mac OS X.

Umbrello UML podporuje všechny standardní typy UML diagramů. Tento CASE-nástroj umožňuje import zdrojového kódu C++, IDL, Pascal, Delphi, Ada, Python, Java, Perl a dalších programovacích jazyků a umí vygenerovat zdrojový kód různých jazyků z modelu.

Při ukládání diagramu je použit formát souborů XMI. Následně Umbrello umožňuje exportovat data modelu ve formátech DocBook a XHTML, a také jako obrázek [43].

4.2.2 Modelování

Prvním krokem vývoje byla instalace poslední verze Umbrello 2.32 na vlastní počítač bez nutnosti registrace nebo dalších činností.

Umbrello má standardní uživatelské rozhraní, kde nahoře ve vodorovném menu jsou umístěné ikonky elementů odpovídající zvolenému pro modelování

diagramů, zleva se nachází navigace, která obsahuje všechny prvky použité v modelu.

Některé elementy kódu nebyly vhodně importované do modelu. Vector a array byly generovány jako Class, proto této třídy byly ručně nahrazeny elementy DataType. Třídy `tag` a `post` byly identifikované jako Interface, přičemž měly byl Class. Do názvů datových typů kontejnerů byly přidané zbytečné mezery, což mohlo působit následnou nekorektní generování kódu, proto existující prvky DataType byly odstraněné a vytvořené nové se správnými názvy.

Přestože Umbrello vyžaduje určení, jaký jazyk je použit pro import, nebylo rozeznáno značení prostoru jmen C++ `std::` před `string`. Řádka `std::string` se stala celým názvem datového typu a pro `std::` byla vytvořena vlastní třída. Každý datový typ `std::string` byl opraven na `string` a třída `std::` byla odstraněna.

Atribut třídy `wp_posts post_status` se objevil jako Class, který spojen vztahy kompozice s touto třídou, co se očekávalo, protože v kódu nebyl poznamenaný jako Enum. Následně Class `post_status` byl nahrazen Enum `post_status`.

Zmizely vztahy mezi `wp_terms` a `post`, `wp_terms` a `wp_term_taxonomy`, ale všechny vztahy dědičnosti a abstraktnost tříd byly zachovávané.

Potom byla doplněna chybějící v modelu informace, která nemohla být vygenerována ze zdrojového kódu z důvodu neexistenci - vlastnosti jednotlivých operace (`virtual`, `const`, `abstract` a další). Navíc byly přidané nově použité datové typy. Z důvodu zlepšení přehlednosti Class Diagram na obrázku 4.4 triviální elementy DataType byly skryty.

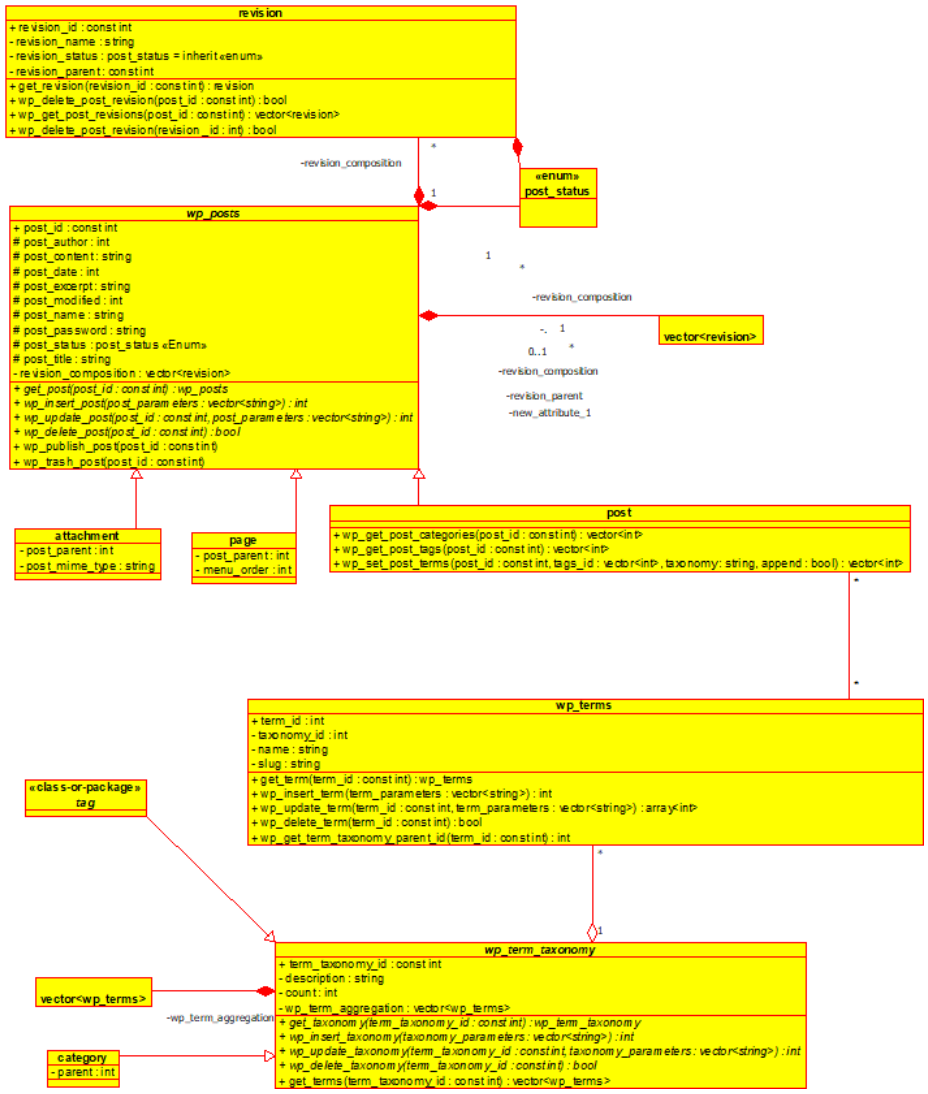
V případě přidání vztahu mezi třídy, Umbrello se snaží realizovat tuto vazbu přidáním nového atributů do jedné z tříd. Vztah kompozice mezi třídami `wp_posts` a `revision` byl uskutečněn novým atributem třídy `wp_posts revision_composition` s datovým typem `vector<revision>`. Přidání vztahu agregace mezi `wp_terms` a `wp_term_taxonomy` působilo vytváření atributů `wp_term_aggregation` s datovým typem `vector<wp_terms>`. Je potřeba upřesnit, že názvy atributů a datové typy byly zvolené ručně, nikoliv nabízené Umbrello.

Vzhledem k tomu, že zdrojový kód vygenerovaný pomocí metody GenMyModel neobsahoval žádné informace o State Machine Diagram nebo Activity Diagram, diagramy State Machine Diagram a Activity Diagram byly vymodelované od začátku v Umbrello, jak je ukázáno na obrazcích 4.5 a 4.6.

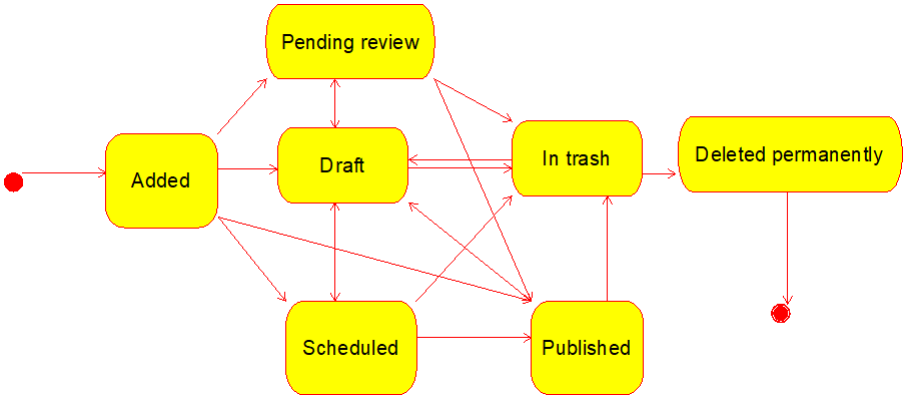
4.2.3 Generování kódu

Tlačítko „Code“ se nachází v hlavním menu, které obsahuje v rozevíracím seznamu buňku „Code Generation“.

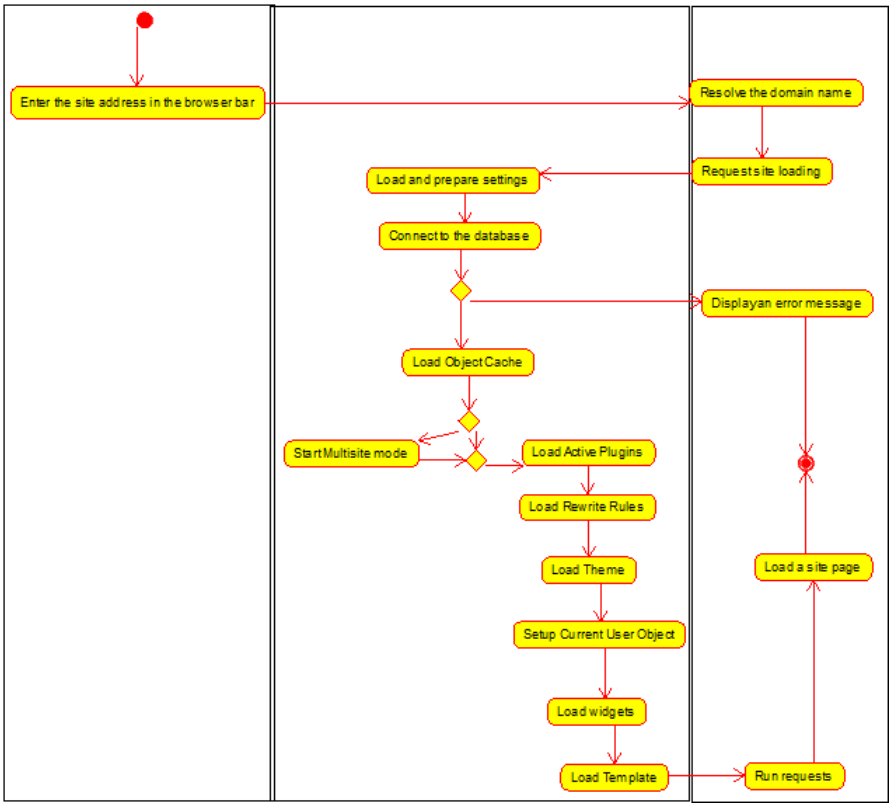
Dalším krokem uživatele je změnit parametry používané generátorem kódu při generování kódu. V otevřeném okénku je potřeba nastavit obecné nastavení, nastavení formátu a nastavení jazyků, v kterém bude generován kód.



Obrázek 4.4: Class Diagram Umbrello



Obrázek 4.5: State Machine Diagram Page Umbrello



Obrázek 4.6: Activity Diagram Umbrello

Obecné nastavení vyžaduje uvést jazyk, pro který kód bude generován, složku, kam budou uložené soubory, a jestli nahradit, přejmenovat nebo zeptat se uživatele v případě nalezení duplicity jmen vytvořených souborů s už existujícími soubory v složce.

Nastavení formátu jsou zodpovědné za to, jak bude vypadat dokumentace. Nutné určit, jestli bude generovaná dokumentace pro elementy nebo celou sekce, kde uživatel neuvedl žádný komentář, rozsah mezi řádkami, velikost písmen a styl komentářů, který se liší v závislosti na systému (Windows, *NIX, Mac).

Nastavení jazyků umožňuje zvolit, jak bude poznamenán komentář (Slash-Slash nebo Slash-Star) a upřesnit, jestli budou navíc generované: virtuální destruktory, prázdné konstruktory, funkce `get` pro každou třídu a další možnosti. Taky lze určit, jak budou vypadat třídy ze standardního prostoru jmen C++ string a vector v vygenerovaném kódu.

Po potvrzení nastavení je zobrazen seznam tříd, ze kterých bude generován zdrojový kód, kde automatické jsou vybrány všechny třídy modelů, ale uživatel může nějaké třídy odstranit. Jakmile bude spuštěno generování kódu, vedle nazvu třídy se objeví zelený kroužek, když to dopadlo bez chyb, nebo červený, když nastala nějaká chyba, kterou Umbrello nahlásí a po její opravě lze generovat kód pro tuto třídu znovu.

Pro vygenerování maximálně možné kompletní aplikace v jazyce C++ byly zvolené všechny parametry zodpovědné za generování a povoleno generování prázdné dokumentace. Kromě toho, generování kódu bylo provedeno z každé třídy. Pro účely analýzy přidání komentářů k jednotlivým třídám, atributům, operací a dalším elementů modelu bylo považováno za zbytečné, protože neovlivní ohodnocení metody. Ve výsledku generování zdrojového kódu z modelu proběhla bez chyb.

4.2.4 Výsledný kód a dokumentace

Generování kódu Class Diagram proběhla korektně s výjimkami:

- Přesto že Umbrello se snažilo propojit třídy v případě vytvoření vztahu mezi nimi pomocí přidání atributu, v kódu soubory nejsou nějak propojené. Třídy které používají objekty jiné třídy, nevědí o její existenci, což znamená, že chybí řádka kódu:

```
1 #include   nazev  . h
```

- Enum `post_status` byl generovaný jako Class, přičemž má být Enum Class.

- Funkce přidání virtuálního destruktoru byla realizována pro všechny třídy, ale očekávalo se, že bude použita pouze pro třídy propojené vztahy dědičnosti.

- Třídy `post`, `attachment`, `page`, `tag` a `category` byly virtuální, ale nemají být.

Ostatní transformace elementů Class Diagram byla správná, jak je vidět v kódu níže. Pro State Machine Diagram byla vygenerována prázdná třída, pro Activity Diagram nebylo vygenerováno nic.

Kompletnost dokumentace záleží na vývojáře, ale Umbrello vytváří vhodnou šablonu, proto lze považovat, že ze strany metody dokumentace byla vygenerována kompletně.

```
1
2  #ifndef WP_POSTS_H
3  #define WP_POSTS_H
4
5  #include <string>
6  #include <vector>
7
8
9  class wp_posts
10 {
11 public:
12  // Public attributes
13  //
14
15  const int post_id;
16
17
18  /**
19   * @return wp_posts
20   * @param  post_id
21   */
22  virtual wp_posts get_post(const int post_id) const = 0;
23
24
25  /**
26   * @return int
27   * @param  post_parameters
28   */
29  virtual int wp_insert_post(vector<string> post_parameters) = 0;
30
31
32  /**
33   * @return int
34   * @param  post_id
35   * @param  post_parameters
36   */
37  virtual int wp_update_post(const int post_id, vector<string>
      post_parameters) = 0;
38
39
40  /**
41   * @return bool
42   * @param  post_id
```

```
43   */
44  virtual bool wp_delete_post(const int post_id) = 0;
45
46
47  /**
48   * @param  post_id
49   */
50  void wp_publish_post(const int post_id);
51
52
53  /**
54   * @param  post_id
55   */
56  void wp_trash_post(const int post_id);
57
58 protected:
59  // Protected attributes
60  //
61
62  int post_author;
63  std::string post_content;
64  int post_date;
65  std::string post_excerpt;
66  int post_modified;
67  std::string post_name;
68  std::string post_password;
69  post_status post_status;
70  std::string post_title;
71
72 private:
73  // Private attributes
74  //
75
76  vector<revision> revision_composition;
77
78 };
79
80 #endif // WP_POSTS_H
```

4.2.5 Dokončení aplikace

Aplikace vygenerována z modelu Umbrello je kompletní na deset procent, protože poskytuje vytváření, změnu a další základní funkce pro elementy webů, které jsou schopné spolupracovat. Ale vzhledem k tomu, že v vygenerovaném kódu je potřeba provést opravy a dva ze tří diagramů jsou v ní vynechané, to není postačující pro fungování aplikaci v praxi.

4.2.6 Identifikované problémy

- Výrazným nedostatkem Umbrello je nefunkčnost automatického ukládání modelů spolu s častým nečekaným zavřením programů bez následné

možnosti ručního ukládání modelů po vyskytnutí chyby. Ve výsledku se doba modelování Umbrello zvětšila o dvakrát.

- Přechody State Machine Diagram nemají Trigger[Guard]/Effect, je možnost pouze doplnit obecný popis přechodu nebo dokumentace.
- Přechody Activity Diagram neumožňují přidat podmínku přechody do hranatých závorek po rozhodovacímu uzlu
- Nejsou pomocné linie, které pomáhají vyrovnávat přidaný element diagramu podle už existující elementů
- Nekorektní funkcionalita tlačítka „Undo“, protože přeskočí některé později uskutečněné změny a zruší dřívější.
- Globální nastavení generování kódu nefunguji a uživatel musí ji nastavovat pro každé generování kódu.

4.3 Enterprise Architect

S cílem udělat vstupy do metod co nejpodobnější, byl udělán pokus importu souboru .xmi exportovaného z metody GenMyModel. V EA se objevil balíček s jménem souboru .xmi, ale nefunkční. Nepodařilo se ho otevřít žádným způsobem, proto se muselo použít jiný základ pro modelování v EA.

Zdrojový kód, vygenerovaný z modelu GenMyModel, neodpovídá vytvořenému v teto metodě modelu, a proto, co bylo očekáváno, při generování na jeho základě v modelu v EA byla ztracená největší část informace. Model vytvořený v EA vypadal podobně modelu vytvořenému jako základ v Umbrello.

Zdrojový kód, vygenerovaný Umbrello z modelu, byl rozsáhlejší a obsahoval více informace, vzhledem k tomu se očekávalo, že soubor .xmi, ve kterém je uložen tento model, je schopný při jeho importu předat více informace metodě EA. Proto s cílem ušetřeni ruční práce jako základní model EA byl importován soubor .xmi modelu Umbrello.

4.3.1 Popis metody

EA je výkonný CASE-nástroj pro vizuální modelování a návrh softwaru založený na OMG UML. Vzhledem k tomu, že EA nabízí širokou škálu funkcí, je určena pro profesionály, kteří se zabývají vývojem, testováním a implementací softwaru. EA plně podporuje notaci jazyka UML a umožňuje vytvářet diagramy všech typů.

EA nabízí tři základní druhu edice: Desktop Edition, Professional Edition, Corporate Edition.

- EA Desktop Edition je určen pro samostatné vývojáře, protože nepodporuje multi-user přístup. Používá se především pro modelování designu,

protože neumožňuje některé funkcionality vývoje (import-export kódů a další).

- EA Professional Edition je plně funkční UML programovací prostředí.
- EA Corporate Edition obsahuje všechny funkce Desktop a Professional Edition, a také poskytuje možnost připojení k MySQL, SQL Server, PostgreSQL, Sybase Adaptive Server Anywhere, Oracle.
- Kromě toho existuje verze EA Lite free read-only určená pro zákazníky a zaměstnanci, které se nezabývají vývojem, s cílem umožnit registrovaným uživatelům zobrazovat UML diagramy.

Pro analýzu byl použit EA Academic, pro který ČVUT v Praze poskytuje licence, protože v plně míře splňuje požadavky k funkcionalitě zvoleného CASE-nástroje [44].

4.3.2 Modelování

Na začátku EA byl nainstalován na počítač a následně byl v něm vytvořen projekt, ve kterém se modelovalo. Při importu modelů z souboru .xmi do projektu program vyžadoval zvolit jméno balíčku, kde se bude nacházet model, cestu k souboru .xmi, jazyk, podle kterého budou importované DataTypes, a určit následné vlastnosti importu:

- Import diagrams: označuje ze importují se diagramy.
- Strip GUIDs: odstraňuje Universál Identifier a tím umožňuje import do stejného modelu dvakrát, přičemž druhý import vyžaduje zaškrtnutí tohoto políčka s cílem vytvoření nových GUIDs, co působí vyhnutí kolizím prvků.
- Write log soubor: určuje, jestli dělat zápis protokolu importní aktivity.
- Import using single transaction: pokud je zaškrtnuto této políčko, soubor XMI je importován v jedné transakci, která se nedoporučuje pro import velkých souborů. Není-li zaškrtnuto políčko datové položky jsou importovány zvlášť samostatně, co pomůže identifikovat problémové položky, aniž by byl celý import blokován.

Pro import byly zvolené jazyk C++ dvě vlastnosti: Import diagrams a Write log soubor. Poslední bylo vybráno s cílem sledovat a odhalit problémy, když takové nastanou během importu.

Navíc EA nabízí dvě různých možností sloučení obsahu souboru XMI s balíčkem, se kterým se aktuálně pracuje. Kvůli tomu, že funkce importu byla použita pouze pro výměnu modely mezi jednotlivými metodami, což znamená, že původní projekt byl prázdný, žádná z možností nebyla zvolena.

Při importu nastal problém s přidáváním klasifikátory každé třídy Class Diagram. Té elementy, které EA byl schopný přečit, byly v UML 1.4 profilu, ale například Class Diagram je v <XMI.extensions xmi.extender="umbrello», a proto se nemohl importovat do EA. Žádné jiné digramy nebyly importované do metody, protože .xmi soubor její neobsahoval.

Ve výsledku byly importovány všechny třídy, ze kterých byl modelován Class Diagram, jak je ukázáno na obrázku 4.7. Diagram EA odpovídal diagramu Umbrello za výjimkami:

- zmizel vztah kompozice mezi `wp_posts` a `revision`;
- zmizela vlastnost `const` u operací;
- zmizela implicitní hodnota `post_status` třídy `revision`;
- mezi třídy `wp_terms` a `wp_term_taxonomy` byl uskutečněn vztah realizace místo agregace.

V modelu byly provedena řada změn včetně odstranění vztahu mezi třídy a odpovídající vztahům přidané atributy, protože se očekávalo, že EA nabídne vlastní způsob propojení tříd. Při vytvoření vztahu dědičnosti mezi třídami byly zvolené operace, které dědí class-child od class-parent. Vztahy, kterými Enum `post_status` byl propojen s jinými třídami, zmizely a EA nevytvořil její znovu.

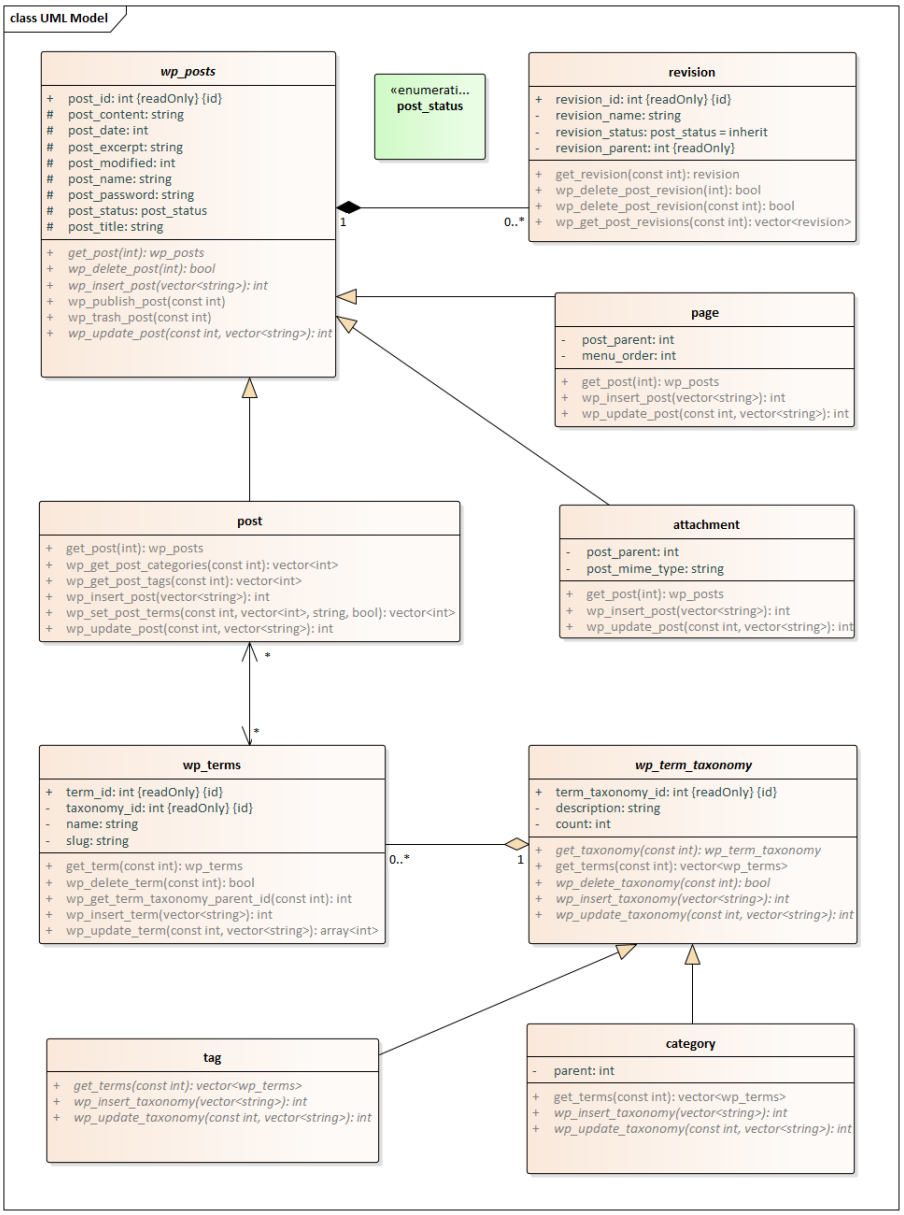
Umbrello neumožňovalo nastavit vlastnost `const` u atributů, a proto byly vytvořené nové odpovídající datové typy. EA má zvláštní parametr atributu `const`, kde lze nastavit `true` nebo `false`, ale metoda nebyla schopna rozeznat `const` u datového typu jako parametr a vytvořila nové datové typy, proto změny byly provedené ručně.

EA umožňuje vygenerovat gettery a settery pro každý atribut třídy přímo v diagramu, ale nelze nastavit takovou možnost pro všechny atributy najednou, je potřeba určovat zvlášť pro každý atribut. Existence takových operace nemůže výrazně změnit kompletnost budoucí aplikace, a proto tento krok byl vynechán.

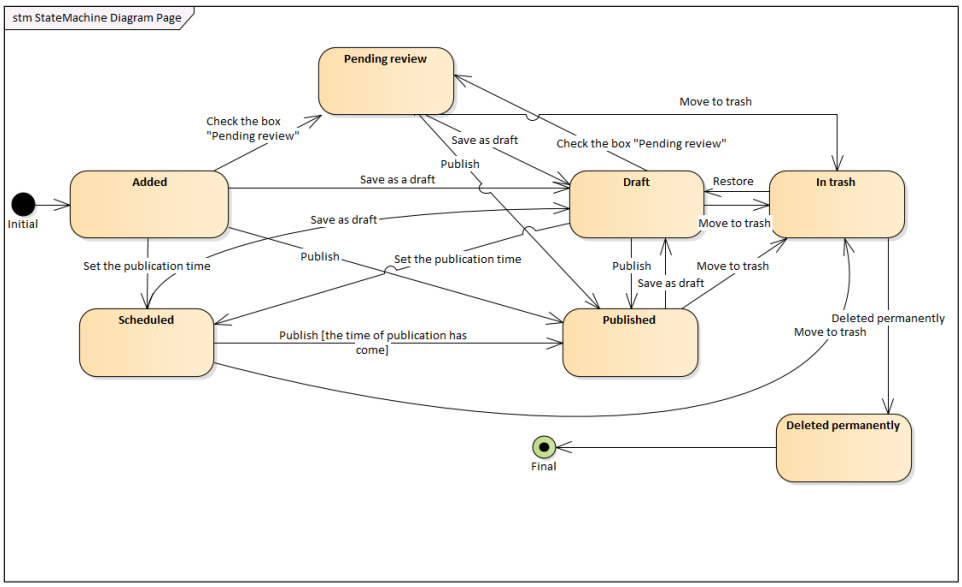
Chybějící State Machine Diagram a Activity Diagram byly přidané k třídě `page` a vymodelované ručně od začátku. Výsledné modely jsou ukázané na obrázcích 4.8 a 4.9. Na rozdíl od ostatních metod, které to neposkytovaly, přitom, že se taková funkcionalita považovala za základní, EA umožňoval popsat přechody v State Machine Diagram, což bylo uděláno s cílem zvýšení přesnosti modelu.

4.3.3 Generování kódu

Prvním krokem generování kódu bylo uvedení programovacího jazyku v parametrech třídy. Když jazyk nebude nastaven, taková třída není nabízena ke generování jakoby EA ji nevidí.



Obrázek 4.7: Class Diagram Enterprise Architect



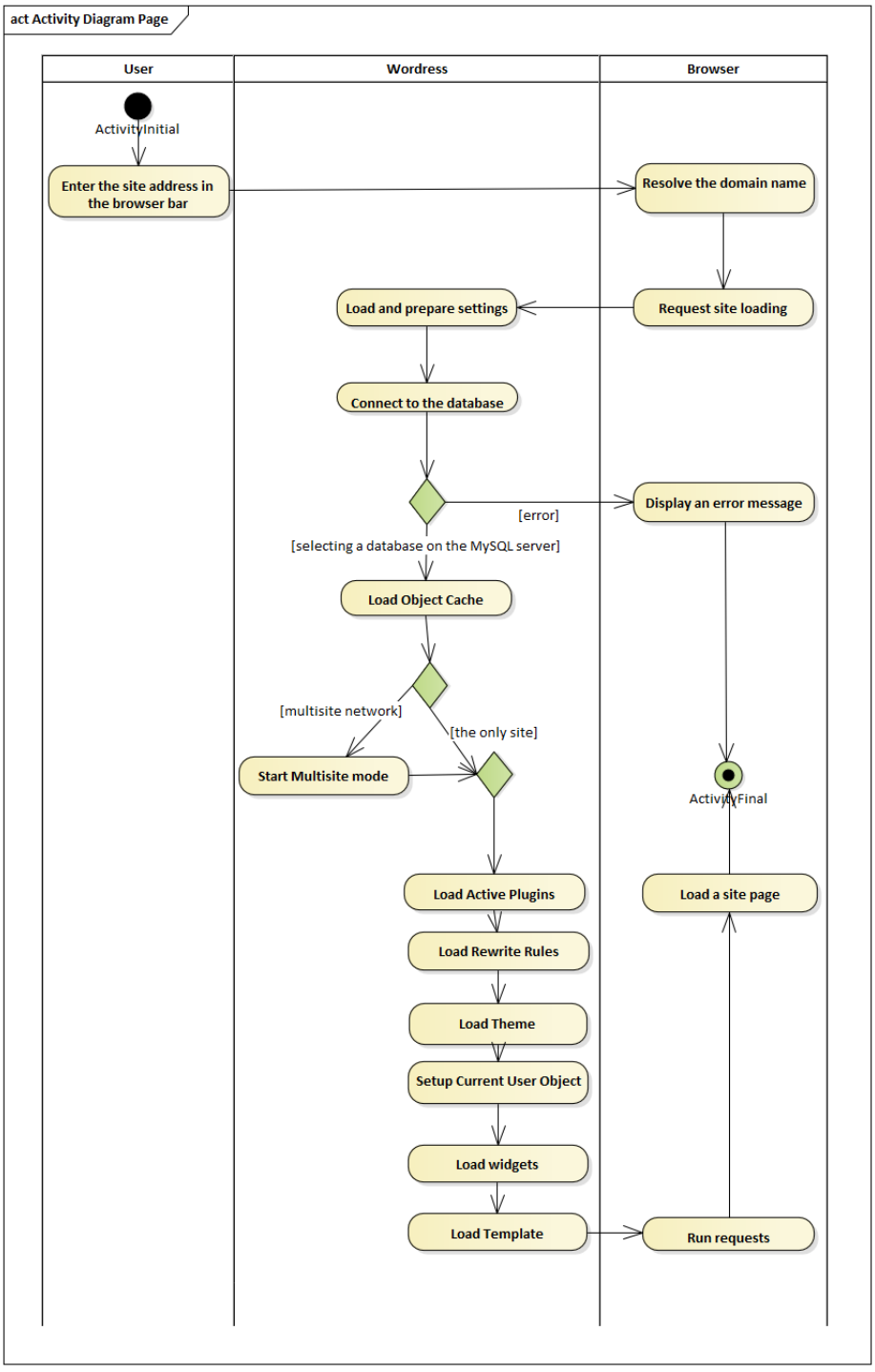
Obrázek 4.8: State Machine Diagram Enterprise Architect

Následné kliknutí tlačítka „Generate“ ve složce „Code“ hlavního menu působilo otevření okénka pro nastavení parametru generování: balíčku, kde se nachází elementy ke generování, cesty ke složce, kam budou uložené výsledné soubory, a vlastnosti synchronizace, která určuje, zda synchronizovat už existující v souboru kód s vygenerovaným nebo přepsat. Pro generování byly zvolené všechny třídy, ale EA umožňuje jakékoliv třídy vynechat.

4.3.4 Výsledný kód a dokumentace

Generování kódu proběhlo bez problémů. Ale zjistilo se, že generování kódu třídy `page` nepůsobilo generování kódu diagramů s ní spojených. Následně podle postupu uvedené v dokumentace v modelu byl vytvořen element Artefact ExecutableStateMachine a do něho přidán State Machine Diagram, co melo přivést k poskytnuti možnosti generování kódu tohoto diagramu zvlášť. Potom po kliknutí na artefakt se otevřelo rozevírací menu, ale tlačítko „Simulate“, které je definováno v dokumentace, tam nebylo. Protože žádný jiný způsob, jak objevit toto tlačítko, nebo vygenerovat zdrojový kód z State Machine Diagram s použitím jiného postupu nebyl v dokumentaci uveden, generování kódu z State Machine Diagram Page se nepodařilo [45].

Activity Diagram nebyl dostatečný a chyběly v něm některé elementy (Call Actions, CreateObjectActions a další), aby byl kompletní pro generování zdrojového kódu v EA. Ve výsledku proběhlo generování pouze z Class Diagram a byly vytvořené soubory .h a .cpp každé třídy Class Diagram. Výsledný kód třídy `wp_posts` je ukázán níže.



Obrázek 4.9: Activity Diagram Enterprise Architect

Datový kontejner vector byl brán jako nový datový typ, proto EA pro něho vytvořil vlastní soubor. Vzhledem k tomu, ze metoda měla umět pracovat s STL C++, ve konfiguračním nastavení jazyka C++ vector byl přidán do Additional Collection Classes a vygenerován nový kód, kde vector už nebyl novým datovým typem a neměl vlastní soubor. Ale hlavičkový soubor vector nebyl uveden v jiných souborech, jak se očekávalo, a proto všechny potřebné hlavičkové soubory je nutné přidat ručně stejně jako prostory jmen.

Vztah kompozice mezi třídy `wp_posts` a `revision` byl realizován přidáním atributů `*m_revision` datového typu vector do třídy `wp_posts`. Vztah agregace mezi třídami `wp_terms` a `wp_term_taxonomy` byl uskutečněn stejně – přidáním atributu `*m_wp_terms` datového typu vector ve třídě `wp_term_taxonomy`. Vztah asociace mezi třídami`posta` `wp_terms` na rozdíl byl založen dvěma novými atributy: `*m_post` datového typu vector třídy `wp_tems` a `*m_wp_terms` datového typu vector třídy `post`. Výsledný kód byl korektní a nepotřeboval žádné opravy.

Generování dokumentace probíhá zvlášť a stačí vybrat balíček, pro kterou je bude generována, a stisknout F8 na klávesnici. Následně lze vybrat parametry pro budoucí dokumentace: formát souboru, šablonu, vzhled diagramu, velikost písmen a další. Kvůli tomu, že šablony dovolí vygenerovat dokumentace, zohledňující model z různých stran a obsahující určitou konkrétní informaci o elementech modelů nebo naopak model jako celek, lze říct, že dokumentace, vytvořená v EA je kompletní a postačující.

```
1  ///////////////////////////////////////////////////////////
2  //  wp_posts.h
3  //  Implementation of the Class wp_posts
4  //  Created on:      06-May-2021 15:18:02
5  ///////////////////////////////////////////////////////////
6
7  #if !defined(EA_D720E535_A008_418c_A1F5_BEB305190B18__INCLUDED_)
8  #define EA_D720E535_A008_418c_A1F5_BEB305190B18__INCLUDED_
9
10 #include "post_status.h"
11 #include "revision.h"
12
13 class wp_posts
14 {
15
16 public:
17   wp_posts();
18   virtual ~wp_posts();
19   const int post_id;
20   vector<revision> *m_revision;
21
22   virtual const wp_posts get_post(int post_id);
23   virtual bool wp_delete_post(int post_id);
24   virtual int wp_insert_post(vector<string> post_parameters);
25   wp_publish_post(const int post_id);
26   wp_trash_post(const int post_id);
```

```
27   virtual int wp_update_post(const int post_id, vector<string>
       post_parameters);
28
29 protected:
30   string post_content;
31   int post_date;
32   string post_excerpt;
33   int post_modified;
34   string post_name;
35   string post_password;
36   post_status post_status;
37   string post_title;
38
39 };
40 #endif // !defined(
       EA_D720E535_A008_418c_A1F5_BEB305190B18__INCLUDED_)
```

4.3.5 Dokončení aplikace

Přesto že vztahy v modelu byly implementované korektně a není potřeba provádět změny, nebyly nedostatečně zohledněné omezení vztahu. Objekt třídy `revision` nemůže existovat zvlášť a vždy musí spojen s objektem třídy `wp_posts`, ale ve výsledné aplikací je to umožněno.

Vzhledem k tomu, že pomocí vygenerovaného kódu lze přidávat, měnit a propojovat elementy aplikace bez nutnosti provedení oprav kódu, lze považovat výsledný kód za minimální základ WP. Přesto že nenabízí v plné míře všechny základní funkcionality kvůli neexistenci zdrojového kódu vygenerovaného z Activity a State Machine Diagrams, aplikace je funkční. K dokončení aplikace je potřeba přidat řadu nových prvku do Activity Diagram a vyřešit problém s neexistenci možností generování kódu z State Machine Diagram.

4.3.6 Identifikované problémy

- Nečekané chování tlačítka „Undo“: při použití po přidání elementu, pouze smaže prvek z diagramu, ale neodstraní trvale.
- V případě, že pro nějakou třídu není nastaven jazyk, nebude upozorněno, že tato třída nebude generována do kódu, co může působit problémy při generování velkého množství tříd.
- Chybí realizace omezení vztahů typu agregace.

Kapitola 5

Vyhodnocení přínosů analyzovaných metod

5.1 Porovnání přístupů mezi sebou

S cílem jasně vydefinovat přínosy a nevýhody každé metody byly vymezené pět kritéria, podle kterých metody byly srovnány: snadnost použití, přehlednost kódu, kompletnost aplikace, možnosti budoucího rozvoje a udržitelnost aplikace a další funkcionality. Srovnání zohledňuje jednotlivou metodu s různých stran a poskytuje možnost posoudit, jaká z nich zefektivní práci a usnadní vývoj. Ocenění možných přínosů pomůže vybrat nejvhodnější metodu pro použití v praxe při vývoje určité systémy pomocí MDD.

5.1.1 Snadnost použití

Během instalací Umbrello a EA nenastal žádný problém. Registrace na webové stránce GenMyModel proběhla bez komplikací.

Pro pracování v GenMyModel a Umbrello nenastala nutnost obrátit se k dokumentace s cílem vyřešení problému nebo zajištění postupu. Modelování v obojích metodách bylo uděláno intuitivně.

Rychlé a efektivně pracovat v EA novému uživatelovi bez čtení dokumentace nevyjde, protože metoda má rozsáhlou funkcionalitu. I když je potřeba použít pouze základní, jednoduché se lze ztratit ve velkém množství rozevíracích menu a nastavení vlastností a parametrů, přičemž neporozumění toho co dělá jednotlivá funkcionalita nebo vlastnost může vést k neočekávaných výsledkům. Kromě toho, těžko odhalit, co přesně působilo určitý efekt, co vynutí prostudovat celou oblast.

Vzhledem k rozsáhlé funkcionalitě, EA má obrovskou dokumentaci, která není schopna zohlednit všechny situace, které můžou nastat během vývoje. Lze očekávat narážení na nějaké detaily, které nejsou jasně popsané nebo nedostatečně vysvětlené v dokumentace, a proto můžou být prozkoumaný pouze

v praxi, například, nutnost zvolení jazyku pro každou třídu, jak bylo zjištěno v předchozí kapitole. Přičemž, po analýze zůstal problém, které nebylo možné vyřešit pomocí dokumentací.

Ve všech metodách při modelování diagramů jsou v menu elementů modelu (toolbox) ukázané pouze elementy, které můžou být použité v tomto diagramu. V GenMyModel a EA ikonky vypadají jako obrázky s názvy, ale Umbrello má pouze ikonky bez názvu. Vzhledem k většímu množství funkcionalit, z čeho vyplývá větší množství elementů modelu, všechny ikonky v EA jsou uložené do složek, které je potřeba otevírat kliknutím. Při modelování jednoduchých systémů hledání jednotlivého elementu může zbytečně zabrat více času, při tom, že se bude používat do 20 základních prvků. Mezitím Umbrello naopak umožňuje nastavit ikonky hlavního menu a menu elementů modelu a následně vyloučit nebo přidat nějaké.

Jak bylo zmíněno v předchozí kapitole Umbrello nemá automatické ukládání modelů, což ve praxi při modelování rozsáhlého systému bude mít velký význam. Nelze očekávat, že při použití tohoto nástroje vývojář bude ručně ukládat model každou minutu, co nepochybně bude působit ztrátu části už udělanou práci a nutnost opakovat stejné kroky po znovuotevření Umbrello.

5.1.2 Přehlednost kódu

Pravidla formátování kódu byly dodrženy v každé metodě. Jedinou výjimkou v Umbrello je mezera mezi klíčovými slovy specifikátorů přístupu jazyka C++ a dvojtečkou, co nemělo správně být. Žádné názvy tříd, atributů nebo operaci nebyly opravené podle pravidel formátování kódu.

Kromě toho, Umbello a EA je možnost samostatné specifikovat, jak bude vypadat kód. Obě metody umožňují vybrat písmo, velikost a styl kódu, takže obarvit jednotlivé části kódu různými barvami. Při tom, jestli Umbrello nabízí omezený seznam vlastností, které lze vypnout nebo zapnout, EA má na výběr několik desítek šablon formátování kódu. Každou šablonu lze změnit podle uživatelských požadavků nebo vytvořit vlastní. Ve výsledku, formátování kódu v Umbrello je jednoduché, ale omezené, v EA formátování kódu má širokou funkcionalitu, ale obtížnější k zvládnutí, protože potřebuje porozumění šablonu a kódů v něm. Metoda GenMyModel nenabízí žádné uživatelské změny formátování kódu.

V Umbrello v případě datového typu string bylo určeno do jakého prostoru jmen C++ spadá, ale v případě vektorů informace o prostoru jmen chyběla u vektorů i datového typu, kterým vektor byl inicializován. To vynucuje buď určit prostor jmen před každým vynechaným prvkem nebo naopak smazat u označených a napsat jako hlavičkový soubor nahoře.

5.1.3 Kompletnost aplikace

Vzhledem k tomu, že metoda GenMyModel ve vygenerovaném kódu odráží pouze třídy a atributy, použití metody GenMyModel nezrychlilo vývoj systému. Výsledná aplikace nemůže fungovat, protože neumožněno vytvořit instalaci třídy ani provést s ní nějakou změnu. Třídy nejsou propojené v kódu, takže vygenerovaný kód lze nazvat složkou souboru, ale ne souvislou aplikací.

Při tom, že při modelování Umbrello rozumí že vztahy elementů modelu není pouze grafická záležitost a přidává atribut do jedné z tříd, pojmenování a určení datového typu je na uživatelovi. Navíc nějaké vztahy například asociace stejně zůstávají pouze šipkou na diagramu a neodráží se v kódu. Přičemž v kódu třídy nejsou propojené použitím hlavičkových souborů a uživatel musí prohlédnout kód, aby porozuměl jestli třída používá objekty jinou třídy a ručně přidat nutně hlavičkové soubory.

Vygenerovaná aplikace je schopna vytvářet objekty a měnit je, ale některé vztahy mezi třídy uživatel musí dělat manuálně, tato metoda jenom naznačuje, že třídy musí být propojené novým atributem. Ostatní vztahy nejsou zohledněné v kódu. Z toho vyplývá, že třídy nefungují mezi sebou ve výsledné aplikaci korektně a nelze tento problém opravit s pomocí metody Umbrello.

Kód vygenerovaný v metodě EA je postačující pro fungování základních funkcionalit systémů a v plné míře odpovídá vytvořenému modelu Class Diagram. Vzhledem k tomu, že kód ze Activity Diagram nebyl vygenerován kvůli neexistenci v diagramu určitých elementů nezbytně nutných pro generování, stačí přidat této prvky bez provedení oprav v už modelovaném Activity Diagram. EA umí generovat zdrojový kód z State Machine Diagram, proto je nutné s použitím jiných zdrojů mimo dokumentace zjistit, jak uskutečnit generování kódu jiným způsobem, kromě popsaného v dokumentaci, a podle výsledků postupovat dále.

5.1.4 Možnosti budoucího rozvoje a udržitelnost aplikace

Metoda GenMyModel a Umbrello vygenerovala kód z Class Diagram, ale generování kódu z State Machine a Activity Diagrams nebyla provedena, protože metody to neumějí. GenMyModel umožňuje vytvořit vlastní šablonu generování kódu, ale není zřejmé, že šablona umožní ovlivnit nějaký jiný diagram kromě Class Diagram.

Umbrello poskytuje licence GPL, a proto lze přidat vlastní kód, který umožní generování kódu z těchto diagramů. Pro realizace takové funkcionality je potřeba důrazně prozkoumání kódu tohoto CASE-nástroje a dobrou znalost jazyků C++, protože Umbrello je napsáno na C++. Kromě toho, je určitě možnost opravit základní kód tak, aby vytvářel správné vztahy mezi třídy Class Diagram v kódu.

Přesto že EA nabízí generování kódu ze State Machine Diagram a Activity Diagram uživatel může vytvořit jakoukoliv vlastní šablonu generování

kódu podle svých požadavků, ale pozměnit kód samou metody EA nemůže, to znamená, že je možné ovlivnit, jak funguje nějaká funkce EA, ale odstranění ji nebo přidání nové funkce je zakázáno kvůli licence.

Při dalším vývoji systému může nastat situace, když bude napsán zdrojový kód a pak je potřeba vytvořit z něho model. S tímto cílem Umbrello a EA nabízí reverzní inženýrství pro sadu programováních jazyků.

Na rozdíl od GenMyModel a Umbrello EA, kromě toho, umožňuje synchronizaci už existujícího kódu s modelem, co dovoluje přidávat nové elementy, atributy, operace, popisy do modelu a do zdrojového kódu najednou a následně doplňovat vygenerované z modelu zdrojový kód do programu automatické bez obav je ztratit nebo poškodit změny udělané v kódu.

Práce ve velkých týmech vyžaduje důkladné řízení procesu vývoje a zajištění omezení přístupových práv pro ochranu dat uživatele před neoprávněným přístupem ostatních uživatelů. Pro usnadnění prací v týmu GenMyModel a EA poskytují online přístup v reálném čase, registrace uživatele s přístupem pouze k prohlížení bez možnosti prováděn změny v projektu a centrální úložiště modelů, který umožňuje snadno a současně simulovat spolupráci. Metoda Umbrello není zaměřena na týmovou práce, takže podporuje režim pro současný přístup nebo spolupráce více uživatelů.

5.1.5 Základní parametry a další funkcionality přístupů

Funkcionality metod je klíčovým faktorem přínosu použití metody během vývoje systému a předběžně její prozkoumání pomáhá ohodnotit kolik času průměrně se dá ušetřit při zvolení určitého CASE-nástroje. Proto byly prozkoumané další možnosti, které nabízí jednotlivá metoda, které nebyly aplikované během analýzy přístupů a shrnuté v tabulce níže, protože informace v tabulce má lepší přehlednost a jasnost, než v textu.

Tabulka odráží konkrétní informace o funkcionalitách, který už byly zmíněné dříve, a upřesňuje její určitá omezení použití: programovací jazyky, formáty souborů a verze, pro která funkcionalita je určena. Kvůli tomu, že některé metody mají více než jednu edici, v porovnání byly zohledněné možnosti edice která nabízí nejvíce funkcí. Vzhledem k tomu, že každá z metod dovoluje vždy přepnout se na jinou edice během vývoje bez ztráty žádné části projektu, nebylo nutné upřesňovat v tabulce, pro jakou edice je dostupná funkcionalita.

Ve výsledku v tabulce byly zohledněné funkcionality edici metod: GenMyModel Enterprise, Umbrello 2.32 (nemá žádné edice) a EA Ultimate. Kvůli tomu, že každá metoda má vlastní rozlišující mezi sebou požadavky na systém, na kterém může běžet, kromě funkcionalit, byly porovnané základní parametry metod v tabulce 5.1: platformy, na kterých může běžet metoda, a jazyky, ve kterých je dostupné uživatelské rozhraní, jazyky modelování a zdrojového kódu metody.

V tabulce 5.2 byly prozkoumané následující funkcionality: jazyky přímého a inverzního inženýrství a formát souboru importu, exportu a generování do-

kumentace.

Import a export XMI je zvlášť, protože brán jako jazyk výměny metamodelu mezi jednotlivými CASE-nástroje a je ukázaná verze XMI, protože kompatibilita nástrojů pomoci v jedné verze XMI nezaručuje kompatibilitu při výměně metamodelu v dalších verzích.

Import souborů vytvořených v jiných CASE-nástrojích je taky zohledněn zvlášť s cílem vymezit ty, s kterými určitá metoda je zaručeně kompatibilní.

V případě, když metoda nenabízela určitou funkcionalitu v odpovídající buňce tabulky je uveden "-". V případě, že metoda funkcionalitu umožňovala, ale nespecifikovala omezení jazyku, verze a formátů je uveden v buňce tabulky "+".

5.2 Porovnání přístupů s tradičním vývojem software

Pro určení vhodnosti použití MDD při vývoje informačního systému, bylo provedeno porovnání, kolik času ušetřil vývoj systému WP pomocí MDD v srovnání s tradičním vývojem software. Přesto, že přístupu tradičního vývoje existuje více, že jeden, základní klíčovou vlastností tradičního vývoje je ručně psaní kódu a pro odhadnutí průměrného času v níže vymezeních fázích není podstatné, jaký z tradicích přístupů byl použit.

Nicméně logika procesu vývoje informačního systému jsou téměř ve všech případech společná a obsahuje tvorbu koncepce, vypracování technické úlohy, navrhování, tvorbu systému, uvedení systému do provozu [46]. K porovnání vývoje pomocí MDD s tradičním vývojem software, byly vymezené následující základní fáze vývoje, slučující nebo vylučující některé už vydefinované: příprava k tvorbě systému (tvorba koncepce, vypracování technické úlohy, navrhování), modelování systémů a vytváření kódu a dokumentace.

Pro začátek se předpokládalo že tým který se zabývá návrhem, modelováním a implementaci a případným provedením změn či oprav systému má dobrou znalost jazyka C++ a praktickou zkušenost s programováním. Bez této podmínky nebylo možné korektně odhadnout čas, který zabere jednotlivá fáze, protože nešlo vymezit, kolik času bude stráveno na řešení problémech vzniklých z důvodu neexistenci zkušenosti programátora.

5.2.1 Příprava k použití metod

V obou přístupech není automatizován nebo zrychlen proces navrhování architektury a vydefinování většiny funkčních požadavků systému, proto trvá stejný čas a nemá smysl ho počítat. Kroky, pro které nebylo možné přesně odhadnout čas, například, zlepšení komunikace v týmu a tím zvýšení kvality a rychlosti vývoj, byly vynechané. V tradičním přístupu vývoje software po těchto krocích fáze přípravy k tvorbě systému byla dokončena.

Tabulka 5.1: Základní parametry metod

Základní parametry	**GenMyModel**	**Umbrello**	**Enterprise Architect**
Platformy	online	Windows, Linux, macOS	Windows, Linux, macOS
Jazyky uživatelského rozhraní	Angličtina, francouzština	angličtina	Angličtina, němčina, japonština, španělština, čínština, francouzština
Jazyky modelování	Archimate, BPMN, UML, RDS, Flowchart, SysML, ArchiMate, JPA, EMF, DMN	UML	UML 2.5, SysML 1.5, BPMN 2.0, DMN, BMM, MARTE 1.2, BPEL, SoaML, SPEM, WSDL, XSD, DDS, ArchiMate 3.0, ArcGIS, IFML, CMMN, Geography Markup Language (GML), ODM, OWL and RDF, VDML 1.0
Jazyk zdrojového kódu metody	Javascript and HTML5	C++	C++

Tabulka 5.2: Funkcionality metod

Funkcionalita	**GenMyModel**	**Umbrello**	**Enterprise Architect**
Jazyky přímého inženýrství	C#, C++, Java, PHP, JavaScript, HTML, Lua, Python, Ruby, SQL,TypeScript	ActionScript, Ada, C++, C#, D, IDL, Java, JavaScript, MySQL, Pascal, Perl, PHP, PostgreSQL, Python, Ruby, SQL, Tcl, Vala, XMLSchema	C++, Java, C#, VB, VB.Net, Visual Basic , Delphi, PHP, Python
Jazyky reverzního inženýrství	-	Ada, C++, C#, IDL, Java, MySQL, Pascal, PHP (od verze 2.24), PostgreSQL, Python	C++, Java, C#, VB, VB.Net, Visual Basic , Delphi, PHP, Python
Import	XMI	XMI	XMI, CSV
Export	SVG, PNG, JPEG	DocBook, XHTML, PNG	
Generování dokumentace	PDF, MS Word	Přímo ve zdrojovém kódu	PFD, MS Word
Import a export XMI	-	XMI 1.2/XMI 1.4	XMI 2.x
Podporované formáty importu souborů vytvořených v jiných CASE-nástrojích	StarUML	Argo UML, Enterprise Architect, NSUML, Poseidon for UML, UNISYS	Eclipse Visual Studio

Vývoj pomocí MDD by rozdělen na dvě situace. Když se vývojem zabývá tým který má zkušenost s modelováním a práci v zvoleném k vývoje v CASE-nastojí, školení zaměstnanců není potřeba, co znamená, že fáze přípravy
k tvorbě informačního systému je stejně dokončena. V opačném případě se nelze vyhnout útraty času nutnému k alespoň základnímu seznámení vývojáře s jazykem modelování a aplikaci pro modelování a vývoj pomocí MDD.

Vzhledem k tomu, že podle analýzy EA je nejefektivnější metoda, protože jediná vygenerovala zdrojový kód kompletní aplikaci, použitím této metody lze ušetřit nejvíce času ve fáze psaní kódu. Během analýzy každé metody nebylo možné vymezit všechny změny, které budou nezbytně nutné k opravě vygenerovaného kódu případných dalších diagramů informačního systému, proto s cílem přesněji odhadnout, kolik hodin zabere fáze vytvoření kódu, pro srovnání byla zvolena metoda EA, která skoro nevyžadovala změny. Při zvolení jiné metody, čas na školení v CASE-nástroji není potřeba počítat vzhledem k možnosti intuitivního ovládání bez čtení dokumentací.

Pro korektního odhadu času, kolik bude trvat přiměřenému zaměstnancovi IT společnosti naučit se modelovat v UML a ovládat EA, za délku školení byla uvedena délka reálného školení společnost ICT Pro[47] - dva dny. ICT Pro je poskytovatelem komplexních služeb v oblasti IT a školení jednotlivcům a organizacím. Protože základní kurz UML probíhá v prostředí EA, nebyla potřeba uvažovat čas nutný pro zvládnutí UML a EA zvlášť.

Následná příprava prostředí pro vývoj pomocí MDD zabírá přibližně pět minut, protože registrace a instalace systému je jednoduchým krokem.

5.2.2 Modelování

Protože v předchozí fázi byly vymezené všechny požadavky na systém a navrhnuto řešení, ale v případě tradičního vývoje se nemodeluje, fáze modelování nezabrala žádný čas.

Při vývoje pomocí MDD je tato fáze klíčová, protože určuje, jak bude vypadat výsledný systém. Kvůli tomu, že zaměstnancově buď uměli pracovat v CASE-nastojí nebo měli školení, fyzické umístění prvků a vztahů včetně nastavení její vlastnosti na modelu trvalo hodinu.

V případě vývoje jednoduchého systému s malým počtem funkcí jasně vymezenými požadavky TV dovoluje se obejít bez vytváření jakýchkoliv modelů a může se počítat s diagramem nakresleným na tabule, což nezabírá žádný čas.

5.2.3 Vytvoření kódu

Při vývoje pomoci MDD stačilo pouze zmáčknout tlačítko a výsledkem byl kompletní zdrojový kód. Kromě toho, na zvolení vhodné šablony ke generování případně nezbytně nutné nastavení vlastnosti generátoru nebo obecných

parametrů pro určitý programovací jazyk bylo stráveno patnáct minut. Následně porozumění kódu a drobné opravy trvaly ještě patnáct minut vzhledem k dobré znalosti projektované aplikací, její tříd, atributů a operací. Nicméně odhadnutá doba záleží na zkušenosti týmu a může být zkrácena, protože zkušený vývojář ví, jaké problémy lze očekávat od konkrétní metody nebo jaké opravy kódu budou nespíš nutné.

Tradiční vývoj vyžaduje psaní kódu manuálně od začátku a do konce. Vzhledem k tomu, že existující model systému je nekompletní ale pouze orientační, vývojář měl strávit jednu až dvě hodiny na vymezení jednotlivých vhodných atributů a operací včetně datových typů a klasifikátorů typu. Protože bez diagramu je obtížněji pochopit jak mají fungovat vztahy mezi třídy, byla potřeba jedna hodina na promyšlení vazeb a jejich omezení. Nad vytvářením dvou souboru .h a .cpp pro každou třídu a přípravou kódu (přidání knihoven, prostorů jmen, hlavičkových souborů) se strávila polovina hodiny, nad samostatným programováním atributů a operaci – hodina a půl až dvě hodiny.

Vzhledem k tomu že při vývoje pomocí MDD se nelze vyhnout psaní dokumentaci, při tradičním vývoje vývoje bylo zahrnut pouze čas na přípravu vývojářem šablony dokumentace – dvě hodiny.

Musela se předpokládat i situace, že se změní pracovník, který plnil určitý úkol. V případě použití tradičního přístupu nový vývojář musí se seznámit s cizím zdrojovým kódem. Vzhledem k tomu, že na začátku bylo uvedeno, že každý člen vývojového týmu programuje kvalitní kód a dokumentace zdrojového kódu je postačující, konkrétní a popisuje lehce pochopitelně jakýkoliv část systému. Jako nejefektivnější případ byla uvažována situace, když nový vývojář měl možnost komunikace a projednání nejasnosti s bývalým vývojářem systému. V takovém případě čtení dokumentace, pochopení systému jako celku a zároveň detailu zabere čtyři hodiny. Samostatné zvládnutí tohoto úkolu může zabrat až dvakrát více času – až osm hodin.

V případě použití modelu nový vývojář musí se seznámit s klíčovými vlastnostmi systému a pak samostatně na základě existujících modelů pochopit detaily. Protože byla předpokládaná situace, že každý člen týmu UML umí a je schopen pracovat v určitém CASE-nástroji, kde se nachází modely a diagramy, potřebuje hodinu na pochopení modelů a základů celého systému. Následné čtení dokumentace je výrazně jednodušší a vývojář bude potřebovat taky hodinu.

5.3 Přínosy analyzovaných metod

V následující kapitole jsou popsané přínosy vývoje řízeného modelem vymezených na základě analýzy použití metod GenMyModel, Umbrello a EA a předpokladů, jak lze využít probrané funkcionality metod, během vývoje, provozu a údržby informačního systému a při případném rozvoje.

5.3.1 Vývoj informačního systému

Použití modelu poskytuje významné výhody, protože člověk je schopen pochopit schéma mnohem rychleji než text. Navíc model umožňuje prezentovat informace kompaktně a strukturovaně. V důsledku toho se při použití modelů zvyšuje výkon vývojového týmu.

Model je jakýmsi mostem mezi zákazníkem a vývojáři, protože pochopení modelu zákazníkem je možné, zatímco kód není. Některé změny v modelu je možné provést přímo na schůzce se zákazníkem a okamžitě mu ukázal výsledek, že zvyšuje rychlost vývoje systému a snižuje pravděpodobnost, že z důvodu nedorozumění se zákazníkem se nakonec otočil výsledek, než se očekávalo.

Model navíc zkracuje propast mezi architekty a programátory. Použití modelu neumožňuje návrh architektury systému, který následně nelze naprogramovat. Za zmínku také stojí, že se zkracuje čas na sladění dokumentace. Mezi tím, MDD zlepšuje komunikaci v samotném týmu vývojářů mezi sebou, což umožňuje centralizovat vývoj, rychle a efektivně sdílet informace, a také usnadňuje integraci nových členů týmu v případě potřeby, protože pochopit model výrazně rychlejší, než cizí kód.

Model zajišťuje, že vytvořený systém bude odpovídat celkové obchodní logice, která byla projednána v prvních fázích vývoje, protože většina nástrojů podporují automatické generování CIM na PIM, a pak v PSM. Proto je opět vyloučeno, že systém nakonec nesplní původní požadavky.

MDD snaží o maximální automatizaci, což pomáhá snížit množství práce pro vývojáře a eliminovat případné náhodné chyby, které dělají lidé, než snižuje množství vad v konečném kódu. I ten primitivní automatické generování kódu z modelu umožňuje ušetřit čas na psaní opakující se, nudné a šablony částech kódu jak bylo zjištěno při analýze všech metod.

Díky uvedeným výhodám, výrazně snižuje čas a složitost vývoje, že snižuje jeho hodnotu, a celkově zvyšuje efektivitu a kvalitu systému.

5.3.2 Provoz a údržba informačního systému

V případě, že na konci vývoje se zjistí, že systém byl nesprávně navržen, lze se vrátit k modelu, provést potřebné úpravy a znovu vygenerovat kód z něho. Provádění změn v modelu trvá podstatně méně času, než provedení změn v samotném kódu. Navíc MDD umožňuje opravit konkrétní část systému, aniž by došlo k narušení fungování ostatních částí. V případe provedení změn v systému lze vygenerovat aktuální dokumentaci jediným kliknutím.

Pokud je nutné provést změnu v důsledku změny technologii jako celku, je snadnější přenést model a spravovat jej v jiném nástroji, než přepsat celý kód. Model umožňuje rychle reagovat na vnější faktory, bez použití velkého úsilí pro překódovaní systému, protože ve většině případů model a jeho zařízení zůstávají beze změny bez ohledu na technologii, s jakou pracují.

5.3.3 Rozvoj informačního systému

MDD nabízí model nezávislý na platformě, díky kterému je systém přenosný a umožňuje vám kdykoli vytvořit PSM specifickou pro platformu. Kromě toho je možné zajistit kompatibilitu mezi platformami díky mostům generovaným mezi PSM, které jsou převedeny z jednoho PIM. Schopnost jednoduše generovat model znovu v novém jazyce.

MDD umožňuje použití částí systému v nových projektech, které začnou šetřit čas na vývoj a testování systému.

Závěr

Pro výzkum byly vybrány tři nástroje, které automatizují vývoj řízený modely a jak již bylo napsáno v textu teto práce výše, podporují generování kódu z modelů, které v nich byly vytvořené: GenMyModel, Umbrello a Enterprise Architect. Na základě praktického použití těchto tří nástrojů prostřednictvím procesů modelování diagramů, generování kódu a dokumentace a hodnocení kompletnosti aplikace byly vyvozené závěry o jejich funkcionalitách. GenMyModel umožňuje intuitivní ovládání, ale má velmi omezený generátor kódu, který není schopen snížit potřebu manuálního programování. Umbrello má nečekané chyby v uživatelském rozhraní, netransformuje vztahy v modelu do vztahů v kódu korektně, ale nabízí free licence a následně možnost libovolných úprav zdrojového kódu podle požadavků uživatele. Enterprise Architect má rozsáhlou funkcionalitu a generuje z modelů kvalitní kompletní zdrojový kód, přičemž vyžaduje čas pro dobré ovládnutí funkcionalit.

Jako příklad modelovaného systému byl použit open source redakční systém WordPress zaměřený na vytváření vlastních webových stránek a následnou jejich správu. Nakreslené diagramy WordPress jsou vytvořené pomocí reverzního inženýrství a dokumentace popisující strukturu jádra a základní procesy, které probíhají v systému. Protože WordPress je rozsáhlý systém se širokou funkcionalitou, v praxi diagramy mohou pomoct novým uživatelům WordPress pochopit základy fungování systému.

Ve výsledku je možné doporučit pro vývoj jednoduchých systémů, kde lze jasně vydefinovat funkční požadavky na systém, použít tradiční přístup, protože manuální psaní kódu bude trvat míň než modelování, generování kódu a provedení oprav výsledného kódu. Provedená analýza taktéž odhalila sadu problémů při užívání jednotlivých metod, s kterými se může uživatel setkat, ale vymezení a následně její řešení chybí v odpovídající dokumentaci. Z čeho plyne, že pro efektivní využívání nástrojů je nezbytně nutná zkušenost. Kromě toho, jestli tým neumí žádný jazyk modelování ani nástroj, který podporuje vývoj řízený modely, školení zaměstnanců se vyplatí pouze v případě, jestli vývoj rozsáhlým informačních systémů je každodenním úkolem společnosti,

nikoliv jednorázovou záležitostí.

Vzhledem k výsledkům porovnání vývoje pomocí vývoje řízeného modely s tradičním vývojem software prostřednictvím odhadnutí času nutným k přípravě tvorby, modelování a vytvářením kódu informačního systému vývoj řízený modely ušetří čas a zefektivní vymezené procesy pro rozsáhlé informační systémy. Použití nástrojů vývoje řízeného modely kardinálně zlepšilo vývojový výkon a usnadnilo fáze manuálního programování. Kromě toho, pomocí prozkoumání základních parametrů a funkcionalit nástrojů vývoji řízeného modely bylo odhaleno, že vývoj řízený modely zlepšuje komunikace se zákazníkem a ve velkém týmu, zvětšuje kvalitu kódu, pomáhá pravidelně kontrolovat správnost procesu vývoje a poskytuje prostor pro další rozvoj systému efektivněji než tradiční přístup.

Literatura

[1] Fakhroutdinov, K.: The Unified Modeling Language. [online], accessed on 2021-05-09. Dostupné z: https://www.uml-diagrams.org/

[2] Automattic Inc.: Database Description. [online], accessed on 2021-05-09. Dostupné z: https://codex.wordpress.org/Database_Description

[3] Object Management Group: Common Warehouse Metamodel. [online], 2003, accessed on 2021-05-09. Dostupné z: https://www.omg.org/spec/CWM/About-CWM/#document-metadata

[4] T. Stahl, S. E. A. H., M. Völter: *Modellgetriebene Softwareentwicklung : Techniken, Engineering, Management*. Heidelberg, Dpunkt-Verlag: BibSonomy, 2007, ISBN 978-3-89864-448-8.

[5] Frankel, D.: *Model Driven Architecture: Applying MDA to Enterprise Computing*. New York: John Wiley Sons, 2002, ISBN 978-0-471-31920-7.

[6] Object Management Group: MDA specifications. [online], accessed on 2021-05-09. Dostupné z: https://www.omg.org/mda/specs.htm

[7] Kleppe A, B. W., Warmer J: *MDA Explained: The Model Driven Architecture: Practice and Promise*. Arlington Street, Suite 300 Boston, MAUnited States: Pearson, 2003, ISBN 978-0-321-19442-8.

[8] Object Management Group: MDA Guide Version 2.0. [online], 2014, accessed on 2021-05-09. Dostupné z: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01

[9] Gennadievich, G. K.: *Delphi Model Driven Architecture.* Разработка приложений баз данных. St. Petersburg: Peter, 2004, ISBN 5-469-00185-7.

[10] Object Management Group: Model Driven Architecture. [online], accessed on 2021-05-09. Dostupné z: https://www.omg.org/mda/

[11] Henning, M.: The Rise and Fall of CORBA. *ACM Queue*, ročník 4, č. 5, 2006, accessed on 2021-05-08. Dostupné z: https://doi.org/10.1145/1142031.1142044

[12] Object Management Group: MDA Vendor Directory Listing. [online], accessed on 2021-05-09. Dostupné z: https://www.omg.org/mda-directory/vendor/list.htm

[13] Richard Soley and the OMG Staff Strategy Group: Model Driven Architecture. [online], 2000, accessed on 2021-05-09. Dostupné z: https://www.omg.org/~soley/mda.html

[14] Object Management Group: MDA Guide Version 1.0. [online], 2003, accessed on 2021-05-09. Dostupné z: https://www.omg.org/mda/mda_files/MDA_Guide_Version1-0

[15] Object Management Group: Mission Vision. [online], accessed on 2021-05-09. Dostupné z: https://www.omg.org/about/index.htm

[16] S. Mellor, T. F., A. Clark: Model-driven development. *IEEE Software*, ročník 20, č. 5, 2006, ISSN 1937-4194. Dostupné z: https://ieeexplore.ieee.org/document/1231145

[17] Selic, B.: The pragmatics of model-driven development. *IEEE Software*, ročník 20, č. 5, 2003, ISSN 1937-4194. Dostupné z: https://ieeexplore.ieee.org/document/1231146

[18] Object Management Group: Technology standards by industry. [online], accessed on 2021-05-09. Dostupné z: https://www.omg.org/industries/index.htm

[19] Object Management Group: OMG Meta Object Facility (MOF) Core Specification Version 2.5.1. [online], 2019, accessed on 2021-05-09. Dostupné z: https://www.omg.org/spec/MOF/2.5.1

[20] Object Management Group: Common Warehouse Metamodel (CWM) Specification Version 1.1. [online], 2003, accessed on 2021-05-09. Dostupné z: https://www.omg.org/spec/CWM/1.1/PDF

[21] Object Management Group: XML Metadata Interchange (XMI) Specification. [online], 2015, accessed on 2021-05-09. Dostupné z: https://www.omg.org/spec/XMI/2.5.1/PDF

[22] Fowler, M.: *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition*. Addison-Wesley Professional, 2004, ISBN 0321193687.

[23] Object Management Group: The Unified Modeling Language specification 1.1. [online], 1997, accessed on 2021-05-09. Dostupné z: https://www.omg.org/spec/UML/1.1

[24] Object Management Group: The Unified Modeling Language specification 1.2. [online], 1999, accessed on 2021-05-09. Dostupné z: https://www.omg.org/spec/UML/1.2

[25] Object Management Group: The Unified Modeling Language spefication 2.5.1. [online], 2017, accessed on 2021-05-09. Dostupné z: https://www.omg.org/spec/UML/2.5.1

[26] Grady Booch, I. J., James Rambeau: *The Unified Modeling Language Usere Guide*. DMK-Press, 2006, ISBN 5-94074-334-X.

[27] Ambler, S. W.: *The Elements of UML 2.0*. Cambridge University Press, 2005, ISBN 978-0521616782.

[28] Grady Booch, I. J. B., James Rumbaugh: *The Unified Modeling Language User Guide*. Addison-Wesley Professional, druhé vydání, 2005, ISBN 978-0521616782.

[29] Babich, A. V.: *Uml:* первое знакомство. Пособие для подготовки к сдаче теста *UML-100*. Moscow: БИНОМ. Лаборатория знаний (BINOM. Knowledge laboratory), 2008, ISBN 978-5-94774-878-9.

[30] Russ Miles, K. H.: *Learning UML 2.0*. O'Reilly Media, 2006, ISBN 978-0-59-600982-3.

[31] Drusinsky, D.: *Modeling and Verification Using UML Statecharts*. Newnes, 2005, ISBN 9780080481470.

[32] Dorodnykh N.O., Y. A.: Использование диаграмм классов UML для формирования продукционных баз знаний. Программная инженерия, , č. 4, 2015, ISSN 2220-3397, accessed on 2021-05-09. Dostupné z: http://novtex.ru/prin/rus/10.17587/prin._4_2015_1.html

[33] Object Management Group: QVT specification Version 1.2. [online], 2015, accessed on 2021-05-09. Dostupné z: https://www.omg.org/spec/QVT/1.2

[34] Tratt L.: Model transformations and tool integration. *SpringerLink*, 2005, ISSN 112–122, acceseed on 2021-05-09. Dostupné z: https://link.springer.com/article/10.1007/s10270-004-0070-1

[35] Object Management Group: MDA FAQ. [online], accessed on 2021-05-09. Dostupné z: https://www.omg.org/mda/faq_mda.html

[36] Automattic Inc.: WordPress Codex. [online], 2003, accessed on 2021-05-09. Dostupné z: https://codex.wordpress.org/

[37] Automattic Inc.: Zdrojový kód WordPress. [online], accessed on 2021-05-09. Dostupné z: https://github.com/WordPress

[38] Automattic Inc.: WordPress. [online], 2003, accessed on 2021-05-09. Dostupné z: https://wordpress.org/

[39] Scott, A. D.: *WordPress for Education*. Birmingham: Packt Publishing Ltd, 2012, ISBN 978-1-84951-820-8.

[40] Awesome Motive: Beginner's guide for WordPress. [online], 2009, accessed on 2021-05-09. Dostupné z: https://www.wpbeginner.com/

[41] Staff, E.: How WordPress Actually Works Behind the Scenes. [online], 2020, accessed on 2021-05-09. Dostupné z: https://www.wpbeginner.com/wp-tutorials/how-wordpress-actually-works-behind-the-scenes-infographic

[42] GenMyModel Team: GenMyModel. online, accessed on 2021-05-09. Dostupné z: https://www.genmymodel.com

[43] Umbrello Team: UmbrelloProject. [online], accessed on 2021-05-09. Dostupné z: https://umbrello.kde.org/

[44] Sparx Systems Pty Ltd: Enterprise Architect Version 15.2. [online], 2021, accessed on 2021-05-09. Dostupné z: https://sparxsystems.com/products/ea/

[45] Sparx Systems Pty Ltd: Enterprise Architect 15.2 User Guide. [online], 2021, accessed on 2021-05-09. Dostupné z: https://sparxsystems.com/enterprise_architect_user_guide/15.2/index/index.html

[46] Alena Buchalcevová, M. , Iva Stanovská: *Základy softwarového inženýrství - základní témata*. Praha: Oeconomica, 2002, ISBN 8024503468.

[47] ICT Pro s.r.o.: Základy UML modelování v prostředí Enterpise Architect (UML1). [online], accessed on 2021-05-09. Dostupné z: https://www.skoleni-ict.cz/kurz/Zaklady-UML-modelovani-v-prostredi-Enterpise-Architect-UML1.aspx?

Příloha **A**

Seznam použitých zkratek

API Application programming interface

BPEL Business Process Execution Language

BPMN Business Process Management Notation

CASE Computer-Aided Software / System Engineering

CIM Common Information Model

CIV Computation Independent Viewpoint

CMMN Case Management Model and Notation

CMS Content Management System

CORBA Common Object Request Broker Architecture

CVUT České vysoké učení technické

CWM Common Warehouse Metamodel

DDS Data Design System

DTD Document Type Definition

EA Enterprise Architect

EMF Eclipse Modeling Framework

ER Entity-Relationship

GML Geography Markup Language

GPL General Public License

ICT Information and Communication Technologies

IDL Interface Definition Language

IFML Interaction Flow Modeling Language

IT Information technology

JPA Java Persistence API

MDA Model Driven Architecture

MDD Model-Driven Development

MOF Meta Object Facility

ODM Original Design Manufacturer

OLAP On-Line Analytical Processing

OMA Office for Metropolitan Architecture

OMG Object Management Group

OMT Object Modeling Technique

OOP Object Oriented Programming

OOSE Object-Oriented Software Engineering

OS Operating system

OWL Web Ontology Language

PDF Portable Document Format

PIM Protocol Independent Multicast

PIV Platform Independent Viewpoint

PSM Persistent Stored Modules

PSV Platform Specific Viewpoint

RDF Resource Description Framework

RDS Radio Data System

RFP Request for proposal

SPEM Software Process Engineering Metamodel

SoaML Service-oriented architecture Modeling Language

SysML The Systems Modeling Language

UML Unified Modeling Language

VDML Value Delivery Modelling Language

WP WordPress

WSDL Web Services Description Language

XHTML Extensible Hypertext Markup Language

XMI XML Metadata Interchange

XML Xtensible Markup Language

XSD XML Schema Definition Language

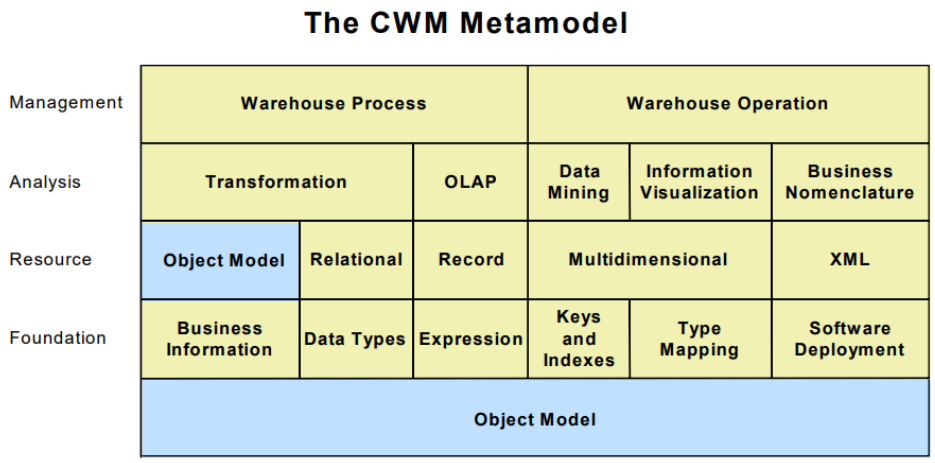
Příloha **B**

Obsah přiloženého CD

```
readme.txt ..... stručný popis obsahu CD
analysis/ ..... analyzované metod
    Enterprise Architect/ ..... metoda Enterprise Architect
        code/ ..... zdrojové kódy Enterprise Architect
        EA_wp.eapx ..... projekt v Enterprise Architect ve formatu .eapx
        EA Diagrams.xml ..... model ve formátu .xml
    GenMyModel/ ..... metoda GenMyModel
        code/ ..... zdrojové kódy GenMyModel
        Documentation_wp.pdf ..... dokumentace ve formátu PDF
        GenMyModel_wp.xmi ..... model ve formátu .xmi
    Umbrello/ ..... metoda UMbrello
        code/ ..... zdrojové kódy v Umbrello
        Umbrello_wp.xmi ..... model ve formátu .xmi
thesis/ ..... text práce
    src/ ..... zdrojová forma práce ve formátu LaTeX
    thesis.pdf ..... text práce ve formátu PDF
```

Příloha C

CWM Metamodel



Obrázek C.1: CWM Metamodel [3]