Úvod do sémantiky UML 1.3 podle standardu OMG s komentářem

Autor	RNDr. Ilja Kraval, výhradní autorské právo
Adresa	Firma Object Consulting, Lipina 100, 766 01 Valašské Klobouky
Mail	objects@objects.cz
URL	http://www.objects.cz
datum vydání	19.8.2001

Obsah

Úvodní slovo autora

Proč vznikla tato kniha

Tato kniha navazuje na e-knihu „Objektové modelování a UML v praxi 2000“ vydanou na serveru *www.objects.cz*. Předešlá kniha popisovala techniku a praxi v oblasti objektového modelování, které autor získal při konzultacích a tvorbě informačních systémů v mnoha firmách za období posledních dvou let. Kniha se po svém vydání v lednu roku 2001 setkala s velmi příznivým ohlasem. Pokud knihu nemáte, můžete si ji objednat na již zmíněných stránkách *www.objects.cz*.

Na druhé straně se ze strany některých čtenářů objevily výtky, že se předešlá e-kniha nedrží přesně standardu UML 1.3. Přitom nutno poznamenat, že e-kniha o objektovém modelování neměla za úkol uvést čtenáře do problematiky přesné syntaxe UML 1.3 (což není vzhledem k povaze této e-knihy ani možné), ale seznámit je se základy objektově orientovaného přístupu a objektového modelování v obecné rovině za použití příkladů z některých modelovacích technik. Je vcelku pochopitelné, že praxe získaná za několikaleté období spojuje několik způsobů modelování a také několik modelovacích škol dohromady. Mohu jenom potvrdit, že pokud se někdo pohybuje jako konzultant v prostředí mnoha softwarových firem, potom má vcelku dobrou příležitost seznámit se nejenom s velkým počtem CASE nástrojů, ale také s mnoha metodologiemi tvorby softwaru, a to nejenom s těmi nejlepšími, ale i s horšími. Přitom každý z CASE nástrojů používá některou verzi modelovacích jazyků a málokterý z nich používá poslední verzi UML 1.3. Každá trochu rozsáhlejší praxe vždy vytváří určitý komplexní obraz, který není samozřejmě přesným obrazem syntaxe některého z používaných jazyků. Při psaní předešlé knihy jsem si jako autor neuvědomil skutečnost, že by někdo mohl e-knihu o objektovém modelování pochopit jako „přesnou příručku pro syntaxi UML 1.3“, přičemž takto předešlá e-kniha zamýšlena určitě nebyla.

Myslím, že znalost UML jazyka podle syntaxe verze 1.3 patří mezi nutné znalosti vývojového pracovníka (analytika, designéra, programátora...). Stejně tak vývojový pracovník musí velmi dobře chápat objektové modelování v obecné rovině.

Jazyk UML verze 1.3 je jedním z možných souhrnů syntaktických pravidel pro to, co chápeme jako modelování v objektově orientovaném prostředí a co bylo předmětem předešlé knihy.

Na straně druhé musím podotknout, že samotná specifikace UML podle standardu OMG v originále (viz například stránky *www.omg.org*) není v této své původní podobě žádné učebnicové a vysvětlující čtení ve smyslu nějaké napůl populární a napůl odborné knihy. Jedná se o soubor definic standardů pro modelování na samotné hranici matematického vyjadřování. Kdo z vás studoval matematiku na vysoké škole a vzpomene si na suchopárné definice z matematických skript začínající slovy *„nechť existuje alespoň jeden prvek z množiny A, který...apod.“,* nyní přesně ví, o čem hovořím. V tomto duchu je originální specifikace UML 1.3 napsána. Opravdu není vhodné originál ve svém původním tvaru považovat za dobrou učebnici UML, protože její studium vyžaduje velmi dobré znalosti a zkušenosti z objektového modelování. Paradoxně specifikace UML podle standardu OMG není v žádném případě vhodnou literaturou pro toho, kdo se chce naučit UML.

Na základě těchto všech poznatků jsem uvážil, že by vskutku bylo vhodné uvést na knižní trh něco jako „příručku UML 1.3“ napsanou přesně podle standardu, ale hlavně s vysvětlujícím výkladem. Čtenář by se podle ní mohl seznámit s korektním výkladem UML poslední verze tak, jak jej zavádí standard, a současně by čtenář získal také vysvětlující komentář. Suchopárná syntaxe UML se tak umístí do obecného kontextu objektového modelování.

A tak vznikla tato e-kniha, jejíž elektronický výtisk nyní držíte „v e-ruce“.

Poznámka:

*V předešlém odstavci je použito slovo kontext, které se v objektovém modelování velmi často používá. V jednom z e-mailů jsem od čtenáře obdržel dotaz, co vlastně chápu pod pojmem **kontext**. Musím tento pojem vysvětlit z toho důvodu, že je pro objektové modelování a následné objektové programování velmi důležitý. Kontext zde budeme chápat přesně tak, jak je vysvětlen ve Slovníku cizích slov:*

kontext = ...významová souvislost, soubor souvislostí v nějakém dění...

Každý objekt (obecněji pojem) může vystupovat v různých souvislostech vůči jiným pojmům a jinak pod jiným pohledem, tedy v různých kontextech. Znamená to, že jedna a ta samá entita (resp. část systému) může figurovat odlišně v různém kontextu, tedy jinak v různých významových souvislostech.

Jak říká staré přísloví: „jinak voní seno koňům a jinak zamilovaným". Záleží na kontextu sena a tedy jeho významové souvislosti. Vše kolem nás je vždy chápáno v různých kontextech. Například fyzikálně vzato se kolem sebe pohybujeme jako tělesa, citově vzato se jedná o životně důležitá setkání dvou jedinců...

V objektovém programování tomuto kontextovému pojetí odpovídají:

- *úrovně abstrakce (jinak je chápán informační systém v analýze a jinak je viděn jako skupina instalovaných souborů zkompilovaného programu),*
- *v modelování existují v různém kontextu různé role stejných objektů v systému (každý objekt může v jiném kontextu hrát jinou roli),*
- *v konečném důsledku tomu odpovídá také pojem interface objektu (různé chování téhož objektu v různém kontextu díky jinému interfacu).*

Tedy pokud řekneme „v kontextu" máme tím na mysli „ve významové souvislosti".

Komu je určena tato e-kniha

Tato e-kniha je určena všem vývojovým pracovníkům v oblasti tvorby softwaru. Patří mezi ně analytici, designéři, programátoři, ale také testeři, vedoucí pracovníci atd., tedy všichni pracovníci, kteří se nějakým způsobem podílejí na tvorbě softwaru moderními způsoby a postupy objektově orientovaným přístupem na všech úrovních abstrakce.

Kromě toho doporučuji tuto knihu také těm pracovníkům, kteří se podílejí na tvorbě informačních systémů ze strany zadavatelů tvorby softwaru. Patří sem konzultanti informačních systémů pro problémovou doménu resp. osoby odpovědné za převzetí softwarových produktů ze strany odběratele apod.

Poznámka k překladu anglických pojmů

Mezi největší problémy odborných česky psaných knih patří vcelku pochopitelně potíže s dobrým a přesným překladem anglických termínů do odpovídajících českých pojmů . Pracoval jsem na projektech tvorby informačních systémů v několika týmech a zkušenost při práci s anglickými texty mne přivedla k jednomu zásadnímu praktickému až pragmatickému poznatku:

Lepší než špatný český překlad je ponechání původního anglického názvu.

V technické literatuře se mnohdy setkáváme s anglickými pojmy, které pojem svým překladem úplně přehodnotí a změní jejich význam, protože se jedná o skutečně zavedený *terminus technicus,* nejedná se tedy v žádném případě o slovo určené k překladu (například *Windows* apod.).

V některých případech se slova při překladu technicky přebírají jako slang. V tom případě se jedná o skutečné znečištění češtiny, které je technicky účelové, ale jazykově nesprávné. Překlad je sice možný, ale bohužel vede u vývojářů k nutným vedlejším zamýšlením se „o čem je vlastně řeč" a tedy k vedlejší zátěži při četbě a to z toho důvodu, že vývojáři jsou na tyto slangové výrazy zvyklí.

Například je otázkou, co je lepší zvolit u českého překladu „buffer": zda ponechat anglický „buffer" nebo zvolit „vyrovnávací paměť. Z hlediska čtenáře vývojáře je z hlediska splavnosti čtení určitě lepší „buffer", z pohledu češtináře je správné „vyrovnávací paměť"

Největší chybou je použít překladu tam, kde se jedná o přímý a nezaměnitelný název dané syntaxe. Takřka každý chápe nesmyslnost překladu syntaxe programovacího jazyka (například přeložit *Connection* v objektovém modelu ADO jako *Spojení* apod.), i když v jedné blíže nejmenované knize jsem setkal přímo s překladem syntaxe objektového modelu (jednalo se konkrétně *Document Object Model* u *Internet Exploreru*). Kniha tak ztratila svou hodnotu, protože v češtině na rozdíl od angličtiny samozřejmě model nefunguje. Navíc se v knize ani neuvedly anglické ekvivalenty. Podobný problém vznikl například v Excelu při překladu funkcí, ale tam má uživatel logicky možnost výběru mezi anglickými a českými názvy funkcí (například může použít buď *Average* nebo *Průměr*).

Praxe jak moje vlastní, tak praxe shodná u všech kolegů ve vývojových týmech mne dovedla k poznatku, že většina vývojářů oprávněně nesnáší tzv. „přesné puritánské překlady", které v konečném důsledku vedou k totální ztrátě kontextu. S takovým dokumentem, u kterého se použije přehnaně čistý překlad, potom nelze vůbec vývojářsky pracovat.

Problém překladů z anglické literatury v oblasti UML je však ještě hlubší. Jak bylo uvedeno, z hlediska programovacích jazyků je vcelku zřejmé, že nemá smysl překládat syntaxi daného jazyka. Velké dilema nastává s překladem z anglické literatury v oblasti UML. Potíž je v tom, že UML je vskutku opravdový jazyk jako každý jiný jazyk (například programovací), pouze se týká modelování. S trochou nadsázky si jej můžeme představit jako „*obdobu programovacího jazyka určeného pro modelování*". Tedy jedná se o jazyk se svou vlastní přesnou syntaxí a také se svým vlastním přesným modelem, podobně jako má svůj model již zmíněný *Document Object Model.*

A teď jak se rozhodneme: Překládat syntaxi jazyka? Překládat dané prvky modelu do češtiny? To by bylo určitě hrubou chybou, protože uvedené prvky modelu UML mají svůj obraz v syntaxi UML a to dokonce v jeho konkrétním přesném modelu. Zejména se jedná o *model tříd UML* a názvy tříd by se ze zásady neměly překládat. Na druhou stranu ponechání pouze anglických názvů vede k češtinářským excesům, kdy se anglické názvy skloňují (například: *...s actorem... s use casem* apod.), nebo se anglické pojmy používají v kombinaci s českými pojmy a vznikají podivné anglo-české hybridy.

Musíme přijmout určitý kompromis a to vždy znamená určit si priority. Nechť češtináři prominou, ale zde v této knize použiji následující hlavně technický a přesně definovaný přístup k překladu:

- z důvodu zachování správné syntaxe se jako primární pojem u prvků modelu UML považují názvy anglické, tj. názvy podle standardu UML.
- Ze všech autorovi známých překladů je poté vybrán jeden český ekvivalent. Při tomto překladu je hlavní snahou dodržet správný kontext pojmu (tj. správnou významovou souvislost pojmu). Jedná se tedy nikoliv „češtinářský přístup", ale o „technický přístup", který nemusí vždy vést k zachování čistoty jazyka.

- protože se jedná o technickou literaturu, není v žádném případě snahou hledat synonyma pro získání bohatosti jazyka. Naopak používá se pouze jeden již zvolený ekvivalent překladu s tím, že se velmi často opakuje také jeho anglický název a to hlavně tehdy, mluvíme-li o daném prvku z modelu tříd UML. V některých případech se anglický název pro zachování kontextu bude dávat do závorky za již zvolený český jednotný unikátní překlad. Vyplývá z toho, že pro daný prvek modelu UML budou vždy k dispozici pouze dva pojmy: Jeden anglický (z hlediska modelu UML je chápán jako hlavní) a druhý český (chápán jako vedlejší pouze pro česky psaný popis). Na konci knihy je uveden malý slovník takto autorem zvolených ekvivalentů anglických termínů (prvků modelu UML) a jejich česky zvolených překladů. Tento přístup má také výhodu v tom, že existence jednoznačné dvojice překladu (kde „levá“ anglická strana je převzata z UML) může být dobrým základem pro zpracování některým specializovaným pracovištěm zabývajícím se českým jazykem (námět pro diplomku z českého jazyka apod.).
- v některých případech se samotný anglický název bude považovat za obdobu jména (něco jako například pan *Smith*) nebude se vůbec překládat (určitě nemá smysl psát *UMJ* jako zkratku *Unifikovaného Modelovacího Jazyka* namísto UML jako *Unified Modeling Language*).

Mapování do češtiny, pokud k němu dojde, je pouze a jenom v jednoznačné dvojici a proto vždy „technicky přesné“. Jediná výtka, která by takto mohla vzniknout, spočívá leckde ve znečištění češtiny, což považuji za chybu sice možnou, ale myslím, že mnohem vážnější by bylo dopustit se chyby technické a v důsledku toho chyby odborné.

Úplnost specifikace sémantiky UML a doplňující komentáře v této knize

Tato kniha není v žádném případě pouhým překladem specifikace UML uvedené na stránkách organizace OMG. Uvedený dokument standardu UML sice patří k významným zdrojům pro napsání této knihy, ale tato e-kniha obsahuje také vlastní komentáře autora.

Je pochopitelnou snahou dodržet syntaxi UML verze 1.3, a proto se nemůžeme ve výkladu odlišovat od předepsaných definic v dokumentu OMG, ale jak zdůrazňuji, nejde zde v žádném případě o věrnou kopii pouhým překladem. Pokud bychom chtěli vystihnout abstraktní konstrukci této e-knihy, dala by se shrnout do těchto bodů:

- kniha vychází ze specifikace UML 1.3, při výkladu se dodržuje přesně syntaxe UML 1.3, ale kniha je podána jako vysvětlující výklad UML 1.3 a nikoliv jako pouhá specifikace UML, která je velmi strohá a z hlediska vysvětlujících výkladů „špatně čitelná“.
- ze specifikace jsou vybrány podstatné a důležité pasáže nutné pro pochopení konstrukce a ovládání jazyka UML a některé pasáže jsou vynechány, tato volba výběru je subjektivní.
- specifikace je doplněna a rozšířena o vlastní výklad včetně vlastních příkladů resp. postřehů z praxe, o poznámky a o vlastní myšlenky autora.

Toto vysvětlení zde podávám zejména z toho důvodu, aby nevznikl dojem, že držíte v ruce pouhý přesný překlad z anglického originálu specifikace UML 1.3 od skupiny OMG.

0. Úvodem o UML

0.1 Co je UML

Unified Modeling Language (dále také UML) by se dal přeložit jako *„unifikovaný modelovací jazyk“.* Tvůrcům systémů, kteří se zabývají objektovou analýzou a designem, UML poskytuje jeden standardní unifikovaný konzistentní modelovací jazyk pro specifikaci, vizualizaci, konstrukci a dokumentaci produktů jak v oblasti objektově orientovaných softwarových systémů, tak v modelování podniku (tzv. *business modeling*), ale také u jiných „ne-softwarových systémů“. Jedinou podmínkou pro modelování pomocí UML je zavedení objektově orientovaného přístupu.

Pod pojmem *business modeling* zde přeloženém jako *modelování podniku* rozumíme modelování samotné problémové domény, tedy problémovou doménu chápeme jako „business = podnik“. Toto modelování, kdy předmětem není informační systém, ale podnik, se provádí z několikerých důvodů. Nejčastějším z nich je tvorba modelu podniku předcházející analýze informačního systému, tj. jedná se o analýzu prostředí, do kterého bude informační systém dosazen. Mnohdy se toto modelování provádí také z důvodů optimalizace chodu podniku, odhalení chyb chodu podniku, jeho vylepšení apod.

Příklad: Představme si, že budeme mít za úkol vytvořit informační systém úřadu práce. Přitom na počátku projektu nemáme vůbec žádné znalosti o tom, jaké procesy na takovémto úřadě probíhají (jak podle zákona, tak podle vnitřních předpisů a pokynů příslušného ministerstva). Jako první krok se může vytvořit analýza podniku, v tomto případě je podnikem úřad práce. Zjistí se, „jak to vlastně na úřadě chodí a to nejenom z hlediska uchazeče o zaměstnání, ale celého podniku“. Předmětem analýzy a modelování pomocí UML zde není informační systém, ale samotný podnik, tedy úřad práce. Je třeba si uvědomit, že požadavek na tvorbu modelu podniku úřadu práce již jednou v minulosti existoval a to tehdy, když na odpovídajícím ministerstvu vznikaly předpisy a navrhované zákony pro tento úřad. Tehdy se „někdo zamýšlel nad tím, jak to vlastně všechno bude fungovat“.

Specifikaci UML lze také považovat za soubor nejlepších konvergentních praktik modelování v oblasti tvorby systémů objektově orientovaným přístupem. UML je dobrým následovníkem tří předešlých vedoucích škol syntaxe modelování v OOP:

- *Booch (název této školy je odvozen od jména autora)*
- *OMT Object Modeling Technique*
- *OOSE Object-Oriented Software Engineering*

UML není pouze sloučením těchto tří škol, ale obsahuje další nové přístupy, které uvedené tři školy nepodporují. UML tedy jednak dále rozvíjí kladné přístupy těchto škol, ale rovněž také doplňuje tyto školy o další přínosné prvky.

Jedním z hlavních cílů UML je vylepšit současný stav v činnosti tvorby v OOP vytvořením jednotného modelovacího jazyka. Tím se slučují a standardizují vyjadřovací prostředky modelování napříč různými technikami při použití různých modelovacích nástrojů. V důsledku to také znamená, že lze vyměňovat informace mezi různými nástroji pro modelování (export a import částí modelů je umožněn

již z principu, existuje totiž stejná syntaxe pro modelování u obou nástrojů). Z důvodu možné výměny informací mezi modelovacími nástroji je také nutná shoda v syntaxi a notaci modelování.

Z toho pro UML vyplývají následující požadavky, které jsou ve specifikaci UML zahrnuty:

- Specifikace UML zavádí formální definici společného *metamodelu* pro objektovou analýzu a design (dále také OA & OD). Tento metamodel obsahuje *statické modely*, *modely chování*, *modely užití* a *modely architektury.*
- Specifikace UML zavádí IDL (tj. *Interface Definition Language*) pro standardní mechanismus výměny informací z modelů mezi nástroji.
- Specifikace UML zavádí uživatelsky čitelnou (angl. *„human-readable“*) notaci pro reprezentaci OA & OD modelů systémů. Notace je podstatnou součástí specifikace UML (vedle sémantiky).

Zde se poprvé setkáváme s pojmem *metamodel*, který je v specifikaci UML velmi důležitý. Co si pod tímto pojem představit? Zkratka „*meta*“ obecně znamená něco jako „*o jednu abstrakci výše*“.

Například pojem metaclass znamená v OOP třídu tříd, tedy takovou třídu, jejímiž instancemi jsou třídy.

Podobně *metamodel* není nic jiného, než model modelů. Je to tedy takový model, jehož instancí (tj. dosazením do prvků modelu konkrétními instancemi prvků) vznikne konkrétní model, například model informačního systému úřadu práce a navíc tento model odpovídá syntaxi UML (protože vznikl jako instance metamodelu UML) .

S pojmem metamodel jsme se již nepřímo setkali v předešlé kapitole pojednávající o zásadách překladu z angličtiny do češtiny. Tam jsme tento „metamodel UML“ nazývali zkráceně pouze jako „model UML“.

Poznámka:

Přesněji bychom měli v kapitole o zásadách překladu psát o „prvcích metamodelu UML“ a nikoliv o „prvcích modelu UML“.

Pokud se vrátíme k příkladu s Document Object Modelu IE, jeho instancí vzniknou nějaké instance objektů v prostředí IE, kdežto instancí modelu UML vznikne nějaký konkrétní model nějakého informačního systému zapsaný v syntaxi UML, například úřadu práce.

Jinými slovy „třída“ a „atribut“ jsou meta-prvky metamodelu UML a mají mezi sebou určitý vztah, který je dán právě vztahem v tomto metamodelu. Nesmíme metamodel zaměňovat se samotným již konkrétním modelem, který popisuje nějaký informační systém, například model informačního systému úřadu práce.

Není bez zajímavosti, že samotný model UML, tedy správněji metamodel UML, je vytvořen v jazyce UML. Znamená to, že vztahy mezi prvky UML v tomto metamodelu jsou modelovány pomocí UML, zejména pomocí modelu tříd (používá se *class model)* .

Z uvedeného vyplývá, že v této chvíli již známe celkem dvě hlavní oblasti předmětů modelování UML, tj. známe tři možné oblasti „co se v praxi modeluje pomocí UML“:

1. Předmětem modelování je samo UML. V tom případě výsledný model nazýváme *metamodel UML* a tento *metamodel UML* plně vystihuje možnosti použití UML „komerčně v projektech“,

tj. v oblasti označené v tomto výčtu jako 2. Metamodel UML je základem pro vyjádření sémantiky (možná povolená pravidla) a abstraktní syntaxi UML.

2. Předmětem modelování je informační systém nebo podnik. V případě informačního systému se jedná o klasické „softwarové“ využití UML v určitých fázích vývoje, kdy se v jednotlivých abstraktních úrovních modeluje informační systém. Nejčastější použití UML je právě zde, tj. v tvorbě softwaru. Na druhé straně předmětem modelování může být i nějaký podnik (angl. *business modeling*). Modeluje se chování a skladba podniku (zde máme na mysli podnik v širším slova smyslu, například továrna, výrobní jednotka, škola, úřad, hotel... atd.). Modelování podniku se používá buď jako fáze předcházející modelování informačního systému anebo jako se jedná přímo o proces optimalizace chodu podniku (resp. obojí).

V předešlém výčtu je v odstavci 1. předmětem modelování samotný metamodel UML, který slouží k vyjádření sémantiky UML. Oblast označená jako 2. slouží modelování v UML k tvorbě produktů jako součást „komerčních“ projektů.

Z tohoto hlediska lze použití UML rozdělit do dvou základních okruhů podle výstupů (produktů modelování, *„artifacts“*). První oblastí je užití UML pro definování sebe sama tvorbou metamodelu, druhou oblastí je užití UML pro modely v projektech vývoje.

V dalším výkladu v této knize se budeme velmi podrobně zabývat zobrazením a popisem právě metamodelu UML, který určuje možné prvky použitelné v UML a vztahy mezi nimi. Pomocí metamodelu UML tak získáme možnou syntaxi UML pro konkrétní modely konkrétních informačních systémů.

V této knize není již předložena notace UML, tj. možné grafické zobrazení prvků modelu v UML. Je plánována další publikace (navazující na tuto e-knihu) pojednávající pouze o notaci UML.

Poznámka: Notace znamená „jak se co maluje“, kdežto sémantika a abstraktní syntaxe vysvětluje podstatu toho, co se maluje. Je pochopitelné, že to druhé je mnohem důležitější, protože notace je pouze otázkou dohody, jak co namalovat.

Souhrnně se dá říci, že cílem této knihy je podat dost podrobný popis a vysvětlení metamodelu UML.

0.2 Co je OMG

Object Management Group (OMG) je mezinárodní organizace založená v roce 1989. Sdružuje více než 800 členů. Jejím posláním je zavádět standardy v oblasti vývoje a tvorby softwaru v objektově orientovaném prostředí. Její stránky jsou dostupné na adrese http://www.omg.org

Jedním ze standardů zaváděných OMG je také jazyk pro modelování v objektově orientovaném prostředí – UML.

Poznámka: Například dalším takovým standardem zavedeným OMG je specifikace pro komponentní technologii CORBA.

Dá se říci, že garantem a tvůrcem jazyka UML je zastřešující organizace OMG. Na tvorbě specifikace UML 1.3 se pod hlavičkou OMG podílelo mnoho firem a jednotlivců. Můžeme z těch nejznámějších jmenovat například tyto firmy:

- *Rational Software Corporation*
- *Platinum Technology, Inc.*
- *IBM*
- *Microsoft Corporation*
- *Hewlett-Packard Company*
- *ObjecTime Limited*
- *Oracle Corporation*
- *Unisys Corporation*
- *SAP*
- aj.

Určité „vůdčí“ postavení mezi těmito firmami má *Rational Software Corporation.* Je to díky tomu, že tři hlavní protagonisté UML, pánové *Jacobson*, *Booch* a *Rumbaugh* (někdy tak trochu s úsměvem přezdívaní na Internetu jako *„three amigos“*) jsou v úzkém vztahu právě s touto firmou.

Spojení a kontakt na OMG je následující:

OMG Headquarters

492 Old Connecticut Path

Framingham, MA 01701

USA

Tel: +1-508-820 4300

Fax: +1-508-820 4303

pubs@omg.org

http://www.omg.org

1. Souhrnně o UML

1.1 Modely v UML

1.1.1. Abstrakce a různé pohledy na systém

Velmi důležitým přístupem při modelování systémů je přístup pomocí abstrahování (tj. použití abstrakce). Chápe se tím soustředění se na určitý druh pohledu při zanedbání jiného druhu pohledu a tím vynechání určitých prvků v modelu, které nejsou pro tento pohled vůbec přínosné nebo dokonce jsou v tomto pohledu irelevantní.

Zavádějí se následující obecné postupy, které zjednodušují a zefektivňují modelování díky použití abstrakce:

- Každý trochu složitější systém je modelován pomocí několika druhů pohledů reprezentujících nějaký druh modelu.
- Každý model může být vyjádřen s větší nebo menší přesností a s možnými úrovněmi podrobností (podobně jako zoom na mapě), které dekompozicí odkrývají, anebo naopak schovají hlubší detaily.
- Nejlepší modely odpovídají realitě.

Poznámka:

Velmi podobný přístup je při plánování stavby domu. Existují pohledy na dům z hlediska rozmístění stěn, rozvodů vody, odpadů, rozvodů elektřiny atd., což jsou různé pohledy, v nichž se některé detaily specifické pro jeden druh pohledu v jiném druhu pohledu jednoduše neuvažují. Navíc lze modely zjemňovat podle potřeby, například jako „hrubý náčrtek“ nebo detail místnosti apod.

Modely domu by samozřejmě měly souhlasit s realizovanou stavbou, jinak jsou k nepotřebě.

Pro tuto možnost použití přístupu abstrakce se zavádí několik takovýchto možných pohledů na předmět modelování a tyto druhy pohledů jsou konkrétně reprezentovány tzv. ***diagramy UML***.

UML zavádí následující diagramy. Uveďme je nejprve angličtině, přičemž je pro další výklad zvolen vždy jeden překlad:

- use case diagram, tj. diagram případů užití.
- class diagram, tj. diagram tříd.
- behavior diagrams, tj. diagramy chování, kam spadají diagramy
    - statechart diagram, tj. stavový diagram
    - activity diagram, tj. diagram aktivit

- interaction diagrams, diagramy interakcí, kam spadají diagramy
  - sequence diagram, tj. sekvenční diagram
  - collaboration diagram, tj. diagram spolupráce
- implementation diagrams, implementační diagramy, kam spadají diagramy
  - component diagram, tj. komponentní diagram (také překládán jako komponentový diagram)
  - deployment diagram, tj. diagram rozmístění zdrojů

Poznámka: Všimněme si, že v tomto výčtu nenalezneme „klasický" objektový diagram (object diagram), který se též nazývá instance diagram, tj. diagram instancí. Tento diagram je chápán jako součást jiných diagramů.

1.2 Důvody pro modelování

Se vzrůstající složitostí softwaru spolu se snahou po zvýšení kvality také vzrůstají požadavky v oblasti modelování. Důvod je prostý: Modelování umožňuje zachytit a zdokumentovat jednotlivé úrovně abstrakce tvorby softwaru tak, jak se to vyžaduje v současné moderní tvorbě SW.

Poznámka: Jako standardní úrovně abstrakce se pro modelování uvádějí tyto:

- *strategické modelování (tj. úvodní fáze projektu),*
- *modelování analýzy (tj. modelování bez závislosti na prostředí),*
- *modelování designu (tj. modelování pro dané prostředí),*
- *modelování implementace (tj. dosazení modelu do daného prostředí i s implementačními detaily),*
- *kódování.*

Modelovací jazyk musí obsahovat:

- Prvky modelu (angl. *„model elements"*), tj. základní koncepci a syntaxi jazyka s definicí prvků jazyka,
- Notaci (angl. „*notation"*) tj. grafické vyjádření prvků modelu,
- Směrnice (angl. „*guidelines*") tj. ustálená a použitelná vyjádření modelů v praxi.

1.3 Industrializace tvorby SW

Zvláštností současných trendů v softwarovém průmyslu je snaha po industrializaci i v oblasti tvorby SW. Všimněme si, že i v tak abstraktní oblasti, jakou tvorba SW bezesporu je, se zavádí obdoba

automatizace jako v jiných oblastech výroby. Kdysi tak „romantický způsob“ vytváření softwaru, kdy se využila kreativita pouze jednoho zaměstnance, který sám program navrhl a sám jej většinou za bezesných nocí napsal, je dnes velmi výjimečná a u rozsáhlých systémů dokonce nemožná. Dnes již tvorba SW spíše připomíná výrobu na klasické výrobní lince než individuální tvorbu jako „za starých dobrých časů“.

Moderní techniky vedoucí k větší automatizaci a mající tedy industrializační charakter v tvorbě SW jsou tyto:

- *komponentní technologie (component technology)*
- *visuální programování a visuální modelování (visual programming)*
- *použití vzorů (patterns)*
- *frameworks* (což by se dalo přeložit významově jako *„opětovně použitelné soustavy SW knihoven“*, ale dále není překládáno a je ponechán výraz frameworks jako *terminus technicus*)

1.4 Hlavní cíle UML

Základní cíle UML lze shrnout do těchto bodů:

- Poskytuje uživatelům již hotový, smysluplný a standardní modelovací jazyk, pomocí kterého lze nejenom modely vyvíjet, ale také je vzájemně předávat jak mezi jednotlivci, tak mezi týmy a firmami.
- Umožňuje dále rozvíjet základní koncepty a myšlenky v modelování.
- Zavádí specifikaci modelovacího jazyka, který není závislý na žádném programovacím jazyce.
- Zavádí obecné zásady pro pochopení základů modelovacích jazyků.
- Podporuje rozvoj nejmodernějších konceptů vývoje SW, jako jsou OOP, komponentní technologie, použití vzorů, použití *framework* aj.
- Sjednocuje nejlepší známé techniky modelování.

Je zřejmé, že tyto vlastnosti staví modelovací jazyk UML do jedinečného postavení mezi ostatními modelovacími jazyky.

1.5 Co není předmětem UML

Jak bylo řečeno, UML poskytuje jeden standardní unifikovaný konzistentní modelovací jazyk pro specifikaci, vizualizaci, konstrukci a dokumentaci SW produktů v různých úrovních abstrakce jejich vývoje. Existují oblasti, které ze zásady nejsou předmětem UML. V této kapitole je podán jejich výčet i s vysvětlením.

1.5.1. Programovací jazyky a UML

UML není zaveden jako visuální programovací jazyk (angl. *visual programming language*). UML je jazykem pro vytváření modelů, ze kterých lze přechodem do dalších více implementačních úrovní abstrakce v konečném důsledku programový kód vytvářet. V žádném případě není UML zaměřeno na nějaký programovací jazyk a dokonce není ani zaměřeno na visuální programování v nějaké obecnější rovině.

Poznámka:

Existují skutečně visuální programovací jazyky. Jako nejznámější a také nejvýkonnější jmenujme řadu produktů označenou jako „Visual Age“. Měl jsem kdysi tu možnost pracovat s produktem „Visual Age for Smalltalk“. Jedná se de facto o skutečné programování s velmi silnou podporou „visuálního programování“ , kde i tzv. „non-visual“ prvky, jako kolekce objektů, business objekty apod. bylo možné zapojit do procesu tvorby kódu pomocí visuálního programování („naklikáním ikonek“ a „propojením mezi nimi“).

1.5.2. Nástroje

Je zřejmé, že zavedení nějakého libovolného standardu má svůj vliv také na nástroje, kterých se tyto standardy týkají. Zavedení UML má bezprostřední vliv na funkcionalitu CASE nástrojů pro modelování v objektově orientovaném prostředí. Na druhou stranu UML si v žádném případě neklade za cíl zaměřit se na nějaký nástroj resp. na specifikaci jeho vnitřní funkcionality, jeho rozhraní, způsobu ukládání dat apod.

Vztah UML ke CASE nástrojům je takový, že UML pouze definuje standardní sémantiku modelování, kterou by měl daný nástroj dodržet, pokud má splňovat požadavky UML (a nic víc).

Poznámka: Není bez zajímavosti, že součástí specifikace UML je také poměrně dost rozsáhlá definice DTD pro XML v UML.

1.5.3. Pracovní postupy

Zde je zvolen český překlad *pracovní postupy* pro anglický pojem *process*. Pod tímto pojmem máme na mysli takové pracovní postupy v nějaké firmě, které vedou k tvorbě produktů, zde většinou míněno SW produktů.

UML sice poskytuje sémantiku obecného modelovacího jazyka, což znamená, že různé firmy používají stejnou sémantiku a stejnou notaci při modelování, avšak mohou používat různé modely v různých částech svých specifických pracovních postupů. Tvůrci UML se tedy nezaměřují na určitý daný pracovní postup, ale vřele doporučují používat dobré pracovní postupy při tvorbě SW a za použití UML. Doslova se ve specifikaci UML uvádí, že hlavní rozdíl mezi velmi úspěšnými projekty („*hyperproductive projects*“) a neúspěšnými projekty („*unsuccessfull ones*“) spočívá v dobře formulovaných a dobře řízených pracovních postupech.

Poznámka: Vzpomeňme při té příležitosti na velký význam normy ISO.

2. Architektura UML

2.1 Čtyřvrstvá architektura jazyka UML

Architektura UML je založena na obecné čtyřvrstvé struktuře metamodelu. Čtyřvrstvová architektura metamodelu se skládá z následujících vrstev:

- meta-metamodel
- metamodel
- model
- uživatelské objekty (angl. *user objects*)

Určitá část rozvrstvení již byla vysvětlena v předešlých kapitolách při výkladu pojmu metamodel UML (tj. metamodel UML = model pro modely v syntaxi UML). Při porovnání těchto čtyř vrstev s výkladem pojmu metamodel, vidíme, že samotný model UML (tj. metamodel) je pouze jednou z těchto čtyř vrstev (konkrétně v předešlé výčtu druhá v úrovni abstrakce).

Pro pochopení vztahu těchto vrstev je třeba si uvědomit triviální fakt, že vyšší úroveň abstrakce určuje pravidla pro instance na nižší úrovni abstrakce. Je to podobné jako například když úroveň „typ proměnné“ určuje pravidla pro „instance typů proměnných“, tj. samotné obsahy proměnných (například string a „Jan Novák“).

Pro uživatele UML, tj. pro tvůrce modelů pomocí UML, je vrstva označená jako metamodel samozřejmě nejzajímavější a je také předmětem studia v této knize. Přechod mezi úrovní metamodel a úrovní modelu není nic jiného, než tvorba modelu v syntaxi UML, čímž vzniká konkrétní instance metamodelu jako jeden konkrétní model nějakého konkrétního systému (například informačního systému úřadu práce).

Všimněme si, že nad vrstvou metamodel existuje ještě jedna abstraktnější úroveň tzv. meta-metamodel. Ta určuje pravidla pro samotný metamodel. Tato vrstva by byla pro nás zajímavá tehdy, kdybychom chtěli vytvářet nějaký vlastní jazyk podobný UML a jak UML, tak tento nový jazyk by byly dvě instance meta-metamodelu.

Poznámka: Vztahy mezi meta – úrovněmi se v teorii programování používají velmi často, protože se jedná o jeden ze základních přístupů pro určování pravidel nějaké sémantiky. Velmi podobná situace je například v jazyce XML, kde existují také určité vrstvy meta-úrovní. Nejnižší meta-úrovní je jeden konkrétní tag, například

```
<jmeno>Jan Novák</jmeno>
```

O jednu abstrakci výše, tj. „jedno meta“, je formulace pravidel v tzv. části dokumentu DTD. O další abstrakci výše jsou pravidla pro XML atd.

Vlastnosti čtyř vrstev pro meta-modelování jsou vysvětleny v následující tabulce:

Vrstva	**Popis vrstvy**	**Příklad**
meta-metamodel	Definuje pravidla jazyka pro tvorbu metamodelu (viz nižší vrstva)	MetaClass, MetaAttribute
metamodel	Instancí meta-metamodelu je metamodel Definuje pravidla pro tvorbu modelu	Class, Attribute
Model	Instance metamodelu je model. Definuje pravidla popisující informační doménu („typy“)	Osoba, Úvěr, Ručitel úvěru, Cenný papír, Rodné číslo, atd.
uživatelské objekty	Instancí modelu jsou uživatelská „data“	Jan Novák, Úvěr číslo 10215, 541212/2312

Pro praktické studium UML je velmi důležitá vrstva označená jako metamodel, která popisuje model samotného UML.

Z hlediska teoretického je třeba pouze poznamenat, že čtyřvrstvá architektura počínající meta-metamodelem odpovídá přístupu definován pomocí tzv. *Meta-Object Facility* zavedeném také organizací OMG. Z hlediska samotného praktického studia UML pro použití v modelování systémů nejsou tyto informace příliš významné a uvádíme je pouze pro úplnost.

Dále se budeme zabývat zejména popisem vrstvy označené jako metamodel.

2.2 Struktura svazků vrcholové úrovně v UML (top-level packages)

Celý metamodel UML je rozložen do útvarů zvaných angl. *package.* V této knize je zvolen český ekvivalent *„package = svazek“*. Představme si pod svazkem (package) skupinu elementů sdruženou do tohoto útvaru - svazku, který má funkci obdobnou jako knihovna v programování. Celý model se rozdělí do několika „knihoven“, tj. svazků, které sdružují elementy patřící významově k sobě. Mezi svazky existuje vztah skládání, tj. jeden svazek (package) může obsahovat jiný svazek (package).

Poznámka: Zavedení pojmu svazek (package) odpovídá plně zobecnění přístupu pomocí knihoven. Můžeme si představit, že jeden svazek reprezentuje knihovnu v modelování podobně jako knihovna zdrojového kódu reprezentuje knihovnu v programování. Z tohoto hlediska bychom mohli zavést také zobecnění pojmu „přilinkovat si knihovnu“ i v modelování podobně jako ve zdrojovém kódu. Vznikají tak jednosměrné vztahy popsané v dalším odstavci.

Mezi svazky (packages) existuje kromě skládání druhý základní jednosměrný vztah zvaný angl. *Dependency* (zde přeloženo jako *Závislost*). Dva svazky jsou závislé, tedy ve vztahu *Dependency*, pokud alespoň jeden prvek z daného svazku potřebuje ke své plné funkcionalitě alespoň jeden prvek z druhého svazku. Nesmíme zaměňovat dva vztahy *Závislost (Dependency)* a implicitní skládání

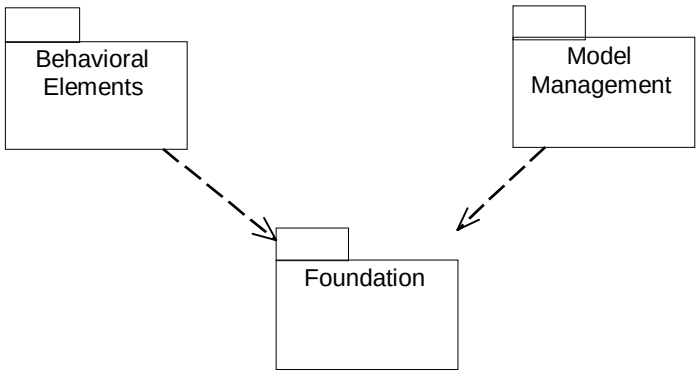
svazků (implicitní je skládání svazků proto, že svazek může obsahovat obecně prvky modelu, přičemž prvkem modelu může být zase prvek typu svazek).

Vztah dependency je ze zásady jednosměrný (kdo je na kom závislý), UML však nezakazuje zavést závislost do kruhu (*circular reference*). Například mezi dvěma svazky A a B závislými do kruhu by se kruhová závislost vyjádřila tak, že A je ve vztahu *Dependency* vůči B a současně B je ve vztahu *Dependency* vůči A. Tato konstrukce sice není přímo zakázána, ale je třeba si uvědomit, že vede k efektu „siamských dvojčat“. Pokud pracujeme s A, musíme pracovat s B a naopak, například při exportu částí modelu musí jít tyto dva svazky vždy spolu. Vyplývá z toho, že takovéto rozdělení na dva svazky je pouze formální.

Na nejvyšší úrovni dekompozice metamodelu UML se zavádějí *svazky vrcholové úrovně, top-level packages*, kterými jsou:

- Package *Behavioral Elements* (svazek *Prvky chování*)
- Package *Model Managment* (svazek *Řízení modelu*)
- Package *Foundation* (svazek *Základ*)

Graficky se dá vztah mezi uvedenými svazky (packages) vyjádřit diagramem, viz obrázek 1 Základní svazky metamodelu UML na nejvyšší úrovni



obrázek 1 Základní svazky metamodelu UML na nejvyšší úrovni

Poznámka: Uvnitř těchto svazků se nacházejí další svazky.

Na uvedeném obrázku jsou zavedeny tři svazky vrcholové úrovně a také je znázorněna závislost mezi nimi, tj. vidíme také *Dependency* mezi svazky vrcholové úrovně UML.

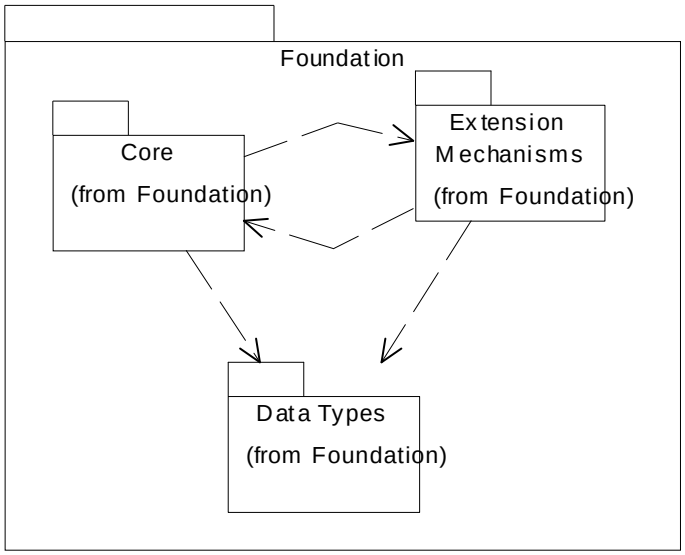
Poznámka: Směr šipky ukazuje Závislost „koho potřebuji“.

V souvislosti s tímto diagramem je třeba si uvědomit, že se jedná o vrcholovou úroveň svazků (*top-level packages*), a proto vidíme v této jednoduché podobě rozložení opravdu celého UML. Jakýkoliv prvek z UML tedy spadá do jednoho z těchto tří svazků.

Poznámka: Připomeňme při té příležitosti zásadu modelování s možností jemnějšího a hrubšího pohledu na systém

Šipky ukazují, který svazek metamodelu používá jiné prvky metamodelu z jiného svazku. Je vidět, že svazek *Foundation* je obdobou „centra“ pro ostatní dva svazky *Behavioral Elements* a *Model Managment*, protože „nepotřebuje“ žádný jiný svazek. Naopak tyto dva svazky *Behavioral Elements* a *Model Managment* potřebují ke své plné funkcionalitě prvky svazku *Foundation*.

Package *Foundation* (Svazek *Základ*) se dále dělí na další svazky, viz obrázek 2 Skladba svazku



obrázek 2 Skladba svazku Základ

Svazek *Základ* (Package *Foundation*) se skládá ze tří dalších svazků:

- *Core* (*Jádro*)
- *Extension Mechanisms* (*Mechanismy extenze*)
- *Data Types* (*Datové typy*)

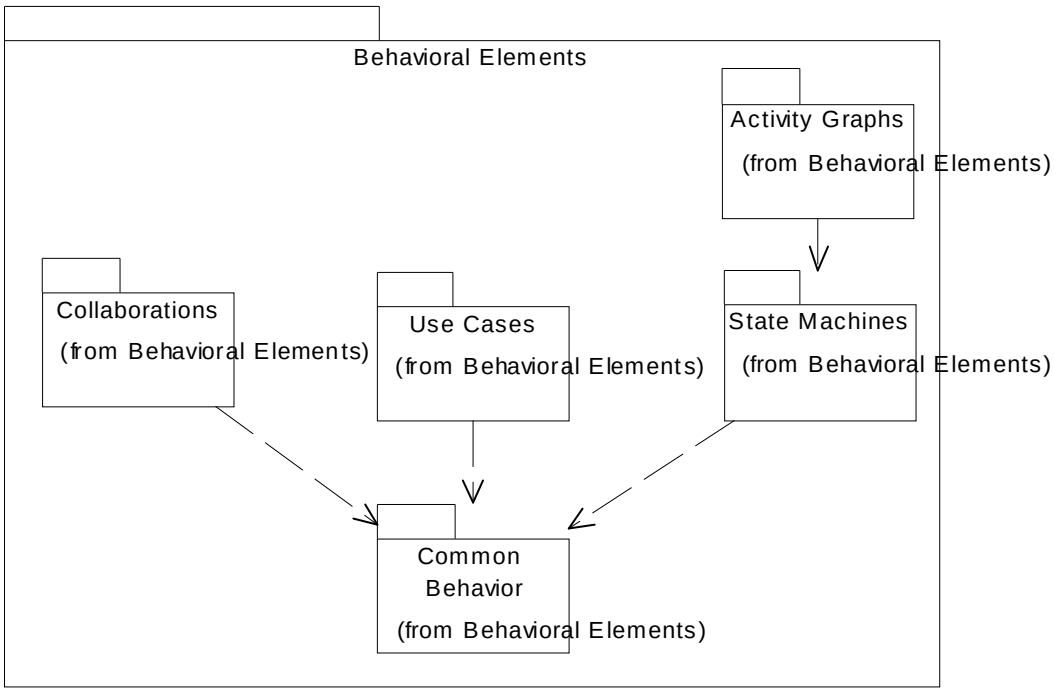
Opět ze vztahu *Dependency* mezi zobrazenými svazky vyčteme (viz zmíněný obrázek 2 Skladba svazku) dvě zajímavé skutečnosti:

- Mezi svazky *Core* a *Extension Mechanisms* (*Jádro* a *Mechanismy extenze*) existuje vztah *Dependency* do kruhu. Znamená to, že tyto dva svazky jsou odděleny pouze formálně, protože „stejně musí chodit všude spolu“.
- Svazek *Data types (Datové typy)* je svazkem nezávislým uvnitř svazku *Foundation* a ostatní dva svazky, *Core* a *Extension Mechanisms,* jsou na něm závislými. Protože na vrcholové úrovni svazků je svazek *Foundation* nezávislým a ostatní dva svazky vrcholové úrovně (*Behavioral elements* a *Model managment*) jsou na něm závislé, znamená to, že svazek *Datové typy* (*Data types*) je svazkem „jádro všech jader“.

Podobně také svazek *Behavioral elements* (svazek *Prvky chování*) se skládá z dalších svazků. Jsou jimi:

- Package *Common Behavior* (svazek *Společné chování*)
- Package *Collaborations* (svazek *Spolupráce*)
- Package *Use Cases* (svazek *Užitné činnosti*)
- Package *State Machines* (svazek *Stavové stroje*)
- Package *Activity Graphs* (svazek *Grafy aktivit*)

Rozložení těchto svazků uvnitř svazku *Prvky chování* ukazuje další diagram, viz obrázek 3 Skladba svazku Behavioral Elements



obrázek 3 Skladba svazku Behavioral Elements

Ve svazku *Common Behavior* (*Společné chování*) jsou umístěny „společné“ elementy používané ostatními svazky v rámci svazku *Prvky chování (Behavioral Elements*). Svazek *Activity Graphs* (*Grafy aktivit*) je napojen přímo na svazek *State Machines* (*Stavové stroje).*

Nyní si rozebereme jednotlivé svazky s popisem jejich obsahů.

3. Popis sémantiky metamodelu UML

3.1 Package *Foundation* (svazek *Základ*)

Připomeňme, že svazek *Základ* (package *Foundation*) patří do vrcholové úrovně svazků metamodelu UML spolu se svazky *Prvky chování* a *Řízení modelu (Behavioral Elements a Model Managment),* viz obrázek 1 Základní svazky metamodelu UML na nejvyšší úrovni.

Svazek *Základ* (*Foundation package*) reprezentuje statickou strukturu modelů a zavádí základní elementy modelů společné pro všechny modely. Skládá se ze svazků *Jádro (Core*), *Mechanismy extenze (Extension Mechanisms*) a *Datové typy (Data Types).* Struktura tohoto svazku je dána diagramem, viz obrázek 2 Skladba svazku .

Svazek *Jádro* (package *Core*) zavádí základní koncepty metamodelu UML, tj. nejobecnější elementy modelování s následnou možností přidání dalších případných konstrukcí jazyka (například dalších typů elementů modelu).

Svazek *Mechanismy extenze* (package *Extension Mechanisms*) specifikuje, jak je možné provést vlastní úpravy a dodatky, tedy uživatelské extenze modelů v souhlase se sémantikou UML.

Jak název napovídá, svazek *Datové typy* zavádí základní datové typy jazyka UML.

3.2 Package *Core* (svazek *Jádro)*

Svazek *Jádro* (package *Core*) patří k velmi důležitým svazkům celého metamodelu UML. Zavádí kromě jiného také *základní abstraktní model jazyka UML a také pravidla pro jeho potomky.*

Pomocí tohoto přístupu se exaktně definuje „kostra“ (*„skeleton“*) metamodelu UML a tato část modelu se označuje jako páteř (*„backbone“*) celého metamodelu UML. Tuto část metamodelu stejně jako ostatní části zobrazíme vcelku a poté po částech, přičemž provedeme rozbor detailů.

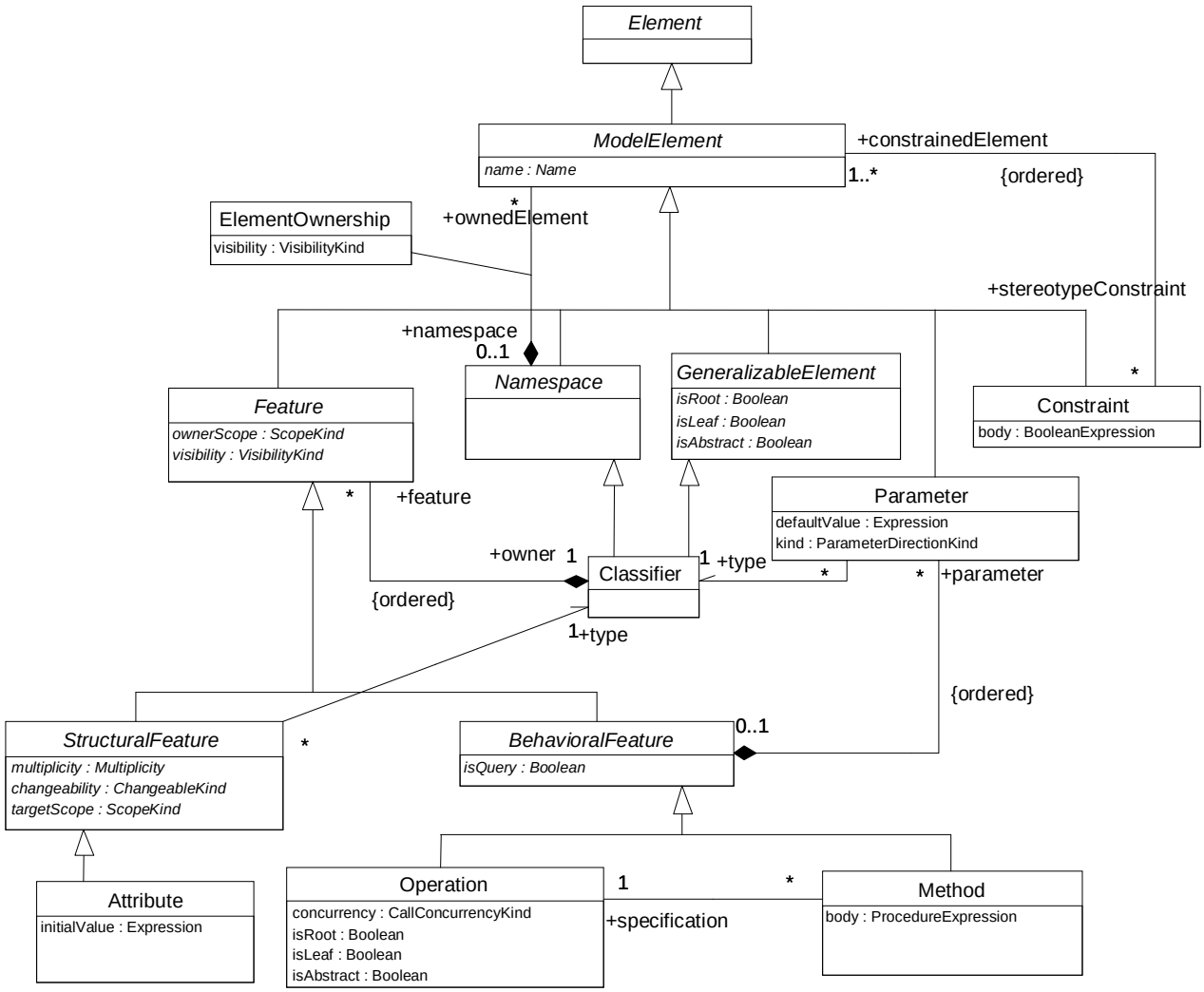
Celý svazek *Jádro* (package *Core*) obsahuje několik částí modelu UML vyjádřených několika diagramy. Každý z diagramů se týká nějaké své specifické oblasti a vyjadřuje určitou jednu nebo několik podstatných myšlenek.

Diagramy popisující *Jádro* jsou následující:

- páteř UML (*backbone*)
- relace (*relationships*)
- klasifikátory (*classifiers*)
- závislosti (*dependencies*)
- prvky příslušenství (*auxiliary elements*)

Nejprve si všechny tyto diagramy zobrazíme v celistvosti a poté je postupně rozebereme.

Celkový pohled na část modelu označovanou jako „páteř UML“ (což je pouze jedna část svazku *Jádro*), ukazuje následující diagram, viz obrázek 4: tzv. páteř metamodelu UML ve svazku Jádro (package Core)



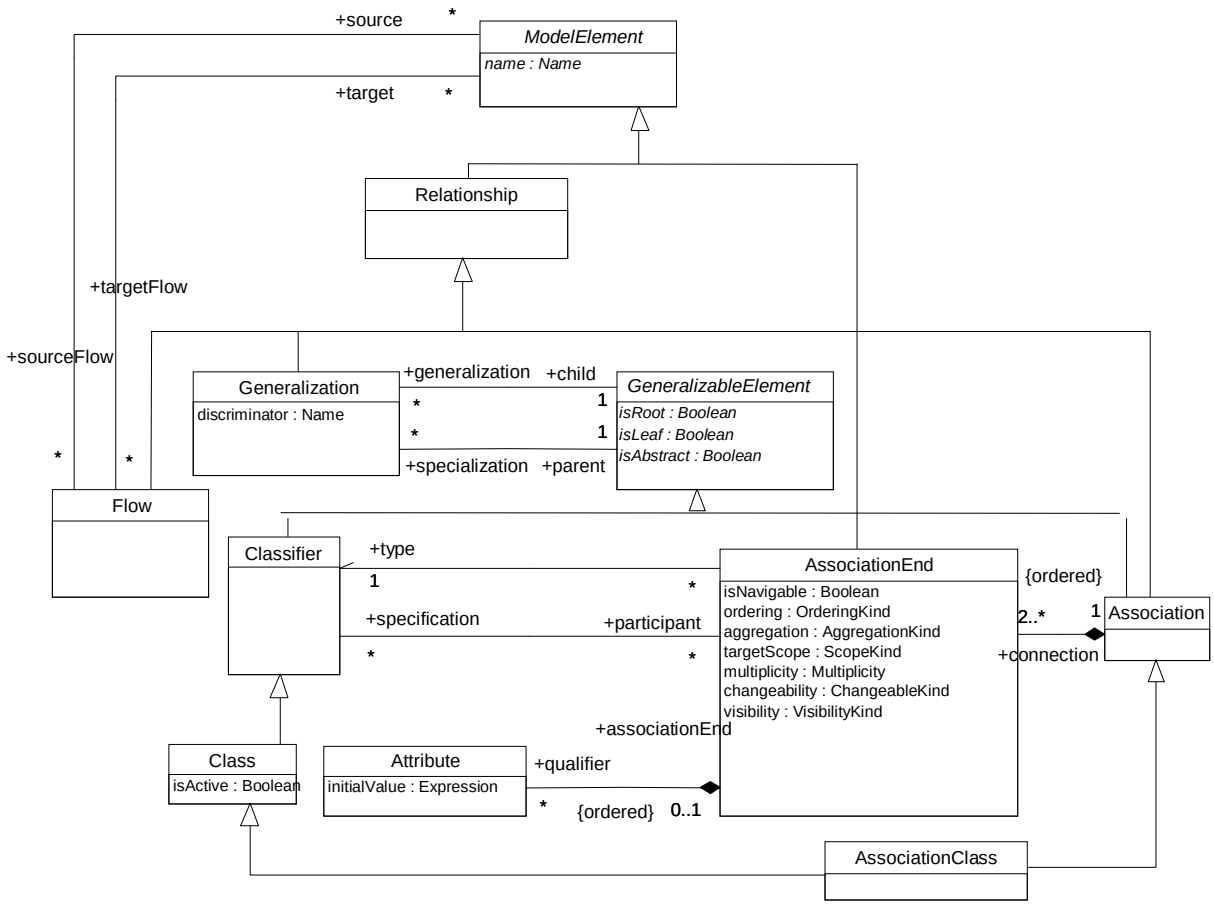
obrázek 4: tzv. páteř metamodelu UML ve svazku Jádro (package Core)

Znázorněná část metamodelu UML zavádí základní pojmy modelu jako jsou *Prvek* (*Element*), *Prvek modelu* (*ModelElement*), *Pojmenovaný prostor* (*Namespace*), *Klasifikátor* (*Classifier*) apod.

Prvky uvedeného diagramu včetně vztahů mezi nimi budou vysvětleny podrobně v dalších kapitolách.

Další částí svazku *Jádro* (package *Core*) je ta část modelu, která zavádí tzv. *relace* (*relationships*).

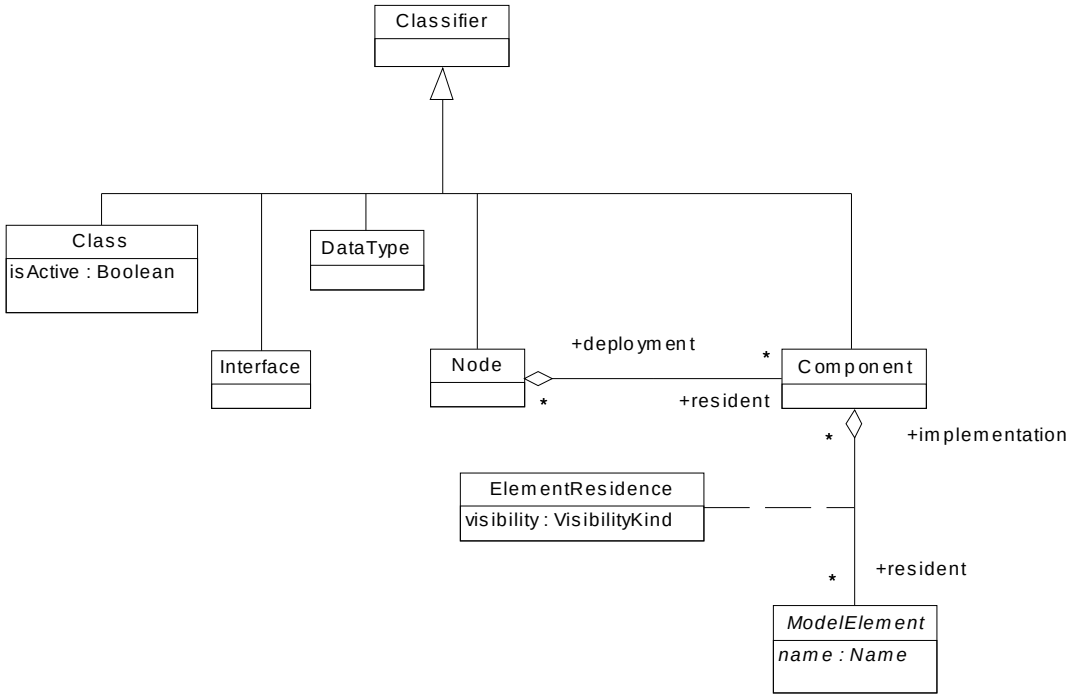
Tato část metamodelu UML je zobrazena na dalším diagramu, viz obrázek 5: část metamodelu z Jádra (Core) určující tzv.relace.



obrázek 5: část metamodelu z Jádra (Core) určující tzv.relace, diagram Jádro - Relace

Uvedená část metamodelu nazvaná *Jádro - Relace* (*Core – Relations*) na předešlém obrázku (obrázek 5: část metamodelu z Jádra (Core) určující tzv.relace, diagram Jádro - Relace), zavádí základní prvky pro modelování, které reprezentují vztahy mezi prvky modelu (tj. jedná se o prvky modelu vyjadřující relace v modelu). Rozbor částí tohoto metamodelu provedeme v dalších kapitolách.

Jako další část metamodelu ze svazku *Jádro* (*Core*) se zavádí tzv. *Klasifikátory* (*Classifiers*).



obrázek 6: část modelu z Core (Jádro) - Classifiers (Klasifikátory)

Klasifikátor (*Classifier)* je postaven na horní úrovni abstrakce pro několik svých potomků, viz již zmíněná část modelu *Core - Classifiers*, obrázek 6: část modelu z Core (Jádro) - Classifiers (Klasifikátory), Ve zkratce řečeno, pod pojmem *Klasifikátor* máme na mysli zobecnění pojmu třída z objektového programování.

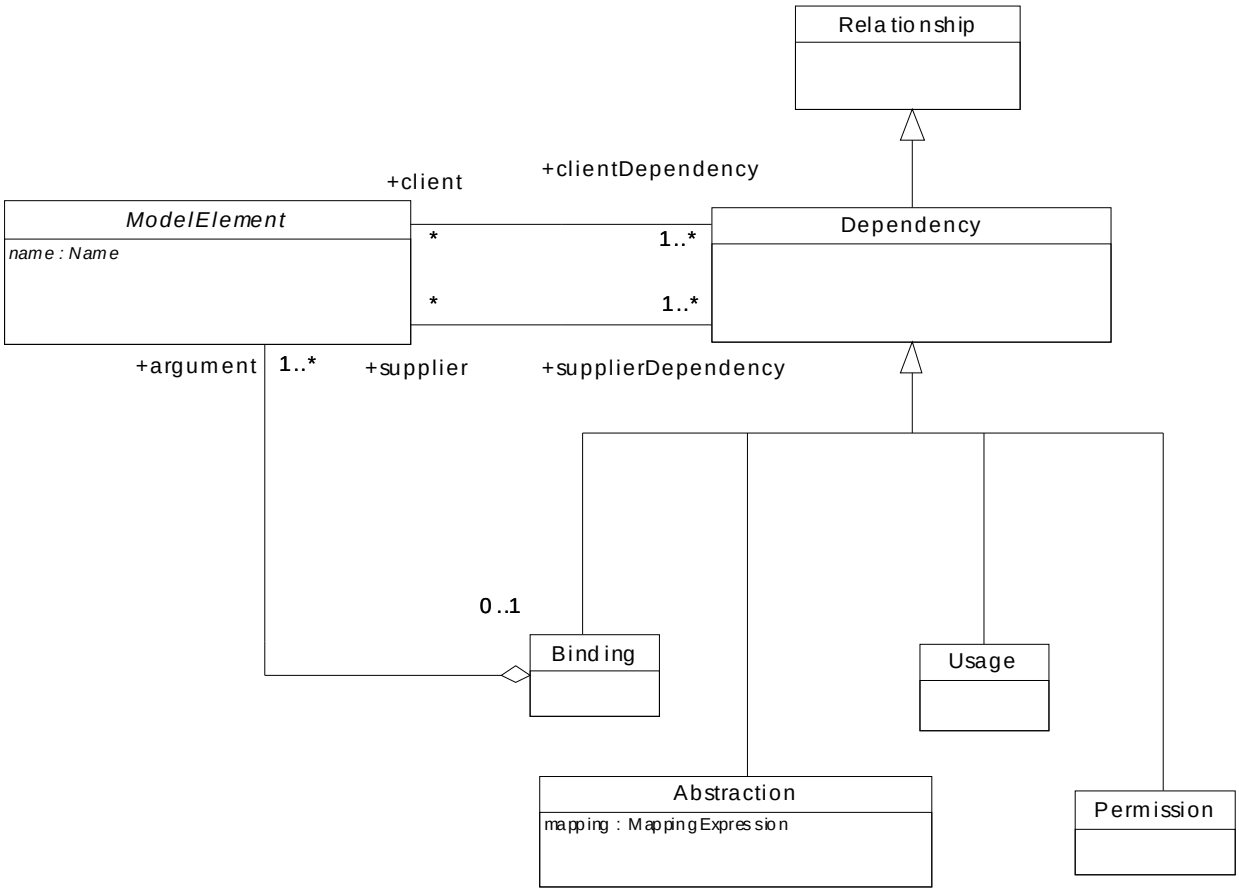
Poznámka: V publikacích o OOP na serveru http://www.objects.cz se u vysvětlení třídy uvádí, že třída je zavedena v OOP jako „forma, kopyto“ pro objekty, tj. dávající vlastnosti svým prvkům.

Klasifikátor (*Classifier*) je zobecněním i pro jiné typy instancí, nesoucích nějakou strukturu a chování, než jsou pouze klasické „objekty“vzniklé jako instance ze tříd (*Classes*). Zavádějí se další „formy, kopyta“ pro další instance s chováním a strukturou, tj. které také mohou nést informaci o chování nebo struktuře, tzv. *Klasifikátory,* a to potomci :

- *Data Type (Datový typ)* kde instancí datového typu je proměnná („čistá data“)
- *Interface (*česky také *Interface)*
- *Node* (*Uzel*), kde instancí je instance uzlu (zde je uzel chápán jako implementační jednotka systému, například stroj, procesor apod.)
- *Component* (*Komponenta*), kde instancí je instance komponenty

Další částí metamodelu v rámci svazku *Jádro* je diagram nazývaný jako *Dependencies* (*Závislosti*).

Tato část metamodelu UML je vyjádřena pomocí obrázku následovně – viz obrázek 7: část modelu UML zavádějící vztahy typu Dependency (Závislost)



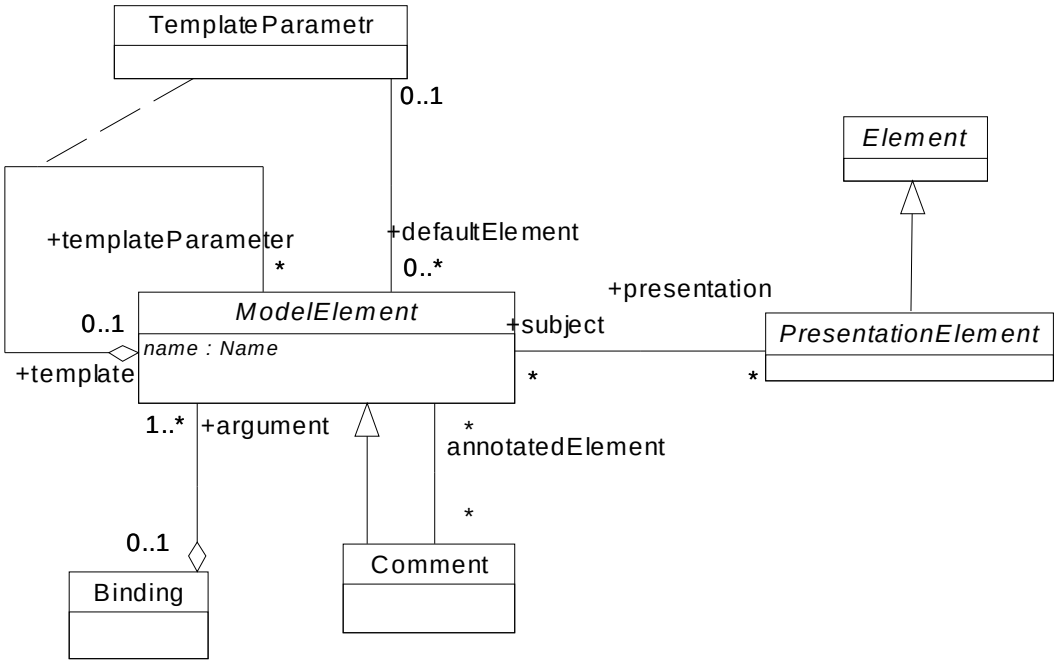
obrázek 7: část modelu UML zavádějící vztahy typu Dependency (Závislost)

Tato část modelu UML zavádí možnost používat vztahy mezi prvky modelu typu *Dependency* (*Závislost*). Všimněme si, že *Dependency* (*Závislost*) je potomkem prvku *Relationship* (*Relace*). Sama *Závislost* (*Dependency*) se dále dělí na různé podtypy vyjádřené další úrovní ve stromu generalizace na tyto úrovně:

- *Binding* (*Vazba*)
- *Abstraction (Abstrakce)*
- *Usage* (*Užívání*)
- *Permission* (*Povolení*)

V tomto výčtu je důležité si uvědomit, že každý z nich je potomkem metatřídy *Závislost* a tedy také metatřídy *Relace* (*Relationship*).

Posledním diagramem popisujícím obsah svazku *Jádro* (*Core*) je diagram nazvaný jako *Prvky příslušenství (Auxiliary elements)*. Tento diagram je uveden na následujícím obrázku, viz obrázek 8: Diagram Prvky příslušenství - Auxiliary elements



obrázek 8: Diagram Prvky příslušenství - Auxiliary elements

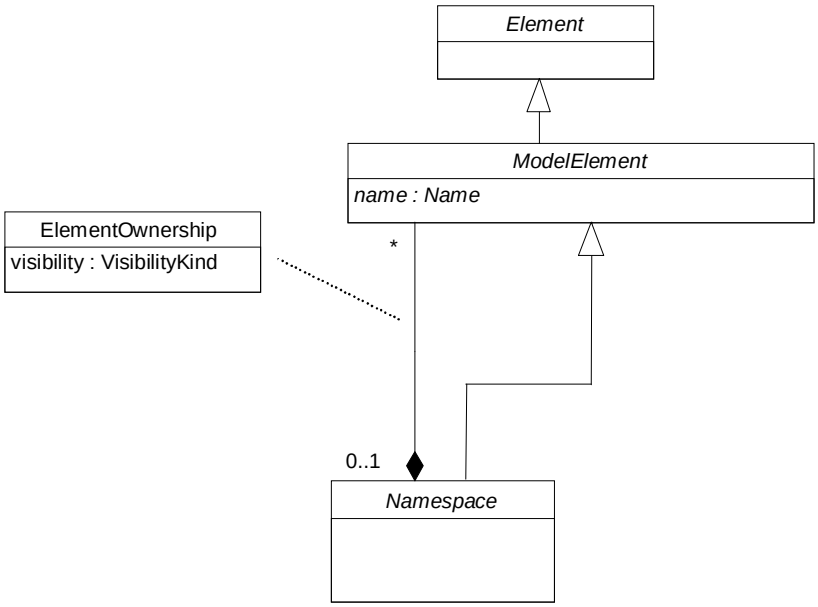
Na tomto diagramu je velmi důležitý vztah mezi *Prvkem modelu* (*ModelElement*) a *Prvkem Prezentačním* (*PresentationElement*). Oba dva jsou totiž přímými dědici „nejvyšší“ abstrakce metamodelu UML, kterým je *Prvek* (*Element*), viz obrázek 4: tzv. páteř metamodelu UML ve svazku Jádro (package Core). Vztah mezi *Prvkem modelu* a *Prvkem prezentačním* slouží k následnému přechodu ke grafické prezentaci modelu, kdy každý *Prvek model* (také Relace jsou Prvky modelu) obsahuje („vidí“) svůj jeden nebo více *Prvků prezentačních*. Tím je na nejvyšším stupni abstrakce metamodelu vyjádřen prostý fakt, že prvky modelu UML lze namalovat do diagramů a tento vztah určuje, který prvek v prezentaci *Prvek prezentační* odpovídá kterému *Prvku modelu*.

V další části kapitoly je podrobně popsán předešle uvedený metamodel ze svazku *Jádro* (*Core*).

Výklad zahájíme popisem určitých částí tzv. páteře UML (*backbone*).

3.2.1.1 Element (Prvek)

Z obrázku obrázek 4: tzv. páteř metamodelu UML ve svazku Jádro (package Core) vyjmeme část zavádějící *Element (Prvek)* a *Namespace (Pojmenovaný prostor)*:



obrázek 9: Vztah Prvku modelu a Namespace

Tato konstrukce je velmi zajímavá. Zavádí hned čtyři prvky v modelování. Nejvyšší je abstraktní *Prvek (Element)*. Pomocí diagramu „páteře UML“ se v metamodelu UML vlastně grafickým způsobem vyjadřuje skutečnost, že „v modelu UML je (mimo jiné) každý prvek modelu *Prvkem*“.

Namespace (*Pojmenovaný prostor*) je chápán jako *Prvek modelu* (protože je potomek metatřídy *ModelElelement*, tj. *Prvku modelu*) a *Prvky modelu* mohou a nemusí být obsaženy jedinečně v *Namespace (*vztah je ze strany metatřídy *Namespace* kardinality 0..1).

3.2.1.2 ModelElement (Prvek modelu)

Prvek (*Element*) je atomickou složkou každého modelu. Jak je vidět na diagramu, v metamodelu má *Prvek* (*Element*) postavení jako nejvyšší metatřída mezi ostatními metatřídami ve stromu generalizace. Je to vrchol metamodelu UML.

Prvek je abstraktní metatřídou a má dva potomky ve stromu generalizace:

- první je metatřída *ModelElement* (*Prvek modelu,* viz obrázek 9: Vztah Prvku modelu a Namespace resp. obrázek 4: tzv. páteř metamodelu UML ve svazku Jádro (package Core))
- druhý je metatřída *PresentationElement* (*Prvek prezentační* viz obrázek 8: Diagram Prvky příslušenství - Auxiliary elements)

Protože *Prvek modelu* obsahuje atribut *name (název),* každý potomek *Prvku modelu* má svůj *název*.

Poznámka: Například vztah Generalizace (Generalization) má z tohoto důvodu také svůj název, i když se většinou v modelech nepoužívá.

3.2.1.3 Namespace (Pojmenovaný prostor)

Namespace (Pojmenovaný prostor) je *Prvek modelu*, označující určitou vymezenou část modelu. Tato část modelu obsahuje množinu *Prvků modelu* daného *Pojmenovaného prostoru* (jedním z těchto vnitřních prvků může být opět *Pojmenovaný prostor*).

Je důležité si uvědomit, že i *Namespace* (*Pojmenovaný prostor*) je *Prvkem modelu* jako každý jiný *Prvek modelu* a tedy práce s ním je součástí modelování v UML. Použití *Pojmenovaných prostorů* (*Namespace*) rozděluje celý model do přesně definovaných částí modelu.

Všimněme si na diagramu na obrázek 9: Vztah Prvku modelu a Namespace, že vztah mezi metatřídami *Namespace* a *ModelElement* (*Pojmenovaný prostor* a *Prvek modelu*) je namalován jako tzv. *komposice* (tj. existuje černě vybarvený kosočtverec u metatřídy *Namespace*). Tato notace znamená, že jeden daný *Prvek modelu* vzniká a je vlastněn pouze v tomto *Pojmenovaném prostoru*. Pro daný *Prvek modelu* (*ModelElement*) existuje tedy pouze jeden „domovský“ *Pojmenovaný prostor* (*Namespace)*.

Poznámka: Kromě mechanismu dělení modelů pomocí Pojmenovaných prostorů, existuje ještě druhý velmi podobný mechanismus dělení modelů pomocí svazků (package), které jsou potomky Pojmenovaných prostorů.

3.2.1.4 ElementOwnership (Vlastnictví prvku)

Všimněme si na diagramu na obrázek 9: Vztah Prvku modelu a Namespace asociativní metatřídy označené jako *ElementOwnership* (*Vlastnictví prvku*) . *Vlastnictví prvku* je zavedeno proto, aby

definovalo tzv. *Viditelnost* (*Visibility*) *Prvku modelu* vzhledem k danému *Pojmenovanému prostoru* (*Namespace*).

Visibility (*Viditelnost*) je vlastností, která se zavádí i pro jiné účely, než je definice viditelnosti v rámci *Namespace. Viditelnost* (*Visibility*) může nabývat pouze těchto hodnot:

- *public* - libovolný *Prvek modelu* vně *Prvku modelu* vidí daný *Prvek modelu*
- *protected* - libovolný potomek *Prvku modelu* vidí daný *Prvek modelu*
- *private* – daný *Prvek modelu* je viditelný pouze uvnitř *Prvku modelu*

Poznámka: Jak je známo z OOP, stejně je zavedena viditelnost atributů a metod ve třídách v některém z programovacích jazyků (Delphi, C++ apod.). V UML se samozřejmě s pojmem Visibility (Viditelnost) setkáme ještě několikrát včetně případu vztahu metod a atributů vůči třídám.

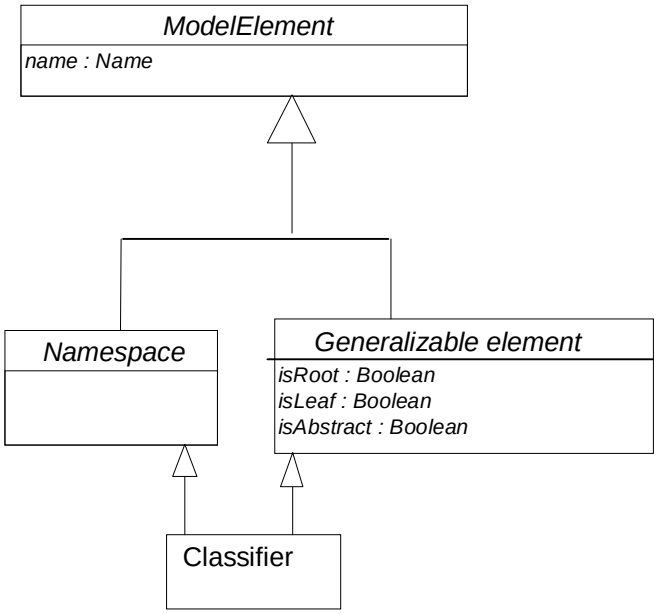
3.2.1.5 Classifier (Klasifikátor)

Další velmi důležitou částí modelu v „páteři UML“ je vyjádření vlastností *Klasifikátoru* (*Classifier*). *Klasifikátor* je takový prvek modelu, který obecně nese nějakou informaci o struktuře a chování.

Poznámka: Připomeňme, že Klasifikátor může být buď Datový typ, Třída, Interface, Uzel nebo Komponenta. V závěrečné kapitole k tomuto výčtu přibude ještě Subsystém.

Samotný *Klasifikátor* je abstraktní metatřídou a proto se v modelu mohou objevit až jeho potomci. Pro popis vlastností *Klasifikátoru* (a tím všech jeho potomků) vybereme opět tu část modelu, která se *Klasifikátoru* bezprostředně týká a poté ji popíšeme.

Následující diagram zobrazuje „odkud“ klasifikátor pochází:



obrázek 10: Původ Klasifikátoru

Z diagramu je patrné, že metatřída *Klasifikátor* (*Classifier*) je potomkem jak metatřídy *Namespace* (*Pojmenovaný prostor*), tak metatřídy *Generalizable element* (*Zobecnitelný prvek*).

Znamená to, že *Klasifikátor* může mít vlastnosti převzaty z obou těchto metatříd. Kromě toho je potomkem metatřídy *Prvek modelu (ModelElement),* viz o jednu úroveň abstrakce výše díky tranzitivnosti vztahu generalizace.

Metatřída *Zobecnitelný prvek* (*Generalizable element*) je definována takto: *Zobecnitelný prvek* je takový prvek, který může vstupovat do vztahu *Zobecnění* (*Generalization*). Podědění ze strany *Zobecnitelného prvku* (*GeneralizableElement*) vede k tomu, že obecný *Klasifikátor* a všichni jeho potomci mohou také vstupovat do vztahu *Zobecnění* (*Generalization*).

Podotkněme, že vztah *Generalization* je zaveden v jiné části svazku *Jádra* (viz obr. obrázek 5: část metamodelu z Jádra (Core) určující tzv.relace, diagram Jádro - Relace).

Všimněme si, že každý potomek metatřídy *GeneralizableElement* získává k dispozici tři atributy, které mohou doplnit jeho vlastnosti ve stromu generalizace:

- *IsRoot* – typu Boolean. Pokud je pravda, potom daný *Prvek modelu* je vrcholovým prvkem ve vztazích *Generalization* (nemá již předka, tj. nemá vyšší zobecnění).
- *IsLeaf* – typu Boolean. Pokud je pravda, potom daný *Prvek modelu* již nemá ve vztahu *Generalization* potomka, je posledním ve stromu. Pokud není pravda, nemusí sice potomek přímo v té chvíli existovat, ale je povoleno jej zavést.
- *IsAbstract* – typu Boolean. Pokud je pravda, potom daný prvek je na takové úrovni zobecnění, že nemá smysl zavádět instance z tohoto prvku. Znamená to, že v tomto případě se instance musí objevit až v některém z potomků tohoto prvku.

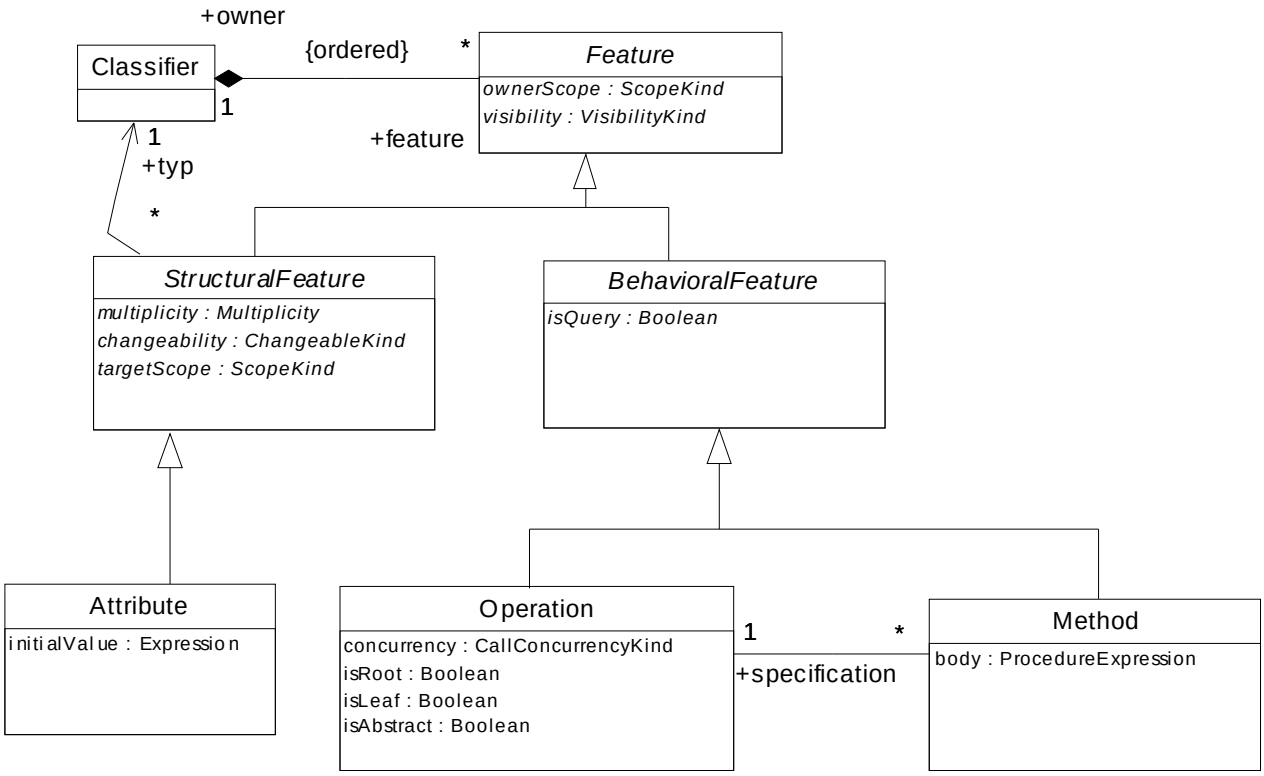
Poznámka: Z logiky věci vyplývá, že prvek nemůže být současně IsLeaf pravda a IsAbstract také pravda.

Z praktického hlediska nebývají informace *IsRoot, IsLeaf, IsAbstract* při samotném modelování mnohdy uvedeny, resp. se uvádějí až po dokončení modelu a vyjasnění všech vztahů. Důvodem je to, že tyto informace mají spíše povahu sice důležitého, ale přece jen pouze kontrolního omezení (tj. vyjadřují „co by nemělo nastat"). Nejedná se tedy o nějaké „kreativní informace", a proto se někdy v úvodních fázích tvorby modelu bez nějakých fatálních následků vynechávají. Avšak pro zachování konzistence modelu jsou uvedené informace důležité.

Z druhé strany je *Klasifikátor* (*Classifier*) také potomkem *Pojmenovaného prostoru* (*Namespace*), viz obrázek 10: Původ Klasifikátoru . Znamená to, že samotný *Klasifikátor* se může chovat jako *Pojmenovaný prostor*. V tomto případě je to umožněno proto, aby *Klasifikátor* mohl mít v sobě vnořeny jiné *Klasifikátory.* Toto vnoření *Klasifikátorů* do jiného *Klasifikátoru* je však povahy příslušnosti do *Pojmenovaného prostoru* a nejedná se v žádném případě o vztah asociace nebo agregace mezi *Klasifikátory.* Jinak řečeno, nezaměňujme tuto vlastnost vnořování („*nested Classifiers*") za vztah *Relace* mezi *Klasifikátory*. K daným vnořeným *Klasifikátorům* lze přistupovat podle nastavené vlastnosti *Visibility* (viz předešlé kapitola Namespace (Pojmenovaný prostor)).

Poznámka: Například bychom mohli zavést nějakou Komponentu X (která je také Klasifikátorem a tedy Pojmenovaným prostorem) a začali bychom k ní přistupovat při modelování jako k Pojmenovanému prostoru X, vnořovat do ní jiné Klasifikátory apod. Tento vztah vůbec nesouvisí se vztahy mezi Klasifikátory jako je Relace (například v konkrétním modelu tříd vztah mezi Fakturou a Řádkem faktury).

Hlavním posláním *Klasifikátoru* je zavést v modelu systému nějak informaci o struktuře a chování částí systému. Metatřída *Klasifikátor* proto vstupuje do vztahů s jinými metatřídami, které dávají *Klasifikátoru* právě tyto vlastnosti, viz následující obrázek 11: Klasifikátor jako nositel struktury a chování



obrázek 11: Klasifikátor jako nositel struktury a chování

Z uvedeného diagramu vyplývá, že metatřída *Classifier* (*Klasifikátor*) obsahuje ve vztahu kompozice N prvků metatřídy *Feature* (*Rys ve smyslu rys = vlastnost, črt).*

Feature může být buď *StructuralFeature* (*Rys struktury*) anebo *BehavioralFeature* (*Rys chování*).

Metatřída *BehavioralFeature* (*Rys chování*) má potomky *Operation* (*Operace*) a *Method* (*Metoda*).

Metatřída *StructuralFeature* (*Rys chování*) má potomka *Attribute* (*Atribut*).

Tímto modelem je plně vystižen *Klasifikátor* jako nositel informace o struktuře a chování.

3.2.1.6 Feature (Rys)

Feature (*Rys*) je vlastnost, která je zapouzdřena do *Klasifikátoru* a definuje tak vlastnost instance vzniklé z *Klasifikátoru*. *Feature* je abstraktní metatřídou, tedy v modelech se přímo nepoužívá, v modelování se používají až její potomci.

Atributy Rysu

- *name* (*název*), poděděn z *Prvku modelu* (*ModelElement*), musí být jednoznačný v rámci stromu dědičnosti *Klasifikátorů* včetně názvů příslušných *Konců Asociace* (o *Asociaci* a jejích *Koncích* viz dále)
- *ownerscope* (*rozsah platnosti držitele*) určuje, zda se daný *Rys* objeví ve všech instancích z *Klasifikátoru* anebo zda existuje pouze u samotného *Klasifikátoru*. Může nabývat dvou hodnot:
    - hodnota je „instance“ – každá instance *Klasifikátoru* vlastní svou hodnotu daného *Rysu.*
    - hodnota je „classifier“ – existuje pouze jedna hodnota daného *Rysu* pro daný *Klasifikátor* a je společná pro všechny instance *Klasifikátoru.*

Poznámka: U metatřídy Třída (Class), která je potomkem metatřídy Klasifikátor (Classifier) se používá synonymum pro tento stav ownerscope= „classifier“ jako metoda třídy resp. atribut třídy.
V programovacím jazyce se potom označují například rezervovaným slovem static resp. se deklarují v části CLASS SCOPE apod.

- *visibility* (v*iditelnost*), podobně jako u *Pojmenovaných prostorů* určuje, zda daný *Rys* je použitelný jinými *Klasifikátory*. Tato viditelnost se musí navíc zkombinovat s viditelností *Klasifikátorů* a teprve výsledek této kombinace je pro výslednou viditelnost určující (viz další popis). Viditelnost může nabývat těchto hodnot:
    - public – libovolný vnější *Klasifikátor*, který vidí daný *Klasifikátor*, může tento jeho *Rys* použít
    - protected – libovolný *Klasifikátor* - potomek může daný *Rys* použít
    - private – pouze *Klasifikátor* sám může daný *Rys* použít

3.2.1.7 StructuralFeature (Rys struktury)

Metatřída *Feature* má dva potomky, z nichž jeden je *StructuralFeature* (*Rys struktury*).

Rys struktury Klasifikátoru dává instancím *Klasifikátoru* resp. samotnému *Klasifikátoru* statické vlastnosti struktury, jako je *Atribut* (*Attribute*).

StructuralFeature je abstraktní metatřída, tedy v modelu se nepoužívají její instance, ale až instance jejích potomků (v tomto případě se jedná pouze o jednoho potomka, *Atribut*).

Protože v současné verzi metamodelu UML 1.3 existuje pouze jeden potomek metatřídy *Rys struktury* a tím je metatřída *Atribut,* budeme popisovat přímo vlastnosti *Atributu*.

Poznámka: Vyplývá z toho, že existuje abstraktní úroveň mezi metatřídou Feature a metatřídou Attribute a nazývá se StructuralFeature. Tato úroveň je z hlediska modelování „formálně“ zbytečná. Teoreticky by nemusela existovat, protože „nic nepřináší“ . Zavádí se pouze z důvodu flexibility – existuje úroveň dělení na „Rys struktury“ a „Rys chování“ jako symetrické dělení na dva Rysy, ale Rys struktury má pouze jednoho potomka, a tím je Atribut. V modelování pomocí UML je takovýto systematicky „taxonomický“ přístup velmi častý. Pro získání flexibility a stability je to přístup velmi účinný.

3.2.1.8 Attribute (Atribut)

Atributte (*Atribut*) je pojmenovanou strukturální částí *Klasifikátoru* držící ve své hodnotě nějaký stav tohoto *Klasifikátoru* resp. stavy instancí tohoto *Klasifikátoru*.

Metatřída *Atribut* má k dispozici tyto atributy (pozor, tento výčet souvisí i s převzetím atributů od předků metatřídy *Atribut*, zejména *Rysu struktury*), viz *obrázek 11: Klasifikátor jako nositel struktury a chování*

- *changeability (měnitelnost)* udává, zda může být hodnota *Atributu* měněna. Může nabývat těchto hodnot:
  - changeable (měnitelný), tj. hodnotu *Atributu* lze kdykoliv měnit
  - frozen (zmražen), tj. hodnota *Atributu* se nastaví po inicializaci a poté se nemůže měnit, tedy jedná se o obdobu konstanty. Pokud má *Atribut násobnost (Multipilicity)* větší než jedna (viz dále), potom nelze přidávat, mazat a měnit v množině hodnot tohoto *Atributu*.
  - addonly (pouze k přidání), tato hodnota je smysluplná pouze tehdy, pokud má *Atribut násobnost* větší než jedna. Tato hodnota značí, že lze pouze přidávat další hodnoty do množiny hodnot *Atributu*
- *InitialValue (počáteční hodnota)*, je *Výraz* (*Expression*) specifikující počáteční hodnotu po zrodu objektu.
- *multipilicity (násobnost)*, udává možný počet hodnot dat atributu. Nejčastější případ je tzv. skalár s násobností 1.

Poznámka: V programování by atributu s násobností větší než jedna odpovídalo zavedení vnitřního člena třídy (member) ve tvaru indexované proměnné, případně množiny apod.(například atribut jako pole array apod.). Při modelování informačních systémů musíme být vždy velmi opatrní při zavádění násobného atributu, protože ve valné většině násobnost mezi analytickými pojmy větší než jedna bývá signálem pro zavedení relace mezi třídami a nikoliv násobných atributů. Při zavádění násobného atributu musíme mít vždy jistotu, že násobné hodnoty nemají analytickou identitu (jako například Řádky faktury). Je zřejmé, že násobné atributy lze vždy převést na kompozici 1 : N zavedením třídy „obalující" atribut, což doporučuji.

- *targetscope* (*rozsah platnosti cíle*) určuje, zda je hodnota *Atributu* naplněnou instancí *Klasifikátoru,* anebo zda se jedná přímo o *Klasifikátor*. Možné hodnoty jsou
  - instance – hodnota je instancí *Klasifikátoru.*
  - classifier – hodnota ukazuje přímo na *Klasifikátor.*

Poznámka: Nesmíme zaměňovat ownerscope (rozsah platnosti držitele) a targetscope (rozsah platnosti cíle). Ownerscope je zaveden na úrovni abstraktní metatřídy Feature (Rys) pro všechny potomky a označuje polohu držení Atributu vzhledem k držiteli Atributu (zda je v instancích, anebo na úrovni Klasifikátoru). Připomeňme, že zvláštním případem je zavedení metod a atributů třídy. Vlastnost ownerscope určuje „kde je Atribut", zda v instancích nebo v Klasifikátoru (například atribut třídy, tj. static, nebo atribut instance, tj. dynamic).

Na rozdíl od toho targetscope určuje, „co“ je Atribut, zda instance Klasifikátoru anebo sám Klasifikátor. Nejčastějším případem hodnoty targetscope je instance pocházející z Datového typu.

Příklad na targetscope instance z Klasifikátoru Datový typ:

```
Class COsoba
Private mJmeno As String
...
End Class
```

Použitím instance z Klasifikátoru jako Třídy vznikne atribut „objekt“, což není příliš korektní konstrukce, protože modelováním můžeme správného vyjádření dosáhnout relacemi mezi třídami. Objektovou referenci potom zavede Konec Asociace. Jinak řečeno v pseudokódu zavedená konstrukce:

```
Class CPujcka
Private Rucitel As COsoba
..
End Class
```

není chápána jako zavedení atributu s názvem Rucitel (instance Rucitel z třídy COsoba), ale jako realizace jednoho Konce Asociace s názvem „Půjčka vidí svého Ručitele“

Použití targetscope s hodnotou nikoliv „instance“ ale „classifier“ znamená zavedení atributu rovnou jako Klasifikátoru, tj. zavede se atribut jako metainformace. Atributem je potom sám Klasifikátor (a nikoliv instance Klasifikátoru). U Datových typů jsem se nikdy s takovou konstrukcí nesetkal, ale u Tříd je mnohdy používaná. Ve většině OOP jazyků se musí takovýto vztah implementovat nějakou oklikou, protože není v programovacích jazycích většinou podporován.

3.2.1.9 BehavioralFeature (Rys chování)

BehavioralFeature (*Rys chování*) je potomkem *Feature* na stejné úrovni jako *StructuralFeature* (viz *obrázek 11: Klasifikátor jako nositel struktury a chování* a odpovídá dynamice chování. Specifikuje chování *Klasifikátoru*.

Metatřída *BehavioralFeature* je abstraktní třídou, tj. v modelech se používají až její potomci, metatřídy *Operation* a *Method* (*Operace* a *Metoda)*.

Atributy BehavioralFeature

- *name* (*název*) získal od předka *Prvku modelu (ModelElement)*
- *isQuery (je dotazem),* Boolean. Pokud je pravda, tak chování odpovídající tomuto prvku nemění stav systému („chování pouze čte a nemění stavy“).

3.2.1.10 Operation (Operace)

Operace je služba, kterou můžeme vyžadovat po nějakém objektu, abychom docílili nějakého požadovaného chování.

Připomeňme, že v metamodelu UML je *Operace* potomkem *Rysu chování* a tranzitivně *Rysu*, který je v kompozici s *Klasifikátorem* viz obrázek 11: Klasifikátor jako nositel struktury a chování. Tedy v předešlé definici je pod objektem míněna buď instance *Klasifikátoru* anebo sám *Klasifikátor* podle hodnoty atributu *ownerscope.*

Poznámka: Jinak řečeno, lze zavést Operace společné pro všechny instance na úrovni Klasifikátoru, například jako operace třídy.

Atributy Operation

- *concurrency (souběžnost)* určuje pravidla pro konkurenční (souběžné) volání dané *Operace* několika klienty objektu současně. Může nabývat těchto hodnot:
  - sequential (sekvenční). Volající musí koordinovat svá volání, protože operace není pro současné volání nikterak uzpůsobena. Pokud nedojde ke koordinaci, není zaručena stabilita a integrita systému.
  - guarded (chráněné). Volající mohou volat souběžně, avšak vždy bude v daném okamžiku obsloužen pouze jeden klient a volání od ostatních je dočasně blokováno, případně frontováno apod. Musí být v designu zabezpečeno, že tento režim nezpůsobí deadlock, resp. předčasné „odpojení klienta“ kvůli dlouhé odezvě objektu apod.
  - concurrent (souběžné). Jsou umožněna násobná volání z různých threadů. Objekt poskytující službu *Operace* je navržen tak, že je i přes souběžné volání zaručena stabilita a integrita systému.
- IsAbstract – pokud je pravda, potom daná *Operace* není přímo implementována. Musí být implementována v některém z potomků daného *Klasifikátoru*.
- IsLeaf – pokud je pravda, potom implementace této *Operace* nemůže být již přepsána (*overriden*) potomky daného *Klasifikátoru*. Pokud není pravda, potom může být přepsána (ale nemusí).
- IsRooot – pokud je pravda, potom třída nesmí podědit deklaraci *Operace*, pokud není pravda, může (ale nemusí) podědit deklaraci *Operace*.

3.2.1.11 Method (Metoda)

Metoda je implementací *Operace*. Všimněme si na diagramu obrázek 11: Klasifikátor jako nositel struktury a chování, že vztah mezi *Operací* a *Metodou* (*Operation* a *Method*) je 1 ku N a nejedná se o agregaci.

Základní rozdíl mezi *Operací* a *Metodou* je v tom, že *Operace* ukazuje „navenek“ uživateli objektu, jakou službu chování může u objektu použít, aniž by se nějak ukazoval vnitřek při poskytnutí této služby. Naopak implementace *Metodou* je již realizace této služby v konkrétní implementaci.

Tedy *Operace* je synonymem „co se může volat“ a *Metoda* je synonymem „jak se to provede“.

Všimněme si, že pokud je *Operace* abstraktní (tj. vyjádřeno pomocí metamodelu takto: Její atribut má hodnotu IsAbstract jako pravda), znamená to, že tato *Operace* nemá implementaci a zavádí ji až potomek. V objektovém programování je tato situace velmi často používaná.

Atributy Metody

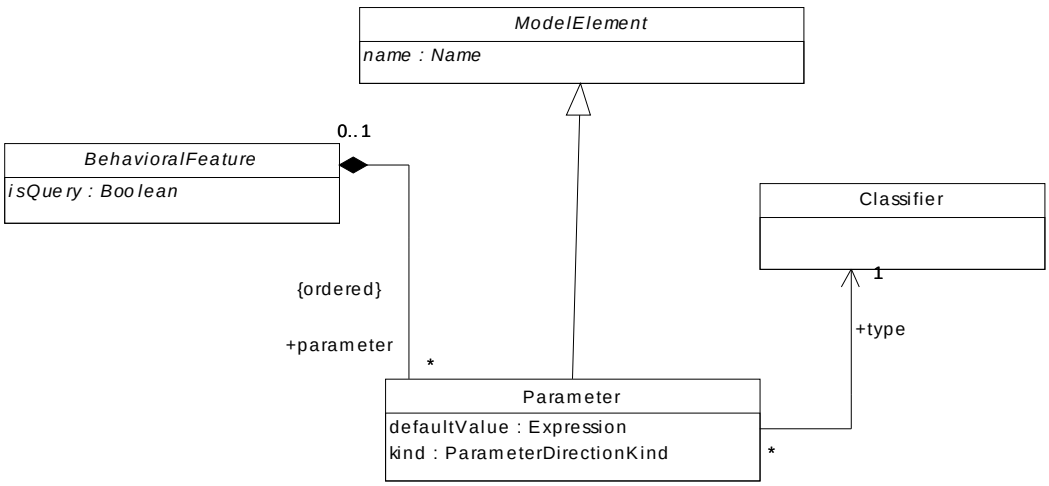
Metatřída *Metoda* má svůj jediný atribut a tím je body:

- *body* (tělo) – implementace *Metody* jako *ProcedureExpression* (*Procedurální výraz*)

Poznámka: ProcedureExpression (Procedurální výraz) je metatřídou zavedenou ve svazku Datové typy (bude dále pojednáno).

3.2.1.12 Parameter (Parametr)

Další část diagramu zavádí tzv. *Parametr* (*Parameter*), viz obrázek 12 Zavedení Parametru



obrázek 12 Zavedení Parametru

Všimněme si, že *Parametr* je přímým potomkem *Prvku modelu* (*ModelElement*). Metatřída *BehavioralFeature* (*Rys chování*) je ve vztahu *Kompozice* jedna ku N vůči *Parametru.* Znamená to, že každý *Rys chování*, tj. *Operace* nebo *Metoda*, může obsahovat N *Parametrů*.

Navíc je z diagramu jasné, že každý *Parametr* vidí svůj typ jako *Klasifikátor,* tj. může jím být například *Třída* nebo *Datový typ*.

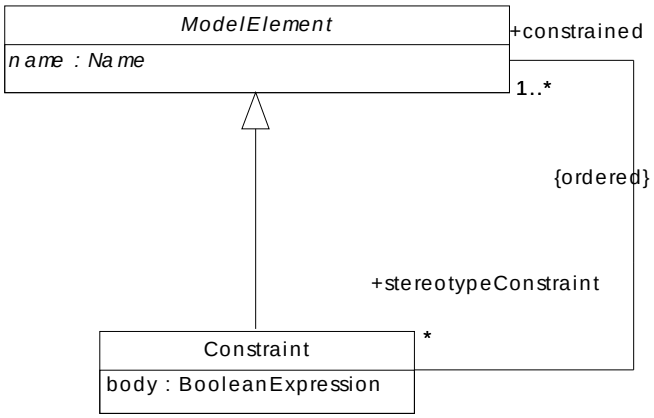
Atributy Parametru

- defaultValue (počáteční hodnota) je *Expression* (*Výraz*), jehož vypočtená hodnota se použije, pokud není dodána hodnota *Parametru*.

- kind (druh) specifikuje druh *Parametru* a může nabýt těchto hodnot:
  - in – vstupní *Parametr*, hodnota se nemění
  - out – výstupní, hodnota se mění jako informace pro volajícího
  - inout – vstupně výstupní, je vstupní *Parametr* s měnící se hodnotu jako informace pro volajícího
  - return – návratová hodnota
- name (název), poděděno z *Prvku modelu*

3.2.1.13 Constraint (Omezení)

V modelech UML se používá tzv. *Omezení* (*Constraint*), které je zavedeno v metamodelu UML podle obrázku obrázek 13 Zavedení Omezení (Constraint)



obrázek 13 Zavedení Omezení (Constraint)

Prvek z metatřídy *Omezení* (*Constraint*) vyjadřuje nějaké omezení, které je obsaženo jako výraz v jeho atributu body a vyjadřuje podmínku, která musí být splněna pro správnost fungování navrženého modelu.

Omezení (*Constraint*) je přímým potomkem *Prvku Modelu* (*ModelElement*) a je ve vztahu vůči *Prvku modelu* v násobné asociaci.

Povšimněme si, že ze strany *Omezení* je násobnost „hvězdička“, tedy N. Ze strany *Prvku Modelu* (*ModelElement*) je násobnost vztahu 1 až N. Tedy pokud *Constraint* existuje, je přiřazen alespoň k jednomu *Prvku modelu*. Protože *Omezení* je ve vztahu vůči abstraktnímu *Prvku modelu*, může být přiřazeno k libovolnému *Prvku modelu* (*Omezení* tedy může „použít“ libovolný *Prvek modelu)*.

Atributy Omezení (Constraint)

- *body* (tělo) je Booleanovský výraz (*BooleanExpression*). Pokud není pravda, potom systém není konzistentní

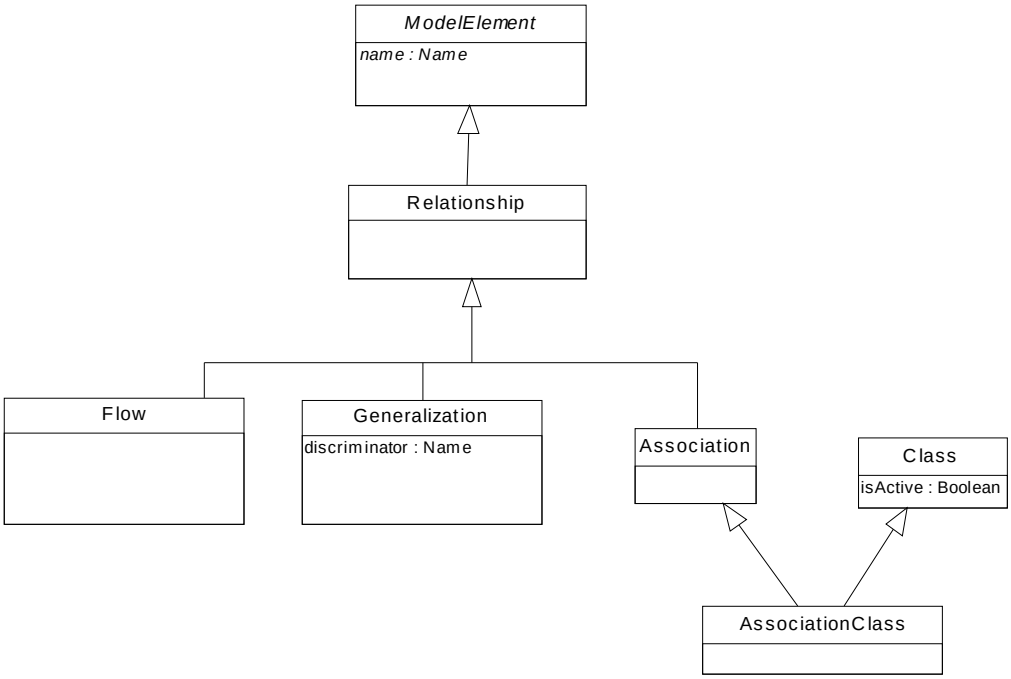
Poznámka: Constraint (Omezení) v UML má v podstatě stejné použití, jako jeho obdoba zavedená v relačních databázích. Zde se však týkají modelování v UML a jsou přiřazeny k Prvkům modelu..

3.2.2. Relationship (Relace)

Předešlá část metamodelu UML z *Jádra* (*Core*) nazvaná *„backbone“ (páteř)* zavedla základní konstrukci pomocí abstraktní metatřídy *Prvek modelu* a současně také zavedla velmi důležitou konstrukci *Klasifikátoru*.

Relace (*Relationships*) jsou takové *Prvky modelu*, které umožňují v modelu vyjádřit vztahy mezi *Prvky modelu.*

Základní model *Relací* je ukázán na následujícím *obrázek 14: Základní konstrukce stromu generalizace u Relací*



obrázek 14: Základní konstrukce stromu generalizace u Relací

Jak je vidět, v UML existují tři základní *Relace* (*Relationships*):

- *Generalization* (*Generalizace*)
- *Association* (*Asociace*)
- *Flow* (*Tok*)

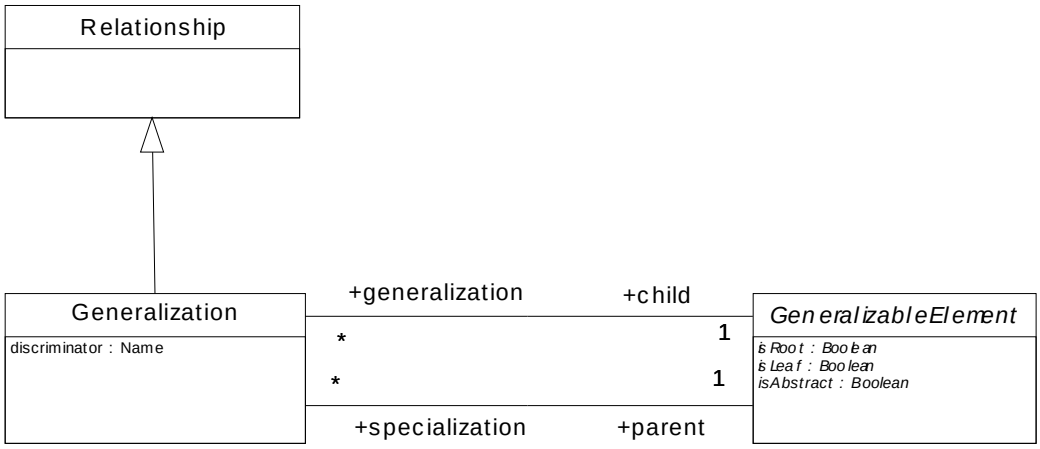
Jako zvláštní případ *Asociace* je zavedena metatřída *Asociativní třída* (*AssociationClass*), která je potomkem jak metatřídy *Asociace*, tak metatřídy *Třídy* (viz dědění z *Association* a *Class* na diagram obrázek 14: Základní konstrukce stromu generalizace u Relací).

3.2.2.1 Generalization (Generalizace)

Metatřída *Generalizace* je potomek metatřídy *Relace (Relationship)*. *Generalizace* vyjadřuje taxonomický vztah mezi více obecným prvkem a speciálnějším prvkem, přičemž speciálnější prvek je plně konzistentní s obecnějším prvkem.

Poznámka: Taxonomie je věda o systematickém třídění, používá se zejména v biologii.

Použitelná syntaxe tohoto vztahu je vyjádřena pomocí metamodelu následujícím způsobem:



obrázek 15 Relace Generalizace v UML 1.3

Metatřída *Generalizace* (*Generalization*) je ve dvojím vztahu k metatřídě *GeneralizableElement* (*Zobecnitelný prvek*). Připomeňme, že *Zobecnitelný prvek (GeneralizableElement)* je abstraktní třídou a že do vztahu vstupují až její konkrétní potomci, například *Klasifikátor* (*Classifier*) apod. Tím že se stávají potomky *Zobecnitelného prvku*, mohou vstoupit do *Relace* typu *Generalizace*.

Uvedené dva vztahy mezi metatřídou *Generalizace* a *Zobecnitelným prvkem* vypovídají, že *Generalizace* vidí „dva konce“. V nich figurují po jednom *Zobecnitelném prvku.* Jeden hraje roli *rodiče* (*parent*) a druhý hraje roli dítěte (*child*).

Z druhé strany existuje vztah N, tj. každý *Zobecnitelný prvek* může mít N specializací a N generalizací.

Poznámka: Pozor na malý chyták: Tento vztah mezi GeneraziableElementem a Generalization je vztahem mezi Zobecnitelným Prvkem a Relací typu Generalizace a nikoliv mezi dvěma prvky ve stromu generalizace. Jinak řečeno dva prvky ve stromu dědičnosti mají k sobě vztah až přes Relaci typu Generalizace, přičemž oba prvky mají vztah k této stejné instanci Generalizace (a tak se přes ni k sobě dostanou).

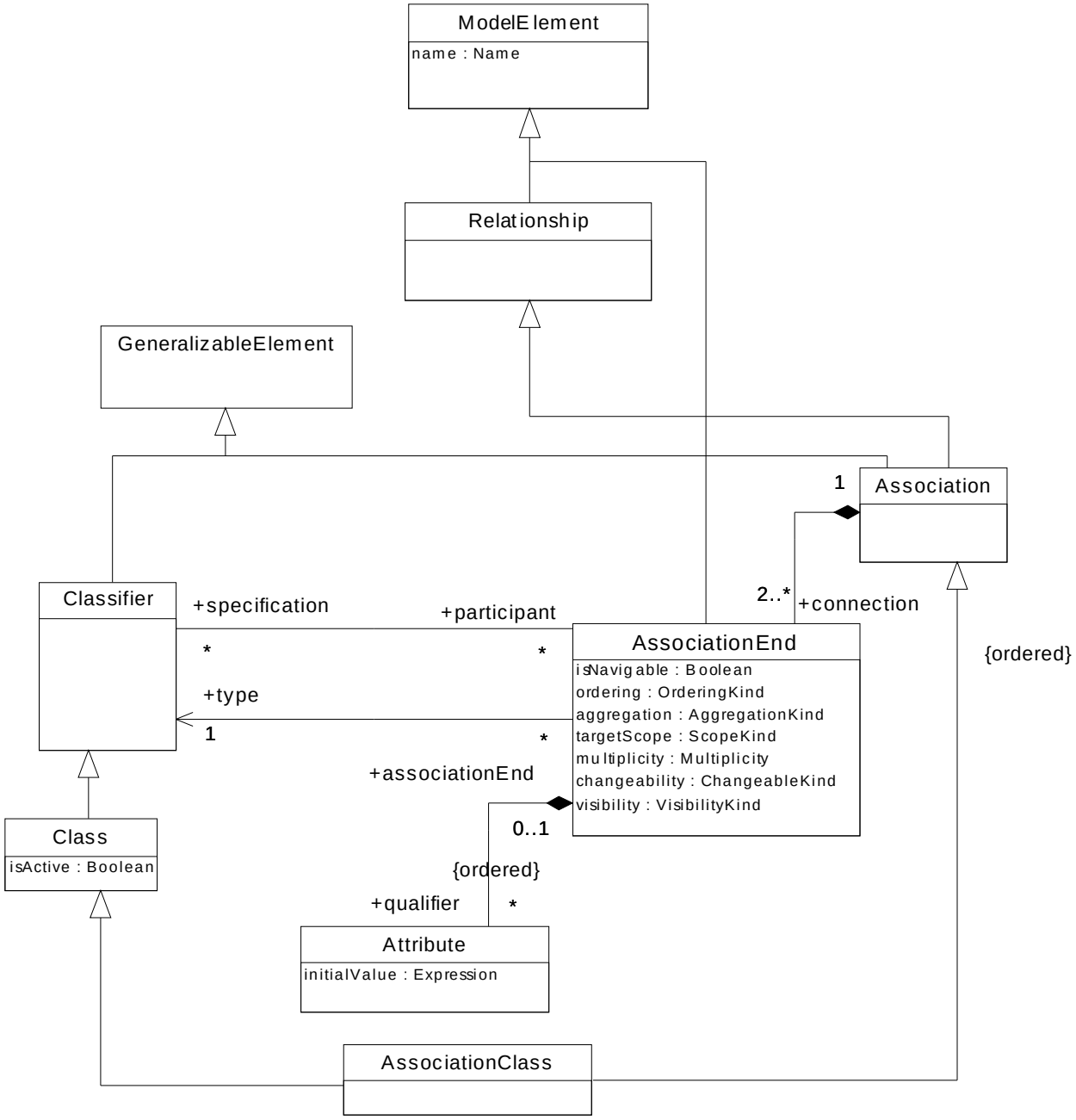
Atributy Generalizace

- discriminator (diskriminátor) – určuje, kam patří dané *spojení Generalizací*. Všechny instance *Generalizace*, které sdílejí stejného rodiče (parent), lze rozdělit do disjunktních množin. Určující k příslušnosti k dané množině je právě hodnota diskriminátoru.

Poznámka: pomocí diskriminátoru lze tedy rozdělit do disjunktních množin děti daného rodiče, tj. přímí potomci se takto rozčlení do skupin, aniž by se vytvářela další úroveň abstrakce mezi rodičem a dětmi.

3.2.2.2 Association (Asociace)

Zatímco vztah *Generalizace* vede v modelech k vyjádření stromů generalizace (tj. k taxonomickému vztahu), *Asociace* jako další podtyp *Relace* vyjadřuje vztahy mezi *Klasifikátory* při jejich vzájemném použití. Sémantika pro *Asociaci* je zavedena ve svazku metamodelu *Jádro*, část diagramu *Relace*, která je následující:



obrázek 16: Zavedení Asociace

Hlavním smyslem zavedení *Asociace* (*Association*) je možnost vyjádření vztahu mezi dvěma a více *Klasifikátory* (resp. instancemi *Klasifikátorů*).

Metatřída *Asociace* je také přímým potomkem (dítětem, *child*) *Relace*. Má vztah kompozice 2 až N ke *Konci Asociace*, tedy obsahuje minimálně dva a více instancí z metatřídy *Konec Asociace*

(*AssociationEnd*). Metatřída *Konec Asociace* je přímo potomkem metatřídy *Prvek modelu* (*ModelElement*). Tyto dva *Konce Asociace* jsou vůči *Asociaci* jako tzv. *připojení* (connection). Každý *Konec Asociace* „vidí“ svůj *Klasifikátor,* tedy vidí svůj typ.

Poznámka: Konec Asociace se mnohdy v OOP programu projevuje jako přímá objektová reference, nesmí se tedy zaměnit s atributem Klasifikátoru.

Konec Asociace odděluje přímou viditelnost mezi *Klasifikátorem* a *Asociací* (obaluje toto spojení) a tím mu dává možnost zavést další možné vlastnosti koncového bodu ve formě atributů a dalších vztahů.

Poznámka: Oproti předešlé konstrukci při zavedení Generalizace je zde, jak vidět, ještě jedna „obalující meziúroveň“ vyjádřená metatřídou Konec Asociace. Ve vztahu Generalizace byly „na koncích“ tohoto vztahu přímo prvky modelu, kterých se tento vztah týkal (jako potomci Zobecnitelného prvku) . Vztah mezi dvěma Klasifikátory přes Asociaci je tedy následující: První Klasifikátor - první Konec Asociace – Asociace - druhý Konec Asociace - druhý Klasifikátor.

Potomkem metatřídy *Asociace* je metatřída *Asociativní třída* (*AssociationClass*), která je současně také potomkem metatřídy *Třída* (Class). Jinak řečeno, *Asociativní třída* je *Asociací*, která je také *Třídou*.

3.2.2.3 AssociationEnd (Konec Asociace)

Konec Asociace je koncovým bodem *Asociace*, který spojuje *Asociaci* s *Klasifikátorem*.

Atributy Konce Asociace

- aggregation (agregace). Specifikuje, zda je *Konec Asociace* agregací. Pouze jeden *Konec Asociace* dané asociace může být agregací. Agregace je vyjádřením vztahu celek / část. Může nabýt těchto hodnot:
    - none (není agregace), *Konec asociace* nevstupuje do agregace, tj. neskládá.
    - aggregate (agreguje) znamená, že *Konec Asociace* vstupuje do agregace (skládání), tj. vztah *Konce Asociace* je „jsem součástí“, agreguji. Tato část může být obsažena jako část i jiného celku, tedy tato část může agregovat i jinde.
    - composite (kompozice) je „silnější“ agregace, kdy druhý *Konec Asociace* je také část celku, avšak jedná se o silné vlastnictví, kdy daný prvek nemůže být součástí nikde jinde, navíc bez tohoto zakomponovaného prvku postrádá celek smysl.

Poznámka: Z uvedeného vyplývá, že hodnota atributu aggregation Konce Asociace dává tuto vlastnost celé Asociaci: Asociace buď má nebo nemá právě jeden Konec Asociace ve stavu aggregate resp. composite. Jinak řečeno buď jsou všechny Konce Asociace dané Asociace ve stavu aggregation = none nebo nikoliv. Pokud nikoliv, potom existuje právě jeden Konec Asociace, který nemá aggregation = none (má aggregation ve stavu aggregate nebo composite).

Proto se mnohdy označuje celá Asociace, která obsahuje takový Konec Asociacie s aggreagation různým od none, jako Agregace, resp. Komposice podle hodnoty tohoto atributu. V literatuře jsem se setkal také s označením takové Asociace, která má všechny své Konce Asociace s atributem aggregation = none jako Běžná asociace. Z hlediska praktického modelování považuji toto označení Asociace za velmi přínosné. Vznikne tak následující výčet dělení Asociací:

- *Agregace,*
- *Kompozice,*
- *Běžná asociace,*

což je při modelování velmi praktické.

Rozdíl mezi těmito pojmy je tento:

- *Agregace je Asociací, která obsahuje právě jeden Konec Asociace s aggregation = aggregate (ostatní jsou none).*
- *Kompozice je Asociací, která obsahuje právě jeden Konec Asociace s aggregation = composite (ostatní jsou none).*
- *Běžná asociace je Asociací, která obsahuje všechny Konce Asociace s aggregation = none.*

Je třeba připomenout, že toto názvosloví je proti UML něco navíc (ale je v přesné souvislosti se syntaxí UML).

Existuje také nesprávné označení pro Běžnou Asociaci jako „Asociace“, což je z hlediska syntaxe UML 1.3 již velmi nepřesné a kolizní, protože Asociace v UML je obecnější pojem zahrnující také Agregaci a Kompozici.

- changeability (měnitelnost) specifikuje, zda propojení na *Klasifikátoru* na *Konci Asociace* může být měněno. Může nabývat hodnot:
  - changeable – není omezení
  - frozen – po vzniku zdrojového objektu (tj. druhý *Konec Asociace*) již v dalším průběhu jeho života nemůže být propojení změněno a ani přidáno (slovo „přidáno“ má význam pouze při násobnosti větší než jedna)
  - AddOnly – lze pouze přidávat další propojení (má význam pouze v násobnosti větší než jedna)

Poznámka: Zde se projevuje význam odstínění samotné instance Klasifikátoru od Asociace přes Konec Asociace. Atribut changeability je atributem Konce Asociace. Tedy měnitelnost (changeability) se týká možnosti změny „dosazením“ jiné instance Klasifikátoru do Konce Asociace anebo přidání takovéto instance, pokud je násobnost větší než jedna. Netýká se tedy měnitelnosti vlastností uvnitř samotné instance (uvnitř instance) Klasifikátoru.

V OOP situaci „frozen“ odpovídá dosazení určitého objektu z dané třídy do objektové reference bez možnosti dosadit do ní později jiný objekt, tedy daná reference na objekt je konstantní (například ukazatel na interface objektu je již neměnný, konstantní).

- *ordering* (uspořádání) uvádí, zda je množina propojení nějak uspořádaná. Má význam pouze u násobnosti větší než jedna. Nabývá hodnot
    - *ordered* (uspořádáno)
    - *unordered (neuspořádáno)*
    - jiná zvláštní možnost, podle situace
- isNavigable (navigován). Pokud má hodnotu ano, specifikuje, zda je možný přechod ve směru od zdrojové instance k cílové instanci. Specifikace každého směru je nezávislá na specifikaci v jiném směru.
- multiplicity (násobnost). Specifikuje počet koncových instancí, které mohou být napojeny k jedné instanci na zdrojovém konci.
- name (název) – poděděno z *Prvku modelu*.
- targetscope (rozsah cíle). Specifikuje, zda je cíl *Klasifikátorem* anebo instancí *Klasifikátoru* (má stejný význam, jako targetscope u *Atributu*).
    - instance: pro každé propojení je hodnotou instance. Implicitně se uvažuje tato hodnota (také bývá běžná).
    - classifier: pro každé propojení sám *Klasifikátor* je součástí propojení, tj. jedná se o vazbu na metainformaci. Málokdy se používá, a pokud, tak jenom v modelu, ale v implementaci modelu se použije jiná konstrukce umožněná daným prostředím.
- visibility (viditelnost) je obdobou již zavedené viditelnosti u atributu. Specifikuje viditelnost *Konce asociace* cílového *Klasifikátoru* v rámci druhého zdrojového *Klasifikátoru*:
    - public - Ostatní *Klasifikátory*, které vidí zdrojový *Klasifikátor*, vidí tento *Konec Asociace* a mohou jej používat.
    - protected - pouze potomci *Klasifikátora* vidí *Konec Asociace*.
    - private – pouze daný *Klasifikátor* vidí *Konec Asociace*.

3.2.2.4 Qualifier (Kvalifikátor)

Jak je vidět na diagramu obrázek 16: Zavedení Asociace, metatřída *Konec Asociace* (*AssociationEnd*) má vazbu kompozice 1 : N na metatřídu *Attribute* s názvem *kvalifikátor* (*qualifier*).

Metatřída *Attribute* je stejnou metatřídou, kterou jsme zavedli již v diagramu viz obrázek 11: Klasifikátor jako nositel struktury a chování.

Atribut tedy může vystupovat buď jako součást *Klasifikátoru* (atribut objektu), anebo jako součást *Konce asociace* (tj. jako *Kvalifikátor*), ale nikoliv jako součást obou současně, protože do vztahů vstupuje jako kompozice.

Kvalifikátor může tak zvaně kvalifikovat asociaci hodnotami atributů u *Konců Asociace*. Kvalifikování znamená přiřazení hodnot atributů ke každému propojení pomocí atributu *Kvalifikátoru*. Běžně se používá pro určení a identifikaci rozložení do množin všech propojení podle těchto hodnot daného atributu.

Zvláštním, ale nejčastějším případem použití kvalifikátoru, je zavedení tzv. indexu. Index není nic jiného, než *Qualifier* (*Kvalifikátor*) jako *Atribut Konce Asociace*. Pochází z *Datového typu* odpovídajícího typu čísla (např. Integer) a je přiřazen ke *Konci Asociace*. Pomocí *Kvalifikátoru – indexu* lze přistupovat k danému určitému *Konci Asociace* za pomoci hodnoty *Indexu*.

Poznámka: Stojí za povšimnutí, jak se vlastně chová index (například implementován v kolekci), protože na něm lze velmi pěkně ukázat vlastnosti Kvalifikátoru. Předně index není vlastností (atributem) samotné instance Klasifikátoru, která vystupuje v interakci, ale Index je přiřazen ke každému Konci Asociace. Přes index dostaneme odpovídající Konec Asociace.

Představme si, že by existoval nějaký pseudoindex takový, že by nebyl jednoznačný a k jedné hodnotě indexu by existovalo hned několik Konců Asociace. Je samozřejmé, že takovýto index ztrácí svou hlavní funkcionalitu a slouží ke „grupování" Konců Asociace. Obecný Kvalifikátor jako Atribut Konce Asociace pracuje právě takto.

Jiným častým případem použití *Kvalifikátoru* je zavedení tzv. klíče. Ke každému *Konci Asociace* je přiřazena unikátní hodnota atributu, *Kvalifikátor* zvaný klíč. Pomocí klíče lze opět zpětně kvalifikovat vazbu a získat tak pouze *Konec Asociace* s daným klíčem.

Poznámka: V modelování v některých případech vedou úvahy o rozložení vazby a zavedení informace na této vazbě nikoliv na použití kvalifikátoru, ale k zavedení asociativní třídy anebo u kompozice na vznik atributu u daného Klasifikátoru.

3.2.2.5 AssociationClass (Asociativní třída)

V metamodelu UML je *Asociativní třída* zavedena pomocí násobné dědičnosti: *Asociativní třída* je dítětem jak metatřídy *Třídy*, tak *Asociace* (viz obrázek 16: Zavedení Asociace. Znamená to, že se chová jako obě metatřídy „dohromady". Podle definice v UML je *Asociativní třída Asociací*, která je současně *Třídou*. Nejenom že tedy spojuje množinu *Klasifikátorů* do interakce tak, jak to činí *Asociace*, ale nese s sebou také strukturu a chování jako *Třída* (*Class*).

Rysy, které *Asociativní třída* nese, patří přímo *Asociaci* a nepatří k žádnému z *Klasifikátorů*, které se *Asociace* účastní.

Poznámka: Praktické použití v modelování viz blíže zmíněná kniha o objektovém modelování.

3.2.2.6 Flow (Tok)

Flow (Tok) je dalším podtypem *Relace*. Vyjadřuje vztah mezi *Prvky modelu* reprezentující buď jeden a tentýž objekt v různých stavech (tj. tentýž objekt, ale s jiným hodnotami atributů) anebo mezi dvěma objekty, z nichž jeden je věrnou kopií druhého objektu.

Flow (*Tok*) je spojen s modelováním aktivit, které vedou ke změnám stavů objektů.

3.2.3. Typy Klasifikátorů

Připomeňme, že typy *Klasifikátoru* jako jeho potomky zavádí diagram obr. obrázek 6: část modelu z Core (Jádro) - Classifiers (Klasifikátory) a jeho vztahy pro získání *Rysů* obrázek 11: Klasifikátor jako nositel struktury a chování. *Klasifikátory* jsou také potomci *Zobecnitelného prvku (Generaziable element)* a proto mohou vstupovat do *Relací Generalizace*.

V této kapitole popíšeme podtypy *Klasifikátorů*.

3.2.3.1 Class (Třída)

Třída (Class) je popisem množiny objektů, které sdílejí stejné atributy, operace, metody, vztahy a pravidla (*semantics*).

Třída může použít množinu *Interfaců* pro specifikaci všech operací, které mohou objekty poskytovat.

Atributy Třídy

- IsActive (aktivní). Pokud je pravda, potom objekt z této *Třídy* žije ve svém samostatném threadu. Pokud je nepravda, potom objekt žije v threadu jiného aktivního objektu v rámci volajícího objektu.

3.2.3.2 Interface (Interface)

Interface je pojmenovanou množinou operací, která charakterizuje chování nějakého prvku.
V metamodelu *Interface* obsahuje *Operace.*

Interface nesmí mít *Atributy*, nesmí vstupovat do *Asociací* a nesmí mít *Metody*. Toto omezení definuje uvedený vztah *Interface – Operace* jako zvláštní případ modelu na obrázek 11: Klasifikátor jako nositel struktury a chování .

Interface také může být *Zobecnitelným prvkem* a vstupovat do vztahu *Generalizace*.

3.2.3.3 Data type (Datový typ)

Datový typ je typ, jehož instance má hodnotu, ale nemá žádnou identitu, tedy jsou to pouze čisté hodnoty.

Poznámka: Tato definice se mi jeví jako velmi přesně vystihující rozdíl strukturálního a objektového přístupu. Ve strukturálním přístupu (kde neexistují objekty) mají na rozdíl od OOP data také svou identitu. Pro analytickou identitu jsou v OOP důležité objekty, které si mohou předávat data v rámci navzájem poskytnutých služeb (viz již zavedená metatřída Parametr)

V metamodelu *Datový typ* je zvláštní podtyp *Klasifikátoru*, který má *Operace* pouze jako čisté funkce, tj. funkce vracejí hodnotu, ale nemění původní hodnotu. Například „add“ operace aplikovaná na číslo jiným číslem vrací výsledek jako součet čísel, ale nemění původní čísla.

Speciální *Datové typy* se dále zavádějí ve svazku *Datové typy* (viz dále).

3.2.3.4 Component (Komponenta)

Dalším podtypem *Klasifikátoru* je *Komponenta* (*Component*).

Komponenta je fyzická, vyměnitelná (replaceable) část systému, která svazuje dohromady určitou implementaci a realizuje množinu interfaců. Komponenta tak reprezentuje vyměnitelný kousek implementace zahrnující softwarový kód (buď zdrojový, binární nebo spustitelný) anebo je skriptem resp. příkazovými soubory.

Poznámka: V UML se komponentou nerozumí pouze binární komponenta (jak bývá mnohdy zvykem v literatuře), ale také komponenta zdrojového kódu.

Sama *Komponenta* nejenom že svazuje dohromady implementaci svých specifických prvků (například *Klasifikátorů* apod.), ale sama může mít jako potomek *Klasifikátoru* své vlastní *Rysy*, jako jsou *Atributy* a *Operace*.

3.2.3.5 Node (Uzel)

Uzlem (Node) se v UML rozumí „run-time“ fyzický prvek reprezentující nějaký zdroj (computational resource) většinou mající alespoň paměť a často i schopnosti procesoru. V *Uzlech (Node)* se implementují konkrétní *Komponenty*.

Vztah mezi *Uzlem* a *Komponentou* je dán diagramem obrázek 6: část modelu z Core (Jádro) - Classifiers (Klasifikátory).

Komponenta je umístěna do *Uzlu* (agregace), kde se *Uzel* chápe jako rozmístění (deployment) a Komponenta je resident (synonymum pro „usídlenec“).

Komponenta realizuje (implementuje) různé *Prvky modelu* (*ModelElement*), například *Třídy*, *Interfacy*, *Atributy, Operace* atd., což vyjadřuje vztah agregace N *Prvků modelu* v *Komponentě*.

3.2.3.6 ElementResidence (Umístění prvku)

Všimněme si existence asociativní třídy *ElementResidence (Umístění prvku)*, která je zavedena pouze pro viditelnost (*visibility*) *Prvku modelu*. Tato viditelnost má stejný význam jako již zavedené viditelnosti v předešlých situacích, pouze zdrojový prvek v posuzování hranice viditelnosti je *Komponenta.*

Atributy ElementResidence (Umístění prvku)

- visibility nabývá hodnot:

- Public – *Prvek modelu* je viditelný a použitelný mimo *Komponentu*
- Protected - *Prvek modelu* je použitelný pouze potomky *Komponenty*
- Private – *Prvek modelu* je použitelný pouze *Komponentou*

3.3 Package Auxiliary elements (Svazek Prvky příslušenství)

Prvky příslušenství zavádí diagram metamodelu, viz obrázek 8: Diagram Prvky příslušenství - Auxiliary elements

Tento diagram vyjadřuje tři základní myšlenky:

- mechanismus *Šablony* (*Template*)
- mechanismus *Komentáře* (*Comments*)
- mechanismus *Prezentace elementu* (*PresentationElement*)

3.3.1.1 ModelElement (Prvek modelu) jako Template (Šablona)

Diagram na obrázek 8: Diagram Prvky příslušenství - Auxiliary elements zavádí mechanismus pro *Template* (*Šablonu*). Pod pojmem *Template (Šablona)* míníme obecnou šablonu s možností zavést parametry a tím převést šablonu z pozice vzoru do její konkrétní realizace. Jinak řečeno *Šablona* je obdobou vzoru pro zavedení konkrétních prvků dosazením určitých konkrétních hodnot do prvků parametrů.

V UML může každý *Prvek modelu* (*ModelElement*) vystupovat nejenom jako konkrétní prvek pro modelování, ale také jako *Template,* tedy jako *Šablona*. V tom případě *Prvek modelu* obsahuje jako svoji kompozici alespoň jeden *template parametr,* který je typu *Prvek modelu,* viz zmíněný diagram na obrázek 8: Diagram Prvky příslušenství - Auxiliary elements - kompozice z metatřídy *ModelElement* sama na sebe jedna ku N.

Prvek modelu je chápán jako *Template*, pokud má v této kompozici alespoň jeden *template parametr*.

Každý *template parametr* je zaveden do šablony jako tzv. „dummy“ *Prvek modelu* (prázdný prvek modelu), který dává smysl až po dosazení konkrétního prvku. Po naplnění dummy *Prvků modelu* přechází daný *Prvek modelu* v roli *Šablony* do role běžného *Prvku modelu*. Dosazovaný *Prvek modelu* za *template parametr* musí být ze stejné metatřídy anebo z metatřídy potomka dummy *Prvku modelu*, který reprezentuje *Parametr šablony*.

Navíc je zavedena asociativní metatřída *TemplateParameter* připojená na kompozici parametrů *Prvku modelu* – template obrázek 8: Diagram Prvky příslušenství - Auxiliary elements. Její význam spočívá v implicitní hodnotě: Pokud není specifikována konkrétní hodnota parametru a je jí třeba, musí být poukázáno na implicitní parametr.

Je třeba zdůraznit, že *Prvek modelu (ModelElement)* může vystupovat v UML ve dvou rolích:

- *Prvek modelu* (*ModelElement*) je *Šablonou* (*Template*), což platí tehdy, má-li *Prvek modelu* alespoň jeden parametr šablony (*template parameter*).
- *Prvek modelu* je běžným *Prvkem modelu*, nemá parametr šablony.

3.3.1.2 Binding (Vazba)

Hlavním smyslem prvku *Binding (Vazba)* je zavést *Relaci* mezi *Prvkem Šablonou* a *Prvkem modelu* vygenerovaným z této šablony. Metatřída *Binding* (*Vazba*) je proto potomkem metatřídy *Dependency* (*Závislost*), viz obrázek 7: část modelu UML zavádějící vztahy typu Dependency (Závislost). Tím jsou dost hutně vyjádřeny dvě důležité skutečnosti:

- *Vazba* (Binding) je chápána jako *Závislost (Dependency)* instance *Šablony* na *Šabloně.*
- *Vazba* vidí instance *Prvků modelu* ve vztahu *Závislosti*, v horní úrovni generalizace označené jako client (klient) a supplier (dodavatel), viz obrázek 7: část modelu UML zavádějící vztahy typu Dependency (Závislost) . Pro *Vazbu* však mohou být jenom v násobnosti jedna (nikoliv N, jak je obecně uvedeno na diagramu), tj. jedna *Šablona* versus N instancí *Šablony*.

Vazba (*Binding*) obsahuje seznam *argumentů* (*argument*), které jsou v souladu s *template parametry* (viz obrázek 8: Diagram Prvky příslušenství - Auxiliary elements). Jedna *Vazba* (*Binding*) jako potomek *Relace* vidí jednu instanci zdroje *Šablony* a jednu instanci této *Šablony*. Znamená to, že pokud jedna *Šablona* realizuje několik svých instancí, tak pro každou instanci realizace *Šablony* vznikne nová *Vazba*.

3.3.1.3 Comment (Komentář)

Komentář (Comment) je poznámkou, která se připojuje k jednomu nebo více *Prvkům modelu* (viz obrázek 8: Diagram Prvky příslušenství - Auxiliary elements). *Komentář* nepodléhá žádným sémantickým pravidlům a slouží jako dodatečná informace pro tvůrce a čtenáře modelu.

Poznámka: Je zajímavé, že se nevyžaduje poznámka jako text (tj. není specifikován atribut Komentáře jako text). Je to asi proto, aby Komentářem mohlo být opravdu cokoliv, třeba i grafika apod.

3.3.1.4 PresentationElement (Prezentační prvek)

Prezentační prvek (PresentationElement) je textovým nebo grafickým znázorněním jednoho anebo více *Prvků modelu*. Z toho plyne i vztah vůči *Prvkům modelu* jako (běžná) asociace M ku N viz obrázek 8: Diagram Prvky příslušenství - Auxiliary elements. Z jedné strany se jedná o roli subjektu (subject), tedy co se má prezentovat, z druhé strany se jedná o prezentaci (prezentation), tedy jak bude prezentováno.

Metatřída *Prezentační prvek (PresentationElement)* je chápána jako základní třída pro tvorbu všech prezentačních prvků, tj. jedná se o nejvyšší abstraktní metatřídu pro všechny sub-metatřídy se stejným účelem, tj. pro prezentaci.

Poznámka: Vzniká tak klasická struktura paralelních stromů generalizace (tzv. parallel hierarchy) s potomky Prvků modelu a s potomky Prezentačních prvků.

Posledním diagramem svazku *Jádra (Core)* je diagram pro zavedené *Závislosti (Dependencies)*, viz obrázek 7: část modelu UML zavádějící vztahy typu Dependency (Závislost).

3.3.1.5 Dependency (Závislost)

Aby se zavedla možnost modelovat závislost mezi prvky modelu, je na nejvyšší úrovni diagramu zavedena metatřída *Dependency (Závislost)*, která je potomkem *Relace (Relationship)*. Tato metatřída má dvě asociace na *Prvky modelu*. Jedna *Asociace* je z jednoho *Konce Asociace* zavedena jako client (klient), druhá má *Konec Asociace* zaveden jako supplier (dodavatel). Směr se chápe tak, že klient je závislý na dodavateli.

Poznámka: Obecně jsou implicitně závislé Prvky modelu, které vstupují i do jiných vztahů, například do kompozice. Tyto vztahy není třeba pomocí Dependency vyjadřovat. Vztah Dependency se zavádí tam, kde není možné jinak závislost z modelu odvodit, resp. tam, kde zvyšují přehlednost modelu.

Obecně je povoleno mít v jednom *Dependency* (*Závislosti*) N klientů (client) a N dodavatelů (supplier), avšak v některých případech může dojít k omezení tohoto počtu z povahy věci (například vzpomeňme na *Vazbu Binding*, kde může být pouze jeden klient a jeden supplier v jedné dané *Vazbě*).

Tyto vlastnosti *Dependency* pro vztah mezi dodavateli a klienty podědí její potomci.

3.3.1.6 Binding (Vazba)

Metatřídu *Vazba (Binding)*, která vyjadřuje závislost mezi *Šablonou* a instancí *Šablony*, jsme již popsali v předešlé kapitole, viz *Binding (Vazba)*.

3.3.1.7 Abstraction (Abstrakce)

Abstrakce je *Závislostí* mezi dvěma *Prvky modelu* vyjadřujících tentýž pojem, ale na různých úrovních abstrakce. Pod úrovní abstrakce máme na mysli zejména úrovně abstrakce odpovídající fázím vývoje.

Poznámka: Nezaměňujme Abstrakci se vztahem Generalizace.

Různé úrovně abstrakce dávají do vzájemného vztahu různé *Prvky modelu*, které by jinak nemusely mít žádný jiný vztah. Klasickým příkladem je zavedení *Klasifikátoru* ve fázi analýzy a následně dalšího *Klasifikátoru*, který navazuje na tento pojem ve fázi designu. Oba dva *Klasifikátory* popisují „tutéž věc“, ale na různých úrovních abstrakce, tj. mají spolu souvislost *Abstrakce*.

Poznámka: Návaznost fází modelování jako tvorba úrovní abstrakce často tento vztah používají.

Atributy Abstrakce

- mapping (mapování) je výraz popisující způsob mapování více abstraktního prvku do více implementačního prvku, tento atribut není povinný.

3.3.1.8 Usage (Užití)

Užití (Usage) je *Závislostí*, kdy jeden *Prvek modelu* potřebuje ke své implementaci druhý *Prvek modelu* a používá jej. Klient si vyžaduje existenci dodavatele. Pokud jej nemá k dispozici, je narušena konzistence modelu.

3.3.1.9 Permission (Povolení)

Permission (*Povolení*) je zvláštním typem *Závislosti* mezi klientem (client) a dodavatelem (supplier), kde dodavatelem (supplier) musí být *Namespace* (*Pojmenovaný prostor*) resp. jeho dědic *Package* (*Svazek*). V tom případě se klientovi zpřístupňuje obsah dodavatele podle zavedeného stereotypu (viz dále).

3.4 Package Extension Mechanisms (svazek Mechanismy extenze)

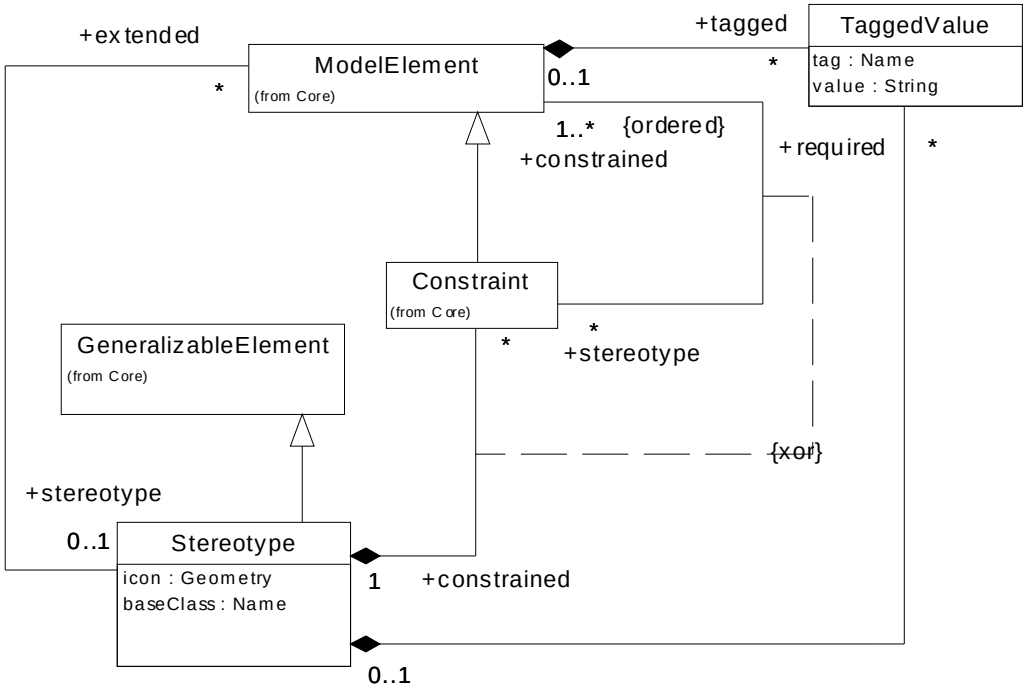
Svazek *Mechanismů extenze* patří stejně jako svazek *Jádro (Core)* do svazku *Základ* (*Foundation*). Tento svazek zavádí možnosti extenze v modelech UML, tj. zavádí možnost přidávat uživatelsky definované informace do modelu, aniž by byla narušena sémantika UML. Je třeba podotknout, že tvůrci UML se opravdu snažili zavést takový modelovací jazyk pro tvorbu softwaru, který by byl konzistentní a dostačující použitím libovolného potřebného prvku modelu. Přesto se zavádí mechanismus extenze tak, aby i uživatel měl možnost zavést některé uživatelsky definované vlastnosti.

K tomu slouží zavedení modelovacích prvků vyjádřené jako metatřídy v metamodelu:

- *Tagged Value (Tagovaná hodnota)*
- *Constraint (Omezení)*
- *Stereotype (Stereotyp)*

Jako další takový prvek extenze, který sice v této kapitole (a ve svazku *Mechanismy Extenz*e) není uveden, bychom mohli jmenovat *Komentář (Comment)*. Ten lze také „kdekoliv“ umístit. *Komentář* však tvůrci UML neumístili do svazku *Mechanismy extenze*, ale rovnou do svazku *Jádro (Core)*.

Následující obrázek 17: Mechanismy extenze v UML 1.3 ukazuje vztah těchto nově zavedených metatříd v metamodelu UML:



obrázek 17: Mechanismy extenze v UML 1.3

3.4.1.1 Tagged Value (Tagovaná hodnota)

Význam *Tagované hodnoty* je stejný, jako je zaveden například v XML. Jedná se vlastnost vyjádřenou jednoduše dvojicí (význam, hodnota), jak už napovídají atributy této metatřídy. Jak vyplývá z diagramu na předešlém obrázek 17: Mechanismy extenze v UML 1.3, *Tagovaná hodnota* vystupuje v metamodelu dvakrát, buď je přiřazena jako kompozice k *Prvku modelu* anebo ke *Stereotypu* (nikoliv současně) a to v násobnosti N.

Přiřazení *Tagované hodnoty* k *Prvku modelu*, tj. přiřazení do kompozitního seznamu *Tagovaných hodnot* daného *Prvku modelu* je právě vyjádřením možnosti přidat k libovolnému *Prvku modelu* *Tagovanou hodnotu* jako svou uživatelsky definovanou vlastnost. Kromě použití stringu v hodnotě není v zavedení *Tagované hodnoty* žádné další omezení.

Druhá část metamodelu vztahu *Tagované hodnoty*, která vyjadřuje kompozici vůči *Stereotypu*, se může jevit v metamodelu jako zbytečná, protože *Stereotyp* je současně také chápán jako *Prvek modelu*. Vysvětlení kompozice „*Stereotyp* obsahuje *Tagované hodnoty*“ bude vysvětlen u metatřídy *Stereotyp*.

Atributy Tagované hodnoty

- tag (nepřekládáme) název vlastnosti přidané k *Prvku Modelu*

- value (hodnota) hodnota vlastnosti přidané k *Prvku Modelu*, musí být string (string je zaveden ve svazku *Datové typy*)

3.4.1.2 Stereotype (Stereotyp)

Stereotyp zavádí mechanismus klasifikace (značkování) *Prvků modelu* uživatelskými kategoriemi neboli *Stereotypy*. Jeden *Stereotyp* je jedna tvůrcem modelu zavedená kategorie, do které lze zařadit *Prvek modelu*. Existují standardní *Stereotypy* předdefinované v UML (bude předmětem v další publikaci o UML).

Stereotyp může definovat novou kategorii *Prvků modelu*, které sice mají stejnou strukturu a chování jako nekategorizované *Prvky modelu*, ale mají nějak jiný význam. Někdy se hovoří o použití „virtuální metatřídy".

Všimněme si, že vztah mezi *Prvkem modelu* a *Stereotypem* (*ModelElement* a *Stereotype*) je (běžnou) asociací a navíc vztah ze strany od *Prvku modelu* ke *Stereotypu* má násobnost 0..1 . Znamená to, že pokud má daná instance *Prvku modelu* přiřazen nějaký *Stereotyp*, je tento *Stereotyp* maximálně jeden.

Další skutečnost, kterou je třeba vysvětlit, je existence kompozice od *Stereotypu* k *Tagované hodnotě* s násobností N, přestože podobná kompozice již existuje na vyšší úrovni generalizace mezi *Prvkem modelu* a *Tagovanou hodnotou*.

V UML je zaveden mechanismus, že každý *Prvek modelu*, který má přiřazen daný *Stereotyp*, přebírá také *Tagované hodnoty* přiřazené k tomuto *Stereotypu*. Existují tedy dvě možnosti, jak přiřadit k *Prvku modelu Tagovanou hodnotu*:

- Přiřadíme ji rovnou k *Prvku modelu.*
- Přiřadíme ji ke *Stereotypu*, všechny *Prvky modelu* tohoto *Stereotypu* mají tyto *Tagované hodnoty.*

Protože se jedná o dva velmi odlišné mechanismy přiřazení, existují na diagramu dvě kompozice *Tagovaných hodnot*, jedna je vůči *Prvku modelu* a druhá je vůči *Stereotypu*.

Atributy Stereotypu

- baseClass (nepřekládáme). Specifikuje název UML modelujícího elementu, vůči kterému *Stereotyp* působí, jako například *Class, Component* apod. Nedoporučuje se volit za obsah baseClass prvky samotného (vašeho) modelu, ale raději se má použít prvek metamodelu UML.
- icon (ikona) grafické vyjádření prvku, který patří do daného *Stereotypu*. Extenzivní mechanismus umožňuje při tvorbě nového *Stereotypu* změnit i grafiku prvku.

Poznámka: Například u stereotypu s názvem << ActiveX DLL >> bude mít baseClass hodnotu Component a svou vlastní ikonku jako „žlutá do sebe zapadající ozubená kolečka".

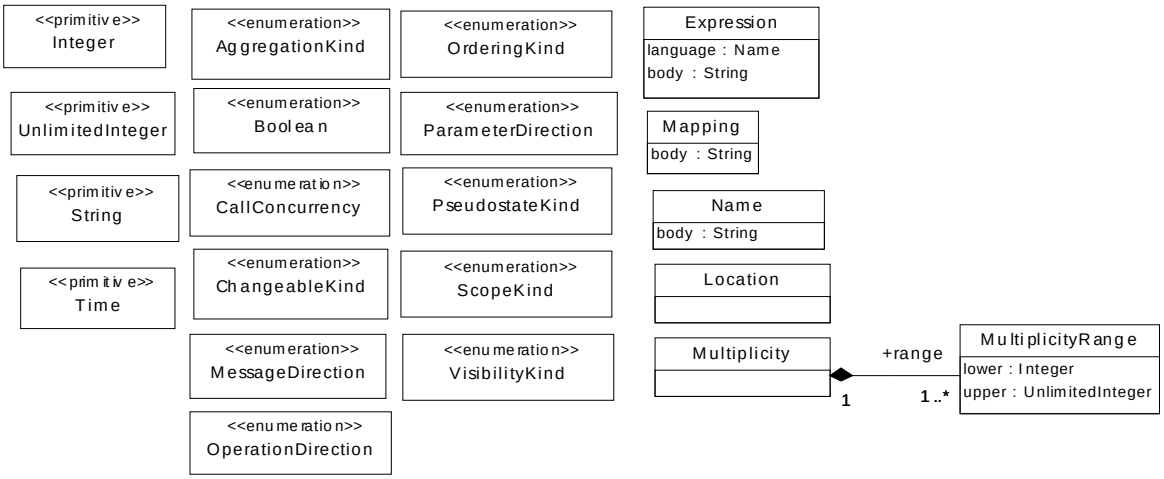
3.4.1.3 Constraint (Omezení)

Omezení již bylo vysvětlena v kapitole Constraint (Omezení). Všimněme si určitého rozšíření v dalším modelu obrázek 17: Mechanismy extenze v UML 1.3. *Constraint* (*Omezení*) může být přiřazen nejenom k *Prvku modelu*, ale i ke *Stereotypu*. Význam je úplně stejný, jako u *Tagované hodnoty*. Přiřadit *Omezení* ke *Stereotypu* znamená přiřadit *Omezení* ke všem *Prvkům modelu*, ke kterým je přiřazen tento *Stereotyp*.

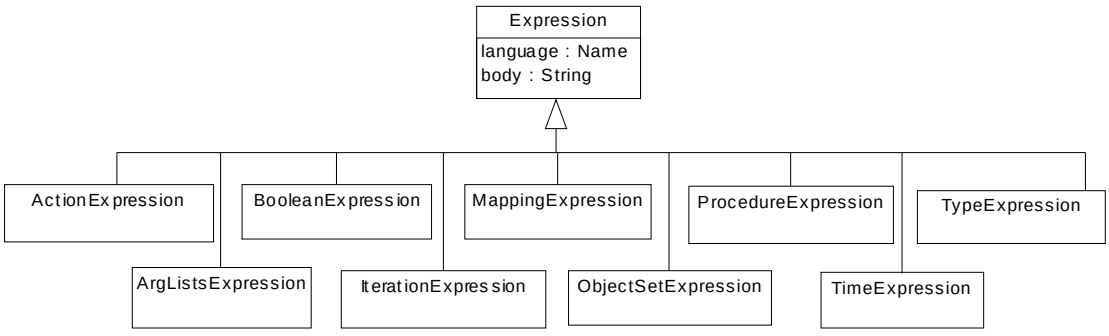
3.5 Package Data Types (svazek Datové typy)

Všechny datové typy jsou v UML umístěny do jednoho svazku *Datové typy*, který je součástí svazku *Základ* (*package Foundation*).

Následující dva obrázky zavádějí abstraktní syntaxi *Datových typů*:



obrázek 18: Zavedení datových typů (první část)



obrázek 19: Zavedení datových typů (druhá část)

Jedním z důvodů, proč jsou zavedeny *Datové typy*, je pochopitelně možnost jejich použití v rámci modelování softwarových produktů. Podle mého soudu mnohem důležitějším důvodem pro zavedení datových typů v té podobě, jak jsou uvedeny na předešlých dvou obrázcích, je získání konzistence metamodelu UML. Při zkoumání libovolného diagramu metamodelu UML se snadno zjistí, že typy u metaatributů daných metatříd se zavádějí jako *Datové typy* ze svazku *Datových typů*. Tím jsou definovány i typy, které používá sám metamodel UML.

Z uvedeného důvodu nemá příliš velký smysl vysvětlovat všechny *Datové typy* UML dopodrobna. Zájemce mohu odkázat na soubor specifikace UML na stránkách organizace OMG www.omg.org .

V souvislosti s vysvětlením syntaxe UML u *Datových typů* stojí za zmínku pouze skutečnost, že existují stereotypy zavedené u datových typů, kterými jsou primitive (primitivní typ) a enumeration (výčtový typ).

Z diagramu na obrázcích obrázek 18: Zavedení datových typů (první část) a obrázek 19: Zavedení datových typů (druhá část) lze vcelku dobře vyčíst všechny *Datové typy*. Z hlediska vzájemných vztahů mezi metatřídami je zajímavý pouze typ *Multiplicity* (násobnost), který může obsahovat několik *MultiplicityRange* (*Rozsahů násobnosti*), tj. několik intervalů, a druhým zajímavým modelem je strom *Generalizace* pro *Expressions* (*Výrazy*). Strom generalizace ukazuje různé podtypy *Výrazů*.

3.6 Package Behavioral Elements (svazek Prvky chování)

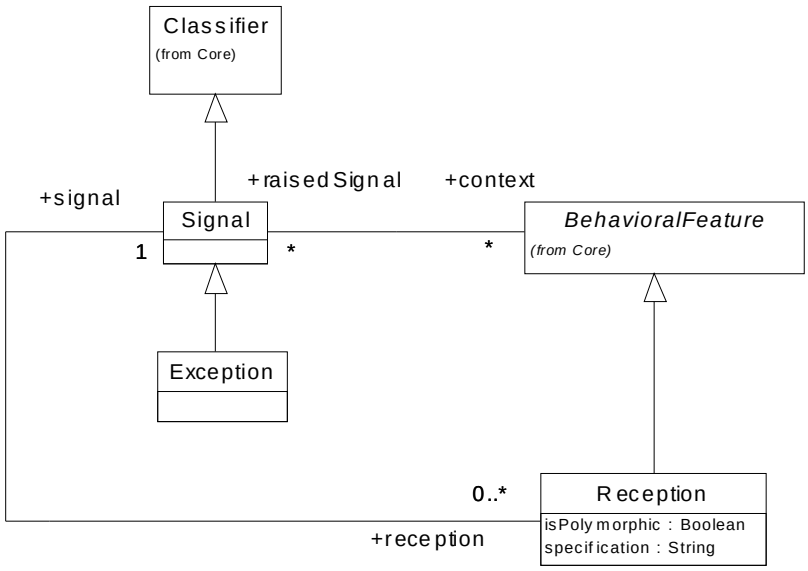
Prvky ze svazku *Prvky chování* jsou určeny k modelování dynamiky systémů, chování prvků apod. Vnitřní struktura svazků je dána diagramem viz obrázek 3 Skladba svazku Behavioral Elements. Je vidět, že package *Common Behavior* (svazek *Společné chování*) je centrálním svazkem, na němž jsou ostatní svazky závislé.

3.7 Package Common Behavior

Svazek *Společné chování* je základní svazek ze svazku *Prvky chování*. Specifikuje základní vlastnosti, které jsou po prvcích popisujících chování požadovány.

Prvky v tomto svazku jsou zavedeny pomocí několika částí metamodelů, které si nyní uvedeme.

Část metamodelu zavádějící metatřídy *Signál (Signal)* a *Příjem (Reception)* je uvedena na následujícím obrázku:



obrázek 20 Zavedení Signálu a Příjmu

3.7.1.1 Signal (Signál)

Signál (Signal) je specifikací asynchronního podnětu, impulsu (angl. *stimulus*), který komunikuje mezi instancemi. Přijímací instance zpracovává *Signál* podle zavedeného stavového stroje. *Signál* je vždy asynchronní a je definován nezávisle na definici instance, která jej zpracovává.

Metatřída *Signál* je potomkem metatřídy *Klasifikátor (Classifier)* a tedy i metatřídy *Zobecnitelného prvku (Generalizable Element)*. Dědění ze strany *Klasifikátoru* umožňuje prvku *Signál*, aby mohl mít *Atributy (Attribute)*, které v tomto případě reprezentují parametry *Signálu*.

Signál je v asociačním vztahu s *Prvkem chování (BehavioralElement)*, který tento *Signál* vyvolává (viz obrázek 20 Zavedení Signálu a Příjmu).

3.7.1.2 Reception (Příjem)

Reception (Příjem) zavádí určitý *Rys chování* specifikující, jak je daný *Klasifikátor (Classifier)* připraven na příjem *Signálu*. Všimněme si, že ze strany *Příjmu (Reception)* je vidět pouze jeden *Signál*, naopak jeden *Signál* může být přijat několika *Příjmy (Reception).*

Reception je potomkem *BehavioralFeature (Rys chování)* a proto vstupuje v modelování do vztahu kompozice vzhledem ke *Klasifikátoru* podobně jako například *Operace* nebo *Metody*.

Atributy Reception

- isPolymorphic – (boolean) pokud je *Příjem Signálu* polymorfní, znamená to, že se chování může měnit v závislosti na hodnotě stavu anebo se mění u podtříd.
- specification –specifikuje přesně reakci na *Signál*, je typu String.
- isAbstract, isLeaf, isRoot (Zobecnitelný prvek) mají stejný význam jako u každého *Zobecnitelného prvku*

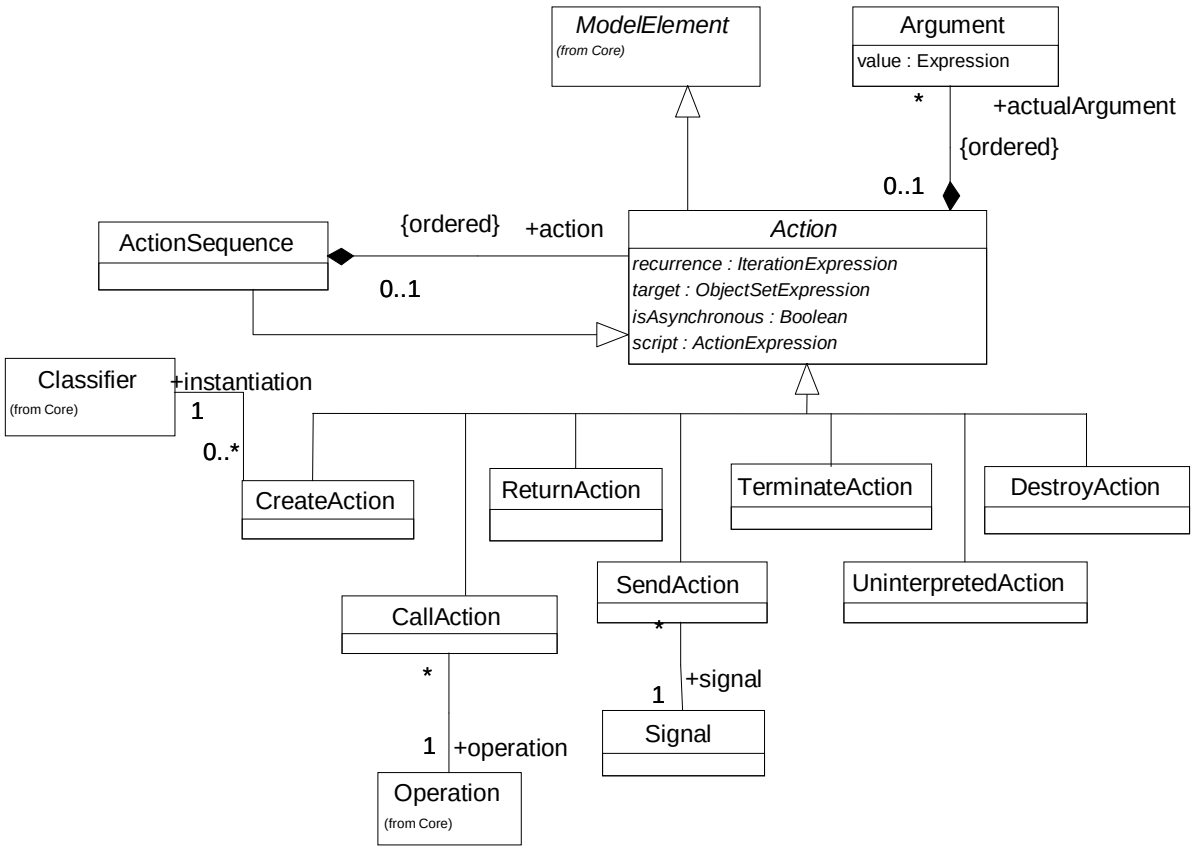
3.7.1.3 Exception (Výjimka)

Exception (Výjimka) je *Signál*, který je vyvolán *Prvkem chování* zejména v případě chyby v systému.

Výjimka je zavedena jako přímý potomek *Signálu*.

3.7.1.4 Common Behavioral – Actions (Společné chování – Akce)

Další část modelu, *Actions (Akce)* zavádí základní konstrukci prvků *Akce (Actions)*:



obrázek 21 část modelu zavádějící Akce

Základním prvkem v této části metamodelu je metatřída *Action* (*Akce*), která je přímým potomkem *Prvku modelu* (*ModelElement*).

3.7.1.5 Action (Akce)

Action (Akce) je přímým potomkem *Prvku modelu (ModelElement)* a je abstraktní metatřídou.

Specifikuje nějaký vykonatelný příkaz (executeble statement). Formuluje abstrakci nějaké počítačové vykonávané sekvence kódu, která nějak mění stav *Prvků modelu* v modelu. Většinou abstrahuje zaslání zpráv mezi objekty resp. změny atributů v objektech atd.

Prvek modelu *Action (Akce)* se používá se v situacích , kdy modelujeme systém a vyjadřujeme určitou *Akci* systému jako příkaz. I když se samozřejmě pohybujeme v objektovém prostředí, lze při modelování v určitém pohledu chápat činnost systému jako určitou *Akci (Action)*.

Rozdělení dalších podtypů *Akcí (Action)* ukazuje jejich možné použití v modelech. *Akce* může mít *Argumenty* a může vytvářet *ActionSequence (Sekvence Akcí)*, která je chápána také jako *Akce*. Také v objektovém prostředí lze probíhající činnosti systému chápat jako posloupnosti *Akcí*, tedy posloupnosti příkazů. Přitom (jak ukazuje výčet typů *Akcí*) jsou možnosti „co lze všechno vlastně chápat jako *Akce*“ mnohem širší, než ve strukturálním programování.

Atributy Action

Je třeba podotknout, že následující atributy mohou díky dědičnosti používat také podtypy *Akcí* vyjmenované v předešlém diagramu.

- isAsynchronous – pokud je pravda, potom *Akce* probíhá asynchronně.
- reccurence – (opětovný výskyt), *Výraz (Expression)* udávající, kolikrát se daná *Akce* opakuje.
- script - *Výraz Akce (ActionExpression)* popisující efekt *Akce* (co se odehraje v rámci *Akce*).
- target – *ObjectSetExpression (Výraz pro dosazení objektu)* udávající cíl *Akce*.

3.7.1.6 ActionSequence (Sekvence Akcí)

Prvek *Sekvence Akcí* slouží v UML k popisu chování systému uvnitř nějakého *Stavu (State)* anebo v rámci *Přechodu (Transition)*, který tuto *Sekvenci Akcí* vlastní.

Prvek *ActionSequence (Sekvence Akcí)* je v diagramu na obrázek 21 část modelu zavádějící Akce zaveden velmi zajímavým způsobem: *Sekvence Akcí* je přímým dědicem *Akce*. Znamená to, že *Sekvence Akcí* je chápána také jako *Akce*. Přitom *Sekvence Akcí* obsahuje (jako kompozice) N uspořádaných *Akcí* („*ordered*“). Tím je vyjádřena jednoduchá možnost „skládání“ *Akcí* do větších celků posloupností příkazů. Je třeba zdůraznit jednu skutečnost: Protože *Sekvence Akcí* je přímým dědicem *Akce* a může přitom obsahovat *Akce*, tak *Sekvence Akcí* může obsahovat také *Sekvence Akcí*.

Poznámka: Na tomto příkladu z metamodelu UML lze také pěkně ukázat na význam abstraktní třídy. Metatřída Akce není abstraktní třídou, je metatřídou konkrétní a vznikají z ní instance. Jsou to ty instance, které již nejsou Sekvencemi Akcí. Diametrálně odlišná situace by nastala, pokud by Akce (Action) byla abstraktní metatřídou.

3.7.1.7 Argument (Argument)

Argument je výraz, který popisuje, jakým způsobem jsou definovány aktuální hodnoty vstupující do zpracovaných požadavků. Je agregován do *Akce (Action)* (viz obrázek 21 část modelu zavádějící Akce.

S pojmem „argumenty“ jsme se již setkali a nesmíme je zaměňovat. Zde je *Argument* metatřídou. U *Prvku modelu* jsme se setkali s možností použít jej nikoliv přímo jako prvek vystupující v modelu, ale také jako *Šablonu (Template)*, do které se dosazují argumenty a teprve tím vznikne konkrétní prvek modelu (viz obrázek 8: Diagram Prvky příslušenství - Auxiliary elements a další kapitoly vysvětlující vztah mezi *Vazbou Binding* a *Prvkem modelu – ModelElement*. U *Akcí* jsou *Argumenty* přímo metatřídou a hrají roli vůči *Akci* jako *actualArgument*).

Atributy Argumentu

- value (hodnota) - je *Výraz (Expression)*, který popisuje, jak je argument determinován v okamžiku výkonu *Akce*.

3.7.1.8 CreateAction (Akce tvorby)

Akce tvorby (CreateAction), je přímým dědicem *Akce*, tedy je *Akcí (Action)*. *Akce tvorby* ve svém důsledku vede ke vzniku nějaké instance z nějakého *Klasifikátoru*. Všimněme si, že má (běžnou) asociaci ke *Klasifikátoru (Classifier)*, která ukazuje, z jakého *Klasifikátoru* vznikne instance. Ze strany *Akce tvorby* je vztah jeden *Klasifikátor*, naopak *Klasifikátor* má N možných *Akcí tvorby*.

3.7.1.9 DestroyAction (Akce zrušení)

Akce zrušení (DestroyAction) je *Akcí* vedoucí ke zrušení instance. Neuvádí se *Klasifikátor* jako v případě *Akce tvorby*, ale přímo se specifikuje, která instance se má zrušit, k čemuž slouží údaj uvedený v atributu target nadřízené metatřídy *Akce (Action)*.

3.7.1.10 CallAction (Akce volání)

Akce volání je *Akcí*, jejímž výsledkem je vyvolání nějaké *Operace* instance *Klasifikátoru*. *Akce volání* má asociaci na jednu *Operaci*. *Operace* může mít několik svých *Akcí* volání (viz diagram). *Akce volání* může být synchronní nebo asynchronní (viz nadřízená metatřída *Akce*), což udává, zda bude daná instance volána synchronně anebo asynchronně.

3.7.1.11 TerminateAction (Akce ukončení)

Akce ukončení (TerminateAction) vede k sebezničení daného objektu. Jako cíl (target) je sama instance, tedy u *Akce ukončení* se nemusí hodnota atributu target uvádět, protože je jí sama instance vykonávající *Akci*.

3.7.1.12 ReturnAction (Akce návratová)

Akce návratová je *Akcí*, která vrací hodnoty zpátky volajícímu objektu. Argumenty jsou u této *Akce* návratovými hodnotami.

Poznámka: Z logiky věci vyplývá, že této Akci návratové musí předcházet činnost volajícího objektu, kterému se tyto hodnoty navracejí.

3.7.1.13 SendAction (Akce zaslání)

Akce zaslání je Akcí, jejímž výsledkem je (asynchronní) vyslání *Signálu*. *Signál* může být zaslán excplicitně vyjmenovaným cílům (viz atribut target) anebo se jako cíl specifikuje pouze obecný mechanismus systému (zpracování signálu apod.), někdy se cíl neuvádí a má se za to, že signál odchytí ten, kdo jej potřebuje odchytit.

3.7.1.14 UninterpretedAction (Akce nespecifikovaná)

Akce nespecifikovaná reprezentuje *Akci*, která není předefinovaná v UML.

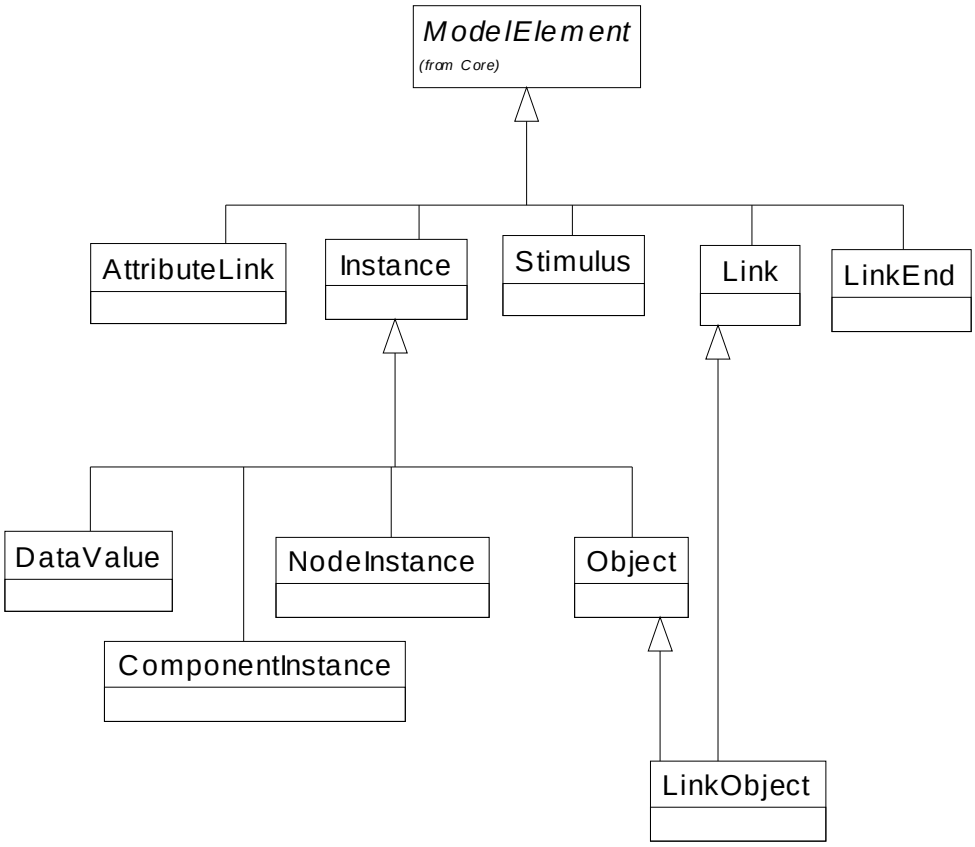
Poznámka: Pomocí mechanismu stereotypu bychom tedy mohli zavést svůj vlastí virtuální podtyp Akce.

3.7.2. Common Behavioral – Instances (Společné chování - Instance)

Dalšími prvky ve svazku *Společné chování* (*Common Behavioral*) jsou *Prvky modelu* popisující tzv. *Instance*. Pro získání přehlednosti jsou rozsáhlé diagramy popisující tuto část modelu rozloženy do několika menších diagramů v posloupnosti jejich důležitosti.

Poznámka: Je zajímavé, že metamodel zavádějící instance je umístěn do Prvků chování. Má se tím na mysli to, že aplikace „žije“ svými instancemi, které se nějak chovají.

Základní strom generalizace ukazuje následující obrázek.



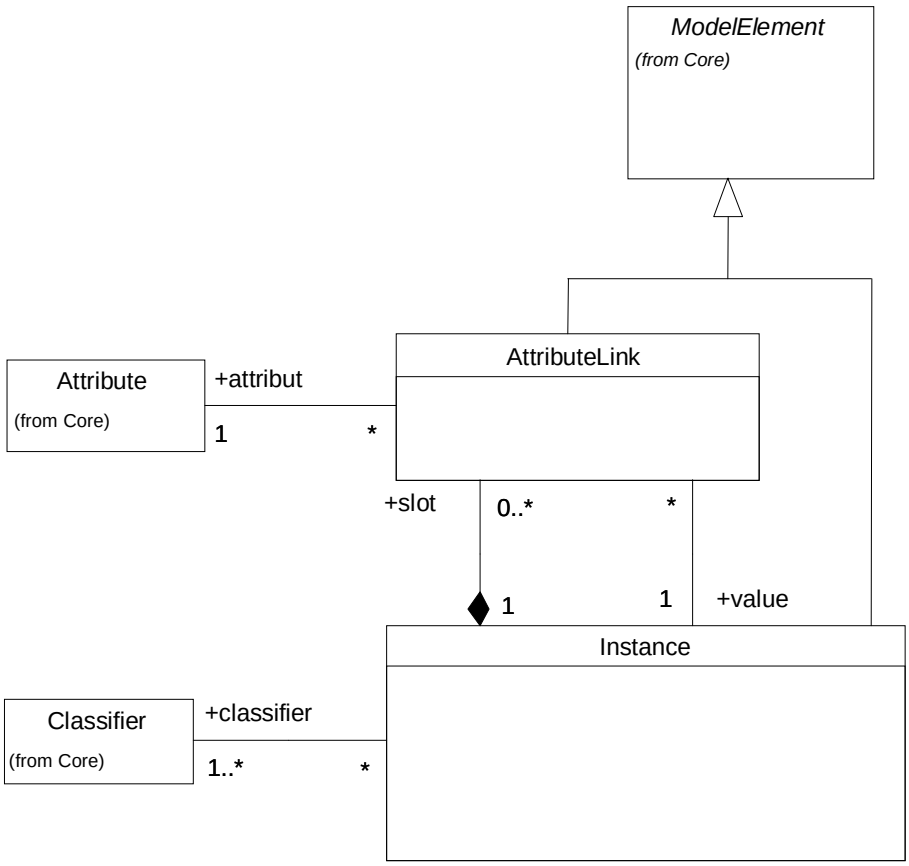
obrázek 22 Strom generalizace Instancí v UML 1.3

Uvedený diagram zavádí nové základní *Prvky modelu* pro modelování tzv. *Instancí*. Obecně jsou v UML instance chápány jako realizace svých předloh. Tyto předlohy jsou již v této knize deklarované

Prvky modelu, jako *Klasifikátor*, *Atribut*, *Třída*, *Asociace*, *Asociativní třída* apod. Vzniká tak jeden ze základních odvozených vztahů UML – vztah typu a její instance nazývaný někdy jako dichotomie.

Postupně probereme všechny tyto *Prvky modelu*, přičemž doplníme předešlý model stromu generalizace instancí o další prvky a vztahy.

3.7.2.1 Instance (Instance)



obrázek 23 Instance a její vztah ke Klasifikátoru

Metatřída *Instance* je přímý potomek *Prvku modelu* a je abstraktní metatřídou. Znamená to, že při modelování se používají až její potomci (viz obrázek 22 Strom generalizace Instancí v UML 1.3).

Každá *Instance* má vztah alespoň k jednomu *Klasifikátoru*, který deklaruje její strukturu a chování. Všimněme si, že UML 1.3 dovoluje, aby jedna instance měla vztah k 1..N *Klasifikátorům*, které definují její vlastnosti (nikoliv pouze jeden).

Instance obsahuje ve svých 0 až N slotech (myšleno podobně jako „slot = štěrbina“ při zasouvání součástek do strojů) prvků *Spojení atributů* (*AttributeLink*).

3.7.2.2 AttributeLink (Spojení atributu)

(viz obrázek obrázek 23 Instance a její vztah ke Klasifikátoru)

Spojení atributu (*AttributeLink*) reprezentuje konkrétní pojmenovaný slot - „štěrbinu v instanci“ (*slot in instance*) držící v sobě konkrétní hodnotu *Atributu*. *Instance* obsahuje 0..N takovýchto slotů pro hodnoty *Atributů*.

Všimněme si zajímavé asociace od Spojení atributu zpět na metatřídu *Instance*. *Instance* v tomto vztahu se jmenuje *value* . Je to *Instance* rovnající se hodnotě *Atributu*, většinou z metatřídy *Datová hodnota* – *Datavalue* (potomek *Instance*, viz dále).

Asociace z metatřídy *Spojení atributu (AttributeLink)* na metatřídu *Atribut (Attribute)* ukazuje, z jakého *Atributu* dané *Spojení Atributu* vzniklo.

3.7.2.3 DataValue (Datová hodnota)

Datová hodnota je instance bez žádné identity, tj. pouze hodnota dat. Bez identity znamená, že *Datová hodnota* je sice identifikována, ale nemá přímý analytický význam. *Datové hodnoty* jsou používány hlavně jako hodnoty *Atributů*.

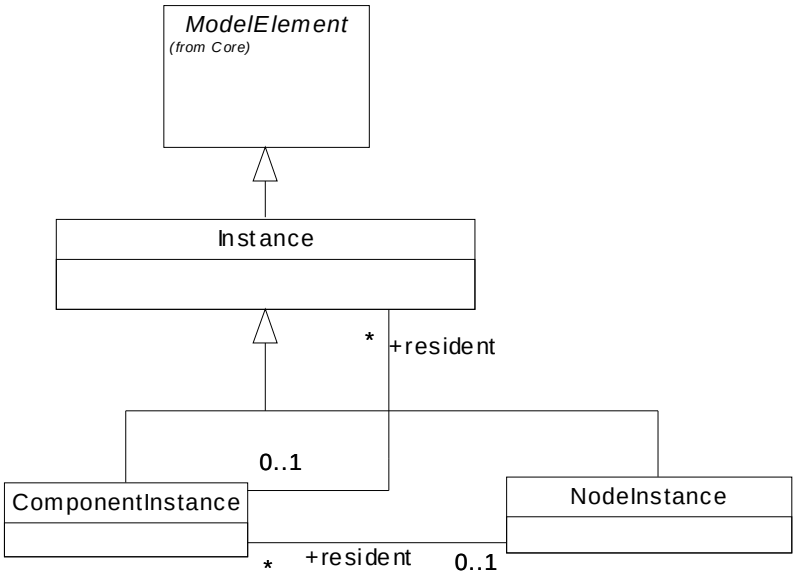
Poznámka: Neexistence identity je příznačná pro data. Na druhou stranu, data mají svůj kontext (významovou souvislost), která je dána samozřejmě okolnostmi (data z DB, parsování dat, data z formuláře atd.). Na druhou stranu Objekt má svoji identitu, má svůj analytický význam.

3.7.2.4 Object (Objekt)

Objekt (*Object*) je *Instance* (přímý potomek *Instance*), která pochází z *Třídy* (*Class*). UML povoluje, aby *Objekt* měl přiřazeno několik *Tříd*, což znamená, že daný *Objekt* může měnit svoje vlastnosti v průběhu svého života na základě přiřazení k dané *Třídě*.

3.7.2.5 ComponentInstance (Instance komponenty)

Zavedení syntaxe pro *Instanci Komponenty* ukazuje následující obrázek:



obrázek 24 Instance Komponenty a Instance Uzlu

Instance Komponenty (ComponentInstance) reprezentuje jednu „žijící“ instanci komponenty. Informace, ze které *Komponenty* pochází daná *Instance Komponenty*, je zavedena již na úrovni metatřídy *Instance* a to vztahem asociace vůči *Klasifikátoru*. Konkrétní *Instance Komponenty* residuje (tj. sídlí) na konkrétní *Instanci Uzlu*, například na konkrétním žijícím systémovém zdroji s pamětí a procesorem.

Uvnitř *Instance Komponenty* residuje několik *Instancí* (různého podtypu *Instance* včetně *Instancí Komponent*), viz vztah asociace *ComponentInstance – Instance* (nikoliv generalizace).

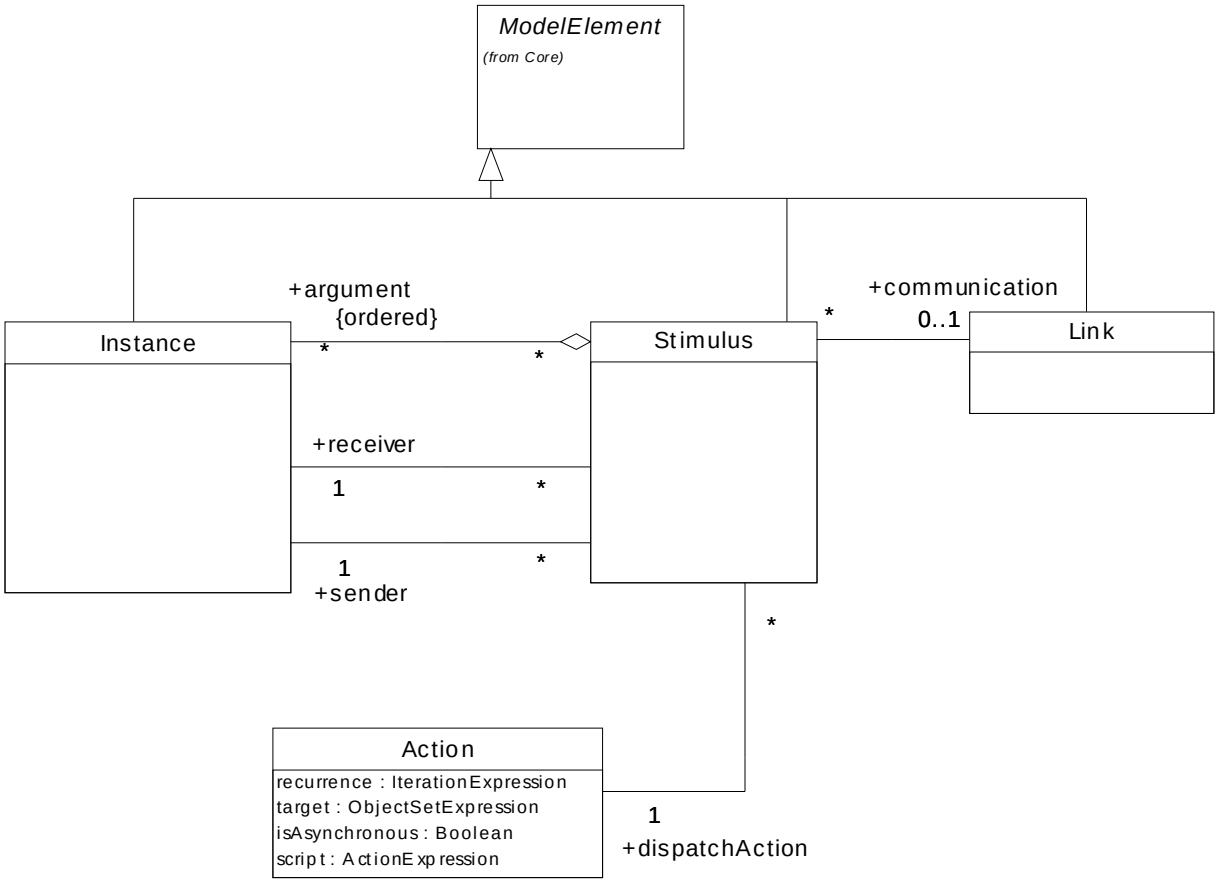
Instance Komponenty jako každá *Instance* může držet své hodnoty atributů (díky *Spojení atributů – AttributeLink*) a tedy může mít ještě své vlastní vnitřní stavy nezávislé na stavech *Instancí*, které sídlí uvnitř *Komponenty*.

3.7.2.6 NodeInstance (Instance Uzlu)

Instance Uzlu (NodeInstance) je instancí z *Uzlu*. Na dané *Instanci Uzlu* sídlí (residuje) několik *Instancí Komponent*.

3.7.2.7 Stimulus (Stimul)

Diagram pro zavedení *Stimulu* je následující:



obrázek 25: Stimul jako komunikace mezi dvěma instancemi

Stimul (Stimulus) definuje komunikaci mezi dvěma *Instancemi*. Může to být například *Signál*, nebo vyvolání *Operace* apod. Jedna z *Instancí* je sender (odesílatel), druhá je receiver (příjemce), viz dvě asociace na *Instance* s uvedenými názvy sender a receiver.

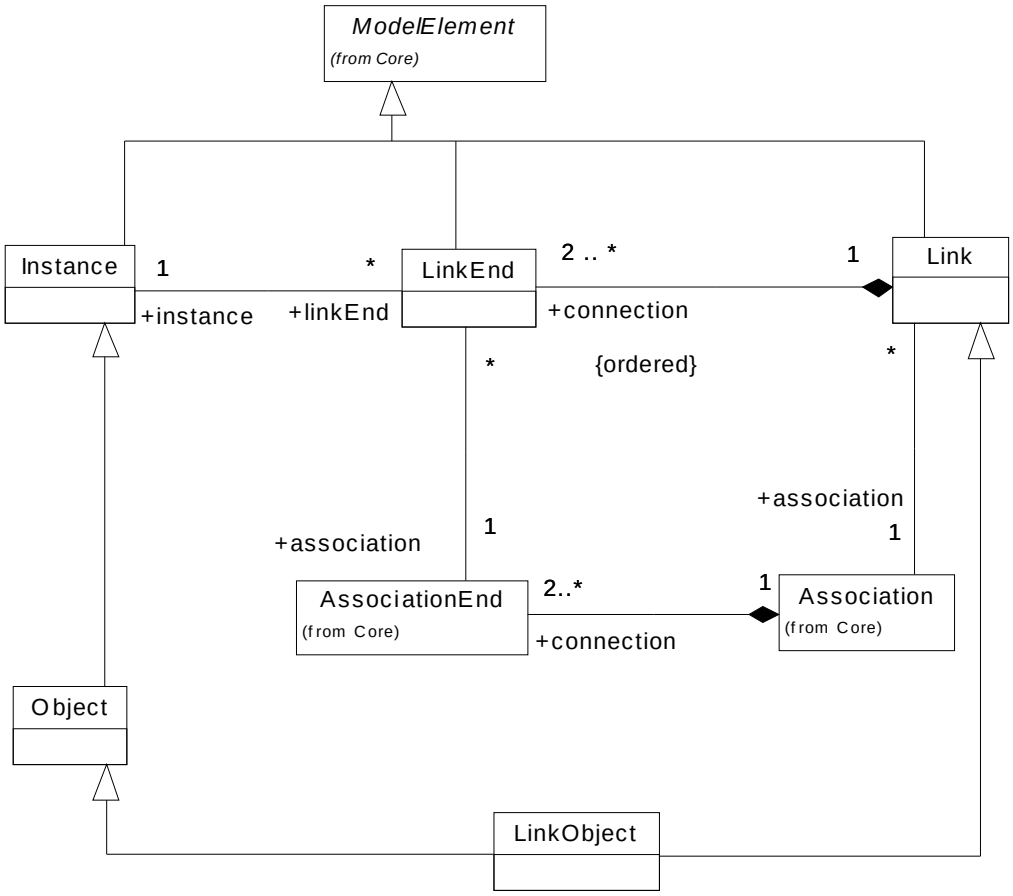
Komunikace *Stimulu* se děje podél jednoho *Spojení Instancí* (*Link*), což v modelu vyjadřuje *Asociace* na metatřídu *Link* s rolí metatřídy *Link* jako communication.

Stimul obsahuje argumenty (argument), které jsou reprezentovány *Instancemi* (viz agregace v metamodelu).

Stimul má vztah na právě jednu *Akci* (*Action*) s rolí *dispatchAction*. Je to ta *Akce*, která vyvolává daný *Stimul* (tj. rodič stimulu).

3.7.2.8 Link (Spojení Instancí) a LinkEnd (Konec Spojení Instancí),

Spojení Instancí (Link) reprezentuje propojení dvou *Instancí*, je konkretizací, tj. instancí *Asociace*. Vztahy v UML spjaté se *Spojením Instancí (Link)* zavádí následující diagram:



obrázek 26: Zavedení Spojení Instancí

Uvedený diagram je obrazem tvorby instancí již zavedených *Prvků modelu* spojených s *Asociací* a s *Asociativní třídou*.

Spojení Instancí (Link) je instancí *Asociace*. Má vztah k dané *Asociaci*, ze které pochází, *Asociace* v tomto vztahu se jmenuje association.

Spojení instancí obsahuje minimálně dva *Konce Spojení Instancí (LinkEnd)* jako kompozici. Tyto *Konce Spojení Instancí* odpovídají *Koncům Asociace*, ze kterých pocházejí.

Konec Spojení Asociace vidí jednu *Instanci*, která jej reprezentuje.

Poznámka: Modelování informačního systému pomocí těchto Prvků modelu se někdy nazývá jako instance model (model instancí) nebo object model (objektový model). Je přesným obrazem „instanciování“ Asociace.

3.7.2.9 LinkObject (Spojení Instancí - Objekt)

Na předešlém diagramu je zavedena metatřída *LinkObject (Spojení Instancí - Objekt)*. Metatřída je *LinkObjekt* potomkem jak metatřídy *Link (Spojení Instancí)*, tak metatřídy *Object (Objekt)*. Tento prvek reprezentuje jednu instanci *Asociativní třídy*.

3.8 Package Behavioral Elements - Collaboration (Spolupráce)

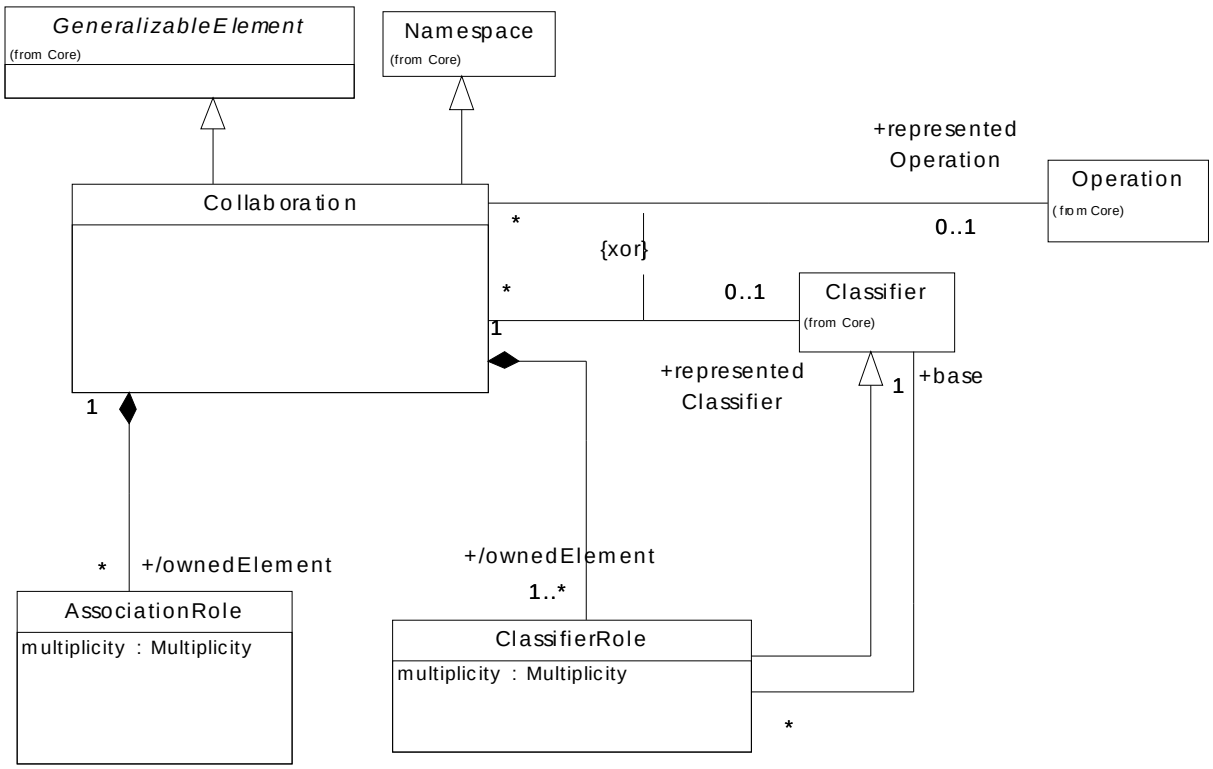
Svazek *Spolupráce (Collaboration)* patří do svazku *Prvky chování*. Zavádí základní pojmy modelu, které nějak specifikují spolupráci prvků. *Spolupráce* se týká zejména *Klasifikátorů*, *Asociací* a *Instancí* z těchto prvků.

Jeden a tentýž *Prvek modelu*, jako *Klasifikátor* nebo *Asociace* případně *Instance* může vystupovat v několika *Spolupracích* v různých rolích, také se může vyskytovat v několika *Spolupracích*, nebo dokonce v jedné *Spolupráci* v různých rolích.

Jedna *Spolupráce* určuje specifickou množinu vlastnosti *Prvků modelu*, které jsou nezbytně nutné k tomu, aby *Spolupráce* mohla proběhnout.

3.8.1. Collaboration (Spolupráce)

Diagram zavádějící prvky popisující *Spolupráci* je rozdělen do několika konceptů vyjádřených následujícími diagramy:



Metatřída *Spolupráce (Collaboration)* je v asociaci k metatřídě *Operation (Operace)* resp. *Classifier (Klasifikátor)*. Všimněme si použití *Omezující podmínky* na dvojici asociací *{xor}*, tj. platí „buď anebo“. Jeden z těchto vztahů asociace se použije pouze tehdy, když *Spolupráce (Collaboration)* popisuje buď přímo nějakou *Operaci*, anebo přímo chování nějakého *Klasifikátoru*.

Prvek modelování *Spolupráce (Collaboration)* zavádí jako základní pojmy *ClassifierRole (Role Klasifikátoru)* a *AssociationRole (Role Asociací)*. Tyto *Prvky modelu* jsou jako role jsou obsaženy ve *Spolupráci*.

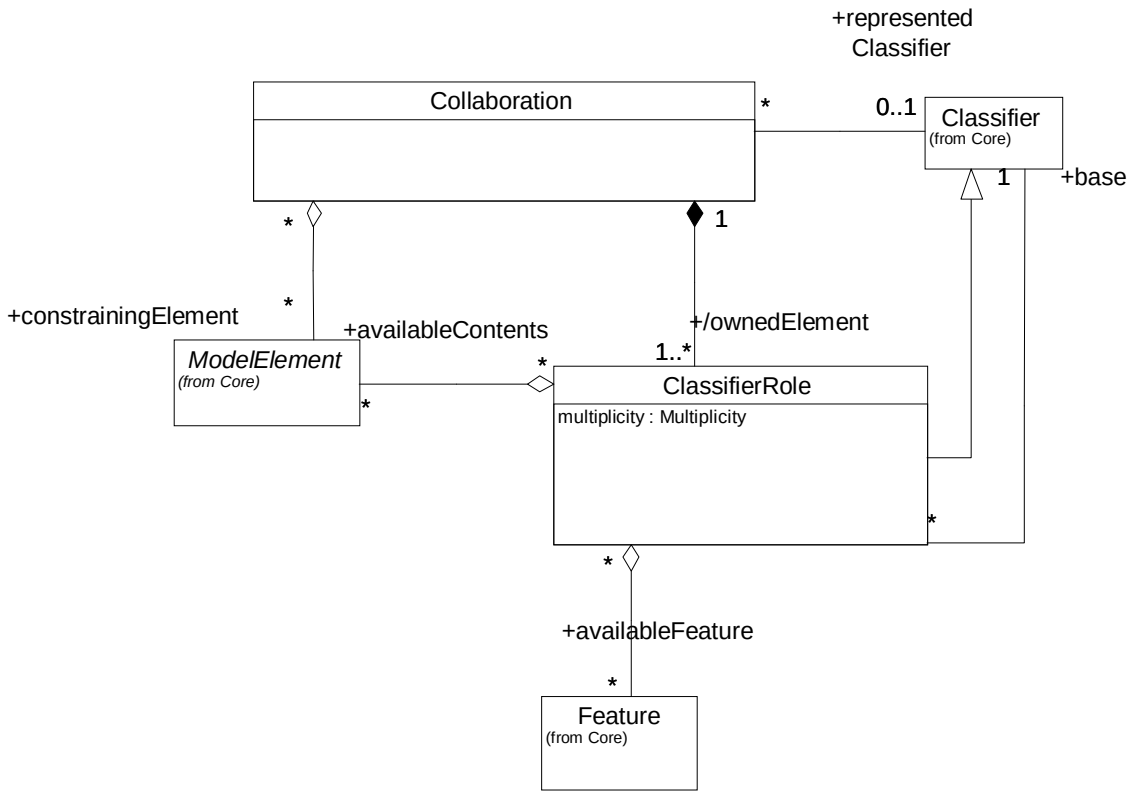
Role, jak bude podrobněji vysvětleno dále, je zúženým pohledem na vlastnosti *Klasifikátoru* anebo *Asociace* vzhledem k souvislostem nutných pro danou *Spolupráci*.

3.8.2. ClassifierRole (Role Klasifikátoru)

Role Klasifikátoru definuje určitou roli Klasifikátoru hrající v dané *Spolupráci*. *Role Klasifikátoru* specifikuje zúžený pohled na vlastnosti *Klasifikátoru*, které jsou definovány podmínkami nutnými pro *Spolupráci*. S trochou nadsázky můžeme *Roli Klasifikátoru* chápat jako „podmnožinu vlastností“ *Klasifikátoru*, kterou určujeme na základě požadavků pro dobře fungující *Spolupráci*.

Poznámka: „aby Klasifikátor uměl hrát tuto Roli Klasifikátoru, musí mít tyto vlastnosti...“

Pohled na zavedení vlastností *Role Klasifikátoru* poskytuje následující obrázek:



obrázek 27: Zavedení Role Klasifikátoru

Všimněme si, že *Role Klasifikátoru (ClassifierRole)* vstupuje jako násobná kompozice do *Spolupráce*, tj. *Role Klasifikátorů* „skládají v kompozici“ *Spolupráci*.

Metatřída *Role Klasifikátoru* má vztah asociace ke *Klasifikátoru*, který se jmenuje *base*. Je to *Klasifikátor*, jehož zúžený pohled *Role Klasifikátoru* reprezentuje. Kromě toho sama *Role Klasifikátoru* je potomkem *Klasifikátoru*, tedy sama je chápána jako *Klasifikátor*.

Role Klasifikátoru obsahuje jako agregaci skupinu *Rysů (Feature)* nazvaných availableFeature (dostupné rysy), které jsou podmnožinou *Rysů Klasifikátoru* base a jsou použity v této *Spolupráci (Collaboration)*. Jinak řečeno, z hlediska modelování jsou to ty *Rysy*, které byly odvozeny na základě existence této *Spolupráce* v systému, a patří do *Klasifikátoru*, který odpovídá dané *Roli Klasifikátoru*.

Role Klasifikátoru dále obsahuje jako agregaci množinu *Prvků modelu (ModelElement)* nazvaných jako availableContents (dostupný obsah). Je to množina dalších *Prvků modelu*, které jsou nutné pro to, aby *Spolupráce* mohla proběhnout, avšak nejsou to *Rysy Klasifikátoru*. Většinou se zde uvádějí další *Klasifikátory* ze stromu generalizace včetně odpovídajících vztahů *Generalizace*, které poskytují vlastnosti danému *Klasifikátoru* base díky dědění a jsou nutné pro *Spolupráci*.

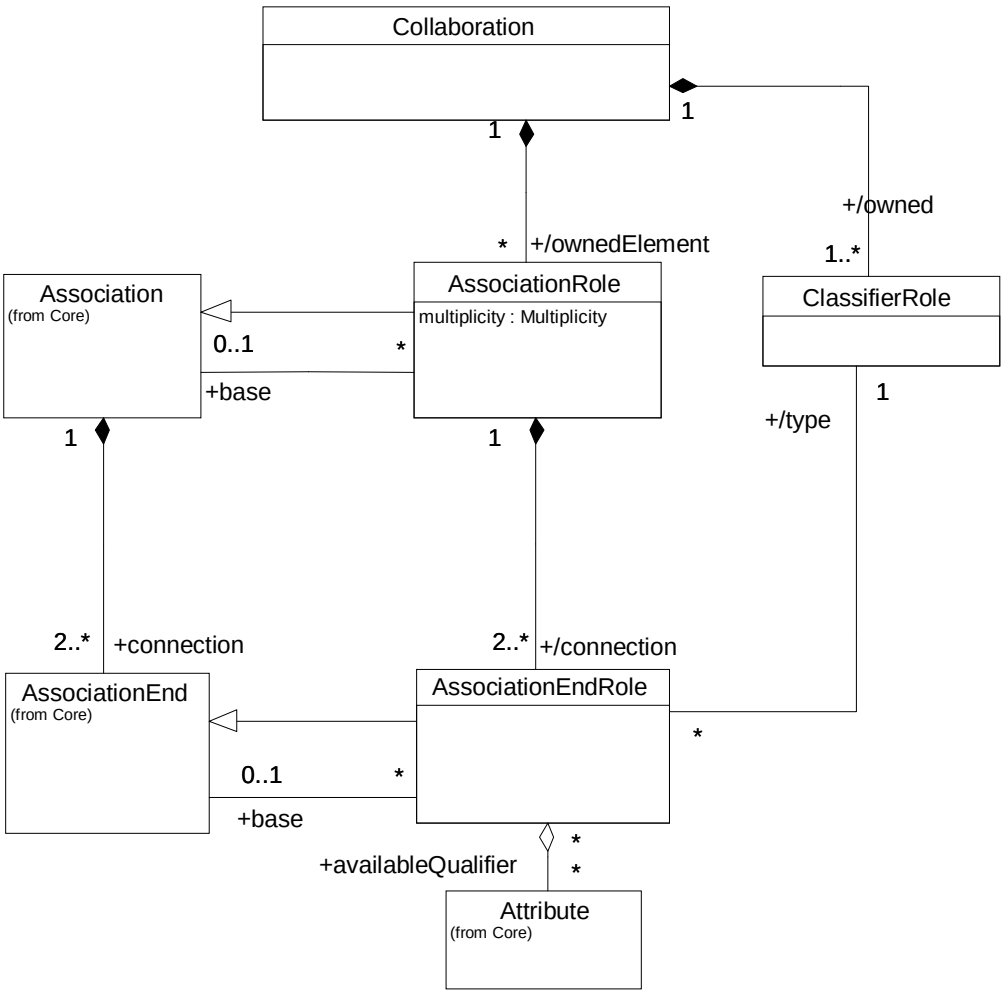
Poznámka: Pomocí zavedení dostupných Rysů (availableFeature) vždy vyjádříme, jaké vlastnosti má mít samotný Klasifikátor base dané Role Kalsifikátoru. Pomocí dostupného obsahu (availableContents) vyjádříme jinou skutečnost, například to, že Klasifikátor je potomkem jiného Klasifikátoru a díky tomu je Spolupráce možná.

Atributy ClassifierRole

- multiplicity – násobnost. Udává, jaká je násobnost dané instance *Role Klasifikátoru (ClassifierRole)* ve *Spolupráci (Collaboration)*.

3.8.3. AssociationRole (Role Asociace)

Na následujícím obrázku je diagram zavádějící Role Asociace:



obrázek 28: Zavedení Role Asociace

Podobně, jako je zavedena *Role Klasifikátoru*, je zavedena i *Role Asociace (AssociationRole)* a s ní *Role Konce Asociace (AssociationEndRole)*.

Připomeňme, že *Spolupráce* (*Collaboration*) určuje, jaké má mít vlastnosti daný *Klasifikátor* a to takové, aby *Spolupráce* mohla proběhnout. Zavádí se proto *Role Klasifikátoru*, který reprezentuje zúžený pohled na vlastnosti *Klasifikátoru*. Těmito vlastnostmi jsou tzv. dostupné rysy (availablefeature) z metatřídy *Feature* a dostupný obsah availableContents z metatřídy *ModelElement* („libovolné“ *Prvky modelu*).

Podobně *Spolupráce* určuje, jaké má mít vlastnosti *Asociace* a její *Konce Asociace*. K tomu se zavádí zúžený pohled na *Asociaci* v rámci této *Spolupráce* a zúžený pohled na *Konce Asociace*, což je reprezentováno odpovídajícími rolemi.

Hlavním důvodem zavedení *Rolí Asociace* a *Rolí Konců Asociace* je skutečnost, že komunikace mezi instancemi probíhá podél propojení mezi instancemi (link), tj. je třeba vyjádřit nějak *Role Asociací* jako role těchto propojení v komunikacích.

Role Asociace (AssociationRole) určuje použití dané *Asociace* v dané *Spolupráci*. *Role Asociace* má vztah k base *Asociaci*, tedy k *Asociaci*, kterou reprezentuje ve své roli (viz předešlý diagram na obrázek 28: Zavedení Role Asociace). *Role Asociace* podává zúžený pohled na *Asociaci* pouze v kontextu dané *Spolupráce*. Současně je potomkem *Asociace*, takže je také chápána jako *Asociace*.

Důležitý je vztah *Role Asociace* vůči *Rolím Konců Asociace*, které jsou jako kompozice uvnitř *Role Asociace* obsaženy (viz název connection). Odpovídající *Role Konců Asociace* mají vztah ke *Koncům Asociace* (viz base u *AssociationEnd* vůči *AssociationEndRole*), přičemž tyto *Konce Asociace* odpovídají *Koncům Asociace* dané *Asociace*.

Poznámka: Jak je možné pozorovat, existuje zde několik paralelních vztahů mezi meta-Prvkem a jeho meta-Rolí. Vždy je vyjádřen jako vztah s názvem konce na straně meta-Prvku jako base.

Prvním je vztah Klasifikátor versus Role Klasifikátoru. Dalším je vztah Asociace versus Role Asociace, další je Konec Asociace versus Role Konce Asociace. Aby byl model konzistentní, musí platit pravidlo, že existuje-li nějaký vztah mezi meta-Rolemi, potom existuje odpovídající souhlasný vztah mezi meta-Prvky, tj. musí odpovídat vztahu mezi Rolemi.

Atributy AssociationRole

- multiplicity – násobnost. Udává, jaká je násobnost dané instance *Role Asociace (AssociationRole)* ve *Spolupráci (Collaboration)*.

3.8.4. AssociationEndRole (Role Konce Asociace)

Role Konce Asociace specifikuje, že daný *Konec Asociace* vstupuje do *Spolupráce* (viz obrázek 28: Zavedení Role Asociace). Je součástí *Role Asociace* jako kompozice v násobnosti (2..*).

Role Konce Asociace specifikuje v modelu pomocí agregace tzv. availableQualifier, tj. dostupné kvalifikátory (viz diagram). Jsou to *Kvalifikátory*, které vstupují do *Spolupráce*.

Atributy AssociationEndRole

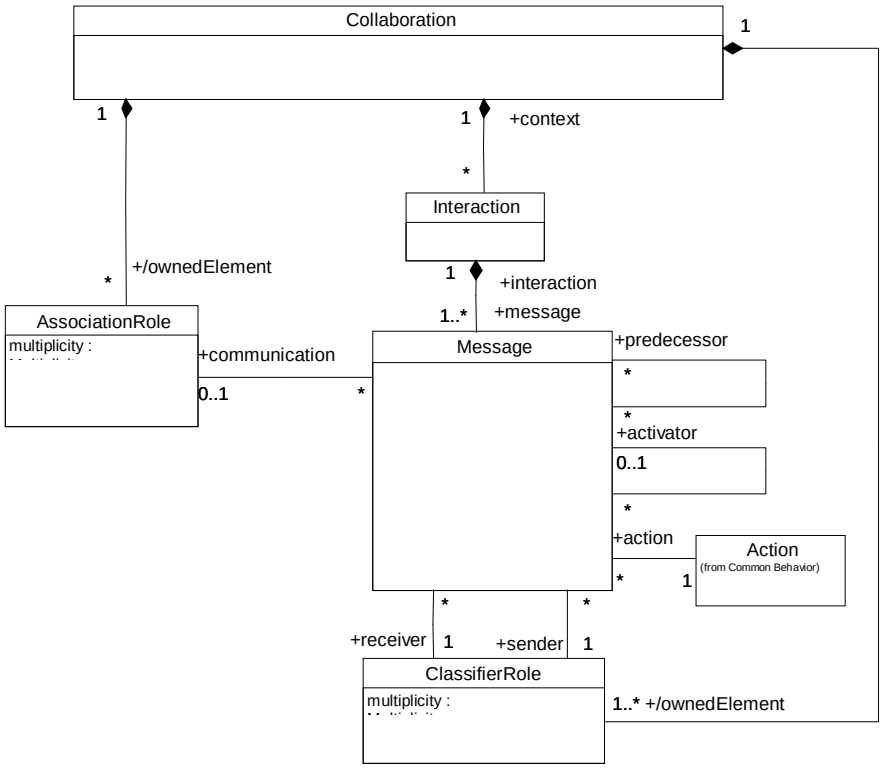
- collaborationmultiplicity (násobnost spolupráce) počet *Konců spojení Instancí* vystupujících v dané *Spolupráci*

3.8.5. Interaction (Interakce)

Předešlé úvahy o rolích by vedly v modelování *Spolupráce* pouze k vyjádření toho, kdo spolupracuje, (tj. rolí). V žádném případě neurčují, co je vlastně míněno touto spoluprací (určují kdo, ale nikoliv kdo s kým a jak).

K popisu interakcí a způsobu provedení těchto interakcí slouží metatřídy *Interaction (Interakce)* a *Message (Zpráva)*.

Zavedení těchto metatříd ukazuje následující diagram:



obrázek 29: Zavedení Interakce a Zprávy

Spolupráce (Collaboration) obsahuje N *Interakcí (Interaction)* jako kompozici. Každá Interakce obsažená ve *Spolupráci* se zavádí v tzv. kontextu (souvislosti) této *Spolupráce*. Jedna *Interakce* obsahuje v kompozici N *Zpráv (Message)* jako komunikace mezi rolemi.

Modelování spolupráce má tak několik úrovní, viz jednotlivé kompozice v metamodelu:

- *Spolupráce* určuje širší kontext (viz diagram), tj. souvislost. Skládá se z *Interakcí*.
- Každá *Interakce* obsažená ve *Spolupráci* řeší nějakou úlohu pomocí několika *Zpráv*, které obsahuje.
- *Zprávy* reprezentují určitou jednu komunikaci mezi rolemi.

3.8.6. Message (Zpráva)

Zpráva (Message) je zavedena v diagramu viz obrázek 29: Zavedení Interakce a Zprávy.

Zpráva reprezentuje určitou jednu komunikaci mezi rolemi, vede například k vyvolání *Signálu*, provedení *Operace*, ke zničení objektu atd.

Zpráva také definuje přesně odesílatele (*sender),* což jedna *Role Klasifikátoru* a určuje příjemce (*receiver*) *Zprávy*. Obě *Role Klasifikátoru* jsou v seznamu *Rolí Klasifikátorů* dané *Spolupráce* jako *ownedElements*. Zprávu může doprovázet také *Akce (Action)* způsobující *Stimul* (viz diagram).

Zpráva si může ukazovat na aktivátor (activator), což je *Zpráva*, která aktivuje v daném prostředí vyvolání této zprávy.

Zpráva je svázána s nějakou *Rolí Asociace*, tj. probíhá „podél“ nějakého spojení instancí daného touto asociací.

Navíc *Zpráva* může vidět všechny své *Zprávy* předchůdce (predecessor), které musí nejprve proběhnout, aby tato *Zpráva* byla zahájena.

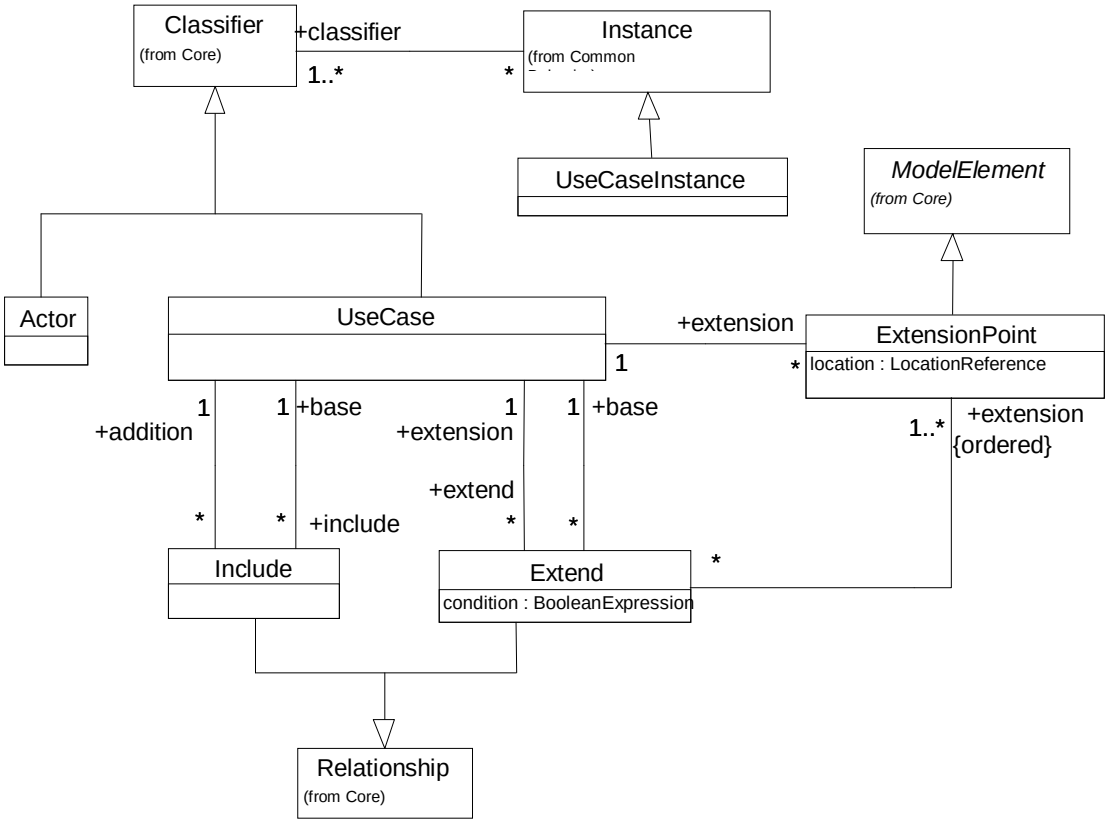
3.9 Package Use Cases (Svazek Případy užití)

Poznámka: V předešlých publikacích jsem se přikláněl k překladu „use case = užitná činnost“. Nyní myslím, že překlad „případ užití“ více odpovídá anglickému „use case“ a také lépe odpovídá významu tohoto slova.

Prvky modelu ze svazku *Případy užití* umožňují zavést v modelování další možný pohled na systém. Jak sám název napovídá, jedná se o pohled jako na „možné užití systému“. Tento pohled je nejblíže k raným fázím analýzy, kdy se v komunikaci s budoucím uživatelem získává obraz případů užití systému. Synonymem pro získání tohoto obrazu jsou „klasické dotazy“: Co má systém vlastně umět, k čemu vlastně je, jaké je jeho užití? Modelování pomocí *Případů užití* zavádí do těchto jinak chaotických dotazů pravidla (tj. přesnou sémantiku).

Poznámka: O použití modelování pomocí Případů užití viz blíže kniha Objektové modelování a UML v praxi 2000. V této knize, kterou nyní držíte v ruce, je podána přesná sémantika UML 1.3 vzhledem ke standardu OMG.

Diagram zavádějící metatřídu *Případ užití* (UseCase) a další příslušné metatřídy ukazuje následující obrázek:



obrázek 30: Prvky modelu související s Případy užití

3.9.1. UseCase (Případ užití)

Případ užití (UseCase) se používá k definici chování systému nebo jiné samostatné entity (například *Třídy*) bez podrobné specifikace vnitřní struktury systému, tj. jedná se o pohled při modelování na systém z hlediska jeho užití.

Každý *Případ užití* specifikuje řadu nějakých akcí a také interakce s okolím.

Poznámka: Vyhnout se popisu struktur pojmů nelze úplně. Definice chování se neobejde bez zavedení pojmů problémové domény, které toto chování popisují a které vstupují do různých vztahů. Tyto pojmy odpovídají pohledu uživatele na systém v pojmech problémové domény .

Příklad: Samotný název Případu užití "Založení účtu klientovi" automaticky implikuje pojmy Účet a Klient a nejenom to, měl by implikovat přesnější pojem Účet Klienta. Blíže viz zmíněná publikace o objektovém modelování.

Metatřída *Případ užití* je definována jako potomek metatřídy *Klasifikátor (Classifier)*. Znamená to, že spadá do stejné rodiny, jako je *Třída, Datový typ*, atd. (viz obrázek 6: část modelu z Core (Jádro) -

Classifiers (Klasifikátory). Z *Případu užití* lze vytvářet tzv. *Instance Případů užití (UseCaseInstance*, popis viz dále)

Připomeňme, že *Klasifikátor* je *Zobecnitelným prvkem* (*GeneralizabelElement*), proto může *Případ užití* vstupovat do vztahů *Generalizace*. Vztahu *Generalizace* mezi dvěma *Případy užití* použijeme tehdy, když jeden *Případ užití* jako potomek potřebuje převzít vlastnosti jiného *Případu užití* (viz příklad „Svoz odpadu“ v publikaci o objektovém modelování).

Možné typy vztahů mezi *Případy užití* jsou pouze tři:

- *Generalization (Generalizace)*
- *Extend (Rozšíření)*
- *Include (Začlenění)*

Ještě než bude vysvětlena podstata interakce *Extend* a *Include* (*Generalization* tj. *Generalizace* již byla vysvětlena), musíme upozornit na jednu důležitou okolnost. Interakce mezi dvěma *Případy užití* má vždy takovou povahu, že oba dva *Případy užití* popisují jednu a tu samou oblast chování. Je to jakási obdoba nějaké „algebraické operace užití nad jedním případem užití“. Pokud tedy dva *Případy užití* vstupují do jedné z těchto tří vyjmenovaných interakcí, potom dohromady dávají popis nějakého užití systému. Nejsou tedy vzdáleny v tom smyslu, že by každý z nich popisoval úplně jiné chování (něco jiného). V určitém smyslu se *Případy užití* v interakci doplňují a jeden z nich je na druhém závislý. Mezi dvěma *Případy užití*, které nemají takovýto vztah (nemají takovouto souvislost), nemůže existovat interakce vyjmenovaná v předešlém odstavci.

Poznámka: Pro vysvětlení, které si můžeme dovolit u Zobecnění. Představme si, že máme odhalen v modelu Případ užití, nazvěme jej „Případ užití A“. Máme v něm popsáno chování systému jako určité jedno jeho užití. Vznikne „Případ užití B“ a zjistíme, že B je pouze specializací A, tj. má všechno z B jako potomek A, ale něco trochu navíc jako svou specializaci. Zavedeme interakci mezi A a B jako A zobecňuje B pomocí vztahu Generalizace. Pokud si přečteme obsah B bez A, nalezneme uvnitř B pouze to, co je speciální.

Avšak celé B je chápáno jako „Případ užití B“ takto: Čti B a současně použij A jako předka.

Sémantika UML 1.3 nepopisuje, *jak* se mají Případy užití vyhledávat. Je zřejmé, že u velmi rozsáhlých systémů je vyhledání všech *Případů užití* poměrně složitou záležitostí. Pro nalezení všech *Případů užití* se doporučuje použít metody hierarchického postupu odshora dolů, ale zde je určitý háček: Vztah hierarchie (ve smyslu dekompozice *Případů užití*), který se často užívá pro vyhledávání všech *Případů užití* a je v literatuře doporučován, není korektně zavedenou interakcí mezi *Případy užití*. Dva *Případy užití* ve vztahu hierarchie se totiž netýkají „téhož problému“, tj. jednoho užití systému, ale mohou být úplně disjunktní.

Z hlediska čistoty UML by měl být nahrazena interakce hierarchie mechanismem hierarchie *Svazků* (*Package*) *Případů užití*, které mohou zahrnovat jak další *Svazky Případů užití* anebo přímo *Případy užití*.

Prakticky se však mnohdy používá sice nepřesný, ale přehledný hierarchický vztah mezi *Případy užití* (tedy rozklad *Případů užití* jako by to byly *Svazky Případů užití*).

Poznámka: Můj osobní názor je ten, že by Případ užití měl být také potomkem Svazku. Pak by se mohl on sám „hierarchicky rozpadat“ na další Svazky resp. Případy užití .

V některých případech se zavádí tzv. *business process modeling* jako tzv. modelování procesů (není součástí přímo syntaxe UML), které v konečném důsledku obsahují *Případy užití*, přičemž mezi procesy je umožněn vztah hierarchie. Koncové procesy této dekompozice zavádějí *Případy užití*.

Poznámka: Je třeba si uvědomit, že vztah dekompozice je v UML „jednou provždy" vyřešen korektně pro všechny Prvky modelu jedním ze základních mechanismů UML, kterým je zavedení Svazku (Package). Proto například pro tzv. business process modeling UML doporučuje v části zvané Standard Profiles zavést pro pojem „process" ekvivalent tzv. UseCasePackage (Svazek Případů užití) jako stereotyp pro Svazek.

Prvek *Případ užití* může jako *Klasifikátor* nejenom vstoupit do vztahu *Generalizace*, ale také se účastnit vztahu *Asociace* vůči jinému *Klasifikátoru*. U *Případu užití* se tímto druhým *Klasifikátorem* vystupujícím v *Asociaci* může stát pouze prvek okolí *Actor (Aktér)*. Z toho důvodu není tento vztah na předešlém diagramu obrázek 30: Prvky modelu související s Případy užití zobrazen, je odvozen z dědění po *Klasifikátoru*. Asociace *Aktér* versus *Případ užití* definuje existenci komunikace přes rozhraní zkoumaného subjektu (například systému, *Třídy* apod.).

Poznámka: nezaměňujme Asociaci Aktér – Případ užití vedoucí ke komunikaci přes rozhraní systému s dalšími interakcemi pouze mezi Případy užití, Include a Extend.

Metatřída *Případ užití* obsahuje tzv. *ExtensionPoints* (*Body rozšíření*), viz diagram, jejichž vysvětlení bude podáno až po objasnění *Extend (Rozšíření)*.

3.9.2. Include (Začlenění)

Vztah *Začlenění (Include)* označuje, že jeden *Případ užití* začleňuje, tj. obsahuje (beze zbytku) druhý *Případ užití*.

Začlenění (Include) je metatřídou, která je potomkem *Relace*, tj. spadá tedy do téže rodiny, jako například *Asociace*, *Generalizace* a *Tok*, viz obrázek 5: část metamodelu z Jádra (Core) určující tzv.relace, diagram Jádro - Relace.

Začlenění (Include) má svůj směr vyjádřený dvěma *Prvky modelu* na jeho koncích, kterými jsou *Případy užití*. Jeden *Případ užití* je označen v metamodelu jako base, druhý *Případ užití* jako addition (viz diagram). Vztah je takový, že addition ukazuje na *Případ užití*, který přidává chování do druhého, tj. base *Případu užití*. Směr je tedy definován tak, že *Případ užití* na straně base „používá" *Případ užití* na straně addition.

Použití *Include (Začlenění)* je nejjednodušší vyjádření opětovné použitelnosti již na úrovni modelování v *Případech užití*.

3.9.3. Extend (Rozšíření)

Vztah *Extend (Rozšíření)* označuje, že jeden *Případ užití* může být extendován (rozšířen) o chování jiného *Případu užití*. *Případ užití*, který je extendován (rozšiřován) je na diagramu v modelu označen jako base. *Případ užití*, který rozšiřuje chování extendovaného, je označen jako extension (extenze)

Existuje rozdíl mezi *Začleněním (Include)* a *Rozšířením (Extend).* Vztah *Začlenění (Include)* pouze „natvrdo“ přebírá a vkládá část chování jednoho *Případu užití* do chování jiného *Případu užití* a další informace nejsou z hlediska modelování proto vůbec třeba. Oproti tomu *Extend (Rozšíření)* definuje navíc určitou podmínku k tomu, aby extenze nastala. Tato podmínka je vyjádřena v atributu *Rozšíření (Extend)* jako condition (podmínka).

Poznámka: Připomeňme, že mezi Případy užití existuje ještě třetí interakce, Generalizace.

Body, ve kterých se podmínka extenze uplatňuje, tj. body, kde se chování rozšiřuje, se označují jako *ExtensionPoint (Body rozšíření)*. Každý vztah *Rozšíření (Extend)* vidí 1 až N *Bodů rozšíření*, kde dochází právě k tomuto *Rozšíření* v exetendovaném *Případu užití*.

Atributy Extend

- condition – podmínka. Výraz typu Boolean, který specifikuje podmínku, která musí být splněna, aby došlo k extenzi.

3.9.4. ExtensionPoint (Bod rozšíření)

Bod rozšíření označuje lokaci v *Případu užití*, ve které nastává extenze při splnění podmínky (condition) daného *Rozšíření*. Všimněme si (viz obrázek 30: Prvky modelu související s Případy užití), že vztah mezi *Extend* a *ExtensionPoint* má násobnost 1 .. *, tj. pokud existuje prvek *Extend*, pak tento *Extend* má alespoň jeden *Bod Rozšíření*.

Atributy ExtensionPoint

- location (lokace) určuje místo (lokaci), kde může dojít v daném *Případu užití* k extenzi.

3.9.5. Actor (Aktér)

Aktér (Actor) definuje množinu logických rolí, kterou hraje uživatel zkoumané entity (systému apod.) při interakcích s touto entitou. *Aktér* hraje omezenou roli vzhledem k jednomu konkrétnímu *Případu užití* ze všech, se kterým interaguje.

Poznámka: Zjednodušeně řečeno, jedná se o prvek okolí, který komunikuje se systémem. Reprezentuje vnější prvek zavedený v logickém kontextu Případů užití, se kterými daný Aktér komunikuje. Například určitě není Aktérem klávesnice, pokud nemodelujeme přímo funkcionalitu počítače.

Metatřída *Aktér (Actor)* je potomkem metatřídy *Klasifikátor*. *Aktér* má svůj název a komunikuje (přes *Asociace*) s jedním nebo několika *Případy užití*, ve kterých hraje roli.

Protože je *Aktér* potomkem *Klasifikátoru* a tedy také *Zobecnitelného prvku*, je možné v modelu zapsat vztah *Generalizace* mezi *Aktéry*. Málokdy se však v praxi používá.

Aktér reprezentuje prvek okolí, a proto při modelování musíme být velmi obezřetní, abychom se nedopustili prohřešku proti zásadě dodržet tzv. anonymitu klienta zkoumané entity. V některých případech je určení *Aktéra* bezpředmětné a nemá význam, protože prvkem může být „*kdokoliv, kdo potřebuje služby entity*".

Příklad: Pokud vytvoříte komponentu Počítadlo pro obecné počítání a sledování výskytu zkoumaných veličin, potom toto počítadlo lze dosadit jako objekt do libovolné části systému, tj. tam, kde je třeba. Aktérem pro počítadlo je ten, kdo potřebuje počítat a nelze jej dopředu předjímat.

Hlavním posláním hledání *Aktérů* je nalézt všechny *Případy užití* systému. V případě, že máme všechny *Případy užití* správně identifikovány, pro samotný systém jsou nadále *Aktéři* bezpředmětní, protože jsou mimo systém a tedy anonymní.

Poznámka: Klasickým příkladem správné úvahy při práci s Aktéry je následující dotaz vůči uživateli systému při úvodních konzultacích: „A co vedoucí oddělení, nebude on potřebovat nějaké údaje ze systému?" Když se zjistí, co má systém vykonávat, potom role Aktéra skončila.

Pojem *Aktér* je v určitém pohledu relativní: Pokud systém rozdělíme na menší *Subsystémy*, které mezi sebou komunikují, potom ostatní okolní Subsystémy původního systému se vůči zkoumanému *Subsystému* stávají *Aktéry*.

Poznámka: Nejmenší smysluplnou jednotkou takovéhoto dělení je nakonec Třída. Lze tedy vytvořit samostatný diagram Případů užití pouze pro Třídu.

V ojedinělých případech se používá v modelování také interakce mezi *Aktéry* vyjadřující nějakou důležitou komunikaci probíhající mimo systém, avšak systému se týkající. V tom případě lze použít *Asociaci* mezi *Aktéry* podobně, jako se používá *Asociace* mezi *Aktérem* a *Případem užití* (připomeňme, že *Aktér* je potomkem *Klasifikátoru*). Tato *Asociace* reprezentuje komunikaci mezi *Aktéry*. Protože se jedná o komunikaci mimo systém, musí existovat pádný důvod pro prezentaci takovéto interakce v modelu.

Poznámka: V praxi jsem se setkal s nutností znázornit komunikaci mezi Aktéry mimo systém při tvorbě systému, který vyžadoval vyšší stupeň bezpečnosti (GSM Banking, Internet Banking) a kde v určitých krocích došlo k předávání informace „na papíře" mimo systém. Jednalo se o princip stránkovaných sešitů s podepsáním předávky šifrovacích kalkulaček připravených k aktivaci pro klienty mezi administrátory apod. Díky tomuto zápisu se podařilo vyjádřit celkový chod budování bezpečnosti celého systému.

3.9.6. UseCaseInstance (Instance Případu užití)

Instance *Případu užití* se chápe jako jeden výskyt *Případu užití*, který vykonává konkrétní sekvenci *Akcí* specifikovaných v *Případu užití*. Metatřída *Instance Případu užití* je potomkem metatřídy *Instance*.

3.10 Package State Machines (Svazek Stavové stroje)

Svazek *Stavové stroje (Package State Machines)* je sub-svazkem svazku *Prvků chování* (package *Behavioral Package*).

Prvky modelů ze svazku *Stavové stroje* slouží k popisu chování pomocí konečných stavových a přechodových systémů.

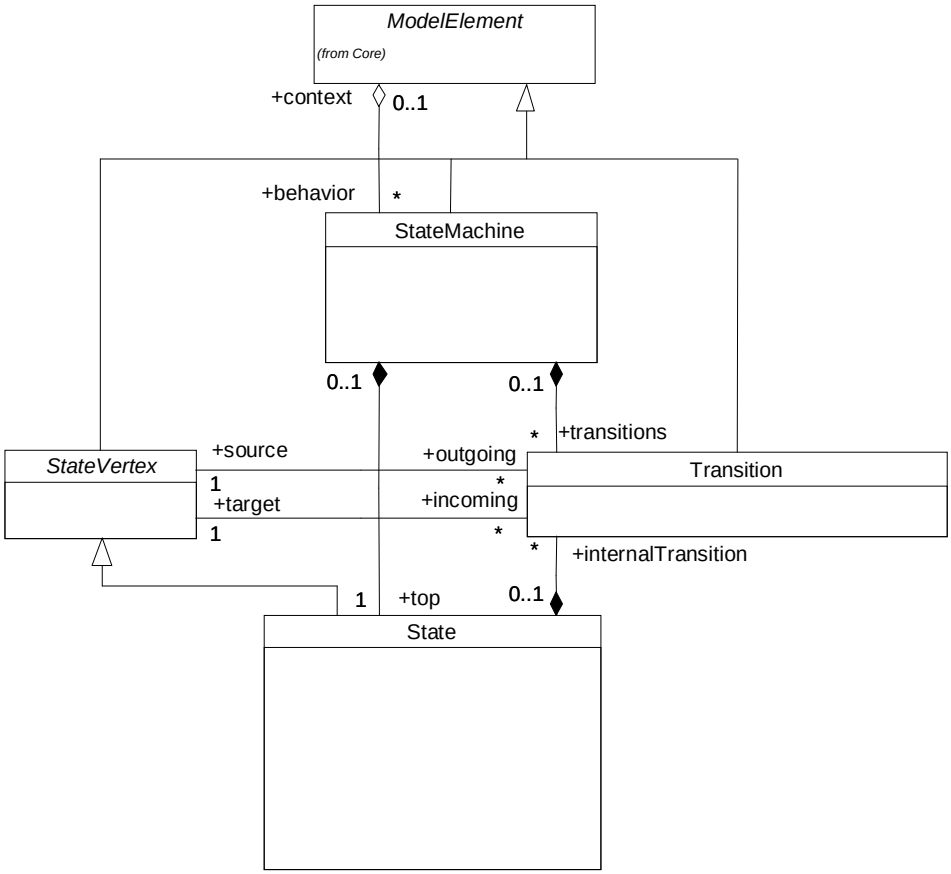
Poznámka: Teorie stavových strojů je jednou z prvních v teorii programování (konečné automaty). V UML je modelování pomocí stavových rozpracováno do objektově orientovaného prostředí.

Stavové stroje mohou popisovat chování rozličných *Prvků modelu*, nejčastěji jsou však předmětem zkoumání *Třídy*, *Komponenty* resp. jiné části systému.

Syntaxe prvků ve svazku *Stavové stroje* dává základ pro další syntaxi ve svazku *Grafy Aktivit*. Svazek *Grafy Aktivit* lze považovat za pokračování syntaxe převzaté ze svazku *Stavové stroje*.

3.10.1. State Machine (Stavový stroj)

Základní diagram zavádějící *Stavový stroj (State machine)* je následující:



obrázek 31: Zavedení Stavového stroje

Základní myšlenková konstrukce zavedení *Stavového stroje* je následující:

Libovolný *Prvek modelu (ModelElement)* může obsahovat N *Stavových strojů*, které popisují jeho chování (behavior). Většinou je tímto *Prvkem modelu Klasifikátor*. Metatřída *Stavový stroj (StateMachine)* je přímým potomkem *Prvku modelu*.

Stavové stroje patří do jednoho kontextu (významové souvislosti) daného *Prvku modelu*. To je vyjádřeno agregací, která z druhé strany vyjadřuje, že *Prvek modelu* obsahuje N *Stavových strojů* jako svoje chování (behavior).

Stavový stroj obsahuje jako svoji kompozici tzv. top *Stavy (State)* a *Přechody (Transition)*. Top *Stav* je jeden hierarchicky nejvyšší *Stav* daného *Stavového stroje*. Top *Stavy (State)* a *Přechody (Transition)* jsou v daném *Stavovém stroji* obsaženy jako kompozice (jsou pouze jeho).

V metamodelu existuje abstraktní metatřída *StateVertex (Stavový vrchol)*, jejíž význam není v tomto diagramu patrný, protože nejsou znázorněny její další potomci, ale pouze jeden, *State (Stav)*. *Stav* je potomkem *Stavového vrcholu* a přebírá všechny jeho vlastnosti včetně možnosti vstupovat do interakce *Přechodu* mezi *Stavy*. Proto následující odstavec týkající se *Stavového vrcholu* platí i pro *Stavy*.

Mezi *Stavovým vrcholem* (a tedy i *Stavem*) a *Přechodem* existují dvě asociace, které vyjadřují jednoduchou skutečnost, že *Přechod* myšlený jako přechod mezi *Stavovými vrcholy* spojuje výchozí *Stavový vrchol* s následným cílovým *Stavovým vrcholem*. Z jedné strany má *Přechod* jeden *Stavový*

vrchol jako source (zdroj), z druhé strany má jeden *Stavový vrchol* jako target (cíl). Viděno ze strany *Stavového vrcholu (StateVertex)*, existují pro něj *Přechody* vycházející (outgoing) a vcházející (incoming).

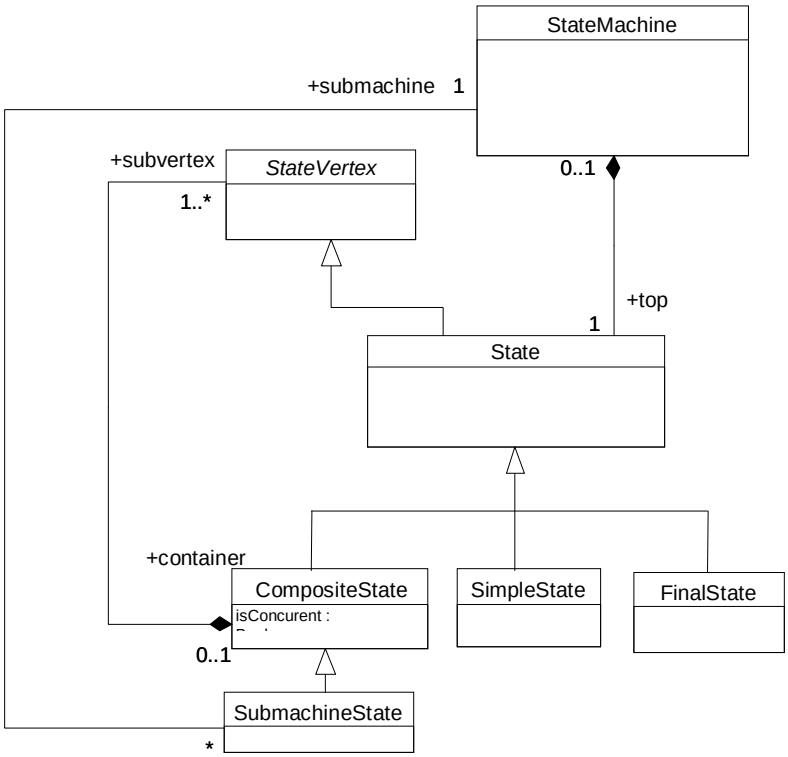
Poznámka: Všimněme si, že vztah mezi Přechodem a Stavem není zaveden na úrovni Stavu, ale na úrovni jeho předka Stavového vrcholu. Znamená to, že Stavový vrchol je ten, kdo nese tento vztah vůči Přechodům a dává proto tuto vlastnost všem svým potomkům (které nejsou na tomto diagramu znázorněny všechny). Jedná se o například o PseudoState (Pseudostav), reprezentující inicializační stav (počátek posloupnosti Přechodů v diagramu) apod.

Vztah *Přechodu (Transition)* mezi *Stavovými vrcholy* (a následně *Stavy*) je tedy směrový, protože rozlišuje *Stavový vrchol* - zdroj a *Stavový vrchol* - cíl. Pomocí tohoto mechanismu dvojnásobného vztahu (zdroj + cíl) versus *Přechod* se vyjadřuje posloupnost následných *Stavových vrcholů* (a *Stavů*) včetně popisu určitých vlastností *Přechodů* mezi nimi.

Navíc *Stav* může mít své vnitřní přechody (internaltransition), což jsou *Přechody*, které nemění daný *Stav*. Zdroj a cíl těchto přechodů jsou totožné *Stavy*.

3.10.2. State (Stav)

Strom generalizace pro *Stav* je vyjádřen na následujícím obrázku:



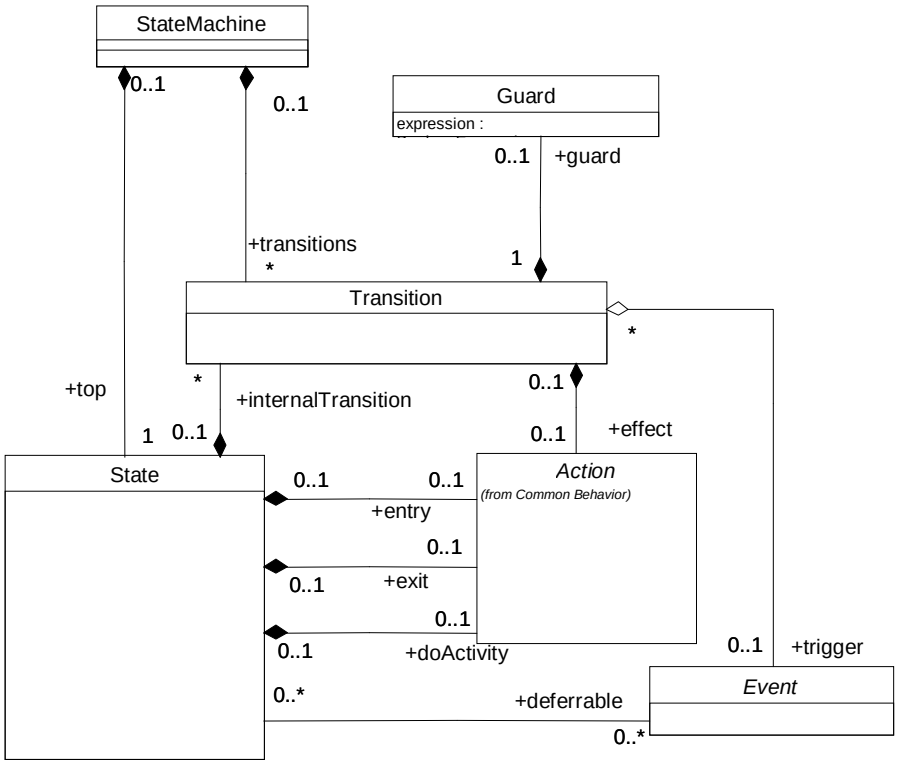
obrázek 32: Strom Generalizace pro Stav

Stav (State) je abstraktní metatřídou, do modelů vstupují až její potomci: *SimpleState (Jednoduchý stav)*, *FinalState (Konečný stav)* a *CompositeState (Kompozitní stav)*. *Stav* modeluje situaci, ve které se určité definované podmínky nemění. Může reprezentovat buď nějakou entitu „čekající“ na nějakou událost, anebo naopak může reprezentovat průběh nějaké akce, která svým v činnosti udržuje entitu v tomto stavu až do ukončení aktivity.

Příklad: Pračka ve Stavu „praní“ apod.

Stav dává svým potomkům určité vlastnosti. Jedny z nich jsou dány již tou skutečností, že *Stav* je potomkem *Stavového vrcholu* a proto vstupuje do *Přechodů*. Stejně tak všichni potomci metatřídy *Stavu* jsou také obdařeni tím, že se asociují s *Přechody.*

Další vlastnosti *Stavu* vyjádřeny následujícím diagramem:



obrázek 33: Vlastnosti Stavu

Stav (State) může obsahovat *Akci (Action)* maximálně jednu (nebo žádnou) označenou v asociaci jako entry (vstup). Je to nepovinná *Akce*, která se vždy spustí tehdy, když entita vstoupí do daného *Stavu* nezávisle na tom, odkud (z jakého jiného *Stavu*) se do tohoto *Stavu* dostala.

Podobně *Stav* může (a nemusí) obsahovat maximálně jednu *Akci* označenou jako exit. Je to *Akce*, která spustí vždy, když se tento *Stav* opouští a to nezávisle na tom, kam (do kterého *Stavu*) entita přechází.

Podobně *Stav* může (a nemusí) obsahovat maximálně jednu *Akci* označenou jako doActivity. Je to aktivita, která běží po dobu, kdy je daná entita v daném *Stavu*.

Stav může obsahovat několik *Přechodů*, které nemění daný *Stav*. Používá se tam, kde chceme zdůraznit, že „něco se změní“, ale *Stav* nikoliv.

Stav může mít 0 až N tzv. defferableEvents (pozdržené události). Jsou to *Události*, které jsou sice vyvolány, ale nejsou zpracovány a budou zpracovány až v nějakém jiném *Stavu*, kdy mají být zpracovány (v tomto jiném *Stavu* nebudou již pozdrženy).

3.10.3. Transition (Přechod)

Přechod je přímým potomkem *Prvku modelu (ModelElement)* a je chápán jako směrovaný vztah mezi *Stavovým vrcholem (StateVertex)* – zdrojem a *Stavovým vrcholem* cílem (viz diagram na obrázek 31: Zavedení Stavového stroje.

Přechod obsahuje maximálně jeden trigger (spouštěč), což je *Událost (Event)*, která vyvolává tento *Přechod*. (Zavedení metatřídy *Události* viz dále)

U *Přechodu* se může v modelu specifikovat nějaká aktivita vyjádřená *Akcí* nazvanou effect, kterou *Přechod* může a nemusí obsahovat (viz diagram).

3.10.4. Guard (Průvodce)

Přechod může mít svůj *Guard (Průvodce)* (viz obrázek 33: Vlastnosti Stavu), což je výraz typu Boolean, který se vyhodnocuje ve chvíli, kdy nastává *Událost* ve spouštěči (trigger). Pokud není *Průvodce* pravdivý, byť *Událost* nastala, k *Přechodu* nedojde.

Poznámka: Průvodce (Guard) je obdobou zvlášť zavedeného Omezení (Constraint). Nesmí se zaměňovat se samotnou událostí. Guard není podmínkou, která když nastane, tak se Přechod uskuteční. Je to podmínka, která se vyhodnocuje „těsně před" přechodem a pokud není splněna, Přechod nenastane (obdoba blokování přechodu).

Atributy Guard

- GuardExpression – výraz průvodce. Výraz typu Boolean, pokud je nepravdivý, k přechodu nedojde

3.10.5. SimpleState (Jednoduchý stav)

Protože *Stav (State)* je abstraktní metatřídou, do modelů nevstupují její instance, ale až její potomci (viz obrázek 32: Strom Generalizace pro Stav). Prvním potomkem je *Jednoduchý stav (SimpleState)*.

Jednoduchý stav je *Stavem*, který nemá žádné sub-stavy. (zavedení sub-stavu viz *Kompozitní stav*)

3.10.6. CompositeState (Kompozitní stav)

Kompozitní stav je stav, který obsahuje další *Stavové vrcholy (StateVertex)*. Uvedený vztah je vyjádřen na obrázek 32: Strom Generalizace pro Stav jako kompozice sub-vrcholů (subvertex) v *Kompozitním stavu (CompositeState)* jako v kontejneru (container). Tímto v *Kompozitním stavu* vznikají *Stavové vrcholy* chápané jako sub-vrcholy jiných stavů. Toto vložení *Stavových vrcholů* do jiného *Kompozitního stavu* má povahu kompozice a daný *Stavový (sub)vrchol* je „výlučným" majetkem daného *Kompozitního stavu* (nepatří již nikde jinam do jiného *Kompozitního stavu*).

Poznámka: Používání Kompozitního stavu je v úvahách velmi běžné. Například při analýze ekonomického sytému se prohlásí: Faktura se de facto vyskytuje ve dvou Stavech: faktura před odesláním a faktura po odeslání. Tyto dva stavy jsou ukázkou příkladu dvou Kompozitních stavů.

3.10.7. FinalState (Finální stav)

Metatřída *Finální stav* je potomkem metatřídy *Stav* a reprezentuje konečný finální stav.

Finální stav nesmí mít žádné vycházející (outgoing) *Přechody (Transition)*, viz diagram na obrázek 31: Zavedení Stavového stroje

3.10.8. SubMachineState (Stav sub-stroje)

Sub-stroj je pojem, který umožňuje získat další prvek flexibility a modularity. Je to určitá zkratka pro pojem obdobný extenzi pomocí makra. Významově je však stejný jako *Kompozitní stav*, kdy se jeden *Stav* „skládá z jiných stavů“. Z toho důvodu je *Stav* sub-stroje potomkem *Kompozitního stavu* a tedy obsahuje své vnitřní *Stavy*. V použití sub-stroje se má na mysli reference na jiný stroj. Vnější stroj tedy může „zavolat“ *Stav* sub-stroje obdobně jako nějakou proceduru obsahující pokračování dalších *Přechodů* uvnitř sub-stroje. Volání takovéhoto sub-stroje je možné z různých míst. Vstupní a výstupní body tohoto „volání“ jsou definovány pomocí *Stubstate (Stub stavu)*, viz dále.

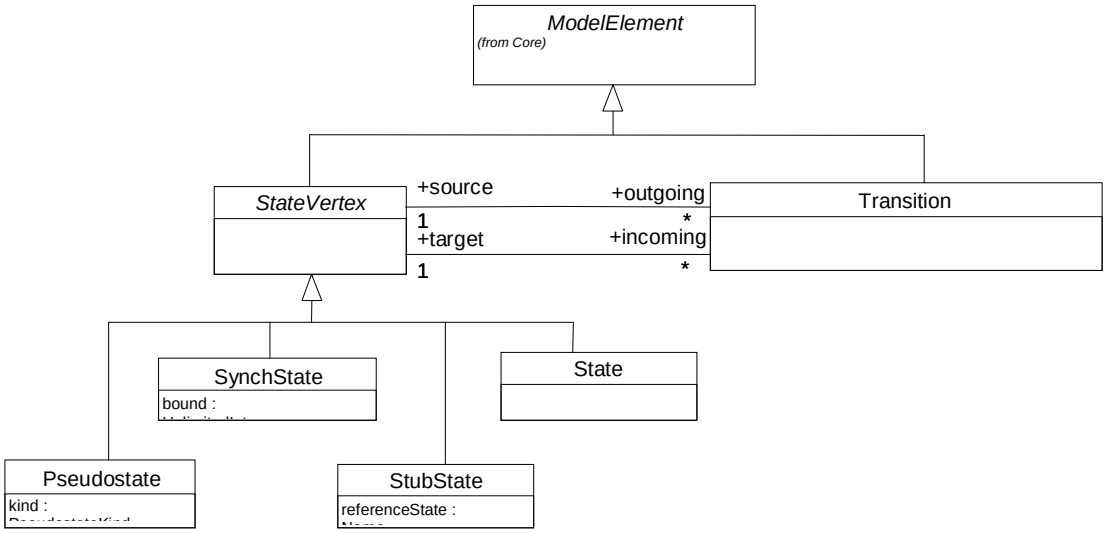
Poznámka: Stub nepřekládám jako terminus technicus, např. stub / proxy v COM apod.

3.10.9. StateVertex (Stavový vrchol)

Metatřída *Stavový vrcholy (Statevertex)* je přímým potomkem *Prvku modelu (ModelElement)* (viz obrázek 31: Zavedení Stavového stroje. Připomeňme, že *Stavový vrchol* je předkem *Stavu* a díky *Stavovému vrcholu* vstupuje *Stav* do vztahů přechodů mezi *Stavy (Stavovými vrcholy)*.

Stavový vrchol je sám o sobě abstraktní metatřídou a zavádí jako vrchol stromu další možné prvky stavového modelu (diagramu), které napomáhají lépe vyjadřovat chování *Stavového stroje*. Zjednodušeně řečeno, *Stavový vrchol* je „všechno“,co se nějak účastní Přechodů.

Základní diagram zavádějící potomky *Stavového vrcholu* ukazuje následující obrázek:



obrázek 34: Stavové vrcholy

Nejdůležitější z nich, *Stav (State)* byl vysvětlen v předešlých kapitolách, nyní následuje vysvětlení dalších prvků stromu.

3.10.10. Pseudostate (Pseudostav)

Pseudostav (Pseudostate) slouží k abstrakci vrcholů v grafu stavového stroje. Pomocí těchto pseudostavů se mnohem snáze a přehledněji vyjádří grafy složitých stavových strojů. *Pseudostavy* jsou různého druhu. Rozlišení typů *Pseudostavů* je pouze pomocí hodnoty atributu, nikoliv dalším stromem generalizace. V modelování se používají následující druhy *Pseudostavů*, které jsou hodnotami atributu *Kind (Druh) Pseudostavu*.

Atributy Pseudostate

- kind – druh, může nabývat těchto hodnot:

  - *initial* (*inicializační*) – pseudostav označuje vrchol, který je pouze zdrojem (sourcem) pro *Přechod* do implicitního *Stavu Kompozitního stavu* (default state). Pomocí inicializačního pseudostavu se dá „ukázat, odkud je třeba graf číst“. Daný *Kompozitní stav* může mít maximálně jeden implicitní stav.
  - *deepHistory* (*hluboká historie*) je vrchol reprezentující stav mezi sub-stavy *Kompozitního stavu*, který byl aktivní, když se *Kompozitní stav* opouštěl. Vytvořit *Přechod* do tohoto vrcholu pseudostavu je ekvivalentní požadavku návratu do stavu, který byl naposledy nastaven před opuštěním *Kompozitního stavu*.

Například pračka má stavy „Předpírka“ - „Praní“ - „Máchání“ - „Odstředění“, což je dohromady Kompozitní stav pračky „V akci“. Druhý stav „Čekání“ nastane, když se otevřou vrátka. Požadujeme,

aby se následným uzavřením vrátek pračka navrátila do stavu, který byl „nastaven“ v okamžiku otevření vrátek, tj. při opuštění Kompozitního stavu „V akci“. V modelu se zavede mezi sub-stavy „Akce“ další vrchol, pseudostav typu deephistory, a provede se Přechod do něj.

Kompozitní stav může mít nejvýše jeden *Pseudostav* typu deephistory.

- *shallowhistory* (*mělká historie*), reprezentuje vrchol, který je v daném okamžiku nastaven jako nejblíže poslední stav. *Přechod* do tohoto stavu znamená „přejdi do nejblíže posledního stavu“.

Pseudostavy hluboká a mělká historie (deephistory a shallowhistory) se chovají jako „stavová paměť“ (buffer) *Kompozitního stavu*, rozdíl mezi nimi je pouze v mechanismu jejich naplnění. Proto jak v případě deephistory, tak shallowhistory by měly být označeny default a inicializační stav, kam přejdou *Přechody*, pokud nebyly Kompozitní stavy dosud aktivní a tyto vrcholy – *Pseudostavy* se nemohly podobně jako paměť „naplnit“.

- *join* (*spojení*) je vrcholem – pseudostavem, který umožňuje spojit dva a více *Přechodů* do jednoho *Přechodu*. Znamená to, že tento vrchol má několik vstupů a jeden výstup a chápe se jako join spojení vstupních *Přechodů* do jednoho *Přechodu*. Logicky při joinu nesmí mít vstupující *Přechody Průvodce*.
- *fork (vidlička)* je opakem joinu. Jeden *Přechod* se pomocí vidličky rozdělí na několik *Přechodů*. Opět podobně vycházející *Přechody* nesmí mít *Průvodce*.
- *choice (výběr)* je vrcholem, ve kterém dochází dynamicky k přehodnocení *Průvodců* u *Přechodů* vycházejících ven z tohoto vrcholu. Na základě tohoto přehodnocení dojde k výběru *Přechodu*. Ve většině případů se volí disjunktní podmínky *Průvodců*. Pokud dojde u několika *Průvodců* najednou ke splnění podmínky pravda jejich *Průvodců*, je další *Přechod* libovolný. Pokud nastane situace, že ani jeden z *Průvodců* nebude pravda, potom se jedná o chybu v modelu.

3.10.11. StubState (Substav)

Dalším potomkem *State* je tzv. *Stubstate* (*Stub stav*). *Stub stav* slouží jako vstupní resp. výstupní stav stavového sub-stroje, slouží tedy jako propojení stroje, který „volá“ se strojem, který vykonává.

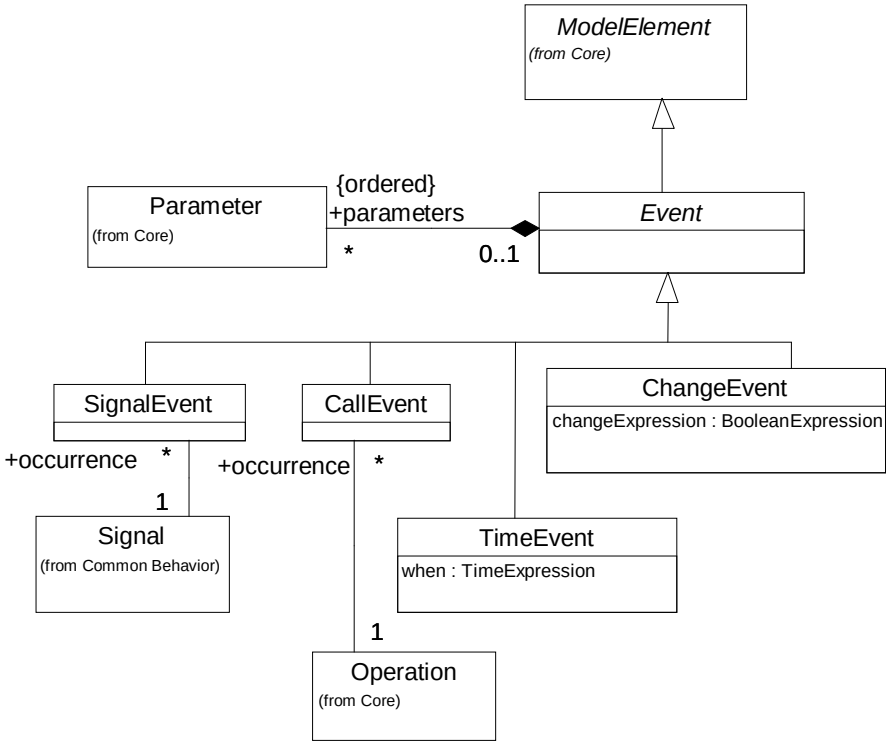
Poznámka: Pokud je Stub stav vstupním vrcholem sub-stroje, potom jeho vstupní (incoming) Přechody přicházejí zvně sub-stroje, pokud je výstupním vrcholem, potom jeho outcoming (výstupní) přechody směřují ven ze sub-stroje.

3.10.12. SynchState (Synchronizační stav)

Synchron-stav (SynchState) je stavovým vrcholem, který slouží pro synchronizaci různých částí stavového stroje, které jsou v konkurenci. Používá se spolu s *Pseudostavy* fork (vidličkou) resp. join (spojení) a svojí funkcionalitou zabezpečuje, že jedna část stroje opustí své stavy před tím, než vstoupí do druhé části stroje.

3.10.13. Event (Událost)

Události jsou v UML zavedeny podle následujícího diagramu *Events* (*Události*):



obrázek 35: Události

Událost (Event) je chápána jako něco, co nastane, nějaký výskyt, událost, „příhoda“ relevantní ve smyslu modelování zkoumaného systému.

Připomeňme, že *Události* vstupují do funkcionality Stavového stroje podle obrázku obrázek 33: Vlastnosti Stavu ve dvou pozicích:

- *Událost (Event)* plní funkci spouštěče (trigger) pro *Přechod (Transition)*. K Přechodu dojde za těchto podmínek:
  1. Stroj nachází ve *Stavu*, který odpovídá *Stavu* zdroj (source) daného *Přechodu (Transition)* (viz obrázek 34: Stavové vrcholy), tj. source *Stav* je aktivní,
  2. Nastane *Událost (Event)* spouštěč (trigger) daného *Přechodu,*
  3. Je splněna podmínka *Průvodce (Guard)* daného *Přechodu*.
- *Událost (Event)* se může vyvolat při samotném průběhu činnosti v nějakém *Stavu* jako tzv. pozdržená událost (viz vztah defferable na obrázek 34: Stavové vrcholy). V tom případě se *Událost* pouze řadí a nezpracovává. Jsou to *Události* vyvolané v daném *Stavu* připravené ke

zpracování „do foroty pro někdy příště“. V okamžiku, kdy se stroj dostane do takového *Stavu*, který nedrží tyto *Události* jako pozdržené, zpracují se.

Striktně vzato *Událost* by se měla chápat jako typ a nikoliv jako instance (tj. *Událost* by měla být typem výskytu). Samotná instance je to, co se konkrétně vyskytne v daném okamžiku. Oproti tomu *Událost* jako typ nese její vlastnosti (podobně jako vztah *Klasifikátoru* a jeho instancí) . Avšak v modelování se tyto dva pojmy nerozlišují a pro oba dva pojmy, jak pro typ, tak pro instanci, se používá pojem *Událost*.

V metamodelu je *Událost* potomkem *Prvku modelu*. Může obsahovat *Parametry* v kompozici (viz předešlý diagram).

Existují různé typy *Události*, které vyjadřuje předcházející strom generalizace.

3.10.14. CallEvent (Událost volání)

Událost volání (CallEvent) reprezentuje událost příjmu požadavku synchronního volání specifické *Operace*. Tato *Operace* je v metamodelu ve vztahu vůči dané *Události* (viz model obrázek 35: Události .

Poznámka: Událost volání není zpracování Události a díky tomu volání Operace, ale naopak, je to volání Operace, která způsobí vznik Události volání.

Dva významné a speciální případy *Události volání* jsou událost tvorby objektu a událost destrukce objektu.

V metamodelu je metatřída *Událost volání* je potomkem metatřídy *Události*.

3.10.15. SignalEvent (Událost signálu)

Událost signálu (SignalEvent) reprezentuje událost příjmu asynchronního signálu (pozn.: signál je vždy asynchronní).

Poznámka: Nesmíme zaměňovat vytvoření signálu s událostí příjmu signálu, to jsou dva odlišné pojmy. Signál se vytvoří (to je jeden pojem), někdo jej přijme a vyvolá se Událost signálu (to je druhý pojem).

V metamodelu je metatřída *Událost signálu* potomkem metatřídy *Události*. *Událost signálu* je ve vztahu asociace vůči *Signálu*, který je přijímán.

3.10.16. TimeEvent (Časová událost)

Časová událost (TimeEvent) modeluje uplynutí určité časové lhůty, tj. deadline. Z hlediska modelování se považuje čas příjmu (zpracování události) za stejný, jako čas vzniku této události. V některých případech je však třeba tyto dva časy (vznik, příjem) odlišovat z technologických důvodů (distribuce, frontování pod.).

Časová lhůta může být buď relativní anebo absolutní. Pokud je lhůta relativní a není specifikován čas zahájení, potom je chápána jako relativní vůči okamžiku vstupu do daného Stavu (zahájení odpočítání je od počátku stavu).

Atributy TimeEvent

- when (kdy) časový údaj lhůty (deadline)

3.10.17. ChangeEvent (Událost změny)

Událost změny je nejčastěji používanou událostí. *Událost* obsahuje výraz typu Boolean, který když se stane pravdivým (změna), vyvolá se tato *Událost*. Výraz obsahuje nějakou podmínku většinou týkající se hodnot atributů, naplnění vazeb apod.

Atributy ChangeEvent

changeExpression – výraz pro změnu. Výraz typu Boolean specifikující změnu.

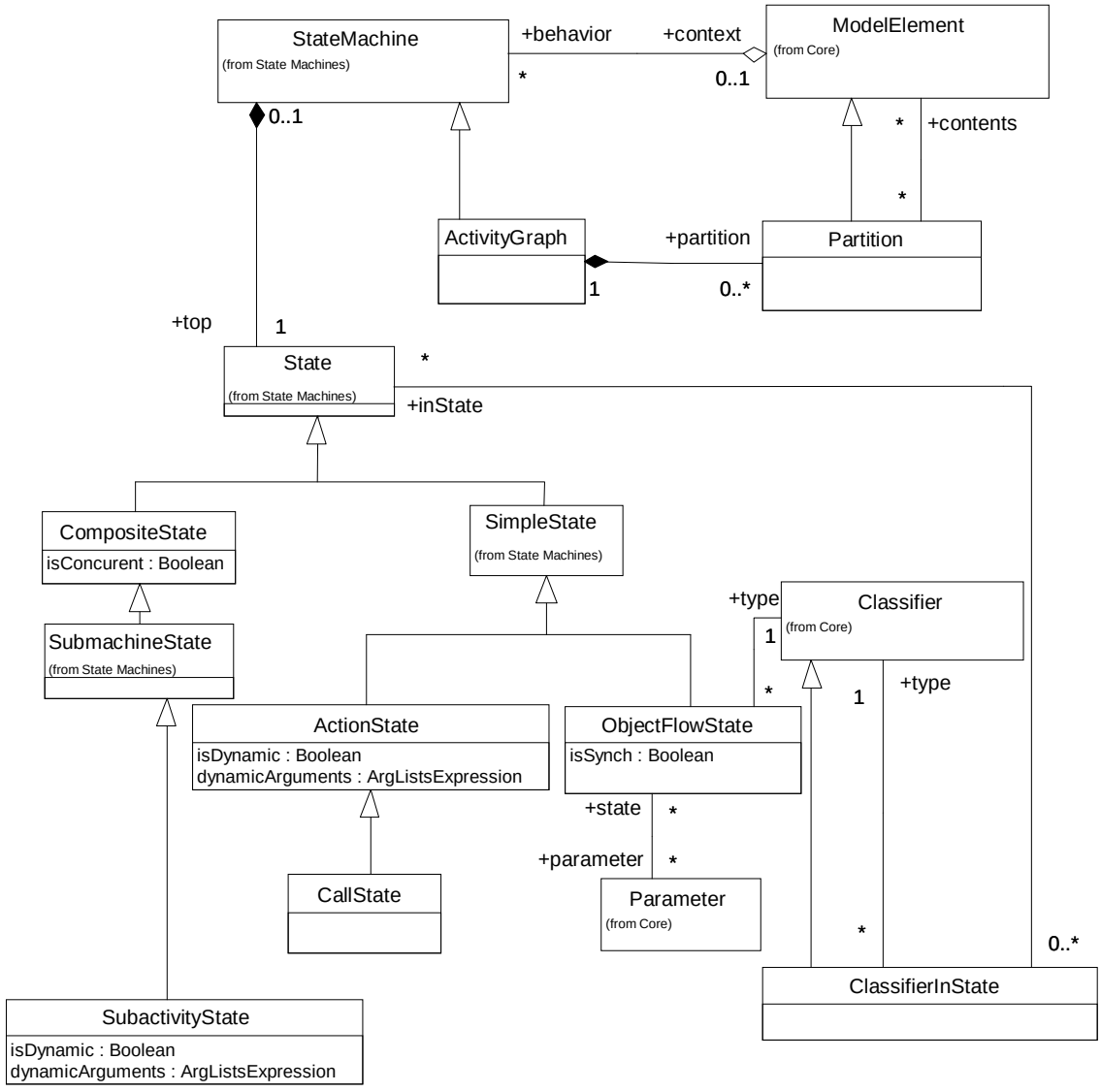
Poznámka: Nesmí se zaměňovat s Průvodcem (Guard), který má charakter splňující a tedy omezující podmínky u Přechodu.

3.11 Package Activity Graph (Svazek Grafy aktivit)

Svazek *Grafy aktivit* reprezentuje rozšíření svazku *Stavové stroje*, tj. prvky ve svazku *Grafy aktivit* slouží také k modelování stavově – přechodových systémů. Pro vyjádření modelů v tomto svazku používá mnoho prvků ze svazku *Stavové stroje.* Svazek *Grafy aktivit* je považován za extenzi *Stavových strojů*. Z hlediska konceptů modelování se jedná spíše o jiný pohled na vlastnosti *Stavových strojů*.

Poznámka: Pokud se vrátíme k příkladu s pračkou se svými stavy „předpírka“, „praní“ atd., tak bychom tyto stavy pračky mohli také chápat jako „aktivity pračky“, což jsou stavy pračky. Přesně takto se chápe Graf aktivit (tj. stavy jako aktivity).

Metamodel *Grafů aktivit* je zobrazen na následujícím obrázku:



obrázek 36: Grafy aktivit

Všimněme si, že v tomto diagramu existuje již několik prvků zavedených v předešlém svazku *Stavové stroje (State, SimpleState* atd.).

Svazek *Grafy aktivit* zavádí několik málo dalších nových pojmů, pomocí kterých lze modelovat systém.

Grafy aktivit se používají velmi často pro modelování podniku (business process modeling), protože stavy, které jsou v něm vyjádřeny, jsou stavy reprezentující probíhající aktivity. Podobně se velmi často *Grafy aktivit* používají v modelování technologických procesů (podnik je de facto také technologický systém).

Poznámka: Např. v technologii blíže nejmenované jaderné elektrárny mají stavy jako aktivity následující názvy: aktivita „spuštění reaktoru“, událost „vibrace turbíny“, aktivita „odstavení reaktoru“ a tak dokola.

3.11.1. ActivityGraph (Graf aktivit)

Metatřída *Graf aktivit (ActivityGraph)* je potomkem *Stavového stroje*, tedy jedná se pouze o jiný pohled na *Stavový stroj*. *Stavy* jsou v tomto případě chápány jako určité aktivity. Tento „nový“ pohled na stavový stroj „nového“ typu (jako pro potomka *Stavového stroje*) spočívá v tom, že existují také „noví“ potomci *Stavů*, které vyjadřují obdobu aktivit. Tito potomci jsou však takovými stavy, které reprezentují nějakou aktivitu, resp. jsou s aktivitami úzce spojeni. Díky tomu lze *Graf aktivit* považovat za model stavového stroje se stavy jako aktivitami.

Poznámka: Všimněme si předešlého diagramu. Zdánlivě to vypadá, že metatřída Graf aktivit nepřináší jako potomek nic nového, protože nemá ani jeden nový atribut a ani jednu novou vazbu. Jeho „novota“ se projeví tím, že je to on, kdo drží „nové“ potomky Stavů (viz popis dále), což jsou aktivity.

3.11.2. Partition (Oddělení)

Oddělení (Partition) zavádí mechanismus umožňující rozdělení modelů do skupin v souvislosti s *Grafem aktivit*. Používá se zejména v modelování podniku, kde *Oddělení* reprezentuje nějakou organizační jednotku (*Účtárna, Odbyt, Sklad* apod.). Jak je vidět na metamodelu (obrázek 36: Grafy aktivit), tak *Oddělení (Partition)* může mít vztah k obecnému (tj. libovolnému) *Prvku modelu* (*ModelElement*, viz vztah *Partition contents ModelElement*).

3.11.3. ActionState (Akce-stav)

Akce-stav (ActionState) je potomkem Jednoduchého stavu - *SimpleState* a vyjadřuje takový stav, který je chápán jako „jednoduchá aktivita“ atomického charakteru. Většinou se projeví v důsledku jako *Operace* apod. Přechod do další *Akce-stavu* reprezentuje ukončení této atomické *Akce-stavu*.

Atributy ActionState

- isDynamic (je dynamická). Pokud nabývá hodnoty pravda, potom daná *Akce-stav* může probíhat souběžně (concurrently). Má význam v souvislosti s hodnotami ostatních atributů
- dynamicMultiplicity (násobnost dynamiky). Udává násobnost limitující počet souběžně běžících aktivit. Má smysl pouze pro hodnotu atributu isDynamic pravda.
- dynamicArguments (argumenty dynamiky) je výraz udávající množinu seznamů objektů, každý seznam reprezentuje argumenty pro jedno paralelní zpracování.

3.11.4. ObjectFlowState (Objektový tok-stav)

Objektový tok-stav vyjadřuje tzv. objektový tok mezi dvěma aktivitami jako další možný *Stavový vrchol (Stav)*.

Určitá aktivita může přivést objekt (instanci *Klasifikátoru*) do určitého stavu a poté následuje další aktivita. V diagramu aktivit lze tuto skutečnost vyjádřit buď bez objektového toku jako přímý vztah dvou aktivit (přechod z jedné aktivity do druhé aktivity), anebo pomocí „vloženého“ dalšího *Stavového vrcholu* typu *ObjectFlowState,* který vyjadřuje připravenost nějaké instance *Klasifikátoru* v daném stavu.

Příklad:

Jeden pohled na určitou část modelu může být následující:

Aktivita „Vyplnění objednávky“ přechází do aktivity „Vyřízení objednávky“.

Druhý pohled, který používá ObjecFlowState, může být takový:

Aktivita „Vyplnění objednávky“ přechází do Objektového toku-stavu „Objednávka [vyplněna] “ a ten přechází do další aktivity „Vyřízení objednávky“.

V prvním pohledu byl vynechán Stavový vrchol typu ObjectFloswState s informací „Objednávka [vyplněna]“. Je třeba si uvědomit, že ObjectFlowState je vrcholem stavového stroje (dokonce níže ještě Stavem).

Objektový tok-stav vidí jeden *Klasifikátor* resp. *Klasifikátor ve stavu (ClassifierInState),* jehož připravenost (stav) instance reprezentuje. *Objektový tok-stav* může mít N parametrů.

Atributy ObjectFlowState

- IsSynch – (je synchronizační). Udává, zda je *ObjectFlowState* považován za synchronizační stav. Pokud ano, potom musí nastat synchronizace přechodů pro danou instanci Klasifikátoru.

Poznámka: Používání Objektového toku-stavu nedoporučuji provádět v raných fázích analýzy, protože se v této fázi obtížně a mnohdy s chybami vyhledávají instance Klasifikátorů ve stavech.

3.11.5. SubactivityState (Subaktivita-stav)

Metatřída *SubactivityState (Subaktivita-stav)* je potomkem metatřídy *SubmachineState* (viz blíže kapitola SubMachineState (Stav sub-stroje) a její prvky slouží v modelování k témuž účelu, tj. vnořování (volání) stavových strojů mezi sebou.

Subaktivita-stav reprezentuje vykonání ne-atomické aktivity složené z jiných aktivit. Jedná se tedy o *SubMachineState* (*Stav sub-stroje*), který vykoná vnořený graf aktivit.

Poznámka: Jednoduše řečeno, mechanismus Subaktivit umožňuje grafy aktivity skládat přirozeným způsobem stejně jako stavové stroje, jehož jsou potomky

Subaktivita stav má stejné atributy, jako *Aktivita-stav*.

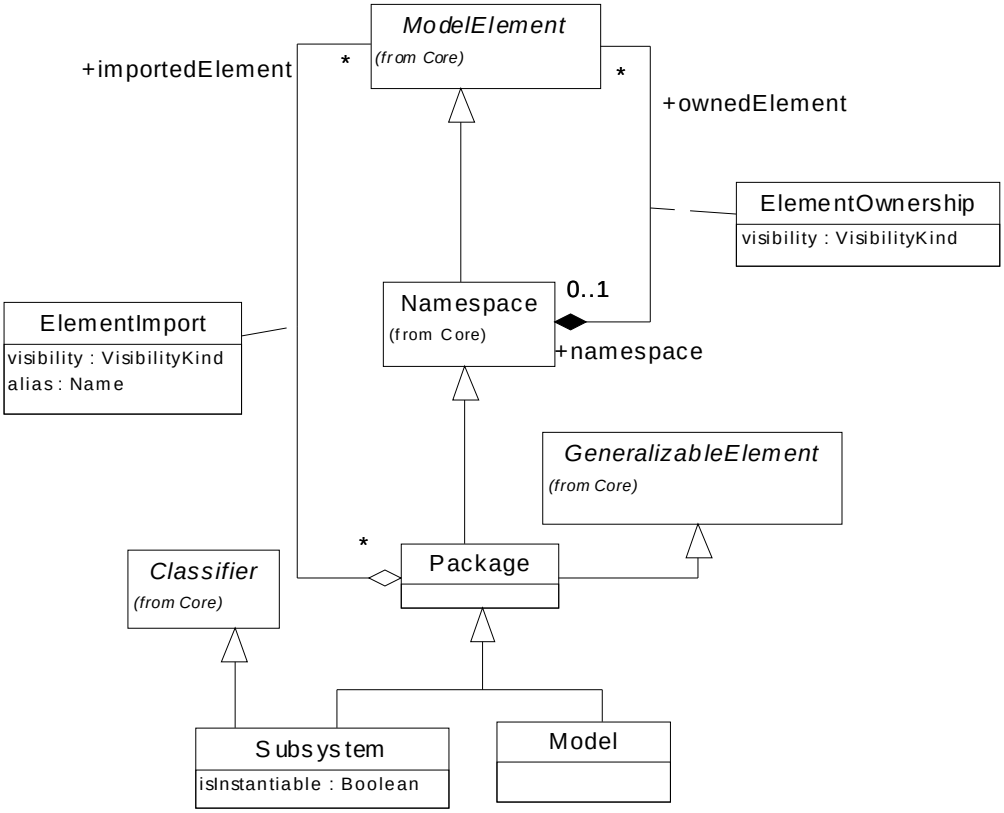
3.11.6. ClassifierInState (Klasifikátor ve stavu)

Klasifikátor ve stavu (ClassifierInState) reprezentuje instanci *Klasifikátoru* s nastavenými hodnotami takovými, které odpovídají stavu v daném *ObjectFlowState* (tj. mezi dvěma relevantními aktivitami). Umožňuje přesněji modelovat vlastnosti *Klasifikátoru* díky zavedení stavu *ObjectFlowState*.

3.12 Package Managment model (Svazek Řízení modelu)

Jak sám název napovídá, svazek *Řízení modelu* obsahuje obecné prvky napomáhající řízení modelu. Tento svazek je závislý na svazku *Základ* (package *Foundation*).

Obrázek diagramu, který zavádí prvky svazku *Řízení modelu*, je následující:



obrázek 37: Svazek Řízení modelu

Zavádí základní tři nové pojmy:

- *Package (Svazek)*
- *Model (Model)*
- *Subsystem (Subsystém)*

3.12.1. Package (Svazek)

Poznámka: vzhledem k nutnosti používat Svazek v metamodelu (viz například dělení kapitol v této e-knize), byl tento prvek již částečně pro tyto potřeby vysvětlen v předešlých kapitolách.

Package (Svazek) je skupinou elementů modelu a slouží k rozumnému rozdělení modelu.

Jako příklad takovéhoto rozdělení do Svazků viz například metamodel UML, který je zde právě popisován.

V metamodelu je metatřída *Package* potomkem *Pojmenovaného prostoru (Namespace)*, díky čemuž může obsahovat *Prvky modelu* jako kompozici ownedElement (vlastněné prvky). Znamená to, že tento vztah vlastnění prvku je unikátní (žádný jiný *Svazek* nemůže vlastnit jako ownedElements daný prvek).

Kromě toho může obsahovat také importované prvky z jiných *Svazků* (které jsou vlastněné jako ownedElements v jiném *Svazku*). Asociativní metatřída *ElementImport* (importovaný element) dává těmto prvkům další dvě vlastnosti, kterými jsou visibility (viditelnost) a alias. Význam visibility je stejný, jako již byl uveden v předešlých kapitolách u jiných prvků. Alias je sekundární jméno, které může importovaný prvek dostat v importovaném *Svazku*.

Poznámka: Připomeňme, že kromě takového klasického importu může být prvek z jednoho Svazku přístupný do druhého Svazku pomocí Dependency, speciálně ***Chyba! Nenalezen zdroj odkazů.****.*

Svazek může vstupovat do vztahů *Generalizace*, protože je potomkem metatřídy *Zobecnitelný prvek (GeneraziableElement)*.

Je pochopitelné, že *Svazek* může obsahovat další *Svazky*.

3.12.2. Model (Model)

Model je abstrakcí fyzického systému, která vznikla za nějakým účelem.

Poznámka: Je důležité vysvětlit pojem fyzický systém. V UML je tím míněn „opravdu reálně“ existující systém. Například při stavbě domu existuje „reálně stojící dům“ (fyzický systém) a existují jeho plány a náčrty (modely).

Metatřída *Model* je potomkem *Svazku*, což znamená, že vytváří hierarchickou strukturu *Prvků modelu* stejně jako Svazek, ale vyjadřuje obraz fyzického systému.

Pro jeden fyzický systém může existovat několik (konzistentních) *Modelů* podle různých pohledů (*Model Případů užití*, *Logický model*, apod.). Každý z těchto *Modelů* je vnitřně konzistentní a úplný.

Poznámka: Je pochopitelné, že úplnost modelu je mnohdy pouze teoretická a technicky neproveditelná.

Model může jako *Svazek* obsahovat další *Modely*.

3.12.3. Subsystem (Subsystém)

Subsystém je skupinou elementů (potomek *Svazku*), která reprezentuje složku fyzického systému (unit) s nějakým specifickým chováním.

Subsystém nabízí *Interfacy* a má *Operace* díky dědění ze strany *Klasifikátoru*. Navíc *Subsystém* má prvky rozděleny do dvou skupin, tzv. prvky specifikace a prvky realizace. Prvky specifikace jsou realizovány (implementovány) pomocí prvků realizace.

Poznámka: Subsystém v UML odpovídá přesně již dříve používanému pojmu subsystém. Svazek na rozdíl od pojmu Subsystém je obecnější, nemá implementační povahu jako Subsystém, a slouží k libovolnému rozčlenění Prvků modelu v libovolné úrovni abstrakce. Subsystém člení systém na složky (unity) se svým chováním (tj. Subsystém jsou části systému s interfacy a operacemi).

Atributy Subsystem

- isInstatiable – pokud je pravda, *Subsystém* může mít instance.

4. Překlad pojmů UML z angličtiny (zavedený v této knize)

V následující tabulce jsou uvedeny autorem zvolené překlady základních pojmů metamodelu UML.

V žádném případě nelze tyto překlady považovat za určitě stoprocentně „ty pravé“, snahou bylo vytvořit dvojice anglických a českých ekvivalentů. Je možné, že nějaký jazykozpytec najde pro pravou stranu tohoto přiřazení lepší český pojem. V tom případě dojde pouze k výměně tohoto překladu.

Abstraction	Abstrakce
Action	Akce
ActionSequence	Sekvence Akcí
ActionState	Akce-stav
ActivityGraph	Graf aktivit
Actor	Aktér
Argument	Argument
Association	Asociace
AssociationClass	Asociativní třída
AssociationEnd	Konec Asociace
AssociationEndRole	Role Konce Asociace
AssociationRole	Role Asociace
Attribute	Atribut
AttributeLink	Spojení atributu
BehavioralFeature	Rys chování
Binding	Vazba
CallAction	Akce volání
CallEvent	Událost volání
Class	Třída
Classifier	Klasifikátor
ClassifierInState	Klasifikátor ve stavu
ClassifierRole	Role Klasifikátoru
Collaboration	Spolupráce
Comment	Komentář
Component	Komponenta

ComponentInstance	Instance komponenty
CompositeState	Kompozitní stav
Constraint	Omezení
CreateAction	Akce tvorby
Data type	Datový typ
DataValue	Datová hodnota
Dependency	Závislost
DestroyAction	Akce zrušení
Element	Prvek
ElementResidence	Umístění prvku
Event	Událost
Exception	Výjimka
Extend	Rozšíření
ExtensionPoint	Bod rozšíření
Feature	Rys (ve smyslu „povaha, črt“)
FinalState	Finální stav
Flow	Tok
Generalization	Generalizace
Guard	Průvodce
ChangeEvent	Událost změny
Include	Začlenění
Instance	Instance
Interaction	Interakce
Interface	Interface
Link	Spojení Instancí
LinkEnd	Konec Spojení Instancí
LinkObject	Spojení Instancí - Objekt
Message	Zpráva
Method	Metoda
Model	Model
ModelElement	Prvek modelu
Namespace	Pojmenovaný prostor
Node	Uzel
NodeInstance	Instance Uzlu
Object	Objekt

ObjectFlowState	Objektový tok-stav
Operation	Operace
Package	Svazek
Package Activity Graph	Svazek Grafy aktivit
Package Auxiliary elements	Svazek Prvky příslušenství
Package Behavioral Elements	Svazek Prvky chování
Package Common Behavior	Svazek Společné chování
Package Core	Svazek Jádro
Package Data Types	Svazek Datové typy
Package Extension Mechanisms	Svazek Mechanismy extenze
Package Foundation	Svazek Základ
Package Managment model	Svazek Řízení modelu
Package State Machines	Svazek Stavové stroje
Package Use Cases	Svazek Případy užití
Parameter	Parametr
Partition	Oddělení
Permission	Povolení
PresentationElement	Prezentační prvek
Pseudostate	Pseudostav
Qualifier	Kvalifikátor
Reception	Příjem
Relationship	Relace
ReturnAction	Akce návratová
SendAction	Akce zaslání
Signal	Signál
SignalEvent	Událost signálu
SimpleState	Jednoduchý stav
State	Stav
State Machine	Stavový stroj
StateVertex	Stavový vrchol
Stereotype	Stereotyp
Stimulus	Stimul
StructuralFeature	Rys struktury
StubState	Stub stav
SubactivityState	Subaktivita-stav

SubMachineState	Stav sub-stroje
Subsystem	Subsystém
SynchState	Synchronizační stav
Tagged Value	Tagovaná hodnota
Template	Šablona
TerminateAction	Akce ukončení
TimeEvent	Časová událost
Transition	Přechod
UninterpretedAction	Akce nespecifikovaná
Usage	Užití
UseCase	Případ užití
UseCaseInstance	Instance Případu užití

KONEC DOKUMENTU