Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

Automatizovaná kontrola UML modelů

Bakalářská práce

Autor: Jan Urbanec
Studijní obor: Aplikovaná informatika - 3

Vedoucí práce: Ing. Pavel Čech, Ph.D.

Hradec Králové Srpen 2017

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 4.9.2017 Jan Urbanec

Poděkování:

Děkuji vedoucímu bakalářské práce Ing. Pavlu Čechovi, Ph.D. za metodické vedení, konzultace, trpělivost, rady a také čas, který mi věnoval.

Anotace

Tato bakalářská práce se zabývá pravidly pro kontrolu analytického a návrhového diagramu tříd v grafickém jazyce UML (Unified Modeling Language). Práce popisuje podobu a pravidla jednotlivých prvků diagramů tříd zkoumaných pomocí odborné literatury a internetu. Na základě analýzy všech prvků, které se používají pro tvorbu analytického a návrhového diagramu tříd, je cílem práce vytvořit aplikaci, která dokáže zkontrolovat tyto dva typy diagramů ze souboru s UML modely. Ty jsou vytvořeny pomocí programu Enterprise Architect od společnosti Sparx Systems. Práce popisuje postup tvorby a podobu aplikace pro kontrolu UML společně s charakteristikou technologií, které pro svoji funkčnost využívá nebo které byly použity k jejímu vytvoření. Aplikace, která je výstupem práce, je napsána v jazyku C# a bude dostupná společně s prací.

Klíčová slova: kontrola modelů, UML, Enteprise Architect, diagram tříd, analytický model tříd, návrhový model tříd, aplikace

Annotation

Title: Automated Checking of UML Models

This bachelor thesis deals with rules for inspection of domain and design class diagram in graphic language UML (Unified Modeling Language). The thesis describes a shape and rules of particular components of class diagrams surveyed with the aid of professional literature and the Internet. The aim of the thesis is to create, based on an analysis of all elements used for creation of domain and design class diagram of categories, an application which is able to check those two types of diagrams from a file with UML models. They are created with the aid of the program Enterprise Architect of the Sparx Systems company. The thesis describes the course of creation and the shape of the application for inspection of the UML together with the characteristic technology that is used for its functionality or technologies that were used for its creation. The aplication is the result of this thesis and is written in C# language. The application will be available within the thesis.

Key words: inspection of models, UML, Enterpsie Architect, class diagram, design class diagram, domain class model

Obsah

Seznam zdrojových kódů

Seznam obrázků

1 Úvod

S automatickou opravou se člověk v době moderních počítačů setkává každodenně už několik let. Těžko by se hledal textový procesor a mobilní zařízení, které by při psaní textu neopravovaly pravopisné chyby. Podobných oprav se v dnešních zařízeních vyskytuje mnoho druhů. Ty napravují lidskou chybu, aniž bychom to my brali na vědomí. V případech, kdy člověk produkuje velké množství informací a pracuje s nimi, není možné, aby lidský faktor dosahoval stoprocentní kontroly, takové, jaké dokáže stroj. Navíc člověk není schopen vidět svýma očima všechny informace najednou, a právě proto zde máme počítače, které můžou kontrolu provádět za nás. Je jim jen potřeba správně interpretovat náš vizuální pohled na danou věc a pak jim říct, ve kterých případech se nachází vše v pořádku a kdy ne.

Počítač může kontrolovat i složité modely. Mezi takové vizuální modely patří Unified Modeling Language neboli zkrácené UML. UML je vizuální jazyk pro specifikaci, konstrukci a dokumentaci částí systému [1]. Nejčastěji je spojován s modelováním objektově orientovaných softwarových systémů, má ale mnohem širší využití, což vyplývá z jeho zabudovaných mechanizmů. Na vývoji jazyka UML se začalo pracovat roku 1994, i když se tehdy ještě nevědělo, jak by mělo UML vypadat. Pouze se hledal způsob, jak sjednotit metodiky pro objektové modelování. Začalo to právě tehdy, když se Grady Booch, který disponoval metodou Booch a Jim Rumbaugh, který měl metodu OMT (Object Modeling Technique), spojili ve firmě Rational Software, která pracovala na vývoji UML. V roce 1996 navrhlo OMG (Objective Management Group) specifikaci RFP (Request For Proposal) pro objektově orientovaný jazyk pro vizuální modelování, v němž jako standart navrhlo jazyk UML. V roce 1997 sdružení OMG jazyk UML přijalo, čímž vznikl první průmyslový standart objektově orientovaného jazyka. Od roku 2001 připravuje sdružení OMG druhou verzi jazyka UML, jejíž první části byly schváleny v roce 2004. Právě ve druhé verzi je jazyk UML na úrovni velice vyspělého modelovacího jazyka. Potvrzuje to tisíce softwarových projektů, které na celém světě byly vytvořeny [2]. Poslední vydaná verze 2.5 je z roku 2015.

UML může být modelováno v jakémkoliv kreslícím programu na počítači nebo případně na papír, ale nejvíce je využit potenciál tohoto jazyka, když je modelován pomocí nástroje CASE (Computer-aided software engineering). Nástroj CASE je

software, který nám slouží pro analýzu a návrh počítačového programu za účelem dosáhnutí vyšší kvality, bezchybnosti, udržovatelnosti. Jazyk UML je explicitně navržený tak, aby jej implementoval nástroj CASE, protože nabízí propojení jednotlivých technik UML. Mezi CASE nástroje patří i Enterprise Architect vyvíjený společností Sparx Systems. Enterprise Architect nabízí modelovací nástroje pro analýzu a návrh po celou dobu softwarového vývoje [3]. Neposkytuje ale opravu modelů podle syntaxe UML.

Pokud si tedy člověk chyby při modelování nevšimne a teprve až při vývoji je tato chyba nalezena, může oprava takového modelu značně prodloužit vývoj softwaru. Autor této práce chce vytvořit program, který bude umět podle pravidel UML zkontrolovat model vytvořený pomocí Enterprise Architect a případně jej opravit.

2 Cíl práce

Cílem autorovi práce je vytvořit aplikaci, která zkontroluje UML modely po načtení souboru EAP obsahující modely ve formě dat, které byly vytvořeny v programu Enterprise Architect. K testování bude použito Enterprise Architect verze 7.1.833 a 13.0.1304, které autorovi poskytuje škola. Jelikož jazyk UML disponuje mnoha diagramy, autor práce si pro vypracování vybral diagram tříd a ten podrobně zanalyzuje, aby zjistil veškerá pravidla pro jeho správnost. UML sice není vázáný na žádnou metodiku, má ale svojí syntaxi [2], podle které bude aplikace opravovat model. Ke kontrole použije autor pravidla, které jsou normovány v UML verze 2.0, ta je velice rozšířena a podporována již zmíněnou verzí Enterprise Architect. Déle je mnoho pravidel, které se podle určitých konvencí při analýze a návrhu softwaru v UML modelech vyskytují.

Autor práce chce tato pravidla shromáždit a využít pro vylepšení aplikace, která bude opravovat syntaxi jazyka. Aplikace tak může nenásilně nabídnout vylepšení modelů pravidly, která jsou mezi vývojáři používána. To pak může vést k srozumitelnější spolupráci mezi kolegy nebo lepší komunikaci se zákazníkem. Dále je potřeba, aby se autor naučil pracovat s knihovnami, které umí číst soubory EAP, a to pomocí ukázkových zdrojových kódů psaných v jazyce C#, které společnost Sparx Systems přikládá k jejich programu. Celá aplikace pak bude psána také v jazyce C#.

3 Analýza UML modelů

Autor této práce v následujícím textu bude zkoumat a hledat pravidla a běžně uznávané konvence UML. Nebude se ale v analýze zaobírat pouze pravidly, která byla stanovena standardizační organizací OMG, ale bude brát v potaz běžně používané modelovací vzory a metodiky popisující, jak by měl vývoj softwaru za pomocí UML vypadat. Je ale možné, že něco ze své analýzy autor vypustí. Cílem je totiž najít prvky, které mu budou užitečné pro tvorbu programu, jenž bude kontrovat vytvořený model.

3.1 Diagram tříd

Základním elementem diagramu tříd, jak už jeho název napovídá, je třída. Třída je také základním prvkem objektově orientovaného programování. Je tedy pravděpodobné, že v případě, když by se člověk někdy setkal s UML nebo vývojem nějakého informačního systému, tak by se setkal právě s diagramem tříd. Pomocí tohoto typu digramu je možné popsat třídy a vztahy mezi nimi [6].

Podle metodiky Agilního modelování se rozděluje modelování diagramu tříd na dvě časti. V českém jazyce můžeme tyto části nazvat jako analytický model tříd a návrhový model tříd. V angličtině se pak setkáme s názvy jako „domain model“ nebo „conceptual class diagram“ a „design class diagram“. To ale stále nejsou všechny způsoby, jak jsou tyto dvě části nazývány. Účel zůstává stejný. Stručně lze říci, že jde vlastně o to, abychom v první části získali nějaký velice abstraktní pohled na modelovaný systém. Tato první část nám pak slouží jako zdroj inspirace pro druhou část, která modeluje přesnou strukturu systému, kde je potřeba i zohlednit například jakým programovacím jazykem bude informační systém tvořen.

3.1.1 Analytický model tříd

Slouží pro zachycení tříd reálného prostředí, ne pro ty, které jsou softwarové [7]. To znamená, že se v modelu neobjeví takové třídy, které souvisí například s databázemi nebo takové, co představují komponenty v grafickém rozhraní, ale takové třídy, které jsou z okolí systému nebo na něj nějak působí. Příkladem takové třídy, který najdeme v analytickém modelu pro informační systém letiště, může být třeba letadlo nebo letenka. Dále je pro tento model důležité najít, mezi kterými třídami je vztah pomocí vazby asociace a taky definovat prvotní atributy třídy.

3.1.2 Návrhový model tříd

Tento druh modelu tříd vychází z předchozího analytického modelu tříd. Důležité je, že jej rozšiřuje do mnohem větších podrobností. Při modelování tohoto modelu je kladen důraz hlavně na softwarovou část, jelikož analýza tříd z reálného prostředí je hotova už z analytického modelu tříd, a protože je modelována už skutečná struktura nějakého systému. Třídy takového modelu by měly být specifikovány na takovém stupni, že je lze implementovat [2].

3.1.3 Třídy

Třída je základem objektového systému. Jedná se o deskriptor pro množinu objektů se stejnými charakteristickými vlastnostmi. Třída pak vytváří instance (objekty), které mají svou jedinečnost. Každá třída je definována názvem, a i když UML nenařizuje žádnou konvenci pojmenování tříd, aplikuje se konvence používaná téměř všemi. Název začíná velkým písmenem. Pokud obsahuje více slov, píší se za sebou bez mezer a počáteční písmeno každého slova je velké, tzv. CamelCase. Dále je vhodné se vyhnout speciálním znakům [3]. Vztahy mezi třídami jsou znázorněny pomocí relací. Třídy obsahují další parametry, které pomohou k jejich definování, jako například atributy, metody a stereotypy, které budou popsány společně s relacemi níže.

3.1.3.1 Kolekce

Kolekce je třída, jejíž instance se specializují na správu kolekcí jiných objektů [2]. Je možné jí nalézt sice až v návrhovém modelu tříd, ale vychází většinou z analytického modelu tříd, kde je reprezentována asociací s multiplicitou 1:M. V návrhovém modelu je pak modelována jako třída mezi dvěma jinými, spojená s nimi pomocí relace agregace.

3.1.4 Rozhraní

Rozhraní je třída, která nemá žádnou implementaci, to znamená, že všechny její vlastnosti jsou abstraktní [4]. Pokud nějaká třída implementuje rozhraní, zavazuje k tomu, že bude poskytovat metody, které rozhraní předepisuje. Rozhraní v UML vypadá stejně jako třída, jen je rozlišeno stereotypem interface.

3.1.5 Atributy

Účelem atributů je uchovávat nějaké informace neboli stav objektu, tudíž je potřeba, aby měly název, který je podobně jako v případě názvu tříd CamelCase, ale tentokrát začíná malým písmenem. Dále je důležité určit typ atributu, aby bylo patrné, zda bude uchovávat řetězec, číslo či jiný datový typ. Atribut může mít ještě svoji počáteční hodnotu, která se nastaví v okamžiku vytvoření instance. V poslední řadě je potřeba nastavit viditelnost, která ukazuje, kdo bude moci přistupovat k atributům. V UML existují čtyři typy viditelnosti atributů, a to public, private, protected a package. Podrobnější specifikaci viditelnosti v jazyce UML najdete v této práci dále.

3.1.6 Metody

Metoda má za úkol dynamicky měnit stav objektu. Vlastníkem metody je třída, která ji definuje a objekt, který je instancí třídy, pak může tuto metodu volat. Charakteristiku metody vystihuje vždy její pojmenování. Název metody by měl jednoznačně vystihovat, co metoda vykonává. Stejně jako v případě atributů je název operace psán bez mezer, způsobem CamelCase a první slovo začíná malým písmenem.

Každá metoda má definovaný nějaký návratový typ. To znamená, že třída nám podle tohoto typu musí vrátit nějakou hodnotu. Může se ale také jednat o tzv. návratový typ void, kdy nedostáváme žádnou návratovou hodnotu metody. Metodě můžeme také předat seznam argumentů. Jedná se o proměnné, které potřebuje metoda znát, aby mohla být provedena. Každému argumentu musíme nastavit datový typ a můžeme mu nastavit implicitní hodnotu, která bude použita v případě, že mu žádná hodnota při volání metody nebude předána.

3.1.7 Stereotypy

Pomocí stereotypu můžeme rozšířit třídu o vlastnosti, které obsahuje element podobný třídě [1]. Příkladem může být rozhraní, které se často v objektových systémech vyskytuje. Je mnoho způsobů, jak značit v UML stereotyp, jakým je například speciální symbol, který ukazuje, o jaký stereotyp se jedná [2]. Nejčastějším způsobem značení stereotypu je i takový, který implementuje Enterprise Architect. A to tak, že se název stereotypu zapíše do dvojitých lomených závorek, například <<interface>>. Takto ho zpravidla najdeme nad názvem třídy.

3.1.8 Viditelnost

Díky viditelnosti můžeme určit, jaké elementy objektu budou viditelné jeho okolním objektům. Existují dva základní způsoby viditelnosti, public a private, neboli veřejné a soukromé [4]. Tyto dva způsoby ale nejsou jediné. Na tom, jaké použijeme, závisí, v jakém jazyce bude software programován. V UML se můžeme často setkat s dalšími dvěma druhy viditelností. Protected, kde přístup k elementu nemá jen třída samotná, ale i její potomci.

V případě, že programovací jazyk využívá strukturu balíčků, můžeme využít i viditelnost package. Ta nám zaručuje, že elementy budou využívat jen třídy, které jsou z téhož balíčku jako příslušná třída, ale i prvky z vnořených dílčích balíčků [2]. Všechny tyto čtyři mají v UML speciální symbol, ten je pro public +, private -, protected #, package ~.

3.1.9 Signály

Signál zastupuje informaci předávanou asynchronně mezi objekty. Předávané informace jsou neseny v atributech signálu. V analýze používáme signál k zobrazení odeslání a příjmu asynchronních obchodních událostí. V návrhu je použijeme ke znázornění asynchronní komunikace mezi systémy, subsystémy nebo jednotlivými částmi hardwaru. Signál je modelován jako třída označená stereotypem <<signal>> [2].

3.1.10 Balíčky

Balíčky nebo také anglicky package jsou používány v případě, že software, který modelujeme, bude psán v jazyce, který balíčkovou strukturu implementuje. Balíček představuje definici zapouzdřeného jmenného prostoru, v němž musí být všechny názvy jedinečné. Znamená to rovněž, že když se prvek v jednom jmenném prostoru rozhodne použít prvek v jiném prostoru, musí určit nejen jeho název, ale rovněž způsob navigace mezi vnořenými jmennými prostory volaného balíčku [2]. Balíčky využijeme spíše v diagramu balíčků, ale UML nám dovoluje je použít i v diagramu tříd. V diagramu tříd tak můžeme už předem znázornit v jakém balíčku bude jaká třída, i když je to vhodnější udělat to až v diagramu balíčků.

3.1.11 Relace

Relace umožňuje ukázat, jaký je vztah mezi dvěma předměty [2]. V UML existuje mnoho druhů relací. Ty, které se vyskytují v diagramu tříd, budou popsány v této práci. Všechny relace, kromě generalizace/specializace, nesting a realizace, by měly správně obsahovat multiplicitu.

3.1.11.1 Multiplicita

Pomocí multiplicity znázorňujeme, kolik objektů jedné třídy má referenci na objekty druhé třídy. Můžeme pak namodelovat, které atributy budou obsahovat kolekce a které budou moci obsahovat prázdnou hodnotou [2]. Znázornění multiplicity najdeme na obou koncích relace nebo u atributů.

3.1.11.2 Asociace

Asociace znázorňuje vztah mezi jednou či více třídami, které jsou abstrakcí množiny spojení mezi instancemi těchto tříd [5]. Předpokládá se, že tok informací mezi třídami bude obousměrný, pokud to tedy není explicitně specifikovaný směr [2]. Asociaci mezi třídami můžeme pojmenovat, tím vyjádříme akci, kterou zdrojový objekt vykonává pomocí cílového objektu. To, který objekt zdrojové třídy řídí objekt cílové třídy, znázorňujeme pomocí průchodnosti [2]. Průchodnost znázorňujeme šipkou na jednom z konců asociace nebo, je-li to potřeba, i na obou. Dále je možné na jednom nebo obou koncích asociace udělit název roli, kterou hraje příslušná třída ve vztahu k té druhé [2]. Každá asociace by měla obsahovat název role, multiplicitu a průchodnost [7]. Znázorňujeme ji pomocí plné čáry. Pomocí symbolu nevyplněného kosočtverce můžeme asociaci rozvětvit mezi více než dvě třídy [3].

3.1.11.3 Agregace

Agregace je vztah, který říká, že jedna třída je součástí jiné třídy. Předpokládá se, že nadřízený objekt využívá dovednosti podřízeného objektu a měl by nést zodpovědnost za jeho vznik a zánik [5]. Agregace je znázorněna plnou čárou a symbolem nevyplněného kosočtverce, který sousedí s nadřazenou třídou.

3.1.11.4 Kompozice

Kompozice je silnější druh agregace, kdy nadřízený objekt nemůže existovat bez podřízeného [5]. Kompozice je znázorněna podobně jako agregace, pouze s tím rozdílem, že zde je kosočtverec vyplněný.

3.1.11.5 Generalizace/specializace

Pomocí generalizace nebo opačného vztahu specializace můžeme znázornit důležitou vlastnost objektově orientovaných jazyků, a tím je dědičnost. Z tříd spojených touto relací se stávají předci a potomci. Potomek získává všechny vlastnosti od předka a doplňuje je o vlastní. V případě, kdy předek má více potomků, zděděná operace, kterou mají potomci od předka, může mít u každého potomka jiný průběh [2]. Generalizace a specializace se znázorňuje pomocí plné čáry, která má na jednom konci nevyplněný trojúhelník (šipku), ten ukazuje na předka třídy.

3.1.11.6 Realizace

Je to relace, která slouží pro znázornění vztahu mezi třídou a rozhraním. Znázorňujeme jí přerušovanou čárou, která má na jednom konci symbol nevyplněného trojúhelníku (šipky), ten směřuje na rozhraní.

3.1.11.7 Nesting

Jedná se o relaci, která poukazuje na to, že definice jedné třídy je uvnitř definice třídy jiné. Tímto způsobem vznikají vnořené třídy známé také jako vnitřní třídy [2]. Tento druh relace znázorňujeme plnou čarou a symbolem kružnice, který obsahuje kříž z přímek tak, že jedna je svislá a druhá vodorovná. Tento symbol sousedí vždy s třídou, která obsahuje vnořenou třídu.

3.1.11.8 Závislost

Závislost je relace mezi dvěma prvky, v níž se změna jednoho prvku (dodavatele) promítá do druhého prvku (klient). Znamená to, že klient závisí do určité míry na dodavateli. Závislost se požívá při modelování relací mezi klasifikátory, kdy jeden klasifikátor je na druhém určitým způsobem závislý, ale relace není fakticky asociací. Při závislosti je předpokládáno, že je předán objekt jedné třídy jako argument metodě objektu jiné třídy [2]. Závislost znázorňujeme pomocí přerušované čáry s šipkou ve směru závislosti. Dále je u relace znázorněn typ závislosti, a to pomocí stereotypu. Pokud závislost neobsahuje stereotyp, jedná se o typ <<use>>. Závislostí může být několik typů, ty budou v této práci ještě popsány.

3.1.12 Asociační třída

Asociační třída nám pomáhá vyřešit problém který nastává, když máme relaci M objektů jedné třídy k N objektů druhé třídy. Existují totiž atributy a metody, které spolu souvisí a ovlivňují konkrétní vztah mezi objekty třídy, toto spojení je vyjádřeno instancí asociační třídy. Jedinečná identita je určována výhradně identitami objektů na obou koncích. Proto asociační třídu lze použít pouze tehdy, je-li mezi dvěma objekty v určitém okamžiku jedno jedinečné spojení [2]. Asociační třída se značí stejně jako třída, pouze její relace, která je přerušovaná, míří na vztah asociace s multiplicitou M:N mezi dvěma třídami.

4 Reprezentace UML modelů

Proto, aby bylo možné uložit informace o modelu, je potřeba vymyslet reprezentaci, jakým způsobem budou data v aplikaci uspořádána. V případě této práce je potřeba takové formy reprezentace, která dokáže pojmout diagram tříd ze souboru ve formátu EAP, který byl vytvořený aplikací Enterprise Architect. V takovém souboru je ale možné uložit několik diagramů, je tedy nutné nejdříve nalézt ty správné, a i v tomto případě se dá předpokládat, že v souboru bude více modelů.

Všechny tyto modely se musí se uložit do nějakého seznamu. Ten se nám pak bude hodit hlavně proto, že jednotlivé diagramy mohou obsahovat referenci na jiný, zejména v případě návrhového a analytického diagramu tříd. Je důležité, aby aplikace sama dokázala poznat tyto dva druhy diagramu tříd, což by nemělo být složité, protože program Enterprise Architect nabízí možnost vytvoření dvou oddělených diagramů, a to domain model a class model. Může ale nastat situace, kdy uživatel nevyužije této možnosti, která mu je nabízena a použije v obou případech pouze class model, protože nástroje jsou v obou stejné. Pak je tedy potřeba, aby aplikace rozeznala modely od sebe, například podle obecně používaných názvů pro tyto modely. Jestliže nebude možné ani přesto určit druh modelu, může je uživatel aplikace sám nastavit, případně změnit, když aplikace rozezná model špatně.

Když už jsou jednotlivé modely v seznamu, uložíme i elementy, které obsahují. Pro tuto část se hodí vlastnosti orientovaného grafu, a to v reprezentaci matice sousedností. Tato reprezentace se hodí hlavně v případě, kdy je potřeba u každé hrany vyjádřit její směr [8]. Ten je určen pomocí hodnoty v matici. Tento způsob bude ale lepší upravit, aby se lépe hodil k implementaci v aplikaci. Místo toho, aby se do matice ukládala hodnota, která určuje směr, bude zde kolekce relací mezi dvěma určitými elementy v modelu. Takový objekt pak sám může obsahovat informace o tom, jakého je relace typu, nebo o její průchodnosti.

4.1 Použití reprezentace při programování

K tomu, aby bylo možné uložit reprezentace, je potřeba zvolit vhodné datové typy. Jelikož aplikace bude psána v jazyce C#, volí takové datové typy, které jsou jazykem podporovány. Nejprve uložíme seznam se všemi modely tříd. Aby mohl být uložen do nějakého seznamu, musí se informace o modelu uložit do nějakého objektu. Proto je nutné, aby ve struktuře aplikace byla třída, která si tyto informace bude umět zapamatovat. Takovou třídu si autor práce může vytvořit v jazyce sám podle potřeb.

Při vzniku objektu ho při inicializaci naplníme v případě této aplikace elementy, které model obsahuje. Je tedy potřeba uložit elementy jako třídy a relace. Nejprve třídy budou uloženy jako objekt a podobně jako v případě modelu budou obsahovat potřebné informace o třídě. Aby aplikace měla informace o všech třídách v modelu, je nutné je uložit do seznamu. Pro tento seznam se hodí generický datový typ List, který jazyk podporuje. Jelikož je generický, znamená to, že nám dovoluje uložit objekt takového typu, který si pro seznam zvolíme. V případě této aplikace je to rozhraní pro třídy. Dále uložíme relace, a ty stejně jako u elementu jsou definovány pomocí nějakého rozhraní a pak ukládány jako instance objektu. Když už je vytvořen objekt, je potřeba jej uložit do nějaké matice sousedností. Jazyk C# umí datový typ dvourozměrného pole, jež dobře poslouží jako matice. Když pak vyvstane potřeba zjistit, zda jsou mezi třídami nějaké relace, bude stačit, když aplikace bude znát pozici zdrojové třídy v seznamu, což bude nějaké n a pozici cílové třídy v seznamu, což bude nějaké m. Když vznikne potřeba dostat kolekci relací mezi těmito třídami, bude stačit zavolat proměnná `_connector[n,m]`. To znamená, že matice, nebo v případě aplikace dvojrozměrné pole bude čtvercové, a tudíž bude mít stejný počet řádků jako sloupců. Pro kolekci relací mezi elementy bude využit generický List.

Objekt modelu je už kompletně definovaný a musíme jej také přidat do seznamu. K tomu poslouží, stejně jako v případě, kdy byly uloženy objekty elementů nebo relace generický datový typ List.

4.1.1 Další operace s modelem

Uložení všech elementů a relací není jediná operace, kterou bude třída zodpovědná za práci s diagramem umět. Dále je také důležité myslet na to, že aplikace by v budoucnu mohla opravovat i další typy diagramů, než jen návrhový nebo analytický diagram tříd. Proto je potřeba mít rozhraní, které bude třída určitého druhy

diagramu implementovat. Toto se pak může hodit v případě, že je nutné uložit více druhů diagramů do nějaké datové struktury. Rozhraní diagramu bude definováno čtyřmi přístupovými metodami: `Name`, `Id`, `Guid`, `Type`. Tyto metody pak budou vracet textový řetězec s názvem, celé číslo s identifikací diagramu, objekt s univerzálním identifikátorem a v poslední řadě typ diagramu. Pro typ diagramu bude vytvořen speciální výčtový typ, který bude určovat jakého typu diagram je. Nyní by se mohlo zdát divné, proč je potřeba dostávat typ pomocí výčtového typu enum, když už pro každý typ diagramu může existovat jiná třída díky rozhraní.

K tomu, aby člověk rozuměl těmto dvěma druhům určování typu, je potřeba pochopit, jak program Enterprise Architect rozděluje typy diagramu. Používá tedy vlastní rozdělení, kde spojuje více typů diagramů pod jeden typ, podle toho, jak spolu mají společné nástroje pro modelování. Jelikož tedy modely tříd, a to jak analytický, tak návrhový, využívají stejné nástroje, jsou oba dva typu Logical.

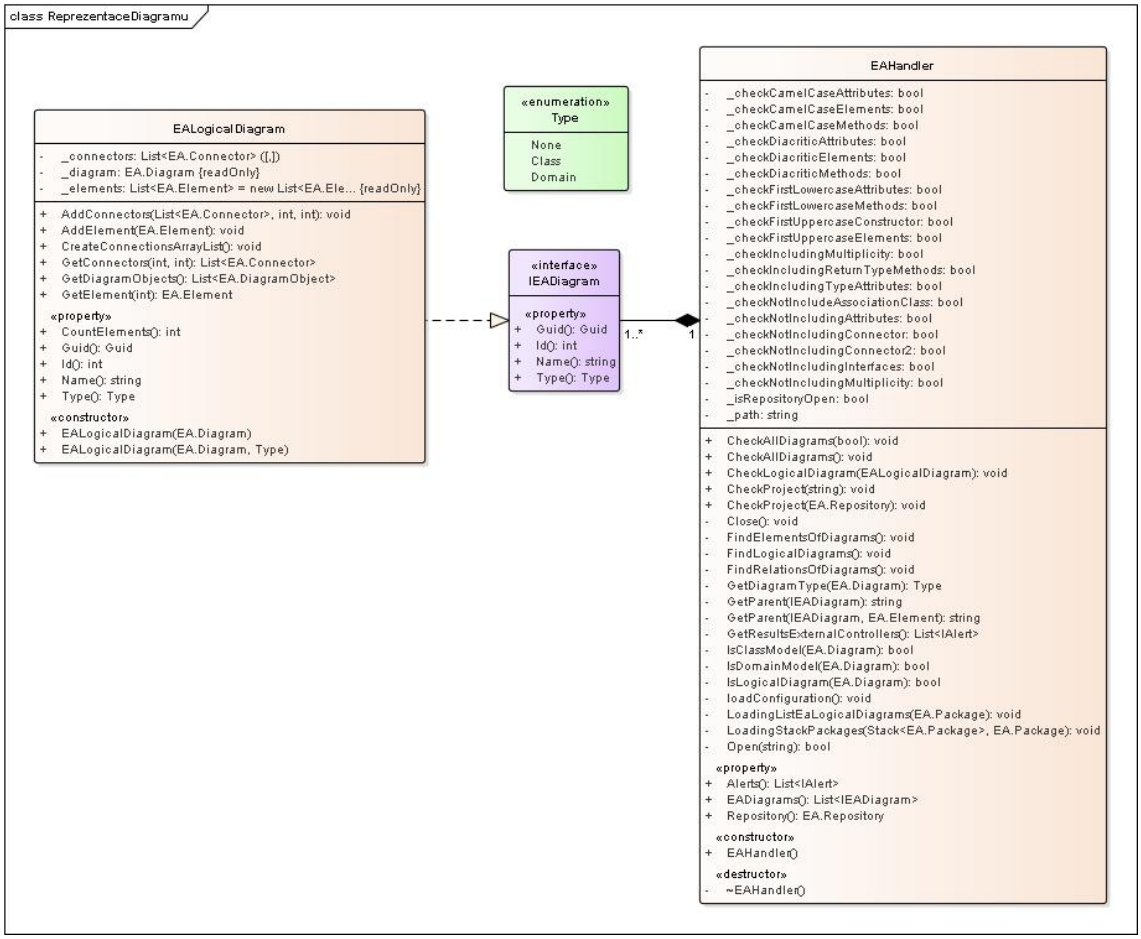
Toto je tedy první rozdělení typu, které v práci bude definováno pomocí třídy pro určitý typ. Další typ, a to, zda se jedná o analytický nebo návrhový model tříd, bude definovaný pomocí výčtového typu. Získání tohoto typu bude ale komplikované. Pomocí knihovny není možné se dostat k informaci, zda diagram je tohoto typu, pouze si zjistíme, že se jedná o typ Logical, tedy typ, který má společné nástroje. Program tedy bude muset umět pomocí názvu poznat, zda se jedná o návrhový nebo analytický model tříd. Jelikož si ale uživatel může zvolit vlastní název diagramu, je potřeba mu dát možnost zvolit si, který je jakého typu. Proto bude mít uživatel možnost nechat si vypsat všechny diagramy typu Logical ze souboru a u nich přepnout, jestli se jedná o analytický model tříd.

Třída, která pak bude zodpovědná za manipulaci s diagramem typu Logical, bude doplňovat rozhraní o další operace, které jsou pro daný model potřebné. Pomocí metody bude možné získat určité elementy, které diagram obsahuje. Dále jejich počet nebo jaké jsou relace mezi dvěma elementy podle jejich indexu v kolekci. Ve výsledku potřebujeme pro práci s diagramem a uložení reprezentace modelu jedno rozhraní. To se bude jmenovat `IEADiagram`. Dále pak třídu, která bude toto rozhraní implementovat a ukládat data z diagramu typu Logical, ta se bude jmenovat `EALogicalDiagram`. V poslední řadě pak výčtový typ enum určující typ diagramu a ten bude uložen pod názvem Typ v stejném soboru jako rozhraní.

4.2 Manipulace s repozitářem

K tomu, aby bylo možné vůbec pracovat s daty ze souboru, který je vytvořený pomocí programu Enterprise Architect, je potřeba otevřít tzv. repozitář. Pro otevření a další práci s repozitářem je využito technologie COM, kterou vytvořila společnost Microsoft pro operační systém Windows a poskytuje jí vývojářům. O této technologii se dozvíte dále v této práci.

Proto, aby bylo možné bezpečně pracovat s daty ze souboru a nenastala situace, že se poškodí, musíme hlídat, aby byl repozitář zavřený, pokud se z něj nenačítají data. Je nezbytné mít jak operaci pro otevření, tak pro zavření souboru, a v případě, že skončí aplikace chybou, musí se soubor zavřít v destruktoru třídy. Dále bude nutno pomocí nějaké metody projít celou strukturu souboru a nahrát do nějaké kolekce všechny diagramy typu Logical. Když budou všechny potřebné diagramy uloženy do objektu, musí se naplnit referencemi na všechny elementy a relace, se kterými se bude pracovat při kontrole. Po načtení všeho, co je potřebné, je možné provést kontrolu. Pro všechny tyto vlastnosti bude v projektu vytvořena třída `EAHandler`. Ta bude zodpovídat za veškerou manipulaci daty ze souboru a k její funkčnosti bude potřeba pouze to, aby jí byla předána cesta k souboru. Vlastníkem instance této třídy bude hlavní formulář, který jí bude předávat cestu k souboru a spustí následnou kontrolu všech diagramů v souboru. Model toho, jak bude vypadat třída `EAHandler` i reprezentace diagramu je možné vidět v ukázce Obrázek 1.



Obrázek 1: Diagram tříd reprezentace diagramu a třídy zodpovědné za manipulaci s nimi
Zdroj: vlastní zpracování pomocí programu Enteprise Architect

5 Korekce UML modelů

Pro to, aby mohl být UML model opravený, není asi problém najít mnoho pravidel v knihách a na internetu, která říkají, co má model obsahovat. Hůře se ale hledá, co model nemá obsahovat. Tento problém se dá ve většině případů odvodit. Například tak, že se dá předpokládat, že když analytický model předchází návrhovému modelu, tak prvky, které fungují jako rozšíření v druhé části modelování, by se určitě neměly objevit v části první.

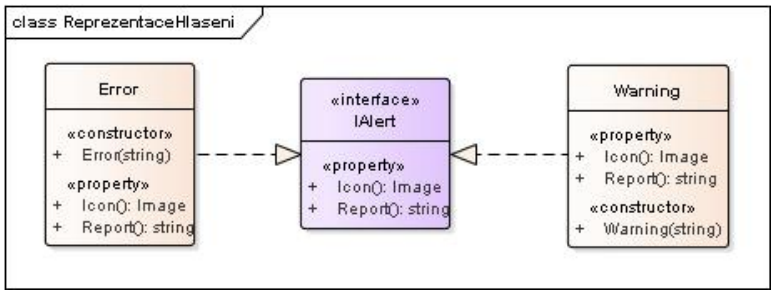
Chyby, kterých se vývojář na modelu může dopustit, se dají rozdělit na dva stupně. Prvním je pouhé varování, kdy se nemusí nutně jednat o chybu, druhý pak je, když se jedná o chybu, která by se určitě v modelu neměla objevit. Příkladem prvního stupně chyby může být například chybějící kolekce mezi třídami, protože je tady ale i možnost, že systém, který je modelován, bude tvořen na platformě, která má kolekce již implementované. V takovém případě může být kolekce znázorněna multiplicitou nebo jiným způsobem. Druhý stupeň pak může být například chyba, kdy název třídy nezačíná velkým písmenem. V části této práce již byla rozebírána analýza diagramu tříd. Dále je ale potřeba říct, co aplikace, která je součástí této práce, bude na modelu opravovat a proč.

5.1 Vyhodnocení modelu

O vyhodnocení, zda je v elementu nějaká chyba, se bude starat třída s názvem `EAController`. Tato třída bude statická, protože bude pouze obsahovat metody, které slouží ke kontrole elementu. V této třídě se budou vyskytovat dva druhy metod. První druh bude neveřejná metoda, které se budou předávat parametry ke kontrole. Tato metoda je zkontroluje a vrátí výsledek. Výsledek bude ve formě kolekce výčtových typů, které budou definovány v třídě `EAStatus` jako veřejný enum. Druhý druh metody bude veřejná operace, která bude vracet seznam s chybovým hlášením. Tato operace vždy předá potřebné parametry prvnímu druhu metody, díky čemuž dostane seznam chyb a podle nich vytvoří chybové hlášení. Druhý druh metody bude volán z instance třídy `EAHandler`. Ten postupně projde celou strukturu všech diagramů a nashromáždí hlášení o všech elementech a relacích.

5.2 Ukládání a prezentace hlášení o chybách

Šablonou pro chybu do hlášeni se stane rozhraní `IAlert`. Toto rozhraní definuje dvě přístupové metody. První, která se nazývá `Icon` a jejíž návratový typ je `Image`, bude vracet ikonu typickou pro druh chyby. Druhá přístupová metoda se bude nazývat Report. Její návratový typ je string a bude vracet text popisující chybu elementu nebo relace. Dále se vytvoří dvě třídy implementující rozhraní. První třída se jmenuje `Warning` a slouží pro varovnou chybu. Druhá se jmenuje `Error` a bude sloužit pro naprostou chybu, která se nemá v UML objevit. Obě třídy obsahují konstruktor, který bude třídě předávat text o chybě neboli obsah přístupové metody Report. Třídy `Error` a Warning také budou mít vždy výchozí hodnotu pro návratovou metodu `Icon`. Výchozí hodnotu třídy získají ze seznamu systémových ikon a to pomocí metody `SystemIcons.Error.ToBitmap()` pro třídu `Error` nebo pomocí `SystemIcons.Warning.ToBitmap()` pro třídu Warning.



Obrázek 2: Diagram tříd prezentující reprezentaci chybových hlášení
Zdroj: vlastní zpracování pomocí programu Enteprise Architect

5.3 Jednotlivé částí kontroly

Aplikace bude provádět kontroly každého prvku jinak. Kontroly je potřeba rozlišit do nějakých společných celků. Rozdělení kontrol a druhů stavů kontroly budou tři typy. Prvním je kontrola textu, která bude prováděna na názvech jednotlivých prvků, které uživatel vytvořil v diagramu tříd. Dalším typem kontroly je to, zda aplikace obsahuje prvek, který může být použit v dané situaci. Posledním druh kontroly je opakem té předešlé a zkoumá to, zdali diagram neobsahuje prvek, který se v diagramu nesmí používat, tak jak je v něm použit. Výsledky všech těchto kontrol budou uloženy pomocí výčtového typu, který reprezentuje stav kontroly. Výčtový typ je stejně jako kontroly rozdělen do tří druhů a to `EAStatus.Text`, `EAStatus.Include` a `EAStatus.NotInclude`.

5.3.1 Kontrola textů v diagramu tříd

5.3.1.1 Mezery v textu

Kontrola, zda je v textu mezera, je jedna ze dvou kontrol, a to v případě, že se jedná o text, který je psaný ve stylu CamelCase. Fakt, že je v textu mezera, se zjistí pomocí podmínky, ve které se kontroluje řetězec `text`, kde je uložený název kontrolovaného objektu. Pomocí metody `IndexOf` s dvěma parametry získá řetězec počet mezer, které obsahuje. První parametr metody je řetězec mezery a druhý parametr je výčtový typ `StringComparison.Ordinal,` jenž nastavuje metodu, tak aby prováděla kontrolu podle symbolů Unicode. Pokud v řetězci `text` bude jedna nebo více mezer, podmínka nechá chybu uložit. Ukázka celé podmínky je v ukázce Zdrojový kód 1.

5.3.1.2 Oddělovač v textu

Pokud je název typu CamelCase, nesmí obsahovat symboly, které oddělují slova. Proto je potřeba od druhého symbolu v řetězci do předposledního zkontrolovat, zda se tam takovýto nepovolený oddělovač nevyskytuje. Toho aplikace docílí pomocí podmínky, která nejdříve zjistí, zda délka názvu elementu je delší než 2 znaky. Pokud tomu tak je, proběhne kontrola pomocí metody `Regex.IsMatch,` která podle parametrů v řetězci zjistí, zda obsahuje povolené znaky. Těmi jsou jakékoliv písmeno z abecedy, číslice, tečka a mezera, která byla zkontrolována v předešlé podmínce. V případě, že v názvu, který byl zbaven prvního a posledního znaku pomocí metody `Substring`, bude znak, který nepatří mezi zmíněné aplikace, zaznamená stav jako chybu, kdy název obsahuje oddělovač.

```
1.  private static List<EAStatus.Text> CheckCamelCase(string text)
2.  {
3.      List<EAStatus.Text> statuses = new List<EAStatus.Text>();
4.
5.      if (text.IndexOf(" ", StringComparison.Ordinal) > 0)
6.          statuses.Add(EAStatus.Text.WhiteSpace);
7.
8.      if (text.Length > 2 &&
9.          !Regex.IsMatch(text.Substring(1, text.Length-2),
"^[\\p{L}0-9\\s\\.]+$")
10.         statuses.Add(EAStatus.Text.Delimiter);
11.
12.     return statuses;
13. }
```

Zdrojový kód 1: Kontrola CamelCase
Zdroj: vlastní zpracování

5.3.1.3 Diakritika

V názvech prvků se nemají objevovat znaky, se kterými by mohly mít problém některé programovací jazyky. Je tedy nutné zjistit, zda název neobsahuje diakritiku, která se v českém jazyce a některých jiných jazycích vyskytuje. To, jestli řetězec obsahuje diakritiku, se zjistí v podmínce za pomocí metody `Normalize,` která oddělí v názvu diakritické znaménko od písmene a vznikne tak další znak. Jestli název obsahuje diakritiku, poznáme podle toho, zda je název delší oproti původnímu, jsou-li diakritická znaménka oddělena od písmen.

```
1.  private static List<EAStatus.Text> CheckTextDiacritic(string text)
2.  {
3.      List<EAStatus.Text> statuses = new List<EAStatus.Text>();
4.
5.      if (text.Length < text.Normalize(
            System.Text.NormalizationForm.FormD).Length)
6.          statuses.Add(EAStatus.Text.Diacritic);
7.
8.      return statuses;
9.  }
```

Zdrojový kód 2: Kontrola diakritiky
Zdroj: vlastní zpracování

5.3.1.4 Malé a velké první písmeno

Názvy prvků při programování mají takové pravidlo, že začínají malým či velkým písmenem, aby se navzájem rozlišily. Tato pravidla se liší podle programovacího jazyka, a proto je obtížné zkontrolovat správnost tohoto problému. Přesto autor práce zvolil konvenci pojmenování z programovacího jazyka Java, jehož konvence pojmenování je hodně podobná ostatním programovacím jazykům.

V případě, že uživatel aplikace bude kontrolovat diagram pro jiný programovací jazyk, bude moci tuto chybu ve výpisu hlášení všech chyb přehlížet. Autor práce při kontrole ještě vzal v potaz možnost, že první znak není písmeno, ale symbol, čehož využívají některé programovací jazyky a kontroluje až druhý znak. Zda název začíná malým písmenem bude kontrolováno u atributů a metod, pokud se nejedná o konstruktor. Velké písmeno bude kontrolováno u názvu tříd, rozhraní a konstruktoru třídy. To, zda první znak začíná velkým nebo malým písmenem, poznáme pomocí metod `IsUpper` nebo `IsLower`. Předtím ale, než bude kontrolovat velikost písmen, bude zjištěno, zda se nejedná o znak, který není písmeno, pomocí metody `IsLower`.

```
1.  private static List<EAStatus.Text> CheckFirstUppercase(string text)
2.  {
3.      List<EAStatus.Text> statuses = new List<EAStatus.Text>();
4.      char[] charactersOfText = text.ToCharArray(0, 2);
5.
6.      if (char.IsLetter(charactersOfText[0]))
7.      {
8.          if (char.IsLower(charactersOfText[0]))
9.              statuses.Add(EAStatus.Text.FirstUppercase);
10.     }
11.     else if (char.IsLetter(charactersOfText[1]))
12.     {
13.         if (char.IsLower(charactersOfText[1]))
14.             statuses.Add(EAStatus.Text.FirstUppercase);
15.     }
16.
17.     return statuses;
18. }
```

Zdrojový kód 3: Kontrola velkého prvního písmena
Zdroj: vlastní zpracování

5.3.2 Kontrola prvků, které nesmí chybět v diagramu tříd

5.3.2.1 Typ atributu a návratový typ metody

V programování při deklaraci atributu se ve všech případech uvádí datový typ atributu. Stejně tak je tomu v případě metody, která musí mít definovaný návratový typ. Jediná výjimka nastává v případě, je-li metoda konstruktorem třídy a v tom případě se návratový typ neuvádí. O kontrolu se budou starat dvě metody, jedna určená pro typ atributu a druhá pro návratový typ metody. O zjištění zda metoda nebo atribut mají svůj typ se postará podmínka pomocí metody `string.IsNullOrEmpty`, které bude předán řetězec typem. Pokud bude řetězec prázdný, kontrola to zaznamená jako chybu.

```
1.  private static List<EAStatus.Include> CheckIncludeType(EA.Attribute attribute)
2.  {
3.      List<EAStatus.Include> statuses = new List<EAStatus.Include>();
4.      if (string.IsNullOrEmpty(attribute.Type))
5.          statuses.Add(EAStatus.Include.Type);
6.      return statuses;
7.  }
8.
9.  private static List<EAStatus.Include> CheckIncludeType(EA.Method method)
10. {
11.     List<EAStatus.Include> statuses = new List<EAStatus.Include>();
12.     if (string.IsNullOrEmpty(method.ReturnType) &&
13.         method.Stereotype.ToLower() != "constructor")
14.         statuses.Add(EAStatus.Include.Type);
15.     return statuses;
16. }
```

Zdrojový kód 4: Kontrola datového typu atributu a návratového typu metody
Zdroj: vlastní zpracování

5.3.3 Chybějící multiplicita

U relací, které se používají v UML pro modelování vztahů mezi elementy u analytického a návrhového diagramu typu asociace, agregace a kompozice, je nezbytné, aby obsahovaly na obou koncích multiplicitu pro správné znázornění jejich účelu. Tudíž je potřebné, aby pro správnost zmíněných typů relací aplikace zkontrolovala, zda mají na obou svých koncích nastavenou kardinalitu. Pro kontrolu toho, zda relace obsahuje multiplicitu, aplikace projde matici všech relací, které jsou v Enteprise Architect reprezentovány rozhraním `Connector` u diagramů určených ke kontrole. U každé relace z matice zkontroluje aplikace, zda je vhodného typu ke kontrole multiplicity. V případě, že se typy shodují, je relace předána ke kontrole. Každý konec relace v Enterprise Architect je reprezentován jako rozhraní `ConnectorEnd` a každý konec relace je kontrolován zvlášť.

```
1.  private static List<EAStatus.Include> CheckIcludeMultiplicity(EA.ConnectorEnd
    connectorEnd)
2.  {
3.      List<EAStatus.Include> statuses = new List<EAStatus.Include>();
4.      if (string.IsNullOrEmpty(connectorEnd.Cardinality))
5.          statuses.Add(EAStatus.Include.Multiplicity);
6.      return statuses;
7.  }
```

Zdrojový kód 5: Kontrola chybějící multiplicity na konci relace
Zdroj: vlastní zpracování

5.3.4 Kontrola prvků, které diagram tříd nesmí obsahovat

5.3.4.1 Kontrola špatně použité multiplicity

Podobně, jako je potřeba zkontrolovat, zda u relace nebylo zapomenuto na multiplicitu, je nutné také zkontrolovat, zda relace, které nemají obsahovat multiplicitu, ji neobsahují. Vybírání relací ke kontrole probíhá stejně, jako když aplikace kontrolovala, zda v relaci multiplicita nechybí. Jen v tomto případě je kontrolováno, zda je relace typu nesting, generalizace, realizace a zavislost. Kontrola probíhá podobně jako je tomu v ukázce Zdrojový kód 5: Kontrola chybějící multiplicity na konci relace. Zdrojový kód se pouze liší v tom, že podmínka se spustí, pokud řetězec `connectorEnd.Cardinality` obsahuje nějakou hodnotu. Dále se kód ještě liší v tom, že je použit výčtový typ `EAStatus.NotInclude` místo `EAStatus.Include`.

5.3.4.2 Použití atributů v rozhraní

Rozhraní je element, který předepisuje metody pro třídy, které ho budou implementovat. Nemůže ale předepisovat atributy, což Enteprise Architect nezakazuje, jelikož se v podstatě jedná o třídu se stereotypem interface. Musíme tedy zkontrolovat, zda uživatel špatně nedeklaroval atributy v rozhraní, což by vedlo k problémům v případě, kdy by se model převáděl na zdrojový kód. To, zda element obsahuje atributy, zjistí podmínka, která proběhne, pakliže počet atributů v kolekci `element.Attributes` je vyšší než nula.

```
1.  private static List<EAStatus.NotInclude> CheckNotIncludeAttribute(EA.Element
    element)
2.  {
3.      List<EAStatus.NotInclude> statuses = new List<EAStatus.NotInclude>();
4.      if (element.Attributes.Count > 0)
5.          statuses.Add(EAStatus.NotInclude.Attributes);
6.      return statuses;
7.  }
```

Zdrojový kód 6: Kontrola použití atributu v elementu rozhraní
Zdroj: vlastní zpracování

5.3.4.3 Použití špatné relace k elementu

Některé elementy nesmí být spojeny s některými typy relací, protože by spojení s nimi nedávalo logický význam a ve výsledku by diagram ilustroval nefunkční systém. Proto je potřeba zkontrolovat, zda rozhraní není spojeno s relací typu asociace, agregace, kompozice a generalizace. Další element, pro který musíme zkontrolovat s jakými typy relací je spojený, je balíček. Balíček nesmí být spojený s relací typu asociace, agregace, kompozice, generalizace a realizace. Pro to, abychom zkontrolovali správnost použití všech těchto relací, je potřeba, aby pro každý element a všechny relace, které jsou s ním spojeny, byly tyto předány ke zkontrolování. Typ elementu a relace pak bude porovnáván v podmínce atributy, které začínají buď na písmeno E a obsahují řetězec s typem elementu nebo začínají na písmeno C a obsahují řetězec s typem relace.

```
1.  private static List<EAStatus.NotInclude> CheckNotIncludeConnector(EA.Element
    element, EA.Connector connector)
2.  {
3.      List<EAStatus.NotInclude> statuses = new List<EAStatus.NotInclude>();
4.      string eType = element.Type.ToLower();
5.      string cType = connector.Type.ToLower();
6.
7.      if (eType == EInterface && cType == CAssociation)
8.          statuses.Add(EAStatus.NotInclude.Association);
9.      if (eType == EInterface && cType == CAggregation &&
10.         connector.SupplierEnd.Aggregation == 1)
11.         statuses.Add(EAStatus.NotInclude.Aggregation);
12.     if (eType == EInterface && cType == CAggregation &&
13.         connector.SupplierEnd.Aggregation == 2)
14.         statuses.Add(EAStatus.NotInclude.Composition);
```

Zdrojový kód 7: Část kontroly použití špatné relace k elementu
Zdroj: vlastní zpracování

5.3.4.4 Špatná relace nebo orientace relace mezi dvěma elementy

Ve chvíli, kdy aplikace zkontroluje, k jakým elementům byla přiřazena nevhodná relace, je nutno zkontrolovat situaci, kdy je použita špatná relace mezi dvěma elementy. Nastává totiž situace, kdy relace je správně použita k elementu v případě, že je brán jako jednotlivec. Ale v případě použití relace mezi dvěma elementy už může nastat situace, kdy je relace použita špatně. Příkladem takové situace může být použití relace typu realizace mezi dvěma rozhraními. Každé takové rozhraní může jako jednotlivec používat realizaci, ale na druhém konci relace musí vždy být jen třída, protože je nesmysl, aby v programování rozhraní implementovalo rozhraní.

Také může nastat situace, ve které je špatně použitý směr relace. A i toto je prověřeno v této kontrole. Ještě, než bude celá kontrola použití špatné relace nebo orientace popsána, je potřeba pro pochopení zdůraznit, že elementy předávané v proměnných element1 a element2 jsou seřazeny od zdroje k cíli. Toto je zkontrolováno metodou `IsSouserceElement` a případně touto metodou opraveno. Tato metoda ověří, zda první element je zdrojový a v případě, že tomu tak není, bude prohozeno pořadí mezi těmito dvěma elementy. V podmínce pak budou podobně jako v předešlé kontrole relace a elementu porovnávány typy obou elementů a relace.

```
1.  private static List<EAStatus.NotInclude> CheckNotIncludeConnector(EA.Element
    element1, EA.Element element2, EA.Connector connector)
2.  {
3.      List<EAStatus.NotInclude> statuses = new List<EAStatus.NotInclude>();
4.      string eType1 = element1.Type.ToLower();
5.      string eType2 = element2.Type.ToLower();
6.      string cType = connector.Type.ToLower();
7.
8.      if (eType1 == EInterface &&
9.          eType2 == EInterface &&
10.         cType == CRealisation)
11.         statuses.Add(EAStatus.NotInclude.Realization);
```

Zdrojový kód 8: Část kontroly použití špatné relace nebo její orientace mezi dvěma elementy
Zdroj: vlastní zpracování

5.3.4.5 Špatné použití relace podle typu diagramu

Jak už bylo v práci několikrát zmíněno, diagram tříd se rozděluje na dva typy. Analytický model tříd má definováno, že některé relace se v něm nesmí objevit. V analytickém modelu tříd nesmí být použita relace typu agregace, kompozice, realizace. To, jestli je relace použita špatně, kontroluje podmínka, ve které se porovnává výčtový typ diagramu a typ relace.

```
1.  private static List<EAStatus.NotInclude> CheckNotIncludeConnector(EA.Connector
     connector, Type diagramType)
2.  {
3.      List<EAStatus.NotInclude> statuses = new List<EAStatus.NotInclude>();
4.
5.      if (diagramType == Type.Domain &&
6.          connector.Type.ToLower() == CAggregation &&
7.          connector.SupplierEnd.Aggregation == 1)
```

Zdrojový kód 9: Kontrola špatné relace podle typu diagramu
Zdroj: vlastní zpracování

5.3.4.6 Použití asociační třídy v návrhovém modelu tříd

Asociační třída je jeden z elementů, který nesmí být použit pro návrhový model tříd. Její použití je čistě pro analytický model tříd a není použita v žádném z programovacích jazyků, tudíž nemůže být v návrhu nějakého informačního systému.

5.3.4.7 Použití rozhraní v analytickém modelu tříd

Analytický model tříd je jakýsi druh modelu určený pro uvažování nad budoucím systémem, ještě, než bude řešena celá struktura systému. V tomto typu modelu se řeší hlavně prvky z reálného světa a neuvažuje se nad softwarovou částí, proto není potřeba, aby zde byly použity prvky, které se využijí až v části návrhu

informačního systému. Proto je potřeba zkontrolovat, zda se v analytickém modelu tříd neobjevil element rozhraní, který se v něm nesmí použít.

6 Návrh řešení aplikace

6.1 Technologie COM Interop a její využití

Component Object Model neboli zkráceně COM je technologie, která umožňuje objektu nabídnout funkcionalitu aplikace jiným komponentám nebo cizím aplikacím [9]. To znamená, že vývojář nějaké aplikace chce funkcionalitu své aplikace nabídnout ostatním. Tuto technologii využila i společnost Sparx Systém ve svém programu Enterprise Architect. Díky tomu může autor této práce ve své aplikaci číst data ze souboru vytvořeném v programu Enterprise Architect. Aby bylo možné využít této služby cizí aplikace, je potřeba, aby její autor poskytl knihovnu interoperability. V případě Enteprise Architect se tato knihovna nachází v kořenovém adresáři programu pod názvem Interop.EA.dll. Aby uživatel hotové verze aplikace pro kontrolu UML modelů mohl aplikaci použít, musí mít nainstalován program Enteprise Architect s platně aktivovanou licencí. Pokud program Enterprise Architect nebude funkční nebo jeho licence nebude platná, nebude možné, aby aplikace pro kontrolu UML modelu mohla načíst soubor s diagramy. Enteprise Architect totiž slouží jako most mezi aplikací autorovy práce a souborem vytvořeným v Enteprise Architect.

6.1.1 Interoperabilita assembly

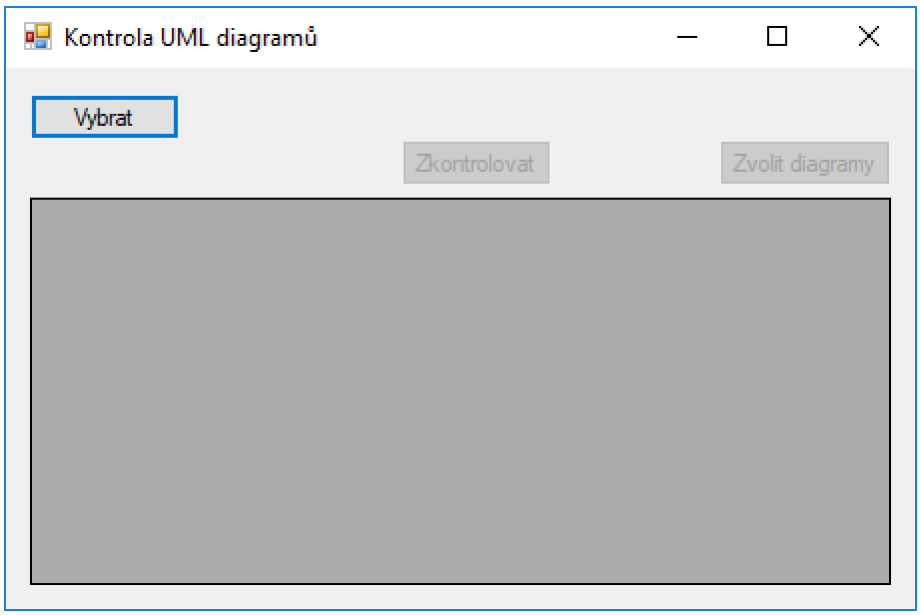
Sestavení nebo anglicky assembly je základním stavebním blokem v .NET Framework. Tento blok představuje prostor obsahující funkcionalitu prostřednictvím souborů spojených do jednoho celku, kterým je právě assembly [9]. V .NET Framework se rozlišuje více způsobů, jakými se prostor rozlišuje. Úplně prvním je jmenný prostor neboli namespace, ten rozděluje logické celky kódu. Pak je zmíněná assembly, ta je určená, aby rozdělovala fyzickou část kódu. Každá assembly je obsažena jako soubor formátu buď dll nebo exe [10]. Aplikaci, která je výstupem této práce, je rozdělena do pěti assembly a to: CheckUMLAddIn.dll, CheckUMLAlerts.dll, CheckUMLExternalController.dll, Interop.EA.dll a Kontrola UML.exe. Assembly Interop.EA.dll je jediná, která není vytvořena autorem této práce a je určena právě k interoperabilitě COM s programe Enteprise Architect. Toto znamená, že tato assembly mapuje objekty COM pro kód, který je tvořen někým jiným [9]. Autor této práce tak

může přistupovat k elementům určeným k interoperabilitě a pomocí jejich metod a funkcionality programu Enteprise Architect číst ze souborů, které byly ve zmíněném programu vytvořeny.

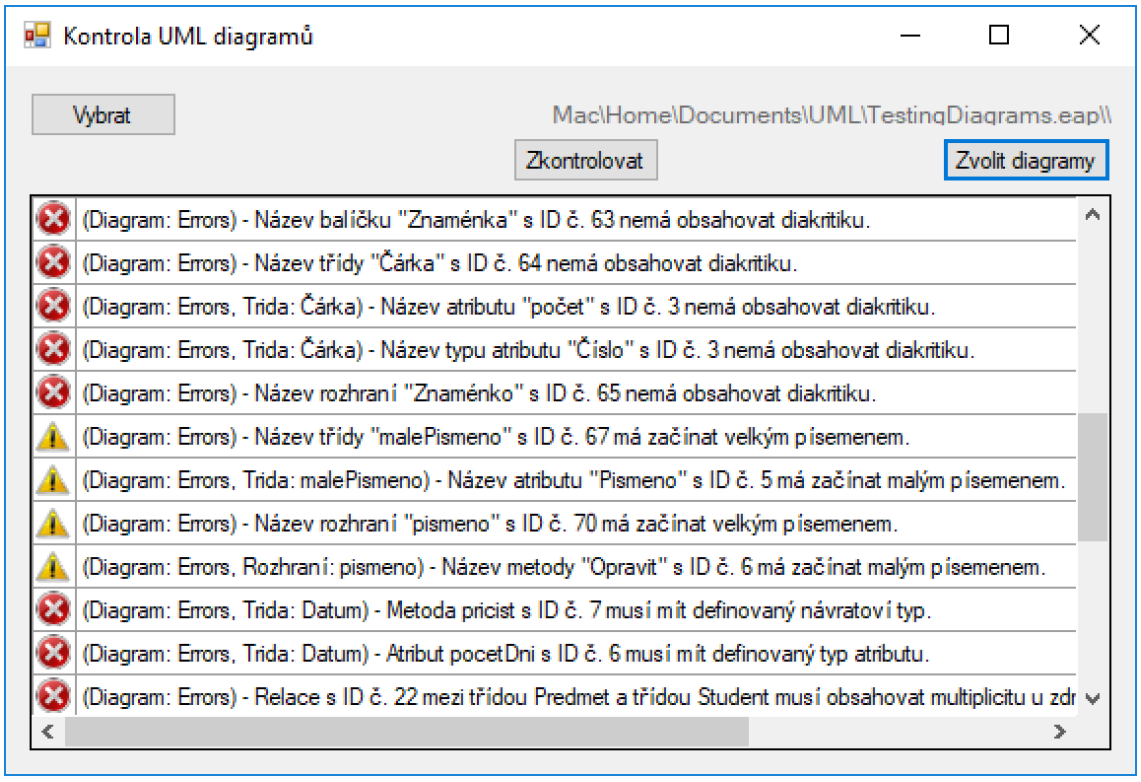
6.2 Hlavní formulář

Hlavní formulář bude jedna z nejdůležitějších součástí celé aplikace. Uživatel prostřednictvím hlavního formuláře bude moci spustit vše, co aplikace bude umět. Formulář je navržený intuitivně tak, že provede uživatele celým průběhem kontroly UML diagramu a také zajistí, aby uživatel neprovedl nechtěné kroky, či aby v nepravou chvíli přerušil proces tak, aby se aplikace ukončila chybou.

Když uživatel spustí aplikaci, první, co se načte, bude právě hlavní formulář. Poté, co se hlavní formulář zobrazí, jediné, co má uživatel možnost zvolit, je tlačítko „Vybrat“. Po jehož kliknutí se zobrazí okno, kde uživatel zvolí cestu k EAP souboru. Ve chvíli, kdy má uživatel vybrán soubor, což pozná podle textového pole v pravém horním rohu hlavního formuláře, který ukazuje, jaká je cesta k souboru, může uživatel kliknout na tlačítko „Zkontrolovat“. Toto tlačítko je aktivní pouze tehdy, je-li vybrán nějaký soubor formátu EAP a provede kontrolu celého souboru. Kontrola souboru může trvat delší dobu. To, že kontrola probíhá, indikuje kurzor přesýpacích hodin. Poté, co proběhne kontrola, se v tabulce, která je v dolní části hlavního formuláře, zobrazí hlášení chyb. Dále se po kontrole zaktivuje tlačítko „Zvolit diagram“, po něm se otevře dialogové okno, ve kterém může uživatel přepínat mezi typy diagramů. Podobu hlavního formuláře po zapnutí je možno vidět na ukázce Obrázek 3, a to, jak vypadá formulář po kontrole na ukázce Obrázek 4.



Obrázek 3: Ukázka aplikace po spuštění
Zdroj : Vlastní zpracování

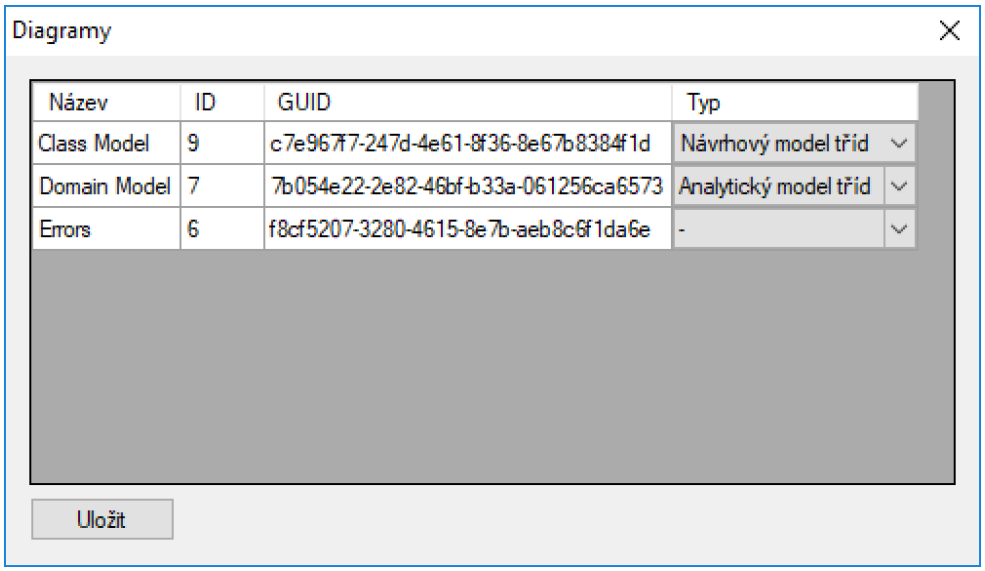


Obrázek 4: Ukázka aplikace po provedení kontroly
Zdroj: Vlastní zpracování

6.3 Dialogové okno

Dialogové okno bude sloužit jako prostředí, kde si uživatel může zvolit, jakého typu je Logical diagram. Uživatel má na výběr ze dvou typů: návrhový model tříd nebo analytický model tříd. Dialogové okno se bude skládat ze dvou částí. První část je tabulka s Logical diagramy a druhá tlačítko na uložení změn. Uživatel zde uvidí název diagramu, jeho identifikační číslo, GUID (globally unique identifier) a jako poslední položka tabulky bude rozbalovací menu, kde uživatel bude mít možnost zvolit typ Logical diagramu.

Když si uživatel bude jistý, že má nastavené diagramy podle jeho představ, potvrdí svoje nastavení tlačítkem „Uložit“. Po kliknutí na tlačítko se spustí znovu kontrola, což bude indikovat kurzor s přesýpacími hodinami. Ve chvíli, kdy vše proběhne v pořádku, dialogové okno se zavře a zobrazí se hlavní formulář s chybovým hlášením podle nastavení uživatele. Po dobu, kdy bude dialogové okno aktivní, uživatel nebude mít možnost manipulovat s hlavním formulářem. To, jak vypadá dialogové okno, je možné vidět na ukázce Obrázek 5.



Obrázek 5: Ukázka dialogového okna
Zdroj: Vlastní zpracování

6.4 Prohledávání struktury souborů a načítaní diagramů

Když byly nalezeny všechny diagramy, které soubor vytvořený v programu Enteprise Architect obsahuje, je potřeba projít celou jeho stromovou strukturu souboru. Na začátku celé struktury je kořen, který Enteprise Architect nazývá jako Model. Enteprise architect , nedovoluje vytvoření více těchto kořenů, přesto ukládá

tento kořen do kolekce, která jich může obsahovat více. V případě, že by se ale toto pravidlo v budoucnu změnilo, je nutno, aby aplikace zkontrolovala, zda kolekce neobsahuje více těchto kořenů a případně prošla všechny. Dalším prvkem souboru je balíček, který rozděluje strukturu na jednotlivé větve. Pro toto rozdělení je potřeba, aby aplikace prošla všechny balíčky obsažené v kořenu souboru a našla v nich diagramy, které uloží podle jejich typu do seznamu všech diagramů typu Logical, které budou podle rozdělení na analytický a návrhový model tříd předány ke kontrole.

O procházení celé struktury se stará metoda `FindLogicalDiagrams,` obsah vidíme v ukázce Zdrojový kód 10. Tato metoda obsahuje cyklus, který prochází všechny kořeny struktury neboli Models. Uvnitř cyklu pak podmínka zjistí, zda model obsahuje nějaké balíčky, a pokud ano, uloží všechny do zásobníku pomocí metody `LoadingStackPackages`. Tato metoda naplnění zásobníku vždy projde kolekci balíčků a pomocí cyklu do zásobníku uloží další balíčky, které předek všech těchto balíčků v kolekci obsahuje. Když jsou všechny balíčky z kořenu struktury uloženy v zásobníku, spustí se cyklus, který postupně prochází všechny balíčky v zásobníku, dokud není zásobník prázdný. Při každém průchodu cyklem se uloží všechny diagramy z kolekce diagramů pomocí metody `LoadingListEaLogicalDiagrams`. Dále se ještě přidají další balíčky, které obsahuje balíček, který byl právě v cyklu odejmut ze zásobníku.

```
1.  private void FindLogicalDiagrams()
2.  {
3.      for ( short i = 0; i < Repository.Models.Count; i++ )
4.      {
5.          EA.Package model = (EA.Package) Repository.Models.GetAt(i);
6.          if (model.Packages.Count > 0)
7.          {
8.              Stack<EA.Package> stackPackages  = new Stack<EA.Package>();
9.              LoadingStackPackages(stackPackages, model);
10.
11.             while ( stackPackages.Count != 0 )
12.             {
13.                 EA.Package package = stackPackages.Pop();
14.                 LoadingListEaLogicalDiagrams(package);
15.                 LoadingStackPackages(stackPackages, package);
16.             }
17.         }
18.     }
19. }
20.
21. private void LoadingStackPackages(Stack<EA.Package> stackPackage, EA.Package
    package)
22. {
23.     for (int j = 0; j < package.Packages.Count; j++)
24.     {
25.         stackPackage.Push((EA.Package)package.Packages.GetAt((short)j));
26.     }
27. }
```

Zdrojový kód 10: Procházení struktury souboru
Zdroj: vlastní zpracování

6.5 Načítání elementů do reprezentace diagramu

K tomu, aby byly uloženy všechny elementy, které diagram obsahuje, je potřeba využít triku, kterým nám interoperabilita Enteprise Architect dovoluje, protože není žádný způsob, jak zjistit z elementu definovaného rozhraním `EA.Element` v jakém diagramu definovaném rozhraním `EA.Diagram` je obsažen. Žádnou z metod, které jsou k dispozici za pomoci interoperability, nelze zjistit v rozhraní EA.Diagram, jaké obsahuje elementy definované rozhraním `EA.Element`. Chceme-li zjistit, jaké diagram obsahuje elementy, je potřeba nejdříve pochopit, jak jsou elementy v repozitáři reprezentovány.

Elementy jsou totiž v repozitáři reprezentovány ve dvou formách. První forma byla v této práci zmíněna již několikrát, a tou je rozhraní `EA.Element,` které se nachází ve struktuře dokumentu v jednotlivých balíčcích. Druhá forma je definována pomocí rozhraní `EA.DiagramObject` a je vždy obsažena v kolekci uvnitř diagramu `EA.Diagam`. Tento druh reprezentace představuje grafickou podobu elementu v diagramu a obsahuje například polohu v diagramu nebo identifikační číslo elementu, který představuje. Díky tomu, že `EA.DiagramObject` obsahuje identifikační číslo

elementu, které je uloženo v atributu s názvem `ElementID`, je možné pak pomocí metody repozitáře `GetElementByID` najít element, který je obsahem diagramu.

O nahrání všech elementů do reprezentace diagramu se v aplikaci stará metoda `FindElementsOfDiagrams`. Ukázku celé metody je možno vidět v Zdrojový kód 11.

```
1.  private void FindElementsOfDiagrams()
2.  {
3.      foreach (EALogicalDiagram eaLogicalDiagram in EADiagrams)
4.      {
5.          List<EA.DiagramObject> diagramObjects = eaLogicalDiagram.GetDiagramObj
    ects();
6.          foreach (EA.DiagramObject diagramObject in diagramObjects)
7.          {
8.              if (Repository.GetElementByID(
9.                  diagramObject.ElementID).Type.ToLower() == "class" ||
10.                 Repository.GetElementByID(
11.                     diagramObject.ElementID).Type.ToLower() == "interface" ||
12.                 Repository.GetElementByID(
13.                     diagramObject.ElementID).Type.ToLower() == "package")
14.             {
15.                 eaLogicalDiagram.AddElement(
16.                     Repository.GetElementByID(diagramObject.ElementID));
17.             }
18.         }
19.     }
20. }
```

Zdrojový kód 11: Načítání elementů do reprezentace diagramu
Zdroj: vlastní zpracování

6.6 Načítání relací do reprezentace diagramu

Po tom, co aplikace uloží všechny elementy, které diagramy typu Logical obsahují, musí být naplněna reprezentace diagramu ještě relacemi, které znázorňují vztahy mezi elementy. Relace se ukládají do dvourozměrného pole, které reprezentuje vztahy mezi dvěma elementy.

Při ukládání relací je potřeba počítat s dvěma skutečnostmi, které jsou u relací neobvyklé, ale v diagramu se mohou objevovat. První možnost, která může nastat v diagramu tříd je, že mezi dvěma elementy může být více než jedna relace. Proto, aby se předešlo tomu, aby jedna relace nepřepsala jinou ve dvourozměrném poli, je typ hodnoty pole v reprezentaci diagramu definován na generický datový typ List, který obsahuje všechny možné relace definované rozhraním `EA.Connector` mezi dvěma určitými elementy. Takto je možno přidat vždy další relaci, která je mezi elementy jednoduše do kolekce. Druhou skutečností, která může nastat, je ta, že element obsahuje relaci na sebe sama a jelikož se relace hledá vždy mezi dvěma elementy, objevovala by se reprezentace mezi dvěma elementy duplicitně. Pro tento účel je potřeba hlídat

pomocí podmínky, zda relace není s elementem už jednou spojena. O načítání všech relací se stará metoda `FindRelationsOfDiagrams`.

Uvnitř metody je cyklus, který prochází všechny uložené diagramy typu Logical. Uvnitř cyklu se nejdříve vytvoří dvourozměrné pole na základě počtu elementů pomocí metody `CreateConnectionsArrayList`, kterou vlastní třída `EALogicalDiagram`. Dále následuje další cyklus, který prochází všechny elementy. Uvnitř druhého cyklu je ještě třetí cyklus, který prochází všechny relace, které element má. Následuje čtvrtý cyklus, který prochází všechny elementy uvnitř diagramu a hledá shodu pomocí identifikačního čísla mezi sousedícími elementy relace, které má relace `EA.Connector` uloženy v atributu `ClientID` nebo `SupplierID` a mezi identifikačním číslem elementů uložených v reprezentaci diagramu.

Poté, co čtvrtý cyklus skončí a aplikace má identifikační číslo obou sousedních elementů relace, se pomocí podmínky zkontroluje, zda není mezi elementy už vytvořena kolekce. Pokud není žádná kolekce ještě vytvořena, vytvoří se nová kolekce mezi elementy a je do ní přidána relace a zbylé tři cykly se opakují. Pokud ale elementy už mezi sebou mají definovanou relaci, tak zkontroluje cyklus, který projde celou kolekci a podmínka, zda se ta samá relace nevyskytuje už v kolekci mezi elementy. Pokud ne, tak aplikace ji přidá do kolekce a pokračuje v opakování předešlých tří cyklů.

6.7 Načítání nastavení aplikace z konfiguračního souboru

U uživatele, který využívá aplikaci pro kontrolu UML, která je vytvořena jako část této bakalářské práce, může nastat situace, kdy nepotřebuje některou z kontrol, které aplikace poskytuje.

Je tedy vhodné dát uživateli možnost vypínat jednotlivé kontroly. Toho lze docílit pomocí konfiguračního souboru, ten je v kořenovém adresáři aplikace a nazývá se: „Kontrola UML.exe.config“. Konfigurační soubor je psán v jazyce XML a vypínače jednotlivých kontrol se v něm nachází pod tagem appSettings. V tomto tagu jsou vnořeny další tagy typu add, které obsahují dva atributy key a value. Hodnotou atributu key je vždy název kontroly a pro value je hodnota buď true nebo false. Každá z kontrol má nad sebou komentář, který popisuje, o jakou kontrolu se jedná. Při spuštění aplikace si pak `EAHandler` nastavení všech kontrol stáhne z konfiguračního souboru a uloží jej do privátních atributů stejně pojmenovaných jako názvy kontrol

v konfiguračním souboru. Těchto atributů si můžeme všimnou na diagramu modelující třídu EAHandler v ukázce Obrázek 1

Všechny tyto atributy jsou defaultně nastaveny na true, tedy tak, aby všechny kontroly proběhly, a to pro případ, kdyby konfigurační soubor v adresáři aplikace chyběl nebo kdyby byl špatně napsaný. O načtení konfiguračního souboru se stará metoda `loadConfiguration` v třídě `EAHandler`. Metoda, která načítá konfiguraci, je spuštěna v kontraktoru třídy. To, jak vypadá konfigurační soubor bez komentářů popisujících, co kontrola provádí, je možné vidět v ukázce Zdrojový kód 12.

```
<appSettings>
  <add key="checkDiacriticElements" value="true" />
  <add key="checkIncludingMultiplicity" value="true" />
  <add key="checkNotIncludingMultiplicity" value="true"
  <add key="checkNotIncludingConnector" value="true" />
  <add key="checkNotIncludingConnector2" value="true" />
  <add key="checkCamelCaseElements" value="true" />
  <add key="checkFirstUppercaseElements" value="true" />
  <add key="checkCamelCaseMethods" value="true" />
  <add key="checkDiacriticMethods" value="true" />
  <add key="checkFirstLowercaseMethods" value="true" />
  <add key="checkFirstUppercaseConstructor" value="true"
  <add key="checkIncludingReturnTypeMethods" value="true"/>
  <add key="checkCamelCaseAttributes" value="true" />
  <add key="checkDiacriticAttributes" value="true" />
  <add key="checkFirstLowercaseAttributes" value="true" />
  <add key="checkIncludingTypeAttributes" value="true" />
  <add key="checkNotIncludeAssociationClass" value="true" />
  <add key="checkNotIncludingAttributes" value="true" />
  <add key="checkNotIncludingInterfaces" value="true" />
</appSettings>
```

Zdrojový kód 12: Konfigurace jednotlivých kontrol bez komentářů
Zdroj: vlastní zpracování

6.8 Načítání dalších kontrol ze složky

Aby uživatelé mohli dále rozšiřovat aplikaci o vlastní kontroly, je potřeba vytvořit systém, který jim to umožní. Systém, který je pro toto v aplikaci navržen, je dělán tak, aby načítal knihovny dll, které budou uloženy ve složce ExternalControllers a hledal v nich třídy které používají rozhraní vytvořené autem práce.

Toto rozhraní se nazývá `IEAExternalController` a to kde ho najít a jak jej použít je popsáno níže v této práci. O načítání všech tříd se stará metoda s názvem `GetResultsExternalControllers`, která je v třídě `EAHandler`. Metoda začíná tak, že nejdříve pomocí podmínky zjistí, zda existuje v kořenové složce aplikace složka ExternalControllers. Potom pomocí statické třídy `Directory` a metody `GetFiles` aplikace získá do proměnné typu pole `nameControllers` všechny knihovny dll, které jsou v dané složce.

Poté, co jsou všechny názvy uloženy, cyklus je všechny projde a získá název sestavení neboli assembly z každé z knihoven pomocí statické třídy `AssemblyName` a metody `GetAssemblyName`. Pomocí názvu sestavení pak nahraje aplikace, každou z nich do kolekce jako třídu `Assembly` pomocí metody `Load`. Když cyklus skončí a aplikace má uloženy všechny sestavení ze složky ExternalControllers, začne druhý cyklus, který prochází všechna sestavení.

Cyklus začíná podmínkou, která kontroluje, zda nebyla sestavení chybně načtena a proměnná tak neobsahuje žádnou instanci sestavení. Pokud je podmínka splněna, aplikace pomocí metody `GetTypes` uloží do proměnné typu pole všechny typy elementů, které sestavení obsahuje. Typy elementů jsou definované třídou `Type` a po uložení spustí vnořený cyklus, který projde všechny uložené typy elementů. Ve vnořeném cyklu jsou pak dvě podmínky. První zjišťuje, zda je element abstraktní nebo rozhraní. Pokud ano, tak je tento typ elementu přeskočen, cyklus pokračuje na další typ v poli. Druhá podmínka se snaží zjistit, zda typ elementu implementuje rozhraní `IEAExternalController`. Pokud toto rozhraní je implementováno, je typ elementu uložen do kolekce všech typů, které rozhraní `IEAExternalController` implementují.

Po tom, co cyklus procházející všechna sestavení i cyklus v něm vnořený skončí, spustí se další cyklus, který prochází všechny uložené typy elementů implementující potřebné rozhraní a vytváří z nich instance pomocí statické třídy `Activator` a metody `CreateInstance`. Tato instance se pak přidá do kolekce.

Když je kolekce naplněna všemi instancemi a cyklus skončí, začne poslední cyklus, který prochází každou instanci elementů implementující rozhraní `IEAExternalController`. Každé instanci předá repozitář z kontrolovaného souboru, který `EAHandler` vlastní a pak získá pomocí zavolání metody `GetResults` z instance všechna chybová hlášení. Jelikož metoda `GetResultsExternalControllers`, která celý tento postup provede, má návratovou hodnotu rozhraní `IAlerts`, definující všechna chybová hlášení vrátí všechna chybová hlášení, které získala. Ty jsou pak v průběhu kontroly přičtena k ostatním chybovým hlášením, které aplikace má uložené ve třídě `EAHandler` a které jsou po kontrole vypsány v tabulce hlavního formuláře.

```
1.  string[] nameControllers = Directory.GetFiles(
2.  AppDomain.CurrentDomain.BaseDirectory+path, "*.dll");
3.  List<Assembly> assemblies = new List<Assembly>();
4.  foreach (string nameController in nameControllers)
5.  {
6.      AssemblyName assemblyName = AssemblyName.GetAssemblyName(nameController);
7.      assemblies.Add(Assembly.Load(assemblyName));
8.  }
9.  System.Type typeController = typeof(IEAExternalController);
10. List<System.Type> typeControllers = new List<System.Type>();
11. foreach (Assembly assembly in assemblies)
12. {
13.     if (assembly != null)
14.     {
15.         System.Type[] types = assembly.GetTypes();
16.         foreach (System.Type type in types)
17.         {
18.             if (type.IsAbstract || type.IsInterface)
19.                 continue;
20.             if (type.GetInterface(typeController.FullName) != null)
21.                 typeControllers.Add(type);
22.         }
23.     }
24. }
25. List<IEAExternalController> controllers = new List<IEAExternalController>();
26. foreach (System.Type type in typeControllers)
27. {
28.     controllers.Add((IEAExternalController) Activator.CreateInstance(type));
29. }
30. foreach (IEAExternalController controller in controllers)
31. {
32.     controller.Repository = Repository;
33.     alerts.AddRange(controller.GetResults());
34. }
```

Zdrojový kód 13: Načítání kontrol z knihoven dll uložených ve složce ExternalControllers
Zdroj: vlastní zpracování

6.9 Tvorba externích kontrol

Dát uživatelům možnost vytvářet si vlastní kontroly, je šikovným způsobem, jak zajistit, aby aplikace podporovala i další druhy diagramů UML nebo všech ostatních, které je možno v programu Enteprise Architect vytvářet.

A jelikož je UML komplexní a existují různé metodiky, jak tento grafický jazyk využít nebo nastane situace, kdy uživatel modeluje systém pro konkrétní programovací jazyk se specifickými pravidly pro UML, může uživatel jednoduchým způsobem aplikaci rozšířit o vlastní způsob kontroly. K tomu, aby vytvořit takovou aplikaci, musí vytvořit projekt, který bude zkompilován do dll knihovny, kterou pak nahraje do složky ExternalControllers. Aby byl projekt funkční, potřebuje, aby měl reference na knihovny dll, které se nazývají CheckUMLExternalController, CheckUMLAlerts a Interop.EA.

Všechny tyto knihovny jsou v kořenové složce aplikace. Poté, co bude mít uživatel referenci na potřebné knihovny v projektu, bude moct využít nejdůležitějšího

prvku, který je potřebný pro tvorbu externího kontroly, a to rozhraní `IEAExternalController`. To je obsaženo v knihovně CheckUMLExternalController. Zbylé dvě knihovny obsahují prvky, které jsou potřebné pro správnou funkčnost rozhraní. Knihovna CheckUMLAlerts obsahuje rozhraní `IAlerts` a všechny typy chybových hlášení, která toto rozhraní implementují. Poslední knihovna Interop.EA obsahuje důležité prvky k práci s repozitářem a interoperabilitou programu Enterprise Architect. To, jak rozhraní vypadá, je možno vidět v ukázce Zdrojový kód 14.

```
1.  public interface IEAExternalController
2.  {
3.      Repository Repository { get; set; }
4.      List<IAlert> GetResults();
5.  }
```

Zdrojový kód 14: Rozhraní předepisující externí kontroly
Zdroj: vlastní zpracování

Aby bylo budoucím uživatelům lépe zřejmé, jak vytvořit takovouto externí kontrolu, udělal autor této práce ukázkový projekt, který mají uživatelé volně k dispozici na tomto odkazu: https://gitlab.com/goryllaz/CheckUMLSampleExternalController. Ukázka je také k dispozici na CD, které je přiložené k práci. Projekt obsahuje jednu třídu, která jmenuje `SampleExternalController`. U třídy je vidět, že implementuje rozhraní `IEAExternalController`, které je potřebné, aby externí kontrola byla načtena při kontrole diagramu v třídě `EAHandler`. Ve třídě je pak vidět implementovaná metoda `GetResult` z rozhraní. Metoda obsahuje kontrolu, která není nějak významná a nemusí být pravdivá, ale ilustruje, jak by kontrola nějakého prvku z repozitáře, který je třídě předán v průběhu načítání kontroly, mohla vypadat. Ukázková kontrola ilustruje, jak cyklus prochází všemi kořeny struktury repozitáře a pak pomocí podmínky kontroluje, zda se jmenují Model. Pokud ne, označí to za chybu a uloží jí do kolekce hlášení. V tomto případě se jedná o chybové hlášení, které definuje třída Error, která je obsažena v knihovně CheckUMLAlerts. Této třídě je v konstruktoru předán řetězec: `$"(Model: {model.Name}) - Název kořenového adresáře musí být Model"`.

Tato část má ukazovat uživateli, jak by měl vypadat vzor takového chybového hlášení, kde v závorce je představen rodičovský prvek nebo prvek, kterého se kontrola týká a po ukončení závorky a po pomlčce je napsán text, který chybu popisuje. Uživatel ale není nikterak omezován na tom, jaký řetězec chybovému hlášení předá a může tak

zvolit vlastní vzor. To, jak vypadá kód ukázkové třídy externího rozhraní, je možno vidět v ukázce Zdrojový kód 15. Aby aplikace správně fungovala při kontrole, musí se uživatel vyvaroval zavírání a otvírání repozitáře. Ten je třídě předán již otevřený a jeho otevření nebo zavření by mohlo špatně ovlivnit chod aplikace.

```
1.  public class SampleExternalController: IEAExternalController
2.  {
3.      public List<IAlert> GetResults()
4.      {
5.          List<IAlert> alerts = new List<IAlert>();
6.          for (short i = 0; i < Repository.Models.Count; i++)
7.          {
8.              Package model = (Package) Repository.Models.GetAt(i);
9.              if (model.Name != "Model")
10.                 alerts.Add(new Error($"(Model: {model.Name}) -
     Název kořenového adresáře musí být Model"));
11.         }
12.         return alerts;
13.     }
14.
15.     public Repository Repository { get; set; }
16. }
```

Zdrojový kód 15: Ukázková třída externí kontroly
Zdroj: vlastní zpracování

6.10 Použití aplikace jako add-in do Enteprise Architect

Možnost, kterou aplikace vytvořená jako výstup této práce disponuje je, že jde přidat do programu Enteprise Architect jako add-in. Díky tomu je možné jí spustit rovnou z programu Enteprise Architect, kde jí bude rovnou předán soubor, který uživatel v programu modeluje. Nemusí tak hledat v souborových adresářích, kde je soubor uložený. Stačí když add-in správně nainstaluje. To, jak nainstalovat add-in je popsáno níže v této práci. Po provedení instalace uživatelem se objeví u verze Enteprise Architect v záložce Extend nová položka v menu s název Kontrola UML.

Když na položku uživatel klikne, objeví se možnost spustit aplikace, potvrdí ji znovu kliknutím. Nyní naběhne formulář podobný hlavnímu formuláři, který se objeví, když je aplikace spuštěna běžným způsobem. Ve formuláři ale chybí tlačítko pro vybrání souboru pro kontrolu a text ukazující cestu k souboru. Uživateli pak už jen stačí kliknout na tlačítko Zkontrolovat a případně pak na tlačítko Zvolit diagram.

Celý add-in je obsažený v kořenovém adresáři aplikace v dll knihovně CheckUMLAddIn. Skládá se ze čtyř prvků. Prvními dvěma jsou formuláře upravené z aplikace tak, aby neobsahovaly zbytečné součásti. Třetím prvkem je abstraktní třída `EAAddinBase` s frameworkem pro tvorbu add-inu do programu Enteprise Architect,

který je k dispozici na GitHub a jehož autorem je Geert Bellekens. Celý framework je k dispozici pod dvoubodovou licencí BSD. Posledním prvkem je třída `CheckUML` která implementuje abstraktní třídu `EAAddinBase`. Třída `CheckUML` pak přepisuje z abstraktní třídy jednu metodu a dva atributy. První atribut definuje název add-inu v menu Enteprise Architect a druhý atribut názvy jednotlivých položek. Metoda, která se jmenuje `EA_MenuClick` pak definuje kliknutí na položku s název Spustit, tak aby spustila hlavní formulář add-inu.

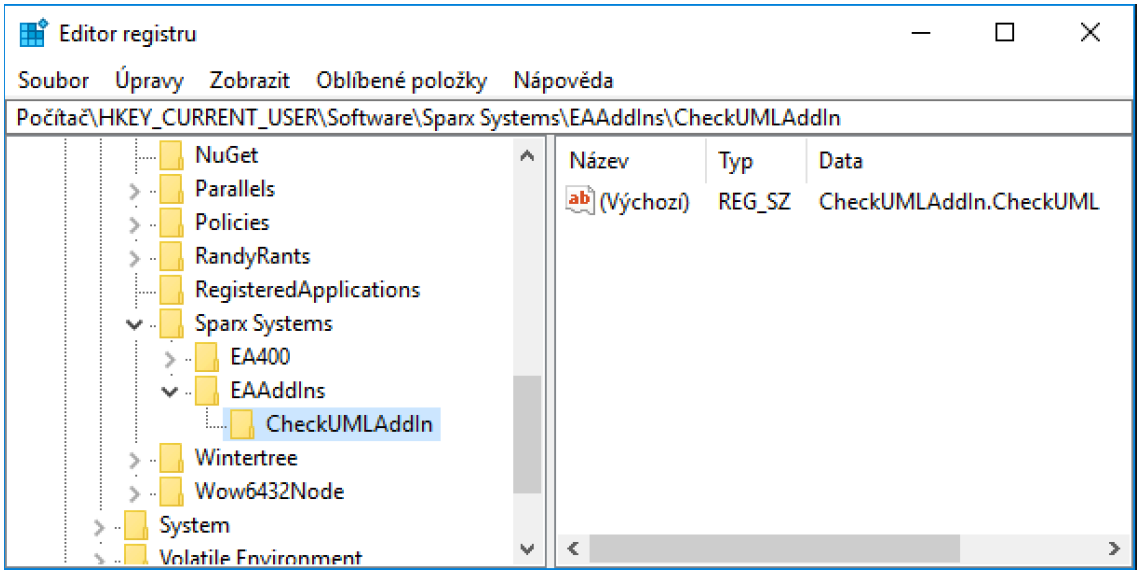
6.10.1 Instalace add-inu do programu Enteprise Architect

Pro instalaci add-inu je potřeba podstoupit několik důležitých úkonů. Nejprve musíme aplikaci pro kontrolu UML nahrát někam do počítače. Adresář aplikace se nesmí přesouvat nebo přejmenovávat, takže je potřeba na toto zvolit vhodný prostor. Když je aplikace umístěna na vhodné místo i se všem knihovnami dll, je nutno spustit příkazový řádek jako správce. Dále se musíme dostat pomocí příkazu cd do adresáře, kde je uložena aplikace a spustit příkaz:

„`%WINDIR%\Microsoft.NET\Framework\v4.0.30319\regasm CheckUMLAddIn.dll /codebase`“.

Po proběhnutí příkazu je potřeba spustit systémový registr Windows regedit. Ve struktuře registru je potřeba najít Sparx Systems. Cesta ke struktuře je takováto: „Počítač\HKEY_CURRENT_USER\Software\Sparx Systems“. Dále je potřeba kliknout pravým tlačítkem myši na složku Sparx Systems a najet myší na položku Nový a zvolit Klíč. Název klíče bude EAAddIns. Je potřeba vytvořit ještě jeden nový klíč tentokrát v nově vytvořené složce EAAddins. Tentokrát se klíč bude jmenovat CheckUMLAddIn. Po kliknutí levým tlačítkem myši na složku CheckUMLAddIn je potřeba kliknout pravým na buňku v tabulce obsahující text „(Výchozí)“ a vybrat možnost změnit. Zobrazí se formulář. V kolonce formuláře Údaj hodnoty je potřeba napsat: „CheckUMLAddIn.CheckUML“.

Výsledek by pak měl vypadat jako na ukázce Obrázek 6. V tuto chvíli je add-in nainstalovaný, je možné ho najít v programu Enteprise Architect. Pokud chce uživatel využít externích kontrol pro aplikaci v add-inu, je potřeba vytvořit složku ExternalControllers v kořenovém adresáři programu Enterprise Architect a nahrát externí kontroly do ní.



Obrázek 6: Ukázka systémového registru Windows po instalaci add-inu
Zdroj: vlastní zpracování

7 Tvorba aplikace

7.1 Diagramy pro testování aplikace

Pro otestování všech částí kontroly, vytvořil autor soubor se třemi diagramy, které modelují chyby, které mohou nastat při tvorbě diagramu tříd.

První diagram, který je nazvaný Errors, obsahuje všechny chyby, které mohou nastat jak u analytického modelu tříd, tak u návrhového modelu tříd. Po načtení souboru do aplikace je potřeba, aby byl tento diagram nastaven nejlépe jako návrhový model tříd, aby u něj proběhla kontrola, protože jeho název nekoresponduje s běžně používanými názvy pro analytický nebo návrhový model tříd. Další dva diagramy jsou analytický a návrhový model tříd, které modelují chyby, jež jsou specifické pro daný typ modelu.

7.2 Nástroje pro tvorbu a testování aplikace

Celá tvorba aplikace proběhla na operačním systému Windows 10, který společnost Microsoft nabízí zdarma na stránkách developer.microsoft.com jako SDK (Software development kit).

Jako vývojové prostředí autor práce použil Visual Studio ve verzi enterprise s licencí, kterou autor získal díky programu DreamSpar od společnosti Microsoft, který autorova škola poskytuje. Dále pak bylo potřeba pro vývoj aplikace funkční aktivovaný program Enteprise Architect od společnosti Sparx Systems. Ten autor získal díky školní licenci a byly použity dvě verze 7.1.833 a 13.0.1304.

Nejdříve autor začal s první verzí a v průběhu tvorby aplikace přešel na novější verzi. A jelikož byly použity funkce, které poskytuje pouze novější verze Enteprise Architect, aplikace se stala nekompatibilní s první verzí. Aplikace sice dokáže opravit modely, ale pro správné spuštění aplikace uživatel musí mít v počítači nainstalovanou novější verzi Enteprise Architect. Testování aplikace pak ještě proběhlo na 32 bitovém operačním systému Windows 7, který stejně jako operační systém použitý pro vývoj aplikace poskytuje Microsoft zdarma pro účely vývoje. Testování aplikace pak probíhalo na diagramu, který je součástí přílohy této práce nebo na diagramech, které autor vytvořil po dobu svého studia na vysoké škole.

7.3 Kompatibilita

K tomu, aby uživatel mohl spustit aplikaci, musí mít nainstalovaný .NET Framework 4 ve svém počítači. Dále uživatel potřebuje mít nainstalovaný Enteprise Architect, nejlépe ve verzi 13 a vyšší, u které je jisté, že je kompatibilní s aplikací. Je ale možné, že aplikace bude fungovat i s některými staršími verzemi. U program Enteprise Architect musí mít také uživatel aktivovanou licenci.

8 Shrnutí výsledku

Teoretická část bakalářské práce popisuje a analyzuje jednotlivé části analytického a návrhového modelu tříd, což bylo u grafického jazyku UML obtížné, protože dokumentace vytvořená společností OMG tohoto jazyk nepopisuje pravidla pro jeho použití, pouze popisuje jeho podobu a funkčnost. Odborná literatura obsahující metodiku použití UML se pak v některých případech rozchází v jeho pravidlech. Dále se pravidla mění podle toho, v jakém jazyce bude informační systém naprogramován. Proto se autor práce musel rozhodovat, která varianta pravidla bude použita při kontrole UML diagramu tříd na základě nejběžnějšího použití pravidla nebo vlastních zkušeností UML.

Dále v teoretické části bakalářské práce autor popsal metody užité k tvorbě aplikace, která kontroluje vybrané diagramy a kterou autor sám vytvořil. V této části se autor zabýval podrobným popisem, jak aplikace převede všechny informace důležité ke kontrole diagramu do aplikace a správně je roztřídí do vhodné reprezentace.

Praktickou částí bakalářské práce bylo vytvoření celé aplikace, která byla ještě doplněna o možnost rozšíření pomocí externích kontrol a použití aplikace jako add-in v programu Enterprise Architect. Zdrojový kód a spustitelná verze je součástí přiloženého CD. Spustitelnou verzi aplikace dal autor také veřejně k dispozici na stránce https://gitlab.com/goryllaz/Kontrola-UML/tags/v1.0.2, odkud si ji kdokoliv může stáhnout. Některé z částí zdrojového kódu byly popsány autorem v teoretické části, a to hlavně zdrojový kód jednotlivých kontrol a implementace externích kontrol.

Proto, aby autor mohl vytvořit aplikaci, která je výstupem této práce, bylo nutné, aby se naučil programovat v jazyce C# společně s .NET Framework a naučil se používat technologii COM Interop. Tato práce byla pro autora první zkušenost se zmíněným jazykem a technologii.

9 Závěr

Cílem této bakalářské práce bylo naprogramovat aplikaci, která bude kontrolovat UML modely vytvořené pomocí programu Enteprise Architect. Autorovi této práce se nakonec povedlo vytvořit takovou aplikaci, která dokáže zkontrolovat analytický a návrhový model tříd.

Tyto dva modely jsou sice důležitou součástí grafického jazyka UML, ale pouze jednou z částí při vývoji informačního systému, kdy je použito UML. Je ještě mnoho diagramů UML, které jsou nedílnou součástí při analýze a návrhu informačního systému. Také by bylo dobré, aby aplikace uměla rozlišovat mezi jednotlivými programovacími jazyky, ale jelikož existuje velké množství takových jazyků a další v průběhu let přibývají, je skoro nemožné, pro každý z nich udělat implementaci. Řešením je vytvořit externí kontroly které toto budou které budou aplikaci o další vlastnosti rozšiřovat. Bylo by také dobré lokalizovat aplikaci do angličtiny nebo případně dalších jazyků, aby se mohla využít v širším okruhu vývojářů.

Proto autor této práce chce i po dokončení bakalářské práce pokračovat ve vylepšování a doplnění funkcí aplikace, která může sloužit jako užitečný nástroj pro vývojáře informačních systému. Lidský faktor totiž zůstane omylný a aplikace dokáže v krátkém čase najít chyby, kterých by si člověk nemusel všimnout, což by vedlo k chybám, které se zpětně špatně opravují.

10 Seznam použité literatury

[1] Unified Modeling Language: Infrastructure. Version 2.0. Needham, Massachusetts: Object Management Group, 2005.

[2] ARLOW, Jim a Ila NEUSTADT. UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky. 2., aktualiz. a dopl. vyd. Brno: Computer Press, 2007. ISBN 978-80-251-1503-9.

[3] Sparx Systems [online]. Creswick, Victoria: Sparx Systems, 200n. l. [cit. 2016-08-10]. Dostupné z: www.sparxsystems.com.au

[4] FOWLER, Martin. Destilované UML. Praha: Grada, 2009. Knihovna programátora (Grada). ISBN 978-80-247-2062-3

[5] KANISOVÁ, Hana a Miroslav MÜLLER. UML srozumitelně. 2., aktualiz. vyd. Brno: Computer Press, 2006. ISBN 80-251-1083-4.

[6] MILES, Russ a Kim. HAMILTON. *Learning UML 2.0*. Sebastopol, CA: O'Reilly, c2006. ISBN 9780596009823.

[7] LARMAN, Craig. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. 3rd ed. Upper Saddle River, N.J.: Prentice Hall PTR, c2005. ISBN 01-314-8906-2.

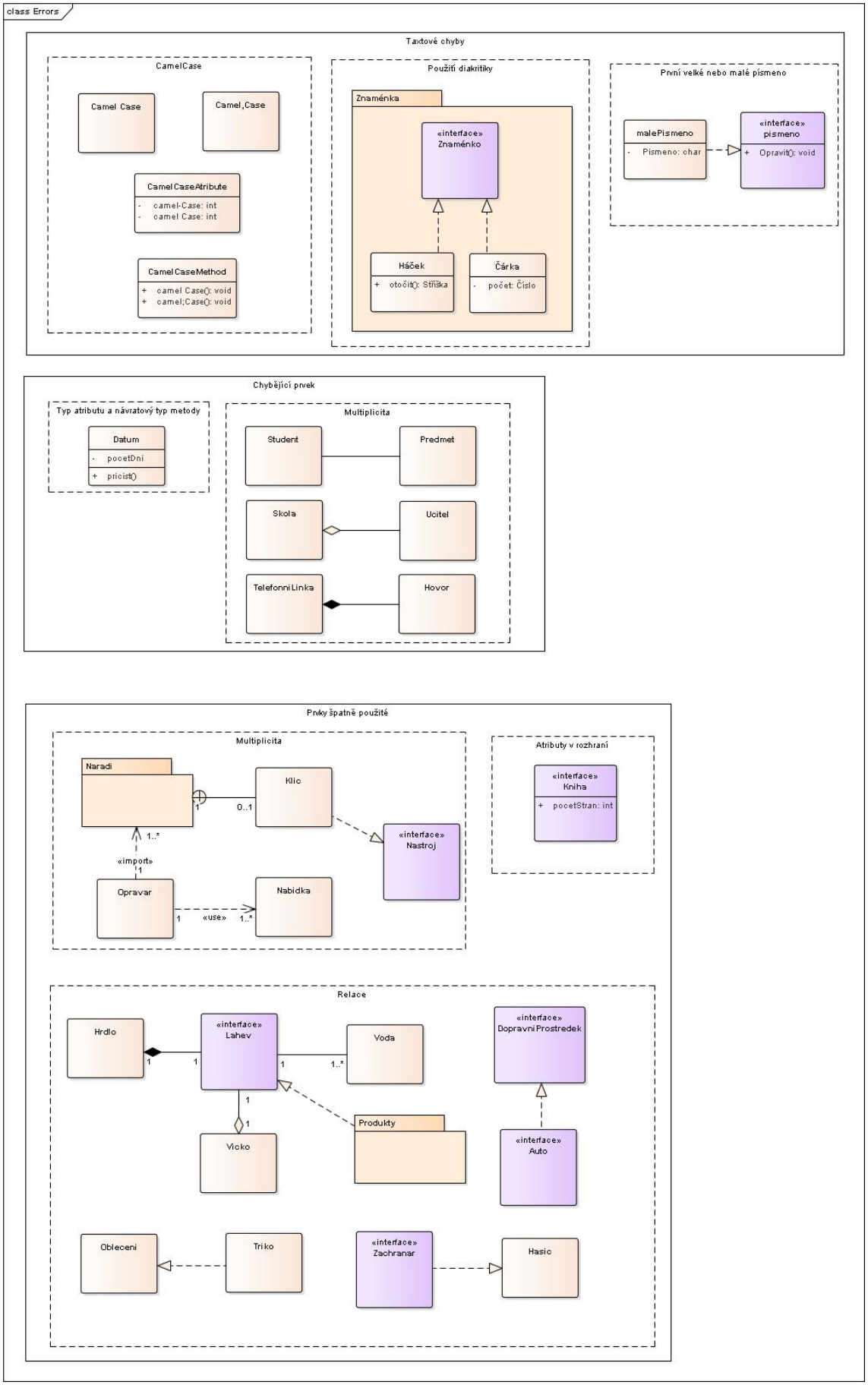
[8] HLINĚNÝ, Petr. Základy teorie grafů [online]. 1 vyd. Brno: Masarykova univerzita, 2010 [cit. 2017-01-29]. Elportál. Dostupné z: <http://is.muni.cz/elportal/?id=878389>. ISSN 1802-128X.

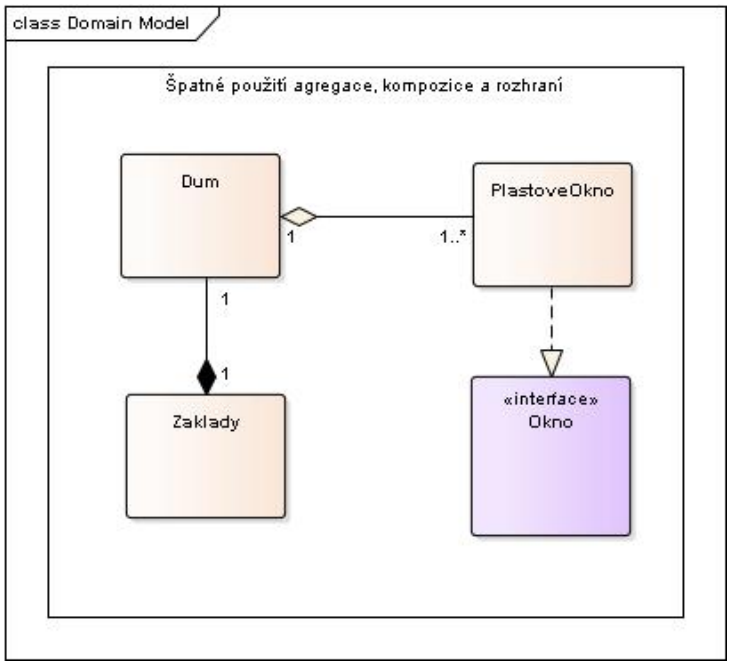
[9] Introduction to COM Interop (Visual Basic). *Microsoft Docs* [online]. Redmond (Washington): Microsoft, 2017 [cit. 2017-07-29]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/com-interop/introduction-to-com-interop

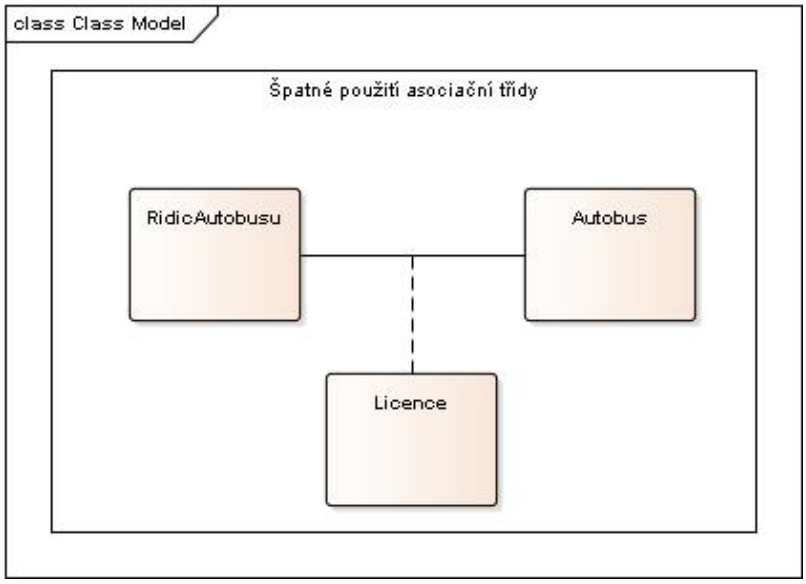
[10] Understanding and Using Assemblies and Namespaces in .NET. *Microsoft Developer Network* [online]. Redmond (Washington): Microsoft, 2002 [cit. 2017-08-20]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms973231.aspx

11 Přílohy

1. Diagramy s chybami vytvořené pro testování aplikace
    a. Diagram chyb v analytickém a návrhovém modelu tříd
    b. Diagram chyb v analytickém modelu tříd
    c. Diagram chyb v návrhovém modelu tříd
2. Přílohy na přiloženém CD
    a. Zdrojový kód aplikace
    b. Zdrojový kód ukázkové externí kontroly
    c. Spustitelná verze aplikace
    d. Diagram pro otestovaní kontrolován


class Errors
Taxtové chyby
CamelCase
Camel Case
Camel,Case
CamelCaseAtribute
- camel-Case: int
- camel Case: int
CamelCaseMethod
+ camel Case(): void
+ camel;Case(): void
Použití diakritiky
Znaménka
«interface»
Znaménko
Háček
+ otočit(): Stříška
Čárka
- počet: Číslo
První velké nebo malé písmeno
malePismeno
- Pismeno: char
«interface»
pismeno
+ Opravit(): void
Chybějící prvek
Typ atributu a návratový typ metody
Datum
- pocetDni
+ pricist()
Multiplicita
Student
Predmet
Skola
Ucitel
TelefonniLinka
Hovor
Prvky špatně použité
Multiplicita
Naradi
Klic
0..1
1
1..*
«import»
1
«interface»
Nastroj
Opravar
Nabidka
1
«use»
1..*
Atributy v rozhraní
«interface»
Kniha
+ pocetStran: int
Relace
Hrdlo
1
1
«interface»
Lahev
1
Voda
1..*
1
1
Vicko
Produkty
«interface»
DopravniProstredek
«interface»
Auto
Obleceni
Triko
«interface»
Zachranar
Hasic


class Domain Model
Špatné použití agregace, kompozice a rozhraní
Dum
PlastoveOkno
1
1..*
1
1
Zaklady
«interface»
Okno


class Class Model
Špatné použití asociační třídy
RidicAutobusu
Autobus
Licence

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2015/2016

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai3-p)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Urbanec Jan	Na Stráni 82, Týnec nad Labem	I1301345

TÉMA ČESKY:

Automatizovaná kontrola UML modelů

TÉMA ANGLICKY:

Automated Checking of UML Models

VEDOUCÍ PRÁCE:

Ing. Pavel Čech, Ph.D. - KIT

ZÁSADY PRO VYPRACOVÁNÍ:

viz metodické pokyny

Osnova:
1. Úvod
2. Cíl práce
3. Analýza UML modelů
4. Reprezentace UML modelů
5. Korekce UML modelů
6. Návrh řešení aplikace
7. Tvorba aplikace
8. Shrnutí výsledku
9. Závěr

Cíl:
Cílem bakalářská práce je podrobně analyzovat UML modely. Dále pak z výsledků analýzy vytvořit vhodnou reprezentaci UML modelů a najít metodu, jak provést korekci UML modelů na základě pravidel a konvencí, které se při tvorbě UML modelů používají. Konečným výsledkem práce bude vytvoření aplikace, která bude umět zkontrolovat alespoň model tříd ze souboru, který byl vytvořený v programu Enterprise Architect.

SEZNAM DOPORUČENÉ LITERATURY:

viz digitální knihovny

Podpis studenta: ..... **Datum:** 13.10.2015

Podpis vedoucího práce: ..... **Datum:** 13.10.2015

(c) IS/STAG , Portál - Podklad kvalifikační práce , I1301345 , 12.10.2015 20:40