Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Rozšíření možností reprezentace komponentových aplikací

Plzeň, 2013 Petr Mošna

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 5. května 2013 ….....

Petr Mošna

Abstract

Component applications vizualization features improving

This thesis deals with an interactive visualization of component based applications and the existing tool for interactive visualization ComAV. The problem of the visualization of large component based application are generality of displayed informations or displaying large amount of information. The first case leads to the shallow understanding of the application and the second case leads to the poor orinentation of a developer in the displayed component graph. These problems are solved by using interactive visualization, in which some information remains hidden and could by displayed on demand. This approach of componen application visualization is implemented in the tool ComAV, or more precisely in its plug-in AIVA, that visualizes loaded information about components. ComAV is build upon the Eclipse RCP Platform and the AIVA plug-in technologically uses the JGraphX library for visualization. ENT meta-model keeps information about the components and it represents the data layer for the AIVA plug-in. This work improves the features for vizualization of component applications using the AIVA plug-in.

Obsah

Seznam obrázků

1 Úvod

Vývoj softwarových aplikací vždy doprovázely situace, kdy byly některé části kódu podobné nebo úplně stejné jako v již existujících aplikacích. Tedy částmi kódu, které by šlo, třeba i s menšími úpravami, znovu použít. Tyto části kódu se začaly znovu používat a poté také sjednocovat do takzvaných komponent, někdy nazývaných moduly. To jsou zjednodušeně logicky sdružené funkční celky kódu nebo zdrojů aplikací.

Aplikace by pak mohla být složena pouze z komponent a kontejneru starajícího se o základní služby, obsluhu životního cyklu komponent a jejich vzájemnou komunikaci. Aby se tyto nastíněná myšlenka standardizovala, vznikly specifikace definující požadavky na komponenty, jejich životní cyklus, meta-model popisující jejich vlastnosti, atd.

Problémem při vývoji komponentových aplikací, je způsob zobrazení komponent nebo závislostí mezi nimi. Základním prostředkem pro zobrazení je jednoduchý „boxes-and-arrows“ model. Ten je však příliš obecný a nedokáže zobrazit všechny potřebné vlastnosti. Druhou možností je popsat aplikaci specifickým komponentovým modelem. Takový popis obsáhne všechny aspekty aplikace, ale nelze jej použít v aplikacích jiných komponentových modelů. Z tohoto důvodu byl Doc. Bradou navržen obecný komponentový meta-model ENT [EMM04].

Pro zobrazení komponent byl na Katedře informatiky na Západočeské univerzitě v Plzni vytvořen software Component Application Visualization Tool (ComAV), používající výše zmíněný ENT meta-model. Program umožňuje načítat komponentové aplikace různých komponentových modelů. Informace o komponentách jsou programem uloženy pomocí prostředků implementovaného ENT meta-modelu. Program samotný je také vytvořen jako komponentová aplikace. Modul starající se o interaktivní zobrazení načtených komponentových aplikací v programu ComAV se nazývá Advanced Interactive Visualization Approach (AIVA). To, že je zobrazení komponent interaktivní, znamená, že se zobrazenými informacemi může uživatel dále pracovat nebo je různě upravovat.

Úkolem diplomové práce bylo vhodně rozšířit právě modul AIVA. Ze zadání, konzultace se zadavatelem práce a vlastního uvážení, vyplynula tato rozšíření:

- alternativní vzhled komponent a jeho přepínání
- zvýraznění elementů (rozhraní komponent) po kliknutí na spojení mezi komponentami
- zvýraznění použitých elementů po kliknutí na element
- alternativní vzhled spojení mezi komponentami a jeho přepínání
- definování pravidel filtrování a jejich použití v grafu
- podpora hierarchických (tzn. komponenta složená z dalších komponent) komponentových modelů
- změna barvy v různých použitých vyznačeních

Cílem textu diplomové práce je přiblížit čtenáři komponenty, interaktivní zobrazení komponentových aplikací a stručně popsat ENT meta-model. Poté následuje stručný popis vývojové platformy Eclipse RCP, pomocí níž je vytvořena aplikace ComAV. Dále je popsána knihovna JGraphX, použitá v modulu AIVA pro interaktivní zobrazení grafů komponent. Následuje stručný popis aplikace ComAV, jehož součástí je modul AIVA. Důležitou částí textu je poté popis implementovaných rozšíření modulu AIVA, která jsou uvedena výše. Díky nim se zlepšila práce s aplikací ComAV, respektive modulem AIVA.

2 Obecně o komponentách

Kapitola pojednává o komponentách a vysvětluje čtenáři základní pojmy, z oblasti vývoje komponentových aplikací.

2.1 Úvod do komponentových technologií

Jak bylo nastíněno v úvodu, jeden ze směrů vývoje softwaru je skládat jej z komponent. Myšlenka vývoje aplikací z již vytvořených komponent byla poprvé představena již v roce 1968 [MPSC68]. Jde tedy o poměrně starý princip, a přesto je živý dodnes. Oblast softwarového inženýrství, jež se zabývá vývojem komponentových aplikací, se nazývá Component-based Software Engineering (CBSE) nebo i též Component-based Development (CBD). CBSE poskytuje metodiky a nástroje pro práci s komponentami a systémy, které jsou z nich složeny. CSBE je vhodný pro rozsáhlé aplikace, pro aplikace, jejichž části (komponenty) se často mění, nebo pro aplikace, kde lze použít některé již existující zdroje.

Vývoj aplikací pomocí komponent přináší výhody jako jsou šetření času, práce a usnadnění údržby. Šetření času a práce vyplývá z předpokladu, že ve vytvářené aplikaci půjde použít již existující komponentu (komponenty). Co se týče údržby, je zřejmé, že změna (odstranění chyby, vylepšení funkčnosti) komponenty se provede pouze jednou a nemusí se provádět ve všech aplikacích, kde by se případná vylepšovaná část vyskytovala. Nevýhodou vývoje komponentových aplikací může naopak být chápána závislost aplikace na komponentách třetích stran, jež mohou obsahovat různé chyby nebo mají jiné nedostatky.

2.2 Komponenta

Pojem komponenta je v softwarovém inženýrství dobře známý, přesto je vhodné uvést definici toho, co představuje. Přestože neexistuje žádná jednotná definice, je pojem komponenta dobře vymezen. Nejuznávanější definice uvedená v [CS02] říká, že

softwarová komponenta je softwarový balík nebo modul zapouzdřující funkce nebo data. Aplikační logika celého systému je rozdělena do jednotlivých komponent a to tak, že každá komponenta řeší jeden logický problém. O komponentách se také někdy hovoří jako o modulech a místo komponentového vývoje se hovoří o modulárním vývoji.

Každá komponenta tedy poskytuje nějakou službu nebo data. Komunikace, nebo-li využívání služeb komponenty, se provádí přes definovaná rozhraní komponenty. To, jak jsou služby implementovány, je často skryto a hovoří se zde o takzvaném zapouzdření komponent.

Komponenty jsou taktéž nahraditelné, což znamená, že je lze ve fungujícím systému měnit za jiné, poskytující podobné služby. Hlavní však je, aby zůstala zachována komunikační rozhraní. Nahraditelnosti komponent se používá například při aktualizaci komponent jejich novějšími verzemi.

2.3 Komponentový model

Nad komponentami stojí takzvaný komponentový model, jež je složen z definice povolených typů komponent, jejich životních cyklů, rozhraní a dalších vlastností, a také ze specifikace povolených vzorů interakce mezi komponentami. Komponentový model tedy definuje pravidla pro komponenty z důvodu, aby vůbec komponentová aplikace mohla běžet.

Komponentových modelů existuje celá řada, například OSGi, SOFA 2, EJB, Spring, CoSi nebo CORBA.

2.4 Komponentový framework

Komponentový framework je implementací komponentového modelu. Poskytuje tedy komponentám prostředí, v němž mohou být nasazeny a spouštěny. Stará se o řízení komponent a jejich životního cyklu, prostředků sdílených mezi komponentami a komunikaci mezi komponentami.

2.5 Blackbox

V oblasti vývoje komponentových aplikací lze narazit na pojem blackbox. Obecně jde o zařízení, o kterém nevíme, jak vnitřně pracuje, a přesto je používáno definovaným způsobem. Analogií v CBSE je komponenta, jejíž vnitřní implementace je skryta a jsou popsána pouze rozhraní (vyžadovaná a poskytovaná), díky kterým lze s komponentou pracovat.

Skrytí vnitřku komponenty přináší snadnější použití (v ideálních případech) komponenty. Opakem pojmu blackbox je whitebox, což v CSBE znamená poskytnutí celé vnitřní implementace komponenty. Díky tomu lze komponentu studovat a lépe pochopit její funkčnost. Speciálním případem whiteboxu je pak glassbox, kdy je umožněno upravování vnitřní implementaci komponenty.

3 Interaktivní zobrazení komponentových aplikací

Následující kapitola se nejprve popisuje obecné zobrazení a interaktivní zobrazení. Poté se text práce postupně přesouvá na popis interaktivního zobrazení komponentových aplikací.

3.1 Zobrazení obecně

Vizualizace (zobrazení) je disciplína počítačové technologie jež transformuje informace do vizuální podoby. Díky tomu lze s informacemi snadněji pracovat a lze jim lépe porozumět. Existují dvě hlavní oblasti vizualizace:

- **vědecká -** zobrazující fyzická data. Používá se především ve fyzice, medicíně, biologii, atd. Příklady jsou např. rentgeny lidí, obrázek sluneční soustavy, zobrazení chemické reakce.
- **informační -** zobrazující abstraktní data, jejíž použití je především v oblasti softwarového inženýrství. Jde o různá zobrazení softwaru, jelikož programy a algoritmy nemají fyzickou formu.

3.1.1 Zobrazení softwaru

Zobrazení softwaru je tedy jednou z oblastí obecné vizualizace poskytující jiný pohled na aplikace nebo různá data. Přesná definice vizualizace softwaru je uvedena v článku [PC95]:

> Vizualizace softwaru je disciplína využívající různé formy zobrazení, díky nimž poskytuje porozumění a bližší pohled, a snižuje složitost existujícího softwaru.

Zobrazení softwaru se navíc dále dělí do různých skupin, podle toho pro jaké případy je použitou. Podle [SV07] jej lze rozdělit do následujících skupin:

- **Strukturální vizualizace** pro zobrazení vnitřních statických vlastností softwaru.

Jde o vlastnosti, které lze zjistit bez spuštění softwaru, jelikož jsou založeny pouze na vnitřní implementaci. Strukturální vizualizaci lze použít především pro modelování architektury softwaru, popsání tříd nebo zobrazen vnitřních algoritmů aplikace.

- **Vizualizace chování** pro zobrazení dynamických vlastností softwaru, což jsou například volání funkcí, použití paměti nebo čas běhu aplikace. Oproti strukturální vizualizaci lze tyto vlastnosti softwaru získat až po jeho spuštění a analýze jeho chování. Vizualizaci lze použít především pro vyhledání částí softwaru, které jsou náročné na zdroje (paměť, čas) nebo pro zobrazení sekvenčního volání funkcí.
- **Evoluční vizualizace** pro zobrazení statických a dynamických vlastností. Jde tedy o kombinaci předchozích oblastí, ale s tím rozdílem, že se zaměřuje na změnu vlastností v čase. Využití například pro zobrazení změn zdrojových kódů nebo jiných zdrojů aplikace.

3.2 HCI

Interakce člověka (uživatele) s počítačem (HCI) spočívá v tom, že se uživateli poskytnou takové prostředky, kterými může měnit a interpretovat zobrazená data. HCI pokrývá několik vědeckých oblastí. Mezi ně patří například lidská psychologie (se zaměřením na paměť a vnímání), počítačová věda a další.

Základním cílem HCI je zlepšení interakce úpravou počítačů (programů) tak, aby byly lépe použitelné z pohledu uživatele. Toho se snaží dosáhnout úpravou programů, respektive jejich ovládání, aby byly snadno použitelné, efektivní, bezpečné a intuitivní pro nové uživatele.

Z pohledu uživatele aplikací je tedy výslednou činností HCI návrh jejich ovládacích rozhraní, ať už webových nebo GUI pro desktopové aplikace.

3.3 Interaktivní zobrazení

Interaktivní zobrazení slouží ke změně zobrazených informací a případně i jejich formě. Největší použití má v komplexních informačních systémech, případně pro velké objemy dat, avšak své opodstatnění najde v téměř všech případech zobrazení dat. Například navigaci nebo základním pohyb v zobrazených datech lze najít téměř všude. V článku [RIIV07] zabývajícím se interakcí uživatele se zobrazenými informacemi, jsou různé interaktivní techniky popsány formou logicky sdružených kategorií. Podle článku existují následující kategorie interaktivního zobrazení:

- **Select** – výběr/vyznačení relevantních informací. Pokud je zobrazeno příliš mnoho informací nebo se reprezentace zobrazených informací mění, může pro uživatele být těžké sledovat zobrazenou oblast, která ho zajímá. Vizuálním vyznačením (vybráním) těchto částí je může snadněji sledovat v obou popsaných problematických případech.

  Příkladem vyznačení relevantních informací je umístění uživatelského bodu do mapy.

- **Explore** – změna zobrazených informací. Pro většinu případů vizuálního zobrazení dat platí, že uživatel nevidí celou sadu dat (např. textový dokument, grafický editor, přiblížení v mapě, aj.). Důvodem je rozsah zobrazovaných dat, velikost monitoru/obrazovky, ale i kognitivní a vjemové limitace lidského mozku při zpracování informací. Uživatelé tedy nejčastěji pracují s podmnožinou dat, díky čemuž jim mohou lépe porozumět a celkově se v nich lépe orientovat, a následně mohou změnit zobrazenou podmnožinu dat. Interakce z této kategorie však nutně nemusí měnit celou zobrazovanou podmnožinu dat.

  Příkladem změny zobrazených informací je pohyb v mapě.

- **Reconfigure** – změna uspořádání zobrazených dat. Zobrazeným datům je změněno prostorové uspořádání, díky čemuž uživatel získává odlišný pohled na data. Jedním ze základních účelu interaktivního zobrazení je totiž zobrazení skrytých informací a vztahů mezi zobrazenými daty. Dobrá statická reprezentace tomu může vyhovovat, avšak pouze jedno samotné zobrazení málokdy

poskytuje dostačující náhled nad daty. Z toho důvodu je uživateli v aplikacích zobrazujících data velmi často umožněno změnit uspořádání nebo zarovnání dat.

Příkladem změny uspořádání zobrazených dat je řazení dat v tabulkách podle požadovaného sloupečku.

- **Encode** – změna vzhledu zobrazených dat. Uživatel může měnit vizuální reprezentaci zobrazených dat, včetně změny různých vizuálních prvků jednotlivých datových elementů. Příkladem těchto vizuálních prvků jsou změny barvy, velikosti, tvaru, aj. Pro interaktivní zobrazovací systémy plní datové elementy důležitou roli. Nejen z důvodu, že mohou ovlivnit kognitivní vnímání, ale také protože přímo ovlivňují, jak uživatelé porozumí vztahům mezi zobrazenými daty. Příkladem tohoto vztahu je přenos informace o výšce do mapy barvou. Oblasti s vyšší nadmořskou výškou mají zpravidla tmavší barvu než níže položené oblasti. Tím je dodána do mapy informace aniž by bylo zasahováno do jejího prostorového uspořádání nebo přímo obsahu.

  Příkladem změny vzhledu zobrazených dat je zobrazení různých výsledků histogramem místo grafovým koláčem.

- **Abstract/Elaborate** – skrytí nebo zobrazení dalších informací o datech. Uživateli je umožněno řídit úroveň abstrakce zobrazených dat. Rozsah zobrazení je tedy od stručného souhrnu až po detaily jednotlivých datových elementů, často také společně s mnoha úrovněmi mezi. Uživatel se nejčastěji pohybuje na vyšších úrovních zobrazení a zobrazuje si více informací (přibližuje se) o hledaných datech. Opakem je pak samozřejmě postup z detailního pohledu na data do pohledu se souhrnem.

  Příkladem zobrazení skrytých dat je přiblížení v mapě na hledané město, kdy jsou postupně zobrazovány jeho čtvrti, ulice, popisná čísla, atd.

- **Filter** – filtrování zobrazených informací. Interakce tedy uživateli umožňuje změnit zobrazená data na základě nějakých specifických pravidel. Pro tento druh interakce uživatelé definují rozsah nebo podmínky takovým způsobem, aby mu vyhovovala hledaná podmnožina dat. Ostatní část dat není zobrazena nebo je zobrazena odlišným způsobem. Samotná data nejsou žádným způsobem

ovlivněna a mohou být od deaktivaci pravidel znovu zobrazena.

Příkladem filtrování zobrazených informací jsou například našeptávače pro hledání v mapách.

- **Connect** – zvýraznění vztahů v zobrazených informacích, případně zobrazení skrytých částí informací, jež jsou relevantní pro vybraná data. V zobrazených datech (typu graf, kresba, aj.) může být obtížné orientovat se v jejich vztazích. Problém je odstraněn právě pomocí interaktivního zvýraznění. Zobrazení skrytých relevantních dat je pouze jiným typem této kategorie.

  Příkladem zvýraznění vztahů v informacích je kliknutí na libovolný uzel grafu a následné zvýraznění jeho sousedních uzlů. Příkladem druhého typu této kategorie je například kliknutí na slovo v překladači a následné zobrazení možných výrazů pro přeložení.

3.4 Zobrazení komponent

Různé komponentové modely popisují komponenty jinak a proto je jejich zobrazení problémem. Jak bylo zmíněno v úvodu, první možností pro zobrazení komponentových aplikací je používat obecný „boxes-and-arrows“ diagram. Ten lze aplikovat na všechny komponentové modely, avšak kvůli své obecnosti ztrácí schopnost popsat všechny vlastnosti komponent a nabízí pouze povrchní popis aplikace.

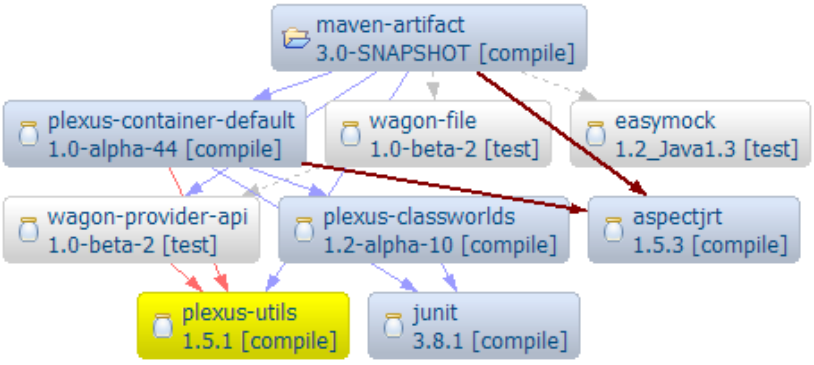
Druhou možností jak zobrazit vnitřní strukturu komponentové aplikace, je popsat ji specifickým komponentovým modelem. Ty zobrazují detailnější informace o komponentových aplikacích, jelikož byly navrženy přímo pro svůj komponentový model. Hlavní výhodou je tedy diagram zobrazující všechny důležité informace o komponentové aplikaci. Dosažení této možnosti s sebou však přináší nepřenositelnost a odlišnou notaci mezi různými komponentovými modely. To je zásadní problém při studiu diagramů komponentových aplikací – vývojář musí dobře znát několik různých grafických notací.

Z důvodu nevýhod uvedených výše, byl navržen obecný komponentový ENT meta-model (více v kapitole 4), který lze použít pro dynamické zobrazení vnitřní struktury

komponentových aplikací různých komponentových modelů. Konkrétně ho tedy používá aplikace ComAV, respektive plug-in AIVA, která díky ENT meta-modelu zobrazuje aplikace různých komponentových modelů stejným způsobem. ENT meta-model pak tedy slouží jako datová vrstva plug-inu AIVA.

3.4.1 Obecné boxes-and-arrows diagramy

Vizualizace pomocí boxes-and-arrows diagramů je užitečná k výměně informací v podobě diagramů. Komponenty jsou zobrazeny pouze formou obdélníků a vztahy mezi nimi jsou vyjádřeny pomocí směrových čar (viz obrázek 3.1). Diagram tedy velmi obecný a poskytuje málo informací a specifických detailů komponent. Proto pak nelze dostatečně dobře pochopit popisovanou komponentovou aplikaci.



Obrázek 3.1: Obecný box-and-arows diagram

3.4.2 UML

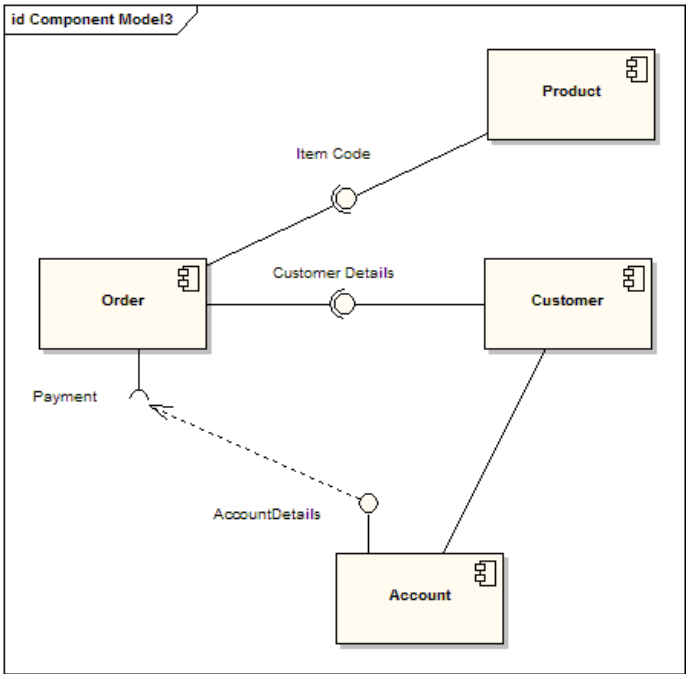
Hlavním představitelem boxes-and-arrows diagramů je UML 2.0 (Object Management Group). Jde o grafický jazyk pro zobrazení, specifikování a dokumentaci artefaktů softwarových aplikací. UML je nejznámější a nejpoužívanější prostředek pro modelování aplikací a UML 2.0 je pak novější verzí základního UML. Mezi používané UML diagramy patří například:

- class diagram

- deployment diagram
- komponentový diagram (na obrázku 3.2)

Jak je na obrázku 3.2 vidět, diagram poskytuje pouze základní informace o komponentách a jejich spojení. Lze poznat že například komponenta *Product* dodává

do komponenty *Order* funkčnost přes interface *Item Code*. Další podrobnější informace však nelze zjistit. Ačkoliv je snaha, tyto nedostatky odstranit v různých vylepšeních základního UML, projevuje se uvedená limitace v menší nebo větší míře ve všech těchto rozšířeních.



Obrázek 3.2: Ukázka UML komponentového diagramu

V článku [ICV11] jsou popsány požadavky na diagram komponent, kterým UML 2.0 nevyhovuje. Přitom se splněním těchto požadavků zrychlí a zlepší orientace a pochopení struktury komponentových aplikací. Mezi nedostatky patří například:

- Vývoj komponentových aplikací zasahuje do různých rolí vývojářů, jejichž

požadavky a potřeby se liší. UML používá diagram nový pro každou roli z důvodu poskytnutí přesného množství informací.

- UML zobrazuje všechny informace naráz. V rozsáhlých aplikacích, které obsahují velké množství komponent, tak roste nepřehlednost.
- Vztah mezi komponentami je zobrazen vždy samostatnou čarou, což znepřehledňuje rozsáhlé diagramy.

3.5 Interaktivní zobrazení komponent

Vyjde-li interaktivní zobrazení komponent z výzkumu uvedeného v kapitole 3.3, zjistíme, že zavedené kategorie interakcí zahrnují veskrze všechny požadavky interaktivního zobrazení komponentových aplikací. Do uvedených kategorií by tak mohly patřit například následující možnosti:

- **Select**
  - označení zkoumané komponenty, spojení mezi komponentami nebo jiných zobrazených prvků
- **Explore**
  - posun náhledu nad zobrazenými komponentami
  - zobrazení požadované komponenty na požádání (fulltext nebo podobné techniky)
  - změna zobrazených implementačních detailů komponenty
- **Reconfigure**
  - změna seřazení/zarovnání komponent
- **Encode**
  - změna reprezentace komponenty
  - změna reprezentace spojení mezi komponentami

- změna barev použitých pro zvýrazňování informací

- **Abstract/Elaborate**
  - zobrazení/skrytí implementačních detailů komponent
- **Filter**
  - zobrazení/zvýraznění komponent nebo jiných prvků grafu splňujících definovaná pravidla
- **Connect**
  - zvýraznění komponent majících mezi sebou nějakou závislost

Jak bude patrné z kapitol 8 a 9, většina výše uvedených techniky se používá v plug-inu AIVA, přičemž část z nich byla přidána v rámci této diplomové práce.

3.6 Meta-Modely pro popis komponent aplikací

Pro zobrazení komponentových aplikací je třeba použít informace, jež je popisují, a tyto informace musejí být uloženy v definovaných datových strukturách. Datové struktury navíc musí taktéž uchovávat informace o použitém komponentovém modelu. Díky tomu totiž lze pro popis komponenty použít definované prvky z popisu komponentového modelu.
Definici požadovaných struktur poskytuje MOF (Meta Object Facility) (více v článku [MOF06]), který popisuje čtyři abstraktní úrovně struktur. Pro popis komponentových aplikací jde o úrovně obsahující:

- M0 – implementace elementů komponentového modelu (komponenty), jež mohou být spuštěny
- M1 – struktury pro popis komponenty – komponentový model
- M2 – struktury pro popis možností modelu komponenty a možných vlastností komponenty – komponentový meta-model

- M3 – definice jazyku pro specifikování meta-modelu – komponentový meta meta-model

V další kapitole bude popsán ENT meta-model, jenž je právě M3 modelem.

4 ENT

ENT je generický MOF M3 model definující strukturu komponentových modelů a vytvářející konkrétní model příslušné komponentově orientované aplikace. Jak bude konkrétní model vypadat, závisí především na komponentovém modelu frameworku, pomocí něhož je aplikace napsána. V kapitole jsou použity znalosti z článků [DLFV10], [ICV11], [ENT11] a také z diplomové práce [LKM11].

4.1 Představení ENT meta-modelu

ENT model je od roku 2002 vyvíjen na Katedře informatiky a výpočetní techniky Západočeské univerzity. Záměrem modelu je zachytit společné charakteristiky komponent z pohledu uživatele (ve smyslu vývojáře, architekta, atd), díky čemuž lze lépe analyzovat vlastnosti komponent (jejich účel a použití).

Model definuje struktury komponentových modelů a aplikací napsaných pro tyto modely. Jeho hlavní charakteristikou je použití klasifikace tak, aby bylo možné vystihnout komponentu dostatečně srozumitelně i pro uživatele s rozdílnými zájmy.

ENT poskytuje obecný sémantický pohled na komponentu. To znamená, že poskytuje informace o účelu komponenty, způsobu její komunikace, jejích poskytovaných a požadovaných službách, aj. Klíčovou konstrukcí použitou v meta-modelu je ENT klasifikátor. Jedná se o osmici identifikátorů, které charakterizují prvky komponenty (např. rozhraní) z různého úhlu pohledu.

Název meta-modelu vychází z počátečních písmen slov – **E**xport, **N**eeds, **T**ies, specifikující zaměření modelu. Jedná se o popis konkrétní komponenty, co komponenta poskytuje a vyžaduje od venkovního světa a nakonec stanovení vazeb uvnitř samotné komponenty.

4.2 Stručný popis architektury

Důležitou vlastností ENT modelu je jeho rozdělení do dvou úrovní, a to na úroveň komponentového modelu a na aplikační úroveň. Úroveň komponentového modelu

definuje specifické vlastnosti příslušného komponentového modelu. Aplikační úroveň pak popisuje pomocí komponentového modelu konkrétní komponenty, jejich rozhraní a závislosti mezi nimi v analyzované aplikaci.

4.2.1 Úroveň komponentového modelu

Hierarchie uspořádání ENT modelu začíná u komponentového modelu, který definuje množinu možných typů komponent. Různé komponentové modely totiž mohou definovat více typů komponent. Příkladem je OSGi, pro nějž existuje jediný typ komponenty s názvem *bundle*. Druhým příkladem je EJB, jenž obsahuje tři typy komponent: *SessionBean*, *MessageDrivenBean* a *Entities*.

Typ komponenty je v ENT meta-modelu definován minimální množinou definic, takzvaných *trait* (*vlastností*), které slouží k popisu různých druhů prvků, které jsou zajímavé a charakterizují komponentu. V ENT meta-modelu jsou typy komponent a jejich možných traitů založeny na ruční analýze příslušného komponentového modelu.

Ostatní důležité informace o komponentě, jež nemá smysl popsat traity, se používají *tagy* (*informace*). U komponentového modelu OSGi to může být například verze bundlu nebo také symbolický název bundlu. Tag je definována trojicí hodnot: název, povolené hodnoty a výchozí hodnota. Název tagu musí být jedinečný v rámci příslušné komponenty. V případě, že nejsou uvedeny povolené hodnoty, je možné do tagu uložit jakoukoliv hodnotu.

Každá komponenta obsahuje množinu komunikačních *elementů*. Viditelným pro okolní komponenty na rozdíl od vnitřku komponenty, která se bere jako černá skříňka. Stejné typy elementů jsou obsaženy v příslušných traitech komponenty.

4.2.2 Aplikační úroveň

Komponenty, z nichž je vytvořena analyzovaná aplikace, jsou reprezentovány na této úrovni ENT meta-modelu a to referencemi na prvky definované komponentovým modelem. Komponenta je tedy zde reprezentována svým typem, možnými traity tohoto typu, elementy a hodnotami tagů, jež jsou zde přípustné i u elementů.

Samotné traity nic neříkají o schopnosti příslušné komponenty. Slouží pouze k seskupení a prostřednictvím odkazu na definici traitu je dán význam všem elementům tohoto traitu.

Na aplikační úrovni jsou taktéž uchovávány informace o spojení mezi komponentami. K tomu se využívají elementy popsané na úrovni komponentového modelu. Spojení se můžou vyskytovat ve stejné nebo jiné komponentě v rámci modelu. V případě hierarchických modelů je rovněž pro komponentu uložen seznam vlastních *podkomponent*, z kterých je komponenta složena.

Zdrojová a cílová trait (skupina pro elementy spojení) vazby musí být tematicky shodná. Tím jsou například myšleny traity *provided_interfaces* a *required_interfaces* (OSGi), kdy první z nich uvádí seznam poskytovaných rozhraní komponenty, druhá naopak uvádí seznam vyžadovaných rozhraní komponenty. Zdrojová trait musí také mít v klasifikaci uvedenou roli jako *provided*, cílová trait musí mít roli *required* (viz následující podkapitola).

4.3 Klasifikační systém

Klasifikační systémem ENT meta-modelu je množina dimenzí, jež charakterizují elementy komponenty. Element může být klasifikován jednou nebo více hodnotami jedné dimenze. Příkladem takové dimenze je například *Role*, která v případě elementu *use package* u OSGi modelu nabývá hodnot *provided* i *required*. Klasifikační systém ENT modelu obsahuje osm různých dimenzí:

1. *Nature = {syntax, semantics, extra-functional}*
2. *Kind = {operational, data}*
3. *Role = {provided, required, neutral}*
4. *Granularity = {item, structure, compound}*
5. *Construct = {constant, instance, type}*
6. *Presence = {mandatory, permanent, optional}*

7. *Arity = {single, multiple}*
8. *Lifecycle = {development, assembly, deployment, setup, runtime}*

ENT klasifikátorem, použitým pro charakterizaci traitů a tedy i elementů, je pak množina výše definovaných dimenzí, pod které příslušný element spadá. U těchto dimenzí klasifikátor nabývá alespoň jednu definovanou hodnotu.

4.4 Category sety

Zobrazením všech trait komponenty lze získat veškeré informace, nicméně za cenu, že obsah komponenty nebude stručný a může být nepřehledný. Lepší by tedy bylo obsah komponenty nějakým způsobem filtrovat. Dalším případem, kde by se hodilo filtrování obsahu komponenty, je zobrazení diagramu uživateli různých funkcí. Softwarového architekta můžou zajímat jiné informace než programátora.

ENT meta-model poskytuje možnost organizovat informace o modelu pomocí takzvaných *category sets* (*množina kategorií*). Jednotlivé category sety jsou definovány pomocí výběrových operátorů nad klasifikačním systémem (viz předchozí kapitola).

Například category set *Export-Needs-Ties* má tři skupiny. Do první skupiny patří elementy zařazené do traitu s dimenzí *role = {provided}*, což jsou elementy, jež komponenta poskytuje (Export). Druhou skupinou jsou požadované elementy (Needs) a poslední skupinou jsou elementy zároveň poskytované a požadované (Ties).

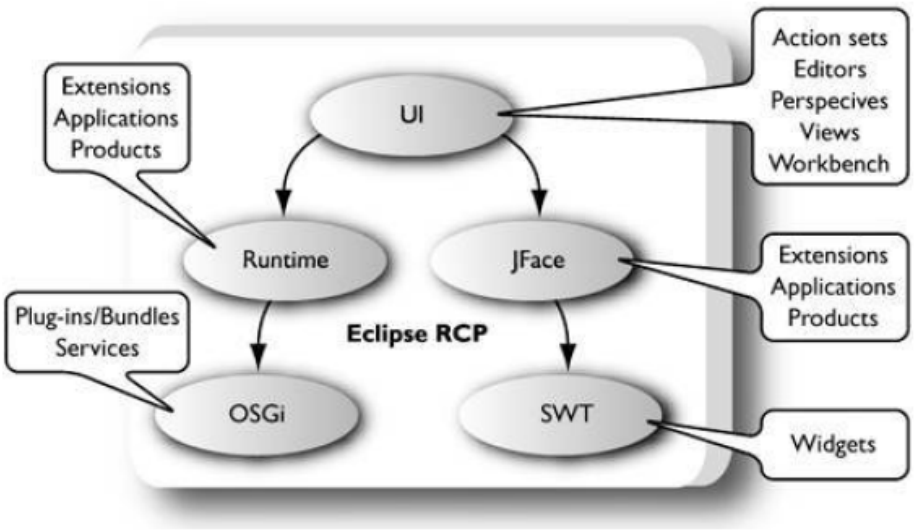
Pomocí category setů lze tedy snadno filtrovat obsah komponenty a tím lze zobrazit pouze požadované informace bez redundantních a nadbytečných informací.

5 Eclipse RCP

Rozšiřovaná aplikace ComAV je postavena nad platformou Eclipse RCP. Ačkoli tato diplomová práce do vývoje pomocí Eclipse RCP zasáhla pouze povrchně, budou nastíněny alespoň základy vývoje v tomto prostředí. V popisu budou uvedeny poznatky získané z knihy [ERCP10], z užitečných internetových tutoriálů [VOG], ale i z diplomové práce [GEPF10]. V těchto publikacích pak lze nalézt detailnější popis Eclipse RCP.

5.1 Základy Eclipse RCP

Eclipse RCP je platformou použitelnou pro vývoj komponentových aplikací. Platforma je skupinou několika základních komponent, respektive features[1], poskytující mezivrstvu pro uživatelské aplikace. Na obrázku 5.1 jsou vidět tyto komponenty a vztahy mezi nimi. Vývojáři tak například odpadají problémy s životním cyklem komponent (tj. jejich instalací, atd.), výsledná aplikace získá nativní vzhled na různých operačních systémech a je usnadněn vývoj uživatelského rozhraní.



Obrázek 5.1: Základní skupiny tvořící Eclipse RCP

1 Pojem features znamená skupinu komponent spojených do logického celku. Ty pak lze přidávat nebo odebírat do aplikace jako jeden celek.

V prostředí Eclipse RCP se místo obecnějšího pojmu komponenta používá pojem plug-in, jenž je bližší implementaci.

5.1.1 Skupina OSGi

Základem platformy je implementace komponentového modelu OSGi, což je framework s názvem Equinox. Stará se o dynamické spouštění plug-inů aplikací a jejich propojení. Dynamickým spouštěním je myšleno spuštění plug-inu až v okamžiku, kdy je nutný pro běh celé aplikace.

5.1.2 Skupina Runtime

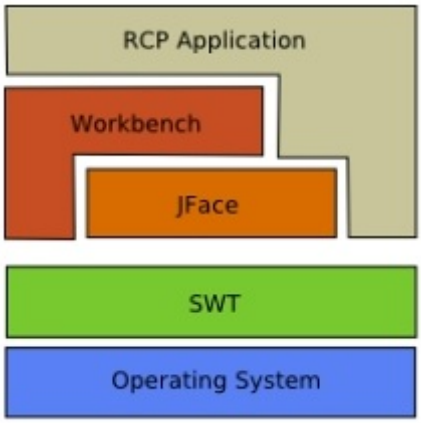
Běhové prostředí platformy Eclipse RCP obsluhuje aplikační model a spravuje registry rozšíření (*extensions*) komponent. Rozšíření umožňuje deklarovat vztahy mezi plug-iny. Plug-in definuje své možné rozšíření pomocí takzvaného bodu rozšíření (*extension point*). Bod rozšíření obsahuje svůj jednoznačný identifikátor, jméno a schéma, ve kterém je popsána struktura případného rozšíření. O případných rozšířeních nemusí návrhář komponenty v době definování bodu rozšíření vědět.
Tímto je umožněno ostatním plug-inům přidat funkcionalitu založenou na kontraktu definovaném v bodu rozšíření. Ty pak vkládají požadované informace ve formě extensions, splňujících požadovanou strukturu.

5.1.3 Skupina UI

Vytváření uživatelského rozhraní je v Eclipse RCP rozděleno do několika skupin. Na obrázku 5.2 je vidět hierarchie vrstev uživatelského rozhraní z pohledu Eclipse RCP.

5.1.4 Skupina SWT

SWT je standardní komponentová knihovna uživatelského rozhraní používaná v Eclipse. Byla vytvořena Eclipse komunitou jako alternativa ke standardní grafické knihovně AWT/Swing. To, že se o vykreslování grafických komponent stará JVM a



Obrázek 5.2: Vrstvy uživatelského rozhraní z pohledu Eclipse RCP

uživatelský systém poskytuje pouze okno či rámec, může být výhodou, protože aplikace všude vypadá stejně. Větší počet grafických AT/Swing komponent má však z následek nadměrné zatížení procesoru.

SWT je také oproti Swingu nízko-úrovňová knihovna využívající JNI (Java Native Interface). Díky tomuto rozhraní přistupuje k API operačního systému a všechny grafické prvky jsou tak poskytovány a ovládány přímo operačním systémem. Knihovna SWT je implementována pro všechny moderní operační systémy a je tedy přenositelná a zároveň také nativní.

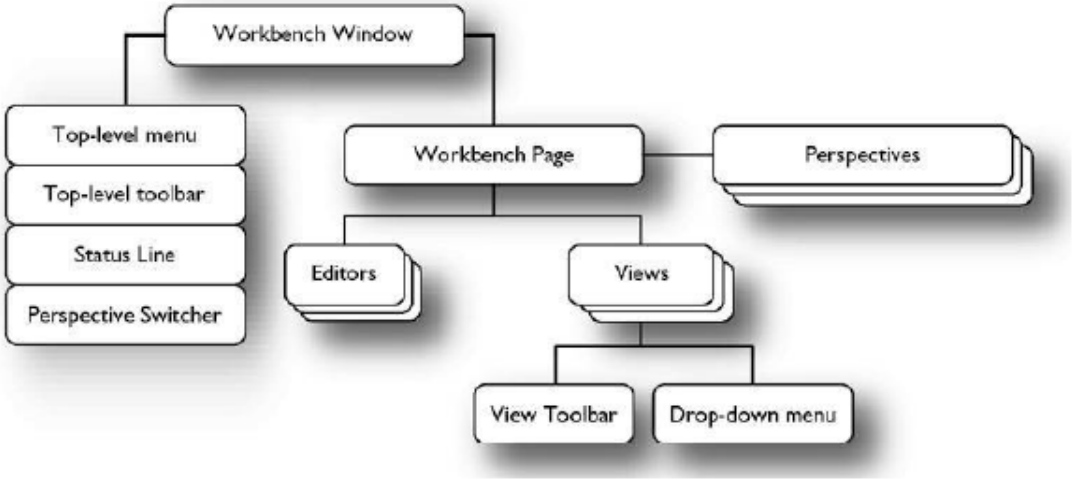
5.1.5 Skupina JFace

JFace poskytuje některá užitečná API pro práci s uživatelským rozhraním a nadstandardní grafické prvky nad SWT. Zahrnuje prostředky pro zpracování běžných úkolů programování uživatelského rozhraní:

- Viewers – obsluha grafických prvků SWT jako jsou seznamy, stromy, tabulky nebo textové prvky
- Akce – poskytuje sémantiku pro definování akcí
- Obrázky a fonty – poskytnutí vzoru pro práci se zdroji uživatelského rozhraní
- Dialogy a průvodci – poskytnutí frameworku pro stavbu komplexních interakcí s uživatelem

5.2 Workbench uživatelského rozhraní

Workbench je nejvyšší vrstvou úrovní pro tvorbu uživatelského rozhraní. Zastupuje správce prvků pro uživatelské rozhraní a umožňuje jejich snadné a efektivní používání. Mezi jeho hlavní funkce patří správa registrů oken a stavebních prvků grafického rozhraní, komunikace mezi těmito prvky a jejich rozložení v hlavním okně. Na obrázku 5.3 je přehledná struktura popisující složení hlavního okna uživatelského rozhraní z pohledu platformy.



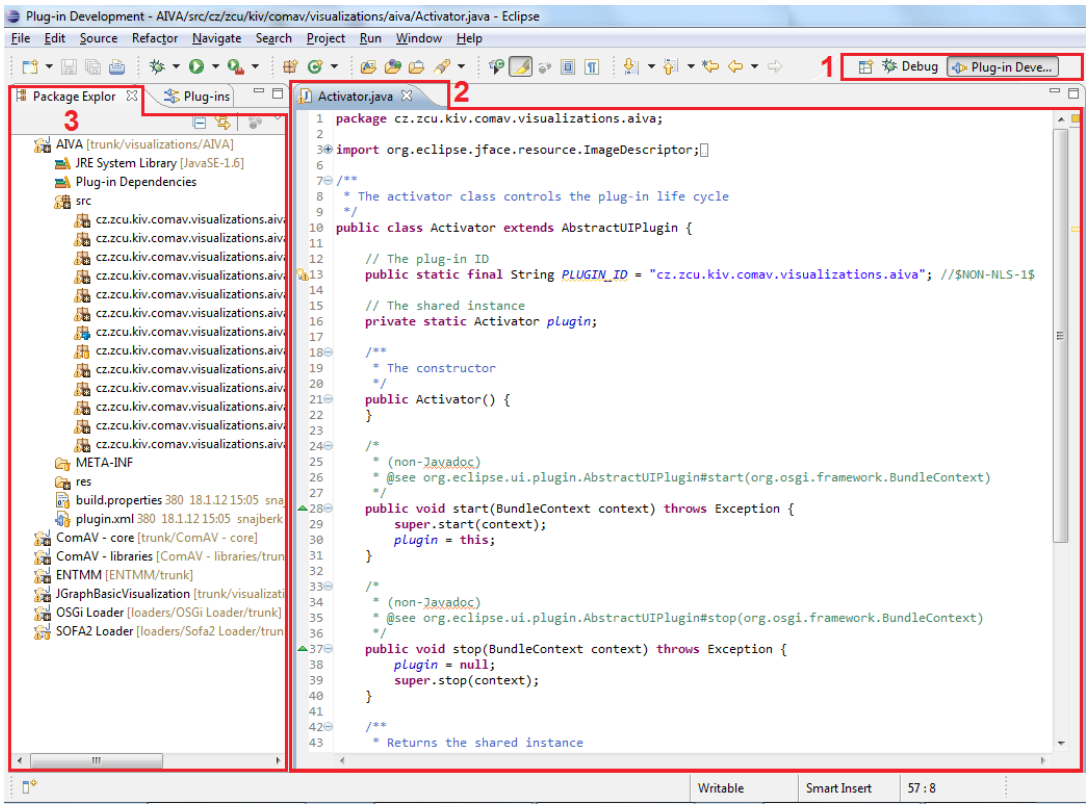
Obrázek 5.3: Kompozice hlavního okna UI z pohledu platformy

Perspectives jsou v zásadě obrazovky shlukující a organizující jednotlivé prvky uživatelského rozhraní. V každém okně lze mít několik těchto obrazovek a uživatel má možnost mezi nimi přepínat (viz známá změna perspective v Eclipse IDE, například z *Javy* na *Debug* perspective). Na obrázku 5.4 pod číslem **1**.

Editor je ve většině Eclipse RCP aplikací hlavním prvkem uživatelského rozhraní. Je to místo, kde se zobrazuje hlavní výstup (textový, grafický) aplikace a s nímž může uživatel dále pracovat. V Eclipse IDE je tímto prvkem hlavní textový editor zobrazujícím zdrojový kód souborů. Na obrázku 5.4 pod číslem **2**.

View je používán jako podpůrný prostředek pro práci s editorem.

V zásadě by měl uchovávat prostředky pro úpravu obsahu editoru. Na obrázku 5.4 pod číslem 3.



Obrázek 5.4: Základní prvky uživatelského rozhraní Eclipse

Důležité je také zmínit, že jako prostředek popisující plug-in, je využíván soubor *MANIFEST.MF* (princip OSGi modelu), v němž jsou uvedeny závislosti plug-inu na ostatních plug-inech, jeho vstupní bod (třída), jeho Java balíky, které mohou být používány v ostatních plug-inech a další vlastnosti. Soubor *MANIFEST.MF* je uložen kořenovém adresáři plug-inu.

Body rozšíření a vlastní rozšíření (popsáno na začátku kapitoly) jsou naopak uloženy v konfiguračním souboru *plugin.xml*, který každý plug-in v prostředí Eclipse povinně obsahuje ve svém kořenovém adresáři.

6 JGraphX

Rozšiřovaná aplikace reprezentuje komponentové aplikace, tedy komponenty a vztahy mezi nimi. To znamená, že v zásadě pracuje s dynamickým grafem. Pro práci s ním je používána knihovna JGraphX, což je Java verze knihovny mxGraph implementovaná pomocí knihovny Swing. Ta je poskytuje funkce služby pro práci s interaktivními diagramy a grafy. Samotná knihovna je vydána mimo Javu jako knihovna jazyka JavaScript, ActionScript (pro Flex/Flash aplikace) a .NET. Ačkoli se tedy implementační detaily knihovny mxGraph různí, poskytuje obecné rozhraní pro práci, kde to je možné. Java verze je vhodná právě pro desktopové aplikace.

6.1 Pojmy a architektura

Graf je základním pojmem vycházejícím z matematické teorie sítí a grafů. Formálně se graf **G** skládá z neprázdné množiny uzlů **V**(**G**) a množiny hran **E**(**G**). Hrany jsou uchovávány jako neuspořádané dvojice spojených uzlů **V**(**G**). Pokud tedy hrana (**x**,**y**) náleží množině **E**(**G**), pak **x** i **y** náleží množině **V**(**G**) a jde o sousední uzly.

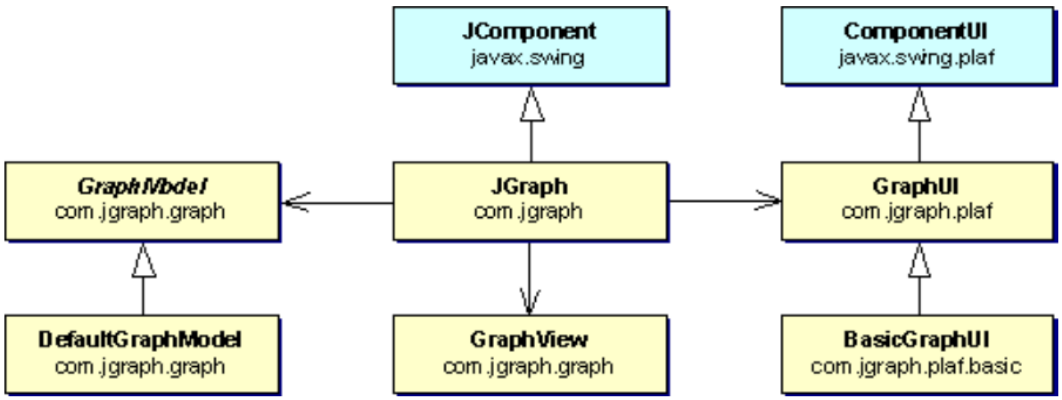
Knihovna mxGraph používá pro prvky grafu následující terminologii:

- *vertex*, dále *uzel*
- *edge*, dále *hrana*
- *cell* dále *buňka* pro obecné prvky (tedy *uzel* nebo *hrana*)

Jednou z hlavních předností JGraphX je rozsah práce s vizualizací grafu. JGraphX podporuje široké spektrum funkcí pro zobrazení *buněk* omezené pouze schopnostmi vývojáře a platformou Swing. Například uzly mohou být reprezentovány základními tvary, obrázky, vektorovými kresbami nebo animacemi [JSC].

V grafu je také povolena interakce s *buňkami*, například posun a kopírování, změna velikosti a tvaru, připojování a odpojování *hran*, a jiné možnosti.

JGraphX je z většiny založen na komponentě Swing pro stromy, tedy JTree. Stejně jako JTree používá návrhový vzor MVC, který rozděluje aplikace na model, pohled a kontroler (na obrázku 6.1 zleva).



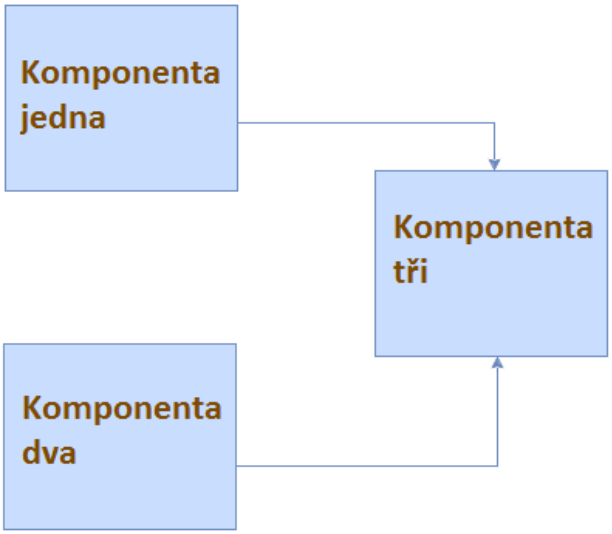
Obrázek 6.1: Rozdělení obecného mxGraph podle návrhového vzoru MVC

Model je jádrem grafu, především popisuje jeho strukturu. Veškeré datové úpravy (jako je přidáváni nebo odebírání *buněk*) se provádí přes jeho API. Respektive jeho API je přístupné přes třídu grafu. A to z důvodu, že koncept „přidej uzel do grafu“ je přirozenější než „přidej uzel do modelu grafu“ [JUM].

Kontroler definuje mapování uživatelských akcí na metody grafu. Je implementován platformě závislý UI delegát (na obrázku 6.1 jej představují třídy GraphUI a BasicGraphUI). V zásadě obsluhuje práci s *buňkami* a jejich úpravu a obnovuje náhled grafu.

6.2 Vytvoření grafu

Uzel je pomocí JGraphX standardně zobrazen pouze pomocí obdélníku obsahujícího zadaný název uzlu. Hrany jsou standardně zobrazeny jako linie, s případným směrovým vyznačením. Malý graf vytvořený pomocí JGraphX s uzly se základním vzhledem, je na obrázku 6.2



Obrázek 6.2: Graf bez použití překrytí

Tento graf by se vytvořil pomocí následujícího krátkého pseudokódu:

```
var graph = new mxGraph();
var component = new mxGraphComponent(graph);
Object parent = null;

//Začátek API pro přidání buněk do grafu v jednom kroku
graph.getModel().beginUpdate();
try
{
	/*** Přidání uzlů ***/
	Object v1 = graph.insertVertex(parent, null, "Komponenta
		jedna,", 20, 20, 120, 120);
	Object v2 = graph.insertVertex(parent, null, "Komponenta dva",
		20, 200, 120, 120);
	Object v3 = graph.insertVertex(parent, null, "Komponenta tři",
		200, 100, 120, 120);

	/*** Přidání hran ***/
	graph.insertEdge(parent, "Hrana 1", null, v1, v2);
	graph.insertEdge(parent, "Hrana 2", null, v2, v3);
}
finally
{
	//Konec API pro přidání buněk do grafu v jednom kroku
	graph.getModel().endUpdate();
}
```

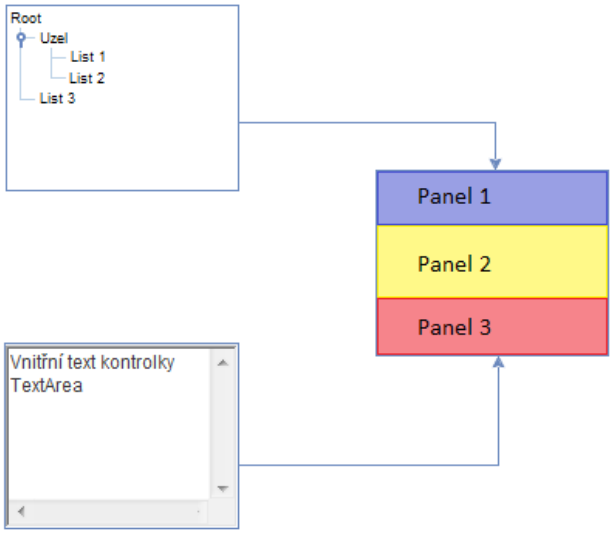
Instance třídy pro komponentu grafu (třída `mxGraphComponent`) dědí od JScrollPane a proto ji lze snadno dále použít, například přidat do panelu, pomocí prostředků knihovny Swing.

Parametry metody `inserVertex()` pro vložení uzlu do grafu jsou rodič uzlu, kterým může být jiný uzel (standardně `null`), hodnota uzlu (standardně `null`), identifikátor, požadovaná x-pozice uzlu, požadovaná y-pozice uzlu, požadovaná šířka uzlu, požadovaná výška uzlu. Pro hodnoty souřadnic je brán jako počátek souřadnic levý horní roh jejich nastaveného rodiče. Od tohoto bodu jsou pozice uzlů relativně vzdáleny o nastavené hodnoty. Pokud má rodič uzlu hodnotu `null`, stejně jako v předchozím případě, je za výchozí bod brán levý hodní roh celého grafu.

Parametry metody `inserEdge()` pro vložení hrany do grafu jsou rodič hrany (standardně `null`), identifikátor, hodnota (standardně `null`) a startovní s cílovým uzlem, určujícím odkud a kam hrana povede.

6.3 Překrytí uzlu

Jak bylo zmíněno, uzly grafu mohou být reprezentovány například pomocí různých tvarů. JGraphX toho dosahuje pomocí takzvaného překrytí uzlů. To lze přiřadit libovolným vytvořeným uzlům grafu pomocí prostředků knihovny Swing. Díky překrytí lze tedy dosáhnout téměř libovolného vzhledu uzlu. Omezením vzhledu překrytí je obdélníková obálka vytvořeného uzlu, kterou překrytí nemůže přesáhnout. Příklad grafu s různými překrytími uzlů je na obrázku 6.3.



Obrázek 6.3: Graf s použitím překrytí uzlů

Vytvoření grafu by obsahovalo identický kód jako v předchozí kapitole. Kód by se rozšířil pouze o nastavení překrytí vytvořených uzlů, provedené následujícím pseudokódem:

```
//...Vytvoření buněk grafu

component.addCellOverlay(v1, new TreeOverlay()):
component.addCellOverlay(v2, new TextAreaOverlay()):
component.addCellOverlay(v3, new PanelsOverlay()):
```

Také by tedy bylo nutné vytvořit třídy pro překrytí (`TreeOverlay`, ...). Ty by pouze dědily od rozhraní `mxICellOverlay` a konstruovány by už byly pomocí prostředků knihovny Swing.

6.4 Layouty

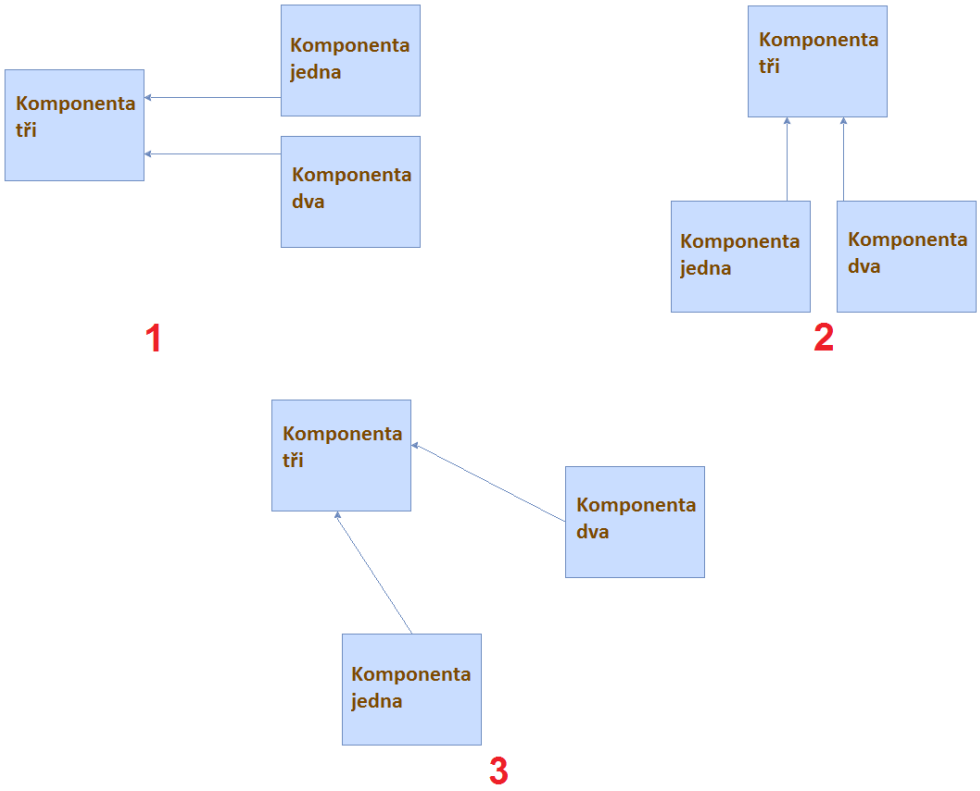
V kapitole 6.2 bylo popsáno, jakým způsobem lze umísťovat uzly grafu a uspořádat si tedy graf do snadno čitelného stavu. Ale například pro dynamicky generované grafy, ve smyslu neznámého počtu uzlů a jejich závislostí, je tento způsob nedostačující. Naštěstí JGraphX poskytuje prostředek, jak snad graf snadno smysluplně srovnat a uspořádat. Je to umožněno layouty, respektive layout třídami.
Graf lze uspořádat v libovolné situaci a lze použít třídy poskytující následující základní uspořádání:

- horizontální (**1**)
- vertikální (**2**)
- kruh (**3**)

Malé ukázky uvedených layoutů jsou na obrázku 6.4 (očíslováno podle seznamu výše). Změna layoutu grafu například na horizontální uspořádání by se pak provedla následujícím krátkým kódem:

```
mxIGraphLayout layout = new mxHierarchicalLayout(graph,
      SwingConstants.WEST);
try {
      Object p = graph.getDefaultParent()//rodič uzlů grafu, defaultně null
      layout.execute(p);
```

```
    }
catch (Exception e)
...
```



Obrázek 6.4: Základní layouty grafu pomocí JGraphX

7 ComAV

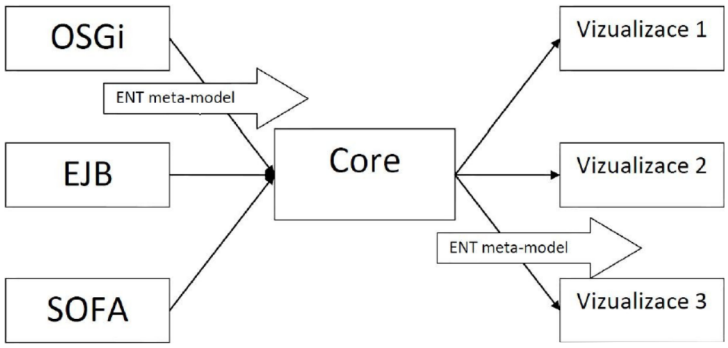
V následující kapitole bude popsána aplikace ComAV, jejíž součástí je plug-in pro zobrazení komponentových aplikací AIVA.

7.1 Architektura ComAV

ComAV je komponentovou aplikací umožňující interaktivně zobrazovat informace o komponentových aplikacích a jejich komponentách. Umožňuje načítat komponentové aplikace a informace o nich jsou ukládány do ENT meta-modelu (viz kapitola 4). Tyto informace je pak možné zobrazit a dále s nimi pracovat. ComAV je vytvořena pomocí Eclipse RCP (viz kapitola 5), tedy je implementována v jazyce Java a je sama komponentovou aplikací. Skládá ze 3 typů plug-inů:

1. *Loader* pro načítání OSGi komponent a uložení informací o komponentách do ENT meta modelu.
2. *Visualizer* pro zobrazení závislostí a také základní volby umožňující upravující zobrazení.
3. *Core* předávající načtené informace z plug-inů typu *Loader* do plug-inů typu *Visualizer.*

Základní obecná architektura ComAV je na obrázku 7.1.

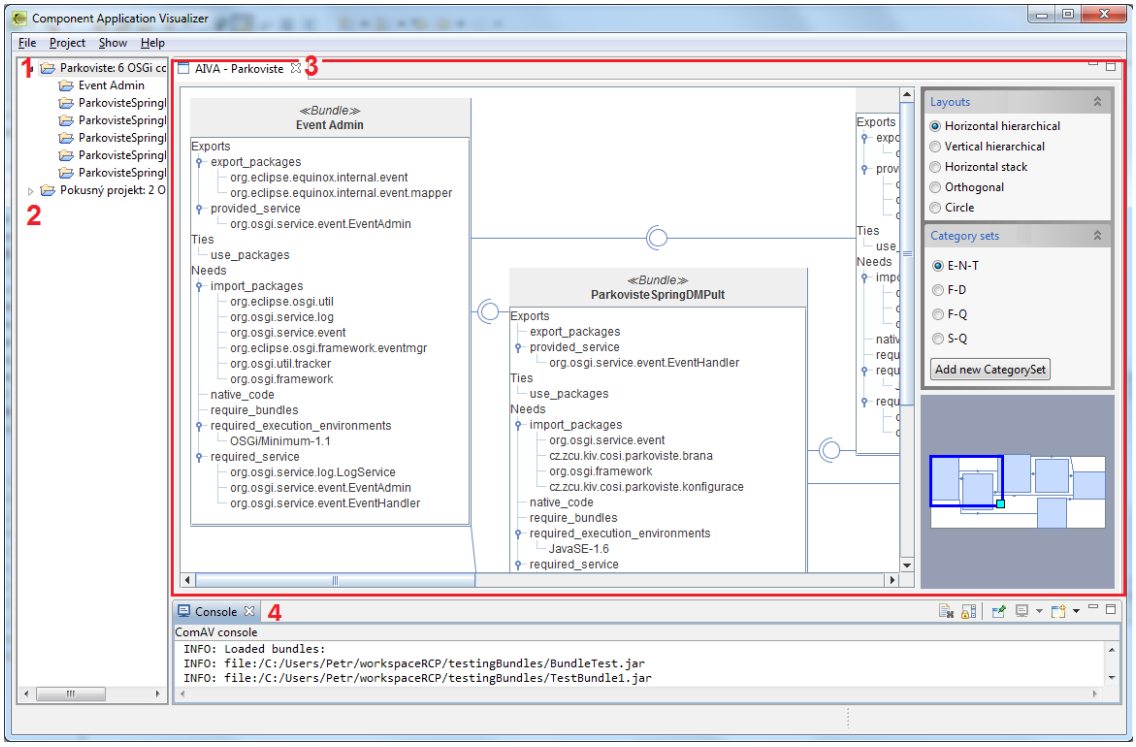


Obrázek 7.1: Architektura aplikace z hlediska plug-inů

Loader plug-iny se starají o načítání aplikací a uložení informací do ENT meta-modelu. *Visualizer* plug-iny se starají o interaktivní zobrazení načtených informací a plug-in *Core* se stará o běh a vzhled aplikace a předávání načtených dat mezi oběma typy plug-inů.

Mimo tyto základní plug-iny stojí ENTMM, což je plug-in obsahující implementaci ENT meta-modelu. Plug-in je tedy používán pro uložení načtených informací, tj. popis komponentové aplikace, a jejich předání do *Visualizer* plug-inů.

Spuštěná aplikace s otevřeným projektem ve *Visualizer* plug-inu AIVA je na obrázku 7.2. Červená čísla popisují důležité části aplikace. Podrobnější popis prvků plug-inu AIVA bude uveden v kapitole 8.



Obrázek 7.2: Ukázka aplikace s otevřeným projektem

Na předchozím obrázku jsou barevně vyznačeny hlavní vizuální prvky plug-inu *Core*:

1. Otevřený OSGi projekt *Parkoviste* skládající se ze 6 komponent
2. View oblast pro načtené projekty aplikace - workspace

3. Editor oblast pro plug-in AIVA
4. View konzole pro výpis informací

7.2 Extension points ComAV

Obě rozhraní mezi plug-inem *Core* a *Loader*/*Visualizer* plug-iny jsou vytvořena pomocí extension points (viz kapitola 5.1.2). Jde o prostředek platformy Eclipse RCP, díky které plug-iny dodávají funkcionalitu do jiného plug-inu. Extension points tedy propojují dva a více plug-inů.

7.2.1 Extension point pro *Loader*

Na data získaná z *Loader* plug-inů se v plug-inu *Core* aplikuje analogie s vývojovým prostředím Eclipse, tedy se z nich vytvoří projekt.

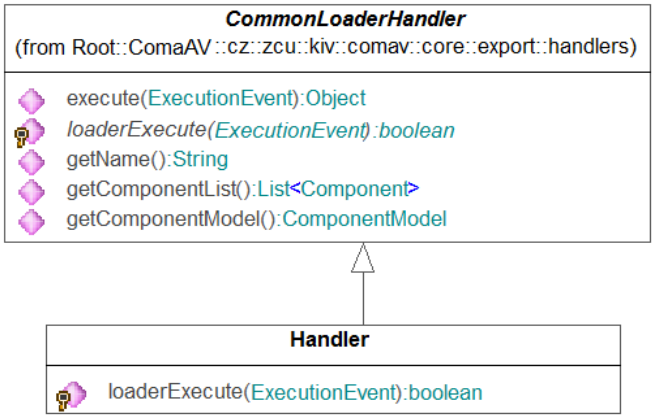
Plug-in *Core* definuje extension point pro *Loader* plug-iny, které musí obsahovat handler třídu pro obsluhu získání dat ENT meta-modelu z načtených komponent. Spouštění vytváření nového projektu a předávání dat z instancí tříd v plug-inech *Loader* do plug-inu *Core* je netriviální. V aplikace je toto řešeno využitím:

- dědičností u handlerů spravujících načítání a předávání dat
- návrhového vzoru Singleton u třídy uchovávající projekty

Dědičnost handleru je znázorněna na částečném diagramu tříd obrázku 7.3.

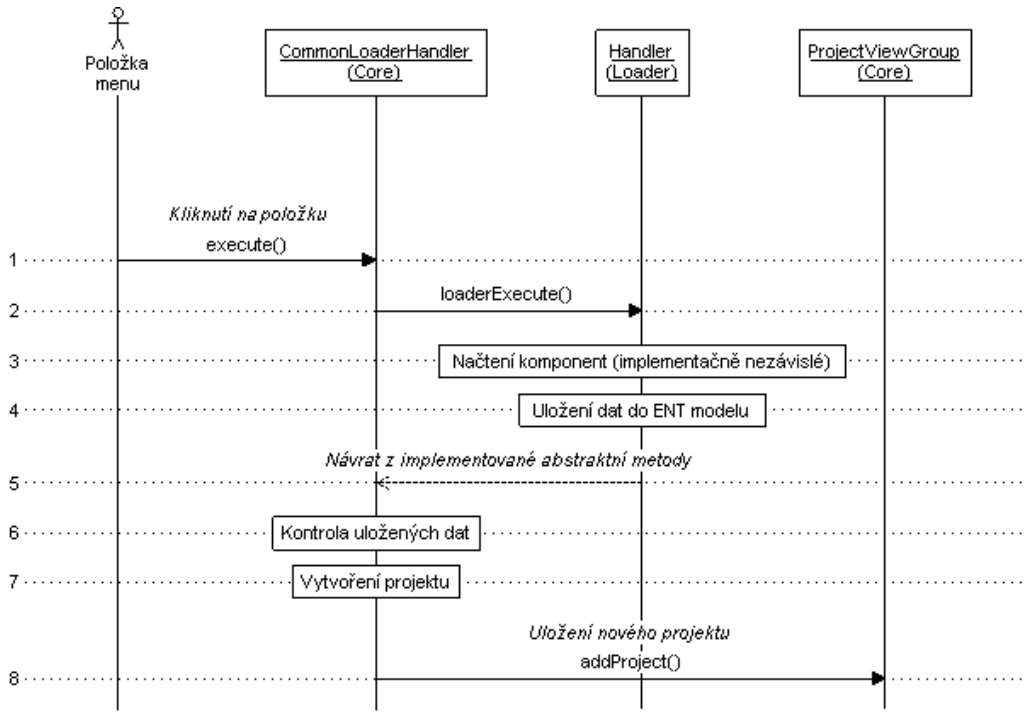
Handler z modulu *Core* dědí od třídy `AbstractHandler` z balíku `org.eclipse.core.commands`, což umožňuje použít jej jako obsluhující objekt příkazu. Příkaz pak může být navázán například na položku menu.

Přesně tímto způsobem je řešena komunikace mezi oběma popisovanými plug-iny. Plug-in *Loader* přidává přes extension point pro menu do aplikace vlastní položku pro vytvoření projektu. Z toho je patrné, že v *Loader* plug-inu je nutná jistá znalost plug-inu



Obrázek 7.3: Class diagram handleru plug-inu *Core* a obecného plug-inu *Loader*

Core. Nutnost přidání položky do menu pro *Loader* plug-in je však zmíněna společně s dalšími detaily v popisu extension point. Na přidanou položku menu je navázán příkaz, jehož obsluhou je handler z *Loader* plug-inu. Průběh spuštění načítání dat a jejich ukládání jako nového projektu v plug-inu *Core* je znázorněn na sekvenčním diagramu obrázku 7.4.



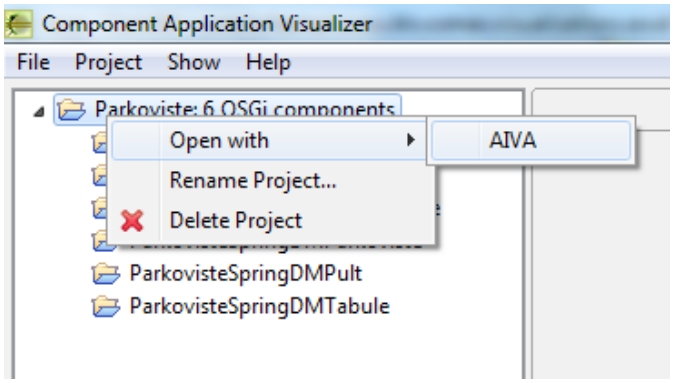
Obrázek 7.4: Sekvenční diagram komunikace mezi plug-iny *Loader* a *Core*

Uvedené poznatky pro extension point *Loader* plug-inů lze shrnout do několika vět. Přidání možnosti, pomocí které se načítají projekty, se provádí přímo v *Loader* plug-inu a to pomocí extension point pro menu. Obslužná třída přidané položky musí být třídou dědící od handleru definovaného v extension point. Handler v plug-inu *Core* provede obsluhu stisknutí položky menu a poté volá metodu pro načtení projektu v obslužné třídě *Loader* plug-inu. Po úspěšném načtení dat vytvoří handler v plug-inu *Core* projekt a uloží jej do třídy starající se o projekty.

7.2.2 Extension point pro *Visualizer*

Extension point pro *Visualizer* moduly je definován velmi podobně jako u *Loader* plug-inů. Opět se využívá dědičnosti u handlerů pro obsluhu příkazu. Pro *Visualizer* plug-iny je ale situace snadnější v tom, že stačí pouze spustit požadavek na otevření projektu (společně s předanými daty projektu). Není nutné získávat nazpět data jako v případě *Loader* plug-inů.

Co je naopak komplikovanější, je přidání možností, jak načtené projekty otevírat pomocí *Visualizer* plug-inů. Nejjednodušší možností by bylo přidat položku do menu přes extension point. Takto jednoduše by však nebylo možné přidat položku do kontextového menu pro načtené projekty ve workspace (přiblížení k analogii s vývojovým prostředím Eclipse IDE, viz obrázek 7.5).



Obrázek 7.5: Kontextové menu načteného projektu

O přidání (instalaci) *Visualizer* plug-inů se tak stará plug-in *Core*, který si sám přidá potřebné položky pro plug-in do standardního menu a do kontextových menu pro projekty. Při instalaci se využívá mechanismu dependency injection (pro Eclipse RCP popsaný blíže v [VOG]).

Díky dependency injection lze získat všechny plug-iny využívající konkrétní extension point. Pro tyto plug-iny lze vytvořit instanci třídy, jež je uvedená v extension point (pro *Visualizer* plug-in tedy handler). Komentovaná část pseudokódu pro instalaci *Visualizer* plug-inů vypadá následovně:

```
...

IExtensionRegistry reg = Platform.getExtensionRegistry();

// "cz.zcu.kiv.comav..." - id extension point pro Visualizer plug-iny
IConfigurationElement[] configExtensions =
      reg.getConfigurationElementsFor("cz.zcu.kiv.comav.visualizations");

for (var it : configExtensions) {
      // Získání jména modulu
      String name = it.getAttribute("name");

      Object handler = null;
      try {
            // Vytvoření instance handleru
            handler = elem.createExecutableExtension("handler");
      } catch (CoreException ce) {
            // Chyba ...
      }
      // Vytvoření instance pro plug-in a její uložení pro práci v aplikaci
}

...
```

Plug-in *Core* pak tedy pro dostupné *Visualizer* plug-iny přidává položku v menu, jehož obsluhou bude instance handleru vytvořená pomocí metody `createExecutableExtension()`. Atributy *`name`* a *`handler`* jsou definovány jako povinné v extension point pro *Visualizer* plug-iny.

Níže je pak uvedena komentovaná část extenstion point pro *Visualizer* plug-in:

```
<!-- Informace o extension point -->
<annotation>
      <appinfo>
            <meta.schema plugin="cz.zcu.kiv.comav.core"
```

```
                  id="cz.zcu.kiv.comav.visualizer"
                  name="Visualizer Extension"/>
      </appinfo>
</annotation>

<!-- Samotná definice extension point, složen z elementu komplexního typu -->
<element name="extension">
      <complexType>
            <!-- Sekvence obsahucící právě 1 prvek Visualizer -->
            <sequence>
                  <element ref="Visualizer"/>
            </sequence>
            <!-- Ostatní atributy extension point -->
            ...
            <attribute name="name" type="string" />
      </complexType>
</element>

<!-- Definice Visualizer elementu -->
<element name="Visualizer">
      <complexType>
            <!-- Handler musí dědit od CommonVisualizerHandler -->
            <attribute name="handler" type="string" use="required">
                  <annotation>
                        <appinfo>
                              <meta.attribute kind="java"
      basedOn="comav.core.export.handlers.CommonVisualizerHandler:"/>
                        </appinfo>
                  </annotation>
            </attribute>
        <attribute name="name" type="string" use="required" />
      </complexType>
   </element>
```

8 AIVA

Interaktivní vizualizace komponentových aplikací je tedy pouze jednou částí celé aplikace ComAV. Plug-in AIVA ke své činnosti využívá plug-inu JGraphBasicVisualization. To je plug-in skládající se z částí plug-inu AIVA, které neobsahují aplikační logiku. Díky tomuto rozdělení je kód plug-inu AIVA přehlednější. Také se použilo přístupu rozdělení relativně samostatných částí aplikace do oddělených plug-inů, což je hlavní myšlenkou komponentového vývoje aplikací.

V dalším textu se u popisů funkčností nebo jiných částí nebude důsledně odlišovat, do kterého z těchto plug-inů patří. Často jde totiž o výsledek prostředků poskytovaných oběma plug-iny.

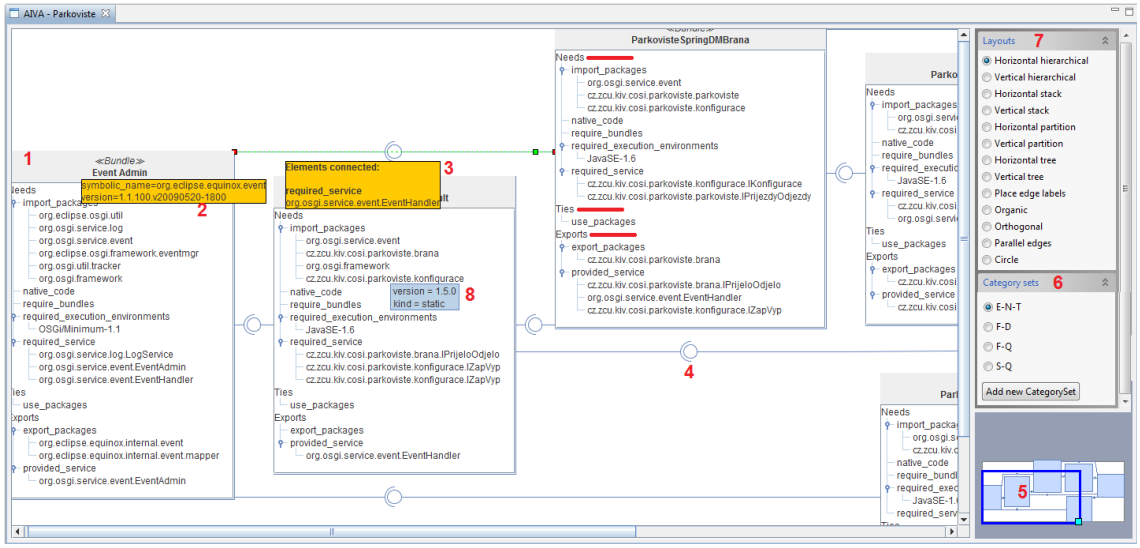
8.1 Architektura a vzhled plug-inu AIVA

Plug-in AIVA se tedy stará o interaktivní vizualizaci komponentových aplikací. Informace o zobrazených komponentových aplikacích získává plug-in AIVA ve formě dat ENT meta-modelu. Tedy ENT meta-model je datovým modelem plug-inu AIVA. Části kódů neobsahujících aplikační logiku jsou většinou součástí plug-inu JGraphBasicVisualization. Jde například o datový model uzlů grafu nebo možná překrytí uzlů grafu.

Na obrázku 8.1 je detail vizualizace poskytované plug-inem AIVA. Jsou na něm vyznačeny některé základní prvky a funkčnosti, které plug-in AIVA obsahuje:

- zobrazení komponenty a jejího obsahu (těla) stromovou strukturou (**1**)
- zobrazení „lízátkového“ spojení mezi komponentami (**4**), v článku [ICV11] je použitý termín lollipop
- zobrazení tagů komponenty po kliknutí na její horní panel (**2**)
- zobrazení základních informací o elementech, kterými jsou spojeny komponenty po kliknutí na spojení (**3**)
- zobrazení pomocného náhledu (**5**)

- přepínání a správa použitých category setů komponent (6)
- přepínání layoutů grafu komponent (7)
- zobrazení tagů elementů (8) jako tooltip



Obrázek 8.1: Ukázka prvků AIVA modulu

Nicméně, mezi nedostatky a chyby, které nelze zařadit jako implementovaná rozšíření patří:

- Časté problémy s přepínáním layoutů
  - komponenty jsou zarovnány příliš u krajů
  - vzdálenost mezi komponentami je malá a někdy není vidět vlastní spojení
  - po zvětšení velikosti komponent některým z layoutů (např. *Vertical Partion*) se nelze vrátit k původní velikosti komponent
  - časté chyby za běhu při přepínání layoutu
- Nerozlišování levého a pravého tlačítka myši při kliknutí na spojení nebo komponenty
- Oddalování a přibližování podhledu u rozsáhlých grafů

- Složité odlišování, který z elementů má nebo nemá tagy (pouze metodou postupného zjišťování klikáním na jednotlivé elementy)

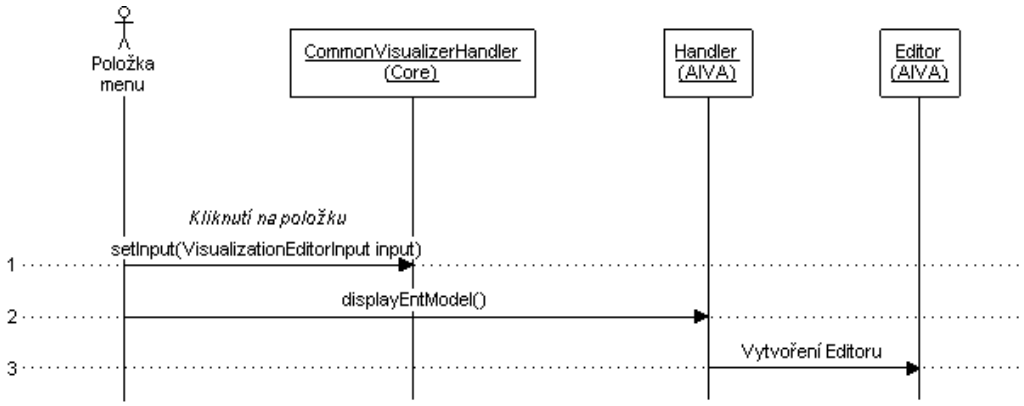
Nedostatky byly opraveny během psaní diplomové práce, avšak nešlo o tak podstatné změny a opravy, aby jim byla věnována další pozornost. I díky těmto malým změnám se o mnoho zlepšila práce s nástrojem.

8.2 Extension point pro Visualizer v plug-inu AIVA

V kapitole 7.2.2 byl popsán poskytovaný extension point pro *Visualizer* plug-iny z pohledu plug-inu *Core*. Tato kapitola popíše tento extension point z pohledu plug-inu AIVA.

Aby mohl plug-in AIVA použít tento extension point, musí být zadány požadované informace. Mimo jména pluginu je nutné zadat třídu handleru umožňující předání dat pro zobrazení z plug-inu *Core*. Předání dat je realizováno přes abstraktní metody třídy zadané v extension point, od níž musí zadaný handler být oděděn.

Sekvenční diagram aktivit požadavku na zobrazení komponentové aplikace z plug-inu *Core* až po zobrazení grafu v plug-inu AIVA je na obrázku 8.2. Z něj je vidět, že předání informací mezi plug-iny je velmi přímočaré. Nejprve je nastaven vstup pro



Obrázek 8.2: Diagram aktivit spuštění okna s grafem komponent

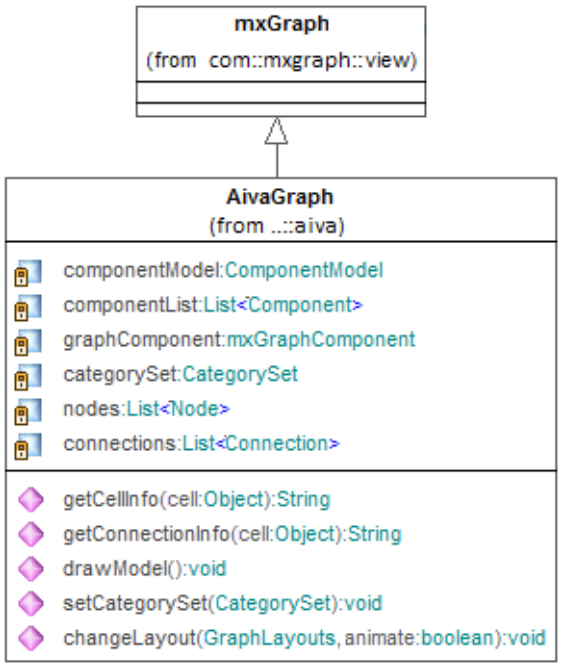
AIVA plug-in (komponentová aplikace popsaná ENT meta-modelem) a poté je poslán požadavek na zobrazení dat. O spuštění editor okna, které obsahuje zobrazenou komponentovou aplikací se již stará AIVA.

Vytvoření editor okna plug-inu AIVA je implementováno pomocí následujícího pseudokódu překryté metody v handleru:

```
public void displayEntModel() throws ComAVException {
      // Ziskani workbench instance aplikace
      var workbench = PlatformUI.getWorkbench();

      var window = workbench.getActiveWorkbenchWindow();
      final IWorkbenchPage page = window.getActivePage();
      try {
            // Otevreni editoru podle jeho identifikatoru
            page.openEditor(input, Editor.ID);
      ...
```

8.3 Základní implementační detaily plug-inu AIVA

Vzniklý editor pak vytváří všechny ostatní prvky plug-inu AIVA. Důležitá je především třída `AivaGraph` pro práci s grafem. Její stručný class diagram je na obrázku 8.3.

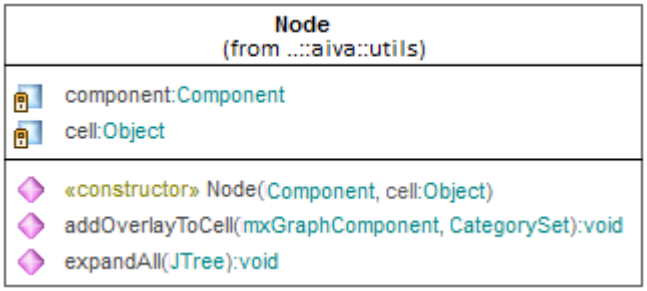


Obrázek 8.3: Class diagram třídy AivaGraph

V metodě `drawModel()` jsou vytvořeny uzly grafu pro komponenty zobrazované aplikace. Po analýze závislostí mezi dostupnými komponentami jsou do grafu přidány i hrany mezi komponentami. Zde je část pseudokódu komentované metody:

```
...
for (Component component : componentList) {
      // Vložení uzlu komponenty do grafu
      Object vertex = insertVertex(null, null, "", 0, 0, 300, 300, "");

      // Vytvoření objektu pro uchováni uzlu grafu
      var node = new Node(component, vertex);

      // Vytvoření a přidání překrytí pro uzel
      node.addOverlayToCell(graphComponent, categorySet);

      nodes.add(node);

      // Analýza spojení a vytvoření objektu pro uchování spojení
      analyzeConnections(component);
}

// Vytvoření hran grafu pro spojení získaná analýzou komponent
for (Connection connection : connections) {
      // Získání uzlu grafu pro local komponentu spojení
      Object local = findVertexToComponent(connection.getLocalComponent());

      // Získání uzlu grafu pro alien komponentu spojení
      Object alien = findVertexToComponent(connection.getAlienComponent());

      // Vložení hrany mezi nalezenými uzly do grafu
      Object edge = insertEdge(null, null, "", local, alien);

      // Uložení vlastní hrany grafu do objektu pro spojení
      connection.setEdge(edge);
}
...
```

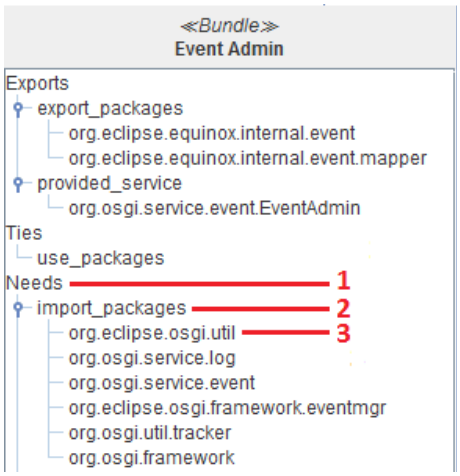
Třída `Node` je mezičlánkem mezi uzlem v grafu (třída `AivaGraph`, respektive `mxCell`) a popisem komponenty (třída `Component` z plug-inu ENTMM). Její jednoduchý class diagram je na obrázku 8.4.



Obrázek 8.4: Class diagram třídy `Node`

Před implementací rozšíření byl v modulu AIVA pro komponenty dostupný pouze základní stromový vzhled jejich těla. Uzel grafu pro komponentu s tímto vzhledem je na obrázku 8.5. V těle komponenty, obsahujícím strom, představují jednotlivé uzly a listy:

- kategorie ze zvoleného category setu (**1**)
- traity komponenty spadající do příslušné kategorie (**2**)
- elementy komponent spadající do příslušné trait (**3**)



Obrázek 8.5: Stromový vzhled uzlu grafu pro komponentu

Uvedené reprezentace komponenty se dosahuje pomocí nastavení překrytí vzhledu uzlu grafu (metoda `addOverlayToCell()` třídy `Node`). Celé překrytí se vytváří pomocí prostředků standardní knihovny `Swing`. Stromová struktura je vytvořena pomocí třídy `JTree`. Zde je komentovaná část pseudokódu metody pro přidání překrytí uzlu:

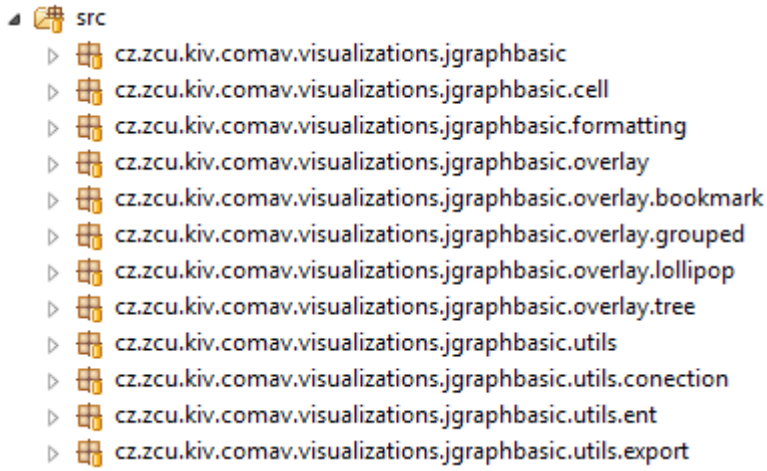
```
public void addOverlayToCell(mxGraphComponent comp, CategorySet catSet) {
      // Třída dědí od JComponent a implementuje mxCellOverlay
      var overlay = new AivaCellOverlay();
      overlay.setLayout(new BorderLayout());

      // Přidání JPanelu s popisem komponenty
      overlay.add(labelPanel, BorderLayout.NORTH);
      // Vytvoření stromu s obsahem komponenty
      var traitsInCategories = catSet.getTraitsInCategories(component);
```

```
var categories = traitsInCategories.keySet();

var root = new DefaultMutableTreeNode("");

// Postupné procházení kategorií komponenty
for (Category category : categories) {
      // Uzel stromu pro kategorii
      var categNode = new DefaultMutableTreeNode(category.getName());
      root.add(categNode);
      ArrayList<Trait> traits = traitsInCategories.get(category);

      // Postupné procházení všech trait kategorie
      for (Trait trait : traits) {
            // Uzel stromu pro trait
            String traitName = trait.getDef().getName();
            var traitNode = new DefaultMutableTreeNode(traitName);
            categNode.add(traitNode);

            Element[] elements = trait.getElementSet();
            //  Postupné procházení všech elementů pro trait
            for (int i = 0; i < elements.length; i++) {
                  var element = elements[i];
                  Tag[] tags = element.getTagset();
                  String tooltip = "";
                  // Postupné procházení všech tagů elementu a
                  // získání textu pro tooltip
                  ...

                  // Uzel stromu pro element s tooltipem
                  var elemNode = new ElementTreeNode(element,
                        traitName, tooltip);

                  traitNode.add(elementNode);
            }
      }
}

// Vytvoření JTree z připravených uzlů
var tree = new Jtree(root);
var scroller = new JscrollPane(tree);
...
// Nastavení překrytí uzlu
graphComponent.addCellOverlay(cell, overlay);
...
```

9 Rozšíření možností plug-inu AIVA

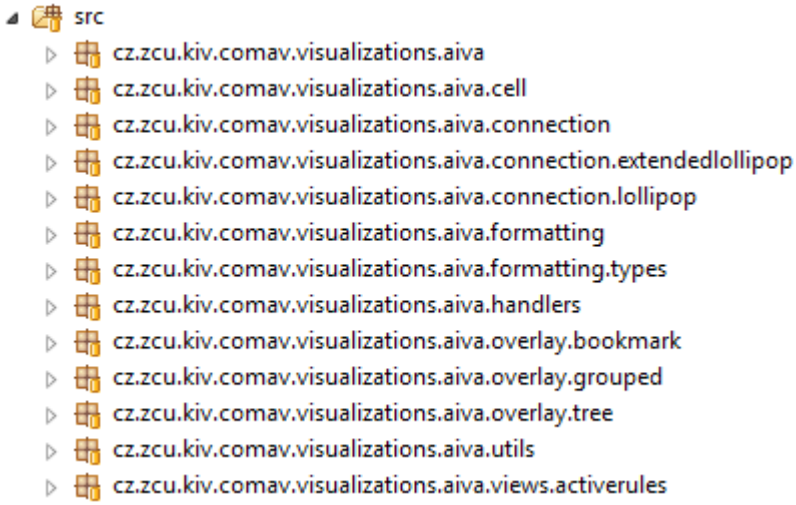
Použité prostředky pro rozšíření plug-inu AIVA tedy vyplynuly z prostředků existující aplikace. Konkrétně se jako vývojová platforma použila Eclipse RCP a grafy komponentových aplikací jsou vykreslovány pomocí knihovny JGraphX. Díky tomu, že nešlo o start nového projektu, odpadl výběr prostředků pro vývoj i návrh architektury aplikace. Naopak přibylo seznámení se se stávající aplikací, platformou Eclipse RCP a knihovnou JGraphX.

Pro srovnání s popisem stavu plug-inu AIVA před diplomovou prací je zajímavé uvést počet přidaných tříd v upravovaných plug-inech. JGraphBasicVisualization obsahuje 39 tříd ve 12 balících (viz obrázek 9.1), z čehož přibližně 18 tříd vzniklo v rámci diplomové práce. Téměř všechny ostatní třídy pak byly upravovány či doplňovány.



Obrázek 9.1: Balíky plug-inu JGraphBasicVisualization

Plug-in AIVA obsahuje 54 tříd ve 13 balících (viz obrázek 9.2), z čehož přibližně 42 tříd vzniklo v rámci diplomové práce. Stejně jako u JGraphBasicVisualization bylo do téměř všech ostatních tříd zasahováno.



Obrázek 9.2: Balíky plug-inu AIVA

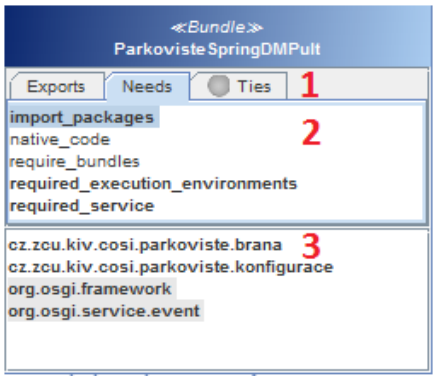
Dále budou popsána jednotlivá rozšíření modulu AIVA, jak byla postupně implementována.

9.1 Alternativní vzhled komponent a jeho přepínání

V plug-inu AIVA existovalo základní stromové zobrazení obsahu komponenty (viz obrázek 8.5). V tělu komponenty jsou zobrazeny kompletní informace podle zvoleného category setu.

Při návrhu alternativního vzhledu byly nejprve uváženy různé možnosti. Zajímavě se jevila možnost zmenšit zobrazené komponenty tím způsobem, že by se zobrazení zaměřovalo na pouze jednu kategorii. Tedy tělo komponenty by obsahovalo pouze informace spadající pod jednu kategorii. V případě, že by uživatel potřeboval zobrazit informace o jiné kategorii, byla by mu nějakým způsobem umožněna změna zobrazené kategorie komponenty.

V článku [FIV04], zabývajícím se vizuálním zobrazením více-komponentových softwarových systémů, je použitý přibližně výše popsaný princip. Po zvážení z něj tedy vychází alternativní vzhled komponent pro plug-in AIVA (obrázku 9.3).

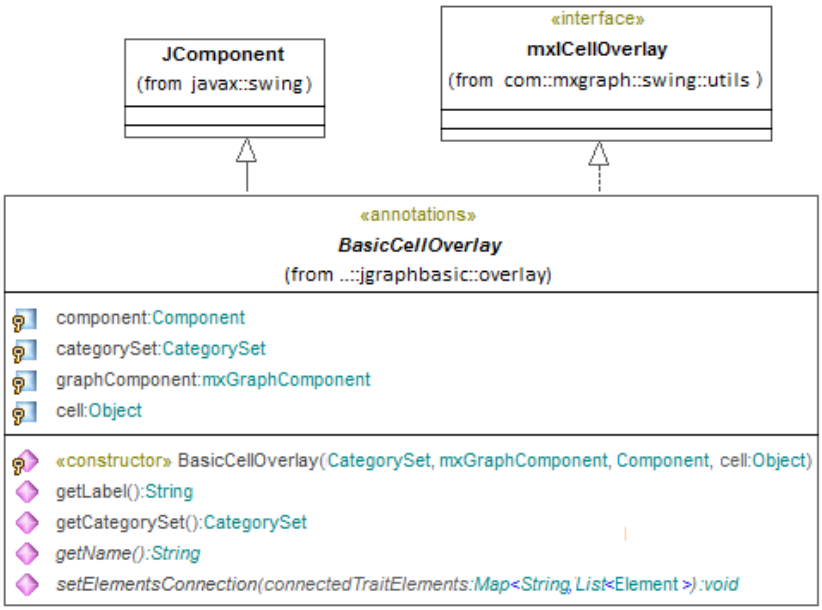


Obrázek 9.3: Alternativní vzhled komponenty

Kategorie komponenty tvoří páskové menu v horní části těle komponenty (oblast **1**). Traity kategorií jsou zobrazené pod páskovým menu (oblast **2**). Jednotlivé traity jde kliknutím myši vybrat a podle zvoleného výběru se v dolní části těla komponenty zobrazí příslušné elementy (oblast **3**).

9.1.1 Abstraktní třída obecného překrytí

Nejprve byla vytvořena abstraktní třída obecného překrytí `BasicCellOverlay`, obsahující společné atributy a metody různých překrytí. Její class diagram je na obrázku 9.4. Přínos třídy se projevil především při implementaci filtrování grafu (kapitola 9.5).



Obrázek 9.4: Class diagram obecného překrytí pro uzly grafu

9.1.2 Implementace alternativního vzhledu komponent

Vzhled komponenty je opět vytvořen pomocí překrytí uzlu komponenty v grafu (viz kapitola 6.3). Výhodou překrytí je, že jednotlivé kategorie komponenty jsou více odděleny. Naopak nevýhodou je, že doplňující informace o prvcích (existují-li traity pro kategorii, existují-li elementy pro trait, je-li element součástí jednoho z existujících spojení mezi komponentami) nejsou na první pohled vidět.

Pro zmírnění uvedených nevýhod bylo do navrženého vzhledu zakomponováno několik vylepšení (všechny jsou vidět na obrázku 9.3). Ta jsou vidět v překrytí pro komponentu na první pohled a jde o :

1. Kategorie, které pod sebou nemají žádné elementy jsou označeny šedou bublinou (na obrázku jde např. o kategorie *Ties*)
2. Traity mající alespoň jeden element jsou napsány tučným písmem (na obrázku jde např. o traity *import_packages*, *required_execution_enviroments*)
3. Elementy, jež nejsou součástí žádného spojení mezi komponentami, mají odlišné pozadí (na obrázku jde např. o elementy *org.osgi.framework*, *org.osgi.service.event*).

   Tuto barvu pozadí lze měnit pomocí kontextového menu grafu (více v kapitole 9.7). Vlastnost se ukázala být velmi užitečná a byla implementovávána i do ostatních překrytí.

Pro zrušení vybraných prvků v překrytí (traity, elementy) bylo při implementování vybráno kliknutí mimo existující uzly a hrany grafu (tato vlastnost byla přidána i do ostatních vzhledů komponent).

Vytvořená třída pro překrytí `ComponentBookmarkOverlay` dědí od třídy obecného překrytí `BasicCellOverlay`. Aby v ní šlo využít již existující třídu `ElementTreeNode` (viz kapitola 8.3), jsou traity a elementy vytvořeny opět pomocí třídy `JTree`. V tomto překrytí budou stromy pro jednotlivé oblasti pouze jednoúrovňové (kořen stromu není viditelný). Díky tomuto rozhodnutí bylo možné v dalším rozšíření (kapitola 9.2) použít stejné prostředky v základním stromovém i v tomto přidaném překrytí.

Díky tomu, že v překrytí nejsou vidět všechny informace o komponentě, ztížila se jeho konstrukce. V překrytí je po spuštění vybrána záložka první kategorie, proto minimálně jeden strom trait musí být ihned vytvořen. Z předpokladu, že trait nebude ve vybrané množině kategorií mnoho, vycházelo rozhodnutí vytvořit stromy trait pro jednotlivé kategorie ihned.

U stromů elementů je tomu jinak. Po inicializaci překrytí není vybrána žádná trait, proto není nutné vytvářet ihned strom elementů. Jejich počet je také alespoň stejný (spíše však větší) jako počet stromů vlastností, protože trait má pod sebou jeden nebo i žádný strom elementů a každá kategorie obsahuje alespoň jednu trait.

Proto jsou stromy elementů vytvářeny dynamicky při přepínání (vybírání) trait. Stromy jsou samozřejmě kvůli snížení režie ukládány do připravené kolekce. Pokud pak v této kolekci existuje strom elementů pro vybranou trait, je použitý znovu a nemusí se vytvářet stále dokola.

Část pseudokódu vytvářející popsané překrytí je níže:

```
// Postupné procházení kategorií komponenty
for (final Category category : categories) {
	categoryTabsOrder.put(index, category);

	// Root strom trait pro kategorii
	var traitRoot = new DefaultMutableTreeNode();
	ArrayList<Trait> traits = traitsInCategories.get(category);

	boolean catHasElements = false;

	// Vytvoření uzlů stromu trait pro kategorii
	for (Trait trait : traits) {
		var traitName = trait.getDef().getName();
		var hasElements = trait.getElementSetLength() > 0;
		var traitNode = new TraitTreeNode(traitName, hasElements);

		traitRoot.add(traitNode);

		if (hasElements)
			catHasElements = true;
	}

	// Vytvoření stromu trait pro kategorii
	var traitsTree = new Jtree(traitRoot);

	// Ikona kategorie
	Icon icon = catHasElements != false ? null : EMPTY_ICON;
	tabbedPane.addTab(category.getName(), icon, traitsTree);

}
```

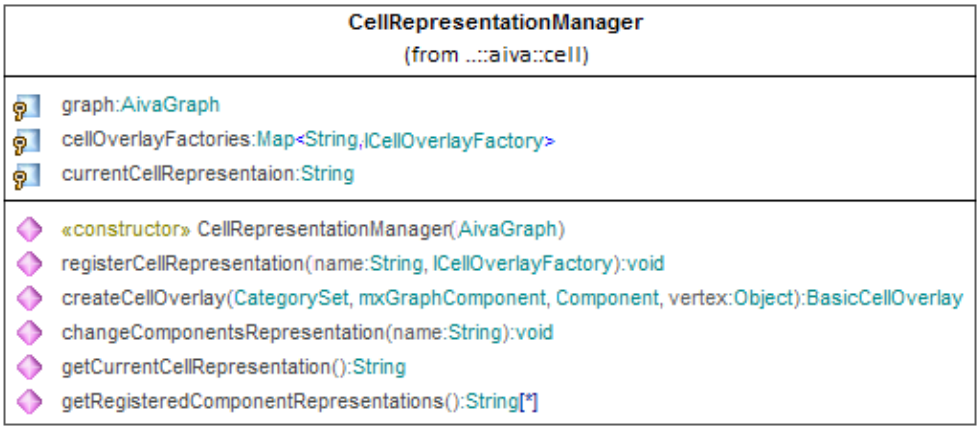
9.1.3 Přepínání vzhledu komponent

Způsob, jakým je umožněno přepnout vzhled komponent, vychází z podobného mechanismu AIVA plug-inu, a sice přepínání layoutů. Přepínání vzhledů komponent je řešeno podobně, tedy také pomocí accordion (rozjíždějícího se) menu umístěného do pravé části okna AIVA nad prvek 7 v horní části obrázku 8.1.

Také bylo nutné vytvořit vazbu mezi nabídkami možných zobrazení komponent v editoru a přepnutím do tohoto zobrazení, což je provedeno odebráním původních překrytí uzlů grafu a poté přidáním nových překrytí.

Jako nejlepší řešení bylo vybráno vytvoření manageru pro registraci možných překrytí. Registrace je umožněna jménem překrytí a instancí třídy vytvářející příslušná překrytí, tedy využití návrhového vzoru Abstract factory. Jednotlivá překrytí ji implementují ve statické metodě, která vytvořenou instanci třídy vrací. Třída dědí od připraveného rozhraní, pomocí jehož metod poté manager pracuje. Díky použití návrhového vzoru Abstract factory se nemusí nikde v kódu objevit řídící struktura `if-else`, která by podle jména určovala, jakou instanci třídy pro překrytí vytvořit.

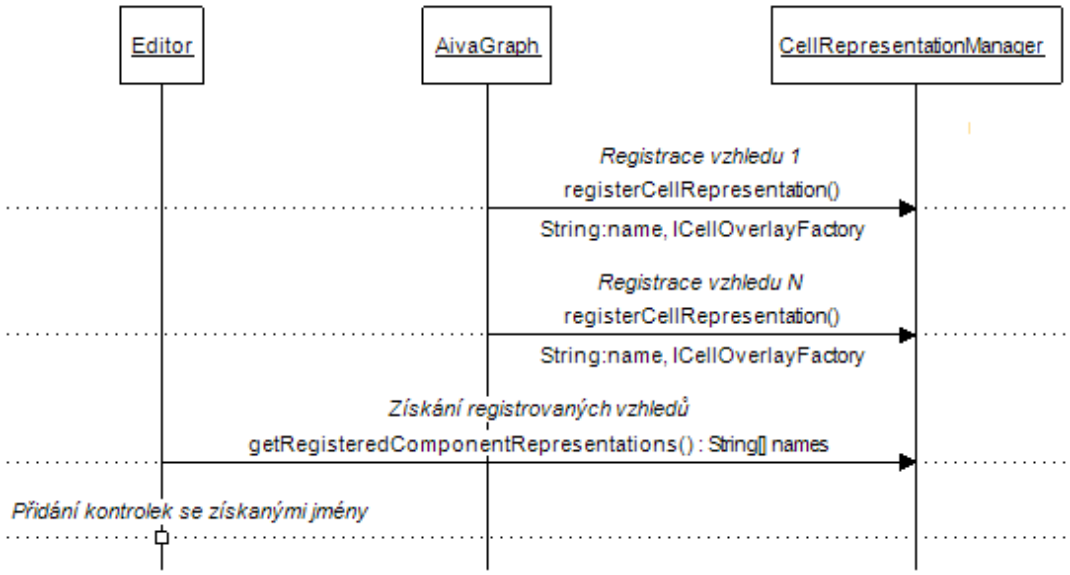
Class diagram manageru je na obrázku 9.5 a class diagram popsaného rozhraní je na obrázku 9.6.



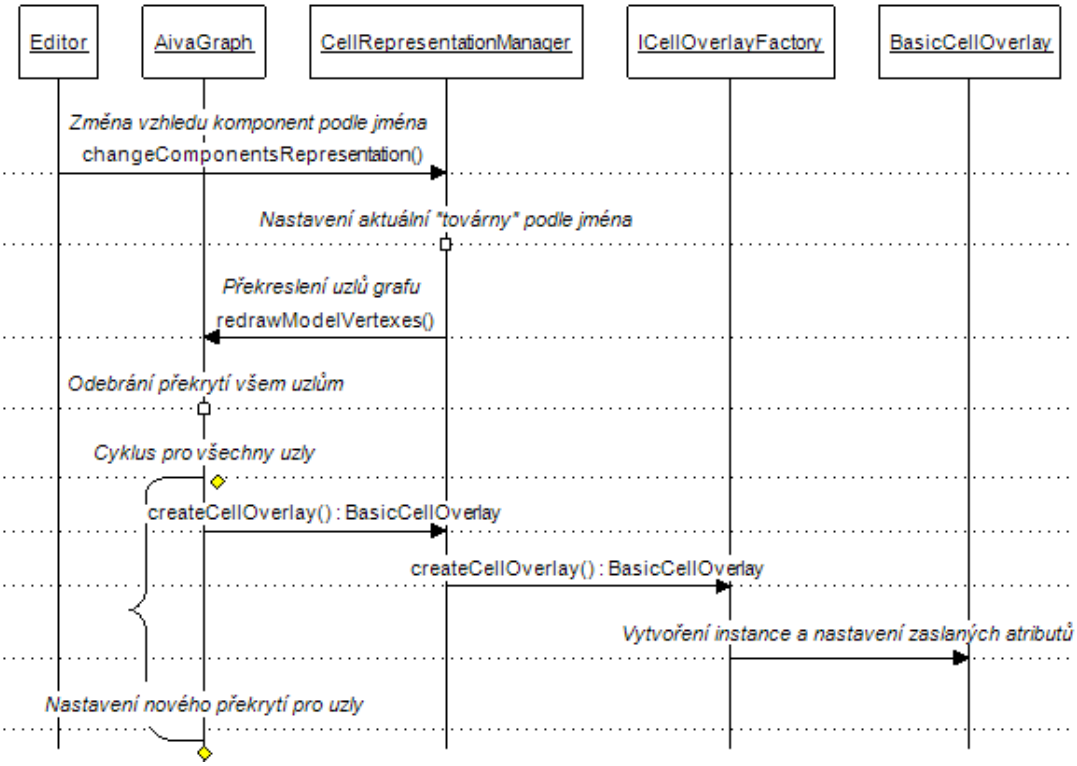
Obrázek 9.5: Class diagram třídy `CellRepresentationManager`

Obrázek 9.6: Class diagram rozhraní `ICellOverlayFactory`

Princip registrace překrytí při startu programu je vysvětlen sekvenčním diagramem na obrázku 9.7. Následné přepnutí vzhledu komponenty je vysvětleno sekvenčním diagramem na obrázku 9.8.



Obrázek 9.7: Sekvenční diagram pro registraci vzhledů komponent



Obrázek 9.8: Sekvenční diagram přepnutí vzhledu komponenty

Manager je tedy řídícím prvkem při změně vzhledu komponent. Třída grafu se stará o možná překrytí pouze při jejich registraci, která probíhá v konstruktoru, a poté se již nemusí starat o to, jaká třída je aktuálně používána.

9.2 Zvýraznění elementů po kliknutí na spojení

Pro lepší práci s grafem komponent by bylo vhodné získávat informace o spojení. Konkrétně jaké komponenty jsou spojeny a jakými elementy jsou spojeny. Tyto informace jsou v grafu obsaženy, ale nelze je snadno dohledat.

Popisované rozšíření implementuje popsaný problém. Po kliknutí na hranu v grafu jsou vyznačeny spojené komponenty a také jejich elementy, jimiž jsou spojeny. Jde tedy o interaktivní zobrazení informací. Za způsob zobrazení toho, jaké komponenty a jejich elementy patří k vybranému spojení, se vybralo barevné vyznačení.

Pro komponenty, respektive uzly grafu, se nastaví barevný rámeček a pozadí v náhledu grafu. Pro elementy se nastavuje pozadí textu v aktuálním vybraném vzhledu komponent. Je zde využívána implementace elementů trait jako stromu, popsaná v předchozí kapitole. Na obrázku 9.9 je vzhled vyznačené komponenty po kliknutí na její spojení.

Obrázek 9.9: Překrytí ComponentBookmarkOverlay se zvýrazněním

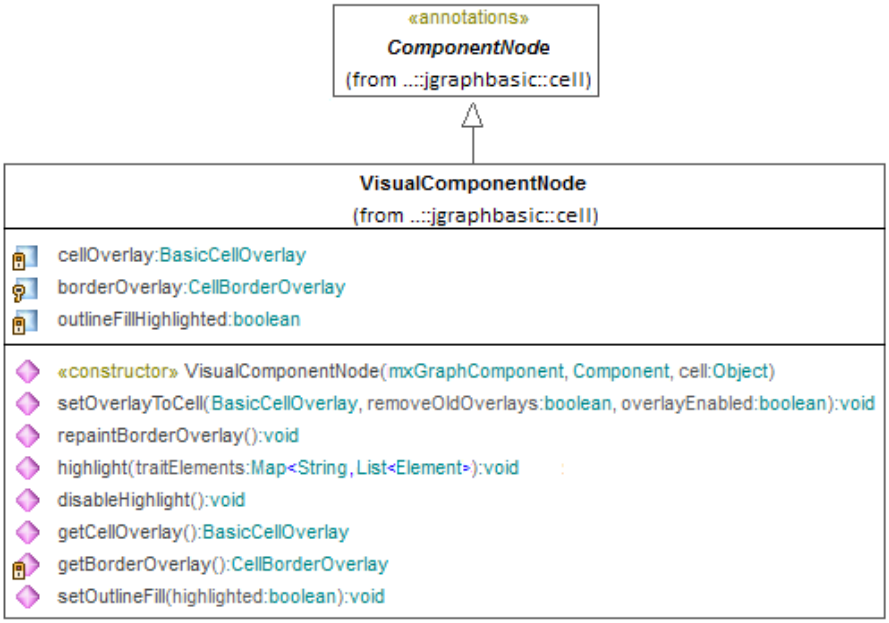
9.2.1 Třída pro vyznačení uzlů komponent

Pro vyznačení komponent je nutné pracovat s jejich překrytími. Třída `AivaGraph` však

uchovává pouze vytvořené uzly grafu, které samy nemají informaci o jejich přiřazeném překrytí.

Než upravovat třídu `Node` (viz kapitola 8.3), bylo raději vybráno řešení vytvořit třídu použitelnou pro vyznačení uzlů komponent a jejich těla. Díky tomu zůstala třída `Node` datovým mostem mezi uzlem grafu a komponentou.

Pro práci s překrytím uzlu grafu se pak již používá třída `VisualComponentNode`, dědící od třídy `ComponentNode` (což je přejmenovaná původní třída `Node`). `VisualComponentNode` přidává a odebírá překrytí pro uzel, předává příkazy pro vyznačení překrytí a také upravuje pozadí uzlů v náhledu grafu (viz stručný class diagram třídy na obrázku 9.10).



Obrázek 9.10: Class diagram třídy `VisualComponentNode`

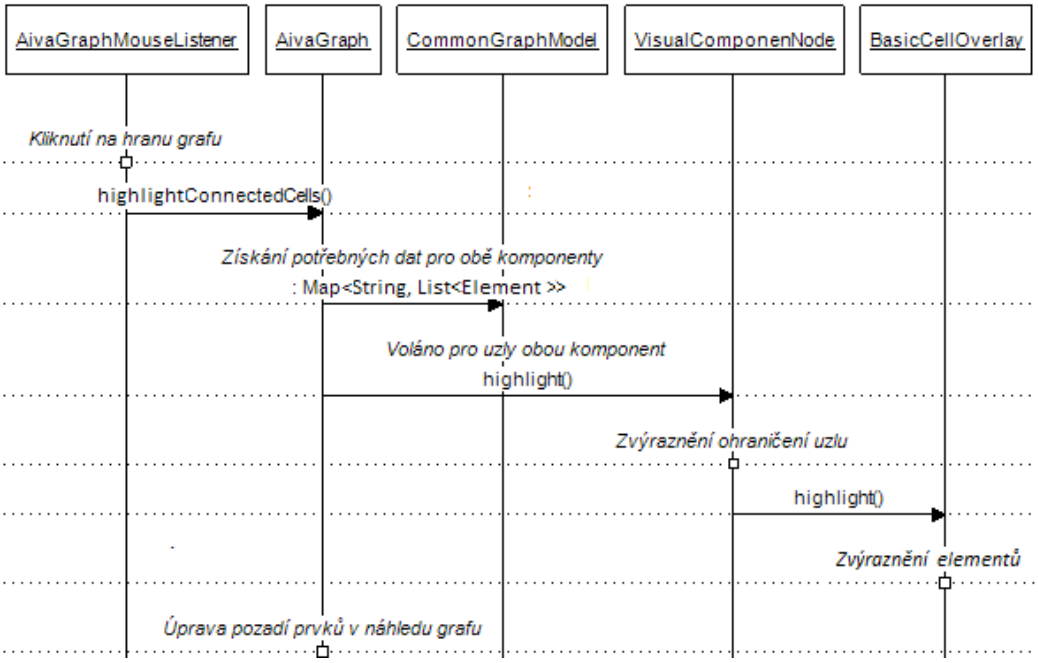
9.2.2 Datový model grafu

V aplikaci se často objevovaly operace pro získání komponenty pro uzel grafu, získání příslušné instance `VisualComponentNode` pro komponentu i pro uzel grafu a jiné komplexnější operace. Například pro toto rozšíření bylo výhodné získat elementy příslušející spojení mezi komponentami.

Pro tyto specifické operace a uchování komponent, instancí `VisualComponentNode` a `Connection` (spojovací články mezi ENT modelem a prvky grafu) byla vytvořena třída `CommonGraphModel`. Díky ní došlo k logickému sjednocení dat grafu, třída `AivaGraph` byla dosti zpřehledněna a došlo k oddělení dat od jejich zobrazení.

9.2.3 Implementace zvýraznění elementů po kliknutí na spojení

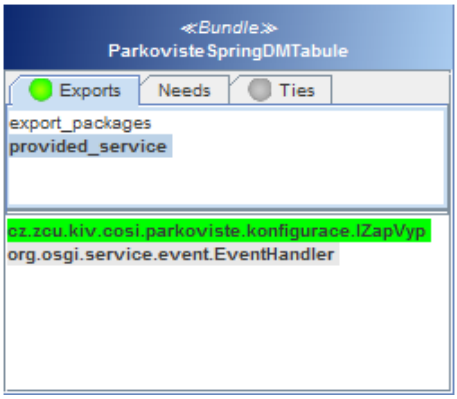
Po kliknutí do grafu se pomocí prostředků knihovny JGraphX zjistí, zda graf má na souřadnicích kliknutí nějaký prvek. Pokud je hranou, je třídě `AivaGraph` předáno spojení, z kterého získá komponenty spojení. Dále je pro obě komponenty získána mapa trait a jejich elementů patřící spojení. Příslušným instancím `VisualComponentNode` pro komponenty jsou pak ve volání metody pro zvýraznění spojení předány nalezené mapy elementů. Ta se pak stará o zvýraznění rámu kolem uzlu (samostatné jednoduché překrytí pouze pro rám kolem uzlů) a předává parametry ve volání zvýraznění pro překrytí uzlu. Popsaný postup je pro znázornění popsán stručným sekvenčním diagramem na obrázku 9.11.



Obrázek 9.11: Sekvenční diagram zvýraznění elementů spojení

Zvýraznění elementů spojení je umožněno rozšířením třídy obecného překrytí `BasicCellOverlay` (kapitola 9.1.1). Instance třídy `ComponentBookmarkOverlay` je volána překrytou metodou `highlight()` (viz obrázek 9.11) a je jí předána mapa elementů trait, které se mají zvýraznit. Protože si třída uchovává informace o vazbě mezi traity a jejich uzly ve trait stromech, a také příslušným stromem elementů pro trait, je nalezení zvýrazňovaných uzlů ve stromech snadné. Nalezeným uzlům se nastaví příznak, že se mají zvýraznit a samotné zvýraznění se provede pomocí nastaveného rendereru stromu `ComponentTreeRenderer`. Ten v metodě pro vykreslení uzlu stromu nastaví patřičné pozadí, pokud má uzel nastavený příslušný příznak.

Z důvodu přehlednějšího zobrazení výsledku zvýraznění, je všem záložkám obsahujícím traity spojení (respektive elementy), přidána zelená ikona a první taková záložka je vybrána (viz obrázek 9.35).



Obrázek 9.12: Zvýraznění elementů v překrytí

Komentovaná metoda `highlight()` vytvořeného alternativního překrytí z kapitoly 9.1 je níže:

```
public void highlight(String traitToHighlight, Element elmToHighlight) {
	// Procházení všemi kategoriemi, resp. Trait uzly kategorií
	foreach (Map<String, TreeNodeHighlight> traitNodes in catTraitNodes) {
		var traitNode = traitNodes.get(traitToHighlight);
		if (traitNode == null)
			continue;
```

```
            // Zvýraznění trait uzlu
            traitNode.setIsHighlighted(true);

            // Získání stromu elmentů pro trait
            TraitElementsTree elemTree = getElementsTree(traitName);

            // Získání elementu pro zvýraznění
            Map<Element, ElementTreeNode> elmNodes = elemTree.getNodes();
            TreeNodeHighlight elementNode = elmNodes.get(elmToHighlight);
            elementNode.setIsHighlighted(true);

            // Výběr příslušné záložky kategorie a natavení ikony
            tabbedPane.setSelectedIndex(i);
            tabbedPane.setIconAt(i, HIGHLIGHTTED_ICON);

            break;
        }
}
```

9.3 Zvýraznění použití elementů

Pro usnadnění práce s grafem komponent by bylo vhodné získávat informace o spojení elementů. Konkrétně s jakou komponentou (a jejím elementem) je element spojen. Jedná se o podobnou situaci jako v předchozí kapitole. Rozdílem je, že získání informace probíhá přes element a místo spojení.

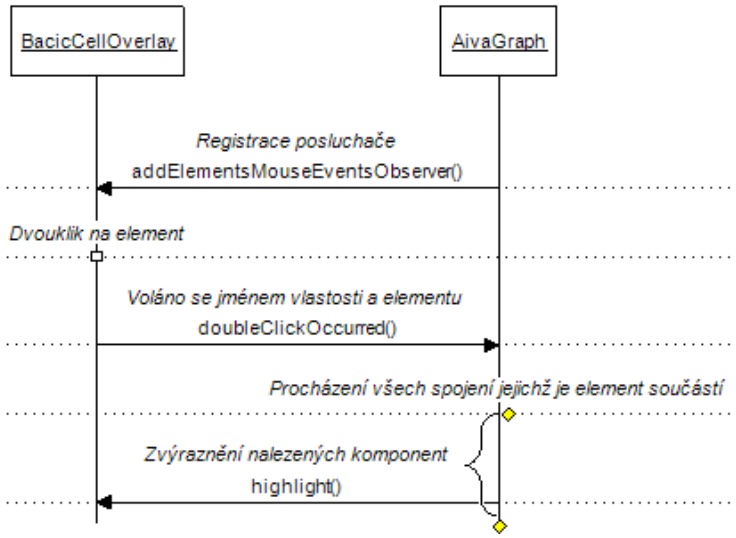
Toto je jednoduché rozšíření plug-inu AIVA, které navazuje na kapitolu 9.2. Po dvojkliku na požadovaný element jsou vyznačeny komponenty a jejich elementy s ním spojené. Pro vyznačení je opět používán barevný rámeček kolem uzlů komponent a barevné pozadí pro elementy v těle komponenty.

9.3.1 Implementace zvýraznění použití elementů

Kliknutí na element je obslouženo přímo u jednotlivých překrytí. Překrytí samo však samozřejmě neřídí průběh zvýraznění. Třída `AivaGraph` implementuje připravené rozhraní `IOverlayElementsMouseEventsObserver` a poté se u překrytí registruje jako posluchač událostí kliknutí. Překrytí pak registrovaným posluchačům oznamuje dvojklik na uzly elementů (implementace návrhového vzoru Observer).

Třída `AivaGraph` pak stejně jako v předchozí kapitole řídí průběh zvýraznění. Zde je

situace o to snadnější, že je znám element a trait komponenty, kde dvojklik proběhl. Pro něj jsou pak nalezena všechna spojení, jejichž je součástí. Překrytím komponent těchto spojení je pak již snadno nastaveno zvýraznění podobně jako v předchozí kapitole. Sekvenční diagram popisující popsaný průběh zvýraznění je na obrázku 9.13.



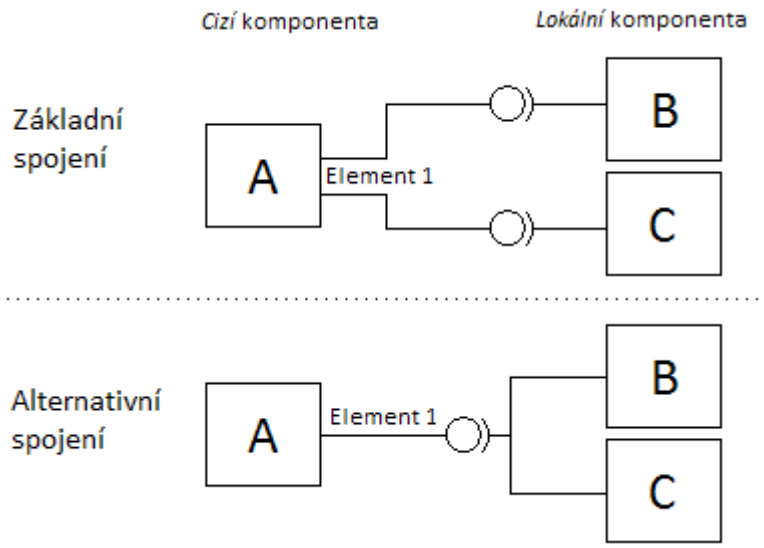
Obrázek 9.13: Sekvenční diagram průběhu zvýraznění po dvojkliku na *element*

9.4 Alternativní vzhled spojení mezi komponentami a jeho přepínání

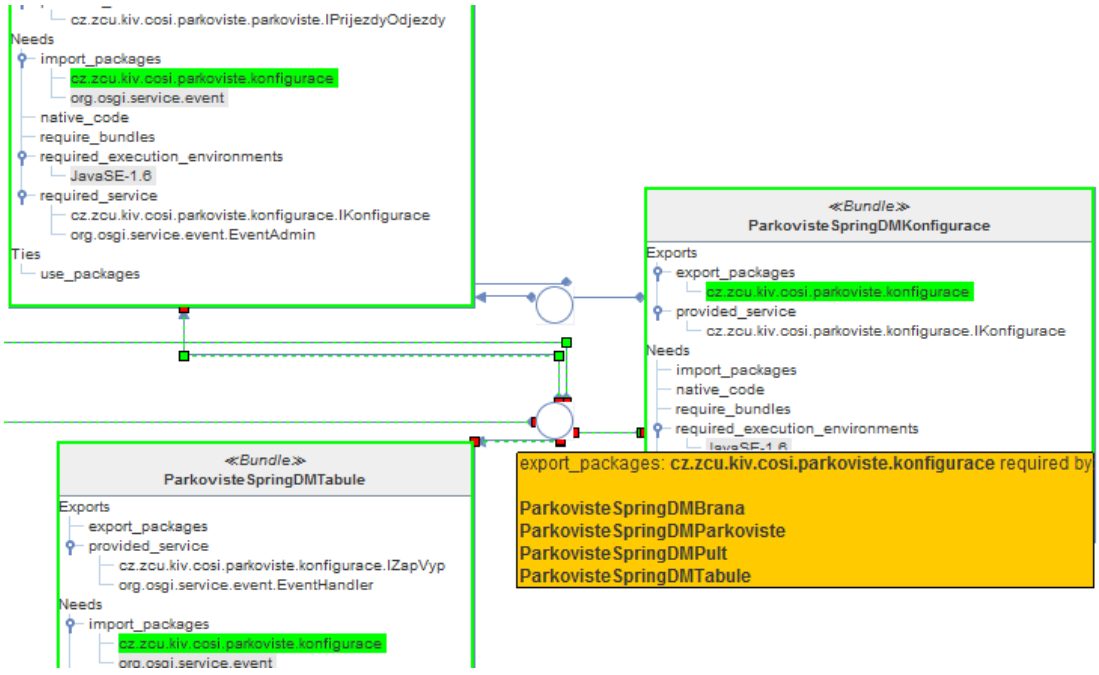
V rozsáhlejších grafech obsahujících mnoho hran mezi uzly se někdy lze hůře orientovat. Z toho důvodu bylo navrženo, aby se vytvořilo úspornější a přehlednější spojení mezi komponentami. Z analýzy možných řešení vzešel návrh sdružovat hrany spojení *cizí* komponenty do jedné hrany, jež by vedla do „lízátka“ spojení. Až z něj by potom vycházely jednotlivé hrany spojení pro *lokální* komponenty. Sjednocené hrany jsou u *cizí* komponenty, protože ta ve spojení plní roli poskytovatele. Proto stačí pro každý poskytnutý element jedna hrana.

Jak se obecně změní struktura grafu, je znázorněno na obrázku 9.14. Implementace narazila na několik problémů (popsané v následující kapitole), a proto nevypadá

výsledné spojení přesně podle popsaného návrhu. Na obrázku 9.15 je výsledný vzhled implementovaného spojení.



Obrázek 9.14: Porovnání základního a alternativního vzhledu spojení komponent

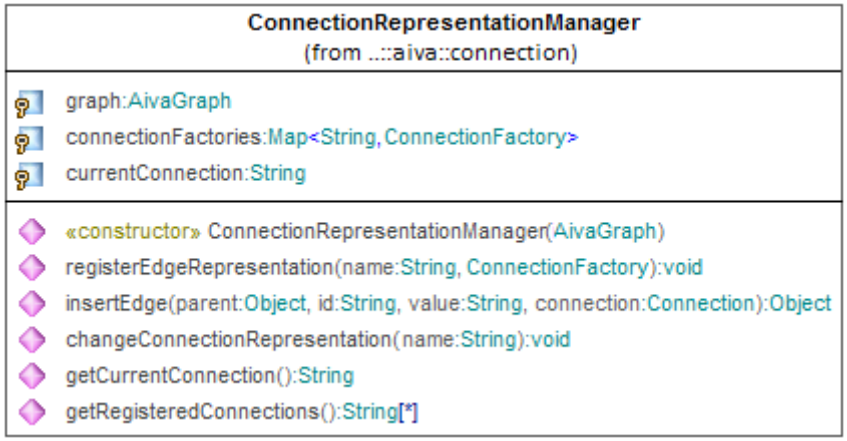


Obrázek 9.15: Implementovaný vzhled alternativního spojení

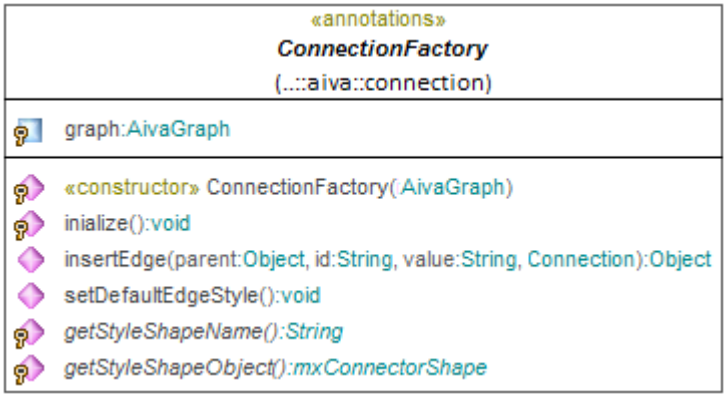
9.4.1 Přepínání vzhledu spojení obecně

Princip přepínání vzhledu komponent z kapitoly 9.1.3 se prokázal jako dobrý a přehledný, proto i zde byla snaha navržený princip duplikovat. U možných vzhledů komponent se dostupná překrytí registrovala, aby později byla vytvářena pomocí připravených továrních tříd.

Pro práci se vzhledy spojení mezi uzly grafu se tedy také připravil manager `ConnectionReprezantationManager` umožňující registraci vzhledů spojení. Class diagram manageru je na obrázku 9.16. O vytváření spojení a nastavení jejich vzhledu se pak starají implementace abstraktní továrny `ConnectionFactory,` jejíž class diagram je na obrázku 9.17).



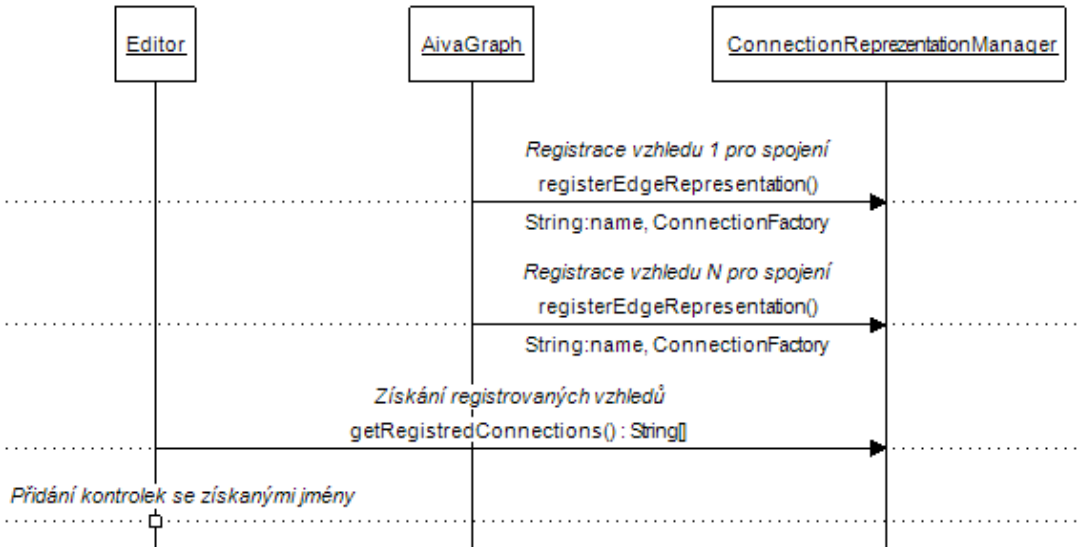
Obrázek 9.16: Class digram třídy `ConnectionReprezentationManager`



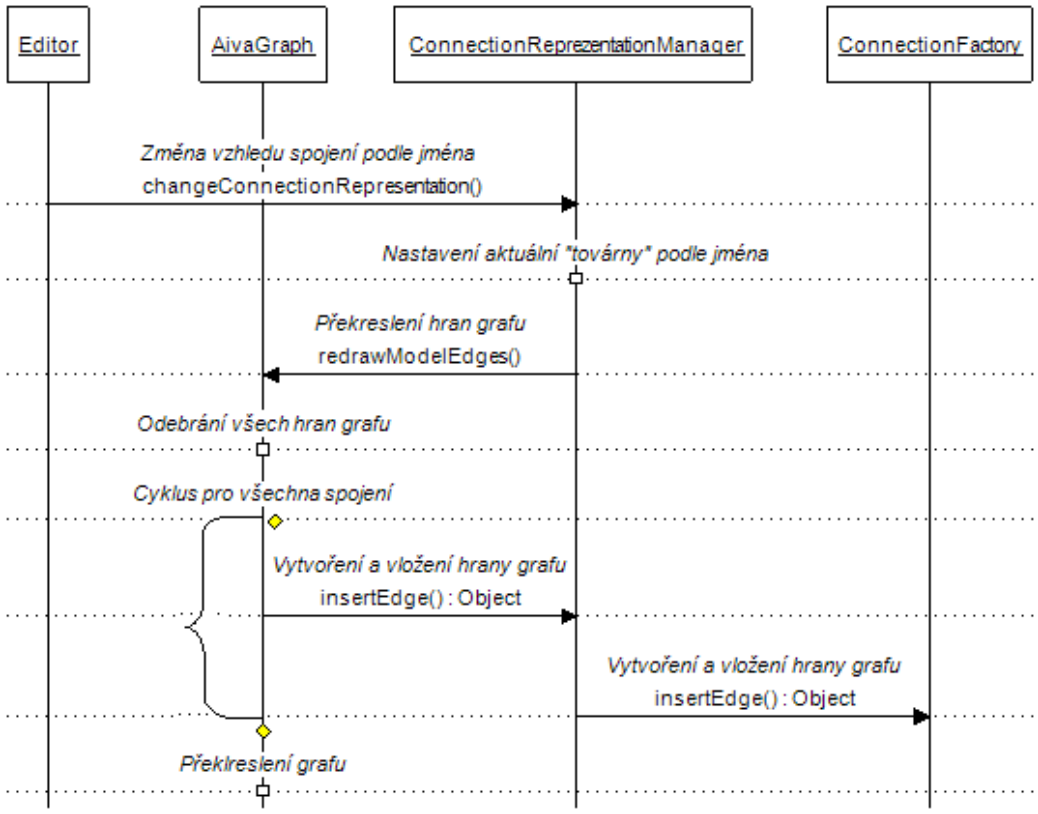
Obrázek 9.17: Class diagram třídy `ConnectionFactory`

Princip registrace vzhledu spojení při startu programu je vysvětlen sekvenčním

diagramem na obrázku 9.18. Následné přepnutí vzhledu spojení je vysvětleno sekvenčním diagramem na obrázku 9.19. Princip je také podobný přepnutí vzhledu komponent. Ovšem situace je zde trošku složitější, což je přiblíženo v kapitole 9.4.6.



Obrázek 9.18: Sekvenční diagram pro registraci vzhledů spojení



Obrázek 9.19: Sekvenční diagram změny vzhledu spojení

9.4.2 Implementace alternativního vzhledu spojení

Na obecném návrhu vzhledu spojení (viz obrázek 9.14) je vidět, že hrana se po dosažení poskytovaného elementu rozdvojuje a vede dál k příslušným komponentám využívajícím toto rozšíření. Nápadů, jak tento vzhled implementovat, bylo několik:

1. rozdvojit hranu po určité vzdálenosti (logicky po dosažení „lízátka“)
2. vést dvě hrany po stejné trase do určité vzdálenosti (dosažení „lízátka“)
3. vytvořit přímo pomocný uzel pro „lízátko“ a poté klasicky vytvořit hrany mezi uzly

Prostředky JGraphX však žádným jednoduchým způsobem neumožňují implementovat první dva nápady. Z toho důvodu byla vybrána třetí možnost. Při přepínání vzhledu spojení na toto alternativní a zpět však vzniklo mnoho problémů:

1. různý počet uzlů grafu po přepnutí
2. různý počet hran grafu po přepnutí
3. definování pevného vztahu mezi uzlem komponenty a „lízátkovými“ uzly poskytovaných elementů pro pohyb s uzlem komponenty
4. složitější konstrukce grafu pro alternativní vzhled spojení
5. složitější zvýrazňování po kliknutí na element nebo hranu spojení
6. problémy s layoutem grafu díky novému „lízátkovému“ uzlu

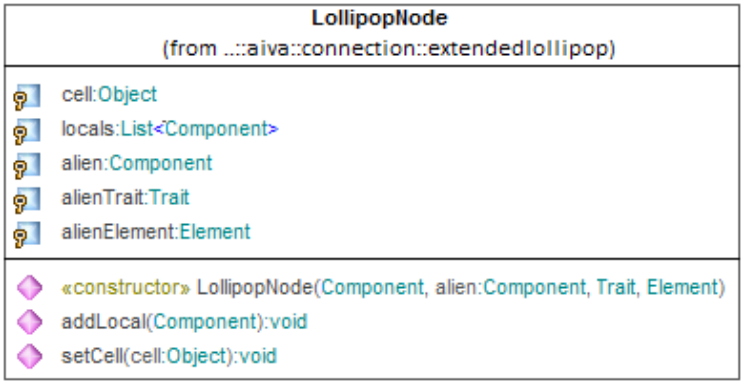
„Lízátkový“ uzel je do grafu vložen pomocí samostatného uzlu, který má jistě vztah k uzlu komponenty. To v zásadě znamená, že pokud se v grafu posune uzel komponenty, měly by se posunout i všechny její „lízátkové“ uzly. A to tak, aby zůstaly ve stejné vzdálenosti od uzlu komponenty jako před posunem. „Lízátkové“ uzly jsou tedy podřízené uzlům komponent. Pro tuto situaci šlo dobře použít prostředků knihovny JGraphX, kdy se „lízátkové“ uzly vytváří jako porty, kterým je možné nastavit rodičovský uzel. Díky tomu bude posun uzlů grafu s „lízátkovými“ uzly fungovat podle požadavků.

Pro opravu layoutu grafu při přepínání na alternativní vzhled spojení se do grafu vkládá hrana mezi oba uzly spojených komponent (tedy tak, jak to funguje v klasickém vzhledu spojení). Tyto hrany jsou však nastavené tak, aby nebyly viditelné, a proto nezasahují do vytvořené struktury grafu, avšak slouží velmi dobře pro layout grafu.

Vzhled „lízátkového“ uzlu se také liší od původní myšlenky. Vzhled uzlu byl zredukován na kroužek, oproti plánovanému lízátku (viz obrázek 9.14). Protože „lízátkový“ uzel se v grafu může libovolně pohybovat (vlastní posun nebo změna layoutu), můžou nastat situace, kdy hrana z lokální komponenty (která by měla mít v lízátku půlkruh) vede do „lízátkového“ uzlu z různých směrů. Uzel by proto musel hlídat, z jaké strany do něj vede hrana z lokální komponenty spojení, a podle toho by se otáčel. Po uvážení byl tedy vzhled „lízátkového“ uzlu pro alternativní spojení zjednodušen, aby se nemuselo hlídat toto otáčení uzlu.

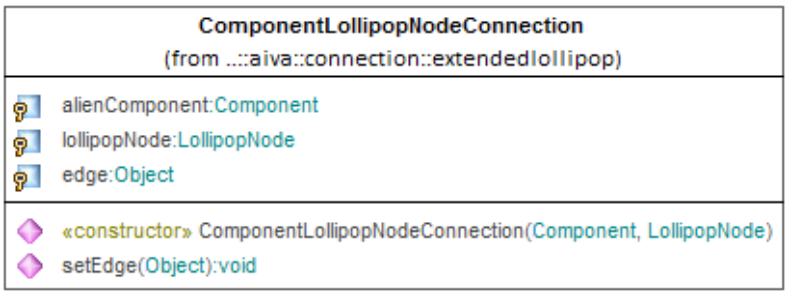
9.4.3 Datové modely pro alternativní vzhled spojení

Stejně jako existuje pro komponenty a jejich uzly grafu datový model v podobě třídy `ComponentNode`, bylo příhodné vytvořit datový most mezi „lízátkovými“ uzly a abstraktním objektem reprezentujícím poskytovaný element komponenty. Aby model plnil funkci spojujícího článku, musí tento abstraktní objekt obsahovat poskytovaný element, jeho trait, komponentu a na druhé straně také seznam všech lokálních komponent spojení. Stručný class diagram vytvořené třídy `LollipopNode` sloužící jako datový model mezi popsaným abstraktním objektem a uzlem grafu je na obrázku 9.20.



Obrázek 9.20: Class diagram třídy `LollipopNode`

Jak již bylo zmíněno, datový model pro hranu spojení komponent v grafu obsahuje reference na *lokální* a *cizí* komponenty a samotnou hranu grafu (viz kapitola 8.3). Používán je především při zvýrazňování komponent. Pro navržené spojení však tento datový model nestačí. Pro hranu mezi cizí komponentou a jejím „lízátkovým“ uzlem bylo třeba navrhnout datový model, který dokáže uchovat potřebná data. Těmi jsou cizí komponenta, datový model „lízátkového“ uzlu (třída `LollipopNode`) a vlastní hrana grafu. Stručný class diagram navržené třídy `ComponentLollipopNodeConnection` je na obrázku 9.21.

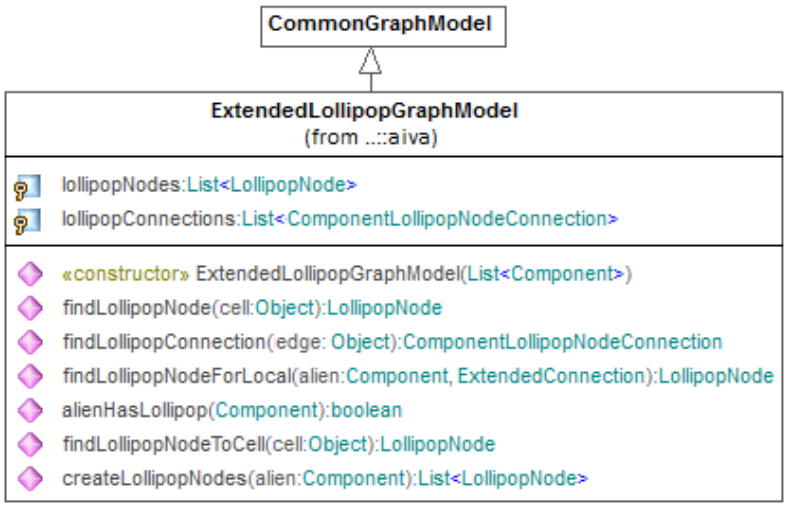


Obrázek 9.21: Class diagram třídy `ComponentLollipopNodeConnection`

9.4.4 Rozšíření datového modelu grafu

Dále bylo třeba, aby model grafu uchovával instance dvou popsaných tříd. Jako logické řešení se nabídlo rozšíření původní třídy pro graf `CommonGraphModel` (viz kapitola 9.2.2). Class diagram třídy `ExtendedLollipopGraphModel` rozšířeného modelu grafu je na obrázku 9.22 a jak je vidět, obsahuje metody pro získání modelů pro vlastní prvky grafu, čímž rozšiřuje možnosti původního modelu grafu.

Jak je patrné z návrhu alternativního spojení, při přepínání vzhledu dochází k přidávání „lízátkových“ uzlů a odebrání a přidání nových hran do grafu. Při přepnutí z alternativního vzhledu spojení na základní naopak dochází k odebrání vytvořených uzlů a úpravě hran grafu.

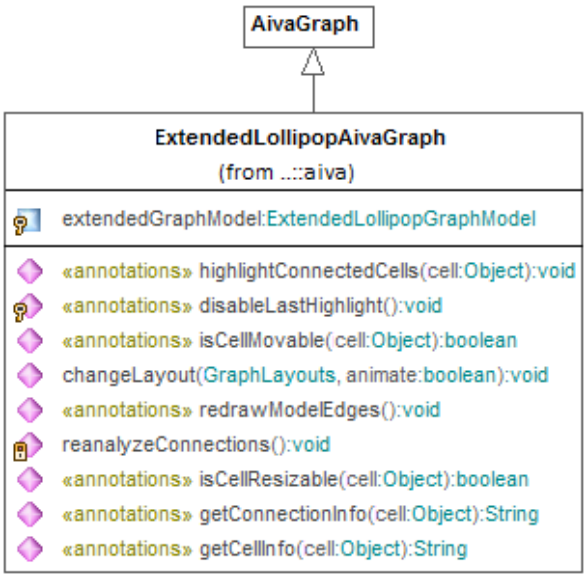


Obrázek 9.22: Class diagram třídy `ExtendedLollipopGraphModel`

9.4.5 Rozšíření grafu

Rozlišování s jakým vzhledem spojení (a tím i s jaký modelem grafu) se právě pracuje by zkomplikovalo třídu `AivaGraph` pro práci s grafem. Architektura grafu se pro obě implementované možnosti vzhledu liší a díky tomu je také odlišný postup při zvýrazňování komponent a jejich vnitřních prvků. Ať už po kliknutí na hranu spojení nebo po dvojkliku na element komponenty.

Z těchto důvodů byla vytvořena třída `ExtendedLollipopAivaGraph`, která je přímo navržena pro práci s implementovaným alternativním vzhledem spojení. Její stručný class diagram je na obrázku 9.23. Princip jejího fungování je takový, že překrývá metody, které se liší pro implementované vzhledy spojení. Pokud je zvolený základní vzhled spojení, pouze volá rodičovskou metodu. Díky tomuto postupu se do základní třídy pro graf nemuselo kvůli alternativnímu vzhledu spojení nijak moc zasahovat a samotná třída si jiných postupů není vědoma.

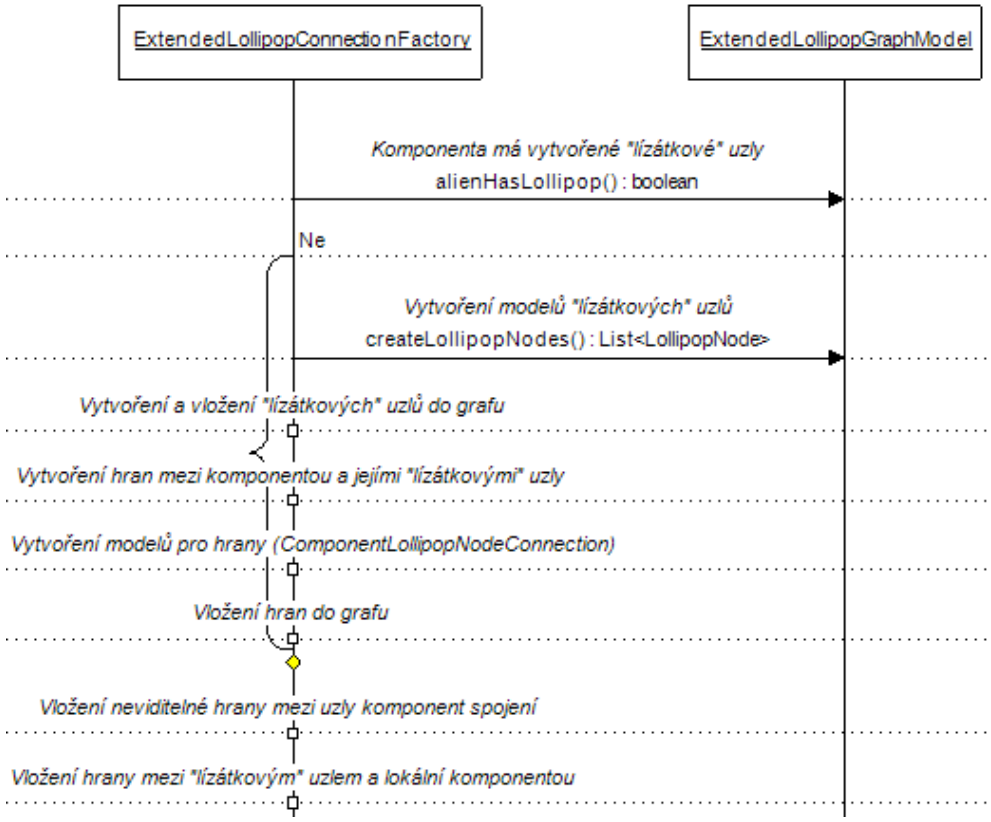


Obrázek 9.23: Class diagram třídy `ExtendedLollipopAivaGraph`

9.4.6 Přepnutí vzhledu spojení ze základního na alternativní

Ze základního popisu přepnutí vzhledu spojení je patrné, že po odebrání všech hran jsou hrany znovu postupně vkládány pomocí manageru (viz obrázek 9.19). Následující popis naváže přibližně na tuto část kódu.

Obrázek 9.24 se sekvenčním diagramem popisuje vložení jedné hrany mezi komponentami pro alternativní spojení. Je vidět, že pro cizí komponenty vkládaného spojení jsou nejprve vytvořeny všechny „lízátkové“ uzly, vloženy hrany mezi nimi a cizí komponentou a také jsou vytvořeny datové modely pro tyto objekty. Až poté je vložena hrana mezi lokální komponentou spojení a příslušným „lízátkovým“ uzlem. Tedy tím, který představuje poskytnutí elementu, jenž využívá lokální komponenta. Z důvodu odstranění problémů s layoutem grafu (popsán v úvodu této kapitoly) se ještě vkládá neviditelná hrana mezi oběma uzly komponent spojení. Uvedený postup se pak opakuje pro všechna spojení grafu.



Obrázek 9.24: Sekvenční diagram vložení hrany mezi komponentami pro alternativní spojení

Komentovaná část metody třídy `ExtendedLollipopConnectionFactory` pro přidání hrany je uvedena níže:

```
public Object insertEdge(Object parent, String id, String value,
            Connection con) {
      Component local = connection.getLocalComponent();
      Component alien = connection.getAlienComponent();

      Object source = model.findVertexToComponent(local);
      Object target = model.findVertexToComponent(alien);

      // Cizí komponenta zatím nemá vytvořeny žádné lollipop uzly
      if ((model.alienHasLollipop(alien)) == false) {
            // Vytvoření modelu lollipop uzlů podle spojení komponenty
            List<LollipopNode> nodes = model.createLollipopNodes(alien);
            // Y offset pro umístění lollipop uzlu
            double yOffset = 1.0 / (nodes.size() + 1);

            // Procházení vytvořených lollipop  modelů
            for (int i = 0; i < nodes.size(); i++) {
                  var node = nodes.get(i);

                  // Vytvoření geometrie a uzlu grafu pro lollipop model
```

```
                    var geom = node.setDefaulCellGeometry((i + 1) * yOffset);
                    var port = new mxCell(null, geom, DEFAULT_STYLE);

                    // Nastavení překrytí lollipop uzlu
                    graphComp.addCellOverlay(port, new LollipopOverlay());

                    // Přidání lollipop uzlu do grafu
                    Object vertex = graph.addCell(port, target);

                    // Nastavení reference uzlu do modelu lollipop uzlu
                    node.setVertex(vertex);

                    // Vytvoření modelu hrany mezi lollipop uzlem a cizí
                    // komponentou a vložení hrany do grafu
                    var con = new ComponentLollipopNodeConnection(alien,
                                node);
                    con.setEdge(graph.insertEdge(...));
                }
            }

            // Získání lollipop modelu a uzlu z lokální komponenty
            LollipopNode nodeModel = model.findLollipopNodeForLocal(alien, con);
            Object ver = nodeModel.getVertex();

            // Přidání neviditelné hrany mezi lokální a cizí komponentou pro
            // odstranění problému s layoutem grafu
            Object edge = graph.insertEdge(parent, null, null, source, target);
            ((mxCell) edge).setVisible(false);

            // Vytvoření hrany mezi lollipop uzlem a lokální komponentou
            return graph.insertEdge(parent, id, value, source, ver, "");
        }
```

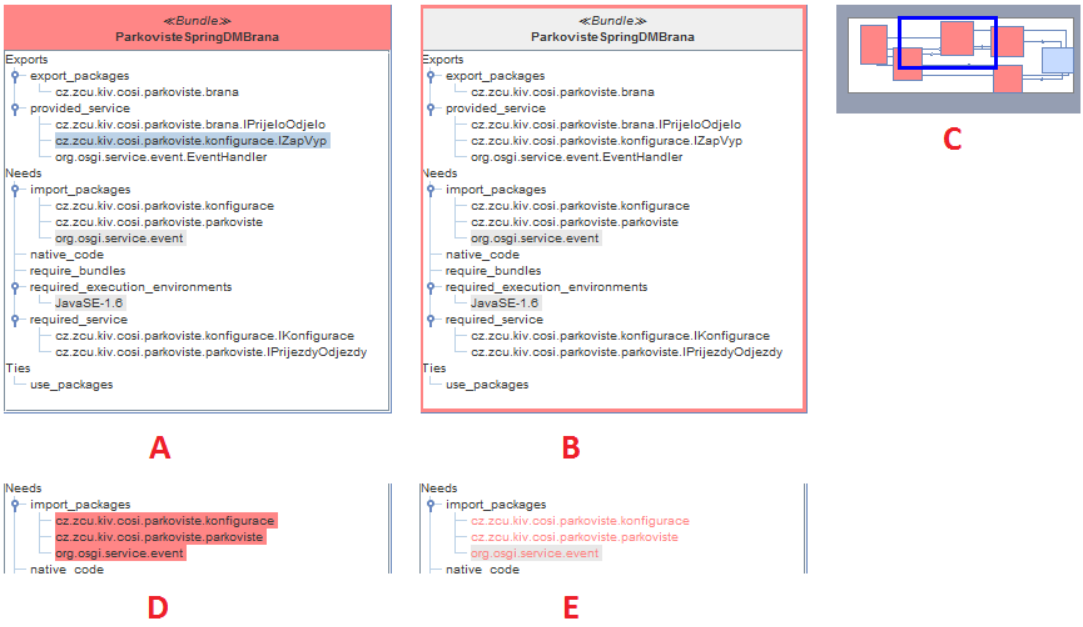
9.5 Filtrování grafu podle pravidel

Pokud by uživatel hledal v grafu komponenty nebo elementy s určitými vlastnostmi (např. jméno, definované hodnoty tagu) bylo by vhodné, aby šlo graf určitým způsobem filtrovat. Následující rozšíření umožňuje vyhledávat uzly komponent grafu a elementy, které splňují definovaná pravidla. Pro výsledek vyhledání je opět použito barevné vyznačení jako v ostatních implementovaných rozšířeních.

9.5.1 Druhy barevného rozlišení

Aktivovaná pravidla barevně zvýrazní prvky grafu, které pravidlu vyhovují. Pro pravidla aplikovaná na komponenty je možné barevně vyznačit následující vlastnosti (uvedené možnosti jsou označeny na obrázku 9.25):

1. pozadí panelu uzlu s názvem komponenty (**A**)
2. rámeček uzlu komponenty (**B**)
3. uzel komponenty v náhledu grafu (**C**)

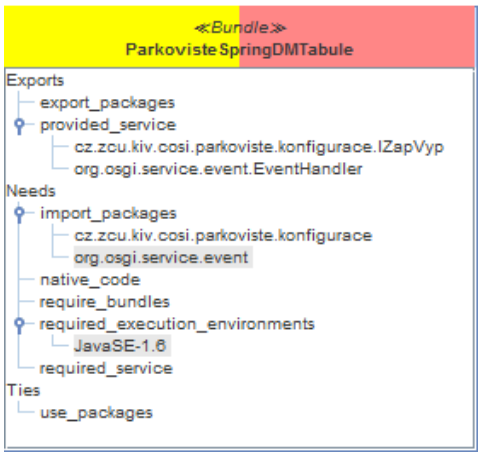


Obrázek 9.25: Typy zvýraznění v grafu po aplikování pravidel filtrování

Pro elementy lze vyznačit pouze jejich komponentu a navíc existují ještě dvě volby:

4. pozadí názvu elementu v překrytí uzlu komponenty (**D**)
5. barva názvu elementu v překrytí uzlu komponenty (**E**)

Některá ze zvýraznění lze na prvky aplikovat pouze jednou (např. barva textu elementu). Jiná zvýraznění lze naopak na prvky používat vícenásobně. Na obrázku 9.26 je pro ukázku komponenta s dvojitým zvýrazněním panelu s názvem zobrazené komponenty.



Obrázek 9.26: Prvek se stejným typem zvýraznění více pravidel

9.5.2 Pravidla pro filtrování

Aby vůbec šla pravidla pro filtrování použít, byla vytvořena podle ENT meta-modelu. Pro filtrování lze tedy definovat dva typy pravidel, a sice pravidla komponent a pravidla elementů. Na komponenty je možné aplikovat pravidla definující jejich:

1. typ
2. jméno
3. množinu hodnot tagů (nepovinné)

U elementů je situace lehce odlišná. U pravidel pro ně se nejprve definuje typ komponenty, poté trait a až poté vlastní pravidla pro elementy definující:

1. jméno
2. množinu hodnot tagů (nepovinné)

U jmen a hodnot tagů lze v programu používat zástupný znak „%“ představující nula až nekonečno různých znaků. Počet nebo umístění zástupných znaků v řetězci není pro jedno jméno nebo tag omezen a lze tak skládat různé vzory pro text.

Na množiny tagů (buď komponent nebo elementů) se uplatňuje logický součin. Aby tedy bylo pravidlo pravdivé, musí prvek splňovat všechny definované hodnoty tagů. Jen pro upřesnění, logický součet by se provedl definováním dvou pravidel a jejich současnou aktivací.

Konstruovaná pravidla jsou ukládána do xml souborů do pracovní složky aplikace. Ta je používána také například pro uložení načtených projektů. Tímto je umožněno zpětné načítání pravidel po startu aplikace.

Aby byly uvedené možnosti konstrukce pravidel jasné, zde je ukázka souboru s definovanými pravidly pro elementy:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ComavConditionalRule name="Pro Elementy">
  <Target>Element</Target>
  <Rules componentType="Bundle" search="%event" traitType="export_packages">
    <Rule>
      <Tag name="version">
      <Contains>%</Contains>
      </Tag>
    </Rule>
  </Rules>
  <Highlight>
    <Type>Border</Type>
    <Color g="0" r="0" b='255" />
  </Highlight>
</ComavConditionalRule>
```

Jak je ze struktury a obsahu xml vidět, pravidlo má název „*Pro elementy*“, jeho cílovým prvkem jsou elementy názvu „*%event*“, které jsou v komponenách typu „*Bundle*“ a jsou zahrnuté ve traitu *export_packages*. Dále pravidlo obsahuje jednoprvkovou množinu tagů elementu, vyžadující, aby tag s názvem *version* obsahoval text „*%*“. Výsledné zvýraznění v grafu bude modrou barvou vyznačovat rámeček kolem uzlů komponent.

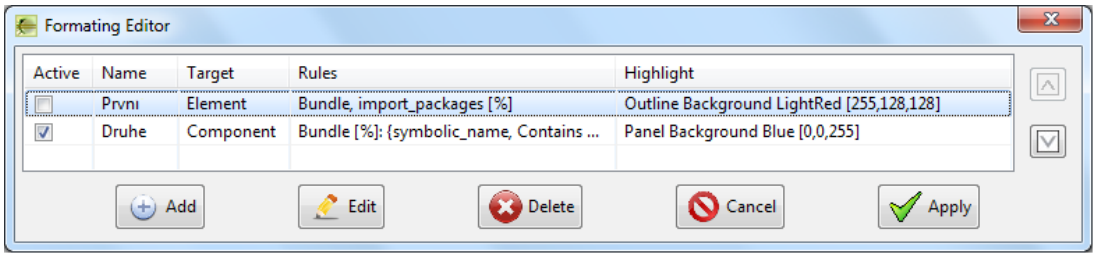
9.5.3 Uživatelské rozhraní filtrování grafu

Definování a správa pravidel pro filtrování grafu je obsluhováno pomocí vytvořených dialogových oken. Ačkoliv by popis jejich vytvoření mohl spadat do kapitoly 9.5.4, nebude jim věnována větší pozornost, jelikož by popis neobsahoval žádné myšlenkově zajímavé postupy.

Hlavní rozhraní s tabulkou pravidel je spouštěno tlačítkem s ikonou na obrázku 9.27, nacházející se v levé části okna plug-inu AIVA.

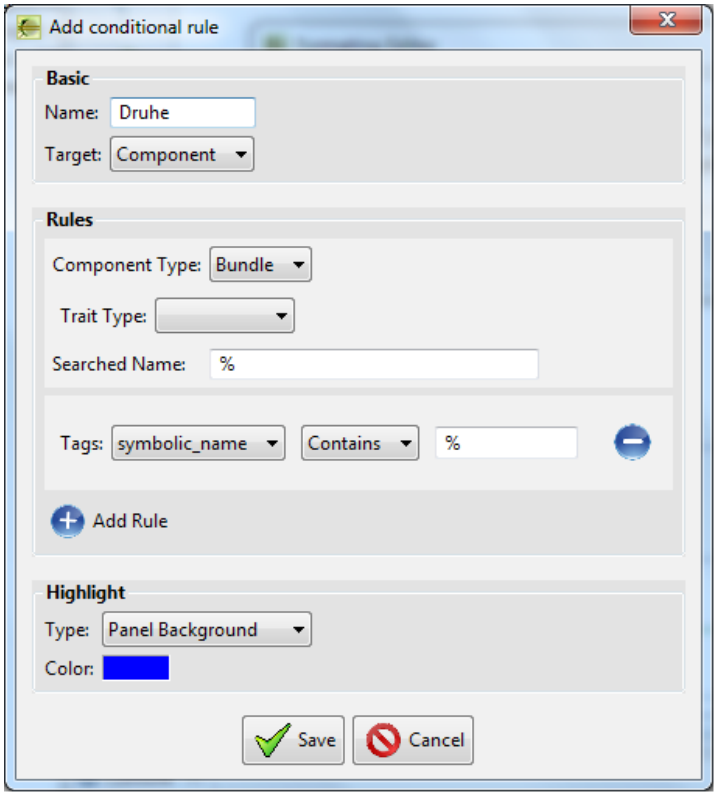
Obrázek 9.27: Ikona tlačítka pro spuštění rozhraní pravidel pro filtrování grafu

Rozhraní pro existující pravidla je tvořeno tabulkou a několika ovládacími tlačítky (viz obrázek 9.28). Jak lze vidět, v tabulce jsou uvedena dvě pravidla definovaná svým jménem, cílovým typem, vlastními pravidly a výsledným typem zvýraznění pro prvky. Pravidla lze různě aktivovat, což je z obrázku patrné. Zajímavá je možnost pohybu (změny pořadí) vybraného pravidla v tabulce pomocí tlačítek v pravé části, nebo pomocí implementované možnosti tahu a vložení (Drag and Drop). Tím lze definovat pořadí, v jakém se budou pravidla na graf aplikovat.



Obrázek 9.28: Dialog s rozhraním pro existující pravidla

Pokud prvek splňuje vícero pravidel majících stejné zvýraznění, bude na něj aplikováno nejprve to, které je v seznamu pravidel v editoru první (nejvíce nahoře). Pokud jde o zvýraznění, jež nelze aplikovat vícekrát (viz kapitola 9.5.1), použije se pouze první pravidlo s tímto zvýrazněním. Rozhraní pro vytváření a úpravu pravidel je na obrázku 9.29.

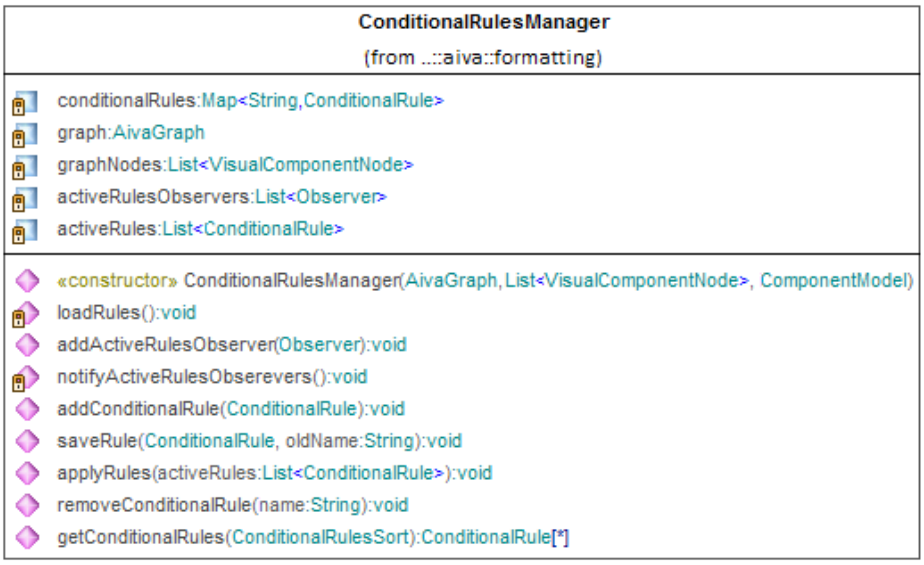


Obrázek 9.29: Dialog pro vytváření a úpravu pravidla

9.5.4 Implementace filtrování grafu

Jak již bylo zmíněno, pravidla jsou ukládána po svém vytvoření nebo změně, aby mohla být opětovně načtena při dalším startu programu. O tuto činnost se stará třída pro správu pravidel `ConditionalRulesManager`. Tato třída obsluhuje i aplikovaní pravidel na výsledné prvky, respektive jejich získání a komunikuje s dialogy popsanými v předchozí kapitole. Její stručný class diagram je na obrázku 9.30.

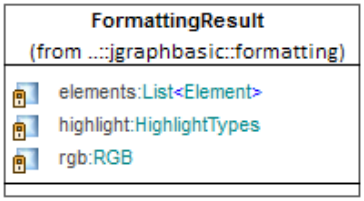
Získání výsledných komponent nebo elementů má na starosti třída `ConditionalRule`, která je modelem pravidla pro filtrování grafu. Uchovává všechny nastavené atributy pomocí dialogu z předchozí kapitoly a stará se například i o uložení a načtení svých atributů inicializovaných managerem pravidel `ConditionalRulesManager`.



Obrázek 9.30: Class diagram třídy `ConditionalRulesManager`

Výsledky jsou pak ukládány do mapy (kolekce `HashMap`), kde:

- klíčem jsou modely mezi uzly grafu a komponentami (tedy třída `VisualComponentNode`)
- hodnotami je seznam instancí třídy `FormattingResult` (její class diagram je na obrázku 9.31), což je třída uchovávající informace o zvýraznění pro jedno pravidlo.



Obrázek 9.31: Class diagram třídy `FormattingResult`

Klíčem je popsaná třída, protože jedna komponenta může být výsledným prvkem více pravidel. Díky tomu zbývá, aby třída `FormattingResult` obsahovala výsledné elementy komponenty (pokud takové existují), typ zvýraznění a jeho barvu.

Získané informace o zvýraznění z aktivních pravidel poté manager `ConditionalRulesManager` zasílá všem získaným instancím třídy `VisualComponentNode` (tedy klíč v získaných výsledcích z pravidla).

Třída `VisualComponentNode` se poté stará o roztřídění a další zpracování získaných zvýraznění. Prochází například všechny požadavky na zvýraznění elementů (pozadí či barva textu) a pokud jde požadavek na obarvení elementů od více pravidel, uloží pouze barvu od prvního pravidla (viz pořadí pravidel v kapitole 9.5.1).

Třída pak dále nastavuje potřebná zvýraznění podle jejich typu. Následující seznam více popisuje postup třídy `VisualComponentNode` při zpracování požadavků ke zvýraznění.

1. **Zvýraznění uzlu v náhledu grafu**

   Uzlu lze v náhledu grafu nastavit pouze jednu barvu, proto je uzel obarven prvním pravidlem. Zvýraznění nastavuje přímo třída `VisualComponentNode`.

2. **Barva rámečku uzlu v grafu**

   Uzlu lze nastavit více barevných rámečků. Pro vytvoření rámečku se používá třída překrytí `CellMultiBorderOverlay`. Toto překrytí nemůže překročit velikost uzlu, proto se vlastní rámečky vykreslují dovnitř uzlu a překrývají základní překrytí s obsahem komponenty uzlu. Obsah uzlu s více jak dvěma rámečky tak přestává být vidět celý.

3. **Pozadí panelu uzlu s názvem komponenty**

   Panelu uzlu s názvem komponenty lze nastavit více pozadí. Šířka uzlu je rozdělena podle počtu zvýraznění panelu a jednotlivé úseky jsou pak vykresleny jinými barvami (viz obrázek 9.26). O nastavení barvy panelu se starají přímo jednotlivá překrytí uzlů. Třída `VisualComponentNode` s překrytím komunikuje přes metodu abstraktního překrytí `BasicCellOverlay`.

4. **Pozadí *elementů* komponenty v uzlu**

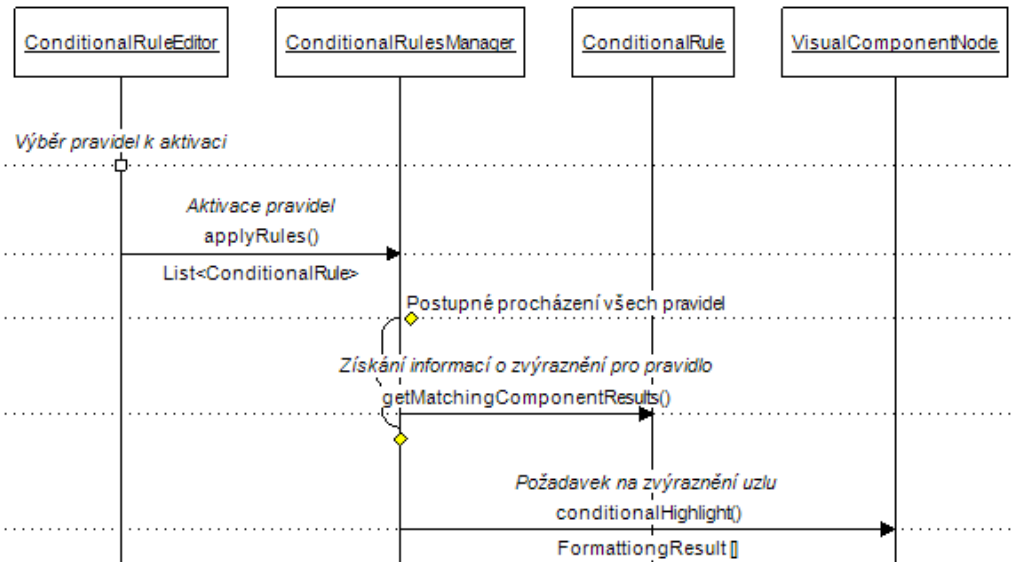
   Jeden *element* komponenty může mít v uzlu pouze jednu barvu pozadí. Avšak uzel může například mít pozadí každého elementu vykreslené jinou barvou (od jiného pravidla). O nastavení barvy pozadí elementů se starají přímo jednotlivá překrytí uzlů. Třída `VisualComponentNode` s překrytím komunikuje přes metodu abstraktního překrytí `BasicCellOverlay`.

5. **Barva textu elementů komponenty v uzlu**

   U barvy textu elementů platí stejné věci jako pro barvu jejich pozadí (tj. bod 4).

Nastavení pozadí nebo textu elementů komponenty v uzlu umožňuje překrytí opět pomocí rendereru `ComponentTreeRenderer` (viz kapitola 9.2.3, zvýraznění elementů po kliknutí na spojení).

Postup při aktivaci pravidel pro filrování grafu je ještě přiblížen sekvenčním diagramem na obrázku 9.32.



Obrázek 9.32: Sekvenční diagram aplikace pravidel pro filtrování grafu

9.6 Podpora hierarchických komponentových modelů

SOFA2 je hierarchickým komponentovým modelem. Komponenta tedy může být složena z několika dalších komponent, tzv. *sub-component* (*podkomponent*). Složená komponenta je pak označována jako *composite*. Složenou komponentou však může být v modelu každá komponenta (tedy i podkomponenty, atd.) a tak nelze určit, do jaké úrovně budou mít komponenty další podkomponenty.

Popisované rozšíření se stará právě o podporu hierarchických komponentových modelů.

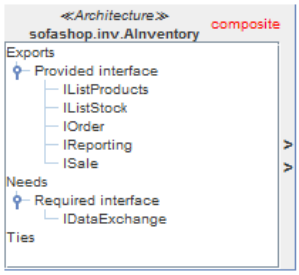
9.6.1 Hierarchické grafy v AIVA

Složené komponenty jsou odlišeny nápisem *composite* v pravé části panelu s názvem komponenty a možností zobrazit/skrýt všechny podkomponenty tlačítkem v pravé části těla komponenty. Pro složenou komponentu se tedy odlišují dva stavy vzhledu:

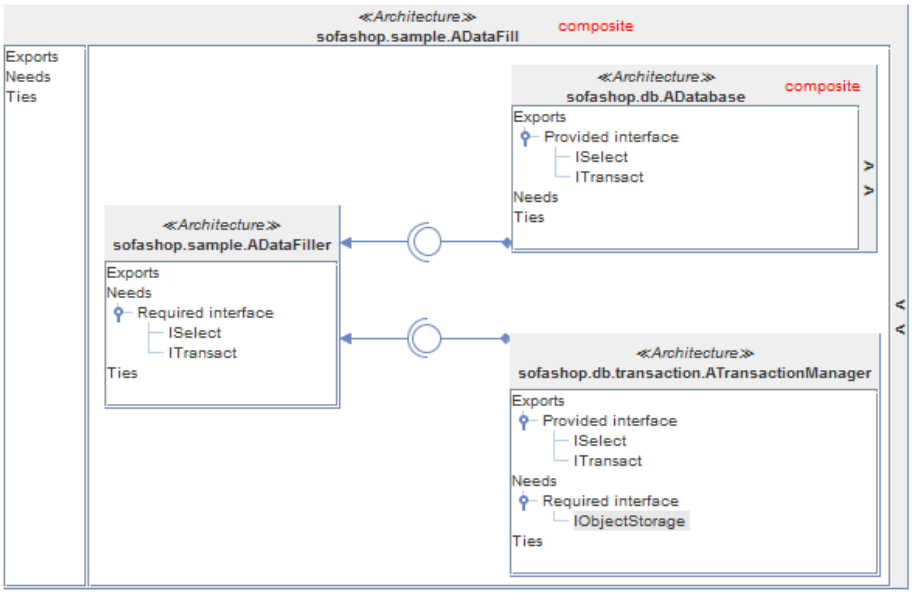
1. základní stav
2. podrobný stav

Základním stavem je myšleno zobrazení pouze obsahu komponenty – traity, elementy. Podrobným stavem pak zobrazení stejných věcí jako základní stav, ale také zobrazení vlastních podkomponent.

Na obrázku 9.33 vidět složená komponenta v základním stavu. Na obrázku 9.34 je pak tato komponenta v podrobném stavu. Z obrázku lze poznat, že komponenta



Obrázek 9.33: Základní stav složené komponenty



Obrázek 9.34: Rozšířený stav složené komponenty

`sofashop.sample.ADataFill` obsahuje tři podkomponenty, z nichž jedna je také složenou komponentou. V levé části překrytí však nadále zůstal její obsah.

9.6.2 Implementace podpory hierarchických komponentových modelů

Je jasné, že implementace překrytí komponent se pro složené komponenty zčásti liší. Prvním krokem při implementaci podpory hierarchických komponentových modelů tedy bylo rozšíření existujících překrytí pro komponenty. Pokud tedy komponenta, pro kterou je překrytí vytvářeno, obsahuje další podkomponenty, vytvoří překrytí příslušný popisek a tlačítko pro přepnutí do rozšířeného stavu.

Jak lze poznat z obrázku 9.34, složená komponenta může mít více podkomponent a ty mohou být také spojeny – tedy jde o graf. Pro vykreslení podgrafu složené komponenty bylo jedinou rozumnou volbou znovu použít třídu `AivaGraph` a to bez žádných velkých zásahů. Pouze do ní byla přidána mapa komponent a jejich podgrafů, použitá pro komunikaci například při změně layoutu. Díky tomu bylo velmi rychle dosaženo požadované funkčnosti.

Další podgrafy jsou instancemi třídy `AivaSubGraph` (dědící od `AivaGraph`) obsahující pouze několik změn pro práci s podgrafem. Jde například o zjištění pozice komponenty, která je nevíce vpravo a dole. Díky tomu lze nastavit velikost uzlu složené komponenty a jejího překrytí, pokud je její podgraf viditelný (tzn. komponenta je v rozšířeném stavu).

Jak bylo zmíněno, překrytí komponent musí být vytvořeno jinak (vytvoření tlačítka pro přepnutí stavu, atd.), proto byly vytvořeny také třídy, dědíce od základních překrytí pro komponenty. Konkrétně tedy byla vytvořena například třída `ComponentHierarchyBookmarkOverlay` dědíc od třídy `BookmarkOverlay`, jenž je popsaná v kapitole 9.1.2, alternativní vzhled komponent.

Část pseudokódu z metody třídy `ComponentHierarchyBookmarkOverlay` vytvářející obsah překrytí je uvedena níže:

```
protected void initialize() {
	/* Vytvoření základního překrytí komponenty */
	super.initialize();

	/* Komponenta nemá podkomponenty */
	if (component.getComponentSetList().size() == 0)
		return;

	/* Panel pro podgraf */
	graphPanel = new JPanel();
	/* Vytvoření podgrafu*/
	var parentGraph = (AivaGraph) graphComponent.getGraph();
	subGraph = new AivaSubGraph(graphPanel, component, parentGraph, cell);
	subGraph.drawModel();

	/* Přidáni podgrafu */
	parentGraph.addSubGraph(component.getName(), subGraph);

	/* Tlačítko pro přepnutí přepnutí stavu */
	expandButton = new JButton();
	expandButton.addMouseListener(new MouseAdapter() {
		@Override
		public void mouseClicked(MouseEvent e) {
			/* Zobrazení/ Skrytí podgrafu */
			graphPanel.setVisible(!graphPanel.isVisible());

			/* Proběhla změna stavu = změna velikosti uzlu */
			compositeExpandChanged = true;

			/* Změna velikosti */
			changeOnPreferredSize();
		}
	)};
...
```

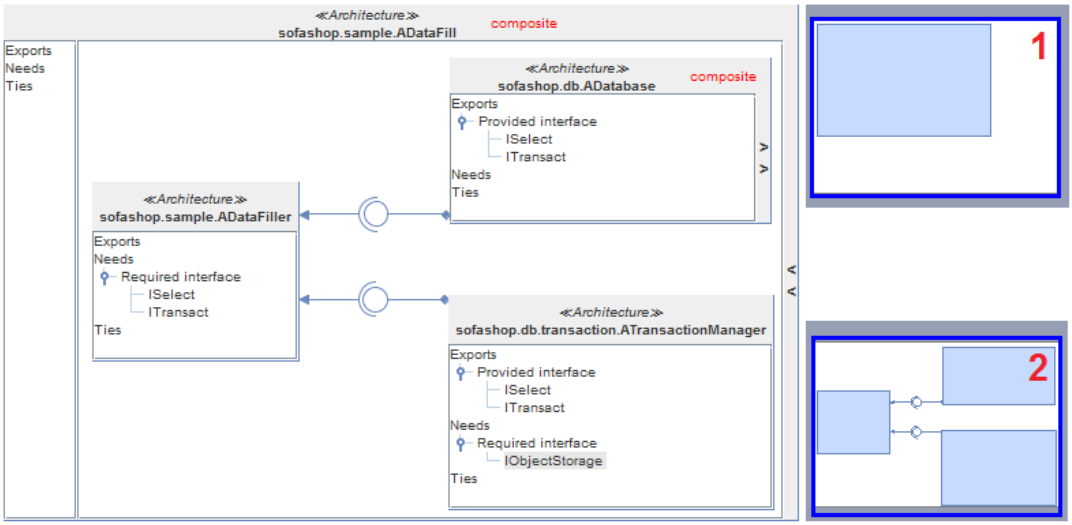
Změny velikosti uzlů byly pro implementaci tohoto rozšíření největším problémem. Je totiž nutné měnit velikost při zobrazení podgrafu a jeho skrytí. To ovšem platí pro všechny podgrafy. Skrytí podgrafu komponenty totiž ovlivní velikost její rodičovské komponenty a dále až ke komponentě, která nemá žádnou rodičovskou komponentu. Při změnách se tak tedy používá rekurzivní volání příslušné metody, dokud komponenta má nějakou rodičovskou komponentu. Komentovaná metoda reagující na změnu velikosti překrytí podkomponenty je uvedena níže:

```
public void changeOnPreferredSizeFromSubcell() {
	Dimension dim = getPreferredSize();
	mxGeometry geom = ((mxCell) cell).getGeometry();
	int height = (int) dim.getHeight();
```

```
        int width = (int) dim.getWidth ();

        // Vynucení změna velikosti překrytí
        graph.resizeCell(cell, new mxRectangle(geom.getX(), geom.getY(),
                    width, height));

        // Vytvoření okrajů pro podgraf
        subGraph.createGraphMargins();

        // Opravení velikosti uzlu rodičovské komponenty pokud existuje
        if (!(graph instanceof AivaSubGraph))
              return;
        var parGraph = ((AivaSubGraph) graph).getParentGraph();
        // Buňka rodičovského grafu
        Object parCell = parGraph.getParentCell();
        // Model rodičovského grafu
        var parModel = parGraph.getGraphModel();

        // Model buňky rodičovkého grafu
        VisualComponentNode parentNode = parModel.findNodeToVertex(parCell);
        if (parentNode != null) {
              // Rekurzivní volání pro pro úpravu velikosti rodičovského uzlu
              parentNode.getCellOverlay()).changeOnPreferredSizeFromSubcell();
        }
}
```

9.6.3 Náhledy hierarchických grafů

Užitečným doplnění tohoto rozšíření je změna náhledu grafu podle toho, do jakého grafu se naposledy kliklo myší. O změnu náhledu se pak stará třída `AivaGraphMouseListener` (viz kapitola 9.2.3, zvýraznění elementů po kliknutí na spojení). Její instance je totiž vytvářena pro každý graf (i podgraf), a tak podle svého grafu může nastavit aktuální náhled po kliknutí do vnitřku grafu. Na obrázku 9.35 je vidět náhled komponenty z vnějšího (**1**) a z vnitřního (**2**) pohledu, tedy po kliknutí do podgrafu v těle komponenty.



Obrázek 9.35: Náhled grafu hlavní komponenty z vnějšku a vnitřku

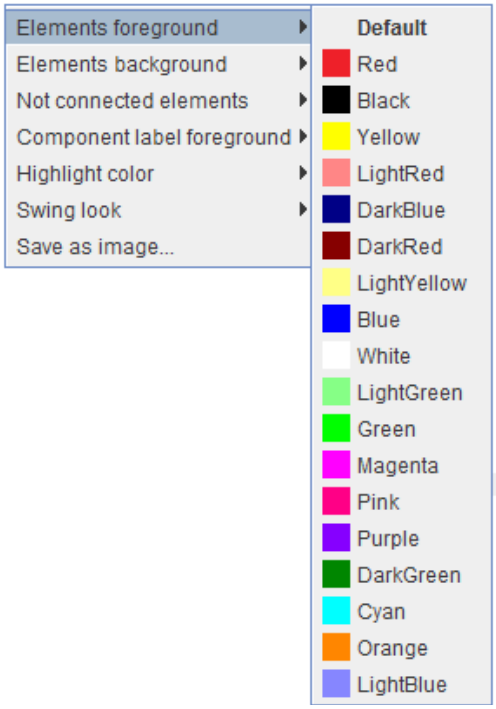
9.7 Ostatní rozšíření

Během implementací rozšíření modulu AIVA bylo vytvořeno také několik dalších menších vylepšení, která se jevila jako užitečná. Tato kapitola se bude zabývat jejich stručným popisem.

Jednoduchým užitečným rozšířením je použití vlastní třídy pro správu náhledu grafu `AivaOutline`. Díky ní je možné kliknutím na pozici v náhledu grafu změnit oblast grafu, kterou zobrazuje modul AIVA. Což nebylo s použitím standardní třídy pro náhled možné.

9.7.1 Menu pro změnu barev

Jak bylo zmíněno v kapitole 9.1, je elementům, které nejsou součástí žádného spojení, nataveno šedé pozadí. Díky tomu tak lze na první pohled rozeznat, je-li element součástí nějakého spojení. To je však základní nastavení, které nemusí vyhovovat každému. Pro zpříjemnění práce proto bylo vytvořeno kontextové menu grafu, kde lze měnit barvy některých zvýraznění grafu (obrázek menu viz 9.35).



Obrázek 9.36: Kontextové menu grafu

Jak je vidět, lze pomocí menu nastavit několik věcí a to konkrétně:

- Barvu textu elementů
- Barvu pozadí elementů
- Barvu pozadí elementů, které nejsou součástí žádného spojení
- Barvu textu názvu komponenty v překrytí
- Barvu použitou při zvýraznění spojení (aplikuje se na všechny zúčastněné prvky, tj. rámeček uzlu komponenty, pozadí elementu, uzel v náhledu grafu a hranu spojení)
- Typ použitého vzhledu prvků knihovny Swing (pomocí jejíchž prostředků jsou vytvořena překrytí uzlů)

O správu barev, jejich přepínání a vytváření částečného menu se stará třída `ComponentsColorsManager`. Při změně některé z možností jsou volány metody třídy `AivaGraph`, které se dále starají o změnu barev.

9.8 Vývojové prostředky a požadavky pro běh aplikace

Pro shrnutí zde budou uvedeny prostředky použité při vývoji aplikace. Aplikace je implementována pomocí technologie **Eclipse RCP**, z čehož vyplývá, že jako programovací jazyk byla použita **Java**, konkrétně vývojová verze **JDK 6.0_21**. Pro běh aplikace je tedy nutné mít nainstalovanou minimálně tuto verzi Javy. Jako vývojové prostředí bylo použito **Eclipse Indigo** pro vývojáře RCP a RAP aplikací. Novější verze Eclipse Juno vyšla až v létě roku 2012, tedy po dokončení implementačních prací a proto nebyla použita. V plug-inu AIVA je používána knihovna JGraphX verze 1.8. Při vývoji byl použitý verzovací systém Subversion (SVN), který přinesl několik výhod:

- Uchování předchozích nahraných verzí programu
- Data jsou přístupná z kteréhokoliv počítače s připojením k internetu
- Porovnávání nahraných verzí
- Možné oddělení vývoje jednotlivých rozšíření aplikace

Vývoj aplikace proto systém SVN značně ulehčil. Především možnost vracet se k předchozím nahraným verzím kódu se ukázala jako velmi potřebná.

Pro zadávání požadovaných vylepšení a nalezených chyb byl používán systém pro správu chyb Assembla [AS].

10 Závěr

Hlavním cílem diplomové práce bylo rozšířit možnosti reprezentace komponentových aplikací existujícího nástroje ComAV, respektive jeho plug-inu AIVA. Nástroj ComAV vznikl na Katedře informatiky a výpočetní techniky jako vyústění výzkumu komponentových aplikací. Nástroj byl postaven nad platformou Eclipse RCP, která je vhodná pro vytváření komponentových aplikací. Informace o komponentách jsou načítána z různých komponentových modelů do generického komponentového ENT meta-modelu. Samotné zobrazení architektury komponentových aplikací provádí plug-in AIVA.

Všechny body zadání diplomové práce byly úspěšně splněny. Implementovaná rozšíření dosáhla očekávání a umožňují lepší a bohatší práci se zobrazeným komponentovým diagramem. Jednotlivá rozšíření byla úspěšně implementována a průběžně testována na několika komponentových aplikacích různých komponentových modelů. Některá rozšíření a jejich funkční detaily, byly přímo zadáním diplomové práce. Navíc také byla přidána další užitečná rozšíření (například změna barev pro různé typy zvýraznění) a v aplikaci bylo opraveno několik chyb a nedostatků.

10.1 Navrhovaná vylepšení

Před implementací dalších vylepšení by bylo vhodné opravit všechny drobné problémy se zobrazením. Knihovna JGraphX poskytuje dobré možnosti pro práci s grafem komponent, avšak někdy se zdá být nespolehlivá (různá překreslení při změně zvýraznění, aj.).

Zajímavým vylepšením by bylo umožnit ve filtru definovat místo barev zvýraznění jiný přístup, založený na zobrazení/skrytí prvků grafu splňujících definovaná pravidla. Tím by se mnohdy graf po aplikování filtru zpřehlednil.

Jako další zajímavé vylepšení by bylo dobré zlepšit úplnost zobrazených informací pro tisk grafu. Při tisku (exportu obrázku) totiž nejsou vidět tagy komponent a elementů a také nejsou vidět informace o spojení mezi komponentami. Pro práci s programem, respektive diagramem komponentové aplikace, je to samozřejmě správné, avšak pro tisk

by možná mohla být užitečná možnost nějakým způsobem tyto skryté informace zobrazit. Případně poskytnout prostředek, jak vybrat pouze část těchto ukrytých informací pro zobrazení v exportovaném obrázku. Díky tomu by mohla být aplikace lépe použitelná při práci s diagramem vytištěným na papíru, tedy pro různou technickou dokumentaci.

Dalším vylepšením by nejspíše měla být analýza nové verze platformy Eclipse RCP a zvážení, zda není vhodné nástroj ComAV přepracovat tak, aby ji používal.

Přehled zkratek a pojmů

AIVA	- Advanced Interactive Approach, plug-in nástroje ComAV pro zobrazování
API	- Application Programming Interface, rozhraní pro komunikaci s komponentou, knihovnou, aj.
CBSE	- Component-based Software Engineering, oblast softwarového inženýrství vývoje komponentových aplikací
CBD	- Component-based Development, oblast softwarového inženýrství vývoje komponentových aplikací
ComAV	- Component Application Visualizer, nástroj pro zobrazení architektury komponentových aplikací
EJB	- Enterprise Java Beans, řízený, serverově orientovaný komponentový model
ENT	- generický komponentový meta-model použitý v nástroji ComAV
Equinox	- Implementace komponentového modelu OSGi, na které je postavena platforma Eclipse RCP
Extension point	- Bod rozšíření, jedná se o způsob, jakým lze rozšířit funkcionalitu plug-in platformy Eclipse RCP
HCI	- Human Computer Interface, interakce mezi uživatelem a počítačem při zobrazování dat
IDE	- Integrated Developement Enviroment, integrované vývojové prostředí pro psaní programů
JGraphX	- Java knihovna pro zobrazení a práci s grafy použitá v plug-inu AIVA
JNI	- Java Native Interface, knihovna určené k zobrazení nativních grafických prvků OS

JVM	- Java Virtual Machine, virtuální stroj který umožňuje spustit bytekód programovacího jazyka Java
Loader	- obecný plug-in pro načtení aplikací z různých komponentových modelů do ENT meta-modelu v programu ComAV
MOF	- Meta-Object Facility, abstraktní jazyk pro specifikaci, vytváření a správu meta-modelů
MVC	- Model View Controller, softwarová architektura rozdělující datový model, uživatelské rozhraní a řídící logiku do nezávislých vrstev
Plug-in	- Jiný výraz pro komponentu, používaný v prostředí Eclipse
OSGi	- Open Services Gateway initiative, komponentový model postavený nad JVM
RCP	- Rich Client Platform, softwarová platforma určená k vývoji rozsáhlých desktopových aplikací založených na komponentách
SOFA	- hierarchický komponentový model
SVN	- Subversion, verzovací systém pro správu zdrojů
SWT	- Knihovna pro zobrazení grafických prvků OS v programovacím jazyce Java v programovacím jazyce Java
UI	- User Interface, uživatelské rozhraní pro ovládání aplikací
UML	- Unified Modeling Language, modelovací jazyk určený k popisu softwarových systémů a fází procesů jejich vývoje
Visualizer	- obecný plug-in pro zobrazení komponentové aplikace v programu ComAV
XML	- eXtensible Markup Language, značkovací jazyk umožňující logické vyznačování

Použitá literatura

[EMM04]: Přemysl Brada, The ENT Meta-Model of Component Interface, version 2, 2004

[MPSC68]: Douglas McIlroy, Mass Produced Software Components, 1968

[CS02]: Szyperski, C, Component Software: Beyond Object-Oriented Programming, 2002

[PC95]: Anneliese von Mayrhauser, A. Marie Vans, Program comprehension duringsoftware maintenance and evolution, 1995

[SV07]: Stephan Diehl, Software Visualization - Visualizing the Structure, Behaviour,and Evolution of Software, 2007

[RIIV07]: Ji S Yi, Youn an Kang, John T. Stasko, IEEE, Julie A. Jacko, Toward a Deeper Understanding of the Role of Interaction in Information Visualization, 2007

[ICV11]: Jaroslav Šnajberk, Přemysl Brada, Interactive Component Visualization, Visual Representaion of Component-Based Applications using ENT meta-model, 2011

[MOF06]: Object Management Group, Meta Object Facility (MOF) 2.0 CoreSpecification, 2006

[DLFV10]: Jaroslav Šnajberk, Přemysl Brada, Implementation of a data layer for the visualization of component-based applications, 2010

[ENT11]: Jaroslav Šnajberk, Přemysl Brada, ENT: A Generic Meta-Model for theDescription of Component-Based Applications, 2011

[LKM11]: Martin Sloup, Loadery komponentových modelů, 2011

[ERCP10]: Jeff McAffer, Jean-Michel Lemieux, Chris Aniszczyk, Eclipse Rich Client Platform : Designing, Coding, and Packaging Java™ Applications, Second Edition, 2010

[VOG]: Lars Vogel, Eclipse 4 RCP – Tutorial, http://www.vogella.com/articles/EclipseRCP/article.html

[GEPF10]: Jan Krákora, Grafický editor pro existujícíkomponentový framework, 2010

[JSC]: Gaudenz Alder, Design and Implementation of the JGraph SwingComponent, http://www.jgraph.com/downloads/jgraph/legacy/jgraph-paper.pdf

[JUM]: JGraphX User Manual, 2013, http://jgraph.github.com/mxgraph/docs/manual_javavis.html

[FIV04]: Alexandru Telea, Lucian Voinea, A Framework for Interactive Visualization of Component-Based Software, 2004

[AS]: http://www.assembla.com

Přílohy

Obsah přiloženého cd

Aplikace

src – zdrojové kódy programu

dist – spustitelná aplikace

doc – javadoc dokumentace plug-inů AIVA a JGraphBasicVisualization

Testovací komponenty – připravené testovací komponentové aplikace podle typu jejich modelu

osgi

sofa2

ejb3

Text – text diplomové práce

Uživatelská příručka

Celá aplikace ComAV je celkem rozsáhlá a samotná diplomová práce se týkala pouze jejího rozšíření, proto bude uživatelská příručka stručná. Všechny texty prvků aplikace jsou napsány v anglickém jazyce. Zobrazené informace o komponentách vycházejí z ENT meta-modelu a proto je vhodné mít o něm alespoň elementární znalosti.

Požadavky pro běh

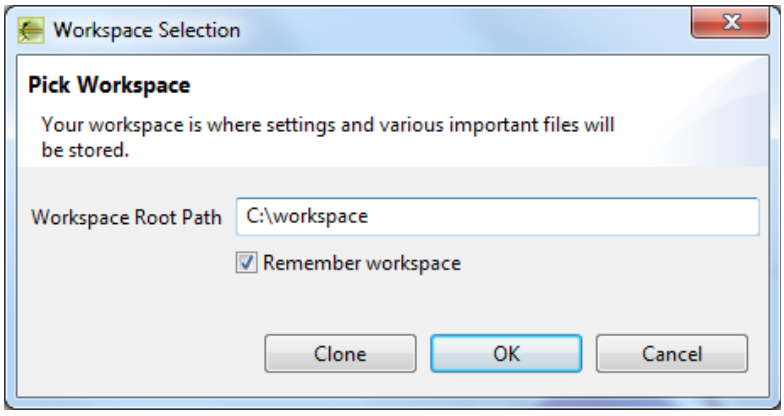
ComAV je klientská aplikace vytvořená pro operační systém Windows. Pro spuštění aplikace je nutné mít nainstalovanou **JRE** (Java Runtime Enviroment) minimální verze **1.6_21**. Co se týká hardwarových požadavků, je vhodné aby počítač, na které se aplikace spustí, měl alespoň **521 MB** operační paměti a procesor s frekvencí **1 GHz**.

Workspace

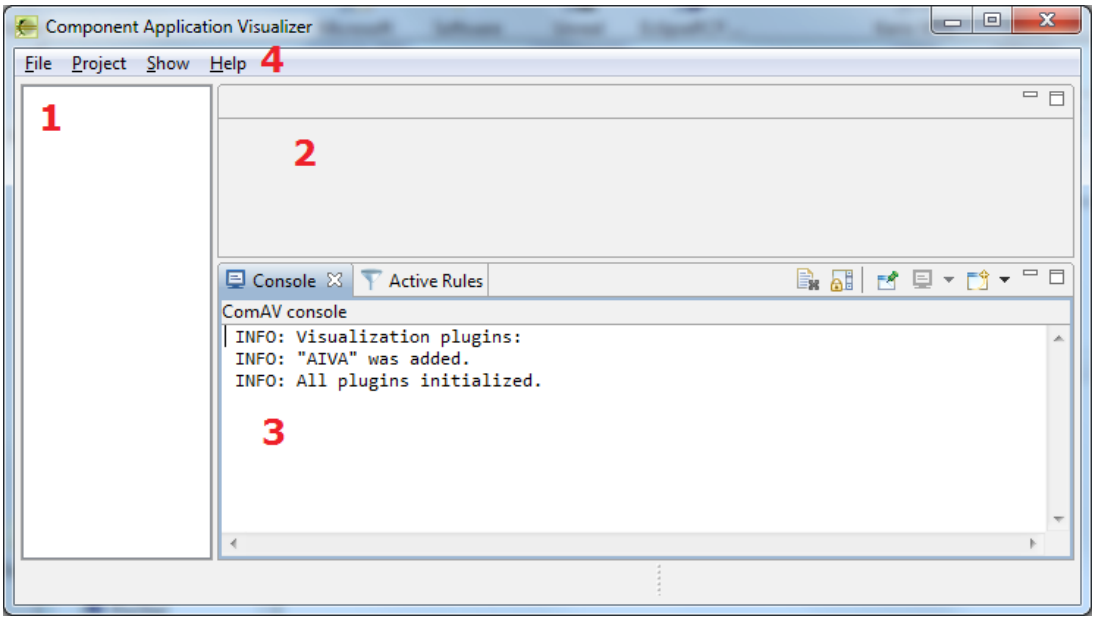
ComAV si spravuje svůj vlastní pracovní složku (workspace). Do ní jsou ukládány informace o načtených projektech a případných vytvořených pravidel pro filtrování grafu. Důvodem je opětovné načítání a použití dat po spuštění aplikace. Pracovní složku lze měnit buď po startu aplikace nebo po spuštění příslušnou položkou v menu.

Spuštění

Aplikace se spouští spustitelným souborem **ComAV.exe**. Po spuštění je nejprve nutné vybrat pracovní složku (viz obrázek A) a po jejím vybrání je spuštěna vlastní aplikace (viz obrázek B). Za pracovní složku je nutné vybrat složku, kam má přihlášený uživatel povolen zápis.



Obrázek A: Výběr pracovní složky ComAV



Obrázek B: Okno ComAV po spuštění

Hlavní prvky ComAV

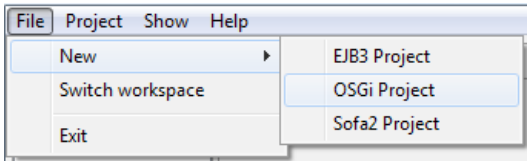
Hlavními prvky ComAV jsou číselně vyznačeny na obrázku výše. Jde o

- View pro vytvořené komponentové projekty (1)
- Editor oblast pro zobrazení komponentových projektů (2)
- View konzole pro výpis základních informací (3)
- Oblast menu aplikace (4)

Vytvoření a zobrazení projektu

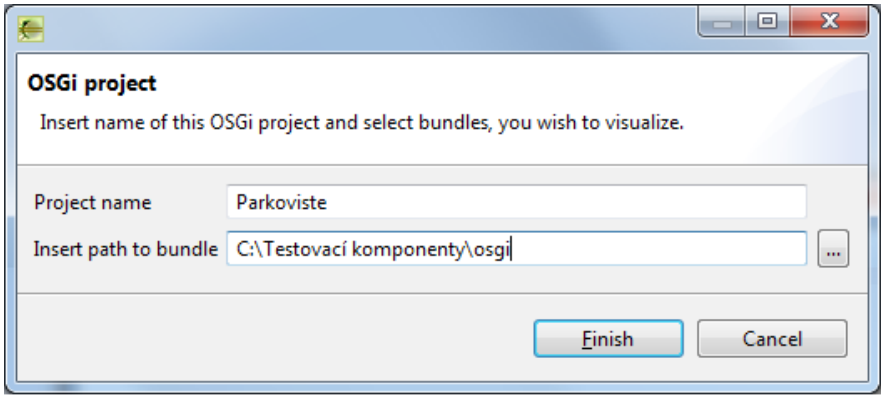
ComAV má připravené plug-iny pro načítání komponentových aplikací modelu OSGi, EJB a SOFA2. Načtení projektů je v ComAV pro všechny typy velmi podobné a proto se následující text omezí na popis načtení OSGi projektu.

Projekt se načte stisknutím menu položky *OSGi project*, umístěné pod menu položkami *File -> New* (viz obrázek C).



Obrázek C: Zobrazení menu aplikace pro načtení OSGI projektu

Poté se objeví dialog pro zadání názvu projektu a výběr cesty k načítané komponentové aplikaci (viz obrázek D).

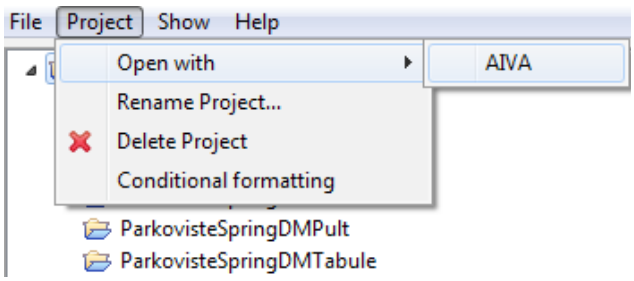


Obrázek D: Načtení OSGi projektu

Po potvrzení zadaných hodnot jsou do konzole uživateli vypsány informace o úspěšném načtení jednotlivých komponent. Také je do oblasti workspace view přidán záznam s názvem zadaného projektu (viz obrázek E).

Obrázek E: Zobrazení načteného projektu v ComAV

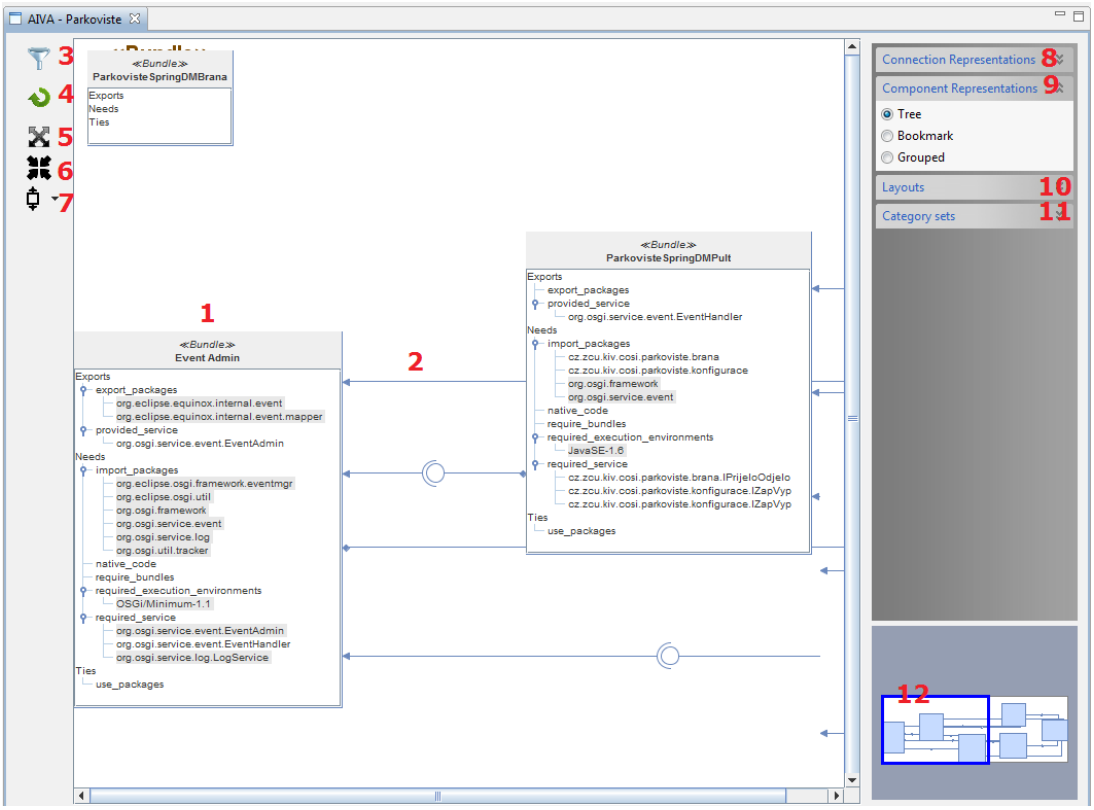
Projekt se nyní otevře kontextovým menu nad položkou ve workspace view nebo vybráním položky projektu (obrázek E) kliknutí myši a následným otevřením pomocí menu *AIVA*, umístěné pod menu položkami *Project -> Open With* (viz obrázek F).



Obrázek F: Otevření načteného projektu

Základní prvky plug-inu AIVA

Po otevření načteného projektu v editoru plug-inu AIVA (viz obrázek G) je tedy interaktivně zobrazena zkoumaná komponentová aplikace.



Obrázek G: Zobrazení načteného projektu v plug-inu AIVA

Důležitými zobrazenými prvky okna AIVA plug-inu jsou:

- Zobrazení komponenty s informacemi z ENT meta-modelu (**1**), v horní části se nachází panel s jejím typem a názvem – zde např. komponenta *Event Admin* typu *Bundle*. Podrobněji v textu diplomové práce.
- Zobrazené spojení mezi komponentami (**2**), podrobněji v textu diplomové práce
- Tlačítko spuštění editoru pravidel pro filtrování grafu (**3**)
- Tlačítko pro opětovné nastavení výchozí pozice prvků podle layoutu grafu (**4**)
- Tlačítko pro zobrazení kompletního těla komponent (**5**)
- Tlačítko pro zobrazení stručného těla komponent (**6**) – na obrázku výše je pro ukázku tímto způsobem zobrazena jedna komponenta vlevo nahoře
- Tlačítko pro nastavení definované výšky zobrazených komponent v grafu (**7**)

- Accordion menu s položkami pro změnu reprezentace zobrazených spojení mezi komponentami (**8**)
- Accordion menu s položkami pro změnu reprezentace zobrazených komponent (**9**)
- Accordion menu s položkami pro změnu layoutu grafu (**10**)
- Accordion menu s položkami pro změnu category setů zobrazených komponent – tj. zobrazené informace v těle komponenty (**11**)
- Pomocné okno s náhledem grafu, kde je zobrazen aktuální náhled na graf(**12**)

Pohyb v grafu plug-inu AIVA a úprava jeho prvků

Plug-in AIVA nabízí pro práci se zobrazenou aplikací různé funkčnosti. Některé z nich byly naznačeny popisem vizuálních prvků v předchozím textu. Mezi ostatní základní techniky ovládání patří také posun náhledu nad zobrazeným grafem, přibližování a oddalování pohledu v grafu a změna polohy a geometrie prvků grafu.

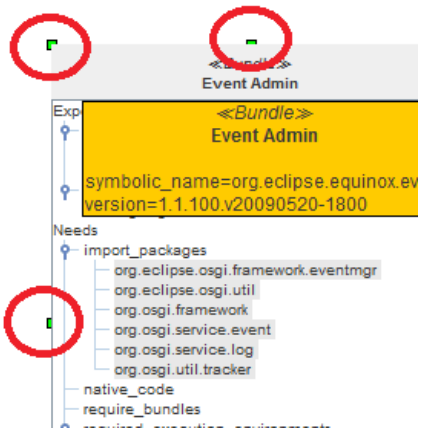
Posun v zobrazeném grafu se provede kliknutím pravého tlačítka myši do grafu a jeho podržením a následným posunem myši.

Graf komponent je po spuštění plug-inu AIVA zobrazen v maximálním přiblížení. Pro rozsáhlé grafy nebo pro rychlé zjištění vztahů mezi komponentami, lze použít oddálení pohledu najetím kurzoru myši nad zobrazený graf a posunem kolečka myši (postup je naznačen na obrázku H).

Obrázek H: Přibližování a oddalování náhledu

Polohu komponenty lze změnit stisknutím tlačítka myši nad horním panelem požadované komponenty, následným posunem a současně držením tlačítka myši na novou pozici. Pro umístění komponenty na novou pozici se uvolní tlačítko myši.

Rozměr komponenty lze měnit kliknutím na komponentu, kdy se zobrazí kolem komponenty zelený rámeček s vyznačenými body rámečku (viz obrázek I). Poté lze uchopením (stisk myši a posun) vyznačených bodů měnit rozměry komponenty.



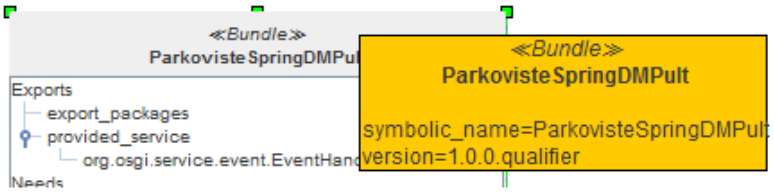
Obrázek I: Možnost změny velikosti komponenty

Geometrie linií v zobrazeném grafu lze měnit uchopením požadované linie a jejím posunem na novou pozici. Graf v tomto neumožňuje žádné bohaté možnosti a je může se stát, že linie se neumístí přesně podle požadavků uživatele.

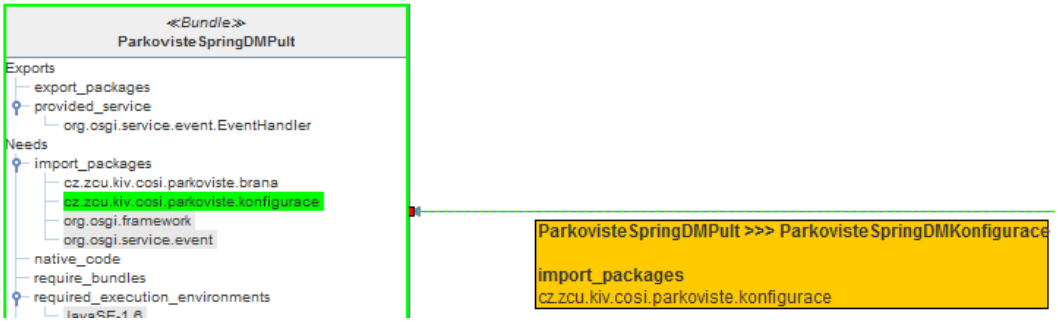
Interaktivní zobrazení informací

V zobrazené komponentové aplikaci lze získat další informace. Jde o informace vycházející z ENT meta-modelu a také doplňující informace (spojení komponent). Z obrázku G nejsou patrné a proto budou uvedeny obrázky i zde. Konkrétně tedy jde o:

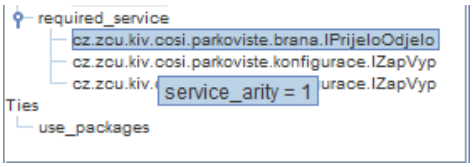
- Zobrazení tagů komponenty – kliknutím myší na horní panel s názvem zobrazené komponenty (viz obrázek J)
- Zobrazení informací o spojení mezi komponentami – kliknutí myší na požadované spojení (viz obrázek K)
- Zobrazení tagů elementů – najetím myší nad požadovaný element (viz obrázek L)
- Zvýraznění spojených elementů a jejich komponent dvojklikem myši na požadovaný element. Vzhled zvýraznění je stejný jako ve druhém případě.



Obrázek J: Zobrazení tagů komponenty



Obrázek K: Zobrazení informací o spojení mezi komponentami



Obrázek L: Tooltip s tagy elementu

Přepnutí vzhledu komponent

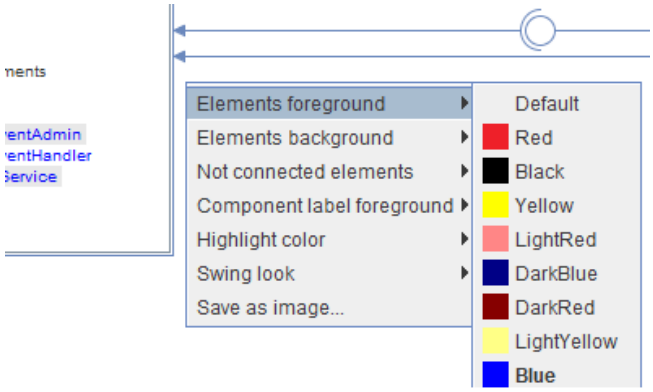
V plug-inu AIVA lze měnit reprezentaci zobrazených komponent načtené aplikace. To lze provést volbou položek v accordion menu *Component Representations* (na obrázku G označeny číslem **9**). Lze tedy vybrat ze tří možných reprezentací komponent. Po stisknutí tlačítka požadované reprezentace je graf překreslen za použití nového vzhledu komponent. V jedné chvíli je možné mít pro graf aktivní pouze jeden z možných vzhledů komponent.

Stejným způsobem je možné přepnout reprezentaci spojení mezi komponentami a layout grafu.

Menu možnosti

Plug-in AIVA má taktéž připraveno kontextové menu pr různé další akce. Kliknutím pravým tlačítkem myši v grafu nad prázdné místo se vyvolá toto menu (viz obrázek M). V grafu lze tedy měnit několik barevných vyznačení. Podle obrázku M jsou to odshora:

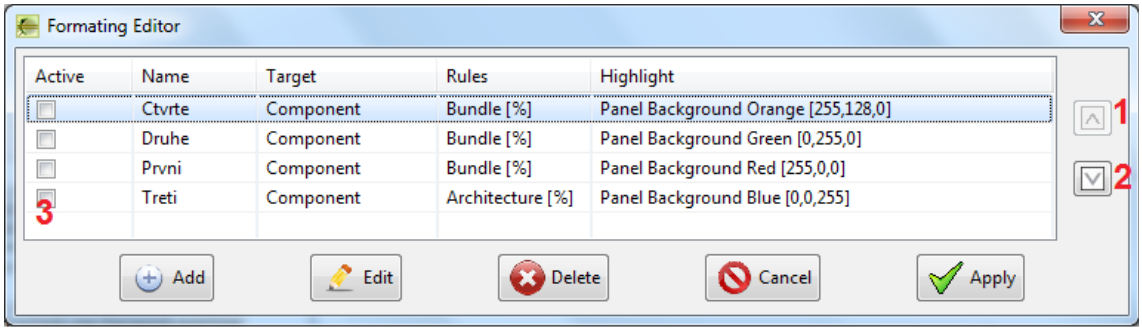
- Text elementů
- Pozadí elementů
- Pozadí elementů, které nejsou součástí žádného spojení
- Pozadí panelu v horní části zobrazené komponenty
- Barva použitého zvýraznění (viz **Interaktivní zobrazení informací**)



Obrázek M: Kontextové menu v grafu

Filtrování grafu podle pravidel

Pravidla pro filtrování lze vytvářet a aktivovat v dialogu spuštěném tlačítkem 3 na obrázku G. V zobrazeném dialogu (viz obrázek N) jsou vypsána existující pravidla. Ta lze upravovat, mazat nebo měnit jejich pořadí (tlačítka 1 a 2). Jejich pořadí hraje důležitou při aplikování pravidel se stejným typem vyznačení. Některé typy vyznačení lze totiž aplikovat pouze jednou (např. barva textu elementů) a proto je použita pouze barva pravidla, které je v tabulce umístěno dříve.



Obrázek N: Dialog s pravidly pro filtrování grafu

Aplikování pravidel se provede zaškrtnutím výběru u požadovaných pravidel (**3**) a stiskem tlačítka *Apply*. Dialog pravidel je poté zavřen a v grafu jsou vyznačeny výsledné prvky. Deaktivace se provede opačným postupem, tj. odškrtnutím výběru pravidel a znovu stiskem tlačítka *Apply*.

V plug-inu AIVA je možné definovat dva typy pravidel. Pro komponenty lze definovat jejich:

- typ
- jméno
- množinu hodnot tagů (nepovinné)

Pro elementy pak lze definovat:

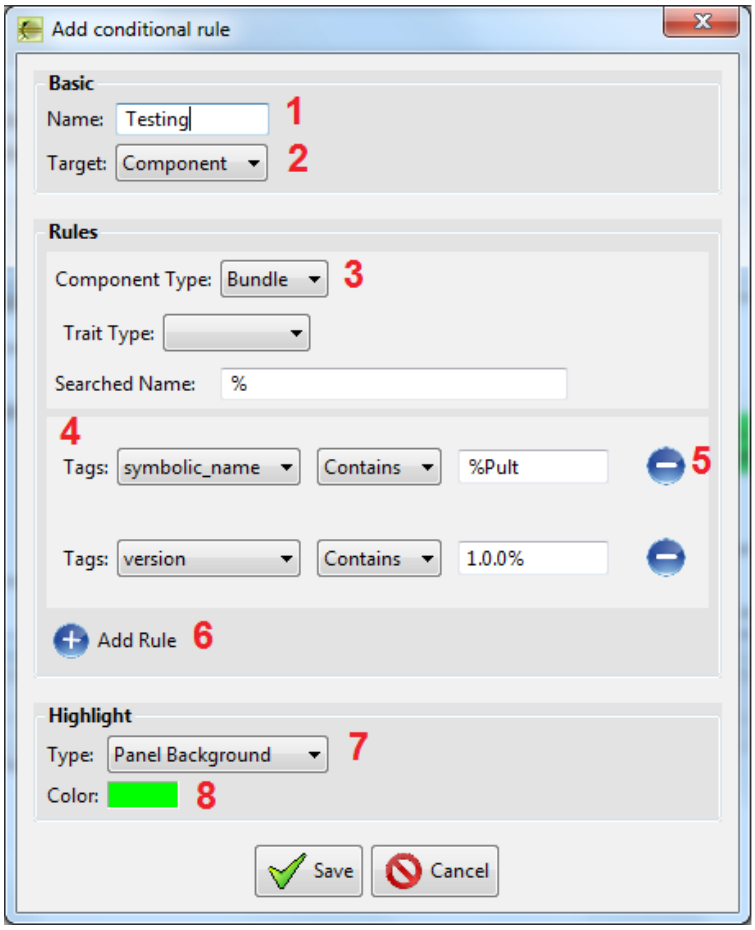
- jméno
- množinu hodnot tagů (nepovinné)

U jmen a hodnot tagů lze v programu používat zástupný znak % představující nula až nekonečno různých znaků. Počet těchto znaků není v řetězci omeze. Lze tak složit například následující řetězce:

- %
- %.event
- org.eclipse.%
- %.konfigurace.%

Na obrázku O je dialog pro vytvoření nového pravidla, který se otevře stiskem tlačítka *Add* v dialogu pravidel. Jak je vidět, pro komponentu lze definovat:

- Jméno pravidla (**1**)
- Cílový typ, tj. komponenta nebo element (**2**)
- Typ komponenty (**3**)
- Hodnoty tagů komponenty (**4**). Těchto pravidel tagů může být několik a přidávají se tlačítkem (**6**) a odebírají se tlačítkem (**5**)



Obrázek O: Dialog pro přidání pravidla pro filtrování grafu

Vytvářenému pravidlu pak lze definovat cílový zvýrazněný prvek a barvu zvýraznění. Pro komponenty nelze definovat typy trait a ani názvy elementů (oblast pod volbou **3**). Ty lze definovat pouze pokud je cílovým type pravidla element.

Možná vyznačení se poté liší podle typu zvoleného pravidla a jsou přesně popsána v textu diplomové práce.

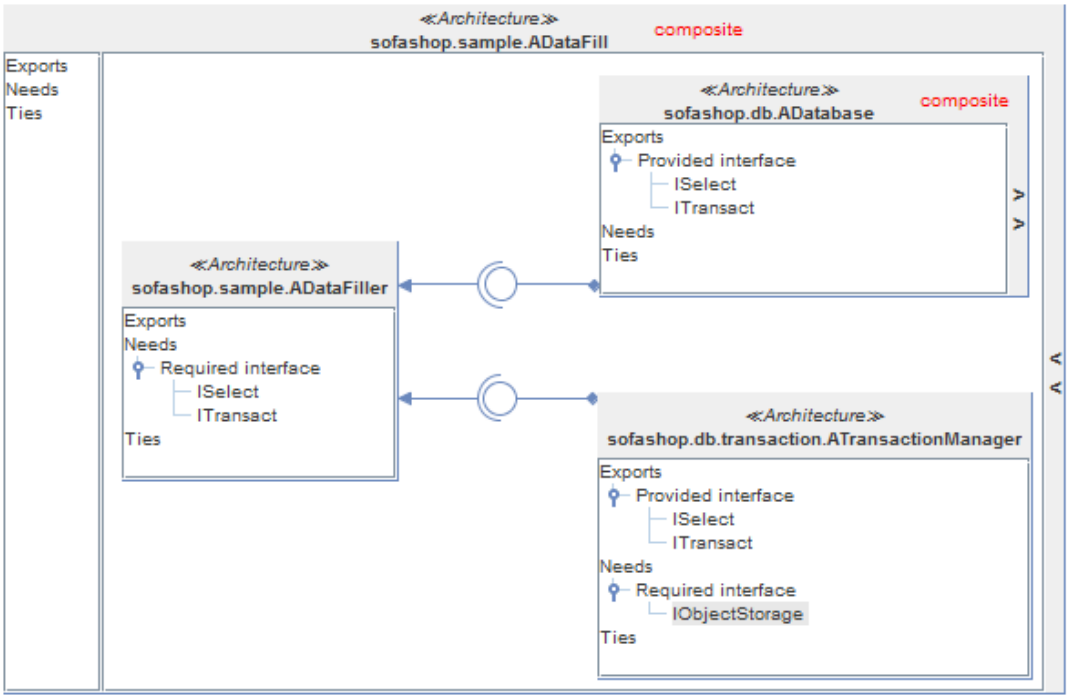
Práce s hierarchickými grafy

Načtené aplikace komponentového modelu SOFA2 mohou obsahovat komponenty, které jsou složené z další podkomponent. Základní práce s takovýmto grafem je v plug-inu AIVA stejná.

Složené zobrazené komponenty jsou pouze označeny nápisem composite v pravé horní části těla komponenty (viz obrázek P) a obsahují tlačítko pro zobrazení/skrytí grafu jejích podkomponent (na obrázku P červeně zakroužkováno). Po kliknutí na toto

tlačítko se zobrazí graf podkomponent (viz obrázek Q). Dalším kliknutím na toto tlačítko se graf podkomponent opět skryje.

Obrázek P: Složená komponenta



Obrázek Q: Složená komponenta s grafem podkomponent

Ukončení programu

Ukončení programu se provede stiskem systémového tlačítka v pravé části okna aplikace (obrázek křížku), případně přes menu položku *Exit*, umístěnou v menu *File*. Načtené projekty a vytvořená pravidla pro filtrování jsou před ukončením aplikace uložena do zvolené pracovní složky.