Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Zobrazování SOFA aplikací v AIVA

Plzeň, 2013 Bc. Jaroslav Kvapil

Originální zadání

Prohlášení

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

V Plzni dne 24. června 2013

.....

Bc. Jaroslav Kvapil

Poděkování

Na tomto místě bych rád poděkoval panu Ing. Jaroslavu Šnajberkovi za cenné rady a motivující podněty, které práci významným způsobem obohatily.

Bc. Jaroslav Kvapil

Abstract

SOFA 2 is a component model developed at the Department of Distributed and Dependable Systems at the Charles University in Prague. ComAV (Component Application Visualizer) project is developed at the University of West Bohemia in Pilsen and its purpose is to create a tool for visualization of the component-based applications. ComAV is based on Eclipse RCP platform and it does not support any component model or visualization style, but it uses plugins to extend functionality. AIVA (Advanced Interactive Visualization Approach) is a visualization plugin for ComAV project.

This thesis is aimed at improving the current implementation of AIVA plugin, primarily to enhance the visualization of applications written in SOFA 2 component framework. It also includes upgrading of the SOFA 2 loader plugin. First, requirements will be collected, then the weaknesses of the current implementation will be analysed, and then there will be an implementation of these requirements.

OBSAH

1 ÚVOD

Způsob programování se stále mění. Nejvýraznější změna v poslední době byla zcela jistě přechod na objektově orientované programování, které je dnes velmi populární a rozšířené. Avšak stále se hledají další možnosti, jak vývoj aplikací zefektivnit, což neznamená pouze urychlit, ale také zohlednit další parametry jako jsou bezpečnost, náchylnost k chybám, testovatelnost a další. Komponentově orientované programování (*Component-based Software Development*, neboli ve zkratce *CBD*) se snaží jít touto cestou a posunout vývoj softwaru na novou úroveň.

Tato práce se zabývá vizualizací komponentově orientovaných aplikací a vznikla ve spolupráci s kolegy z matematicko-fyzikální fakulty v Praze, kteří jsou autory komponentového modelu SOFA 2. Strukturu aplikací vyvinutých v SOFA 2 dokáže zobrazit vizualizační nástroj AIVA, vyvíjený na Fakultě aplikovaných věd, která je součástí Západočeské univerzity v Plzni. Cílem této práce je zpracovat připomínky od autorů SOFA 2 týkající se vizualizace a navrhnout vylepšení stávající implementace nástroje AIVA, tyto vylepšení implementovat a otestovat jejich funkčnost. Na závěr budou shrnuty výhody nové implementace.

2 ÚVOD DO KOMPONENTOVÉHO PROGRAMOVÁNÍ

Jak již bylo řečeno v úvodu této práce, komponentově orientované programování posouvá vývoj softwaru na novou úroveň. Komponentové programování čerpá z výhod objektově orientovaného programování, jako je *zapouzdřitelnost* (encapsulate) a *znovu-použitelnost* (re-use), a posouvá jej dále. V objektově orientovaném programování se pracuje s třídami, které jsou mapovány na objekty reálného světa, což je samozřejmě výhodné, protože každá třída nese zodpovědnost sama za sebe a zvyšuje se tím přehlednost výsledného kódu.

U komponentově orientovaného programování se nepracuje pouze s třídami, ale s hotovými komponentami, které obvykle pokrývají větší rozsah než samotná třída. V technické studii sepsané Felixem Bachmanem a dalšími autory a vydané v roce 2000 [1] je komponenta definována jako neprůhledná implementace, která dodržuje pravidla komponentové modelu a je připravena být nezávisle použita a nasazena třetí stranou. V této definici se skrývá několik rysů komponentového programování. První důležitá vlastnost je ve slově neprůhledná, která jasně vymezuje, že implementace je pro uživatele nedosažitelná a je odkázán pouze na jasně vymezené vstupní a výstupní body komponenty, které se většinou označují jako rozhraní. Jednotlivé komponenty musí mít jasně popsanou svou funkcionalitu a rozhraní. Kvůli této vlastnosti se komponenty často označují jako *černé skříňky*. Dále je v definici zmíněn komponentový model, který jasně vymezuje komponenty a další prvky, a který bude detailněji popsán dále. V poslední části definice je uveden nejdůležitější znak komponentově orientovaného programování a to potenciál znovu-použitelnosti. Skládáním jednotlivých komponent vzniká výsledný software.

Je tedy potřeba přesně specifikovat, co je to komponenta, jak spolu komponenty komunikují a další funkce s tím spojené. Toto definuje *komponentový model*. Bohužel neexistuje jednotná definice, a tak existuje mnoho komponentových modelů. Z těch nejrozšířenějších je to například *OSGi*, *COM+* nebo *Enterprise JavaBeans*. Tato práce se v kapitole 3 detailněji zabývá komponentovým modelem

SOFA 2. Komponentový model je pouze na úrovni teoretického popisu, který pokud je prakticky implementován, vznikne *komponentový framework*. Komponentový framework poskytuje prostředí pro běh a komunikaci komponent, případně další služby.

2.1 VÝVOJ APLIKACE POMOCÍ KOMPONENT

V úvodu této práce byla zmínka o znovu-použitelnosti již hotových komponent. Pokud je do systému potřeba přidat funkčnost, jako první je vhodné poohlédnout se, zda již neexistuje hotová a otestovaná komponenta, která řeší požadovanou funkčnost, a pokud ano, použít ji. Pokud žádná z existujících komponent nevyhovuje požadavkům, je nutné ji vytvořit.

Podobně jako u objektově orientovaného vývoje aplikace, i při komponentovém vývoji je potřeba nejdříve vytvořit *návrh* [2]. Při něm se složitější části dělí na jednodušší a ty se ve výsledku opět skládají do celků. Několik primitivních komponent může tvořit jednu složitější komponentu, dalším skládáním a spojováním komponent se nakonec vytvoří výsledná aplikace. Návrh aplikace je možné popsat jazykem UML, který má ovšem také své slabé stránky. Možná alternativa je použití *ENT meta-modelu*, vyvíjeného na Západočeské univerzitě v Plzni, kterému se bude detailněji věnovat kapitola 4. Jedná se o *obecný meta-model*, který je možné přizpůsobit konkrétnímu komponentovému modelu.

Po návrhu přichází na řadu *vývojová fáze* [2], při které jsou vyvíjeny jednotlivé *primitivní komponenty*. Pro větší pohodlí lze z ve většině komponentových modelů z obecného popisu komponenty vygenerovat základní kostru kódu, do které je postupně doplněn samotný kód. Pokud je vývoj komponenty dokončen, následuje zkompilování jejího zdrojového kódu.

V další fázi sestavení jsou komponenty podle návrhu sestaveny do *stromu sestavení (assembly tree)* [2]. Ve stromě kořen představuje výslednou aplikaci, jeho uzly jsou jednotlivé komponenty, takto se pokračuje až k listům, které reprezentují primitivní komponenty. Pokud je potřeba komponentám nastavit *konfigurační data*

(*properties*), lze tomu tak učinit ve většině případů v *xml* souboru (*assembly descriptor*).

Poslední fázi *nasazení* (*deployment*) řeší výsledné nasazení aplikace do provozu [2]. Komponenty jsou rozděleny do skupin (*deployment units*), skupiny se od sebe odlišují podle toho, na jakém serveru budou provozovány. V případě, že se nejedná o distribuované prostředí, bude existovat pouze jedna skupina. Stejně jako ve fázi sestavení, i zde probíhá nastavení *konfiguračních dat* (zde se jedná o *deployment descriptor*) ve většině případů v *xml* souboru, ovšem zde je tomu na úrovni jednotlivých skupin.

Kromě těchto čtyř fází existuje ještě další, které se říká *certifikace* [2]. Při té se ověří, zda komponenta, aplikace nebo i framework konají přesně to, co mají popsáno ve *specifikaci*. To je velmi důležité, protože pokud by například byla v komponentovém frameworku vložena komponenta, která neodpovídá svému popisu, práce s ní by (také kvůli nedosažitelnosti zdrojového kódu) nebyla možná. V případě, že je komponenta certifikována, je ověřené, že všechny její části pracují podle její definice.

2.2 SPECIFIKACE KOMPONENT

V předchozí podkapitole byla zmínka o důležitosti specifikace a certifikace komponent. Kvůli tomu, že komponenty jsou znovu-použitelné části, je klíčové, aby komponenta měla jasně popsány *funkční* i *mimo-funkční charakteristiky*, *rozhraní*, přes které komponenta komunikuje s okolím, *způsoby použití* a další důležité údaje jako je například verze komponenty [2].

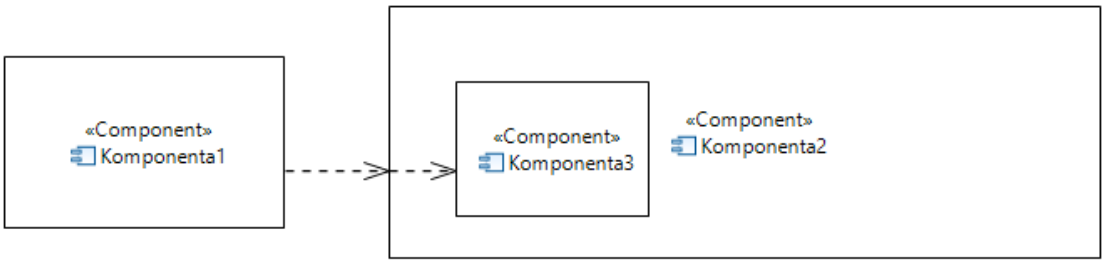
Rozhraní komponenty představuje *služby*, které komponenta *poskytuje*, a které jsou *odděleny* od konkrétní implementace. Rozhraní lze popsat mnoho způsoby, obecně však musí být uvedeno vždy to, co komponenta *vyžaduje* a jaké služby *poskytuje*. Jedna komponenta může obsahovat i několik rozhraní.

Zatímco popis rozhraní je ve většině případů jasný a srozumitelný, definice komponent, zvláště těch komplexních, je obtížná. Způsob a detail popisu závisí na

konkrétním druhu komponenty, obecně by měla být popsána *funkce komponenty*, způsob jejího *nasazení*, *kompatibilita* a případně *další nezbytné údaje*.

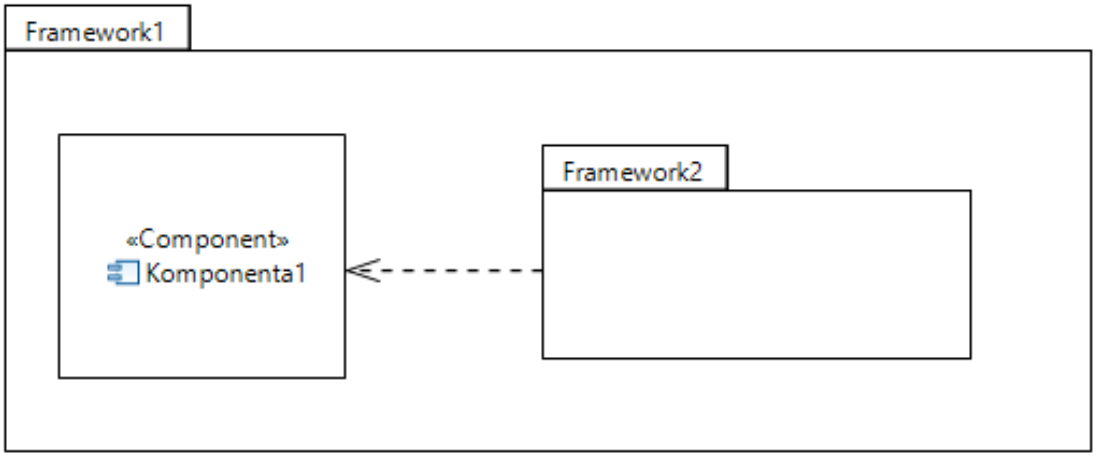
2.3 SKLÁDÁNÍ KOMPONENT

Skládání komponent je spojování dvou nebo více komponent, které spolu mají spolupracovat. Jednotlivé komponenty lze skládat různými způsoby. Typ spojení jednotlivých komponent lze rozdělit na dva druhy [1]. V prvním případě se jedná o spojení dvou nebo více komponent, které jsou na *stejné úrovni* (jsou si navzájem rovné). Druhý typ spojení je orientován na *hierarchickou strukturu*, kdy se *hlavní komponenta* (rodič) skládá z několika menších subkomponent. U tohoto typu spojení je typické, že pokud chce jiná komponenta, které je na úrovni rodičovské komponenty, komunikovat s některou ze subkomponent, musí svůj požadavek směrovat na *rodičovskou komponentu*, která požadavek předá dále (Obrázek 1).



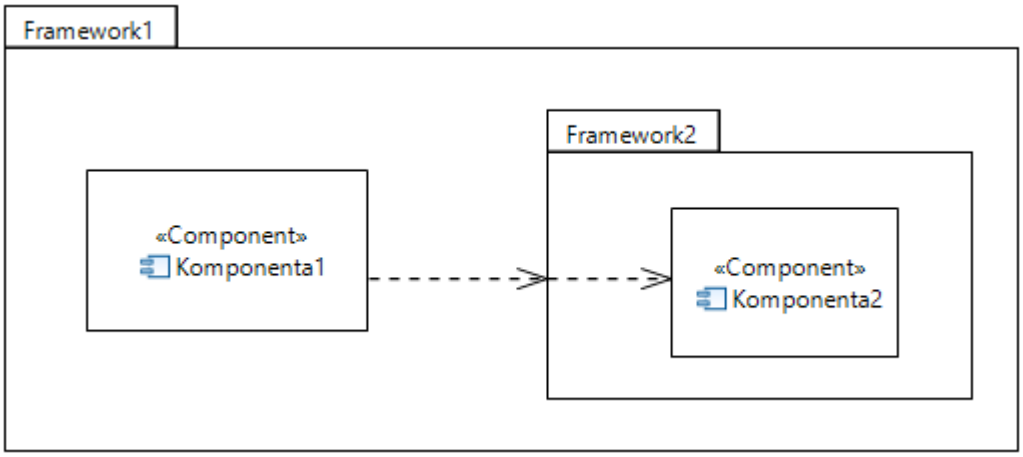
OBRÁZEK 1 – KOMUNIKACE KOMPONENT V HIERARCHICKÉ STRUKTUŘE

Na úrovni spojení komponentového frameworku a komponenty je možné vytvořit nejjednodušší spojení začleněním komponenty do frameworku [1]. Komponentové frameworky lze také hierarchicky skládat a komponenty z takto složených frameworků spojovat. Zde se rozlišují dva druhy spojení, v prvním případě je komponenta z rodičovského frameworku spojena s vnořeným frameworkem, v tomto případě se jedná o *plug-in* (Obrázek 2).



OBRÁZEK 2 – ROZŠÍŘENÍ VNOŘENÉHO FRAMEWORKU (PLUG-IN)

V dalším případě jsou spojeny komponenty z obou frameworků (rodičovského i vnořeného) [1], spoj má stejně jako v případě komunikace komponent na odlišných hierarchických úrovních dvě části, zde jako *zprostředkovatel* vystupuje zanořený framework (Obrázek 3).



OBRÁZEK 3 – SPOJENÍ KOMPONENT HIERARCHICKY SLOŽENÝCH FRAMEWORKŮ

Toto jsou obecné způsoby, jakými lze skládání komponenty, každý komponentový model pak definuje vlastní způsoby skládání komponent. Komponentový model SOFA 2 je detailněji popsán v kapitole 3.

3 SOFA 2

SOFA 2 je *komponentový model* a zároveň *framework* vyvíjený katedrou distribuovaných a spolehlivých systémů na matematicko-fyzikální fakultě Univerzity Karlovy v Praze. Komponentový model se zaměřuje zejména na *znovu-použitelnost* a *čistý design*. Je navržen na skládání komponent *hierarchicky*. Komponentou je zde myšlena zapouzdřená entita, která komunikuje s ostatními entitami (komponentami) skrz rozhraní. Komponenta může být buď jako *black-box* nebo *gray-box*.

Black-box komponentu určuje *rámec*, který zastupuje soubor komponentových rozhraní, poskytovaných i vyžadovaných, a dále určuje typ komponenty [3]. Zároveň také obsahuje specifikaci chování komponenty. Naproti tomu *gray-box* je specifikován jako architektura, která implementuje konkrétní rámec komponenty (nebo několik rámců). SOFA 2 rozlišuje architektury (komponenty) primitivní a složené. Primitivní architektura je implementace komponenty v konkrétním programovacím jazyce, složená architektura znamená složení několika subkomponent. Z toho důvodu neobsahuje složená architektura žádný funkční kód, její funkčnost je určena specifikací subkomponent a jejich složením, to znamená, že volání rozhraní komponenty je přeneseno na příslušné rozhraní subkomponenty.

SOFA 2 dále zavádí *konektory* [3], které umožňují propojení komponent různými typy spojení, například jako *sběrnici*, *sdílenou paměť* atd. Konektor obsahuje v definici *typ komunikace* a k tomu se vztahující *soubor vlastností* (např. úroveň bezpečnosti). Konektory propojují rozhraní komponent mezi sebou a umožňují tak spojit několik rozhraní na jedné lince, což se podobá například modelu serveru s několika klienty. Je také možné propojit libovolné typy rozhraní, to je využito hlavně při zapojení jako sběrnice. Konektor je stejně jako komponenta *znovu-použitelný prvek* a stejný konektor lze tedy používat napříč různými projekty.

3.1 VÝVOJ APLIKACÍ V SOFA 2

Komponenty připravené k použití jsou ve frameworku SOFA 2 uložené a dostupné v *repozitáři*. Při vývoji aplikací se používají a skládají tyto komponenty z repozitáře. Jako první se začíná s definicí architektury. Pokud jsou k dispozici všechny potřebné komponenty v repozitáři, sestavují se dohromady, v opačném případě je nutné komponentu nejprve vytvořit a do repozitáře ji umístit.

Vývoj nové komponenty lze zjednodušeně popsat v několika krocích [3]. Jako první jsou definována rozhraní, které komponenta vyžaduje a poskytuje. Tím je vytvořen rámec komponenty. Dále je nutné vytvořit funkce komponenty, což lze dvěma způsoby, buď funkčnost kompletně naprogramovat, nebo využít některé z hotových komponent a zahrnout je jako subkomponenty. Oba způsoby lze samozřejmě i zkombinovat. V tomto kontextu se hovoří o architektuře, komponenta má buď *primitivní architekturu* a to v případě, kdy neobsahuje žádné subkomponenty, nebo v opačném případě *složenou architekturu*.

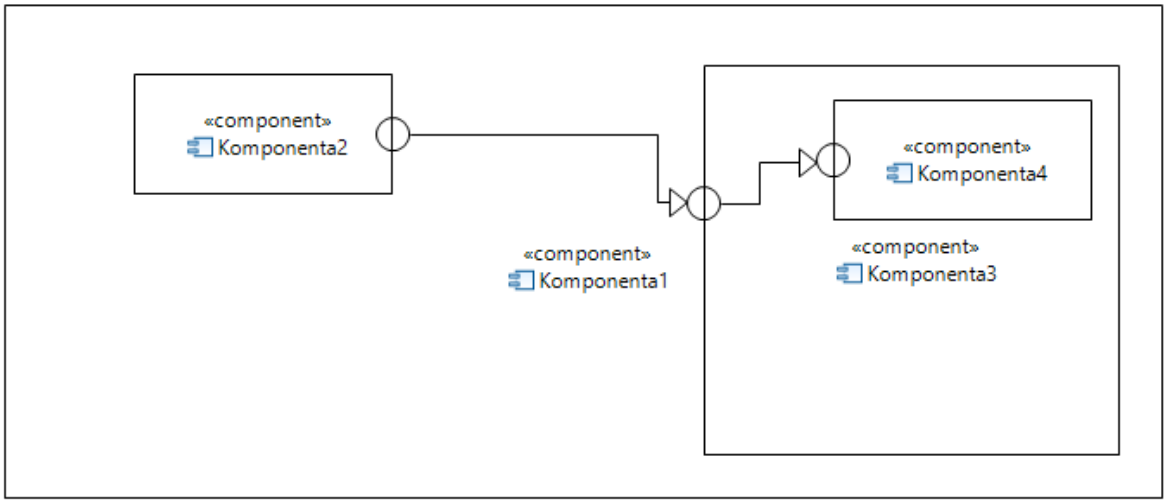
Architektura je popsána v rámci komponenty, ve *fázi sestavení* aplikace jsou rámce zaměněny za konkrétní implementaci. To se začíná od nejvyšší komponenty v architektuře a rekurzivně se postupuje k primitivním. Poté, co je aplikace sestavena a je připravena k nasazení, je vytvořen *plán nasazení* (*deployment plan*), ve kterém jsou k dispozici potřebné informace o běhu aplikace.

3.2 KOMPONENTOVÝ MODEL SOFA 2

SOFA 2 zavádí některé pokročilé rysy, které v jiných rozšířených komponentových systémech nejsou. Z těch významných je to především *hierarchická struktura*, *podpora dynamické konfigurace* aplikace nebo *distribuované prostředí* [3].

Následuje praktický příklad hierarchického složení komponent, podobnému jako byl v kapitole 2.3. Komponenta1 je na vrcholu hierarchie a skládá se ze dvou komponent, z toho jedné primitivní (komponenta2) a jedné složené (komponenta3). Komponenta3 je složena z primitivní komponenty (komponenta4). Tato primitivní komponenta je viditelná pouze pro komponentu3, to znamená, že pro ostatní

komponenty (komponenta1 a komponenta2) není viditelná a mohou s ní komunikovat pouze prostřednictvím komponenty3. Komponenty jsou černé skříňky a komunikují přes rozhraní (Obrázek 4).

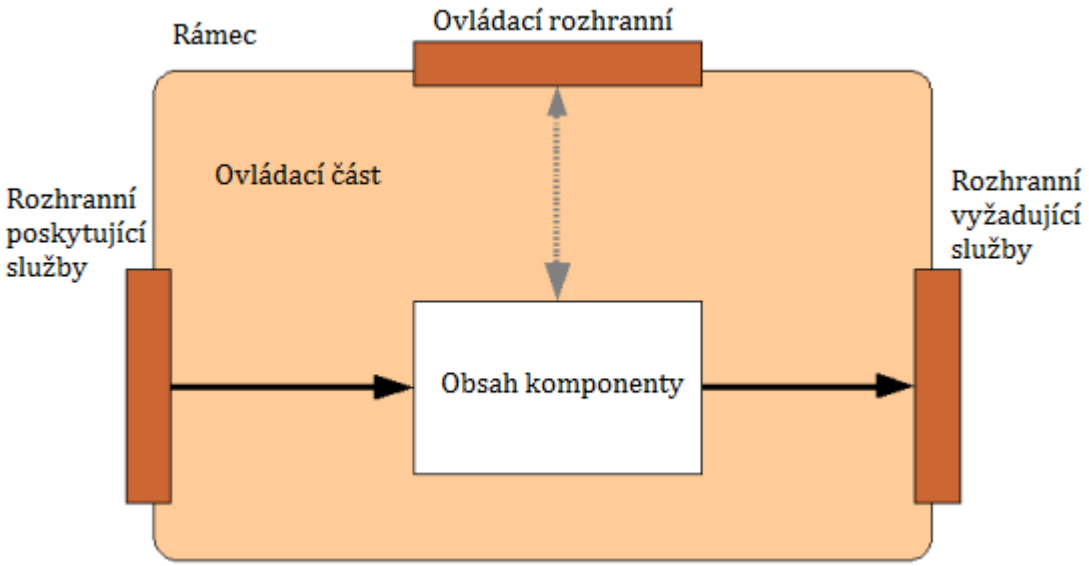


OBRÁZEK 4 – KOMUNIKACE KOMPONENT NA ODLIŠNÉ HIERARCHICKÉ ÚROVNI

Obsah komponenty je možné rozdělit na tři části:

1. *Rámec*
2. *Ovládací část*
3. *Obsah komponenty*

Rámec určuje hranici komponenty, obsahuje soubor rozhraní, kterými komponenta komunikuje se svým okolím. Pojmem hranice je myšleno rozdělení komponenty na dvě části, kde první část je viditelná pro ostatní komponenty, zatímco ostatní dva prvky leží v druhé části, kde jsou obsahem samotné komponenty a nejsou tedy pro okolí komponenty viditelné. *Ovládací část* je poskytována *SOFA 2 prostředím*. Pod *obsahem komponenty* je schována vlastní *implementace* funkcí a služeb, které komponenta poskytuje (Obrázek 5).



OBRÁZEK 5 – POHLED UVNITŘ KOMPONENTY [3]

3.3 PROSTŘEDÍ SOFA 2

Prostředí, ve kterém běží aplikace SOFA 2, je distribuované. Zavádí se zde pojem *SOFAnode* [3], což představuje oddělenou samostatnou jednotku pro běh aplikace. Každý *SOFAnode* tedy obsahuje všechny informace potřebné pro spuštění a běh aplikace včetně uložených dat. Jedna instance běhu aplikace nemůže být rozdělena na více než jeden *SOFAnode*. *SOFAnode* se skládá z několika částí:

- **Deployment Dock** – *Deployment Dock* je nástroj zodpovědný za spouštění a běh komponent. *Deployment Dock* stáhne při sestavení aplikace data z *repozitáře*, vytvoří instance komponent a konektorů mezi komponenty a nakonec spustí aplikaci.
- **Deployment Docks Registry** – Protože různé instance *Deployment Dock* běží na fyzicky odděleném stroji, je potřeba zajistit komunikaci mezi ostatními uzly, což zahrnuje také synchronizaci informací o sestavení mezi všemi dostupnými *Deployment Docky*. Tuto úlohu má na starost *Deployment Docks Registry*. Každý *Deployment Dock* je při svém spuštění zodpovědný za registraci do registru, jinak nebude pro okolí viditelný. Samozřejmě je v jedné instanci *SOFAnode* k dispozici pouze jedna instance *Doplyment Docks Registry*.

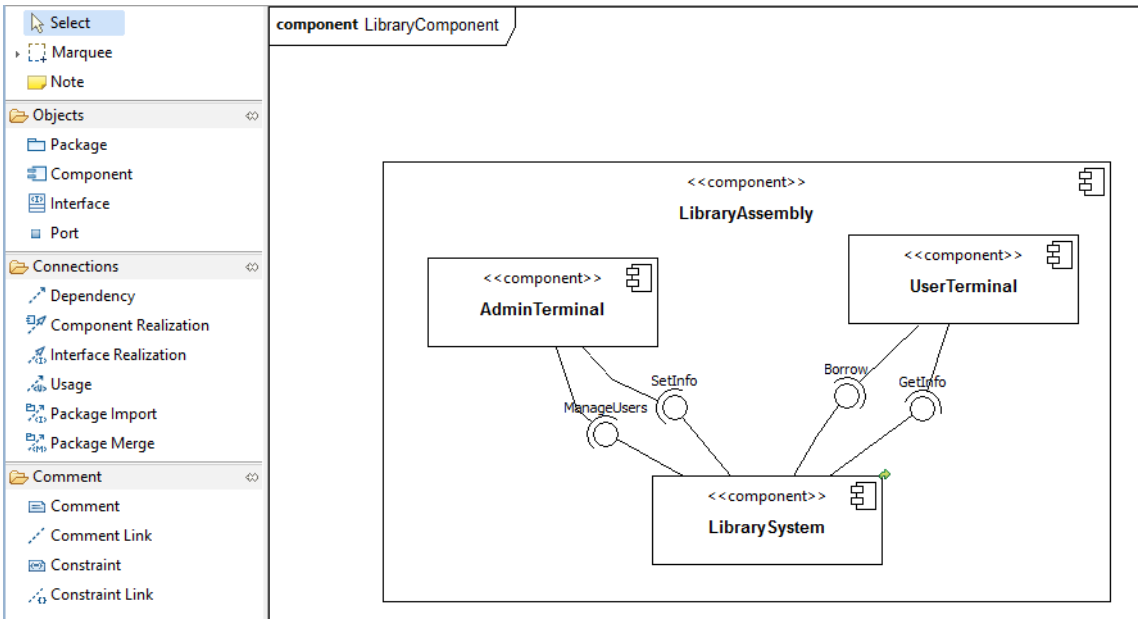
- **Repozitář** – Distribuovanému prostředí SOFA 2 je nutné zajistit jedno centrální uložiště, ve kterém bude moci každá entita v distribuovaném prostředí získat potřebná data. Pro tyto účely je vytvořen *repozitář*, ve kterém jsou uloženy všechny potřebné informace, data a metadata. K těmto datům mají přístup ostatní části systému, například *Deployment Dock* při sestavování aplikace.
- **Global Connection Manager** - *Global Connection Manager* zodpovídá za správné propojení konektorů, které jsou popsány v úvodní části o SOFA 2. Tuto funkci využívá *Deployment Dock* během sestavování aplikace. Stejně jako u *Depoyment Docks Registry* je v jedné instanci *SOFAnode* k dispozici pouze jedna instance *Global Connection Manager*.

3.4 SOFA 2 UML

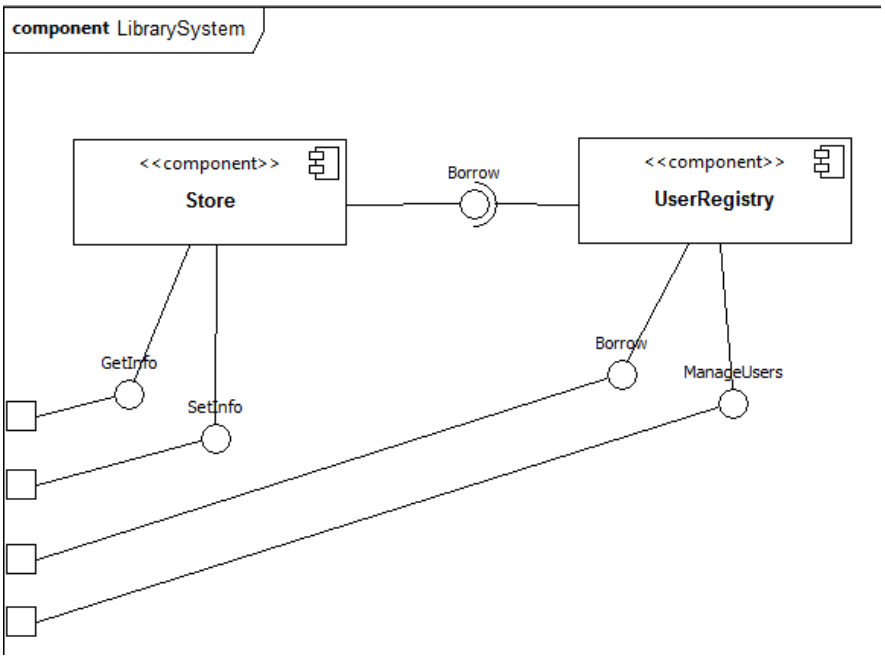
Pro jednodušší vývoj aplikací v komponentovém frameworku SOFA 2 je k dispozici několik pluginů do nástroje Eclipse, jejichž sloučením vznikne prostředí, ve kterém se dají vytvářet entity SOFA 2 pomocí UML [4]. Aby bylo možné prostředí provozovat, je nutné do nástroje Eclipse (ve verzi Modeling Tools, v jiném případě je nutné dále doinstalovat i další pluginy, které verze Modeling Tools obsahuje) doinstalovat 3 pluginy:

- SOFA 2 IDE
- Topcased UML2 Editor – pro vlastní UML modelování
- SOFA 2 UML – pro následné generování struktury SOFA 2

Tím je získáno prostředí pro modelování pomocí UML, ze kterého lze jednoduše generovat strukturu SOFA 2 aplikace. Poté je samozřejmě nutné do vygenerovaných zdrojových souborů dopsat výkonný kód, ale samotná struktura komponent je již připravena k použití. Na následujících obrázcích (Obrázek 6 a Obrázek 7) je zobrazena aplikace *Library*, která je modelována v Topcased UML2 Editor.



OBRÁZEK 6: MODELOVÁNÍ APLIKACE LIBRARY POMOCÍ UML



OBRÁZEK 7: POHLED UVNITŘ KOMPONENTY LIBRARYSYSTEM

4 ENT META-MODEL

ENT meta-model je *MOF (Meta-Object Facility) M3 model*, vyvíjený na katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni. MOF M3 znamená, že se jedná o *abstraktní* meta-model (jazyk), který souží k popisu konkrétních komponentových modelů [5]. Definuje základní elementy a pravidla pro jejich spojování. Konkrétní model *MOF M2* pak slouží k popisu aplikací jednotlivých komponentových modelů, pro něž je navržen a určen. Nabízí se zde analogie k *XML dokumentu* a jeho konkrétní implementaci, například *HTML*.

V současnosti je pro popis architektur aplikací nejvíce používán jazyk UML. Jeho nevýhoda je oproti ENT meta-modelu v malé úrovni abstrakce. Vyšší úroveň abstrakce umožňuje pro odlišné komponentové modely lze v ENT meta-modelu vytvořit *konkrétní MOF M2 model*, který bude *vyhovovat* jeho potřebám a bude moci odpovídajícím způsobem charakterizovat jeho části (komponenty).

ENT meta-model používá pro charakterizaci komponent *traity*, které umožňují náhled na komponentu z různých *úhlů pohledu* [6]. Uživatel je tak odstíněn od nepotřebných informací. ENT meta-model se skládá ze dvou úrovní: *modelová úroveň*, která definuje hlavní charakteristiky komponentového modelu aplikace a *aplikační úroveň*, definující konkrétní komponenty, jejich rozhraní a závislosti ve zkoumané aplikaci.

4.1 OBECNÝ POPIS ENT META-MODELU

Na vrcholu hierarchie meta-modelu je množina přípustných typů komponent komponentového modelu. Typ komponenty je definován jako úplný soubor potencionálních typů *traitů*, popisujících možné skupiny, které daná komponenta podporuje.

Komponentový model definuje konkrétní komponenty, které obsahují viditelný soubor rozhraní, ostatní je skryto jako *černá skříňka*. Tyto prvky jsou identifikovány

traity. Zde se samozřejmě řeší popis komponent na úrovni komponentového modelu, ne konkrétních instancí komponent.

4.2 Systém klasifikace

ENT meta-model charakterizuje rozhraní komponent z osmi druhů pohledu, nazývanými *dimenzemi* [6]. Každá dimenze umožňuje odlišný pohled na komponentu. Dimenze mají předdefinované hodnoty. Zde je seznam dimenzí ENT meta-modelu (ponecháno v originále, český ekvivalent by se v některých případech hledal obtížně):

$Dimenze_{ENT}$ = {Nature, Kind, Role, Granularity, Construct, Presence, Arity, Lifecycle} kde

- Nature = { syntax, semantics, extra-functional }
- Kind = { operational, data }
- Role = { provided, required, neutral }
- Granularity = { item, structure, compound }
- Construct = { constant, instance, type }
- Presence = { mandatory, permanent, optional }
- Arity = { single, multiple }
- Lifecycle = { development, assembly, deployment, setup, runtime }

Tato struktura je výsledkem analýzy různých komponentových modelů a je *jádrem* ENT meta-modelu. Pokud by v budoucnu popis komponent pomocí těchto osmi dimenzí nebyl dostačující, je možné počet dimenzí rozšířit. Stejně tak je možné použít pouze *podmnožinu* dimenzí, což je i případ u komponentového modelu SOFA 2, kde se bude pracovat s dimenzemi: { Contents, Kind, Role }.

4.3 Popis komponentového modelu

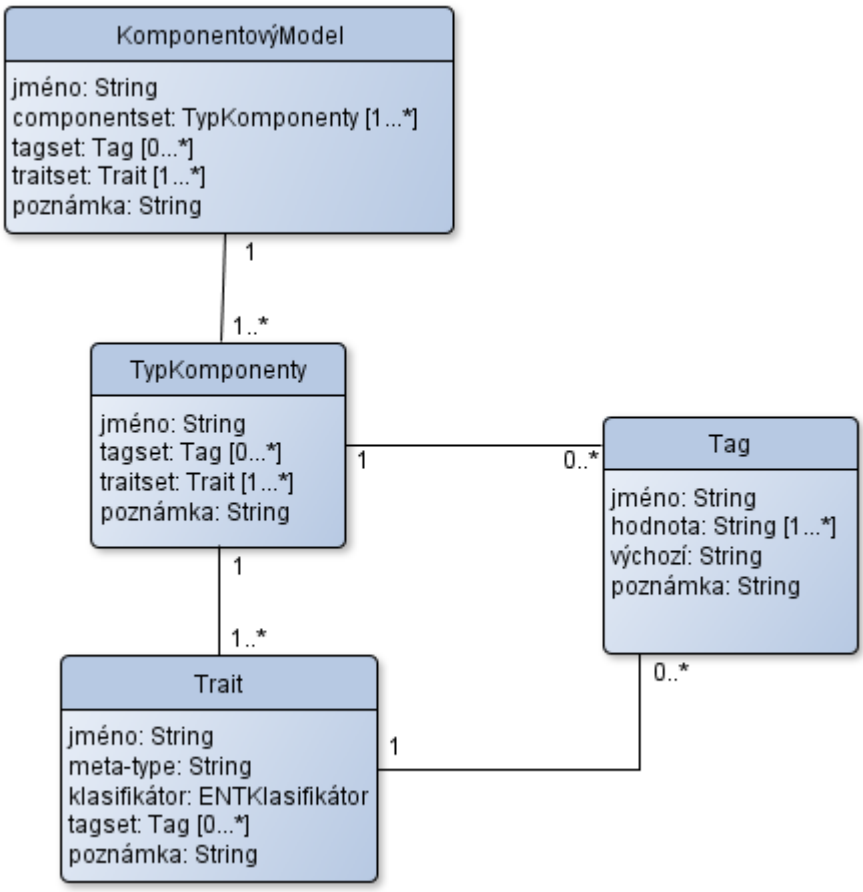
Komponentový model je definován dvojicí *(jméno, C_s)*, kde C_s představuje soubor možných *typů komponent*. Typ komponenty je definován jako trojice *(name, tagset, T)*, kde T je *soubor traitů*, které charakterizují komponentu. Typ komponenty dále

obsahuje *soubor tagů*, které uchovávají *dodatečné informace*, užitečné pro komponentový model, ale které nelze popsat pomocí traitů.

Trait je definován pěticí (name, meta-type, K, tagset, extent). *Meta-type* je typ rozhraní komponenty, K je *klasifikátor*, o kterých byla řeč v minulé podkapitole. Trait dále obsahuje *soubor tagů* pro doplňkový popis a extend definuje *maximální počet prvků v traitu*. Jména tagů v rámci jedné komponenty musí být jedinečná. Trait zastupuje jednotlivé prvky komponenty, které jsou pro okolí *viditelné*. Traity využívají klasifikační systém, kterým se zabývala předchozí kapitola, takže každý typ traitu obsahuje specifikované hodnoty pro různé druhy dimenzí.

Tag je definován jako trojice (name, $valset_i$, d_i), kde $valset_i$ je *soubor přípustných hodnot* a d_i představuje *výchozí hodnotu*, která je buď z množiny přípustných hodnot, nebo nemusí být nastavena (ϵ). Tag slouží k uchování dodatečných informací prvků komponent, které nejsou vhodné popisovat pomocí traitů [7].

Datová vrstva komponentového modelu obsahuje 4 entity, které jsou popsány výše (Komponentový model, Typ komponenty, Trait a Tag – Obrázek 8).



OBRÁZEK 8 – DATOVÁ VRSTVA KOMPONENTOVÉHO MODELU [8]

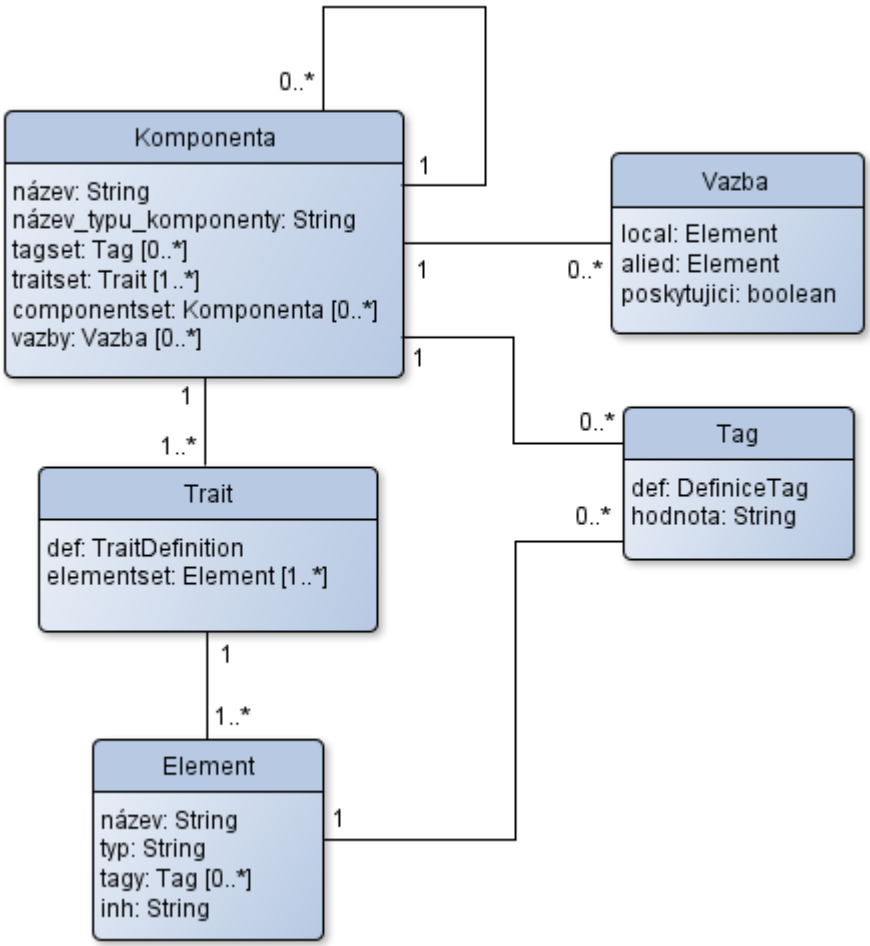
4.4 POPIS APLIKAČNÍHO MODELU

Na *úrovni aplikačního modelu* se řeší použití modelu na konkrétní aplikaci a jejích komponentách. Na úrovni komponentového modelu jsou popsány možné druhy komponent a aplikační model se na tyto prvky odkazuje. To znamená, že je definován soubor traitů, které jsou přiřazovány příslušným komponentám [8].

Komponentová aplikace je definována jako *acyklický graf*, který je složený z trojice *(C, B, m)*, kde C značí *soubor komponent*, B jsou *vazby mezi komponentami* a *m* je *hlavní komponenta* (samozřejmě $m \in C$). Je zde zaveden pojem *kontext aplikace*, který označuje soubor všech dostupných komponent v prostředí, kde je aplikace nasazena. *Konzistentní aplikace* má k dispozici všechny potřebné komponenty.

Konkrétní komponenta je definována prvky: *název*, *název typu komponenty*, *tagy* a *soubor traitů*. Další definice pojmů jako například *typ komponenty* apod. nejsou pro tuto práci bezpodmínečně nutné a jejich definice jsou případně dostupné v literatuře [7].

V případě komponentového modelu SOFA 2 je potřeba brát v úvahu jeho *hierarchickou strukturu*, kdy může být jedna komponenta *složená* z několika dílčích subkomponent. Pokud se jedná o primitivní komponentu, ta již neobsahuje žádné další subkomponenty. Zde je zmínit dva speciální druhy spojení mezi komponentami a subkomponentami a to *subsumption* (*zahrnutí*) a *delegation* (*pověření*). Jedná se o *vnitřní vazby* v neprimitivní komponentě. Dílčí subkomponenty musí být *stejného* komponentového typu jako hlavní komponenta.



OBRÁZEK 9 – DATOVÁ VRSTVA APLIKAČNÍHO MODELU [8]

4.5 Reprezentace SOFA 2 v ENT modelu

Komponentový model SOFA 2 pracuje s několika druhy entit, které jsou vhodné pro analýzu aplikace.

Entita *Assembly* je vstupní bod pro analýzu konkrétní aplikace, protože obsahuje informace o použitých komponentách, tedy entitách *Architecture*. Entitu *Architecture* lze považovat za konkrétní komponentu. Každá entita *Architecture* má dále odkaz na entitu *Frame*, která popisuje především poskytované a vyžadované rozhraní. Oba typy rozhraní jsou v *ENT modelu* reprezentovány pomocí *traitů* – *Required Interface* a *Provided Interface* [9]. Entity *Architecture* a *Frame* dále mohou obsahovat názvy parametrů, které jsou uloženy v traitu *Properties*. Hodnoty těchto parametrů jsou dostupné v entitě *DeploymentPlan*, což je vzhledem k již popsaným entitám nejvyšší entita, která dále obsahuje různé informace o nasazení aplikace. Z hlediska analýzy aplikace jsou zajímavé pouze hodnoty parametrů.

5 KNIHOVNA JGRAPH

5.1 STRUKTURA KNIHOVNY

Knihovna *JGraphX* je rozšířená knihovna pro vytváření grafů a diagramů, grafem zde není myšlena grafická vizualizace hodnot, ale definice grafu zde vychází z teorie grafů, podle které se graf definuje jako množina vrcholů, které jsou spojeny hranami. Knihovna poskytuje funkce pro běžnou uživatelskou interakci, jako je zvýraznění vrcholů a hran, jejich přesouvání na pracovní ploše a další podobné činnosti.

V datovém modelu je entita pro vrcholy i spoje jednotně pojmenována jako *cell* [10], tohoto označení se bude držet i tato práce. Pojem *cell* bude označovat obecně objekt, který může být buď vrchol, nebo hrana mezi vrcholy. To lze nastavit metodami `setVertex()` nebo `setEdge()`. Zde je seznam důležitých parametrů, které má objekt cell:

```
Object value;
Boolean isVertex, isEdge, collapsed;
mxICell parent, source, target;
List<Object> children, edges;
```

- Hodnota `value` představuje textový popisek na příslušném objektu, může mít hodnotu *null*.
- Logické hodnoty `isVertex` a `isEdge` určují, zda je objekt ve stavu vrcholu nebo hrany. Výchozí stav je vrchol, oba stavy zároveň nejsou přípustné, tudíž pokud se nastaví proměnná `isVertex` na hodnotu `true`, je automaticky proměnná `isEdge` nastavena na `false`.
- Proměnná `collapsed` určuje, zda jsou viditelné potomci (viz dále v kapitole 5.4)
- Hodnota `parent` odkazuje na rodičovský objekt. V případě, kdy graf není hierarchicky strukturován, je zde odkaz na výchozí objekt rodiče z instance grafu: `graph.getDefaultParent()`. Naopak v seznamu `children` si rodičovský objekt uchovává své potomky.
- Parametry `source` a `target` jsou pro objekty, které jsou ve stavu hrany a představují zdrojový a cílový objekt k propojení. Z opačného pohledu u objektů ve stavu vrcholu je seznam příslušných hran uchován v parametru `edges`.

Třída, která uchovává množinu vrcholů a jím příslušných hran, je pojmenována *mxGraph* [10]. Tato třída obsahuje i další metadata o grafu, například zda je povoleno a případně jakým způsobem je možné manipulovat s vrcholy, různé nastavení týkající se vizuální podoby a množství dalších detailů, jejichž kompletní přehled je k dispizici v API knihovny. Třída také definuje metody pro vytváření a manipulaci objektů. Jsou zde k dispozici připravené metody pro snadné vkládání objektů cell do grafu:

```
//Vložení vrcholu
graph.insertVertex(Object parent, String id , Object value, double x,
double y, double width, double height);

//Vložení hrany
graph.insertEdge(Object parent, String id , Object value, Object
source, Object target);
```

Další soubor metod je pro manipulaci s vytvořenými objekty:

```
//Změna velikost objektu, v parametru bounds jsou nové rozměry
graph.resizeCell(Object parent, mxRectangle bounds);

//Posunutí objektů v cells o hodnoty dx a dy
graph.moveCells(Object[] cells, double dx, double dy);

//A další metody
...
```

Operace vytváření a manipulace s objekty je vhodné provádět ve stavu, kdy výsledný graf nereaguje okamžitě na změny (je v tzv. „zmraženém stavu“). Tento postup je spíše doporučením než nutností [10]. Po dokončení všech změn jsou pak tyto zásahy přeneseny na plátno. Jinými slovy, změny je dobré provádět *transakčně*. K tomu slouží následující syntaxe:

```
graph.getModel().beginUpdate();
try
{
      //Zde probíhají operace vytváření a manipulace s objekty
      ...
}
finally
{
      //Ukončení všech změn
      graph.getModel().endUpdate();
}
```

Instance třídy *mxGraph* je ve finální fázi předána do třídy *mxGraphComponent*, na kterou se následně provede vykreslení grafu. Knihovna *JGraphX* využívá Java

knihovny *Swing*, proto třída *mxGraphComponent* je potomek třídy z této knihovny, konkrétně *JScrollPane*.

5.2 OBJEKT CELL A SPOLUPRÁCE S KOMPONENTY KNIHOVNY SWING

Objekt cell může být ve dvou stavech: vrchol nebo hrana. Nativní reprezentace vrcholu je jako čtyřúhelník, kterému lze nastavit parametry výšku a šířku. Co se týče grafické reprezentace, lze samozřejmě upravit další nastavení, například nastavení barev pozadí a ohraničení apod.

Knihovna JGraphX umožňuje na vizuální reprezentaci objektu cell umístit i jiné komponenty z knihovny Swing, k tomuto účelu slouží třída mxCellOverlay, která implementuje rozhraní mxICellOverlay. Umístit lze tyto objekty pouze v případě, že je cell ve stavu, kdy reprezentuje vrchol. V základu lze do této třídy vložit pouze objekt ImageIcon, to lze ovšem rozšířit způsobem, kdy se vytvoří nová třída, která je potomkem JComponent a dále implementuje rozhraní mxICellOverlay:

```
public class MyMxCellOverlay extends JComponent implements
mxICellOverlay
```

Tato třída *MyMxCellOverlay* bude obsahovat funkce pro vložení dalších Swing komponent, které je možné vkládat metodou `add()`, tu třída zdědila z rodičovské třídy JComponent. Jelikož ale zároveň implementuje rozhraní mxICellOverlay, je možné instanci předat pro vykreslení na objekt cell, který reprezentuje vrchol. Tím je možné zajistit i vložení více různých instancí Swing komponent na jednu instanci cell.

5.3 NASTAVENÍ POZIC A LAYOUTY

Ještě před vykreslením grafu do *Swing komponenty* je potřeba definovat pozice vrcholů, případně i body, kterými mají procházet spoje. To lze dvěma způsoby, pokud potřebuje mít vývojář kontrolu nad umístěním vrcholů, může jej definovat ručně. Takový postup je ovšem velmi zdlouhavý, navíc v tomto případě knihovna neřeší případné problémy, které se objevují například při překrytí objektů, a proto JGraphX

nabízí automatické řešení pomocí *layoutů*. V současné době je k dispozici celkem 10 druhů layoutů, každý má svou vlastní strukturu rozmístění objektů [10]. Některé z těchto layoutů jsou velmi jednoduché a navržené pro specifickou oblast použití. Například plugin AIVA využívá pouze 3 layouty, jejichž využití se zde hodí:

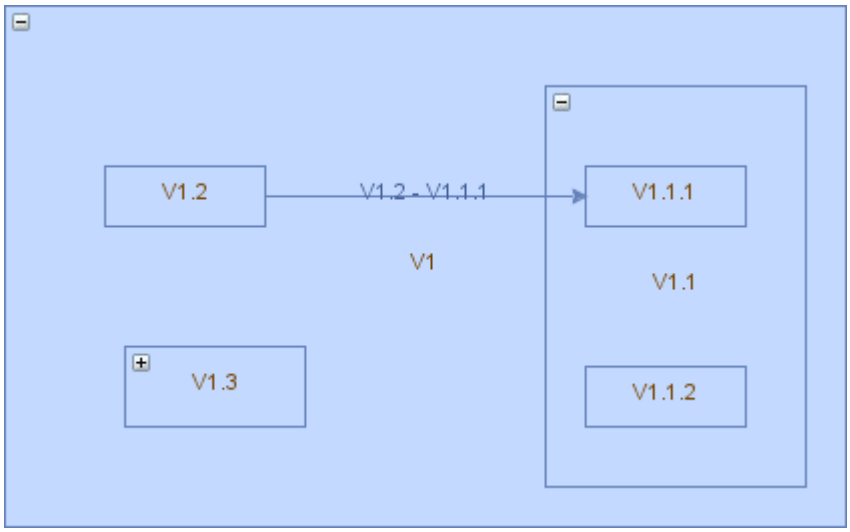
- Hierarchický layout (Hierarchical Layout)
- Kružnicový layout (Circle Layout)
- Organický layout (Organic Layout)

Hierarchický layout se dále dělí na dvě skupiny, *vertikální* a *horizontální*. To se projeví pouze v orientaci vrcholů, kdy je plátno natočeno na šířku nebo na výšku. Jedná se o nejsložitější layout v knihovně, algoritmus rozmístění vrcholů se skládá celkem ze tří hlavních fází – odstranění cyklů, stanovení pořadí vrcholů a umisťování vrcholů. Detailní popis algoritmu by zabral několik desítek stran a není předmětem této práce. Jak název napovídá, hierarchický layout je vhodný pro hierarchicky orientovaný graf.

Ostatní dva druhy layoutů nejsou vhodné pro hierarchickou strukturu vrcholů, což ale úplně nevylučuje jejich možné nasazení na takovou strukturu. V takovém případě se musí layout aplikovat postupně na všechny vrcholy obsahující další vrcholy. Navíc se, zejména u složitějších případů, musí stanovit a dodržet pořadí, ve kterém se bude layout na tyto vrcholy aplikovat. Je tedy nutné předem otestovat všechny možné případy, na které bude layout aplikován, což v případě, kdy nejsou analyzované aplikace předem známé, nelze. Tím je zdůvodněno, proč pro práci s hierarchickou strukturou je uvažováno použití pouze hierarchického layoutu.

5.4 HIERARCHICKÁ STRUKTURA

Objekty cell je možné do sebe hierarchicky vnořovat [10]. V případě, že je objekt cell ve stavu vrcholu, je možné přímo do něj vkládat další objekty cell. To se promítne tak, že vložené objekty cell (vrcholy i hrany) jsou vykresleny do rodičovského objektu cell (Obrázek 10). Parametr collapsed u rodiče určuje, zda objekty cell, které jsou hierarchicky pod objektem, mají být viditelné.



OBRÁZEK 10 – UKÁZKA HIERARCHICKÉ STRUKTURY

V případě, že je cell ve stavu hrany, které spojuje dva vrcholy, je potřeba nastavit parametry *source* a *target*. Do objektu cell ve stavu hrany je teoreticky možné vkládat potomky, při vykreslení jsou tyto potomci vynecháni, stejně tak nastavení parametru *collapsed* nemá žádný vliv na vizuální podobu hrany.

Parametr *collapsed* má význam u vrcholů, které obsahují další vrcholy jako potomky. Na předchozím obrázku (Obrázek 10) je zobrazena struktura, kde vrchol V1.3 má hodnotu parametru *collapsed* nastavenou jako `true`, tím pádem je ve svinutém stavu, naproti tomu u vrcholu V1.1 je hodnota *collapsed* `false` a objekt se nachází v rozvinutém stavu.

Pokud je objekt ve svinutém stavu, jeho rozměry (šířka a výška) se neřídí podle standardně nastavených rozměrů objektu, ale dojde ke zmenšení objektu na minimální možnou velikost. Pokud je potřeba tuto velikost ručně nastavit, je to možné pomocí alternativních rozměrů:

```
cell.getGeometry().getAlternateBounds().setWidth(double value);
cell.getGeometry().getAlternateBounds().setHeight(double value);
```

Zde je dobré zdůraznit, že toto je jediný způsob, jak lze svinutému objektu cell nastavit velikost. Pokud je objekt cell pouze vložen standardním postupem, kdy je zavolána pomocná metoda ve třídě `mxGraph` (kapitola 5.1), objekt má ve svinutém stavu nejmenší možnou velikost, která se řídí především textovým popiskem na objektu.

6 COMAV A AIVA

6.1 COMAV

Component Application Visualizer (*ComAV*) je nástroj pro *zpětné inženýrství* (též také zpětná analýza, anglický zažitý termín je *reverse-engineering*) komponentově orientovaného softwaru. Poskytuje prostředí pro *analýzu* a *vizualizaci* aplikací napsaných v podporovaném komponentovém frameworku. Není vázán na konkrétní komponentový model nebo styl vizualizace. Sám o sobě tedy nástroj neumí nic načítat ani zobrazovat, tuto funkcionalitu lze rozšířit pomocí *pluginů*. *Modularita* projektu je výhodná vlastnost, jelikož projekt nezávisí na pevně implementovaných částech, ale je možné ho *přizpůsobit* nejrůznějším požadavkům. První typ pluginů jsou *loadery*, které slouží k načítání konkrétního komponentového modelu. Vlastní zobrazování je řešeno pomocí dalšího typu pluginu *pro vizualizaci*. Do této skupiny patří například *AIVA* (*Advanced Interactive Visualization Approach*).

Prostředí, ve kterém ComAV pracuje, je vytvořeno jako *Eclipse RCP* (E*clipse Rich Client Platform*). Obecně jsou v Javě RCP nástroje (Eclipse RCP, NetBeans RCP) velmi výhodné zejména v začátcích projektu (ale nejen v začátcích), jelikož oproti implementování vlastního řešení, například využitím knihovny Swing nebo AWT, *posouvají* projekt *velmi rychle kupředu*. Navíc mají *ověřenou* a *funkční architekturu*, takže nepřinášejí další problémy v pokročilých fázích vývoje. ComAV má proto velmi podobné ovládání jako vývojový nástroj Eclipse. Načtené aplikace jsou organizovány do projektů, kterými lze listovat v levé části okna, hlavní část okna je pro vlastní vizualizaci, dále je zde konzole pro výpis zpráv a seznam aktivních pravidel (závisí na pluginu). V horní části okna je standardní menu pro ovládání.

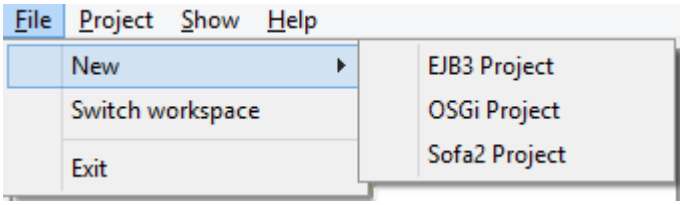
ComAV je složeno ze třech hlavních komponent - *jádro*, *knihovny* a *ENTMM* [11]. Pro svůj běh vyžaduje ještě nejméně *dva pluginy*, jeden pro načítání komponentového modelu (*loader*), druhý pro zobrazování načtených dat (*visualizer*). ComAV jádro obsahuje hlavní okno aplikace, místa pro rozšíření pomocí pluginů a pravidla pro správu projektů v pracovním prostoru. Jedná se o základní část aplikace.

ENTMM je *implementace* ENT meta-modelu. Poskytuje vrstvu pro uchování dat popisujících strukturu analyzovaných aplikací a komponentového modelu. Dále dokáže ukládat a načítat ENT strukturu v XML souborech.

Komponenta *ComAV knihoven* je pro snadnější používání cizích knihoven v celém projektu. Vizualizační plugin může například využívat některou z knihoven pro vizualizaci grafů.

6.1.1 Loader plug-in

Po přidání nového loaderu, je plug-in zaveden při startu ComAV aplikace a je k dispozici v menu položce *New -> [Some] Project* (Obrázek 11). Jak již bylo zmíněno, funkce loaderu je načítání aplikace vyvinuté v komponentovém frameworku, v tom případě, aby bylo možné provádět zpětnou analýzu, musí loader obsahovat specifikaci příslušného komponentového modelu. Jak je vidět na obrázku, v současné době jsou k dispozici loadery pro komponentové modely *OSGi*, *EJB3* a *SOFA 2* [11].



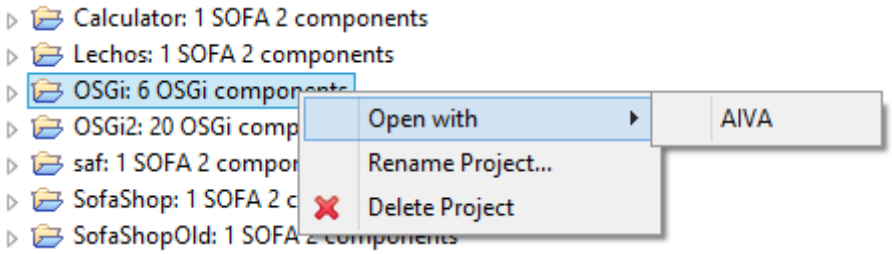
OBRÁZEK 11 – VÝBĚR LOADER PLUGINU PŘI VYTVOŘENÍ NOVÉHO PROJEKTU

Každý nový loader musí implementovat *handler* s metodou `execute()`, navíc tento handler musí být *potomek abstraktního handleru*, který je k dispozici v jádře ComAV. Metoda `execute()` pouze dohlíží na to, zda načítání proběhlo v pořádku, pokud ano, vrací jako výsledek hodnotu *true*. Samotné načítání je třeba implementovat v metodě `loaderExecute()`, která má za úkol nejprve *vytvořit průvodce* pro přidání projektu a poté provést *analýzu* programu.

6.1.2 Vizualizační plug-in

Druhý typ plug-inů slouží pro *rozšíření způsobu vizualizace*. Stejně jako předchozí loader plug-in je načítán při startu aplikace [11]. Rozšíření se projeví v *kontextovém*

menu po kliknutí pravého tlačítka na myši v seznamu projektů (Obrázek 12). Konkrétní vizualizace je *nezávislá* na komponentovém modelu.



OBRÁZEK 12 – OTEVŘENÍ PROJEKTU VIZUALIZAČNÍM PLUGINEM

Vizualizační plug-in má za úkol vytvořit nový panel s pracovní plochou, kde bude zobrazena struktura komponent, nastavit vstupní data a poté s těmito vstupními daty zavolat inicializační metodu `displayEntModel()`. Vstupní data obsahují jméno projektu, komponentový model a seznam komponent k zobrazení. Model vstupních dat je následující:

```
public class VisualizationEditorInput implements IEditorInput,
IPersistableElement {
      private ComponentModel componentModel ;
      private List<Component> componentList ;
      private String projectName ;
      // další metody
}
```

Důležitá je také abstraktní metoda `centerOnComponent(Component c)` ve třídě `CommonVisualizerHandler`, která řeší zaměření určité komponent na pracovní ploše nezávisle na vizualizačním pluginu.

6.1.3 Podporované komponentové modely

V současnosti podporuje ComAV tři komponentové modely, z toho dva komerční (*OSGi* a *EJB3*) a dále komponentový model *SOFA 2*, který je detailněji popsán v následující kapitole 3. Analýza a implementace zkvalitnění vizualizace komponentového modelu SOFA 2 je hlavním předmětem této práce, proto ostatní komponentové modely zde nebudou detailně rozebrány.

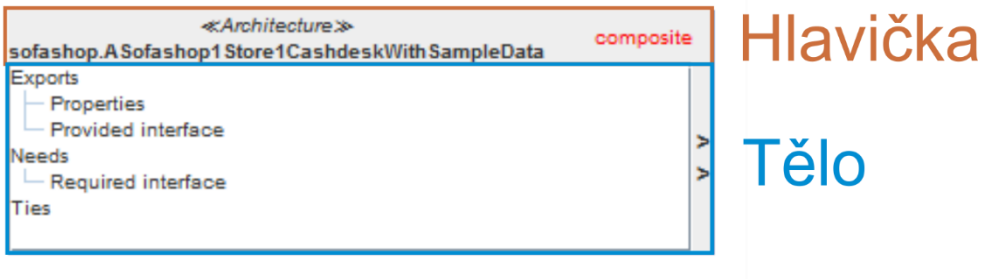
6.2 AIVA

Vizualizační plug-in *AIVA* (*Advanced Interactive Visualization Approach*) pro *ComAV* je založen na ENT meta-modelu, popsaného v kapitole 4. Používá datovou vrstvu ENT meta-modelu, která dokáže velmi detailně popsat jednotlivé komponenty. Jedná se o alternativu k UML, je zde snaha o lepší strukturu informací. Na rozdíl od UML je tento druh vizualizace více *interaktivní* a je možné pracovat na rozdílné úrovni detailů. Není tedy potřeba vytvářet několik druhů diagramů pro odlišné účely. Takový způsob přístupu má ovšem nevýhodu v tom, že jej *nelze přenést na papír* a lze jej provozovat pouze na PC [11].

6.2.1 Vizualizace komponent v AIVA

AIVA zbytečně nezavádí nové značení, ale využívá značek z *UML* a přidává k tomu několik vlastních, které v *UML* nejsou k dispozici. Kvůli přehlednosti pro uživatele je snaha o *redukci* počtu spojení u rozsáhlých diagramů. Hlavním cílem je *přehlednost* a *čitelnost* diagramů.

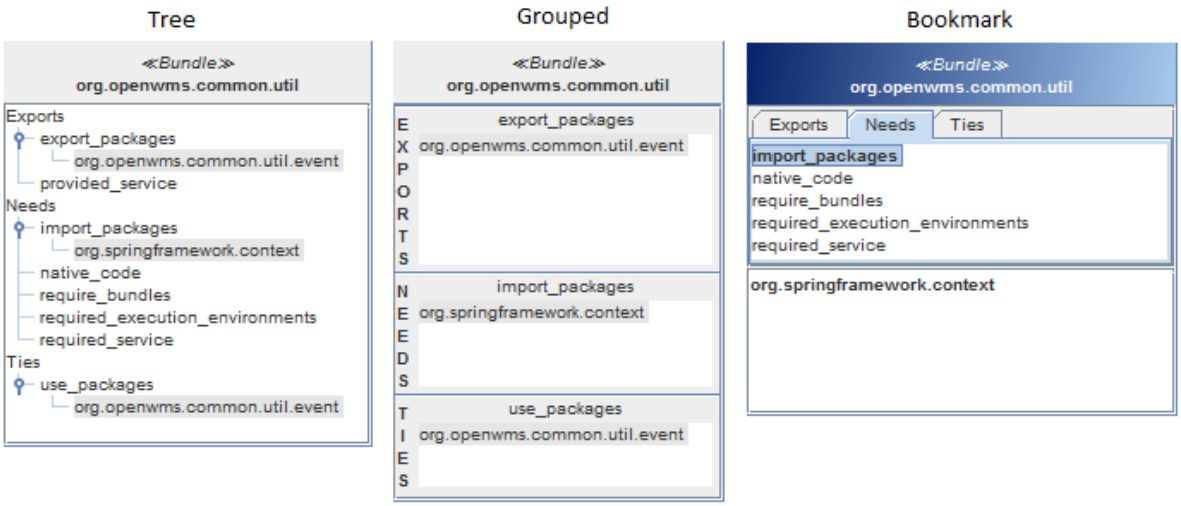
Vizuální podoba komponenty má hlavičku, která má dvě části: typ komponenty ve špičatých závorkách a jméno komponenty. Vzhled hlavičky je inspirován z *UML* a nelze jej změnit. Pod hlavičkou je zobrazeno tělo komponenty, se kterým lze provádět úpravy, co a jak má být zobrazeno (Obrázek 13).



OBRÁZEK 13 – VIZUÁLNÍ PODOBA KOMPONENTY

Standardně jsou informace seskupovány do skupin podle typu (podobně jako v UML). Dále AIVA umožňuje seskupovat podle charakteristik, jak se komponenta

chová vzhledem ke svému okolí. Způsob reprezentace komponent má tři druhy: *tree*, *grouped* a *bookmark*. Každá reprezentace má své výhody i nevýhody (Obrázek 14).



OBRÁZEK 14 – POROVNÁNÍ STEJNÉ KOMPONENTY V RŮZNÝCH REPREZENTACÍCH

Stromová reprezentace (*Tree* – Obrázek 14 vlevo) popisuje komponenty hierarchicky jak název napovídá pomocí stromové struktury, která je snadno čitelná. Reprezentace zobrazuje i prázdné skupiny. Může se ale stát, že jméno skupiny bude kolidovat se jménem prvku.

Naproti tomu *skupinová reprezentace* (*Grouped* – Obrázek 14 uprostřed) má také hierarchické uspořádání, které ale není tak dobře čitelné jako u stromové reprezentace. V levé části jsou zobrazeny hlavní skupiny, prázdné skupiny zde nejsou vůbec zobrazeny. Jako výhodu lze vyzdvihnout, že je zde snadno rozpoznatelný rozdíl mezi prvkem, hlavní skupinou a typem komponenty.

Jak je z názvu patrné, *záložková reprezentace* (*Bookmark* – Obrázek 14 vpravo) je založena na záložkách, které jsou umístěné v horní části a zobrazují se všechny skupiny včetně prázdných. Nezobrazují se všechny informace najednou, ale nejvíce se zde využívá interaktivních vlastností, a proto je vhodná pro aplikace složené ze složitých komponent. Zato na jednoduché komponenty vhodná není, jelikož pokud o sobě komponenta nese jen málo informací, může docházet ke zbytečnému skrytí těchto informací v záložkách, které nejsou primárně zobrazeny.

6.2.2 Pokročilé možnosti vizualizace

V ENT meta-modelu je možné uchovávat i další informace, které se pro přehlednost nezobrazují v diagramu přímo, ale využívají zmiňované interaktivní vlastnosti [11]. Přejetím myší nad různými elementy lze zobrazit další informace, nevýhodou je opět nemožnost tyto informace přenést na papír.

Vizualizace komponent nabízí i další *pokročilé techniky*. Protože detailní popis implementace těchto technik překračuje rozsah této práce, je zde uveden pouze stručný seznam a některým se pak věnuje praktická část.

- Seskupování a filtrace prvků a traitů
- Zobrazení dalších informací kliknutím na prvek (komponentu nebo hranu)
- Postupné rozbalování složených komponent, skládajících se z jedné nebo několika subkomponent
- Obecný pohled, kde je přehledně zobrazena struktura aplikace
- Zvýraznění závislostí
- Rolování, přibližování, atd.

6.2.3 Srovnání AIVA a UML

AIVA a *UML* poskytují odlišný pohled na strukturu komponentově orientovaných aplikací. *Každá* z vizualizací má své *silné* i *slabé stránky*. Při práci s aplikací, která obsahuje velké množství komponent a spojení mezi nimi, je vhodnější použití vizualizace AIVA. Díky interaktivitě je zde možné jednotně pracovat na různých úrovních detailů, zatímco v UML je potřeba vytvářet několik nezávislých grafů.

Na druhou stranu se UML hodí v situacích, kdy je potřeba design aplikace přenést na papír. Navíc je UML velmi známý a rozšířený modelovací nástroj. Pokud je potřeba dynamicky modelovat aplikaci, je také vhodnější použít UML.

6.2.4 Implementace

Jelikož se jedná o plugin pro aplikaci ComAV, jsou zde připraveny rozhraní, proti kterým je nutné napsat implementaci. V pluginu se není potřeba zabývat ostatními

implementačními detaily, které řeší ComAV, je zde pouze funkcionalita ohledně samotné vizualizace.

Plugin AIVA je vyvíjen pomocí knihovny *JGraph* (viz kapitola 5). Tato knihovna je vyvíjena v různých programovacích jazycích, projekt AIVA používá konkrétně knihovnu J*GraphX*, která je určena pro platformu Java. Následující části práce se tedy bude zabývat pouze knihovnou *JGraphX*. Funkce knihovny musely být rozšířeny pro ENT meta-model a tyto přidané funkce nakonec byly odděleny od samotného pluginu AIVA. Tím vznikl nový projekt *JGraphBasic*. *JGraphBasic* tedy v sobě zahrnuje externí knihovnu *JGraphX* společně s *přidanou funkcionalitou*. Plugin AIVA pak využívá pro vizualizaci funkce z JGraphBasic. Důvod pro oddělení JGraphBasic a AIVA bylo to, aby i další vizualizační pluginy měly k dispozici nově přidané funkce. JGraphBasic je tedy *abstraktní plugin*, který je spojen s implementací ENT modelu.

Samotný plugin AIVA je složen z téměř 60 tříd umístěných v 6 balíčcích, jeho podpůrný plugin JGraphBasic obsahuje skoro 50 tříd. Oba projekty tvoří rozsáhlejší implementaci, kde se před každou změnou musí dávat pozor na dopad u ostatních částí pluginu.

7 STAV PŘED ZAHÁJENÍM IMPLEMENTACE

Stav *pluginu AIVA* a *SOFA 2 loaderu* před začátkem implementace je dobrým základem pro předvedení funkčnosti. Jak se píše v kapitole 3, SOFA 2 model má hierarchickou strukturu uspořádání komponent a tím pádem se odlišuje od jiných rozšířených komponentových modelů.

SOFA 2 Loader v této fázi umí načítat data jak ze *zdrojových souborů*, tak z *repozitáře*. Pro načtení veškerých informací se využívá speciálních *ADL* (formát XML) *souborů*, které popisují implementaci programu. *ADL soubory* se dělí na několik různých druhů:

- *Architecture*
- *Assembly*
- *Aspect*
- *Code bundle*
- *Deployment Plan*
- *Frame*
- *Interface Type*
- *Micro Component*
- *Micro Interface Type*

Nejprve dojde k načtení *ADL souboru* s *Assembly*, která obsahuje odkaz na *Architecture*, což zde představuje nejvyšší *entitu* (hlavní komponentu) v analyzované aplikaci. Dále může obsahovat seznam subkomponent, každá subkomponenta má odkaz na vlastní ADL soubor s *Architecture*. Takto jsou rekurzivně načteny a analyzovány informace o komponentách aplikace napsané v SOFA 2 frameworku.

Další fáze spočívá v analýze závislostí u komponent. Závislost komponent vzniká podle rozhraní, které komponenta nabízí a vyžaduje. Pokud tedy jedna komponenta nabízí určité rozhraní, které druhá komponenta naopak vyžaduje, dojde k vytvoření oboustranné vazby, která je uložena do datového modelu obou komponent. Toto je jediný druh závislostí, které SOFA 2 komponenty mají.

7.1 PŮVODNÍ IMPLEMENTACE

7.1.1 JGraphBasic

Původní implementace knihovny JGraphBasic obsahuje celkem 46 tříd rozmístěných v 15 balíčcích:

- ***cz.zcu.kiv.comav.visualizations.jgraphbasic*** – Balíček na nejvyšší úrovni hierarchie obsahuje vstupní bod pro registrování pluginu do aplikace.
- ***cz.zcu.kiv.comav.visualizations.jgraphbasic.connection*** – Zde jsou funkce pro vytvoření a správu spojů (hran).
- ***cz.zcu.kiv.comav.visualizations.jgraphbasic.connection.line.lollipop*** – Obsahuje funkce pro rozšíření klasickému vzhledu hrany z knihovny JGraph, v AIVA je spoj rozšířen o symbol „lollipop“ (Obrázek 19).
- ***cz.zcu.kiv.comav.visualizations.jgraphbasic.ent*** – Místo pro spojení datové vrstvy ENT modelu s funkcemi knihovny JGraph.
- ***cz.zcu.kiv.comav.visualizations.jgraphbasic.overlay*** – V tomto balíčku je implementována funkcionalita pro pestré grafické znázornění komponent. Objekt komponenty se z pohledu knihovny JGraph chová jako vrchol grafu, ale zároveň na sobě nese další grafické komponenty z knihovny Swing, což funkce knihovny JGraph nativně neumožňuje.
- ***cz.zcu.kiv.comav.visualizations.jgraphbasic.overlay.bookmark*** – Zde jsou funkce pro vytvoření komponent zobrazené záložkovou reprezentací.
- ***cz.zcu.kiv.comav.visualizations.jgraphbasic.overlay.grouped*** – Zde jsou funkce pro vytvoření komponent zobrazené skupinovou reprezentací.
- ***cz.zcu.kiv.comav.visualizations.jgraphbasic.overlay.tree*** – Zde jsou funkce pro vytvoření komponent zobrazené stromovou reprezentací.
- ***cz.zcu.kiv.comav.visualizations.jgraphbasic.utils*** – Balíček seskupuje užitečné společné funkce využívané přímo knihovnou JGraphBasic nebo některým dalším vizualizačním pluginem. Jedná se například o správu různých typů textů, odlišné druhy formátování a další funkce.

7.1.2 AIVA

Původní implementace pluginu AIVA se skládá ze 49 tříd, které jsou rozděleny do 14 balíčků. Popis těch nejdůležitějších balíčků je zde:

- ***cz.zcu.kiv.comav.visualizations.aiva*** – Jedná se o nejvyšší balíček v hierarchii balíčků pluginu AIVA. Obsahuje vstupní bod k pluginu, které umožňují jeho zaregistrování do aplikace. Dále obsahuje třídu AivaGraph, která je potomek třídy mxGraph a přídává funkcionalitu pro práci s grafem v AIVA pluginu.
- ***cz.zcu.kiv.comav.visualizations.aiva.hierarchy*** – Balíček má pouze jednu třídu AivaSubGraph, která je potomkem AivaGraph a umožňuje, aby bylo možné vkládat grafy hierarchicky do sebe. Pro vnořené grafy se vytváří instance AivaSubGraph.
- ***cz.zcu.kiv.comav.visualizations.aiva.hierarchy.overlay*** – Funkce tohoto balíčku je rozšířit možnosti zobrazení složených komponent v různých reprezentacích. Složená komponenta obsahuje další funkce pro rozvinutí a zobrazení jejích potomků.
- ***cz.zcu.kiv.comav.visualizations.aiva.highlighting*** – Zde jsou implementovány přidané funkce pro zvýrazňování komponent a jejich spojení.
- ***cz.zcu.kiv.comav.visualizations.aiva.utils*** – Balíček seskupuje užitečné funkce využívané v pluginu AIVA. Jedná se například o posouvání pracovní plochy levým tlačítkem myši nebo zobrazení informačního rámečku po zaměření komponenty myší. Kromě těchto interaktivních vlastností obsahuje funkci pro vytvoření obecného přehledu z celého grafu.
- ***cz.zcu.kiv.comav.visualizations.aiva.views*** – Soustřeďuje funkce sloužící k přidání pravidel, podle kterých se řídí zobrazení komponent. Tyto pravidla se definují v ComAV v okně *Conditional Manager*. Po vytvoření pravidla je potřeba ho nastavit jako aktivní, jelikož AIVA bere v úvahu pouze aktivní pravidla.

7.2 POŽADOVANÉ ZMĚNY

Tato práce vznikla ve spolupráci kolegů z matematicko-fyzikální fakulty Univerzity Karlovy v Praze. Po vyzkoušení původní verze navrhli tato převážně grafická vylepšení:

- Zobrazení všech závislostí mezi komponentami, včetně jejich subkomponent
- Zobrazení všech komponent a subkomponent v seznamu projektů, dodržení hierarchické struktury
- Načtení a zobrazení hodnot u skupiny Properties
- Zobrazení všech traitů

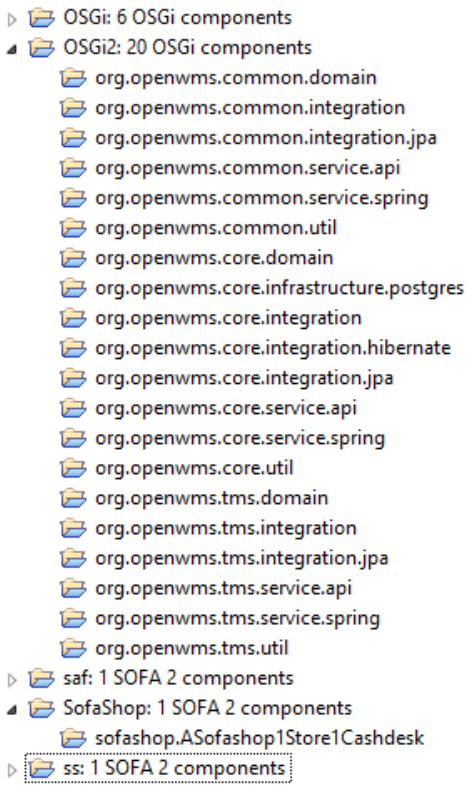
Zobrazení všech závislostí mezi komponentami:

Závislosti jsou v současné implementaci analyzovány pouze na stejné hierarchické úrovni, tudíž *nejsou identifikovány všechny závislosti*. V kapitole 3, která se zabývá SOFA 2 komponentovým modelem, je naznačen způsob komunikace komponent na různých hierarchických úrovních. Vzniká zde nepřímá závislost těchto komponent, která v současné implementaci není zahrnuta a výsledný graf nezobrazuje všechny informace, které lze získat o analyzované aplikaci.

Pro úplnost je důležité tuto funkčnost implementovat, tedy při analýze aplikace získat informace o závislostech komponent na odlišných hierarchických úrovních a dále tyto informace zobrazit ve výsledném grafu.

Zobrazení všech komponent v seznamu projektů:

V *seznamu projektů* (*Project Explorer view*) je možné v každém projektu rozbalit ikony se seznamem komponent, po dvojitém kliknutí na ikonu komponenty je pracovní plocha přesunuta na pozici komponenty a komponenta je zvýrazněna. Tato funkce je plně implementována pouze pro *EJB3* a *OSGi* modely, u *SOFA 2* modelu je kvůli hierarchické struktuře načtena pouze hlavní komponenta (Obrázek 15). Bylo by vhodné tento nedostatek odstranit a zároveň dodržet hierarchickou strukturu při sestavování stromu s komponentami. Dále by bylo dobré změnit ikonu reprezentující komponenty, jelikož nyní je její vizuální podobá stejná jako ikony pro reprezentaci projektu.



OBRÁZEK 15 – SROVNÁNÍ ZOBRAZENÍ KOMPONENT U OSGI A SOFA 2 PROJEKTU

Načtení hodnot parametrů:

V současné implementaci jsou u obou typů loaderu načítány pouze názvy parametrů (trait Properties). Je vhodné k těmto názvům uložit a následně zobrazit i jejich hodnotu. Tyto hodnoty nejsou dostupné přímo v implementaci komponent, ale v souboru s plánem nasazení (deployment plan). Bude tedy potřeba načíst informace obsažené v těchto souborech a následně je přiřadit k příslušným parametrům.

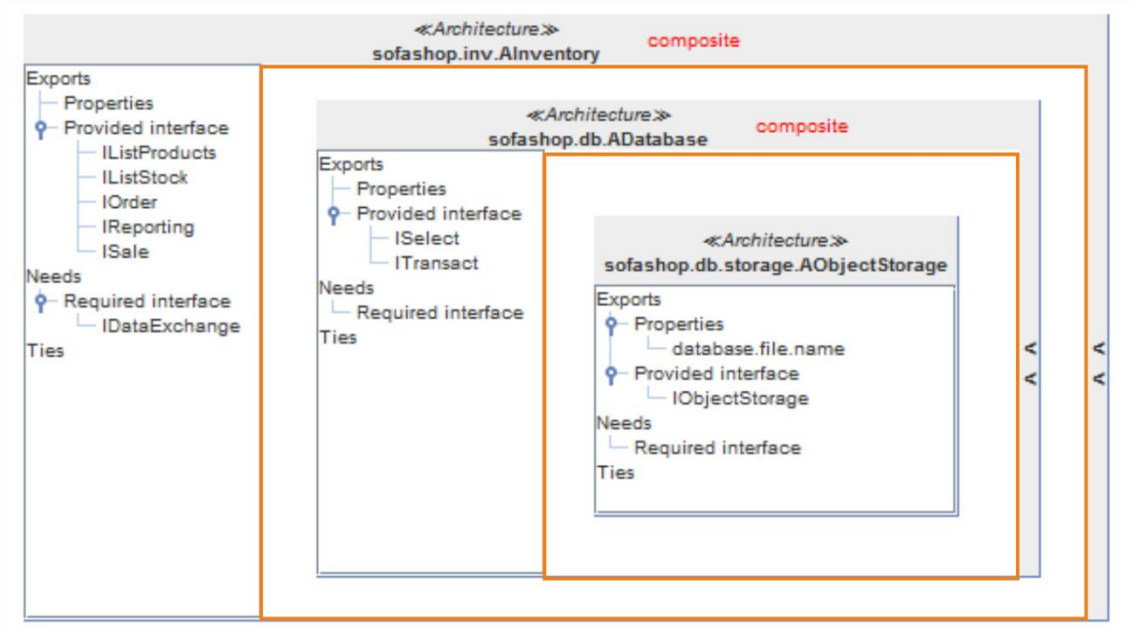
Zobrazení všech traitů:

V současné implementaci jsou zobrazovány pouze traity s neprázdnou hodnotou. Kvůli přehlednosti je požadována úprava loaderů, aby v komponentách byly zobrazeny všechny definované traity včetně prázdných.

7.3 DEFINICE PROBLÉMŮ

Zobrazení hierarchické struktury SOFA modelu je prozatím řešeno pomocí hierarchického vnořování jednotlivých instancí komponent grafu. Čím více má aplikace

složených komponent, tím více instancí grafu je vytvořeno a zobrazeno (Obrázek 16). Jednotlivé instance komponent grafu o sobě vzájemně nevědí a nativní funkce knihovny jGraphX tedy nedokáže propojit prvky z dvou odlišných komponent grafu. Dále tento postup vytváření hierarchického grafu tlačí do vlastního řešení některých problému, které by jinak poskytovala knihovna JGraphX.



OBRÁZEK 16 – HIERARCHICKY ZANOŘENÉ INSTANCE GRAFU (VYZNAČENY ORANŽOVOU BARVOU)

Nemožnost propojit komponenty na odlišných hierarchických úrovních je pouze jeden z problémů, které toto řešení přináší. Zde jsou uvedeny další nevýhody:

- Aplikace layoutu musí být provedena zvlášť na každou instanci komponenty grafu
- Změna parametru v jedné instanci grafu se nepřenese do instance jiného grafu
- Jednotlivé instance komponent grafů o sobě navzájem nevědí, nelze propojit vrcholy ze dvou různých instancí
- Náhled projektu je omezen pouze na jednu instanci grafu, což znemožňuje vytvořit celkový náhled na aplikaci s hierarchickou strukturou
- Změnit velikost komponent lze pouze u primitivních komponent
- Při posunu subkomponenty mimo oblast rodičovské komponenty dochází k částečnému skrytí subkomponenty a vytvoření slideru

- Některé funkce musí být znovu implementovány, i když je nativně nabízí knihovna JGraphX
- Vysoké nároky na výkon

Nová implementace této části je rozsahově poměrně náročná, jelikož se týká velké části pluginu AIVA. Vzhledem k tomu, že stávající řešení přináší velké množství nevýhod a omezení, je tato reimplementace vhodná. Hlavní priorita této práce tedy bude v reimplementaci hierarchické struktury komponent s využitím nativních funkcí knihovny JGraphX. Co se týče vizuální reprezentace komponent, předpokládá se, že zůstane naprosto stejná, tedy budou k dispozici 3 druhy reprezentace, popsané v kapitole 6.2.1.

8 ANALÝZA, NÁVRH ŘEŠENÍ

8.1 HIERARCHICKÉ USPOŘÁDÁNÍ GRAFU S VYUŽITÍM NATIVNÍCH FUNKCÍ JGRAPHX

V kapitole 5 byla představena knihovna JGraphX, která obsahuje řešení pro hierarchickou strukturu vrcholů. Objekt *cell* obsahuje odkaz jak na svého rodiče, tak na seznam potomků. Tak se tvoří výsledná struktura, kterou je možné zobrazit. Problém zde nastává v případě, kdy je třeba určit, jaké pořadí má objekt cell při zobrazení, které objekty budou pod určitým objektem a které nad ním, jinými slovy chybí možnost nastavení Z souřadnice (je možné určit pouze souřadnice v rovině – X a Y).

Tento problém knihovna v základu neřeší, ale spoléhá na to, že programátor bude objekty vytvářet postupně podle hodnoty priority, to znamená ty objekty, které mají být vidět nad ostatními, budou vytvořeny jako poslední. Toto řešení je v současné implementaci nevýhodné, jelikož se první vytváří objekty vrcholů a po té jsou analyzovány závislosti. Právě u závislostí je potřeba zajistit, aby v hierarchické struktuře byla dodržena přehlednost a hrana reprezentující závislost komponent byla viditelná po celé její délce. Je tedy třeba do objektu cell přidat parametr *priority* a při vykreslování brát tuto prioritu na vědomí. To zahrnuje vhodné rozšíření funkcí knihovny JGraphX. S tím souvisí i reakce komponenty při událostech (například posunutí myši nad určitou oblast, kliknutí myši a další), vždy je nutné zajistit, aby na událost reagovala cell s nejvyšší prioritou.

Samotná komponenta na sobě nese, podle zvolené reprezentace komponenty (viz kapitola 6.2 - AIVA), několik dalších komponent z frameworku Swing, jejichž zobrazení a reakce je potřeba vyladit, aby nevznikaly zmatky při ovládání. Dojde ke změně celé struktury grafu, takže celý graf bude vkládán na jedinou komponentu knihovny Swing. Bohužel kvůli odlišné architektuře knihovny Swing a JGraphX nelze zajistit vzájemnou interakci Swing komponent a objektů cell, jedná se o zcela odlišné druhy objektů a ani jedna z knihoven pro to neposkytuje řešení. Musí se tedy graf

ošetřit tak, aby nedocházelo k chybám při vykreslování. Proto dojde k odstranění prvku *JPanel*, na který se původně vkládaly subkomponenty, místo toho budou vkládány přímo na objekt *cell*. Jeho vzhled je samozřejmě nutné upravit, aby co nejblíže odpovídal vzhledu prvku *JPanel*. Tato změna bude provedena v pluginu AIVA, konkrétně jeho části, která je zodpovědná za vizuální reprezentaci komponent.

Jako další problém zde nastává aplikace *layoutů*. V podkapitole 5.3, která se zabývá layouty v knihovně JGraphX je vyjmenováno několik druhů layoutů, pro hierarchickou strukturu se však hodí pouze *hierarchický layout* (ve vertikální i horizontální verzi). Bude nutné důkladně otestovat správnou funkci layoutů u rozsáhlých grafů, především vyřešit případné problémy s překrýváním. Zobrazovací plugin AIVA dává k dispozici i další layouty, které nebude možné aplikovat SOFA projektů.

Změna nastane také ve způsobu určení velikosti komponenty, která obsahuje subkomponenty. Tato komponenta může být v jednom ze dvou stavů:

- **Složena** - její subkomponenty nejsou viditelné, velikost komponenty je podobná jako velikost komponent bez subkomponent. K velikosti komponenty se přistupuje příkazem: `cell.getGeometry()`.
- **Rozložena** – její subkomponenty jsou viditelné a samotná komponenta musí být úměrná svou velikostí počtu jejích subkomponent. K velikosti komponenty se přistupuje příkazem: `cell.getGeometry().getAlternateBounds()`.

8.1.1 Analýza závislostí na odlišných hierarchických úrovních

SOFA 2 Loader nyní analyzuje závislosti pouze na stejné hierarchické úrovni. Aby vůbec došlo k zobrazení ostatních závislostí, musí v implementaci SOFA 2 loaderu změnit funkce pro analyzování závislostí, která bude vzájemně porovnávat komponenty i na odlišných hierarchických úrovních. Jelikož je hierarchie komponent implementována jako strom, jako nejvhodnější algoritmus procházení tohoto stromu se jeví rekurze.

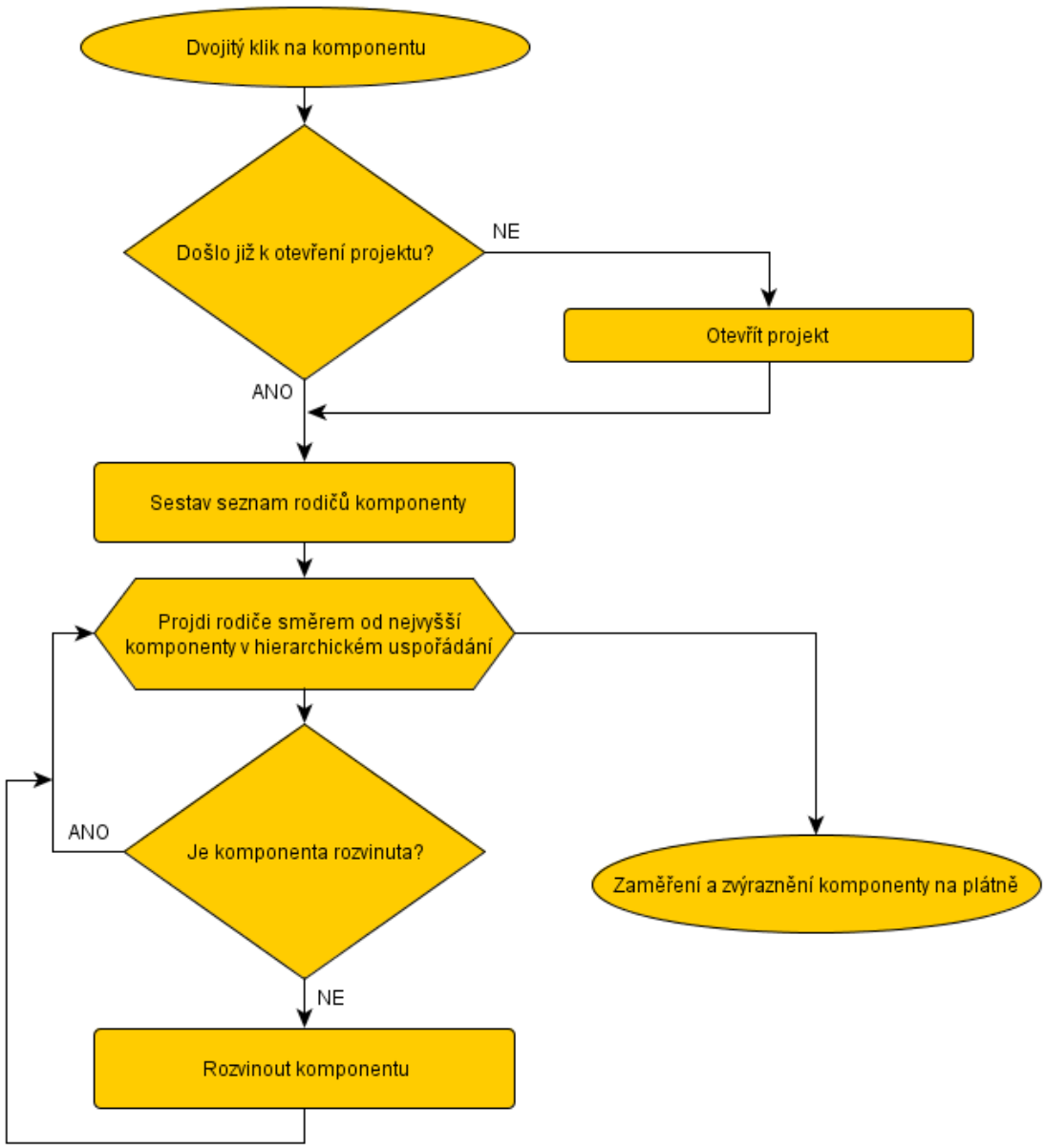
8.2 ZOBRAZENÍ KOMPONENT V SEZNAMU PROJEKTŮ

Funkci vytvoření a zobrazení ikon v seznamu projektů odkazujících na konkrétní komponenty má na starost nástroj ComAV, popsaný v kapitole 6. Na úrovni ENT modelu mají objekty komponent hierarchickou strukturu, kdy každá komponenta obsahuje seznam subkomponent. Po předání hlavní komponenty SOFA aplikace tedy nebude problém získat seznam těchto subkomponent a vytvořit příslušný strom dodržující hierarchii komponent v aplikaci.

Další část tohoto požadavku se týká nástroje AIVA, kde je třeba (pokud k tomu do té doby nedošlo) otevřít příslušný projekt, zajistit nalezení konkrétní komponenty a zaměřit ji na pracovní ploše. Oproti ostatním komponentovým modelům je u SOFA 2 projektů nutné nejdříve ve správném pořadí rozvinout rodičovské komponenty a až po té zjistit pozici komponenty. Popis tohoto algoritmu bude snadnější demonstrovat na vývojovém diagramu (Obrázek 17).

Konečná část algoritmu zahrnuje zaměření a zvýraznění komponenty na plátně. Problém pravděpodobně nastane v získání absolutních souřadnic zaměřované komponenty na plátně. V knihovně JGraphX používá objekt cell u hierarchické struktury grafu relativní souřadnice. Aby bylo možné zaměřit komponentu v rámci celého grafu, bude potřeba získat absolutní souřadnice komponenty.

Funkce pro označení komponenty a její zvýraznění v náhledu jsou již implementovány a nebude do nich nutné zasahovat.



OBRÁZEK 17 – VÝVOJOVÝ DIAGRAM ALGORITMU PO DVOJITÉM KLIKNUTÍ NA KOMPONENTU

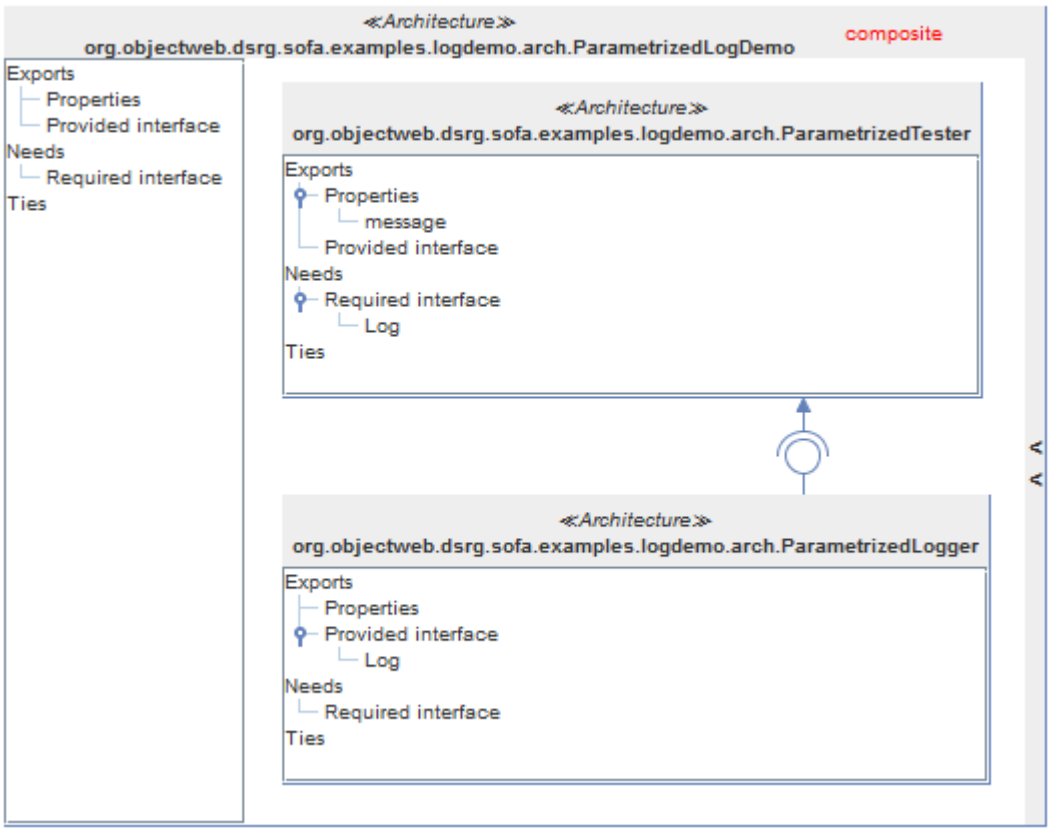
8.3 NAČTENÍ HODNOT PARAMETRŮ

Jak je popsáno v podkapitole 4.5, SOFA 2 loader načítá ADL soubory *Assembly* a *Architecture*. Definice hodnot parametrů je ale uložena na jiném místě, konkrétně v ADL souboru s *plánem nasazení (Deployment Plan)*. Tento soubor obsahuje i další informace, například o uzlu, na kterém bude komponenta provozována. Jednoduchý příklad obsahu ADL souboru s *plánem nasazení* (Obrázek 18):

```
<?xml version="1.0" encoding="UTF-8"?>
<depl-plan
name="org.objectweb.dsrg.sofa.examples.logdemo.deplplan.ParametrizedLo
gDemo" node="nodeA">
  <depl-subc name="logger" node="nodeA"/>
  <depl-subc name="tester" node="nodeA">
    <depl-prop-value name="message">
      Hello world.
    </depl-prop-value>
  </depl-subc>
</depl-plan>
```

OBRÁZEK 18 – UKÁZKA PLÁNU NASAZENÍ

Tato Aplikace obsahuje celkem 3 komponenty, komponenta se zkráceným názvem *ParametrizedLogDemo* je hlavní komponenta a obsahuje 2 subkomponenty – *ParametrizedTester* a *ParametrizedLogger*, které jsou ale v *plánu nasazení* pojmenovány zkráceným jménem (Obrázek 19).



OBRÁZEK 19 – ANALYZOVANÁ APLIKACE ZOBRAZENÁ V AIVA

Subkomponenta *tester* obsahuje parametr *message* a jeho hodnota je v tomto případě *„Hello world.“*. Je samozřejmě možné vytvořit další *plán nasazení*, který bude mít jinou hodnotu parametru *message*. Jedna aplikace tak může mít několik různých souborů

s *plánem nasazení*, který je potřeba zadat při jejím sestavování. To znamená, že pro načtení hodnot parametrů bude potřeba zvolit cílový soubor s *plánem nasazení* ručně.

V datovém modelu jsou jednotlivé parametry uloženy v traitu *Properties*. Pro uložení hodnoty je třeba upravit datový model, konkrétně přidat do definice SOFA 2 ENT modelu tag se jménem *Property value*, na který bude odkazovat trait Properties. Tím je zajištěna změna datového modelu, který je připraven k uložení hodnot parametrů.

8.4 ZOBRAZENÍ VŠECH TRAITŮ

V podkapitole 6.2.1 jsou představeny 3 různé *reprezentace komponent* v AIVA pluginu. Reprezentace *Tree* a *Bookmark* jsou navrženy, aby zobrazovali i *prázdné skupiny*. Je třeba upravit oba typy *SOFA 2 loaderů*, aby byl trait, který neobsahuje žádný další tag, přesto vytvořen a uložen do modelu. Tato změna bude samozřejmě patrná pouze pro zmíněné *Tree* a *Bookmark* reprezentace, kde budou prázdné traity vykresleny, zatímco u reprezentace *Grouped* nedojde k viditelné změně.

9 IMPLEMENTACE NAVRŽENÝCH VYLEPŠENÍ

9.1 ZOBRAZENÍ VŠECH ZÁVISLOSTÍ MEZI KOMPONENTAMI

9.1.1 Implementace nativní JGraph hierarchie

Tato reimplementace pokrývá značnou část pluginů AIVA a JGraphBasic. Došlo ke změně reprezentace objektu složené komponenty, která již není reprezentována jako vlastní graf, ale pouze jako objekt *cell* z knihovny JGraphX. Tím došlo k úplnému odstranění třídy *AivaSubGraph* včetně všech jejích metod a jelikož šlo o jedinou třídu v balíčku *cz.zcu.kiv.comav.visualizations.aiva.hierarchy*, došlo k odstranění i tohoto balíčku. Obecně i u ostatních tříd došlo spíše ke snížení množství kódu, náročnější ovšem bylo ladění a testování nativních funkcí knihovny *JGraphX*. Některá funkčnost byla doimplementována vlastním kódem, ne že by knihovna JGraphX tyto funkce neobsahovala, ale jejich použití se ukázalo jako nevhodné. Tento kód byl doplněn především do abstraktní třídy *BasicCellOverlay* (Obrázek 20), jejíž potomci reprezentují vizuální podobu komponent (Obrázek 21). Třída má tři druhy potomků, každý pro odlišnou vizuální reprezentaci komponent (podkapitola 6.2.1 - Vizualizace komponent v AIVA). Jedná se o tyto funkce:

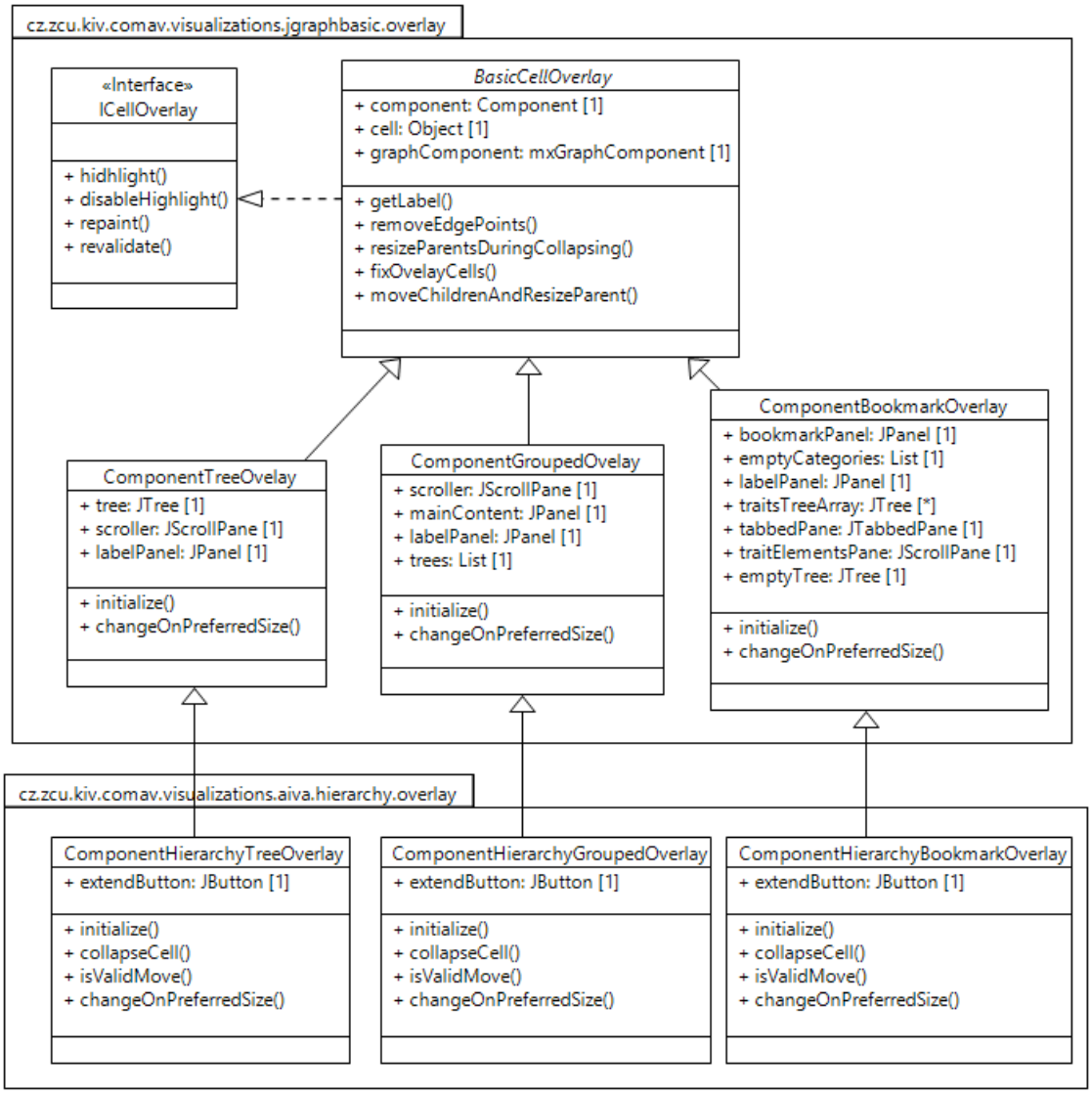
- Posun subkomponent tak, aby nepřekrývali ostatní prvky v rodičovské komponentě, zejména horní panel s hlavičkou a levý panel s informacemi. Dále kontrola při ručním posunu komponent, zda nebude cílová pozice překrývat tyto prvky.

- Vyřešení občasného překrytí některých komponent po aplikaci layoutu – děje se velmi zřídka a nelze predikovat, protože layout při každém aplikování rozmístí komponenty s lehkými změnami.
- Po rozvinutí rodičovské komponenty, která je zároveň subkomponentou (nejedná se o hlavní komponentu) úprava velikostí rodičovských komponent.

```
public abstract class BasicCellOverlay extends JComponent implements
ICellOverlay {
	protected Component component;
	protected Object cell;
	//Další parametry
	...

	/* Odstranění pomocných bodů u hran */
	protected void removeEdgePoints() {
	}
	/* Zvětšení rodičovských komponent */
	protected void resizeParentsDuringCollapsing(mxICell parent,
AEntGraph graph, JComponent leftComponent, JComponent topComponent) {
		//Zdrojový kód
		...
	}
	/* Posunutí komponent, pokud se překrývají */
	protected void fixOvelayCells() {
		//Zdrojový kód
		...
	}
	/* Posunutí komponent potomků, pokud překrývají grafické prvky
rodiče a zvětšení velikosti rodiče */
	protected void moveChildrenAndResizeParent() {
		//Zdrojový kód
		...
	}

	public abstract void cellResized();
	public abstract boolean isValidMove(mxPoint target);
	public abstract void collapseCell();
	//Další metody (většinou se jedná o abstraktní metody)
	...
}
```

OBRÁZEK 20 – KLÍČOVÉ ČÁSTI TŘÍDY BASICCELLOVERLAY

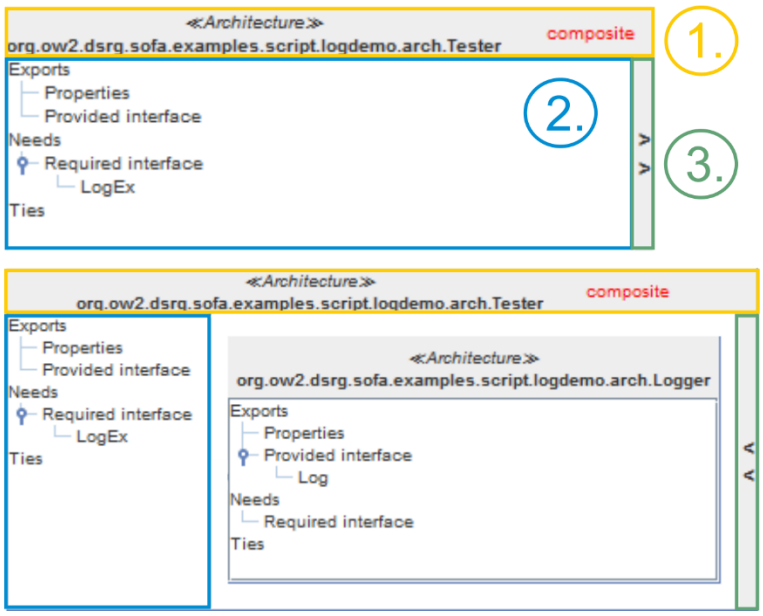
Struktura tříd pro různou vizuální reprezentaci je rozdělena do pluginů AIVA a JGraphBasic. V pluginu JGraphBasic je umístěna abstraktní třída BasicCellOverlay, dále to jsou potomci této třídy: ComponentTreeOverlay, ComponentGroupedOverlay a ComponentBookmarkOverlay, každá zastupuje jednu konkrétní grafickou reprezentaci komponent. Tyto třídy jsou používány v případě primitivních komponent, které neobsahují žádné potomky. V pluginu AIVA jsou pak umístěny další zděděné třídy: ComponentHierarchyTreeOverlay, ComponentHierarchyGroupedOverlay a ComponentHierarchyBookmarkOverlay. Tyto třídy slouží naopak pro složené komponenty. Struktura je na zjednodušeném UML diagramu s klíčovými metodami a parametry na následujícím obrázku (Obrázek 21).



OBRÁZEK 21: UML DIAGRAM STRUKTURY TŘÍDY PRO VIZUÁLNÍ REPREZENTACI KOMPONENT

U vizuální stránky komponent došlo také k několika změnám. U každé reprezentace zůstali pouze 2 hlavní části (v některých případech 3 části - podle toho, zda se jedná o složenou komponentu). Záměrem bylo, aby vzhled komponent zůstal stejný, proto tyto změny nejsou oproti původnímu vzhledu nijak patrné a to u všech dostupných reprezentacích komponenty (Obrázek 22). Každá reprezentace samozřejmě na informačním panelu vlevo nese odlišné grafické prvky. Obecné rozložení lze popsat takto:

1. *Panel s hlavičkou komponenty* - nahoře
2. *Panel s informacemi* - vlevo
3. *Tlačítko pro rozvinutí subkomponent* – vpravo



OBRÁZEK 22 – PRVKY NA KOMPONENTĚ - TREE REPREZENTACE

Objekt komponenty obsahoval dále 2 panely (*JPanel* z knihovny *Swing*). První sloužil k umístění objektu grafu, který tvořil podgraf. Ten byl nahrazen nativní vizuální reprezentací objektu *cell*. Na druhý panel byl umístěn první panel společně s tlačítkem pro rozvinutí komponent. Tím bylo snadněji vyřešeno umístění a velikost jednotlivých prvků na komponentě. Po odstranění těchto dvou panelů musel být přepracován způsob umístění jednotlivých prvků. Obecně u všech typů reprezentací je nutné ručně nastavovat velikost levého panelu s informacemi (2), u něhož platí:

- Pokud je komponenta primitivní nebo případně složená a zároveň svinutá: velikost levého panelu je přes celou komponentu (Obrázek 22 – nahoře)
- Pokud je komponenta složená a rozvinutá – velikost panelu je přizpůsobena podle obsahu (Obrázek 22 – dole)

Používán je BorderLayout z knihovny AWT. Na pozici North je umístěn LabelPanel s hlavičkou komponenty, na pozici West je umístěna komponenta JScrollPane, který na sobě nese komponentu s informačním charakterem podle zvolené reprezentace. Prvku JScrollPane je potřeba ručně nastavovat velikost, změna velikosti této komponenty je vždy při rozvinutí a svinutí složené komponenty.

9.1.2 Zavedení priority v objektu cell

Objekt *cell* byl rozšířen o vlastnost *priority*, potomek objektu cell má název `cellPriority` (Obrázek 23). Jedná se o celé číslo udávající prioritu zobrazení vzhledem k ose Z (analýza problému je v kapitole 8.1). Na ploše je pak priorita dána pořadím při vykreslování, objekty vykreslené později jsou v popředí před objekty, které již byly vykresleny. Dále byla do třídy přidána metoda pro získání seznamu s potomky, namísto metody vracející pouze jednoho potomka - `getChild(int position)`.

```
public class mxCellPriority extends mxCell implements mxICell {
	private int priority = 0;

	//Konstruktor + accesory
	...

	public List<Object> getChildren() {
		return this.children;
	}
}
```

OBRÁZEK 23 – NÁSTIN TŘÍDY MXCELLPRIORITY

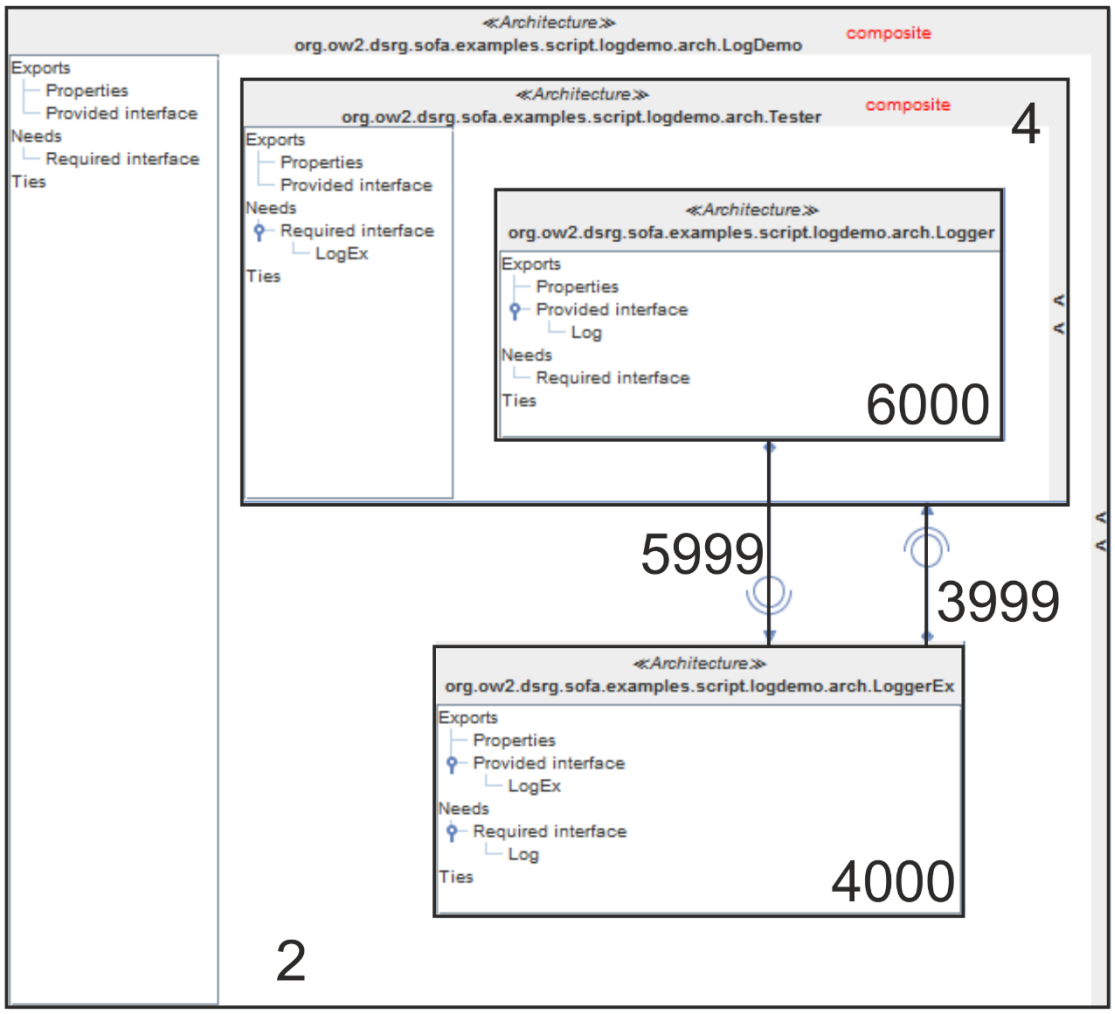
S vykreslením všech vytvořených objektů *cell* do *Swing* komponenty *mxGraphComponent* pomáhá třída mxGraphControl, jedná se o vnořenou třídu ve třídě mxGraphComponent, takže je k dispozici pouze pro tuto třídu. Vlastní vykreslení objektů realizuje metoda `drawCells(mxICanvas canvas, Object cell)`, která je ve třídě mxGraphControl. Před vykreslením je přidán komparátor pro seřazení objektů cellPriority podle priorit, čímž je zajištěno vykreslení objektů ve správném pořadí (Obrázek 24). Po seřazení jsou standardním postupem vykresleny všechny objekty cell.

```
Collections.sort(cells, new Comparator<Object>() {
	@Override
	public int compare(Object arg1, Object arg2) {
		int pr1 = -1, pr2 = -1;
		if (arg1 instanceof mxCellPriority)
			pr1 = ((mxCellPriority) arg1).getPriority();
		if (arg2 instanceof mxCellPriority)
			pr2 = ((mxCellPriority) arg2).getPriority();
		return (pr1>pr2 ? 1 : (pr1==pr2 ? 0 : -1));
	}
});
```

OBRÁZEK 24 – KOMPRARÁTOR SEŘAZUJÍCÍ OBJEKTY MXCELLPRIORITY

Objekty cell jsou dvojího typu, jedná se buď o komponenty, nebo o hrany. Hodnota priority u komponent je vždy sudá a přímo úměrná úrovni zanoření

komponenty v hierarchické struktuře. Dále u komponent, které nejsou složené ze subkomponent, je priorita zvýšena, a to kvůli hranám, které tak přes ně nebudou zobrazeny. Hodnota priority u hran je lichá a závisí na nejvyšší hodnotě zanoření u komponent, které spojuje (Obrázek 25).



OBRÁZEK 25 – PŘÍKLAD PRIORIT U ZOBRAZENÍ OBJEKTŮ CELL

9.1.3 Aplikace layoutu na hierarchickou strukturu

V analýze (kapitola 8.1) je zmínka o dostupných layoutech, kde je popsáno, že u hierarchicky strukturovaných grafů se uvažuje pouze hierarchický layout.

Po aplikaci layoutu je vhodné provést rozvinutí a následné svinutí všech složených komponent, čímž se provede se správné rozmístění a posunutí subkomponent.

Problém nastává při opětovné aplikaci layoutu na stejný graf, kdy se objevují některé zvláštní příznaky, například dochází ke zvětšování mezer (například mezi komponentami). Tento problém naštěstí lze řešit. Nejjednodušší způsob je odstranění celého grafu a jeho opětovné vytvoření, layout se tak aplikuje na konkrétní graf vždy jen jednou. Toto se tedy děje vždy při změně layoutu a také při změně reprezentace komponent. Odstraněním grafu je myšleno odebráním všech jeho vrcholů, příslušné hrany jsou odstraněny automaticky. Klíčové části metody jsou nastíněny zde (Obrázek 26):

```
public mxIGraphLayout changeLayout(GraphLayouts graphLayout, boolean
animate) {
      /* Odstranění všech vrcholů v grafu, druhý parametr určuje
      smazat i příslušné hrany */
      graph.removeCells(allVertexes.toArray(), true);

      //Vytvoření instance layoutu a nastavení jeho parametrů
      ...

      /* Aplikace layoutu na graf */
      layout.execute(graph.getParentCell();

      /* Rozvinutí složených komponent a následné svinutí */
      graph.cellsFolded(groupCells.toArray(), false, false);
      graph.cellsFolded(groupCells.toArray(), true, false);
}
```

OBRÁZEK 26 – NÁSTIN METODY PRO ZMĚNU LAYOUTU

Zmíněné zvětšování mezer dochází, jelikož v hierarchickém layoutu je možné definovat různé druhy mezer pomocí několika parametrů, toho se využívá k tomu, aby byl graf co nejlépe čitelný. Parametry jsou následující:

- *InterHierarchySpacing* – Mezera mezi nepropojenými hierarchiemi
- *InterRankCellSpacing* – Mezera mezi sousedními vrstvami
- *IntraCellSpacing* – Mezera mezi objekty na stejné vrstvě
- *ParallelEdgeSpacing* – Mezera mezi paralelními hranami
- *ParentBorder* – Mezera okolo subkomponent v rodičovské komponentě

Může dojít k situaci, kdy se při aplikaci layoutu nepodaří dodržet všechny nastavené kritéria, v takovém případě algoritmus některé parametry jednoduše vynechá. To je v některých situacích nepříjemné, ale řešení problému by bylo složité a je nad rámec této práce.

Hodnoty těchto parametrů jsou v pluginu AIVA nastaveny podle celkového počtu komponent v grafu. U parametru ParallelEdgeSpacing je záměrně ponechána výchozí hodnota, protože právě u tohoto parametru dochází ve většině případů k vynechání, jelikož by se obtížně dodržovala jeho hodnota. Parametr ParentBorder je nastaven na konstantní hodnotu, zde není třeba hodnotu zvyšovat podle počtu komponent. U ostatních parametrů je závislost logaritmická a postupuje se podle následujícího vzorce:

kompLog = log(počet komponent)
InterHierarchySpacing = kompLog * 50
InterRankCellSpacing = kompLog * 60
IntraCellSpacing = kompLog * 50

ParallelEdgeSpacing = ponechána výchozí hodnota
ParentBorder = 100 (konstatní hodnota)

9.1.4 Kontrola pozice objektů

Byly upraveny některé funkce knihovny *JGraphX*, aby byl zachován způsob práce na pracovní ploše. V předchozím textu je zmínka o kontrole pozice při ručním posunu komponenty. Uživateli nelze povolit posouvat libovolně s komponentami, ale vždy je třeba kontrolovat výslednou pozici, zda nepřekrývá některý z prvků jiné komponenty. Kontrola se provádí pouze při posunu, který vyvolal uživatel, z toho důvodu bylo nutné rozšířit vstupní parametry metody moveCells ve třídě mxGraph, která se volá vždy při změně souřadnic některého objektu v grafu. Volání metody bylo rozšířeno o parametr boolean byUser, který v sobě nese informaci, zda volání metody bylo způsobeno uživatelem či nikoliv.

Kontrola pozice je odlišná v každé reprezentaci, jelikož každá reprezentace má jinak rozloženy Swing komponenty (Obrázek 17). Kontrola se provádí v metodě isValidMove(mxPoint target) u příslušné třídy, která zastupuje konkrétní reprezentaci, na příkladu je metoda pro stromovou reprezentaci (Obrázek 27):

```
public boolean isValidMove(mxPoint target) {
      double treeWidth = tree.getPreferredSize().getWidth();
      double labelHeight = labelPanel.getSize().getHeight();
      if (target.getX() >= treeWidth && target.getY() >= labelHeight
        && target.getX() >= 0 && target.getY() >= 0) {
            return true;
      } else {
            return false;
      }
}
```

OBRÁZEK 27 – METODA PRO KONTROLU PLATNÉ POZICE BĚHEM PŘESUNU KOMPONENTY

V metodě je podmínka, ze které je vidět, že se zároveň kontrolují i záporné souřadnice, které nejsou přípustné.

9.1.5 Oprava načtení závislostí

Poslední fáze činnosti *SOFA 2 loaderu* představuje analyzování závislostí komponent (viz kapitola 7 - Stav před zahájením implementace). Tato funkce byla opravena tak, aby byly definovány vazby i na odlišných vrstvách v hierarchii komponent. Nyní se tedy pro každou komponentu projde *kompletní hierarchický strom* komponent namísto pouze seznamu komponent na stejné vrstvě, takže se provede porovnání se všemi dostupnými komponentami.

Uložení závislosti do modelu je v podobě entity z ENT meta modelu, která se pro každou závislost vytvoří ve dvou různých instancích. V první instanci se nastaví proměnné zastupující provázané komponenty (*local* a *alien*) a dále proměnná s informací, zda se jedná o poskytující (provided) závislost. Tyto proměnné se nastavují vzhledem ke komponentě, ve které bude závislost uložena, v proměnné *local* je tedy komponenta, pro kterou je instance vytvářena. Z toho důvodu má instance pro druhou komponentu oproti první instanci hodnoty proměnných opačné – hodnoty *local* a *alien* jsou přehozené a druh závislosti (provided) má negovanou hodnotu.

```
public abstract class Sofa2ComponentLoader extends ComponentLoader {
	//Final parametry s názvy traitů, tagů, assembly apod.
	...

	/* Metoda pro porovnání každé komponenty se všemi ostatními */
	protected void createBindings(List<Component> componentList,
		ENTMMFactory factory) {
		//Zdrojový kód metody, volání createElementBinding
		...
	}

	/* Metoda pro vytvoření a uložení závislosti */
	private void createElementElementBinding(Component
localComponent, Component aliedComponent, String localTraitName,
Element localElement, Element aliedElement, ENTMMFactory factory) {
		ElementElementBinding localBinding = factory
				.createElementElementBinding();
		localBinding.setLocal(localElement);
		localBinding.setAlien(aliedElement);
		localBinding.setProvided(false);
		localComponent.getBindingsList().add(localBinding);
		//Stejný princip vytvoření závislosti pro alied komponentu
		...
	}

	//Ostatní metody
	...
}
```

OBRÁZEK 28 - ABSTRAKTNÍ TŘÍDA SOFA2COMPONENTLOADER

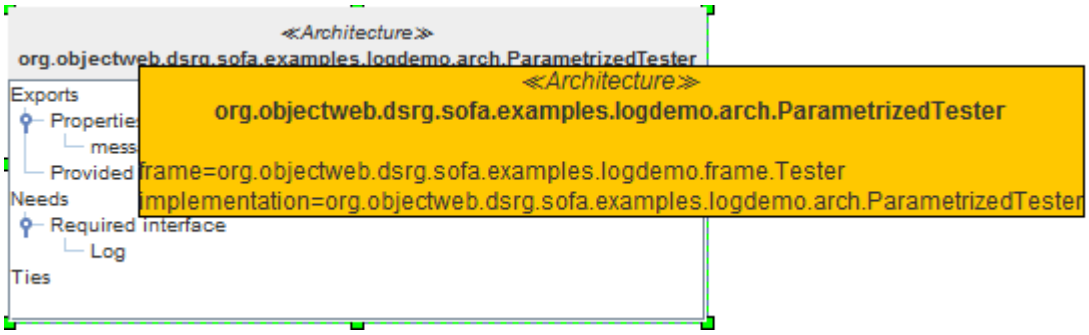
Došlo k vytvoření nové abstraktní třídy `Sofa2ComponentLoader` (Obrázek 28), kam se přesunuly společné metody a parametry z obou typů SOFA 2 loaderů. V konkrétních SOFA 2 loaderech pro načtení dat ze zdrojových souborů nebo repozitáře pak zůstali pouze metody, které se liší. Právě metody pro analýzu závislostí patří do skupiny metod, které jsou pro oba typy loaderů společné.

9.1.6 Další úpravy

Dále byl ošetřen případ, kdy je kurzorem myši zaměřena komponenta a následně zobrazen žlutý rámeček s informacemi o komponentě. Tuto funkci vykonává třída `AivaGraphMouseListener` (OBR). V metodě mouseClicked je ověřeno, zda se na pozici, kde bylo kliknutí myší provedeno, nachází objekt cell. Pokud ano, je o u komponent zjištěna pozice v rámci komponenty. Pokud je kliknutí provedeno v oblasti hlavičky komponenty, je sestaven informační text a se zpožděním je zavolána metoda pro vykreslení žlutého rámečku.

```
public class AivaGraphMouseListener extends MouseAdapter {
	//Definice parametrů a ostatní metody
	...

	/* Metoda spuštěna pokud dojde ke kliknutí na pracovní ploše */
	@Override
	public void mouseClicked(MouseEvent e) {
		final int x = e.getX();
		final int y = e.getY();
		mxCell cell = (mxCell)graphComponent.getCellAt(x, y);
		mxPoint cellClickPos = null;
		if (cell != null && graph.isCellSelectable(cell)) {
			final StringBuilder info = new StringBuilder();
			//Sestavení textu a stanovení cellClickPos
			...

			//Pokud bylo kliknuto v oblasti hlavičky, vykresli
			If (cellClickPos.getY() <= cellPanel.getHeight()) {
				paintInfobox();
			}
		}
	}

	public void paintInfobox() {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				//Vykreslení rámečku
				...
			}
		});
	}
}
```

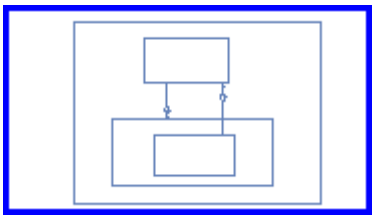
Protože v případě hierarchické struktury jsou nyní subkomponenty přímo na těle rodičovské komponenty, bylo nutné ošetřit zobrazování rámečku v některých nechtěných situacích. Informační rámeček se nyní zobrazuje pouze v případě, že je kurzor myši nad hlavičkou komponenty (Obrázek 29).



OBRÁZEK 29 – ŽLUTÝ INFORMAČNÍ RÁMEČEK

Co se týče obecného náhledu na projekt, řešení tohoto problému bylo vyřešeno automaticky. Jelikož nyní je celý graf pouze v jediné instanci komponenty grafu, není problém vytvořit zmenšený pohled na tuto instanci komponenty grafu. Všechny vrstvy

hierarchie jsou nyní zachyceny v jednom pohledu (Obrázek 30). Zde jsou vidět celkem 4 komponenty na 3 odlišných hierarchických úrovních.



OBRÁZEK 30 – OBECNÝ NÁHLED NA APLIKACI

9.2 ZOBRAZENÍ KOMPONENT V SEZNAMU PROJEKTŮ

Tato úprava má dvě části. První část je úprava nástroje ComAV a spočívá v načtení hierarchické struktury komponent a její převedení do struktury pro nástroj ComAV. V entitě zastupující komponentu v seznamu projektů (třída *cz.zcu.kiv.comav.core.views.utils.ProjectViewComponent*) byl doplněn seznam jejích subkomponent a metody pro přístup k tomuto seznamu (Obrázek 31).

```
public class ProjectViewComponent extends AProjectViewItem {
	private Component component;
	private CommonLoaderPlugin parent;
	private List<ProjectViewComponent> subcomponents;

	//Acesory k parametrům
	...
}
```

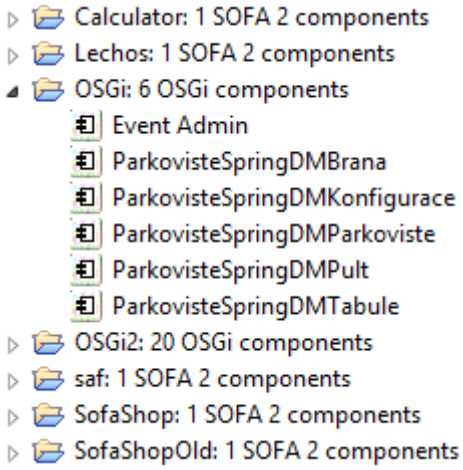
OBRÁZEK 31 – ENTITA REPREZENTUJÍCÍ KOMPONENTU V SEZNAMU PROJEKTŮ

Při sestrojování stromu se tak z komponent rekurzivním algoritmem sestaví hierarchický strom (Obrázek 46). Entita komponenty musí být adaptována do prostředí RCP, což řeší instance třídy `IWorkbenchAdapter` plnící funkci adaptéru, která je umístěna ve třídě `ProjectViewAdapterFactory`. Původní implementace byla připravena tak, že komponenty již neobsahovali v hierarchii další potomky. Metoda `getChildren()` v této třídě byla upravena, jelikož každá komponenta nese odkazy na své subkomponenty (Obrázek 32).

```
private IWorkbenchAdapter componentsAdapter = new IWorkbenchAdapter()
{
	@Override
	public Object[] getChildren(Object o) {
		ProjectViewComponent c = (ProjectViewComponent)o;
		return c.getSubcomponents().toArray();
	}

	//Další metody
	...
}
```

OBRÁZEK 32 – INSTANCE ADAPTUJÍCÍ KOMPONENTY V SEZNAMU PROJEKTŮ

Ikona zastupující komponentu byla změněna, její vzhled je nyní inspirován značkou komponenty z UML a na první pohled je tedy patrné, že se jedná o komponentu (Obrázek 33).



OBRÁZEK 33 – IKONA KOMPONENTY A PROJEKTU

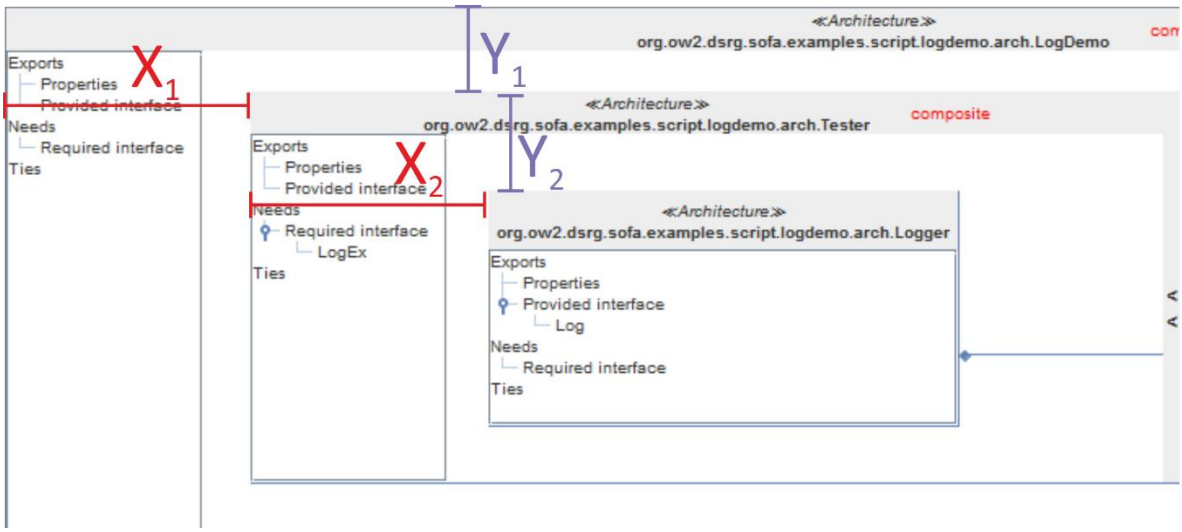
Druhá část implementace je složitější, zahrnuje funkci pro zaměření konkrétní komponenty, která je vykonána po dvojitém kliknutí myší. Tato metoda je v pluginu AIVA ve třídě `AivaGraph` (Obrázek 34). Zdrojový kód této třídy je obsáhlý, třída obsahuje celkem 1277 řádků kódu, proto je zde uvedena pouze část, zabývající se zaměřením komponenty, tento algoritmus je nastíněn v analýze problému (Obrázek 17).

```
public class AivaGraph extends AEntGraph {
	/* Pracuje s objekty na úrovni ENT meta modelu */
	public CommonGraphModel graphModel;

	//Definice parametrů a ostatní metody
	...

	public void centerOnComponent(Component component) {
		for(ComponentNode node : graphModel.getNodes()) {
			if (node.getComponent().equals(component)) {
				//Zaměření komponenty
				...
			}
		}
	}
}
```

OBRÁZEK 34 - NÁSTIN ZAMĚŘENÍ KOMPONENTY VE TŘÍDĚ AIVAGRAPH

Oproti klasické struktuře, kdy jsou všechny komponenty na stejné úrovni, je zde nutné nejprve rozvinout rodiče dané komponenty (pokud k tomu již nedošlo) a to ve stejném pořadí, jako by docházelo v případě ručního rozvinutí, to znamená procházet strom ve směru od shora. V opačném případě totiž dojde k deformaci grafu. Pořadí je zajištěno tak, že se postupně od komponenty projdou všechny rodičovské komponenty až k hlavní komponentě, sestavený seznam se po té projde v opačném pořadí.

Po rozvinutí všech rodičovských komponent jde na řadu výpočet souřadnic komponenty. Při zaměřování místa v grafu, kde komponenta leží, již není možné spoléhat na souřadnice komponenty, protože jsou pouze relativní v rámci rodičovské komponenty. K získání absolutních souřadnic je třeba opět rekurzivně projít hierarchickým stromem až k nejvyšší komponentě a provést součet všech složek souřadnic rodičů uvedené komponenty (Obrázek 35). Až tento součet představuje absolutní souřadnice komponenty a je pak možné přenést pohled na výsledné souřadnice (Obrázek 36).



OBRÁZEK 35 – ZÍSKÁNÍ ABSOLUTNÍCH SOUŘADNIC U SLOŽENÝCH KOMPONENT

```
mxGeometry geometry = new mxGeometry();
mxICell parent = cell.getParent();
while (parent.getParent() != null) {
      //Přidání souřadnic do celkového součtu v geometry
      ...
      parent = parent.getParent();
}
//Přenesení na výsledné souřadnice
scrollRectToVisible(geometry);
```

OBRÁZEK 36 – ZDROJOVÝ KÓD PRO SOUČET SOUŘADNIC

9.3 ZOBRAZENÍ HODNOT PARAMETRŮ

Do obou typů *SOFA 2 loaderů* byla přidána funkce pro načtení dostupných plánů pro *plán nasazení* (*deployment plan*). Do ENT modelu pro SOFA 2 aplikace byl přidán tag *Property value* a do traitu *Properties* odkaz na tento tag (Obrázek 37). Získání seznamu dostupných plánů nasazení je odlišné v obou typech loaderů.

<!-- definice tagů -->

```
...
<tagSet name="Property value"/> <!-- Tag je definován jako 6. -->

<traitSet name="Properties" tagSet="//@tagSet.6" note="Public
properties of implementation">
  <traitClassifier kind="data" construct="instance"
  presence="permanent">
    <role>provided</role>
    <lifecycle>development</lifecycle>
    <lifecycle>assembly</lifecycle>
    <lifecycle>deployment</lifecycle>
    <lifecycle>setup</lifecycle>
    <lifecycle>runtime</lifecycle>
  </traitClassifier>
</traitSet>
...
```

OBRÁZEK 37: PŘIDANÉ PRVKY V DEFINICI ENT MODELU SOFA 2

U loaderu, který získává informace z *repozitáře*, se využívá pomocných tříd z *knihovny SOFA 2* (Obrázek 38).

```
RepositoryAgent agent = new RepositoryAgent(repositoryLocation);
RepositoryQuery query = agent.getQuery();
RepositoryData data = query.getRepositoryDataObject();
if (data != null) {
      EList<DeploymentPlan> plans = data.getDeploymentPlan();
      for (DeploymentPlan plan : plans) {
            //Načtení dostupných plánů nasazení
            ...
      }
}
```

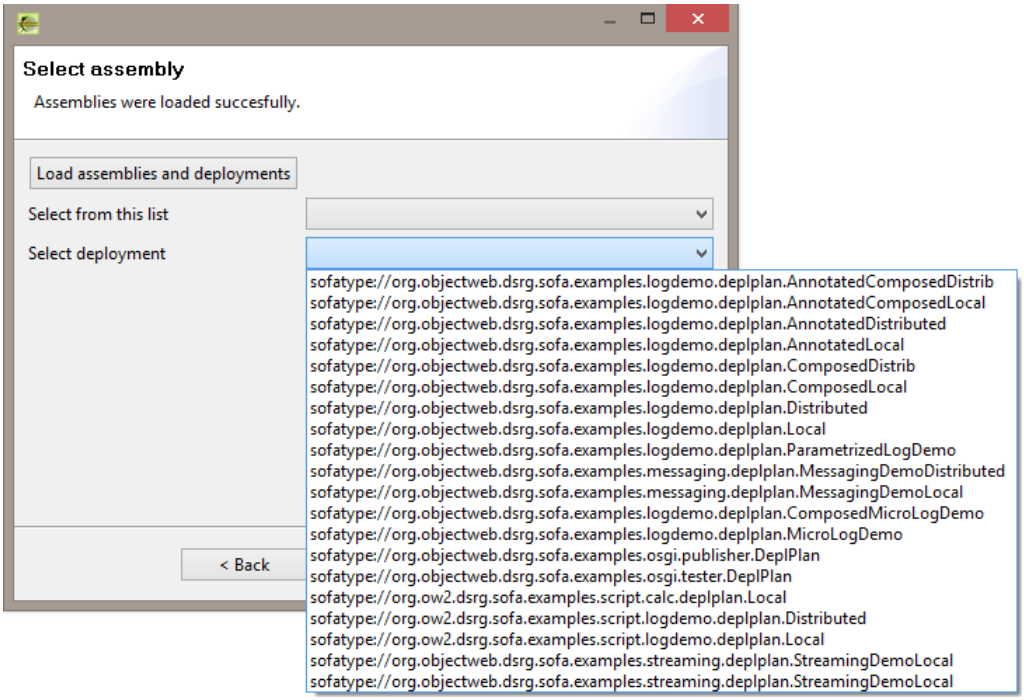
OBRÁZEK 38: NAČTENÍ DOSTUPNÝCH PLÁNŮ NASAZENÍ Z REPOZITÁŘE

Naproti tomu loader, který čte data ze zdrojových souborů, se musí ručně projít všechny dostupné *ADL soubory* a vyfiltrovat ty, které nesou informace o plánech nasazení (Obrázek 39).

```
public class Sofa2LocalComponentLoader extends Sofa2ComponentLoader {
      /* Seznam všech ADL souborů */
      private Hashtable<URI, File> adls;
      /* Seznam dostupných assemblies */
      private ArrayList<URI> assemblies;
      /* Seznam dostupných plánů nasazení */
      private ArrayList<URI> deployments;

      /* Načtení všech ADL souborů,
      roztřídění souborů s assembly a deployment plan */
      private void sniffForAdls() {
            ...
      }
      ...
}
```

OBRÁZEK 39: NAČTENÍ DOSTUPNÝCH PLÁNŮ NASAZENÍ ZE ZDROJOVÝCH SOUBORŮ

Výsledkem u obou typů loaderů je, že spolu s dostupnými *Assemblies* je uživateli dále nabídnut seznam s *plány nasazení* (Obrázek 40). Jelikož *plán nasazení* není nezbytně nutný pro načtení struktury a informací o komponentách, není ani povinné ho zvolit.



OBRÁZEK 40 – SEZNAM S DOSTUPNÝMI PLÁNY NASAZENÍ

Hodnoty parametrů jsou přiřazovány podle shody jména. Z *ADL* souboru jsou načteny všechny hodnoty parametrů a uloženy do struktury *slovníku*. Při načítání informací o komponentách je v případě nalezení parametru vyhledána a uložena její hodnota.

9.4 ZOBRAZENÍ VŠECH TRAITŮ

Úprava zahrnuje nalezení místa, kde dochází k uložení *traitů* do modelu a přepsání této části tak, aby došlo k uložení i v případě *prázdného traitu*. Struktura načítaných souborů je v případě zdrojových souborů a repozitáře odlišná, tudíž i průběh načítání analyzované aplikace je odlišný, a proto je třeba místo ke změně vyhledat v obou typech loaderů a změnu zde provést. Tato úprava není samozřejmě aplikována na stávající projekty, ale projekt musí znovu projít celým procesem počínaje načtením a analýzou.

10 OVĚŘENÍ FUNKČNOSTI

Všechna implementovaná funkčnost byla ověřena v praxi na dvou testovacích aplikacích:

- *LogDemo* – jednoduchá aplikace, načtena ze zdrojových souborů
- *SofaShop* – složitější aplikace, načtena z repozitáře

Testována byla funkčnost jak pro aplikaci načtenou ze zdrojových souborů, tak pro aplikaci umístěnou v repozitáři. Důležitý rys pro testované aplikace bylo dostatečně velké hierarchické zanoření primitivních komponent.

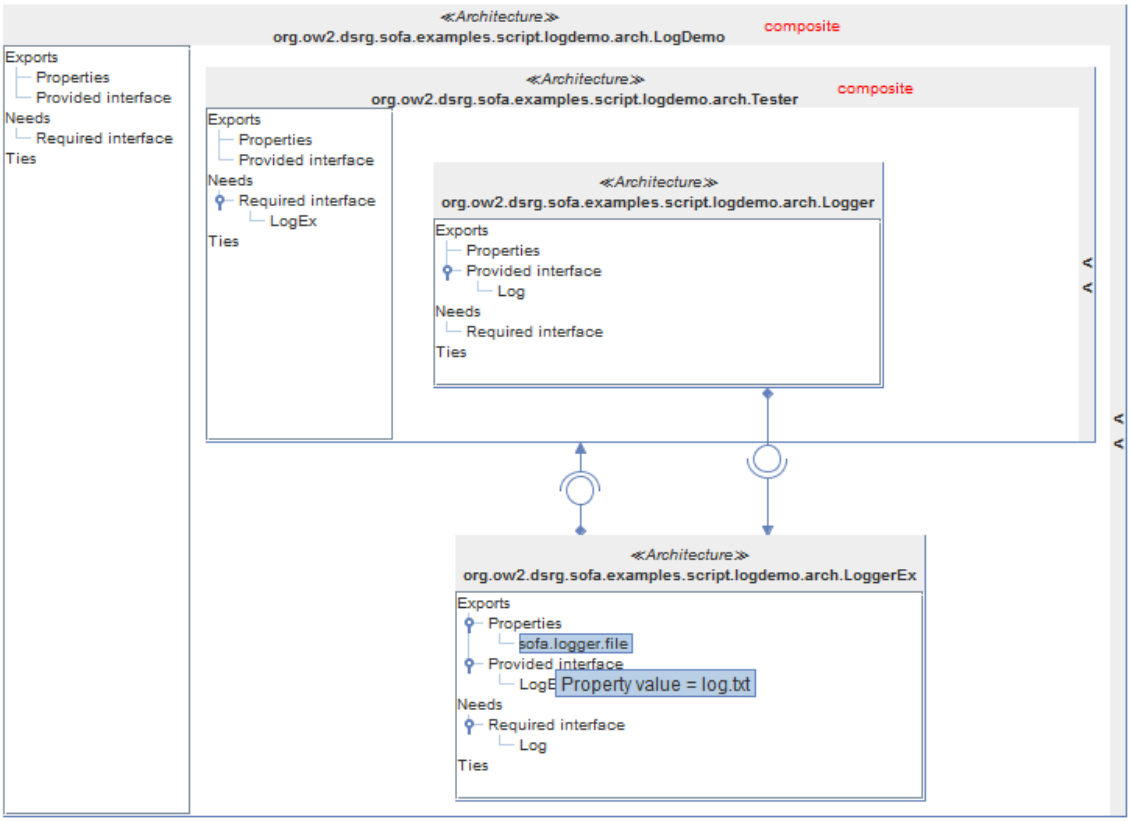
10.1 OVĚŘENÍ FUNKCÍ NA PRIMITIVNÍ APLIKACI - LOGDEMO

Pro otestování základní funkčnosti byla sestavena velmi jednoduchá aplikace LogDemo, která ale obsahuje tyto vlastnosti:

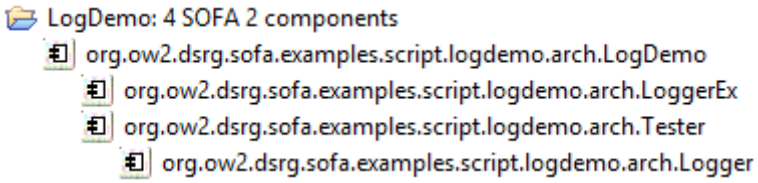
- Hierarchické zanoření úrovně 3
- Závislost komponent na stejné i odlišné hierarchické úrovni
- Obsahuje minimálně 4 komponenty
- Alespoň jeden neprázdný trait Property

Díky těmto vlastnostem mohlo dojít k otestování všech vylepšení, které byly implementovány:

- *Reimplementace hierarchické struktury* (Obrázek 41): Na obrázku je patrné, že jsou zobrazeny všechny závislosti, včetně komponent na odlišných hierarchických úrovních.
- *Zobrazení komponent v seznamu projektů* (Obrázek 42): Struktura komponent je správně načtena.
- *Zobrazení hodnot parametrů* (Obrázek 41): Hodnota parametru je k dispozici, byla tedy správně načtena a uložena.
- *Zobrazení všech traitů* (Obrázek 41): Jsou zobrazeny všechny traity včetně prázdných.



OBRÁZEK 41 – NÁHLED NA TESTOVACÍ APLIKACI LOGDEMO



OBRÁZEK 42 – STRUKTURA KOMPONENT TESTOVACÍ APLIKACE LOGDEMO V SEZNAMU PROJEKTŮ

Na této jednoduché aplikaci byla při vývoji testovaná implementovaná funkčnost. Aplikace byla načítána ze zdrojových souborů, byly zde proto ověřeny všechny úpravy v SOFA 2 loaderu ze zdrojových souborů.

Správná funkce implementovaných vylepšení zde byla úspěšně ověřena a aplikace se chová podle očekávání. Testy porovnávající náročnost zde kvůli jednoduchosti aplikace nebyly provedeny.

10.2 CHOVÁNÍ SLOŽITĚJŠÍ APLIKACE - SOFASHOP

Druhé testovací aplikace byla umístěna v repozitáři a jednalo se o rozsáhlejší aplikaci s názvem SofaShop, která obsahuje celkem 22 komponent. Celkový pohled na aplikaci není možné kvůli velikosti přenést na papír, v následující kapitole je k dispozici náhled na její hierarchickou strukturu komponent. Tato aplikace byla vytvořena autory SOFA 2 pro demonstraci možností frameworku a je k dispozici ke stažení na veřejném SVN uložišti.

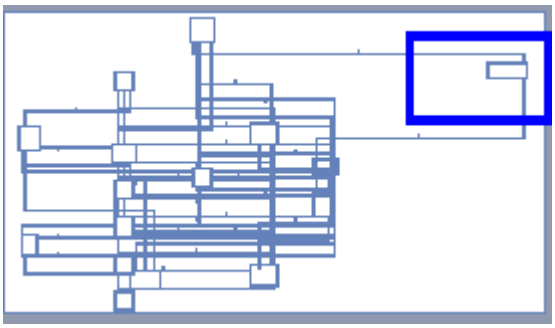
10.2.1 Provedené testy

Aplikace neobsahuje žádné nové vlastnosti, ale díky své složitosti umožňuje důkladně otestovat zobrazování a manipulaci s objekty. Bylo zde provedeno několik druhů testů:

- *Počáteční rozmístění komponent* – v některých případech docházelo k překryvu komponent, což bylo dodatečně opraveno. Dále bylo v některých případech odhaleno umístění složených komponent na zbytečně vysokou hodnotu souřadnic.
- *Rozvinutí složených komponent, reakce okolních komponent* – v pořádku.
- *Přesouvání a změna velikosti komponent* – v pořádku.
- *Zákaz přesouvání potomků mimo rodičovskou komponentu* – v pořádku.
- *Použití SOFA 2 loaderu pro repozitář* – v pořádku.

10.2.2 Nalezené chyby

Byly nalezeny chyby u testu počátečního rozmístění komponent. Nepodařila se opravit chyba, kdy dojde k chybnému rozmístění komponent a některá ze složených komponent dostane zcela zbytečně příliš velké hodnoty souřadnic (náhled na rozmístění komponent – Obrázek 43). Tento jev se děje náhodně a nelze ho předpovídat.



OBRÁZEK 43: NÁHLED NA CHYBNÉ UMÍSTĚNÍ SLOŽENÉ KOMPONENTY

Chyba je v algoritmu hierarchického layoutu, který má na starost rozmístění objektů. Samotná implementace algoritmu je velmi složitá, a proto nebyl dostatek času k nalezení řešení. Nejde o chybu, která by zásadním způsobem ovlivnila práci s grafem, navíc lze předpokládat, že v některé nové verzi knihovny JGraphX bude tento nedostatek opraven.

10.2.3 Paměťové nároky aplikace

Jelikož se jedná o komplexní aplikaci, je tato aplikace vhodná pro srovnání paměťové náročnosti. Zde vzniká problém se způsobem, jak paměťovou náročnost měřit. Byl zvolen způsob dvojího typu měření. První typ měření probíhal v klidu, kdy došlo pouze k otevření projektu a následně stanovení velikosti dat v paměti. V druhém typu měření bylo po dobu 1 minuty manipulováno s objekty (přesouvání, změna velikosti, rozvinutí složených komponent) a přitom byla sledována velikost dat uložených v paměti. Bylo dosaženo následujících výsledků (Tabulka 1):

	V klidu	**Po manipulaci**
Původní implementace	85 MB	až 405 MB
Nová implementace	97 MB	až 517 MB

TABULKA 1 - SROVNÁNÍ PAMĚŤOVÝCH NÁROKŮ

Je celkem zřetelné, že nová implementace je paměťově náročnější. To neodpovídá původnímu předpokladu, kdy bylo počítáno spíše s tím, že reimplementace hierarchické struktury grafů přinese snížení paměťových nároků.

Důvod, proč ke zvýšení došlo, je v knihovně JGraphX. Pokud by byla známá detailní implementace a struktura knihovny, bylo by možné přesněji stanovit příčiny.

Během manipulace s objektem cell není ovlivňován pouze tento jeden objekt, ale je třeba kontrolovat pozice i všech okolních objektů a po každém dokončení manipulace případně provést restruktulizaci grafu. Existují různé typy kontrolních algoritmů, které jsou spouštěny při manipulaci s objekty. V případě předešlé implementace, kdy do sebe byly vnořovány jednotlivé instance grafů, neměly objekty cell takové množství sousedních objektů, protože byly rozmístěny ve více instancích grafů. Tudíž během manipulace s objekty cell bylo provedení kontrolních algoritmů rychlejší. To samé platí už při otevření projektu a rozmístění objektů, čím méně objektů je v instanci grafu, tím méně náročný algoritmus je. Lze tedy předpokládat, že závislost paměťové náročnosti algoritmu a počtu objektů v grafu (složitost algoritmu) není lineární, ale vyšší.

Jelikož došlo k zobrazování závislostí i na odlišných hierarchických úrovních, došlo také ke zvýšení počtu objektů cell. To je jistě další důvod, proč je nová implementace paměťově náročnější než předešlá.

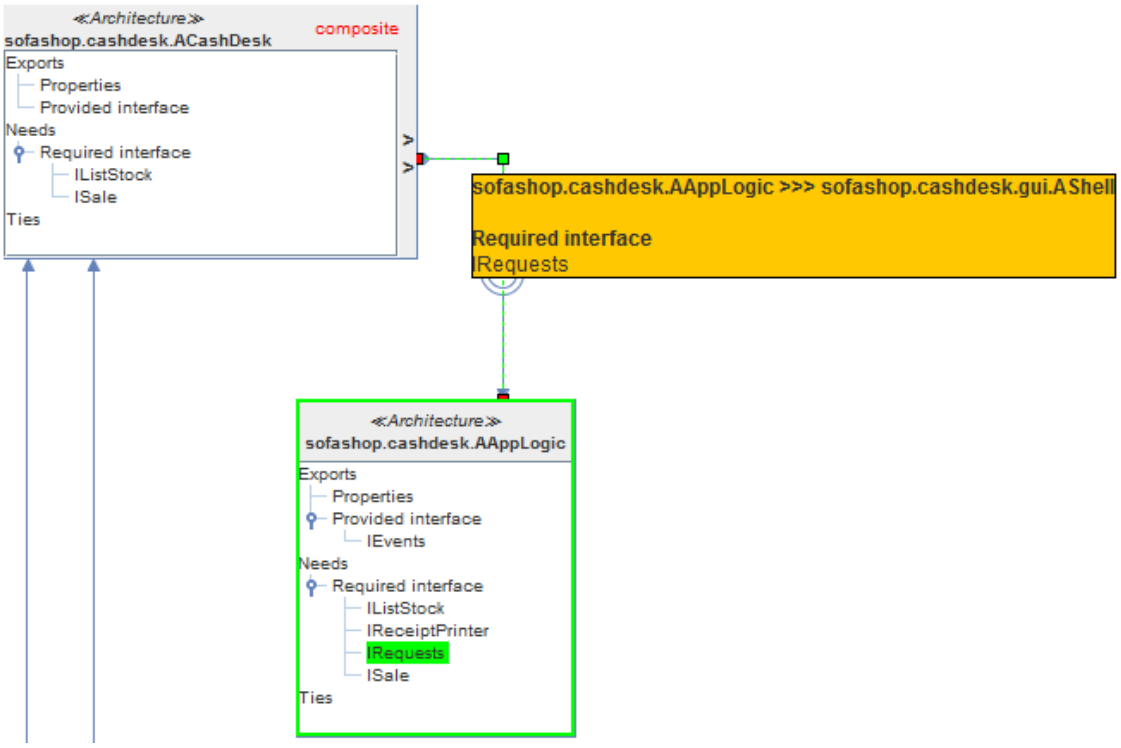
Zvýšení paměťové náročnosti není nijak zvlášť drastické. Pokud se porovnají hodnoty u rozsáhlé aplikace při intenzivní manipulaci s grafem, je navýšení náročnosti přibližně 30%. To je vzhledem k tomu, že jsou zobrazeny všechny závislosti a dalším výhodám této implementace, přípustná hodnota.

11 DEMONSTRACE VÝHOD

11.1 REIMPLEMENTACE HIERARCHICKÉ STRUKTURY GRAFŮ

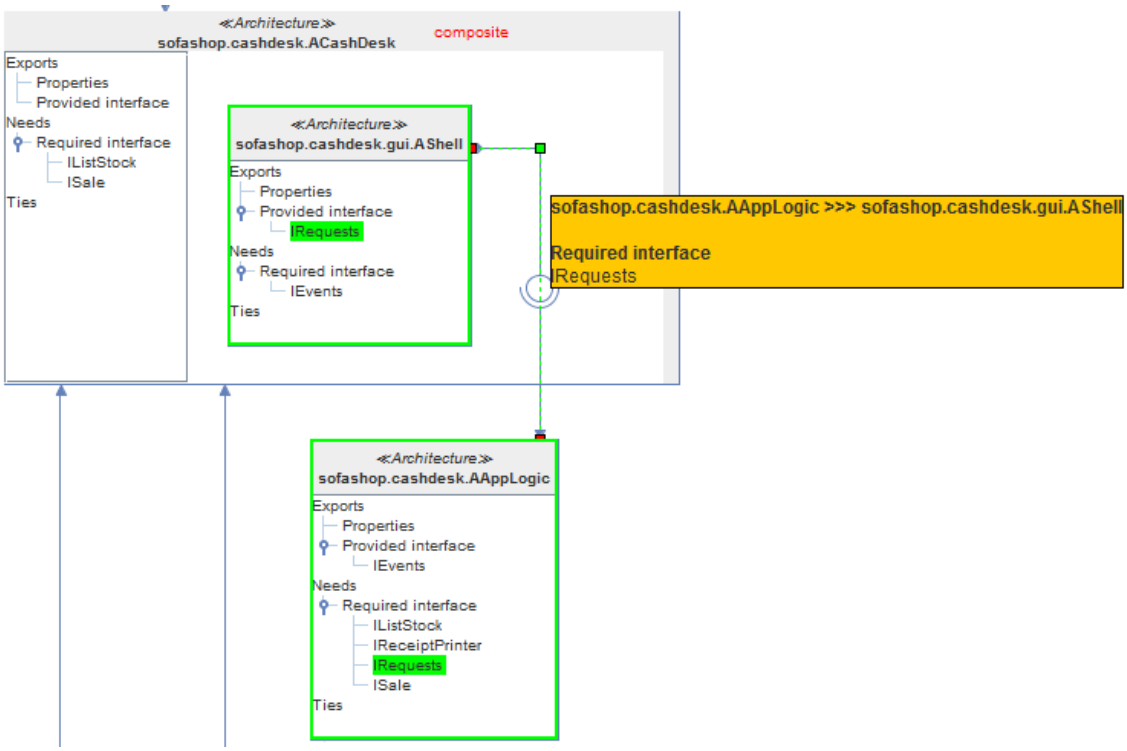
Hierarchická struktura grafů byla v minulosti z časových důvodů implementována ne zcela vhodným způsobem. Nyní se zde využívá nativních funkcí knihovny JGraphX. To na první pohled přináší odstranění problému s vizualizací závislostí na odlišných hierarchických úrovních. Není to ovšem jediný přínos, protože navíc je do budoucna připravena stabilní implementace, na které lze lépe stavět. Výhody této reimplementace lze logicky vyvodit z problémů původní implementace, které jsou vyjmenovány v kapitole 7.3 a které jsou nyní odstraněny.

Závislost komponent na odlišných hierarchických úrovních je zobrazena i v případě, kdy rodičovská komponenta je svinuta (Obrázek 44). Za těchto okolností nemusí být přímo viditelné komponenty, kterých se závislost týká, ale závislost je vedena do rodičovských komponent.



OBRÁZEK 44 - ZOBRAZENÍ ZÁVISLOSTI U SVINUTÉ RODIČOVSKÉ KOMPONENTY

Po rozvinutí rodičovské komponenty se závislost přesune na příslušnou komponentu (Obrázek 45).

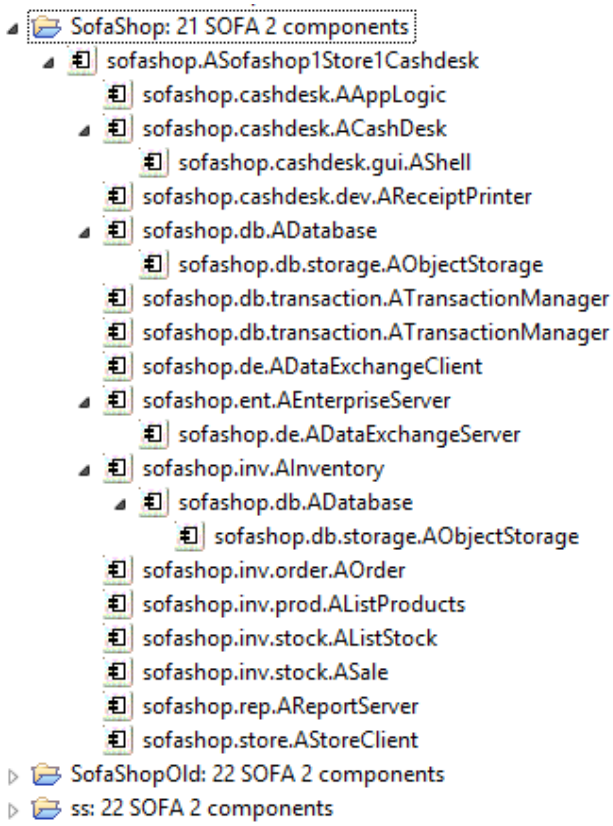


OBRÁZEK 45 - ZOBRAZENÍ ZÁVISLOSTI U KOMPONENT NA ODLIŠNÝCH HIERARCHICKÝCH ÚROVNÍCH

11.2 ZOBRAZENÍ KOMPONENT V SEZNAMU PROJEKTŮ

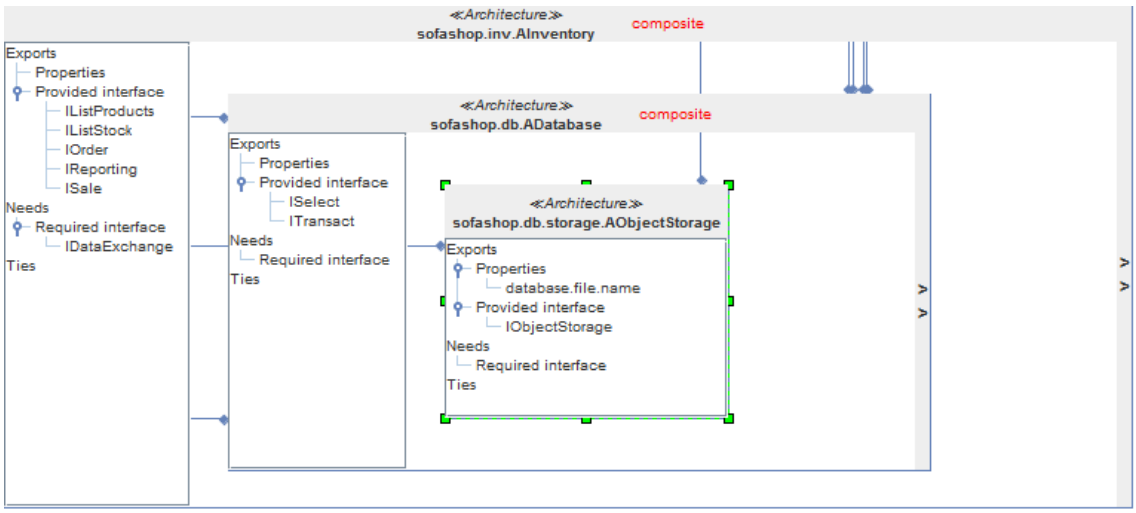
Zde je vylepšeno zobrazení komponent v seznamu projektů, kde je u SOFA 2 aplikací nyní k dispozici kompletní hierarchický strom komponent (Obrázek 46). Jak by tento strom vypadal před reimplementací je k dispozici v kapitole 7, která se zabývá zhodnocením předchozího stavu (Obrázek 15). V současné implementaci jsou komponenty do sebe rekurzivně vnořeny, takže není stanoven limit hierarchického zanoření. Ikona reprezentující komponentu byla změněna, takže je možné na první pohled odlišit komponentu od projektu. Podoba komponenty byla inspirována značkou z UML.

Dále je zde opraven údaj o celkovém počtu komponent, který nyní bere v úvahu všechny komponenty včetně zanořených.



OBRÁZEK 46 – HIERARCHIE KOMPONENT V SEZNAMU S PROJEKTY

Po dvojitém kliknutí na ikonu komponenty je projekt otevřen, rozvinuty všechny rodičovské komponenty a nakonec je přesunuta pozice na pracovní ploše na místo, kde je komponenta umístěna. Tím je především usnadněno hledání komponent, které jsou hlouběji zanořeny. Grafická reprezentace komponenty je zvýrazněna, takže je na první pohled patrné o jakou komponentu se jedná (Obrázek 47).

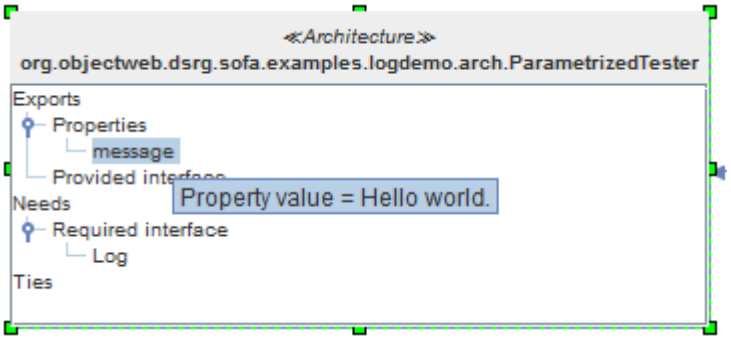


OBRÁZEK 47 – ZVÝRAZNĚNÍ KOMPONENTY

11.3 ZOBRAZENÍ HODNOT PARAMETRŮ

Toto zlepšení přináší možnost načtení konkrétního plánu nasazení, kde jsou uloženy hodnoty parametrů. Načtením hodnot parametrů je odstraněna nutnost prohlížení příslušného xml soubor s plánem nasazení a v něm hledání hodnoty u konkrétního parametru. Vybrání plánu nasazení ovšem není povinné, takže lze případně načítat i aplikace, které zatím nemají vytvořen plán nasazení. V případě, že rozvírací prvek s *plány nasazení* zůstane prázdný, nedojde pouze k načtení hodnot *parametrů*.

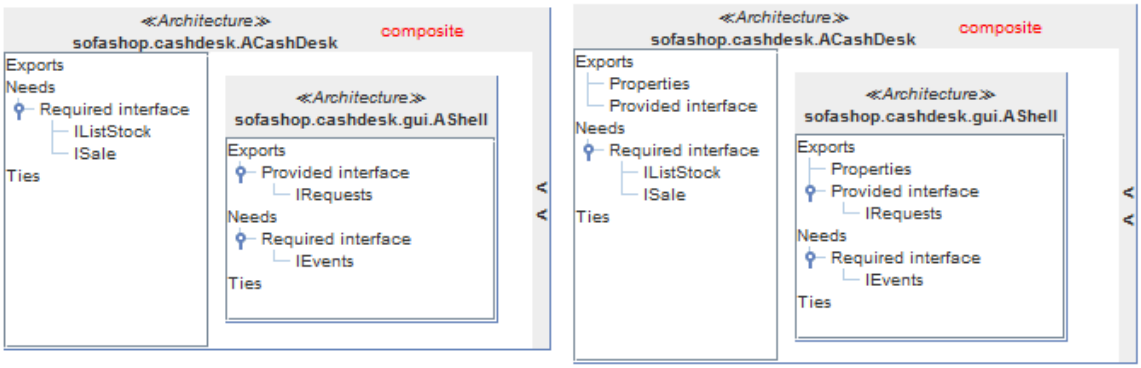
Hodnota parametru je zobrazena po přesunu kurzoru myši nad příslušný parametr (Obrázek 48).



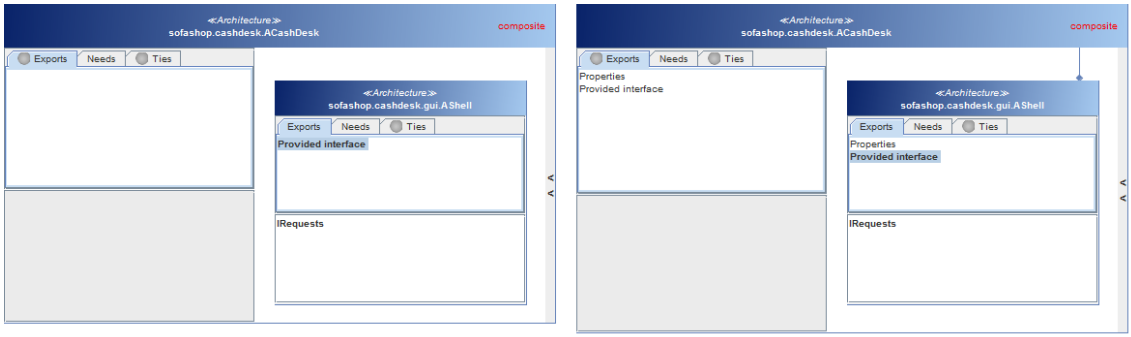
OBRÁZEK 48 – ZOBRAZENÁ HODNOTA PARAMETRU

11.4 OSTATNÍ VÝHODY

SOFA 2 loader byl upraven, aby došlo k uložení i prázdných traitů. Rozdíl je na první pohled patrný, jelikož u každé komponenty je možné zjistit dostupné traity daného komponentového modelu (Obrázek 49 a Obrázek 50). Tato změna se netýká skupinové reprezentace (Grouped), která prázdné traity nezobrazuje.

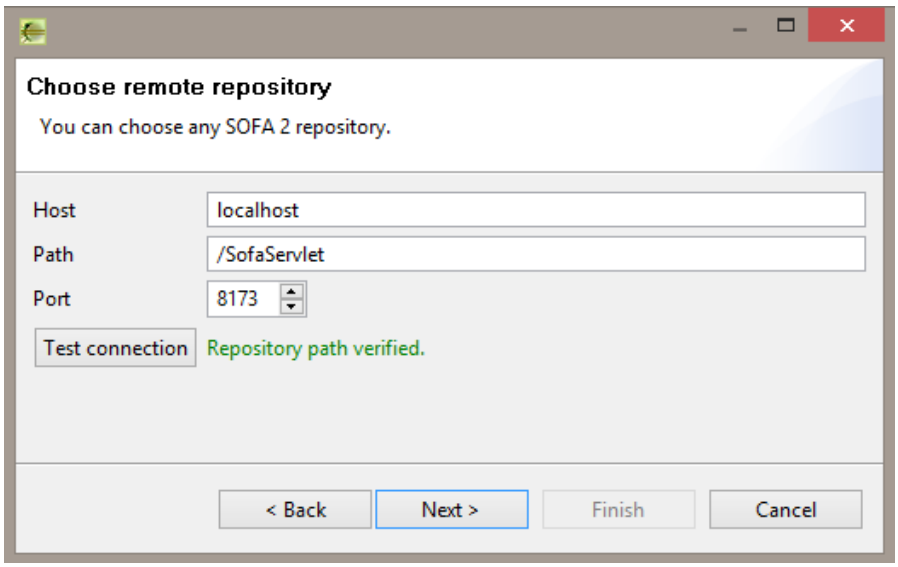


OBRÁZEK 49 – ROZDÍL MEZI ZOBRAZENÍMI KOMPONENT BEZ PRÁZDNÝCH A S PRÁZDNÝMI TRAITY (TREE)



OBRÁZEK 50 – ROZDÍL MEZI ZOBRAZENÍMI KOMPONENT BEZ PRÁZDNÝCH A S PRÁZDNÝMI TRAITY (BOOKMARK)

Bylo upraveno okno průvodce při načtení SOFA 2 aplikace z repozitáře, které je nyní přehlednější (Obrázek 51).



OBRÁZEK 51 - PRŮVODCE NAČTENÍM SOFA 2 APLIKACE Z REPOZITÁŘE

12 ZÁVĚR

Cílem této práce byla identifikace a odstranění nedostatků během načítání aplikací vyvíjených ve *frameworku SOFA 2* a v zobrazování analyzovaných aplikací v nástroji *AIVA*. To v první řadě zahrnuje nastudování teorie o technologiích, které se zobrazování a načítání týkají, tyto teoretické znalosti jsou sepsány v kapitolách 2-6. Poté přišla na řadu praktická část. Po otestování pluginu AIVA byly zanalyzovány požadavky expertů na SOFA 2, z těchto požadavků vznikl seznam možných vylepšení. Všechna tato vylepšení byla implementována a tím byly splněny všechny cíle této práce.

V předchozí implementaci byla odhalena slabá místa. Zásadní problém se skrýval v implementaci hierarchické struktury grafů, kde se do sebe vnořovali jednotlivé instance komponent s grafem. Proto se velká část práce orientovala právě na reimplementování této části, což se konkrétně týkalo pluginů AIVA a JGraphBasic. Překážky, které se během této reimplementace objevily, vzešly především z externí knihovny JGraphX. V první řadě šlo o problém v interakci objektů z této knihovny s komponentami z knihovny Swing. Další problém spočíval v překrývání různých objektů, knihovna JGraphX byla rozšířena o funkci pro nastavení priority viditelnosti. Nakonec se podařilo dokončit stabilní implementaci, která využívá nativních funkcí knihovny JGraphX pro zobrazování hierarchické struktury.

Další vylepšení se zabývala úpravou struktury komponent v seznamu projektů, kde se nyní zobrazují i komponenty na nižších hierarchických stupních. Tato změna se týkala nejen pluginu AIVA, ale i nástroje ComAV. Dále došlo ke změně v SOFA 2 loaderu, kde bylo přidáno načítání hodnot parametrů a uložení prázdných traitů do ENT modelu. V poslední řadě byly provedeny některé úpravy, které nejsou v práci zmíněny, jelikož se jednalo o velmi jednoduché změny.

Celkově reimplementace zasáhla jak do projektu ComAV, tak i do jeho 3 pluginů, které se týkají SOFA 2 vizualizace (AIVA, JGraphBasic a SOFA 2 loader). Celkem bylo upraveno 33 tříd, z toho 14 tříd podstatným způsobem, 2 třídy přibyly, naopak došlo k odstranění 1 třídy.

Prototyp pro načítání a zobrazování SOFA 2 aplikací byl dotažen do funkční verze, lze zde najít i další místa pro zlepšení, zejména v hierarchickém layoutu, který zatím u aplikací se složitou strukturou není bezchybný. Tento problém se ale týká externí knihovny JGraphX, o problémech se podle aktivity na oficiálním fóru ví a postupně se pracuje na jejich odstranění. Zajímavostí určitě je, že používání štítku *layout* u příspěvků na fóru je na druhém místě v četnosti, na prvním místě je použití štítku *jgraphx*, což lze vzhledem k názvu knihovny předpokládat. Právě layouty jsou tedy pravděpodobně nejproblematičtější částí knihovny.

Literatura

[1] BACHMANN, Felix, Len BASS, Charles BUHMAN, Santiago COMELLA-DORDA, Fred LONG, John ROBERT, Robert SEACORD a Kurt WALLNAU. *Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition*. Pittsburgh, 2000. CMU/SEI-2000-TR-008. [online]. [cit. 2013-06-20]. Dostupné z: http://www.diku.dk/OLD/undervisning/2005f/336/bachman00.pdf. Technical Report. Carnegie Mellon University.

[2] LAU, Kung-Kiu. *Component-based software development: case studies*. Singapore: World Scientific, 2004, viii, 303 s. ISBN 98-123-8828-1.

[3] ČERNÝ, Ondřej, Petr HOŠEK, Michal PAPEŽ a Václav REMEŠ. *SOFA 2 Component System*. Prague. [online]. [cit. 2013-06-20]. Dostupné z: http://sofa.ow2.org/docs/pdf/users_guide.pdf. User's Guide. Charles University in Prague.

[4] HERMANN, Lukáš. *User documentation of the SOFA 2 UML*. Prague, 2011. User's guide. [online]. Charles University in Prague.

[5] OMG's MetaObject Facility. OBJECT MANAGEMENT GROUP, Inc. *OMG* [online]. 2013 [cit. 2013-06-23]. Dostupné z: http://www.omg.org/mof/

[6] ŠNAJBERK, J., BRADA, P. ENT: A Generic Meta-Model for the Description of Component-Based Applications. *Electronic Notes in Theoretical Computer Science,* 2011, roč. 279, č. 2, s. 59-73. ISSN: 1571-0661

[7] BRADA, Přemysl. *The ENT Meta-Model of Component Interface, version 2*. Plzeň, 2004. DCSE/TR-2004-14. [cit. 2013-06-20]. Dostupné z: http://www.kiv.zcu.cz/site/documents/verejne/vyzkum/publikace/technicke-zpravy/2004/tr-2004-14.pdf. Technical Report. Západočeská univerzita.

[8] ŠNAJBERK, J., BRADA, P. Implementation of a data layer for the visualization of component-based applications. In *Information Technologies - Applications and Theory.* Seňa: PONT s.r.o., 2010. s. 55-62. ISBN: 978-80-970179-4-1

[9] SLOUP, Martin. *Loadery komponentových modelů*. Plzeň, 2011. Diplomová práce. Západočeská univerzita, Fakulta aplikovaných věd. Vedoucí práce Šnajberk, Jaroslav.

[10] JGraphX (JGraph 6) User Manual. *JGraph* [online]. [cit. 2013-06-05]. Dostupné z: http://jgraph.github.io/mxgraph/docs/manual_javavis.html

[11] ŠNAJBERK, Jaroslav. *Interactive Visualization of ComponentBased Applications*. Plzeň, 2011. DCSE/TR-2011-03. Technical Report. Západočeská univerzita.