ŽILINSKÁ UNIVERZITA V ŽILINE
Fakulta riadenia a informatiky

Diplomová práca

Študijný odbor: Informačné systémy

Bc. Miroslav Špigura

Generátor kódu z UML diagramov

Vedúci: Mgr. Ing. Ľubomír Sadloň, PhD.

Reg. č.: 216/2006 máj 2007

Žilina

Čestné prehlásenie

Prehlasujem, že diplomovú prácu som vypracoval samostatne s použitím uvedenej literatúry.

Žilina, máj 2007 . . . . . .

vlastnoručný podpis

Poďakovanie

Touto cestou ďakujem Mgr. Ing. Ľubomírovi Sadloňovi, Phd. vedúcemu diplomovej práce, za cenné rady, pripomienky a podnety, ktoré mi poskytol pri vypracovávaní diplomovej práce a mojim rodičom, za ich podporu, ktorou ma sprevádzali počas celého vysokoškolského štúdia.

Obsah

Zoznam obrázkov

1 Úvod

Narastajúca zložitosť informačných systémov kladie vysoké nároky na prácu softvérových analytikov, architektov a inžinierov. Od počiatočných záujmov o štruktúru a kvalitu zdrojového kódu, softvéroví inžinieri postupne zameriavajú svoju pozornosť na modelovacie aspekty vývoja softvérových systémov. Modely poskytujú abstrakcie systémov, ktoré umožňujú lepšie zvládnutie väčších a komplexnejších aplikácií jednoduchšími spôsobmi, nezávisle od technológií, ktoré ich implementujú.

Všeobecne platí, že čím neskôr v procese vývoja softvéru sa odhalí chyba v špecifikácii, resp. nesprávne pochopenie požiadaviek zákazníka, tým bude vyššia cena za odstránenie tejto chyby. Takisto sa predĺži aj čas na vývoj daného softvéru. Preto je dôležité, aby sa dbalo na kvalitu a presnosť počiatočných fáz vývoja.

UML, vizuálny modelovací jazyk, sa stal jedným z najviac používaných štandardov na špecifikáciu a dokumentáciu informačných systémov. Skutočnosť, že UML je dosť všeobecný jazyk, môže byť nevýhodou pri modelovaní niektorej konkrétnej domény, pre ktorú by boli vhodnejšie viac špecializované jazyky. Jazyk UML poskytuje určité mechanizmy umožňujúce rozšírenie a úpravu jeho syntaxe a sémantiky na prispôsobenie sa danej doméne.

V tejto práci sa snažím riešiť otázku generovania kódu z UML diagramov do rôznych programovacích jazykov.

V druhej kapitole si stanovíme ciele práce. V kapitole 3 sa budeme venovať jazyku UML najskôr si povieme niečo o UML a potom sa budeme venovať diagramom tried z ktorých sa bude generovať kód. V štvrtej kapitole uvedieme aké sú spôsoby automatického generovania kódu. Piata kapitola v krátkosti popisuje XML, keďže šablóny generátora budú práve v tomto formáte. Šiesta kapitola obsahuje popis modelovacieho nástroja UML.FRI hlavne tých častí, ktoré budú potrebné pre generovanie. V siedmej kapitole bude popísaná vlastná implementácia generovania kódu a v kapitole osem to bude generovanie dokumentácie.

2 Ciele práce

Najprv si uvedieme ciele, ktoré budeme v tejto práci sledovať, a ktoré sa pokúsime touto prácou naplniť. Umožní nám to získať lepšiu predstavu o tom, akým smerom sa bude uberať naše snaženie.

V tejto práci si stanovíme nasledujúce ciele:

- Vytvorenie univerzálnych šablón, ktoré by mohli byť použité nielen na generovanie zdrojových kódov, ale aj na spätné generovanie UML diagramov z o zdrojových kódov. Šablóny musia byť univerzálne aj v tom, že bude možné navrhnúť aj tak aby ich bolo možné použiť aj na iné programovacie jazyky ako sú delphi, python a C++.

- Vytvoriť šablónu pre generovanie dokumentácie vo formáte HTML.

- Vytvoriť generátor, ktorý bude používať navrhnuté šablóny a implementovať ho do nástroja UML.FRI.

- Vytvoriť generátor, ktorý bude generovať dokumentáciu zo šablóny ktorá bola vytvorená pre dokumentáciu.

3 UML

Jazyk UML (Unified Modeling Language) je jazyk vizuálneho modelovania, ktorý je použiteľný pre špecifikáciu, vizualizáciu, projektovanie a dokumentáciu softwarových komponentov, bussiness-procesov a iných systémov. Jazyk UML je jednoduchý a silný nástroj modelovania, ktorý môže byť použitý pri tvorbe konceptuálnych, logických a grafických modelov zložitých systémov rôzneho cieľového použitia. Tento jazyk v sebe obsahuje tie najlepšie vlastnosti metód programového inžinierstva, ktoré sa s úspechom používali behom posledných rokov pri modelovaní rozsiahlych a komplikovaných systémov. Je založený na niekoľkých základných pojmoch, ktoré môžu byť bez väčších problémov naštudované a následne použité väčšinou programátorov a návrhárov, ktorí sú zoznámení s metódami objektovo orientovanej analýzy a návrhu.

Medzi významné charakteristiky UML môžeme zaradiť:

- Poskytnúť k dispozícii užívateľom ľahko pochopiteľný a prehľadný jazyk vizuálneho modelovania, ktorý je špeciálne prispôsobený pre vytvorenie dokumentácie modelov zložitých systémov rozdielneho cieľového použitia.
- Popis jazyka UML musí mať takú špecifikáciu modelu, ktorá nezáleží na použitom programovacom jazyku.
- Popis jazyka UML musí obsahovať sémantický základ pre pochopenie obecných vlastností objektovo orientovanej analýzy a návrhu.
- Umožniť rozšírenie a špecializáciu základných pojmov jazyka UML pre presnejšie zobrazenie modelu systému v konkrétnej oblasti.
- Vyvolať rozšírenie objektovo orientovaných technológií a odpovedajúcich pojmov objektovo orientovanej analýzy a návrhu.
- Integrovat do seba najnovšie a najlepšie poznatky objektovo orientovanej analýzy a návrhu.

3.1 História UML

UML sa v podstate stal štandardom pre modelovanie softvérových aplikácií a jeho popularita narastá aj v modelovaní iných domén. Vznikol v rámci úsilia zjednodušiť a konsolidovať veľký počet objektovo orientovaných vývojových metód, ktoré v tom čase vznikali. Jeho korene sú v troch odlišných metódach: Boochova Metóda, ktorej autorom je Grady Booch, Metóda na modelovanie objektov (Object Modeling Technique), ktorej spoluautorom je James Rumbaugh, a Objectory, autorom ktorej je Ivar Jacobson. Známi ako Traja kamaráti, Booch, Rumbaugh a Jacobson začali v roku 1994 pracovať na niečom, čo sa neskôr stalo prvou oficiálnou verziou jazyka UML.

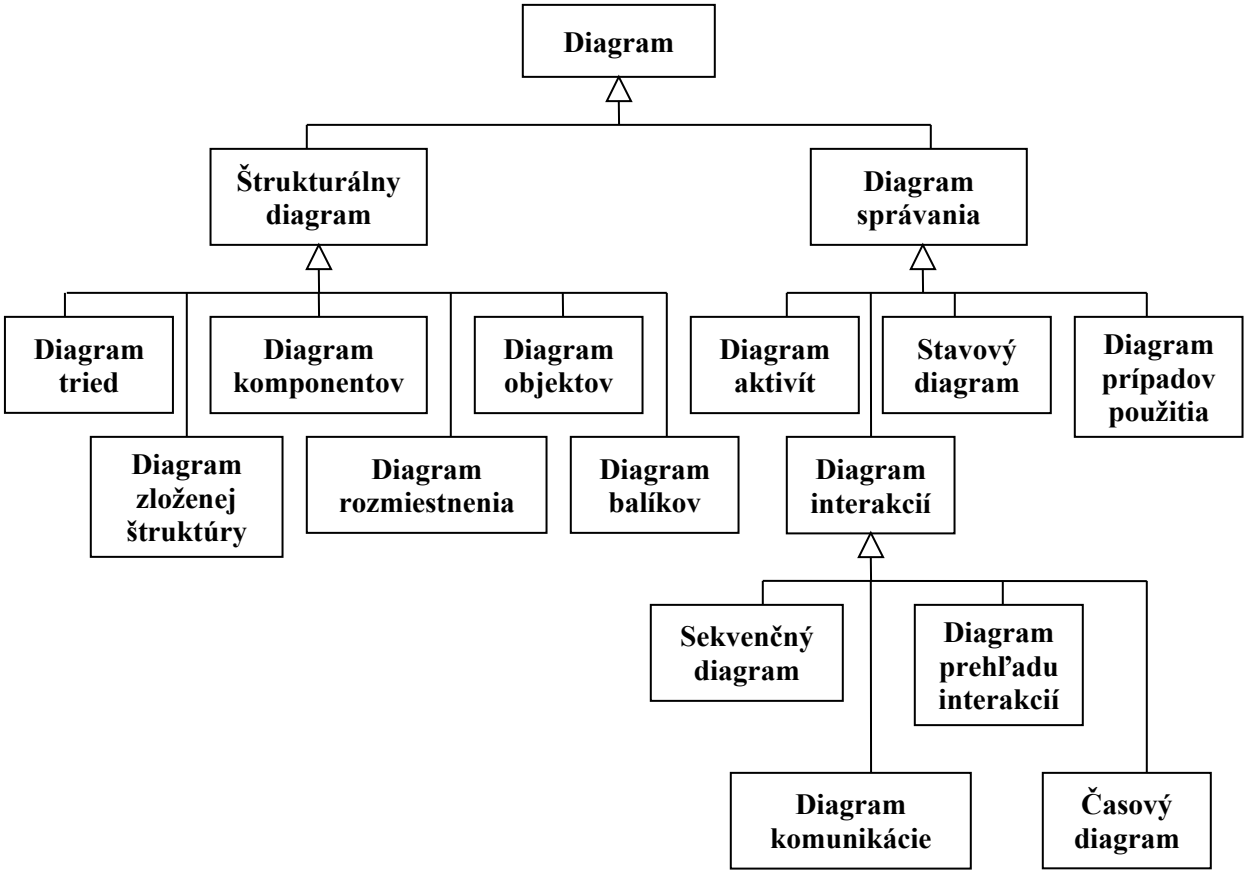
V roku 1996 OMG vydalo požiadavku na návrhy ohľadom štandardného prístupu k objektovo orientovanému modelovaniu. Booch, Rumbaugh a Jacobson začali spolupracovať s metodológmi a vývojármi z rôznych spoločností, aby vytvorili návrh dostatočne zaujímavý pre členov OMG, ako aj modelovací jazyk, ktorý by bol všeobecne akceptovaný tvorcami CASE modelovacích nástrojov, metodológmi a vývojármi, ktorí by sa v konečnom dôsledku stali jeho používateľmi.

Finálny návrh UML bol predložený OMG v roku 1997 a bol výsledkom spolupráce mnohých ľudí. V tom istom roku bol jazyk UML akceptovaný OMG a vydaný ako UML verzia 1.1. UML odvtedy prešiel niekoľkými zmenami a vylepšeniami až po súčasnú verziu 2.0. Každá revízia sa snažila venovať problémom a nedostatkom, ktoré boli identifikované v predchádzajúcich verziách, čo viedlo k zaujímavému rozširovaniu a zmenšovaniu jazyka. UML 2.0 je zatiaľ najrozsiahlejšia špecifikácia vzhľadom na počet strán, ale reprezentuje doteraz najkompaktnejšiu verziu UML.

3.2 Typy diagramov UML

UML model obvykle pozostáva z jedného alebo viac diagramov. Diagram graficky znázorňuje entity a vzťahy medzi týmito entitami. Tieto entity môžu byť skutočné objekty reálneho sveta, softvérové entity a konštrukcie a pod. Je bežné, že jedna entita sa vyskytuje aj na viacerých diagramoch. Každý diagram totiž znázorňuje konkrétny pohľad na danú entitu, ktorá je modelovaná.

UML 2.0 popisuje 13 oficiálnych typov diagramov. Delí ich na dve kategórie: štrukturálne diagramy (structural) a diagramy správania (behavioral). Celkové delenie diagramov je zobrazené na nasledujúcom obrázku.



Obrázok 1– *Delenie typov diagramov v jazyku UML*

3.2.1 Štrukturálne diagramy

Štrukturálne diagramy znázorňujú statické vlastnosti modelu. Používajú sa na zachytenie fyzickej organizácie entít v systéme. UML 2.0 definuje 6 štrukturálnych diagramov:

- **Diagramy tried (Class diagrams) –** používajú triedy a rozhrania (interfaces) na znázornenie detailov o entitách vytvárajúcich systém a na zobrazenie statických vzťahov medzi nimi. Diagramy tried patria medzi najpoužívanejšie UML diagramy.

- **Diagramy komponentov (Component diagrams) –** znázorňujú časti softvéru v implementačnom prostredí. Tam, kde diagramy tried a balíkov modelujú logický návrh softvéru, diagramy komponentov modelujú implementačný pohľad.

- **Diagramy zloženej štruktúry (Composite structure diagrams) –** koncepčne spájajú diagramy tried a komponentov. Ukazujú, ako elementy v systéme spolupracujú na uskutočnení komplexných cieľov, odhaľujú návrh komplikovaných komponentov a takisto rozhranie do komponentov oddelene od ich štruktúry.

- **Diagramy rozmiestnenia (Deployment diagrams) –** znázorňujú, ako je systém realizovaný a priradený k rozličnému typu hardvéru. Obvykle sa používajú na zobrazenie toho, ako sú komponenty nastavené počas behu systému.

- **Diagramy balíkov (Package diagrams) –** sú v podstate špeciálnym prípadom diagramov tried. Používajú tú istú notáciu, ale ich zameriavajú sa skôr na to, ako sú triedy a rozhrania spolu zoskupené.

- **Diagramy objektov (Object diagrams) –** používajú tú istú syntax ako diagramy tried a zobrazujú, ako spolu súvisia konkrétne inštancie tried v špecifickom časovom okamihu. Môžu sa použiť na zachytenie vzťahov v systéme počas behu.

3.2.2 Diagramy správania

Diagramy správania sa zameriavajú na správanie elementov v systéme. UML 2.0 definuje 7 diagramov správania:

- **Diagramy aktivít (Activity diagrams) –** pripomína vývojový diagram. Tvorí pohľad na tok činností v modelovanom systéme. Spravidla obsahuje počiatočný aj koncový stav systému, rozhodovacie, synchronizačné a prechodové prvky, ktoré opisujú činnosť v modelovanom systéme. Môže však obsahovať aj iné útvary a spojky, ktoré pomáhajú pochopiť opisovanú činnosť

- **Diagramy prípadov použitia (Use case diagrams) –** zachytávajú funkčné požiadavky na systém. Poskytujú pohľad nezávislý od implementácie na to, čo by systém mal robiť a umožňujú tvorcom systému zamerať sa viac na potreby užívateľov, ako na detaily týkajúce sa samotnej realizácie.

- **Stavové diagramy (State machine diagrams) –** zachytávajú vnútorné zmeny stavu nejakého elementu. Element môže byť malý ako jedna trieda, alebo veľký ako celý systém.

- **Diagramy interakcií (Interaction diagrams) -** V predchádzajúcej verzii jazyka UML sa výraz *diagramy interakcií* týkal sekvenčných diagramov (sequence diagrams) a diagramov spolupráce (collaboration diagrams). Tieto dva typy diagramov popisovali komunikáciu medzi objektmi za účelom splnenia istej úlohy, napr. zadanie objednávky. V UML 2.0 výraz *diagramy interakcií* zahŕňa sekvenčné diagramy, diagramy komunikácie, diagramy prehľadu interakcií a časové diagramy. Diagramy spolupráce boli nahradené diagramami komunikácie, ktoré sú

ich trošku jednoduchšou verziou. Každý z týchto štyroch typov diagramov reprezentuje určitý aspekt komunikácie medzi objektmi a poskytuje osobitý pohľad na túto komunikáciu.

- **Sekvenčné diagramy (Sequence diagrams) -** zdôrazňujú typ a poradie správ posielané medzi objektmi počas realizácie správania. Sú najbežnejším typom diagramov interakcií a sú veľmi intuitívne pre nových používateľov UML.

- **Diagramy komunikácie (Communication diagrams) –** zameriavajú sa na objekty zúčastňujúce sa určitého správania a ich štruktúru, a takisto si všímajú správy, ktoré si medzi sebou posielajú.

- **Diagramy prehľadu interakcií (Interaction overview diagrams) –** sú určitou verziou diagramov aktivít, avšak namiesto zdôrazňovania aktivity v každom kroku sa zameriavajú na to, ktoré elementy participujú na vykonávaní danej aktivity.

- **Časové diagramy (Timing diagrams) –** zameriavajú sa na podrobné časové špecifikácie pre správy. Majú špecifickú notáciu na zaznamenanie toho, ako dlho má systém spracovávať a odpovedať na správy. Často sa používajú v real-time systémoch.

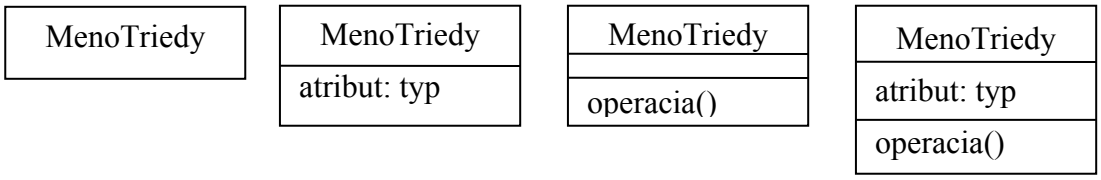
3.3 Diagram tried

Tvorí pohľad na niektoré, alebo aj všetky triedy modelu. Obsahuje prvky opisujúce triedy, rozhrania a ich vzťahy. Trieda môže obsahovať atribúty a operácie. Rozhranie môže tiež obsahovať atribúty. Diagram tried sa používa pre zachytenie statických aspektov systému. Obyčajne odráža funkčné požiadavky na systém. Tieto spravidla predstavujú služby, ktoré systém ponúka používateľovi. Triedy sa môžu sústreďovať do balíkov. Hlavný diagram každého balíka ukazuje jeho verejné triedy. Potom sa vytvárajú diagramy tried, ktoré opisujú implementáciu, štruktúru a správanie jednej alebo aj viacerých tried,

ako aj ich hierarchiu. Diagramy tried sú často základom komponentových diagramov a diagramov rozmiestnenia.

3.3.1 Trieda (Class)

Je kategória, alebo skupina vecí, ktoré majú podobné vlastnosti a rovnaké alebo podobné správanie. Správanie vecí v triede popisujú špecifické operácie (metódy). Trieda sa reprezentuje v jazyku UML obdĺžnikom, ktorý môže byť rozdelený do troch častí. Horná časť obsahuje meno triedy prostredná časť obsahuje vlastnosti a dolná časť obsahuje operácie. Meno triedy sa zapisuje s prvým veľkým písmenom, ak je meno triedy tvorené dvoma a viac slovami, potom sa spoja a každé ďalšie slovo sa zapíše s veľkým počiatočným písmenom. Príklady zobrazenia triedy sú na obrázku 2.



Obrázok 2 – *Rôzne spôsoby zobrazenia triedy*

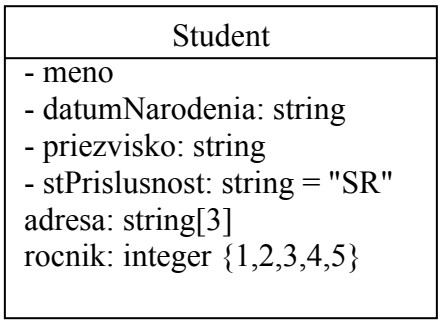
Atribúty

Atribút je vlastnosť triedy, popisujúci rozsah hodnôt, ktoré môžu nadobúdať objekty (inštancie) danej triedy. Trieda nemusí mať žiadny atribút, ale tiež ich môže mať veľa. Atribúty sa v triede zobrazujú pod názvom triedy. Meno atribútu sa zapisuje malými písmami, ak je viacslovné, potom prvé písmeno každého ďalšieho slova sa zapisuje veľkým písmenom. V jazyku UML sa k atribútom môžu pridať ďalšie informácie, a to v ikone pre triedu sa môže špecifikovať typ hodnoty každého atribútu. Atribút môže byť typu string - reťazec, float – reálne číslo, int - celé číslo, bool - logický typ a iné. K atribútom sa typ pridáva tak, že za meno atribútu sa pripíše dvojbodka a potom sa zapíše typ. Za typom môže byť určená násobnosť atribútu. Násobnosť sa udáva v hranatých zátvorkách. Atribútu môžeme tiež priradiť implicitnú hodnotu tak, že za znamienko "= " sa zapíše táto implicitná hodnota. Nakoniec sa v zložených zátvorkách môžu uviesť ďalšie vlastnosti atribútu napr.

{read-only} hodnotu tohto atribútu nie je možné meniť. Syntax zobrazovania atribútov je teda nasledovná:

<<stereotyp>> viditeľnosť / meno :typ násobnosť =implicitná-hodnota {property-string}

Rôzne spôsoby zobrazenia atribútov v triede sú na obrázku 3.



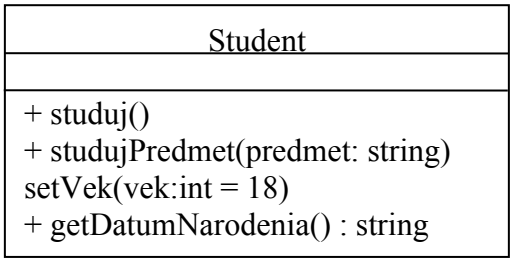
Obrázok 3– *Rôzne zobrazovania atribútov*

Operácie

Trieda môže zobrazovať aj svoje metódy, v UML nazývané operácie. Operácie sú v triede zobrazovaná v tretej časti triedy. Každá operácia by mala byť reprezentovaná aspoň svojim menom. U operácií rovnako ako u atribútov môžeme uvádzať ďalšie údaje. Do zátvoriek, ktoré nasledujú za menom operácie, môžeme zapísať parametre, s ktorými konkrétna metóda pracuje a typ tohto parametru. Potom môže nasledovať návratový typ metódy, ktorý sa zapisuje za dvojbodkou. Nakoniec taktiež ako u atribútov je možné špecifikovať ďalšie vlastnosti. Operácie triedy sa v UML znázorňujú tak, že text operácie je podčiarknutý. Syntax zápisu operácii je :

<<stereotyp>> viditeľnosť meno (názovArgumentu: typArgumentu= základná hodnota, ...) : návratový-typ {property-string}

Rôzne spôsoby zobrazenia atribútov v triede sú na obrázku 4.



Obrázok 4– *Rôzne zobrazenia operácií*

Viditeľnosť

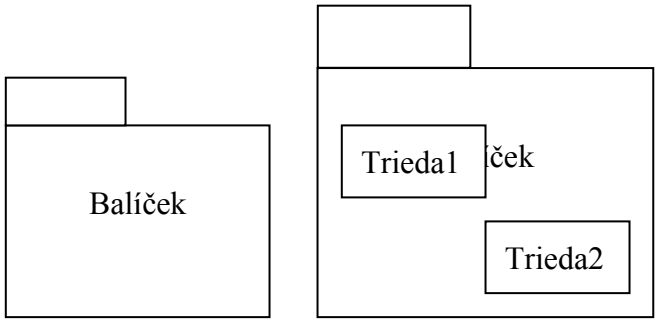
Viditeľnosť sa vzťahuje k atribútom a metódam triedy. Určuje ako môžu ostatné triedy používať atribúty a metódy danej triedy alebo rozhrania. Viditeľnosť sa v UML zobrazuje tak, že pred atribútom alebo metódou je zobrazený symbol príslušného typu viditeľnosti.

Viditeľnosť môže nadobúdať jednu z troch hodnôt, každá z nich sa zobrazuje pomocou špeciálnych symbolov:

- Symbol "+" označuje atribút alebo metódu, ktorá má viditeľnosť typu verejne-prístupný (public), znamená to že, atribút alebo metódu je „vidieť“ z každej inej triedy.
- Symbol "#" označuje atribút alebo metódu, ktorá má viditeľnosť typu chránený (protected), k atribútu alebo metóde potom majú prístup iba potomkovia danej triedy, ostatným triedam nie sú atribúty ani metódy prístupné.
- Symbol "-" označuje atribút alebo metódu, ktorá má viditeľnosť typu privátna (private). Atribúty alebo metódy môže používať jedine pôvodná trieda
- Symbol "~" označuje atribút alebo metódu, ktorá má viditeľnosť typu balíček (package). Členy triedy, ktoré majú tento typ viditeľnosti môžu používať len elementy v tom istom balíčku, alebo elementy z vnorených balíčkov.

3.3.2 Balíček (Package)

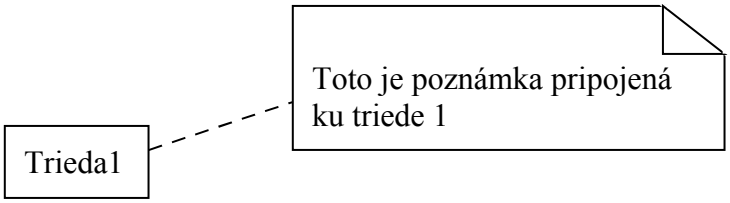
Ďalší prvok jazyka UML je balíček. Pomocou balíčka sa v jazyku UML dajú usporiadať prvky diagramu do skupín. Ak chcem ukázať, že mnoho tried a komponentov je súčasťou určitého podsystému, potom sa zoskupia do balíčku, ktorý je v UML zobrazený ako zložka zo záložkou. Príklad zobrazenia balíčku je na obrázku 5.



Obrázok 5– *Rôzne spôsoby zobrazenia balíčkov*

3.3.3 Poznámka (Note)

Často sa stáva, že niektorá časť diagramu jednoznačne nevysvetľuje svoj účel, alebo neobsahuje pokyn ako s ňou pracovať. V takomto prípade je užitočná poznámka jazyka UML. Poznámku si môžeme predstaviť ako grafický ekvivalent žltého nalepovacieho lístku. Jej ikonou je obdĺžnik s ohnutým rohom. Vo vnútri obdĺžniku je vysvetľujúci text. Poznámka je pripojená k prvku diagramu pomocou prerušovanej čiary.



Obrázok 6– *Poznámka*

3.3.4 Vzťahy v diagrame tried

Vzťah generalizácie

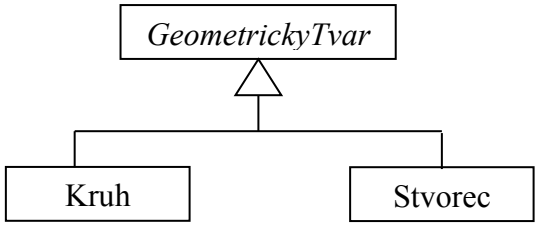
V UML sa dá zobrazovať princíp dedičnosti. Jedna trieda (potomok, odvodená trieda alebo taktiež podtrieda) dedí atribúty a metódy po inej triede (rodičovskej triede alebo základnej - bázovej triede). Potomok dedí od rodiča všetko, ale môže mať aj svoje vlastné atribúty a metódy, ktoré rodičovská trieda nemala, preto je rodičovská trieda všeobecnejšia ako trieda z nej odvodená. Všade, kde sa vyskytuje rodič, môže sa vyskytovať aj potomok. Naopak to neplatí. Hierarchia dedičnosti v tomto nekončí. Potomok jednej triedy môže byť rodičom inej triedy.

V UML sa dedičnosť znázorňuje tak, že spojíme potomka s rodičovskou triedou čiarou, ktorá pri rodičovskej triede bude končiť nevyfarbeným trojuholníkom, ktorý ukazuje na rodičovskú triedu. Príklad kreslenia dedičnosti ja na obrázku 7.

Trieda, ktorá nie je odvodená od žiadnej inej triedy je „bázová trieda“. Ak neobsahuje trieda žiadnych potomkov, potom sa jedná o tzv. listovú triedu. Rozlišujeme taktiež medzi jednoduchou a viacnásobnou dedičnosťou, ak má trieda iba jedného rodiča je to dedičnosť jednoduchá, ak má rodičov viac, potom je to viacnásobná dedičnosť.

Ak budeme zobrazovať v ikonách tried taktiež atribúty a metódy, potom metódy a atribúty, ktoré obsahuje rodičovská trieda a dedia ich potomkovia, sa zapisujú iba v rodičovskej triede.

Dôležitým pojmom je abstraktná trieda. Je to taká trieda, ktorá „nemá objekty“. Abstraktnú triedu zobrazíme tak, že meno tejto triedy napíšeme kurzívou. Na obrázku 7 je to trieda *GeometrickyTvar*.



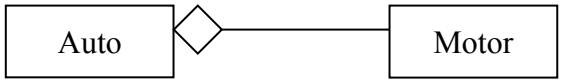
Obrázok 7– *Príklad zobrazenia dedičnosti*

Vzťah asociácie

Asociácie sa zakresľujú medzi triedami, ale reprezentujú vzťahy medzi objektmi. Úloha objektu vo vzťahu asociácie môže byť špecifikovaná na konci asociácie, ako aj na multiciplite (počet objektov, ktoré sa zúčastňujú asociácie).

Vzťah agregácie

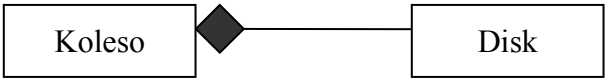
Ak triedu (celok) vytvára jedna, alebo viac tried (komponent) hovoríme o agregácii. Agregácia je zvláštnou formou asociácie a v UML sa znázorňuje ako čiara, idúca od triedy komponentu k celku pričom pri celku končí nevyfarbeným kosoštvorcom.



Obrázok 8– *Vzťah agregácie*

Vzťah kompozície

Kompozícia je silnejším typom agregácie. Každý komponent v sklade môže patriť iba jednému celku. Zvláštnosť vzťahu medzi celkom a časťou spočíva v tom, že časť nemôže byť bez celku. So zničením celku sa zničia aj všetky jeho súčasti. Kompozícia sa zobrazuje tak ako agregácia, ale s vyplneným kosoštvorcom pri celku.



Obrázok 9– *Vzťah kompozície*

4 Automatické generovanie zdrojového kódu

Pod pojmom automatické generovanie kódu mám na mysli prepis modelu jazyka UML do niektorého objektovo orientovaného programovacieho jazyka. Generátorom zdrojového kódu sa nazýva aplikácia, alebo jej časť, ktorá robí automatické generovanie kódu.

Je zrejmé, že žiadny generátor nikdy nevráti ako výsledok kompletný zdrojový kód, ktorý bude presne zodpovedať navrhovanému systému. Vždy sa bude jednať iba o akúsi kostru kódu, pričom zostávajúcu časť je nutné doplniť manuálne.

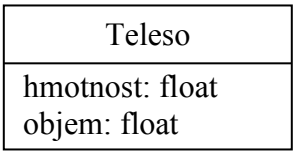
Automatické generovanie kódu nám prináša niektoré výhody:

- Nemusíme celý kód písať ručne, čo značne ušetrí čas programátora.
- Objektové prvky (napr. triedy, balíky atď.) UML zodpovedajú objektovým prvkom programovacieho jazyka generovaného kódu.
- Z vygenerovaného kódu je možné zistiť chyby v návrhu systému.

Hlavnou nevýhodou automatického generovania kódu je skutočnosť, že výstup nebude vždy taký aký by sme si predstavovali a niektoré časti okrem tiel metód musíme doplniť, alebo niekedy aj opraviť ručne. V tomto prípade záleží hlavne na kvalite a flexibilite generátora kódu.

4.1 Metódy automatického generovania zdrojového kódu

V tejto časti si popíšeme niektoré metódy automatického generovania zdrojového kódu z UML diagramov a ku všetkým metódam si ukážeme aj príklad generovania. Príklad bude ukázaný na jednoduchej triede *Teleso* s atribútmi *hmotnosť* a *objem*. Vygenerovaný zdrojový kód bude v jazyku C++.



Obrázok 10 - *Trieda Teleso v UML*

4.1.1 Generovanie kódu pomocou šablón a filtrov

Metóda šablón a filtrov je základná metóda generovania zdrojového kódu. Princíp tejto metódy je nasledujúci:

1. Pomocou šablón (napr. vo formáte XML) sa definuje predpis pre generovanie časti modelu rovnakého typu (napr. tried, balíkov, metód apod.)
2. Z modelu sa najskôr vyberú dôležité časti vhodné na generovanie kódu a prevedú sa do vhodného textového formátu napr. XML
3. Na zjednodušený model v textovom formáte sa potom aplikujú vytvorené šablóny čím sa vygeneruje časť zdrojového kódu.

Príklad:

1. Definujeme šablónu pre generovanie kódu (tučne je zvýraznená pevná časť generovaného kódu)

```
<template match="class">
   class <value-of select="@name" /> {
   <apply-templates select="attribute" />
   }
</template>
< template match="attribute">
   <value-of select="@type" />
   < value-of select="@name" />;
</template>
```

2. Prevedieme model do textového formátu XML:

```
<class name="Teleso">
      <attribute type = "float" name="hmotnost"/>
      <attribute type = "float" name="objem"/>
</class>
```

3. Z textového popisu modelu vygenerujeme podľa šablóny nasledujúci zdrojový kód

```
class Teleso{
    float hmotnost;
    float objem;
}
```

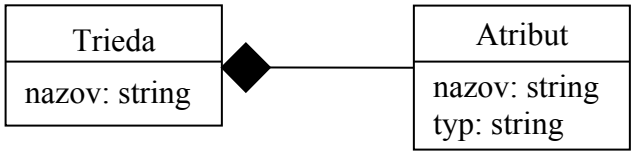
4.1.2 Generovanie kódu pomocou šablón a metamodelu

Táto metóda je istou modifikáciou generovania kódu pomocou šablón a filtrov a je založená na nasledujúcom princípe:

1. Definuje sa metamodel tak, aby bol spoločný pre všetky modely, z ktorých chceme generovať kód.
2. Rovnako ako pri predchádzajúcej metóde je nutné vytvoriť potrebné šablóny ku generovaniu kódu.
3. Na metamodel sa aplikujú definované šablóny za účelom vygenerovania kódu.

Príklad:

1. Definujeme metamodel



Obrázok 11 - *Metamodel triedy*

2. Definujeme šablónu pre generovanie kódu:

```
<? FOREACH Class AS c {?>
class <?c.nazev?> {
    <?FOREACH Attribute AS a { ?>
    private <?a.type?> <a.name?>;
    <? } ?>
}
<? } ?>
```

3. Výsledkom generovania bude nasledujúci zdrojový kód

```
class Teleso{
    float hmotnost;
    float objem;
}
```

4.1.3 Generovanie kódu pomocou generatívnych gramatík

Jedná sa o metódu vychádzajúcu z teórie formálnych jazykov. Programovacie jazyky a jazyk UML sú takzvané bezkontextové jazyky. Tieto sú generované bezkontextovými gramatikami, ktoré sa skladajú z nasledujúcich elementov:

- Množina terminálnych symbolov (terminálov) – obsahuje symboly, ktoré sú konečné, čiže nie je možné z nich odvádzať ďalšie slová. Označujú sa veľkými písmenami (A, B, C...)

- Množina neterminálnych symbolov (neterminálov) – obsahuje symboly, z ktorých je možné odvádzať ďalšie slová pozostávajúce z terminálov, alebo neterminálov. Označujú sa malými písmenami (a, b, c...)

- Inicializačný symbol – neterminálny symbol, z ktorého sú odvodené všetky generované slová. Označuje sa písmenom I

- Množina prepisovacích pravidiel – jedná sa o množinu usporiadaných dvojíc, ktoré pozostávajú z neterminálov a slova zo symbolov obidvoch abecied. Slová sa označujú gréckymi písmenami.

Princíp generovania kódu pomocou generatívnych gramatík je nasledujúci:

1. Definujeme gramatiku G jazyka UML tak, aby generovala jeho textovú formu (viď. predchádzajúce dve metódy generovania kódu).
2. Definujeme gramatiku H výstupného programovacieho jazyka.
3. Vhodným spôsobom definujeme gramatiku T, ktorá skombinuje gramatiky G a H. Inicializačný symbol bude z gramatiky G a generátor bude generovať slová z gramatiky H.

Príklad:

1. Definujeme zjednodušenú generatívnu gramatiku G triedy jazyka UML, respektíve jeho textovej podoby vo formáte XML, s nasledujúcimi prepisovacími pravidlami:

I	►	UMLClass
UMLClass	►	< **class name** = " UMLClassName " > < / **class**>
UMLClass	►	< **class name**=" UMLClassName " > UMLAttributes < / **class** >
UMLAttributes	►	UMLAttribute
UMLAttributes	►	UMLAttribute UMLAttributes
UMLAttribute	►	< **attribute type** = " UMLAttributeType " name = " UMLAttributeName " / >
UMLClassName	►	UMLIdentifier
UMLAttributeType	►	UMLIdentifier
UMLAttributeNam	►	UMLIdentifier

   Terminály sú zvýraznené tučne, ostatné symboly sú neterminály. **I** je inicializačný symbol. UMLIdentifier predstavuje ľubovoľný identifikátor.

2. Definujeme zjednodušenú generatívnu gramatiku H triedy jazyka C++, s nasledujúcimi prepisovacími pravidlami:

I’	►	CppClass
CppClass	►	**class** CppClassName { }
CppClass	►	**class** CppClassName { CppAttributes }
CppAttributes	►	CppAttribute
CppAttributes	►	CppAttribute CppAttributes
CppAttribute	►	CppAttributeType CppAttributeName **;**
CppClassName	►	CppIdentifier
CppAttributeType	►	CppIdentifier
CppAttributeName	►	CppIdentifier;

Terminály sú zvýraznené tučne, ostatné symboly sú neterminály. **I'** je inicializačný symbol. CppIdentifier predstavuje ľubovoľný identifikátor.

3. Definujeme gramatiku T, ktorá bude mať inicializačný symbol I gramatiky G a bude generovať slová z gramatiky H. Gramatika T bude mať nasledujúce prepisovacie pravidlá:

I	►	UMLClass
UMLClass	►	**class** UMLClassName { }
UMLClass	►	**class** UMLClassName { UMLAttributes }
UMLAttributes	►	UMLAttribute
UMLAttributes	►	UMLAttribute UMLAttributes
UMLAttribute	►	UMLAttributeType UMLAttributeName **;**
UMLClassName	►	UMLIdentifier
UMLAttributeType	►	UMLIdentifier
UMLAttributeNam	►	UMLIdentifier

Gramatika T obsahuje neterminály z gramatiky G a terminály z gramatiky H. To nám zaistí spoľahlivý preklad z jazyka UML do jazyka C++.

4. Výsledkom generovania bude nasledujúci zdrojový kód

```
class Teleso{
    float hmotnost;
    float objem;
}
```

5 XML

Keďže generovanie kódu aj dokumentácie je robené pomocou XML šablón, tak si teraz v krátkosti povieme niečo o XML.

5.1 História XML

XML (eXtensible Markup Language) je jazyk, ktorý slúži pre popis údajov a popis ich štruktúry. Bol prijatý ako odporúčanie W3C (World Wide Web Consortium) 10. februára 1998. XML bol vyvinutý hlavne pre výmenu informácií medzi systémami. Dokument XML je textový súbor, ktorého štruktúru si môžeme predstaviť ako dobre uzátvorkovaný výraz. Zátvorkami sú v tomto prípade párové značky, ktoré môžu byť do seba vnorené. Tieto značky nie sú predpísané štandardom XML, značkou môže byť ľubovoľný neprázdny textový reťazec. Vďaka tomuto formátu je dokument XML ľahko prenositeľný medzi rôznymi platformami.

Na definovanie typu dokumentu XML sa používajú takzvané schémy. Schéma presne popisuje obmedzenia štruktúry dokumentu XML. Definuje možné značky, ich typy a vzájomnú organizáciu v rámci dokumentu XML. Schéma akoby zavádzala poriadok do voľnej štruktúry. Existuje niekoľko jazykov schém, medzi najpoužívanejšie patria DTD (Document Type Definition) a schéma XML (XML Schema).

5.2 Dátový model XML

Dátový model vychádza z DOM (Document Object Model), ktorý popisuje štandardy štruktúry dokumentu. XML predstavuje strom pozostávajúci z uzlov (nodes). Rozlišujú sa tieto základne typy uzlov:

- Koreň (root node) - je uzol najvyššej úrovne.
- Elementy (element nodes) - majú vlastný názov. Ich obsahom je usporiadaný zoznam potomkov tvorený z elementov alebo textových uzlov. Elementy môžu

obsahovať neusporiadaný zoznam atribútov. Štruktúra XML je rekurzívna to znamená, že každý element spolu s jeho obsahom a atribútmi tvoria (pod)strom.

- Textové uzly (text nodes) - tiež nazývané aj dátové uzly (data nodes) neobsahujú žiadnych potomkov ani atribúty. Zjednodušene sa dá povedať, že je to všetko čo nie je medzi ”<“ a ”>“ (s výnimkou niektorých oddeľovačov riadkov).
- Atribúty (attribute nodes) - majú svoj názov, typ a hodnotu. Hodnota je buď atomická, alebo je to množina atomických hodnôt.
- Inštrukcie (processing instruction) - obsahujú informácie pre aplikácie. Sú tvaru <? . . . ?>.
- Komentár (comment) - má tvar <!-- . . . -->.

5.3 Správna štruktúra XML dokumentov

XML-dokument je podľa [2] správne štruktúrovaný, ak spĺňa tieto podmienky:

- Obsahuje aspoň jeden element.
- Existuje pravé jeden element, nazývaný koreň, v ktorom je obsiahnutý celý dokument. Každý element začína tzv. začiatočným tagom a končí tzv. ukončovacím tagom, alebo je to element bez obsahu. Napríklad element s názvom jazyk, obsahujúci text (textový uzol) slovensky vyzerá takto:

```
<jazyk>slovensky</jazyk>
```

- Ak vynecháme textový obsah nášho elementu s názvom jazyk dostaneme:

```
<jazyk></jazyk>
```

- Ekvivalentný skrátený zápis takéhoto elementu je:

```
<jazyk />
```

- Obsahy elementov možno vnárať, no nesmú sa prekrývať, t. j. počiatočný a koncový tag každého elementu sú súčasťou obsahu toho istého elementu a žiadneho iného.

6 UML.FRI

UML.FRI je modelovací nástroj jazyka UML vyvíjaný na Fakulte riadenia a informatiky Žilinskej Univerzity. Na vývoji tohto UML case nástroja sa zatiaľ pracuje len dva roky a už sa svojou funkcionalitou priblížil a dokonca aj predbehol iné nástroje vyvíjané dlhšiu dobu. UML.FRI je implementovaný v programovacom jazyku Python.

V UML.FRI sú elementy, spojenia, ale aj diagramy vytvárané na základe definícií, ktoré sú uložené vo formáte XML. Tieto definície obsahujú okrem vzhľadu UML elementu aj jeho vlastnosti.

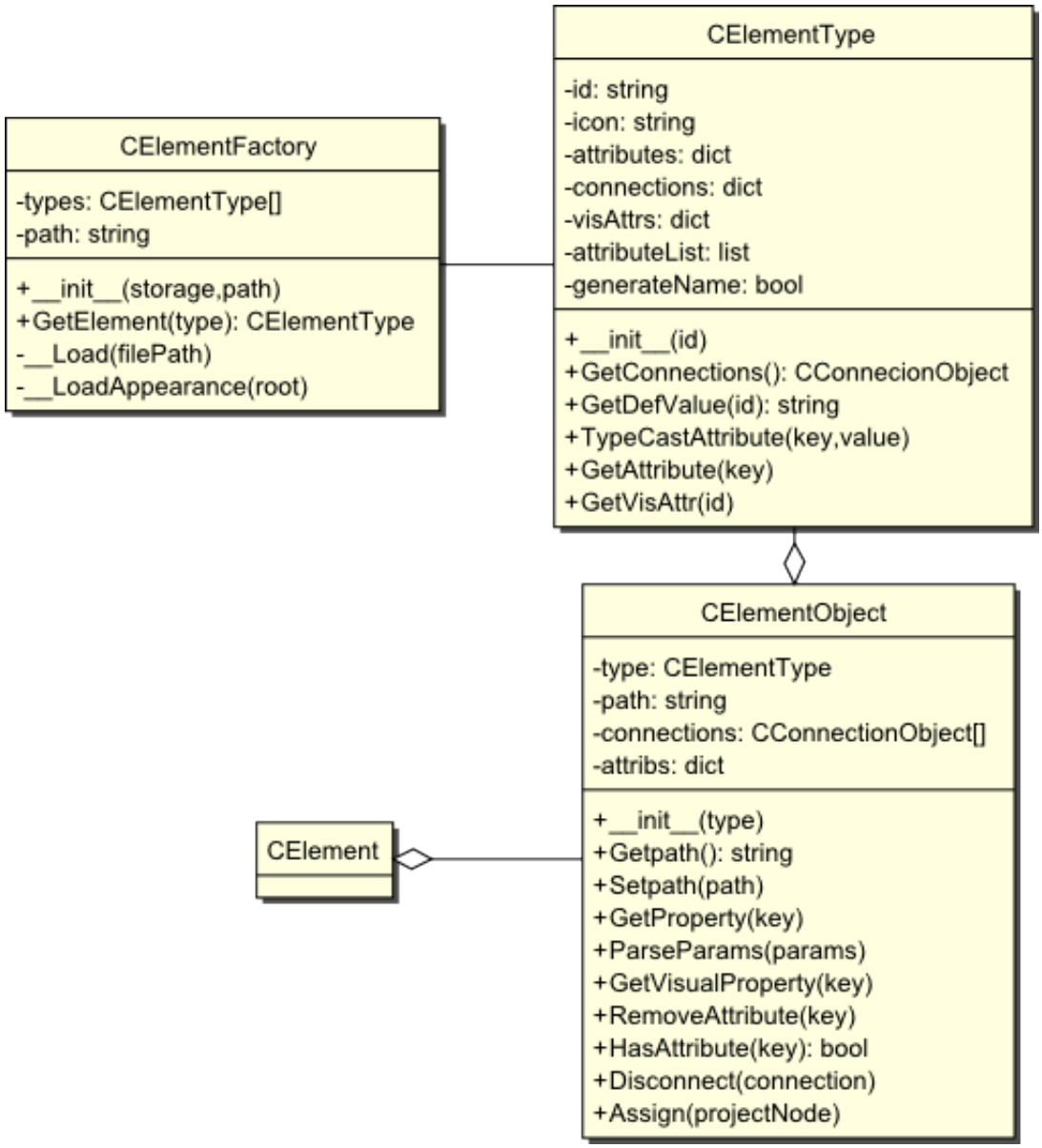
V nasledujúcich podkapitolách si rozoberieme tie časí UML.FRI, ktoré sú potrebné pre generovanie kódu a dokumentácie.

6.1 Popis UML elementov

Definície UML elementov sú v súboroch, ktorých názov je rovnaký ako názov UML elementu. Všetky elementy obsahujú vizuálne (vlastnosti, ktoré sa vykresľujú pri kreslení elementu napr. operácie pri triedach) a nevizuálne (nie sú zobrazované v ikone elementu napr. či má atribút property) vlastnosti. Tieto XML súbory parsuje trieda CElementFactory. Trieda CElementType obsahuje informácie získané z rozparsovania XML súboru a Trieda CElementObject je logickou vrstvou daného UML elementu. Jeho grafickou vrstvou je trieda CElement. Vzťahy medzi týmito triedami sú vyjadrené pomocou UML diagramu na obrázku 12. Tieto diagramy nemajú vyplnené všetky atribúty a metódy, ale úplné diagramy sú na priloženom CD.

Všetky hodnoty vlastností generovaných UML elementov sú získavané práve z triedy CElementObject. Slúžia na to nasledujúce hodnoty:
GetProperty je metóda, ktorá vracia hodnoty vlastnosti UML elementu podľa kľúča daného ako parameter tejto metódy.

ParseParams táto metóda preberá ako parameter parametre generovanej operácie a snaží sa ich rozparsovať a vytvoriť kolekciu, cez ktorú by mohol prebehnúť cyklus. Parsovanie prebieha nasledujúcim spôsobom. Najskôr rozdelíme reťazec na parametre podľa čiarok. Keď už máme jednotlivé parametre oddelené môžeme získať názov parametrov rozdelením reťazca podľa znaku ":", alebo "=". Ak existuje ":", potom za ňou nasleduje typ parametra, ktorý môže obsahovať aj implicitnú hodnotu parametra. Ak bol len znak "=", potom za ním nasleduje implicitná hodnota parametra.

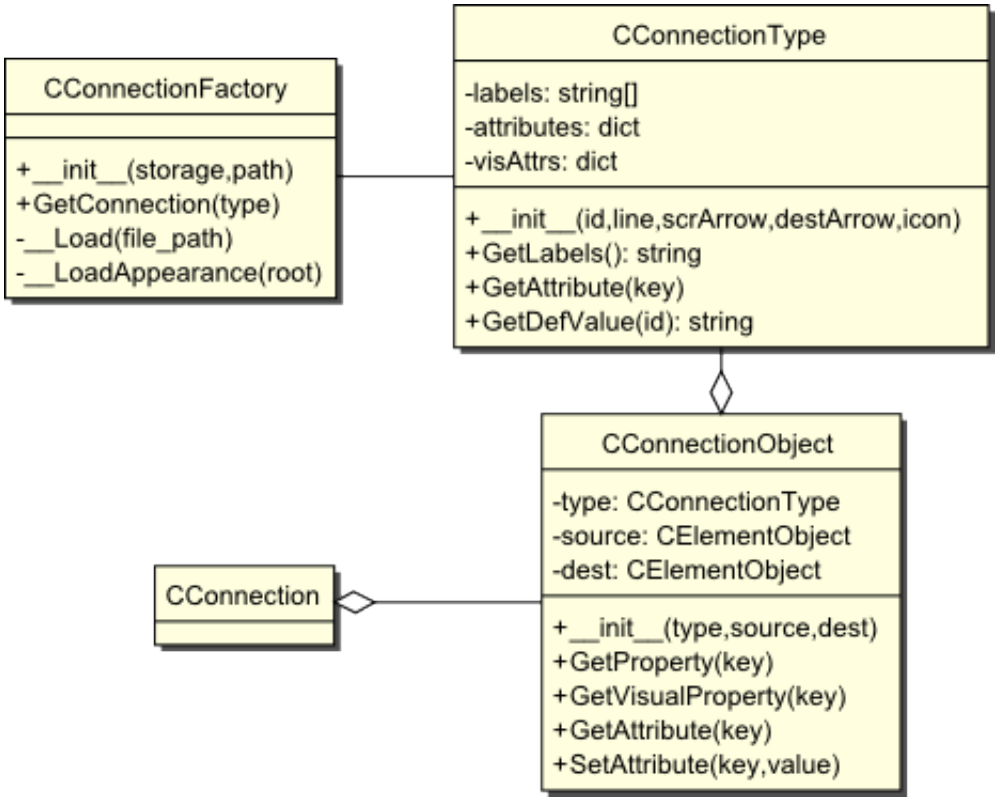


Obrázok 12 - *UML diagram elementov v UML.FRI*

6.2 Popis spojení

Tak ako sú definované UML elementy definované v súboroch vo formáte XML, sú definované aj spojenia medzi UML elementmi. Spojenia majú tak ako elementy triedu na čítanie definície z XML súboru CConnectionFactory, triedu obsahujúcu typ spojenia CConnectionType, triedu, ktorá si uchováva logickú časť spojenia CConnectionObject a triedu CConnection pre konkrétne grafické spojenie vykreslené v diagrame. Zobrazenie vzťahov medzi týmito triedami je na obrázku 13. Tak ako u elementov nemajú zobrazené všetky metódy a atribúty. Úplný diagram je na priloženom CD.

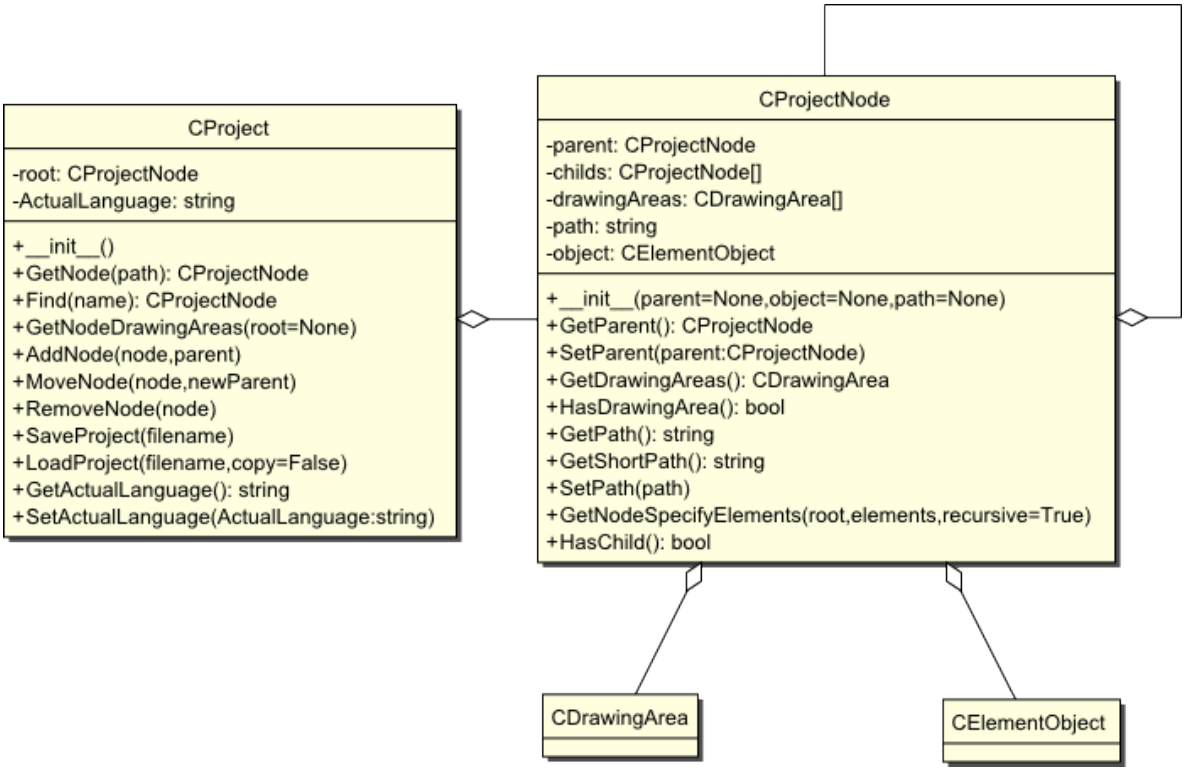
Zaujímavými metódami pre generovanie kódu, alebo aj dokumentácie sú GetProperty, ktorá ako aj u atribútov vracia hodnotu vlastnosti zadanej ako parameter metódy. Inštancia triedy CConnectionObject obsahuje aj referencie na počiatočný UML element a koncový UML element spojenia, tieto sú využívané napríklad pri generovaní dedičnosti.



Obrázok 13 - *UML diagram spojení v UML.FRI*

6.3 Strom projektu

Je uloženie vytvorených UML elementov a diagramov v pamäti. Využitie tohto stromu je pri ukladaní projektu do súboru, ale aj pri generovaní dokumentácie, keď sa musia popísať všetky vytvorené UML elementy. Z programátorského hľadiska je tento strom rozdelený na vizuálnu a logickú vrstvu. Logická vrstva sa skladá s triedy *CProject* a *CProjectNode* čo sú uzly tohto stromu. UML diagram je zobrazený na obrázku 14. Na obrázku nie sú zobrazené všetky metódy zobrazených tried. Sú tam hlavne tie podstatné pre generovanie kódu a dokumentácie. Úplné UML diagramy sú na priloženom CD.



Obrázok 14 – *UML diagram stromu projektu v UML.FRI*

Ako je vidieť z obrázku trieda *CProject* obsahuje atribút root, čo je koreňový uzol projektu a ešte zo zaujímavých atribútov obsahuje ActualLanguage, čo je programovací jazyk v ktorom bude projekt navrhovaný. Tento jazyk je nastavený aj ako predvolený pri generovaní kódu. Ďalšou zaujímavou metódou je *GetNodeDrawingAreas,* ktorá vráti

všetky diagramy uzla daného ako parameter metódy a vráti aj všetky diagramy jeho potomkov. Metóda sa využíva pri generovaní dokumentácie na export obrázkov do nejakého grafického formátu. Momentálny podporovaný formát je SVG.

Trieda *CProjectNode* vytvára uzly z ktorých je zložený strom. Strom je obojsmerne zreťazený, keďže uzly obsahujú aj odkazy na svojich rodičov. Ďalej každý uzol obsahuje zoznam svojich synov a zoznam diagramov. Základom uzlov je *element object* čo je logická časť UML elementov. Posledným atribútom je *path.* Cesta uzla v strome je udávaná v nasledujúcom formáte :

```
názov elementu: typ elementu / názov elementu: typ elementu
```

Ak je elementom diagram tak je to vždy element, ktorý je na konci cesty a ako typ elementu je použitý konštantný reťazec "=DrawingArea=". Metóda *GetShortPath* vráti cestu k uzlu zloženú iba z názvov elementov bez ich typov.

Z metód je pre generátory zaujímavá metóda *GetNodeSpecifyElements,* ktorá vracia UML elementy uzla zadaného ako parameter *root*. Metóda vracia len UML elementy typu, ktorý je udaný ako parameter *elements*, kde môže byť uvedených aj viac typov. Ako parameter *elements* môže byť zadaná aj konštanta *All*, čo znamená, že zoberie všetky UML elementy daného uzla. Posledným parametrom je *recursive,* ktorý udáva či sa majú rekurzívne prehľadať všetky balíčky.

7 Generovanie kódu

7.1 Šablóny pre generovanie kódu

Vzhľadom na to, že UML.FRI už má implementované vykresľovanie elementov pomocou šablón v textovom formáte XML sme sa rozhodli spraviť aj šablóny pre generovanie kódu vo formáte XML. Pre každý generovaný jazyk je vytvorená jedna šablóna. V tejto práci sme sa zamerali na tri jazyky, ktoré sa od seba odlišujú vo viacerých veciach. Sú to jazyky Python, Delphi a C++. Šablóny sú navrhnuté tak, aby sa z nich okrem generovania kódu dalo urobiť aj generovanie diagramov z kódu. V popise vytvorených elementov sa zameriam len na popis XML elementov týkajúcich sa generovania kódu.

7.2 Elementy šablóny pre generovanie kódu

7.2.1 Template

Je koreňový element každej šablóny, obsahuje dva atribúty:

- diagram – je atribút ktorý uvádza UML diagram
- language – je názov programovacieho jazyka, ktorý bude uvádzaný v dialógoch na generovanie kódu.

7.2.2 Element

XML element, ktorý sa zhodou okolností volá element definuje ako bude vyzerať vygenerovaný kód jednotlivých UML elementov. Zatiaľ máme v šablónach len UML elementy typu trieda a balíček. Obsahuje nasledujúci atribút:

- id – tento atribút určuje pre ktorý UML element je nasledujúci spôsob generovania.

7.2.3 Directory

Vytvára adresár podľa hodnôt nasledujúcich atribútov:

- name – je meno vytvoreného adresára
- value – je regulárny výraz, ktorý určuje ako môže vyzerať meno adresára. Tento výraz je tam kvôli spätnému generovaniu diagramov

7.2.4 Recursive

Tento XML element má na starosti rekurzívne prehľadávanie synov generovaného UML elementu.

7.2.5 AllowElement

Je synom XML elementu Recursive a slúži na vybranie elementu určitého typu. Tento typ ja zadávaný pomocou tohto atribútu:

- id – podľa hodnoty atribútu sa vyberie UML element s týmto typom a začne sa generovanie práve tohto UML elementu. Ak je hodnota atribútu rovná *All* to znamená, že berú všetky UML elementy.

7.2.6 File

Vytvára nový súbor podľa hodnôt svojich atribútov a zapisuje výsledok generovania vnorených XML elementov do tohto súboru.

- name – názov nového súboru
- value – je opäť regulárny výraz ako pri elemente Directory
- prefix – je text, ktorý je vložený pred meno súboru. Tento prefix sa používa napríklad pri vytváraní nového unitu v jazyku delphi
- sufix – pridáva sa na koniec mena súboru. Vo väčšine prípadov je ako sufix zadávaná prípona nového súboru.

7.2.7 Condition

Ak je splnená podmienka môžu sa generovať XML elementy, ktoré sú vnorené práve v tomto XML elemente. V prípade, že podmienka nie je splnená vnorené elementy sa ignorujú. O splnení podmienky rozhodujú nasledujúce atribúty:

- id – je názov atribútu UML elementu, ktorého hodnota sa porovnáva s hodnotou atribútu value.
- value – je hodnota, ktorá musí byť dosiahnutá, aby mohla byť splnená podmienka
- negate – tento atribút nadobúda hodnoty 0, 1. Ak sa neuvedie žiadna hodnota implicitne je to 0. V prípade že tento atribút obsahuje hodnotu 1 výsledok podmienky sa neguje.

7.2.8 Property

Je hodnota atribútu UML elementu. Výsledok elementu property určujú tieto atribúty:

- id – je názov atribútu aktuálneho UML elementu, ktorého hodnota je výsledkom generovania property.
- value – regulárny výraz, pre spätné generovanie diagramov
- newLine – v prípade, že je property generovaný na nový riadok je najskôr pridaný text, ktorý je hodnotou tohto atribútu.
- default – ak by mal byť výsledok property rovný prázdnemu reťazcu, tak sa ako výsledok použije hodnota atribútu default. Tento atribút je implicitne nastavený na prázdny reťazec
- prefix – je text, ktorý je vložený na začiatok výsledného reťazca. Ak sa ako výsledný reťazec použije hodnota atribútu default potom sa prefix nepridáva
- sufix – podobne ako pri atribúte prefix, ale text je pridávaný na koniec výsledného reťazca

7.2.9 Indent

Definuje odsadenie textu všetkých vnorených XML elementov. Typ odsadenia definujú tieto atribúty:

- required – je atribút, ktorý je dôležitý hlavne pre generovanie diagramov jazyka UML z kódu. Nadobúda hodnoty True alebo False. Tento atribút hovorí o tom či je odsadenie potrebné. Napríklad v jazyku python je odsadenie povinné a atribút nadobúda hodnotu True. Pri jazyku C++ povinné nie je, ale pri generovaní kódu slúži na krajší zápis kódu. Implicitne je nastavený na hodnotu False.
- defsize – definuje počet medzier, pomocou ktorých je vytvárané odsadzovanie textov vnorených XML elementov. Implicitne je nastavený na 4 medzery.

7.2.10 Optional

Tento XML element je v rámci generovania kódu podobný XML elementu *Condition* s tým rozdielom, že nechá prebehnúť generovanie vo vnorených XML elementoch a ak je výsledok vnorených XML elementov nepočíta sa do toho výsledok XML elementov w*hitespace, text, br, token* prázdny reťazec tak výsledok generovania vnorených XML elementov sa ignoruje a vracia sa prázdny reťazec. Vymenované XML elementy sa do podmienky nepočítajú preto, lebo ich text je vždy rôzny od prázdneho reťazca. XML element Optional má len jeden atribút:

- default – je hodnota, ktorá sa vráti ako výsledok generovania v prípade, že XML element Optional vracia prázdny reťazec. Implicitne je nastavená na prázdny reťazec.

7.2.11 Alternate

Je veľmi podobný XML elementu *Optional*, s tým rozdielom, že generovanie vnorených XML elementov nastane vtedy ak aspoň jeden z vnorených elementov vracia nejakú hodnotu. Tak ako aj u *Optional* nepočítajú sa hodnoty vrátené týmito XML elementmi: w*hitespace, text, br, token.*

7.2.12 ConnectionLoop

Tento XML element slúži na uskutočnenie cyklu cez všetky spojenia UML elementu. Cez aké spojenia má cyklus prebehnúť určujú tieto atribúty:

- id – určuje hodnota ktorého atribútu UML elementu sa bude porovnávať s hodnotou atribútu value.
- value – porovnáva sa s hodnotou získanou atribútom id. Ak je táto hodnota rovnaká potom sa môžu generovať XML elementy vnorené v ConnectionLoop. Ak je hodnota rovná *All* potom je podmienka automaticky splnená.
- separator – je reťazec, ktorý sa vypíše po ukončení iterácie cyklu. Nevypisuje sa po ukončení poslednej iterácie.

7.2.13 PropertyLoop

Je cyklus cez zadanú vlastnosť UML elementu napr. atribúty, operácie. Tento element podporuje aj vnorené cykly. Atribúty, ktoré nastavujú cyklus sú nasledovné:

- id – udáva cez, ktorú vlastnosť UML elementu bude prebiehať cyklus. Vlastnosť UML elementu cez, ktorú má prebiehať cyklus musí byť kolekcia.
- separator – je reťazec, ktorý je vygenerovaný na konci každej iterácie cyklu. Ak nie je zadaný generuje sa nový riadok.
- parse – uvádza, že vlastnosť UML elementu cez ktorú má prebehnúť cyklus nie je kolekcia a najskôr sa musí použiť nejaká funkcia, ktorá z tejto vlastnosti spraví kolekciu. V našom prípade je to zatiaľ implementované na parametre metód.

7.2.14 Block

Pri generovaní si tento XML element zaznamenáva výsledky generovania vnorených XML elementov a ako výsledok svojho generovania vracia prázdny reťazec ak sa nachádza v rekurzii. Keď sa rekurzia skončí vráti text, ktorý si zaznamenal počas rekurzie.

7.2.15 Whitespace

Keďže v našich šablónach, ktoré sú vo formáte XML nie je možné ako text napísať medzeru, vytvorili sme tento XML element, ktorý sa stará len o vygenerovanie medzery do výstupného súboru.

7.2.16 Br

Podobne ako pri XML elemente *whitespace* nebolo možné zapísať ani prechod na nový riadok, vytvorili sme tento XML element, ktorý sa stará práve o toto. Tento element obsahuje nasledujúce atribúty:

- required – je atribút, ktorý je totožný s atribútom required XML elementu indent s tým rozdielom, že sa nestará o odsadenie, ale o prechod na nový riadok.
- count – je počet vygenerovaných nových riadkov. Implicitne je nastavený na 1.

7.2.17 Text

Nie je XML element uzol , ale je to textový uzol zapísaný v XML. Text sa generuje tak, že sa len zapíše do súboru.

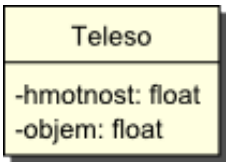
7.2.18 Token

Tento XML element bol pridaný do šablón hlavne kvôli generovaniu diagramov z kódu. Pri generovaní kódu plní tú istú funkcionalitu ako text.

- default – tento atribút obsahuje text, ktorý je potom generovaný.

7.3 Príklad použitia šablón

V tejto podkapitole uvediem príklad ako sa dá s použitím popísaných XML elementov vytvoriť jednoduchá šablóna pre príklad uvedený v kapitole 4. Trieda pre, ktorú bude šablóna definovaná je na obrázku 15.



Obrázok 15 – *Trieda Teleso*

```
<Template diagram="Class diagram" language="C++">
  <Element id="Class">
    <File name="#name" value="[_0-9a-zA-Z]+\.cpp" sufix=".cpp">
      class <whitespace />
      <Property id="#name" value="[a-z0-9A-Z]+" />
      {<br />
      <Indent>
        <PropertyLoop id="attributes">
          <Property id="@type" /> <whitespace />
          <Property id="@name">;
        </PropertyLoop>
      </Indent>
      };
    <File>
  </Element>
</Template>
```

7.4 Generátor kódu

Generuje zdrojové kódy do zvoleného programovacieho jazyka podľa prislúchajúcej šablóny. Ako už bolo vidieť z predchádzajúceho príkladu priklonili sme sa ku metóde generovania kódu pomocou šablón a filtrov. Túto metódu sme trochu upravili tak, že v druhom kroku nevytvárame XML súbor z UML modelu, ale dáta potrebné pre generovanie berieme priamo z tohto modelu.

Generátor je zložený z dvoch častí:

- Časť načítania šablóny z XML súboru. UML diagram tejto časti je na obrázku 16.
- Časť samotného generovania. Táto časť generátor je zobrazená na obrázku 17.

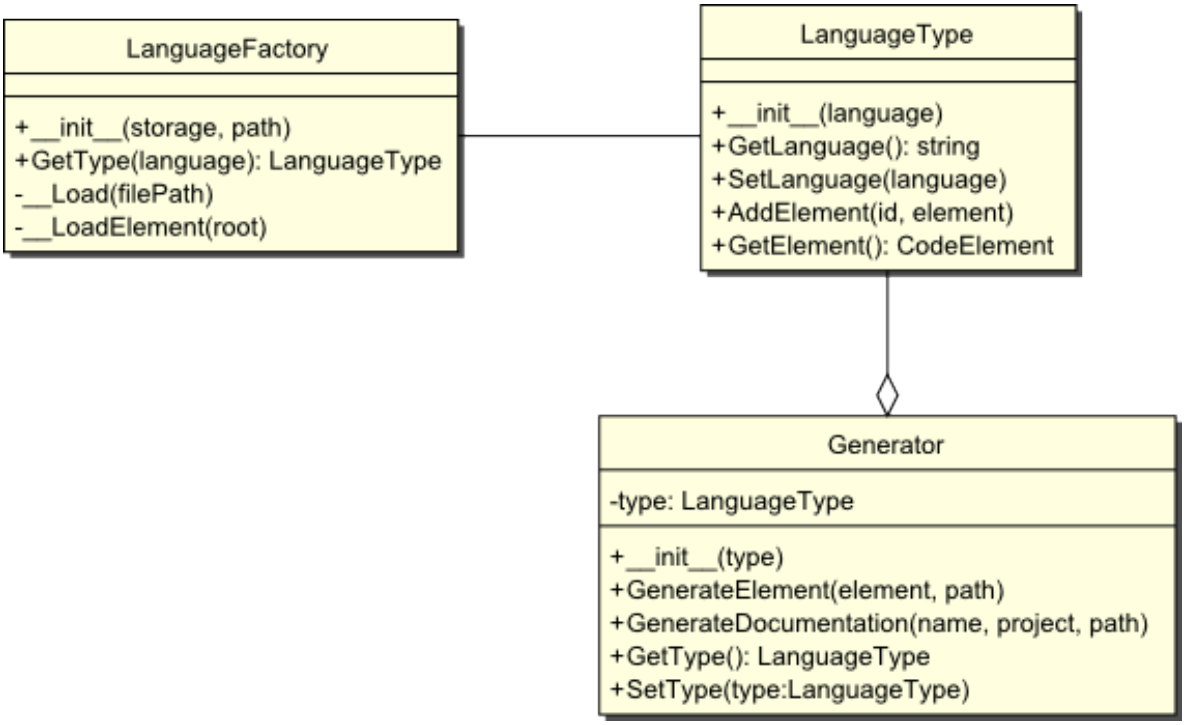
Načítavanie XML šablóny

O načítavanie šablón sa stará trieda LanguageFactory. Pomocou metódy *__Load* sa začne čítať XML súbor v ktorom je šablóna. Po nájdení XML elementu *Template* sa vytvorí inštancia triedy *LanguageType*. Pri nájdení XML elementu *Element* sa zavolá metóda *__LoadElement* a táto metóda rekurzívne prejde cez všetky vnorené XML elementy a vytvorí tak šablónu pre generátor. Po načítaní každého vnoreného XML elementu sa vytvorí inštancia príslušnej triedy. Ak je XML element neznámy vytvorí sa inštancia triedy CodeContainer.

Pre každý programovací jazyk, ktorý má šablónu je vytvorená inštancia triedy *LanguageType*, ktorá si okrem jazyka pamätá celú štruktúru inštancií vytvorených pre generovanie.

Samotné generovanie je však iniciované triedou *Generator*, ktorá má metódy pre generovanie kódu aj pre generovanie dokumentácie. Rozdiel medzi týmito metódami je v tom, že v metóde *GenerateDocumentation* je ešte vytvorený element documentation,

ktorý je potom dosadený do samotného generovania ako parameter element. Parameter path udáva cestu na disku, kde sa má uložiť výsledok generovania



Obrázok 16 – *UML diagram načítavania šablón*

7.4.1 Generovanie kódu

Samotné generovanie je založené na tom, že každá trieda podedí virtuálnu metódu *Generate*, ktorá už generuje príslušný kód. Výsledkom tejto metódy je list s dvoma položkami. Prvou je hodnota True alebo False, ktorá hovorí či výsledkom generovania XML elementu bola nejaká nová hodnota (napríklad ak bol pri generovaní XML elementu P*roperty* vygenerovaný text rôzny od prázdneho reťazca vráti True) a druhou položkou je text, ktorý bol vygenerovaný. Triedy jednotlivých XML elementov šablóny sú rozdelené do dvoch skupín:

- jednoduché – sú to triedy nasledujúcich XML elementov *Text, Token, Whitespace, Br* tieto XML elementy nemôžu mať v šablóne, žiadne vnorené XML elementy. Generujú len jednoduchý text podľa svojich atribútov.

- kontajnerové – sú triedy ostatných XML elementov, ktoré môžu mať aj vnorené XML elementy. Tieto triedy sú potomkom triedy CodeContainer.

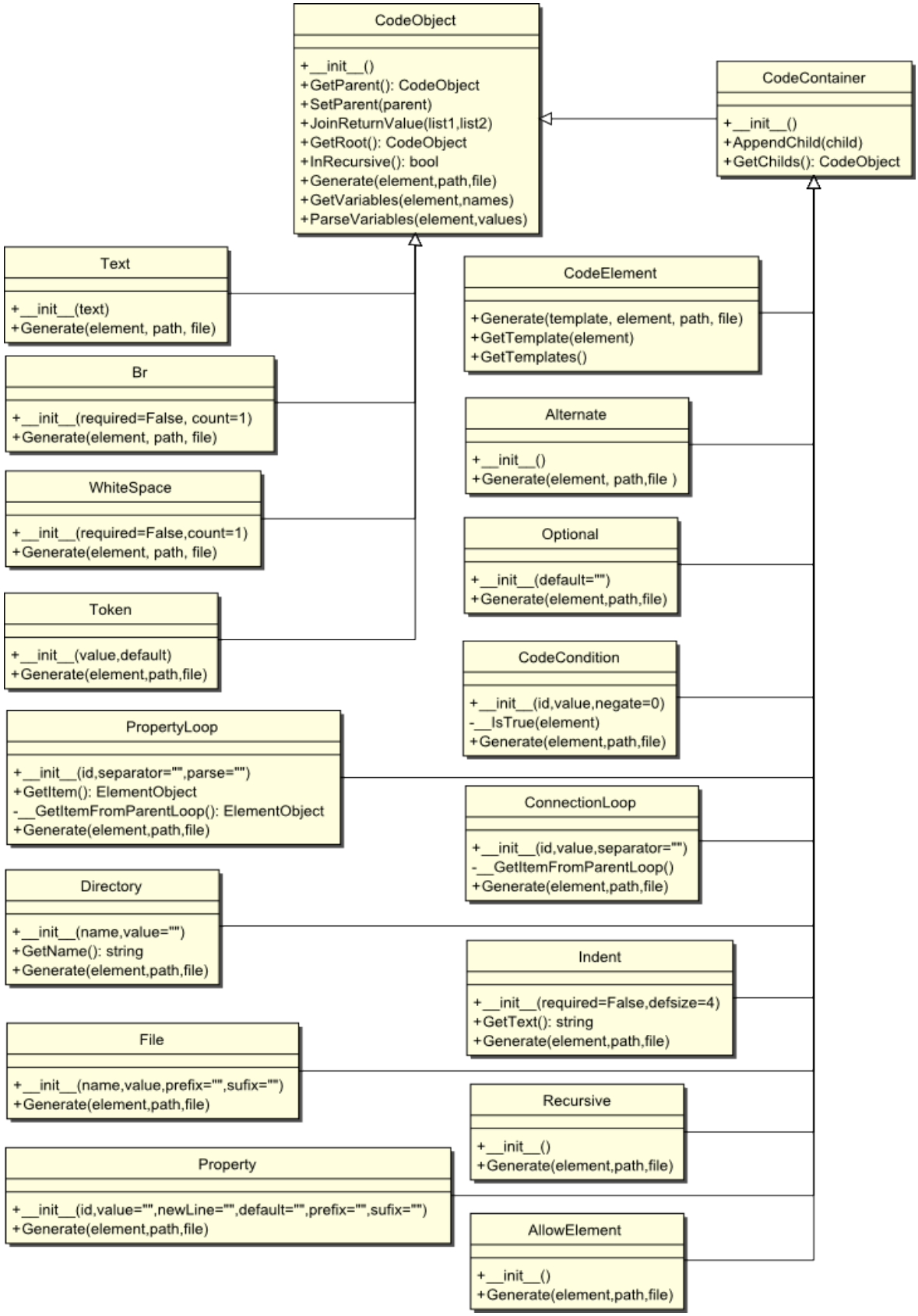
CodeObject

Je predkom všetkých tried XML elementov. Obsahuje tieto zaujímavé metódy:

- *JoinReturnValue* – je metóda, ktorá spája listy vracané generovaním jednotlivých tried XML elementov. Metóda spája tieto tak že z prvých hodnôt listov spraví logický súčin a ako druhú položku listu vráti zreťazenie textov. Ako prvý text bude text z prvého listu.
- *GetRoot* – Vráti inštanciu triedy XML elementu v ktorom sú vnorené ostatné XML elementy. V koreňovom prvku sú uložené hodnoty, ktoré sa týkajú celého generovania (napríklad veľkosť odsadenia).
- *InRecursive* – zisťuje či sa momentálne generovanie nachádza v rekurzii. Výsledkom metódy je True/False.
- *ParseVariables* – táto metóda rozparsuje hodnoty atribútov, ktoré boli zadané zo šablóny. Hodnoty týchto atribútov sa začínajú znakom "#" alebo "@". Ak je na začiatku "#" znamená to, že má zobrať vlastnosť UML elementu s názvom, ktorý ja za týmto znakom. Ak je na začiatku "@" tak to znamená, že sa má zobrať hodnota vlastnosti UML elementu zo súčasnej iterácie cyklu. Tento znak je používaný len v cykloch.

CodeContainer

Je predkom všetkých kontajnerových XML elementov. Obsahuje pole inštancií tried vnorených XML elementov. Obsahuje metódy na pridávanie a vrátenie jednotlivých inštancií vnorených XML elementov. Táto trieda je použitá aj v prípade, že sa v šablóne nachádza XML element, ktorý nemá vytvorenú svoju triedu.



Obrázok 17 – *Objektový návrh generátora šablón*

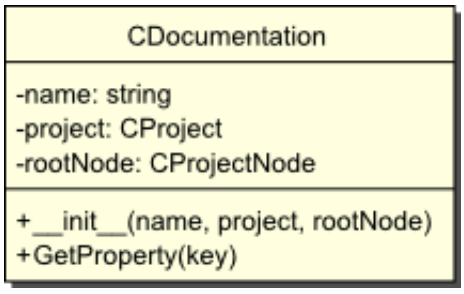
8 Generovanie dokumentácie

Generovanie dokumentácie je taktiež pomocou XML šablón. Keďže výstupný formát dokumentácie je HTML a šablóny na generovanie kódu boli navrhnuté univerzálne je ich možné použiť aj na generovanie dokumentácie. Dokumentácia však obsahuje aj špeciálne funkcie bolo potrebné doplniť nové XML elementy, ktoré by tieto funkcie generovali. Čítanie šablón pre generovanie dokumentácie je implementované ako pri generovaní kódu. Tvar vygenerovanej dokumentácie je ukázaný v prílohe B.

8.1 Objekt CDocumentation

Ako už bolo spomenuté generátor musí dostať nejaký element s ktorého získava potrebné informácie. Keďže UML.FRI neobsahoval žiadny element z ktorého by bolo možné generovať dokumentáciu musel som vytvoriť novú triedu z ktorej by to bolo možné. UML diagram vytvorenej triedy je na obrázku 18.

Trieda pre dokumentáciu sa skladá z názvu generovanej dokumentácie. Ďalej obsahuje odkaz na strom projektu a koreňový od ktorého sa má začať generovať dokumentácia. Všetky tieto atribúty sú zadávané ako parameter konštruktora. Metóda GetProperty vracia vlastnosti projektu. Ako parameter tejto metódy je hodnota, ktorá je získavaná z atribútov XML elementov.



Obrázok 18 – *Trieda CDocumentation*

8.2 Nové elementy šablóny pre generovanie dokumentácie

Oba nové XML elementy sú kontajnerové elementy.

8.2.1 ChangeElement

Nastaví UML element, ktorý sa bude generovať vo vnorených XML elementoch. Má nasledujúci parameter:

- id – tento atribút udáva UML element, ktorý sa má generovať.

8.2.2 CopyFile

Stará sa o skopírovanie súborov zadaných nasledujúcim parametrom.

- what – určuje čo sa má skopírovať z adresára projektu. Momentálne sa takto kopírujú ikonky UML elementov a diagramov, ktoré sú zobrazované v projektovom strome. Vygenerované SVG obrázky UML diagramov a štýly stránky.

9 Záver

Cieľom tejto práce bolo rozšíriť modelovací nástroj UML.FRI o generovanie kódu a dokumentácie. Teraz je len na uvážení používateľov tohto nástroje či využijú túto novú funkcionalitu, ktorá pri dobrom rozanalyzovaní projektu a správnom nakreslení UML diagramov tried vygeneruje kostry týchto tried a ušetrí tým programátorom pomerne dosť času, ktorý by inak strávili vypisovaním definícii tried namiesto toho aby, už pracovali na logických častiach svojho projektu.

Stanovené ciele, ktoré boli definované v úvode tejto práce sme splnili s ohľadom na časové a fyzické možnosti. Hlavnou časťou práce bolo navrhnúť univerzálne šablóny, ktoré by slúžili na generovanie kódu z UML diagramov, ale aj generovanie UML diagramov zo zdrojových kódov. Šablóny museli byť univerzálne aj tak aby ich bolo možné použiť aj pre generovanie do iných programovacích jazykov ako sú delphi, python a C++, ktoré boli v zadaní tejto práce. Tento cieľ sa nám podarilo splniť a preto boli použité aj na generovanie dokumentácie vo formáte HTML, kde museli byť pridané len dva nové XML elementy. Teraz sú šablóny pomerne dosť kompletné na to aby dokázali generovať rôzne jazyky. Tieto generátory boli do aplikácie pridané na miesta v menu, kde to majú aj iné UML case nástroje a preto by nemal byť problém vo využívaní týchto nových funkcií. V prílohe bude ešte ukázaný jednoduchý príklad generovania kódu do spomenutých jazykov a aj generovanie dokumentácie, ktorá je vo formáte XML. Ako možné pokračovanie by bolo spravenie šablón na generovanie ďalších programovacích jazykov napríklad Java, C# a iné, ale aj generovanie dokumentácie do iných formátov.

Zoznam použitých skratiek

UML - Unified Modeling Language

OMG - Object Management Group

W3C - World Wide Web Consortium

XML - Extensible Markup Language

DTD - Document Type Definition

DOM - Document Object Model

SVG – Scalable Vector Graphics

HTML - HyperText Markup Language

Referencie

[1] Schmuller Joseph, *Myslíme v jazyku UML,* Grada, 2001

[2] W3C Recommendation: XML vs. 1.0, 1998,
http://www.w3.org/TR/1998/REC-xml-19980210

[3] PyGTK 2.0 Reference Manual
http://pygtk.org/docs/pygtk/index.html

[4] Harms, D. – McDonald, K. *Začíname programovat v jazyce Python,* Computer Press, 2004

Príloha A

Popis vizuálnych a nevizuálnych vlastností tried

Ako som už spomínal momentálne je implementované len generovanie z diagramu tried a preto v tejto prílohe uvediem, aké majú triedy vlastnosti v modelovacom nástroji UML.FRI

Vizuálne vlastnosti tried:

Za vizuálne vlastnosti sú v UML.FRI považované tie vlastnosti, ktoré sú uvedené v XML definícii UML elementu v tomto prípade triedy. Sú to vlastnosti:

- **Name –** je názov triedy, ktorý je v ikone triedy uvádzaný v prvej časti.
- **Scope –** sú prístupové práva ku triede s iných tried. Môže nadobúdať hodnoty private, protected, public.
- **Attributes –** je zoznam atribútov, ktoré trieda obsahuje.
- **Operations –** je zoznam operácii v triede
- **Abstract –** môže nadobúdať hodnoty True, alebo False a rozhoduje o tom či bude názov triedy vypísaný kurzívou.
- **Notes –** je poznámka ktorá sa týka danej triedy.

Nevizuálne vlastnosti tried:

Medzi tieto vlastnosti sú považované rôzne vlastnosti atribútov a operácii. Ďalej budú popísané jednotlivé vlastnosti aj s tým ako sa generujú v programovacích jazykoch uvedených v zadaní práce.

Atribúty :

- **name –** je názov atribútu
- **type –** udáva typ atribútu
- **scope –** určuje viditeľnosť atribútu z iných tried
- **stereotype –** určuje nejaké nové vlastnosti atribútu
- **containment –** je spôsob odkazovania sa na atribút. Môže to byť hodnotou, alebo odkazom.
- **initial –** je počiatočná hodnota atribútu

- **doc –** obsahuje komentár k atribútu
- **derived –** udáva či je, alebo nie je atribút odvodený
- **static –** ak má táto vlastnosť hodnotu True znamená, že ide o atribút triedy(statický)
- **const –** hovorí o tom, že tento atribút je iba na čítanie
- **property –** určuje či sa môže pristupovať k atribútu ako property. Ak je nastavený na True vytvoria sa mu aj metódy get a set, ktorých názvy sú uložené v getter a setter
- **getter –** obsahuje názov get metódy
- **seter –** obsahuje názov set metódy

Operácie :

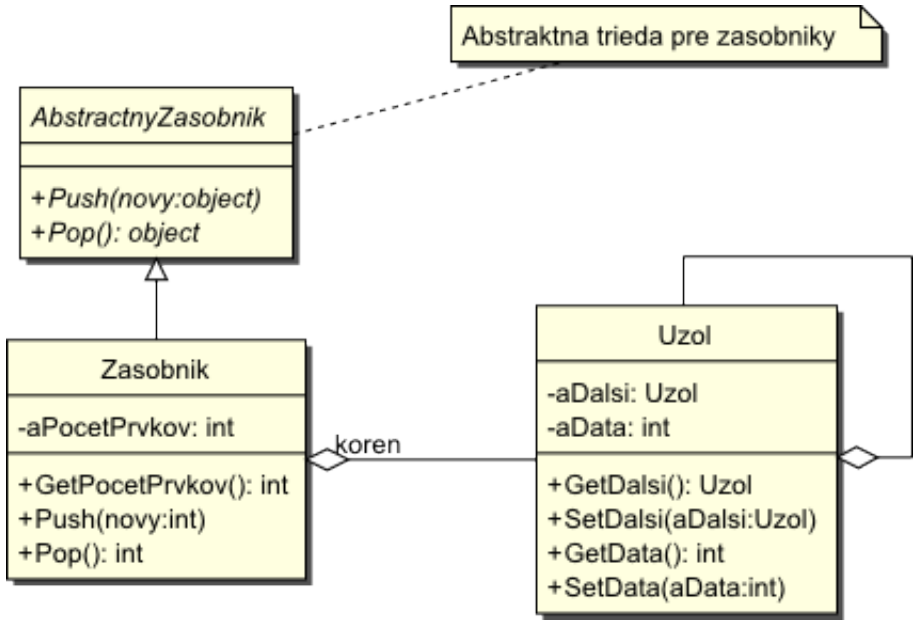
- **name –** je názov operácie
- **params –** sú parametre operácie vo formáte:

  identifikátor[: typ parametra] [= inicializačná hodnota]
- **abstract –** táto voľba znamená, že metóda je abstraktná
- **static –** znamená, že operácia je statickým členom triedy
- **const –** návratový typ tejto metódy je konštantný
- **returnarray –** metóda vracia pole
- **pure –** súvisí s jazykom C++ a hovorí, že táto metóda je čisto virtuálna
- **scope –** je viditeľnosť operácie z iných tried. Nadobúda hodnoty private, protected, public
- **type –** je návratový typ metódy
- **stereotype –** určuje nejaké nové možnosti tejto operácie
- **doc –** je komentár k metóde
- **initial –** je inicializačný kód metódy.
- **isquery –** znamená, že táto metóda nemodifikuje objekt
- **synchronize –** používa sa pre multivláknové metódy v jazyku Java

Príloha B

Príklad generovania kódu a dokumentácie

V tejto prílohe si ukážeme na jednoduchom príklade ukážeme výsledky generovania kódu aj dokumentácie. V novom projekte budeme mať len diagram, ktorý je zobrazený na obrázku 19. Pri vygenerovaní zdrojových kódov do jazykov C++, delphi a python nesmieme pozerať na syntaktické chyby ohľadom typov atribútov, keďže tie sa generujú také aké sú napísané v diagrame.



Obrázok 19 – *Diagram na ukážku generátorov*

Zdrojové kódy v jazyku C++:

Súbor AbstraktnyZasobnik.h

```
/**
 * Abstraktna trieda pre zasobniky
*/
class AbstractnyZasobnik{
public:

    virtual void Push(object novy)=0;
    virtual object Pop()=0;
};
```

Súbor Zasobnik.h

```
/**
 * Toto je trieda pre jednoduchy zasobnik celych cisel.
 * Trieda ma metody pre vkladanie, vyberanie a zistovanie
 * poctu prvkov.
*/
class Zasobnik: AbstractnyZasobnik{
private:
    /**
     * Obsahuje pocet prvkov v zasobniku
    */
    int aPocetPrvkov;
    Uzol *koren;

public:

    int GetPocetPrvkov();
    void Push(int novy);
    int Pop();
};
```

Súbor Uzol.h

```
/**
 * Je trieda pre uzol v zasobniku na
 * uchovavanie celych cisel
*/
class Uzol{
private:
    /**
     * Obsahuje odkaz na dalsi prvok
    */
    Uzol *aDalsi;
    int aData;

public:

    Uzol GetDalsi();
    void SetDalsi(Uzol aDalsi);
    int GetData();
    void SetData(int aData);
};
```

Súbor AbstraktnyZasobnik.cpp

```
#include "AbstractnyZasobnik.h"
```

Súbor Zasobnik.cpp

```
#include "Zasobnik.h"

int Zasobnik::GetPocetPrvkov(){

}
void Zasobnik::Push(int novy){

}
int Zasobnik::Pop(){

}
```

Súbor Uzol.cpp

```
#include "Uzol.h"

Uzol Uzol::GetDalsi(){

}
void Uzol::SetDalsi(Uzol aDalsi){

}
int Uzol::GetData(){

}
void Uzol::SetData(int aData){

}
```

Zdrojové kódy v jazyku Python:

Súbor AbstraktnyZasobnik.py

```
class AbstractnyZasobnik:
    '''Abstraktna trieda pre zasobniky
    '''
    def Push(self,novy):
        pass

    def Pop(self):
        pass
```

Súbor Zasobnik.py

```
class Zasobnik(AbstractnyZasobnik):
    ''' Toto je trieda pre jednoduchy zasobnik calych cisel.
    Trieda ma metody pre vkladanie, vyberanie a zistovanie
    poctu prvkov.
    '''
    def GetPocetPrvkov(self):
        pass

    def Push(self,novy):
        pass

    def Pop(self):
        pass
```

Súbor Uzol.py

```
class Uzol:
    '''Je trieda pre uzol v zasobniku na
    uchovavanie celych cisel
    '''
    def GetDalsi(self):
        pass

    def SetDalsi(self,aDalsi):
        pass

    def GetData(self):
        pass

    def SetData(self,aData):
        pass
```

Zdrojové kódy v jazyku Delphi:

Súbor uAbstraktnyZasobnik.pas

```
unit uAbstractnyZasobnik;

interface

type

    //Abstraktna trieda pre zasobniky
    AbstractnyZasobnik = class
    public

        procedure Push(novy:object);virtual;
        function Pop:object;virtual;
    end;

implementation

procedure AbstractnyZasobnik.Push(novy:object);
begin

end;

function AbstractnyZasobnik.Pop:object;
begin

end;

procedure AbstractnyZasobnik.Push(novy:object);
begin

end;

function AbstractnyZasobnik.Pop:object;
begin

end;
end.
```

Súbor uZasobnik.pas

```
unit uZasobnik;

interface

type

    // Toto je trieda pre jednoduchy zasobnik calych cisel.
    //Trieda ma metody pre vkladanie, vyberanie a zistovanie
    //poctu prvkov.
    Zasobnik = class(AbstractnyZasobnik)
    private
        //Obsahuje pocet prvkov v zasobniku
        aPocetPrvkov:int;
        koren:Uzol;

    public
        property aPocetPrvkov:int read GetPocetPrvkov ;

        function GetPocetPrvkov:int;
        procedure Push(novy:int);
        function Pop:int;
    end;

implementation

function Zasobnik.GetPocetPrvkov:int;
begin

end;

procedure Zasobnik.Push(novy:int);
begin

end;

function Zasobnik.Pop:int;
begin

end;

function Zasobnik.GetPocetPrvkov:int;
begin

end;

procedure Zasobnik.Push(novy:int);
begin

end;

function Zasobnik.Pop:int;
```

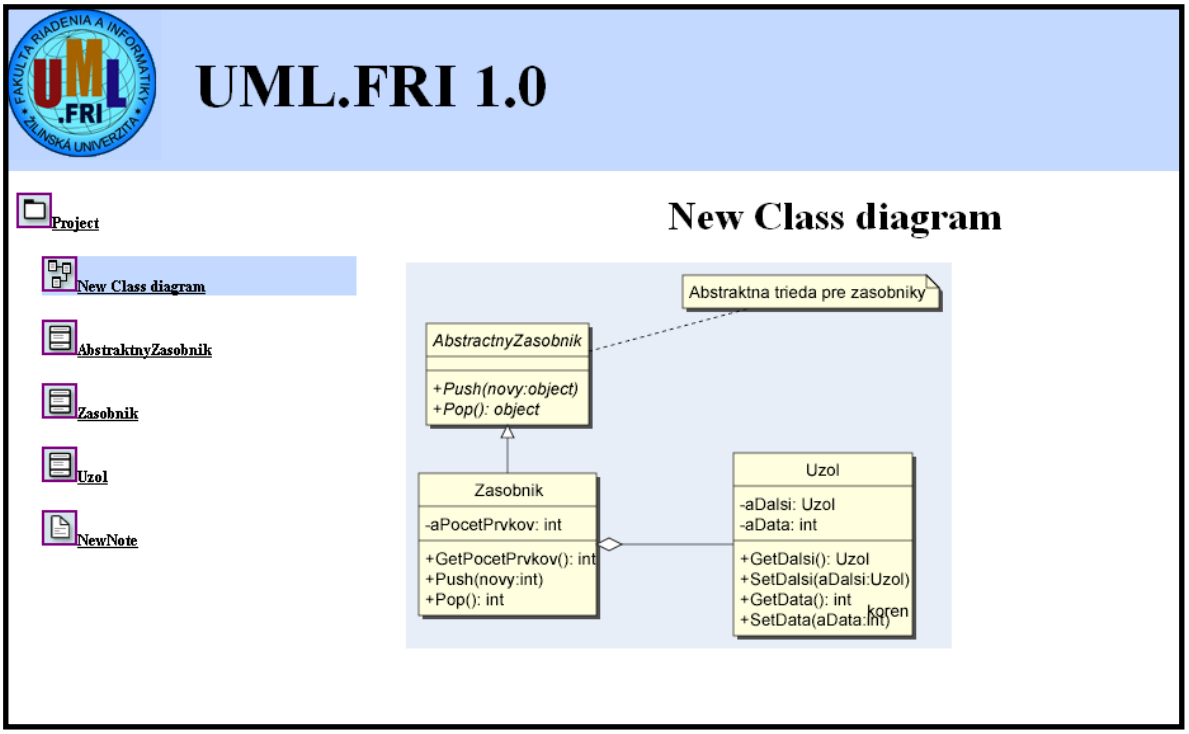
```
begin

end;
end.
```

Súbor uUzol.pas

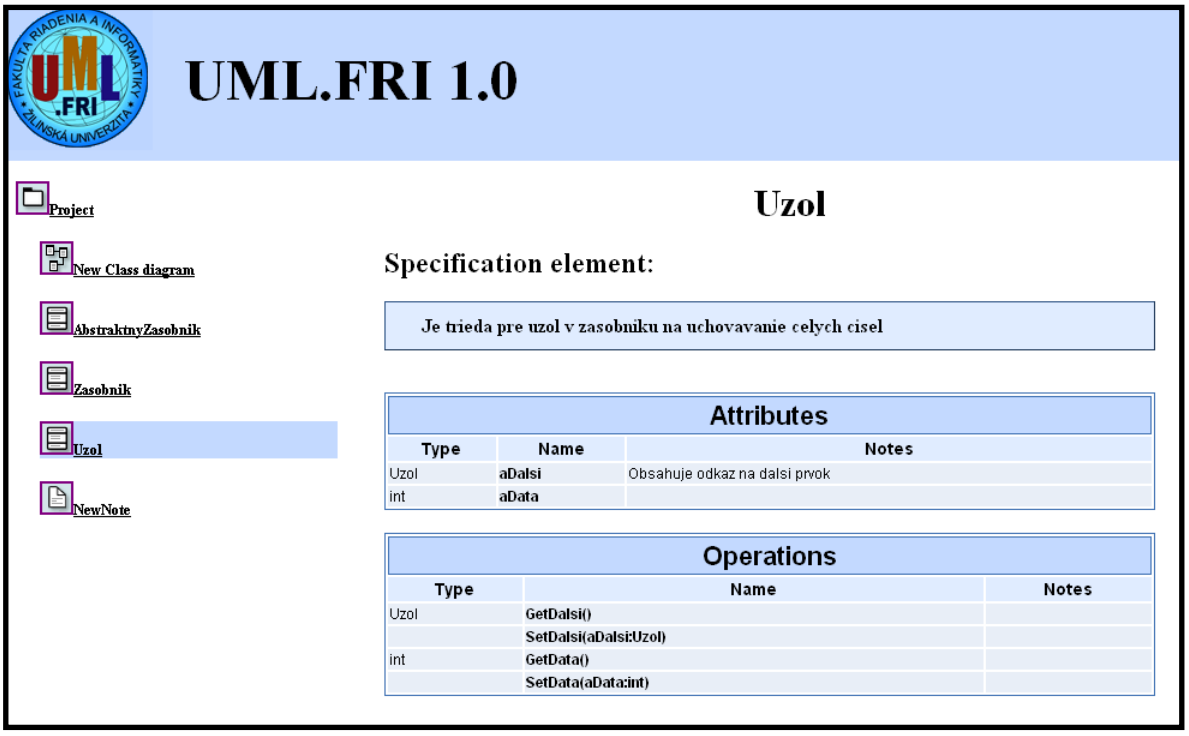
```
unit uUzol;
interface
type

    //Je trieda pre uzol v zasobniku na
    //uchovavanie celych cisel
    Uzol = class
    private
        //Obsahuje odkaz na dalsi prvok
        aDalsi:Uzol;
        aData:int;

    public
        property aDalsi:Uzol read GetDalsi write SetDalsi;
        property aData:int read GetData write SetData;

        function GetDalsi:Uzol;
        procedure SetDalsi(aDalsi:Uzol);
        function GetData:int;
        procedure SetData(aData:int);
    end;

implementation

function Uzol.GetDalsi:Uzol;
begin
end;

procedure Uzol.SetDalsi(aDalsi:Uzol);
begin
end;

function Uzol.GetData:int;
begin
end;

procedure Uzol.SetData(aData:int);
begin
end;

function Uzol.GetDalsi:Uzol;
begin
end;

procedure Uzol.SetDalsi(aDalsi:Uzol);
begin
end;

function Uzol.GetData:int;
begin
end;

procedure Uzol.SetData(aData:int);
```

```
begin
end;
end.
```

Ukážka vygenerovanej dokumentácie:



Obrázok 20 – *Diagram vo vygenerovanej dokumentácii*



Obrázok 21 – *Trieda Uzol vo vygenerovanej dokumentácii*