Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Poloautomatická tvorba scénářů pro testování EFP

Plzeň, 2014 Martin Kožíšek

PROHLÁŠENÍ

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

V Plzni dne 10. 5. 2014 ……

Martin Kožíšek

PODĚKOVÁNÍ

Na tomto místě bych rád poděkoval panu Ing. Richardu Lipkovi, Ph.D. za vstřícný přístup, ochotu a čas věnovaný konzultacím při vypracovávání této diplomové práce.

ABSTRACT

Semi-automic generation of scenarios for testing EFP

The main goal of this diploma thesis is to design and to implement a tool for semi-automatic generation of testing scenarios for the SimCo framework. This program was developed at Department of Computer Science and Engineering at University of West Bohemia. The SimCo framework runs in the OSGi environment using Spring DM and it is used for testing of the component applications in order to verify extra-functional properties.

Developed application provides users GUI to pick components for test, gaining all information needed automatically and makes possible to create events for simulation. Consequently it is possible to generate identifications of components and descriptions of events in form of test scenario for the SimCo framework.

ABSTRAKT

Hlavním cílem této diplomové práce je návrh a implementace nástroje pro poloautomatické generování testovacích scénářů pro SimCo framework, který byl vyvinut na Katedře informatiky a výpočetní techniky na Západočeské univerzitě v Plzni. SimCo framework běží v prostředí OSGi s využitím Spring DM a slouží k testování komponentových aplikací za účelem verifikace mimofunkčních charakteristik.

Vytvořená aplikace poskytuje uživateli GUI pro výběr komponenty určených pro test, automaticky o nich získá veškeré potřebné informace a následně umožní vytvářet události pro simulaci. Identifikace komponent a popisy událostí je pak možné vygenerovat do souboru v podobě testovacího scénáře pro SimCo framework.

OBSAH

1 Úvod

S rozvojem informačních technologií se vyvíjejí i metodiky pro programování a proces vývoje softwaru obecně. Narůstající složitost a stále větší snahy o zefektivňování vývojového procesu a snižování ceny výsledného softwaru vedly od procedurálního programování přes objektově orientované programování až ke komponentově orientovanému programování. To je postavené na principu skládání výsledného systému ze samostatných funkčních celků se skrytou implementací – komponent. Díky tomu je možné znovupoužít dříve vytvořený kód, nebo využít otestované a optimalizované řešení třetích stran.

Při takovém přístupu je důležitý popis funkčních a mimofunkčních charakteristik, aby byly spojované komponenty vzájemně kompatibilní. V případě mimofunkčních charakteristik, jako je například doba odezvy, je problémem zjištění jejich hodnot. Vzhledem k tomu, že analytické metody sice mohou být přesné, ale kvůli své složitosti v praxi těžko aplikovatelné, je řešením získání konkrétních hodnot pomocí simulace.

Pro tento účel vznikl na Katedře informatiky a výpočetní techniky nástroj *SimCo Framework,* který umožňuje testovat komponentové aplikace v prostředí OSGi. Vstupem tohoto simulátoru je testovací scénář obsahující identifikaci testovaných komponent a definice událostí simulace, jež mají být vyvolané *SimCo frameworkem*. Tento scénář je v současné době vytvářen manuálně, což klade vysoké nároky na uživatele, neboť musí zjišťovat informace o komponentách z různých zdrojových a konfiguračních souborů. Vzhledem ke skutečnosti, že se jedná o XML soubor, musí uživatel napsat značné množství režijního kódu a navíc musí znát požadovaný formát XML.

Úkolem této práce je proto návrh a implementace aplikace, která umožní jednoduché vytváření popisu událostí a nastavení parametrů simulace a následné generování testovacího scénáře v podobě XML souboru pro *SimCo framework*. Při tom budou veškeré potřebné údaje o komponentách, které bylo doposud nutné dohledávat, zjišťovány automaticky z dostupných zdrojů.

2 KOMPONENTOVÉ APLIKACE

2.1 Úvod do komponentového programování

2.1.1 Komponenta

Komponenty jsou části, které dávají dohromady celek. Obecně lze říci, že každý softwarový systém se skládá z komponent – jsou to části, jež vzniknou při dekompozici řešeného problému. Myšlenka komponentově orientovaného přístupu tudíž není nic nového pod sluncem. Takovéto označení je ale příliš obecné, a proto je nutné v případě CBSE (*Component-Based Software Engineering*) přesněji definovat, co je to komponenta a jak spolu komponenty komunikují. [BAC00].

Vzhledem k různému chápání problémů a různému přístupu k jejich řešení se objevuje mnoho definic komponenty. Konkrétně podle [BAC00] – komponenta:

- má skrytou implementaci funkcionality,
- je vhodná pro využití třetí stranou,
- odpovídá komponentovému modelu (viz 2.1.3).

Komponentu můžeme chápat jako „black-box“ výpočetní jednotku (vnitřní implementace je neznámá), přičemž komunikace s okolím probíhá přes definovaná rozhraní. Z jednotlivých komponent lze následně vytvořit větší funkční celek.

2.1.2 Proč komponentové aplikace

Se stále zvyšujícími se nároky na software roste i jeho složitost a s tím jsou spojené i rostoucí nároky na vývoj. Postupem času proto vznikala a vznikají různá programovací paradigmata pro vývoj softwarových produktů. S nástupem vyšších programovacích jazyků přišlo Objektově orientované programování (OOP), jehož hlavními pilíři jsou zapouzdření, dědičnost a polymorfismus. Ještě dále zachází komponentově orientovaný přístup vývoje softwaru (CBSE), který klade ještě větší důraz na zapouzdření, znovupoužitelnost a tzv. *separation of concerns* (oddělení zodpovědností).
Komponentově orientovaný přístup s sebou pochopitelně nese své výhody, ale i nevýhody, přičemž výhody ve většině případů převažují [HOL11].

Výhody

- Urychlení vývoje aplikace.
- Zlevnění ceny softwaru, např. díky znovupoužití existujících komponent.
- Modularita (nahraditelnost).

Vývoj aplikace může významně urychlit použití již existujících komponent (i třetích stran), jelikož není třeba danou část aplikace programovat od začátku. Důležité je rovněž, že znovupoužité komponenty jsou již ve většině případů otestované a dobře odladěné, ušetří se tudíž i čas nutný pro testování a opravu chyb. Zejména v případě široce používaných komponent třetích stran lze s vysokou pravděpodobností předpokládat, že případné chyby jsou mnohem rychleji objeveny a následně opraveny. S rychlejším vývojem pak ruku v ruce jde i nižší cena výsledného softwaru. Další výhoda souvisí s vysokou mírou modularity. Změna implementace konkrétní komponenty nevyžaduje změnu dalších částí systému (nedojde-li ke změně rozhraní).

Nevýhody

Při komponentově orientovaném přístupu je ovšem nutné mít na paměti i z toho plynoucí úskalí:

- Systém je často těžké sestavit, udržovat.
- Komponenty se vyvíjejí (sledování jejich vývoje).

Vývoj komponent s cílem vytvořit softwarový systém je náročnější na návrh architektury. Jednotlivé komponenty by na sobě měly být co nejvíc nezávislé a komunikace by měla probíhat výhradně přes daná rozhraní. Tato rozhraní je tudíž nutné mít dobře nadefinovaná, protože pozdější změna rozhraní jedné komponenty může vyžadovat i změny v jiných částech systému. Při používání komponent třetích stran může být komplikovanější integrace s vyvíjeným systémem, neboť ne vždy takovéto komponenty plně odpovídají konkrétním požadavkům. Vzhledem k tomu, že se komponenty běžně v čase vyvíjejí, může po určité době nastat nutnost přechodu na novou verzi, což může být spojeno například se změnou rozhraní (a s tím souvisejí výše popsané komplikace).

2.1.3 Komponentový model

Komponentovým modelem se rozumí specifikace požadavků definující, co je to komponenta, jak probíhá interakce (komunikace) mezi komponentami, jakým způsobem se provádí instalace a odinstalace, jaké jsou stavy komponenty, případně další požadavky. Neexistuje totiž všeobecná shoda na tom, co vše by mělo být v rámci komponentového modelu specifikováno [BAC00].

2.1.4 Komponentový framework

Implementací komponentového modelu je komponentový framework. Jedná se o běhové prostředí pro komponenty, jež má za úkol především řídit zdroje, poskytovat služby pro komunikaci a spravovat životní cyklus komponent. Jeho funkce se dá

přirovnat k operačnímu systému, ačkoliv pracuje na mnohem vyšším úrovni abstrakce [HOL11].

2.1.5 Modularita a Java

Vývoj monolitických aplikací je v dnešní době vzhledem ke stále větším nárokům na software již záležitost minulosti. Aplikace jsou skládány z malých, dobře definovaných modulů, které schovávají svoji implementaci za stabilní API. To vše umožňuje snazší údržbu, testování a porozumění aplikaci. Java, jakožto jeden z nejrozšířenějších programovacích jazyků, ve své podstatě nabízí, co se modularizace týče, pouze rozdělování na metody, z nichž se skládají třídy a ty se dále sdružují do balíků (*package*). Za jednotku modularity v jazyce Java je pak obecně považován JAR soubor. Ten je ovšem pouze jakýmsi obalem pro třídy, rozhraní a další zdroje a po umístění na *classpath* zmizí veškeré ohraničení a jakákoliv veřejná třída je přístupná z jakékoliv jiné třídy. Další nevýhodou je nemožnost verzování [WAL09].

Tato a další omezení, jež neumožňují vytvořit skutečně modulární software s oddělenými „black-box“ komponentami, vedla ke vzniku OSGi specifikace, která přináší mnohem pokročilejší možnosti modularizace.

Modularita

Mezi klíčové vlastnosti modulů patří vysoká soudržnost a nízká míra propojení. Modul by měl mít na starosti jeden úkol, na ten se soustředit a neobsahovat něco, co bezprostředně nesouvisí s danou funkčností. Výsledkem by měly být dobře členěné, robustní, znovupoužitelné a lépe pochopitelné moduly. Zároveň by měl modul interagovat s okolím pouze přes stabilní API bez ohledu na vnitřní implementaci. Změna implementace by tudíž neměla mít žádný vliv na ostatní moduly [WAL09].

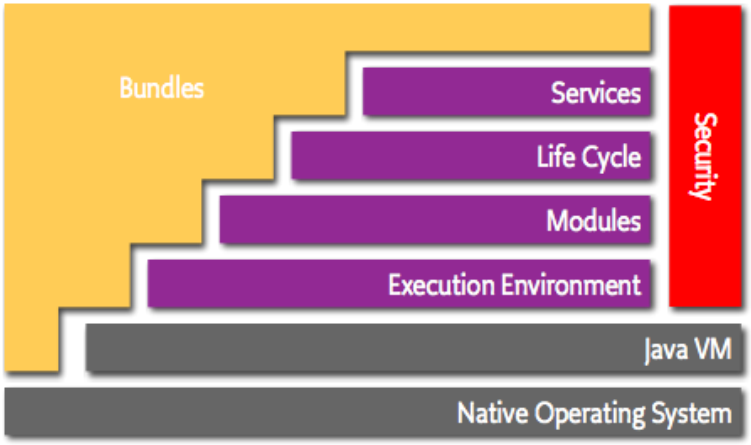
Výhody:

- Měnitelnost – pokud je každý modul znám jen přes své rozhraní, pak je jednoduché změnit modul za jiný se stejným interface.
- Srozumitelnost – kompaktní moduly s dobře definovanými hranicemi jsou mnohem jednodušší pro pochopení, což vede i ke snazšímu pochopení celého systému.
- Paralelní vývoj – moduly lze vyvíjet paralelně a nezávisle na sobě.
- Znovupoužitelnost – vytvořené moduly je možné použít i v jiných aplikacích.

2.2 OSGi

V současné době lze za jednu z nejpokročilejších a nejpoužívanějších komponentových technologií považovat OSGi (*Open Services Gateway initiative*).

OSGi jako takové je soubor specifikací, které definují dynamický komponentový systém pro jazyk Java. Tyto specifikace umožňují vytvářet systém dynamicky složený z mnoha odlišných znovupoužitelných komponent. Vnitřní implementace komponent zůstává v OSGi skryta, přičemž komunikace se provádí prostřednictvím tzv. služeb (*services*) [OSG14].



Obr. 2.1: Zobrazení vrstev modelu OSGi (zdroj: OSGi – www.osgi.org)

Na obrázku 2.1 jsou zobrazeny jednotlivé části OSGi ve vrstvách. V následujících odrážkách jsou definice klíčových pojmů [OSG14]:

- *Bundle* – označení komponenty v OSGi.
- *Služby (Services)* – zajištění komunikace mezi bundly dynamicky na základě modelu *publish-find-bind*.
- *Životní cyklus (Life Cycle)* – API pro instalaci, spuštění, zastavení, update a odstranění bundlů.
- *Moduly* – vrstva definující export a import kódu bundlů.
- *Zabezpečení (Security)* – zajištění bezpečnostních aspektů.
- *Běhové prostředí (Execution environment)* – definuje metody a třídy dostupné na dané platformě.

Třemi klíčovými aspekty OSGi jsou bundly, životní cyklus a služby. Celkově lze OSGi považovat za poměrně vyzrálý standard, neboť první verze byla vydána již v roce 2000 [EDG13]. Aktuální verzí OSGi je verze 5 (R5) uvolněná v červnu 2012.

2.2.1 Výhody OSGi

U složitějších systémů, na kterých pracuje větší množství lidí (případně týmů), může při případných změnách a rozšířeních snadno docházet k narušování systému. Jednoduché změny v implementaci mohou způsobit rozbití aplikace na úplně jiném místě. Další problémy mohou nastat při změně jednoho rozhraní, neboť nikdo s jistotou neví, zda

nezpůsobí nefunkčnost klientů. OSGi díky efektivní modularizaci snižuje zmíněná rizika. Toho lze dosáhnout správným návrhem modulů, které mezi sebou společně interagují k dosažení nějakého výsledku (cíle) [CAL12].

Výhodou OSGi je i fakt, že se v základu nejedná o přehnaně komplexní a komplikovanou technologii, která by se snažila pokrýt veškeré možné požadavky, jež na ni mohou být kladeny. Tím se lze vyhnout situaci, kdy aplikace používá pouze malou část nabízených vlastností, a přesto za běhu zabírá megabajty paměti a start (nasazení) aplikace trvá desítky sekund. OSGi lze uzpůsobit potřebám uživatele, neboť jsou nabízeny pouze základní služby a v případě potřeby lze další jednoduše doinstalovat. Další výhodou je velmi snadné přizpůsobení klasického Java JAR souboru na OSGi bundle. V nejjednodušším případe stačí přidat pár řádek do *manifest* souboru [CAL12]. V neposlední řadě je výhodou přenositelnost, neboť OSGi využívá pouze standardní prostředky jazyka Java bez nativního kódu.

2.2.2 Bundle

Jednotkou modularizace je *bundle*, jenž vynucuje skrytí vnitřních informací, z čehož plynou především následující výhody [CLA12]:

- zaměnitelnost,
- srozumitelnost,
- nezávislý vývoj.

Bundle je v OSGi definován jako modul zabalený do JAR souboru, který obsahuje manifest s povinnou hlavičkou `Bundle-SymbolicName` (identifikátor bundlu).

OSGi headers (hlavičky), obsažené v souboru *manifestu*, představují důležité informace pro OSGi.

Mezi nejdůležitější patří:

- `Bundle-SymbolicName` – symbolické jméno bundlu (jediná povinná hlavička).
- `Bundle-Name` – slouží k označení bundlu uživatelsky přívětivou formou (nemá žádný speciální význam).
- `Bundle-Version` – verze, společně se symbolickým jménem jednoznačně identifikují bundle v OSGi Frameworku.
- `Import-Package` – deklarace závislosti – vyžadovaných balíků (*packages*) s možností určit požadované verze (minimální, maximální atd.).
- `Export-Package` – určení balíků (*packages*), které jsou poskytovány (zpřístupněny) ostatním bundlům.
- `Bundle-ManifestVersion` – určuje použitou OSGi specifikaci, označení však není příliš intuitivní – číslo 2 značí OSGi release 4 a vyšší.

Pokud má být určitý balík tříd bundlu přístupný i v jiných bundlech, musí být v souboru manifestu hlavička `Export-Package` s daným balíkem. Bundle, který naopak chce využívat jiným bundlem exportované třídy, musí mít v manifestu hlavičku `Import-Package`. Příklad *manifest* souboru je na ukázce kódu 2.1.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: SimulationFramework.core
Bundle-SymbolicName: framework.core
Bundle-Description: This bundle provides simco core's interface.
Bundle-Version: 1.0.0
Export-Package: simco.framework.core;version="1.0.0"
Import-Package: simco.framework.scheduler.calendar;version="1.0.0"
```

Ukázka kódu 2.1: Část manifest souboru pro OSGi bundle

Z výše uvedeného mimo jiné vyplývá, že balíky, které nejsou explicitně exportovány, jsou pro okolí skryty, a komponenty lze tudíž označit jako „black-box“. To je významná výhoda oproti klasickým Java JAR souborům, neboť ty jakýkoliv podobný mechanismus postrádají a vše označené jako „public“ je z vnějšku přístupné. Je ovšem potřeba mít na paměti, že pro skrytí implementace komponenty je nutné vhodně zvolit exportované balíky. V případě exportu všech balíků bundlu by se veškeré výhody plynoucí ze zapouzdření vytratily, neboť by nastala stejná situace jako u klasických JAR souborů.

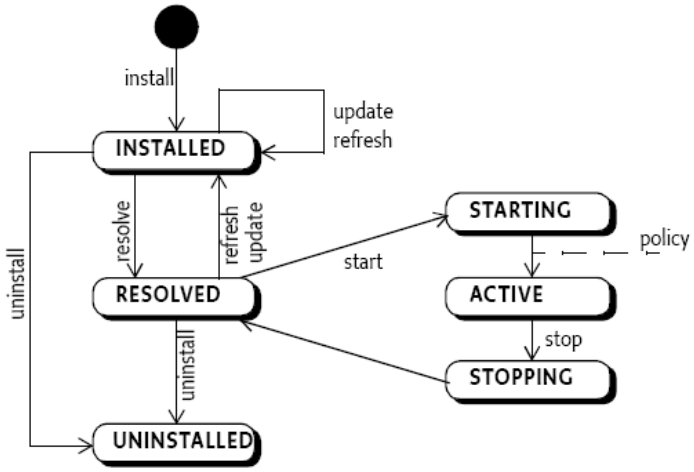
2.2.3 Životní cyklus

Vrstva *Life cycle* specifikuje mimo jiné stavy, ve kterých se může bundle v běhovém prostředí nacházet.

Stavy [EDG13]:

- *installed* – stav bundlu na začátku po instalaci (než bude možno bundle spustit, je nutné zkontrolovat závislosti),
- *resolved* – do toho stavu se bundle dostane po úspěšné kontrole závislostí (bundle stále není aktivní, ale již nic nebrání ve spuštění),
- *starting* – pouze přechodný stav při spuštění, framework následně převede bundle do stavu *active*,
- *active* – bundle je aktivní (nabízí i využívá služby),
- *updating* – aktualizace všech závislostí, přechod zpět do stavu *installed*,
- *stopping* – přechod z aktivního do *resolved* stavu,
- *unistalled* – po odinstalování.

Schematicky jsou stavy a přechody mezi nimi zobrazeny na obrázku 2.2.



Obr. 2.2: Stavy bundlu (zdroj: OSGi Service Platform – Core Specification)

V klasických Java třídách existuje známá metoda `main()`, která je zavolána při spuštění. Analogií v OSGi je metoda `start()` v rozhraní `BundleActivator`. Tato metoda je volána frameworkem po startu bundlu. Třída obsahující tuto metodu musí implementovat zmíněné rozhraní `BundleActivator`. V OSGi je navíc metoda `stop()` volaná při ukončení činnosti bundlu.

2.2.4 Řešení změn komponenty v OSGi

Dojde li ke změně bundlu, mohou teoreticky nastat 2 situace:

- změna vnitřní implementace,
- změna rozhraní.

V prvním případě stačí vytvořit novou verzi bundlu a provést instalaci nebo update.

Ve druhém případě je situace komplikovanější, neboť změny mohou zasáhnout i do dalších komponent (bundlů). Může proto nastat problém se zpětnou kompatibilitou. OSGi nabízí možnost verzovat jednotlivé bundly i exportované package, přičemž čísla verzí se nemusí nutně shodovat [CAL12].

Číslo verze se skládá ze 4 částí: `major.minor.micro.qualifier`.

Nejčastěji se používají 3 čísla bez kvalifikátoru (např.: 2.3.1).

Bundle, který například importuje package z jiného bundlu, může rovněž určit požadovanou verzi. Důležitá je vlastnost, že v OSGi Frameworku může být nainstalován jeden bundle (stejného jména) vícekrát v odlišných verzích. Díky tomu lze, za účelem zpětné kompatibility, v prostředí ponechat i starší verze bundlu [CAL12].

2.2.5 Služby (services)

Služby jsou stěžejní částí OSGi sloužící ke komunikaci mezi bundly.

Služba v OSGi má tři části:

- rozhraní (interface),
- implementace,
- service properties – mapa dvojic klíč-hodnota, slouží ke specifikaci dalších vlastností.

Komunikaci mezi bundly zajišťuje *OSGi service registry*, což je prostředník, který od sebe izoluje poskytovatele a konzumenta služby. Poskytovatel služby vlastní všechny výše zmíněné části – rozhraní, implementaci a volitelně properties. Konzument služby používá pouze rozhraní, případně properties, ale k vnitřní implementaci nemá přístup. Aby toto platilo, je potřeba mít na vědomí, že rozhraní a třídy implementující dané rozhraní musejí být v odlišném package, přičemž balík s rozhraním se pomocí hlavičky Export-Package zpřístupní ostatním bundlům, zatímco balík s implementací zůstane skryt [CAL12].

Pro poskytnutí služby vytvoří bundle objekt, který následně registruje prostřednictvím *OSGi service registry* pod jedním nebo více rozhraními. Ostatní bundly poté mohou v seznamu vybrat požadovaný objekt registrovaný daným rozhraním. Je umožněno registrovat různé služby pod stejným rozhraním. Je však zřejmé, že musí existovat nějaký mechanismus, jak jsou v takovém případě jednotlivé služby rozeznávány. Tím jsou *properities* – představují mapu dvojic klíč-hodnota a tím umožňují přidat atributy a jejich hodnoty. Bundle registrující službu použije vytvořené properties při registraci služby. Bundle, který hledá službu, může pro rozlišení dané služby použít tzv. LDAP filtr (*Lightweight Directory Access Protocol*).

Důležitou vlastností služeb je dynamičnost – služby se mohou dynamicky objevovat a mizet z registru služeb. To s sebou sice přináší jisté komplikace, neboť bundly musí při použití služby kontrolovat, zda je stále dostupná (a nebyla již odstraněna), na druhou stranu takový přístup umožňuje dynamické instalování bundlů za běhu (aniž by bylo nutné aplikaci přerušovat), což je jedna z významných předností OSGi [OSG14].

2.2.6 Události (events)

Jedná se o mechanismus pro interakci fungující na principu *publisher-subscriber*. Publisher generuje události a subscriber je přijímá, přičemž vazba mezi nimi může být m:n, neboť více publisherů může generovat události pro jeden subscriber a naopak jeden publisher může generovat události, které jsou sledovány více subscribery. Zprostředkovatelem této komunikace je *Event Admin Service*.

Subscriber (posluchač) zaregistruje službu s použitím `EventHandler` rozhraní a nastaveným „tématem“ (*topic*), o které má zájem. Události s daným tématem mu následně budou prostřednictvím `EventAdmin` posílány voláním *callback* metody `handleEvent()`, kterou musí subscriber implementovat.

Naproti tomu publisher musí nejprve získat službu `EventAdmin`. Následně vytvoří `Event` a ten voláním metody `postEvent()` (služby `EventAdmin`) pošle. `EventAdmin` poté rozešle, podle nastavené hodnoty *topic*, událost odpovídajícím posluchačům [CAL12].

2.2.7 Spring DM

Spring DM (*Spring Dynamic Modules*) vychází ze Springu a z OSGi. Využívá se zejména tzv. *dependency injection*, což zajišťuje vytvoření a pospojování jednotlivých bundlů, z pohledu Springu označovaných jako beany. Zásadní výhodou z pohledu OSGi aplikace je možnost definovat některé záležitosti jednoduše v XML souboru s tím, že se o dané úkony Spring DM sám postará. Jedná se především o registraci a získání služby nebo o nastavení metody, která se má zavolat po startu bundlu.

Na ukázce kódu 2.2 je příklad XML souboru s konfigurací pro Spring DM. Tento soubor je umístěn v podadresáři `spring` adresáře `META-INF`, v němž je OSGi manifest. Alternativně je možné umístit soubor jinam, cesta k němu je v takovém případě dána hlavičkou `Spring-Context` v souboru OSGi manifestu.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
             http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi-1.0.xsd">

  <osgi:reference id="CalendarService"
      interface="simco.framework.scheduler.calendar.ICalendar" />
  <osgi:reference id="SimCoContextService"
      interface="simco.framework.extension.ISimCoContext" />
  <osgi:reference id="SimcoSimulation"
      interface="simco.framework.scheduler.simulation.ISimcoSimulation" />

  <bean id="CoreService" class="simco.framework.core.CoreImpl">
       <constructor-arg ref="CalendarService" />
       <constructor-arg ref="SimCoContextService" />
       <property name="simcoSimulation" ref="SimcoSimulation" />
  </bean>

  <osgi:service id="CoreServiceOsgi" ref="CoreService"
      interface="simco.framework.core.ICore" />
</beans>
```

Ukázka kódu 2.2: Konfigurační soubor pro Spring DM

Na ukázce je rovněž vidět vytvoření bundlu s id `CoreService`, přičemž implementaci představuje třída `CoreImpl` z balíku `simco.framework.core`. Dále je vidět, že tento bundle využívá tři služby, dvě jsou Springem injektovány přes konstruktor, jedna pomocí *settru*. Příslušné tagy v XML souboru obsahují atribut `ref`, který odpovídá atributu `id` tagu `osgi:reference`. Právě pomocí tohoto tagu je získána OSGi služba, jež je specifikována atributem `interface`. Zde je patrná výhoda použití Springu, který se postará o komunikaci s OSGi, získá požadovanou službu a vrácenou referenci prostřednictvím *dependecy injection* vloží do odpovídající proměnné.

Uvedený bundle rovněž jednu službu poskytuje (registruje). K tomu je použit tag `osgi:service` s atributy `interface` (rozhraní, pod kterým je služba registrována) a `ref` (reference na implementaci – v tomto případe beanu s id `CoreService`).

2.2.8 OSGi Blueprint Service

Deklarativní model Spring DM eliminuje nutnost psaní kódu OSGi API při registraci nebo využívání služby. Vzhledem k tomu, že se tento přístup osvědčil, byl formalizován jako část OSGi specifikace s označením *OSGi Blueprint Service*. V ní je obsažen service model ze Spring DM a části jádra ze Springu. Spring DM se tak stal referenční implementací pro Blueprint Service. [WAL09]

2.2.9 OSGi frameworky

Již dříve bylo zmíněno, že OSGi je pouze soubor specifikací. Konkrétní implementace se označuje jako OSGi Framework, přičemž v současné době existuje několik rozšířených implementací. Mezi nejznámější patří *Equinox*, což je open source projekt společnosti Eclipse. Není bez zajímavosti, že známé vývojové prostředí Eclipse je postavené právě na OSGi a vytvoření pluginu pro Eclipse je ve své podstatě instalace nového bundlu. Dalšími rozšířenými implementacemi jsou *Apache Felix* (open source od společnosti Apache), či *Knopflerfish*.

3 SIMCO

3.1 Simulace

Obecně lze simulaci chápat jako techniku, při které je jeden systém nahrazen modelem, se kterým se experimentuje s cílem získat informace o původním zkoumaném systému.

Počítačová simulace může být využívána pro testování různých systémů, u nichž by reálné testy byly buď příliš nákladné, nebo by trvaly neúnosně dlouhou dobu, nebo by vůbec nebyly realizovatelné. Příkladem může být simulace dopravy ve městě, simulace chemických reakcí (např. v jaderné elektrárně), či strojírenské simulace (např. nového motoru). V dnešní době je spektrum oblastí a odvětví využívajících simulace velmi široké a rovněž účely daných simulací jsou různorodé.

Tato práce se zaobírá simulací za účelem testování komponent. Testování je zaměřeno na funkcionalitu poskytovanou komponentami. Základ frameworku umožňujícího nasimulování prostředí jedné či více reálných softwarových komponent a následné otestování funkčnosti prostřednictvím simulace vznikl na Katedře informatiky a výpočetní techniky v Plzni v rámci dvou diplomových prací [KAB11] a [PRO11] pod názvem *SimCo framework* (SIMulation of COmponent).

3.2 Popis SimCo frameworku

SimCo framework umožňuje testovat softwarové komponenty (konkrétně OSGi bundly) na principu událostní simulace. Základem je vytvořený framework, v jehož prostředí je možné spustit aplikaci, u které chceme testovat jednu nebo více komponent. Testovaná aplikace je označována jako *SimCo aplikace*. K testování je potřeba dodávat testované aplikaci potřebná vstupní data. V reálném systému by tato data pocházela z jiné části systému nebo od uživatele. Úkolem *SimCo frameworku* je umožnit uživateli simulovat funkčnost okolí komponenty a určit tím vstupní data, jež se budou komponentě dodávat. Uživatel má možnost zadat jak konkrétní hodnoty vstupních dat, tak časové intervaly, ve kterých jsou vstupní data komponentně předána [KAB11]. Důležitá je rovněž možnost měřit dobu, kterou komponenta potřebuje pro vykonání dané činnosti.

Kromě toho framework umožňuje kontrolu nad během simulace. Jednotlivé kroky jsou zaznamenávány a je rovněž umožněno simulaci pozastavit a znovu spustit, případně krokovat. Jednotlivé kroky (volání metody, návrat z metody) jsou spojeny s událostí v *SimCo frameworku* – systém je založen na principu událostní simulace s metodou interpretace událostí [KAB11].

Jak již bylo nastíněno, simulována je činnost pouze některých komponent – těch, co poskytují vstupní data. Zbytek tvoří reálné komponenty, které provádějí reálný výpočet a jejichž otestování je hlavním cílem.

3.2.1 Komponenty

V prostředí SimCo Frameworku se rozlišují tři druhy komponent. Prvním typem jsou reálné testované komponenty, druhým typem jsou simulační komponenty. Simulační komponenty mají svůj běh pouze nasimulovaný a jejich hlavním účelem je poskytování vstupních dat pro reálné komponenty. Třetím typem je tzv. intermediate komponenta (prostředník). Jejím úkolem je zajišťovat komunikaci se *SimCo frameworkem* v případě, kdy mezi sebou komunikují dvě reálné komponenty. Tím je zaručeno, že je možné testovat obecně jakoukoliv aplikaci rozdělenou na Spring DM bundly, aniž by o *SimCo frameworku* věděly a aniž by bylo nutné upravovat testované komponenty. Prakticky to funguje tak, že intermediate komponenta má zaregistrované stejné služby jako volaná reálná komponenta. Volající komponenta pak nepoužije služby reálné komponenty, ale služby intermediate komponenty, která následně použije služby volané komponenty. To umožňuje sledovat a řídit komunikaci mezi dvěma reálnými komponentami ze *SimCo frameworku* [KAB11].

Pro úplnost je ještě vhodné doplnit, že se v prostředí vyskytuje čtvrtý typ komponent obsahující implementaci *SimCo frameworku*.

3.2.2 Události

SimCo framework pracuje na principu diskrétní událostní simulace a používá metodu interpretace událostí. Jakákoliv komunikace mezi komponentami je nějaký typ události a je *SimCo frameworkem* zachycena.

Typy událostí:

- `REAL_CALL` – reálná komponenta volá metodu některé jiné komponenty *SimCo aplikace*.
- `REAL_RETURN` – návrat z volané metody reálné komponenty.
- `SIMULATION_CALL` – simulační komponenta volá metodu některé jiné komponenty *SimCo aplikace*.
- `SIMULATION _RETURN` – návrat z volané metody simulační komponenty.
- `SINGULAR` – událost vyskytující se pouze jednou.
- `REGULAR` – událost vyskytující se v pravidelných intervalech.
- `CASUAL` – událost, u které je možno nadefinovat, s jakou pravděpodobností se bude vyskytovat.

3.2.3 Architektura

SimCo framework je aplikace vyvinutá a běžící v prostředí OSGi s využitím Spring DM. Skládá se celkem z pěti bundlů, které jsou blíže specifikovány na následujících řádcích. Pro úplnost je třeba zmínit, že šestým bundlem je OSGi komponenta `EventAdmin`. Ta je použita k publikování událostí mezi jednotlivými bundly. Schéma aplikace je na obrázku 3.1.

framework.scheduler

Zajišťuje činnosti spojené s diskrétní událostní simulací (činnost kalendáře, běh, historie průběhu simulace atd.).

framework.core

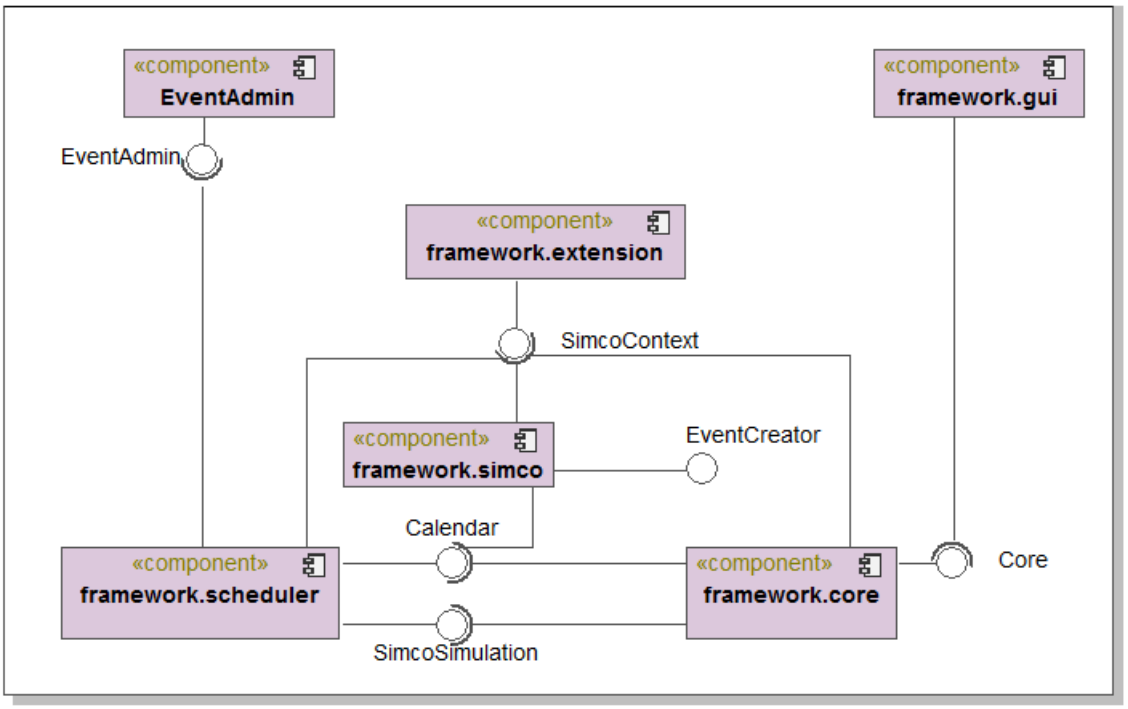
Obsahuje funkce potřebné pro ovládání a informování o činnosti *SimCo frameworku*.

framework.simco

Zprostředkovává spojení *SimCo frameworku* a *SimCo aplikace* – jsou na něj napojeny simulační a intermediate komponenty.

framework.extension

Obstarává práci s XML a obsahuje datové třídy využívané zbytkem frameworku.



Obr. 3.1: SimCo (zdroj: [KAB11])

framework.gui

Zajišťuje grafické uživatelské rozhraní pro ovládání a vizualizaci průběhu simulace.

3.2.4 Napojení SimCo aplikace na SimCo framework

Pro správné napojení testované aplikace na *SimCo framework* je nutné provést několik úkonů.

Komponenta typu SIMULATION

Tato komponenta je vytvořena k simulování okolí testované aplikace. Může například dodávat vstupní data v daných intervalech. Spojení se *SimCo frameworkem* je udržováno přes specifikované OSGi služby. Zároveň musí být zaregistrována jako posluchač událostí bundlu `EventAdmin`, aby mohla reagovat na vzniklé události. Třída implementující rozhraní, přes které je registrována OSGi služba, musí dědit od abstraktní třídy `simco.framework.simco.ASimco` [KAB11].

Ukázka konfiguračního souboru pro Spring DM a další podrobnosti jsou uvedeny v [KAB11].

Komponenta typu INTERMEDIATE

Spojení se *SimCo frameworkem* je podobné jako u komponenty typu `SIMULATION`. Hlavním účelem této komponenty je zachytávání komunikace mezi dvěma reálnými komponentami – být prostředníkem komunikace. Z toho vyplývá, že musí registrovat stejné služby, jako má reálná komponenta, pro kterou je prostředníkem. K těmto službám se navíc přidá parametr `simul=true`, čímž se odliší od služeb reálné komponenty. Na rozdíl od simulační komponenty se v tomto případě využívá dědění od `simco.framework.simco.AIntermediate` [KAB11].

Komponenta typu REAL

Výhodou *SimCo frameworku* je skutečnost, že kód reálné komponenty není nutné upravovat. Jediná výjimka nastává v případě, kdy reálná komponenta komunikuje s jinou reálnou komponentou. Jak již bylo zmíněno, komunikace v takovém případě probíhá přes prostředníka, jenž nabízí stejné služby. Aby byla vybrána služba prostředníka, je nutné pro využívané služby nastavit filtr `simul=true`. Jedná se pouze o přidání jednoho atributu do XML konfiguračního souboru komponenty.

Scénář

Hlavní konfigurace testování komponentové aplikace je umístěna v XML souboru a je označována jako scénář.

V tomto souboru jsou nadefinovány veškeré komponenty *SimCo aplikace*. Pro každou komponentu je uveden typ a jméno OSGi bundlu. Jedná-li se o komponentu typu `REAL` nebo `SIMULATION`, je zde rovněž cesta ke konfiguračnímu souboru komponenty (viz dále). V případě reálné komponenty komunikující přes prostředníka musí být uvedeno jméno OSGi bundlu `INTERMEDIATE` komponenty.

Kromě identifikace komponent jsou zde umístěny definice událostí simulace. Konkrétně se jedná o události typu `SINGULAR`, `REGULAR` a `CASUAL`. Díky tomu lze definovat volání metod (s nastavenými parametry) dané komponenty včetně údaje o tom, kdy má k volání dojít, případně s jakou periodou se má opakovat (záleží na typu události). Dále je možné zadat očekávanou návratovou hodnotu a maximální požadovanou dobu trvání události.

Konfigurační soubory komponent

Každá komponenta typu `SIMULATION` a `REAL` má ještě svůj vlastní XML konfigurační soubor. Ten obsahuje povinně definici všech tříd implementujících rozhraní, přes které je registrována OSGi služba. Dále je možno nadefinovat parametry jednotlivých metod (například zpoždění, které má nastat při volání dané metody).

4 MIMOFUNKČNÍ CHARAKTERISTIKY (EFP)

4.1 Ověření kompatibility komponent

Jak již bylo uvedeno v kapitole 2, softwarové produkty v dnešní době narůstají z hlediska velikosti i složitosti. Dalším znakem současného softwarového vývoje jsou snahy o co nejkratší dobu vývoje a co nejnižší cenu výsledného produktu. Logickým vyústěním jsou snahy o znovupoužití již hotového kódu, či využití hotového, otestovaného a optimalizovaného řešení nabízeného třetí stranou. Výše uvedené skutečnosti vedou na komponentově orientovaný přístup vývoje software, kdy je výsledný systém sestaven z jednotlivých komponent, které poskytují požadovanou funkcionalitu.

Skládání systému z nezávisle na sobě vyvíjených částí (komponent) s sebou ovšem nese rizika, zejména v případě, kdy se využívají komponenty třetích stran. Pro správnou funkčnost celého systému je totiž nezbytné, aby spojované komponenty byly vzájemně kompatibilní.

Požadavky na kompatibilitu jsou nejčastěji děleny na:

- funkční požadavky (*functional requirements*),
- mimofunkční požadavky (*extra-functional / non-functional requirements*).

Funkční požadavky definují hlavní účel a služby, které má daná komponenta poskytovat. Mimofunkční požadavky definují kvalitativní aspekty poskytovaných služeb. Těmi mohou být například doba odezvy, paměťové nároky, spolehlivost, bezpečnost a mnohé další. Podrobnější popis je v kapitole 4.2. Oba typy požadavků (funkční i mimofunkční) jsou pro verifikaci kompatibility důležité. Jelikož funkční požadavky nejsou novým konceptem, jsou dobře podporovány v současných technologiích – např. poskytované a požadované služby v OSGi. Naproti tomu mimofunkční požadavky jsou stále předmětem výzkumu a v současné době nemají podporu v používaných komponentových technologiích. Jako příklad lze uvést OSGi nebo Spring, které neposkytují možnosti pro popis mimofunkčních charakteristik [JEZ12].

Právě nedostatečná podpora popisu mimofunkčních charakteristik vede k problémům spojeným s nekompatibilitou komponent, což ve výsledku brzdí rozšiřovaní komponentově orientovaného přístupu především z pohledu využívání komponent třetích stran.

4.2 EFP a současný svět softwarového vývoje

Zaměříme-li se na samotné mimofunkční požadavky, zjistíme, že neexistuje jednotný prohled na tuto problematiku. Glinz [GLI07] poukazuje na 3 hlavní problémy týkající se mimofunkčních požadavků:

1) Problém definice

> Vzhledem k různému chápání pojmu „mimofunkční požadavek“ a značné různorodosti kontextů, ve kterých lze tento pojem použít, existuje mnoho odlišných definic, přičemž snaha o obecné vyjádření často vede k natolik vágnímu popisu, že může znamenat prakticky cokoliv.

2) Problém klasifikace

> Existuje mnoho přístupů, jak klasifikovat mimofunkční požadavky.

3) Problém reprezentace

> Jakým způsobem a kam zdokumentovat mimofunkční požadavky.

ad 1) Pro naše účely je definice mimofunkční charakteristiky převzata z [JEZ12]:

> *Mimofunkční charakteristika je atribut nesoucí informaci, explicitně poskytovanou softwarovým systémem, za účelem popisu charakteristiky systému bez ohledu na skutečnou funkci systému, rozšířením kontraktu systému podporovanými technickými prostředky.*

ad 2) Klasifikace převzatá z [JEZ12]:

- Výkonnostní požadavky – rychlost, doba odezvy, paměťové nároky a další.
- Uživatelské požadavky – podpora, pravidelné aktualizace, cena a další.
- Chování – synchronizace, souběžný přístup a další.

ad 3) Problematikou reprezentace mimofunkčních požadavků se na Katedře informatiky a výpočetní techniky ZČU v Plzni zabývají projekty, jež jsou blíže rozepsány na následujících řádcích. Jedná se o *Extra-Functional Property Featured Compatibility Checks (EFFCC)* a *Component Repository supporting Compatibility Evaluation (CRCE).*

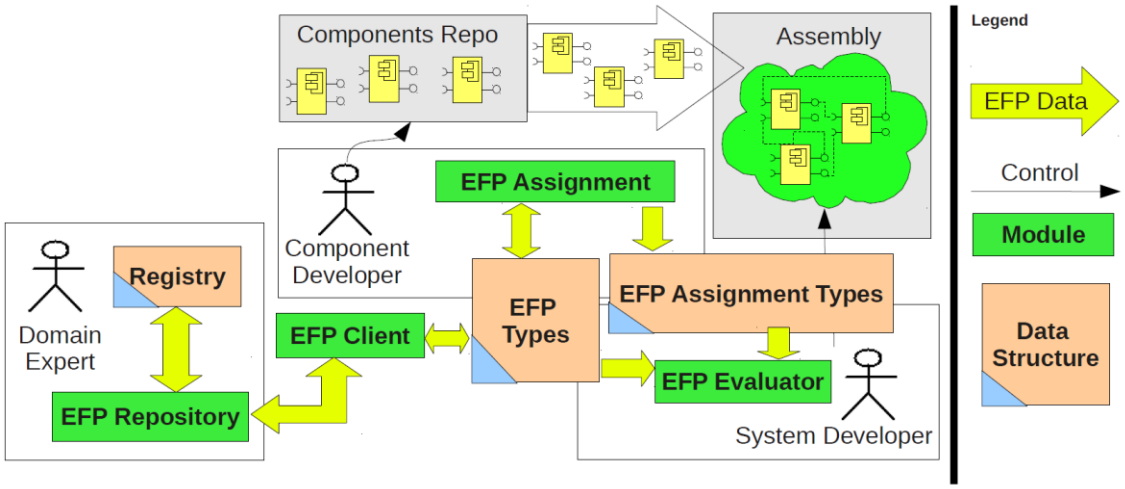
4.3 Nezávislý EFP mechanismus

Existuje mnoho specializovaných jazyků a komponentových modelů, které definují, jak má používání mimofunkčních požadavků vypadat. Avšak tyto specializované přístupy mají nevýhodu danou problematickou integrací s široce používanými komponentovými frameworky. Z tohoto důvodu vznikla myšlenka zaměřit se místo vymýšlení nových

specializovaných přístupů na navržení nezávislého EFP mechanismu, který bude umožňovat integraci s používanými frameworky. Tímto se pod názvem EFFCC detailně zabývá [JEZ12], zde jsou jen stručně popsány základní myšlenky navrženého a implementovaného řešení.

4.3.1 EFFCC

Extra-Functional Property Featured Compatibility Checks (EFFCC) představuje komplexní mechanismus pro komponentně orientovaný přístup vývoje software s podporou pro mimofunkční požadavky. Celý mechanismus je znázorněn na obrázku 4.1.



Obr. 4.1: EFFCC mechanismus (zdroj: [JEZ12])

Celý proces lze rozdělit na několik kroků. Doménový expert nebo architekt nejprve definuje mimofunkční charakteristiky, které mohou mít široké použití napříč spektrem komponentových aplikací. Vývojář konkrétní komponenty následně vybere konkrétní mimofunkční charakteristiky vhodné pro danou komponentu a určí jejich hodnoty. Vývojář, jenž skládá systém z komponent, využije funkci aplikačního assembleru, který verifikuje kompatibilitu použitých komponent [JEZ12].

Pro výše uvedené kroky poskytuje EFFCC tři hlavní moduly [JEZ12].

EFP Repository

- Úložiště pro definice mimofunkčních charakteristik.
- Přistupují k němu ostatní moduly za účelem získání, vytvoření nebo modifikace charakteristik.
- Implementováno jako třívrstvý webový server přístupný přes webové stránky (Spring MVC) nebo přes webové služby (SOAP).

EFP Assignment

- Zajišťuje propojení definovaných mimofunkčních charakteristik s konkrétní komponentou.
- Implementován jako Java modul, který ukládá vybrané mimofunkční charakteristiky přímo do příslušné komponenty i do společného úložiště na serveru.

EFP Evaluator

- Zajišťuje verifikaci kompatibility komponent obohacených o mimofunkční charakteristiky.
- Implementovaný modul vytvoří graf svázaných komponent a provede vyhodnocení kompatibility z pohledu mimofunkčních požadavků.

4.3.2 Univerzální EFP Repository

Základním předpokladem pro porovnávání komponent z hlediska mimofunkčních charakteristik je použití porovnatelných charakteristik. Použijí-li dodavatelé komponent při popisu komponenty různé definice charakteristik, případně různé jednotky pro určené hodnoty, je následné porovnání komplikované či nemožné. Řešením je používání unifikovaných charakteristik, k čemuž může dopomoci centrální úložiště obsahující všechny dostupné charakteristiky. V případě EFFCC má tuto úlohu *EFP Repository* [JEZ12].

4.4 CRCE

Component Repository supporting Compatibility Evaluation (CRCE) je pokročilé úložiště určené především pro OSGi bundly, které je v současnosti předmět vývoje na Katedře informatiky a výpočetní techniky ZČU v Plzni. Úložiště vychází z návrhu a implementace EFFCC (viz kapitola 4.3.1) a je vyvinuto jako komponentová OSGi aplikace. Hlavní vlastností úložiště je popis vkládaných komponent tzv. *popisnými metadaty*. Ta obsahují veškeré dostupné informace popisující komponentu a její vlastnosti i požadavky, jež mohou být použity pro kontrolu kompatibility. Druhou důležitou vlastností je možnost spouštět nad uloženými komponentami různé druhy testů (např. pro kontrolu kompatibility) a výsledky těchto testů rovněž uchovávat [KUC11].

Hlavní funkcionalita

- uložení komponenty do úložiště,
- poskytnutí seznamu uložených komponent,

- stažení komponenty z úložiště,
- testování kompatibility komponent,
- poskytnutí metadat dané komponenty.

Zamýšlenými uživateli úložiště jsou vývojář komponent (vkládá komponenty do úložiště, případně nad nimi spouští různé testy kompatibility), vývojář komponentových aplikací (vyhledává a stahuje vhodné komponenty) a cílové zařízení (automaticky kontroluje, zda neexistuje nová verze komponenty) [KUC11].

4.4.1 Uložení EFP

Komponenta vkládaná do úložiště již má mimofunkční charakteristiky uloženy v souboru `MANIFEST.MF` a v XML souboru `repository.xml`, který je umístěn ve složce `META-INF`. K tomuto spřažení slouží modul *EFP Assignment* z EFFCC (viz kapitola 4.3.1).

Po vložení komponenty do CRCE úložiště se provede indexování EFP vlastností do OBR formátu (viz 4.4.2). V tomto formátu jsou metadata v CRCE uložena jak v hromadném souboru `repository.xml`, tak i v individuálním souboru pro každou komponentu [RUZ12].

4.4.2 Formát metadat – OBR

OSGi Bundle Repository (OBR) je specifikace[1] popisující ukládání OSGi bundlů do úložiště. Součástí je popis XML souboru, který reprezentuje metadata bundlu. Cílem je poskytnout obecný model pro popis závislostí mezi komponentami[2]. Nejdůležitějšími částmi souboru jsou entity *capability* a *require*.

Entita *capability* slouží pro popis schopností, kterými daná komponenta disponuje. Má jméno a atributy (*property*), které k dané položce náleží.

Entita *require* slouží pro popis požadavků dané komponenty.

Tyto entity jsou běžně využívány k popisu balíků, které daný bundle exportuje/importuje. V našem případě je lze využít k definici mimofunkčních charakteristik. EFP, které daná komponenta nabízí (zaručuje), jsou zapsány jako capability, zatímco EFP vyžadované komponentou jsou zapsány jako requirement.

Výběrem formátu pro přepis se zabývá práce [RUZ12]. Pro každou definovanou mimofunkční charakteristiku existuje element `capability`, který může obsahovat

[1] Jedná se o specifikaci OSGi RFC 112, která se ovšem v době psaní práce nenacházela na webových stránkách OSGi. Základní popis lze nalézt na [OBR14].

[2] V terminologii OBR se pro označení komponenty (bundlu) používá termín *resource*.

elementy `attribute` a `property` (s dalšími `attribute` elementy). Jednotlivé `attribute` elementy slouží ke kompletnímu popisu vlastností prostřednictvím dvojice atributů `name` a `value`. Ukázka kódu 4.1 nastiňuje, jak může vypadat formát definice EFP v OBR souboru.

```
<capability namespace="some.namespace">
   <attribute name="API_method" value="service.method" />
      <property namespace='crce.metric'>
         <attribute name="name" value="efp_name" />
         <attribute name="value" value="value" />
         <attribute name="type" value="value_type" />
         <attribute name="unit" value="value_unit" />
   </property>
</capability>
```

Ukázka kódu 4.1: EFP přepsané do OBR souboru – capability

V případě `requirement` entity jsou jednotlivé vlastnosti zaznamenány v položkách atributu `filter`, přičemž jednotlivé položky jsou ve vztahu logického součinu. Z ukázky 4.2 je patrná nevýhoda v podobě nízké přehlednosti pro člověka.

```
<require name='EFP'
         filter='(&amp;(name=efp_name) (value&lt;=750)
                  (type=value_type) (unit=value_unit))'
         extend='false' multiple='false' optional='false'>
</require>
```

Ukázka kódu 4.2: EFP přepsané do OBR souboru – require

4.4.3 Současný stav

CRCE úložiště je současné době předmětem vývoje a není nasazené v produkčním prostředí. Po dokončení by úložiště mělo umožňovat ovládání prostřednictvím webového rozhraní a zároveň by mělo poskytovat webové služby, které by například umožňovaly vrátit pro danou komponentu soubor s metadaty.

5 ANALÝZA

5.1 Specifikace požadavků

5.1.1 Hlavní účel aplikace

Cílem je poskytnout uživatelům *SimCo frameworku* (viz kapitola 3) nástroj, který umožní rychlé a jednoduché vytváření a spravování *SimCo aplikací*. Pro připomenutí uveďme, že *SimCo framewor*k slouží k testování softwarových komponent (OSGi bundlů). Chce-li uživatel otestovat jednu nebo více komponent, musí nejprve vytvořit tzv. *SimCo aplikaci*. To představuje výběr a popis testovaných komponent, případně simulačních a intermediate komponent. Druhým krokem je vytvoření událostí, které představují volání metod komponent *SimCo aplikace*. Obojí (definice komponent i událostí) se nachází v XML souboru nazývaném scénář, jenž je vstupem *SimCo frameworku*. Tento soubor bylo nutné vytvářet manuálně, což obnáší psaní velkého množství režijního kódu a zjišťování potřebných vlastností komponent. Úkolem je tudíž vyvinout aplikaci, která bude prostřednictvím grafického uživatelského rozhraní uživateli umožňovat výběr komponent, definici událostí a následné vygenerování souboru scénáře. Výhodou bude rovněž automatizace některých činností spojených především s vytvářením událostí.

5.1.2 Požadovaná funkcionalita

V této podkapitole jsou uvedeny hlavní požadavky na funkcionalitu aplikace z pohledu uživatele. Slovní popis jednotlivých požadavků je doplněn o use-case diagram na obrázku 5.1.

Výběr komponenty

Aplikace umožní uživateli vybrat komponenty pro *SimCo aplikaci*. To kromě samotného výběru představuje i určení typu dané komponenty. Další akce již závisí na typu komponenty.

Vytvoření události

Po výběru komponenty bude uživatel moci vytvořit událost pro *SimCo framework*. Tu reprezentuje volání metody komponenty doplněné o další informace související mimo jiné s typem vytvářené události (například doba vyvolání události, opakování události atd.). Další možností je přidání mimofunkčních požadavků, jejichž splnění se testuje.

Generování scénáře

Aplikace umožní uživateli vygenerovat scénář pro *SimCo framework* z vybraných komponent a definovaných událostí. Scénář bude XML soubor a bude ve formátu, který vyžaduje *SimCo framework*.

Vytvoření konfiguračního souboru komponenty

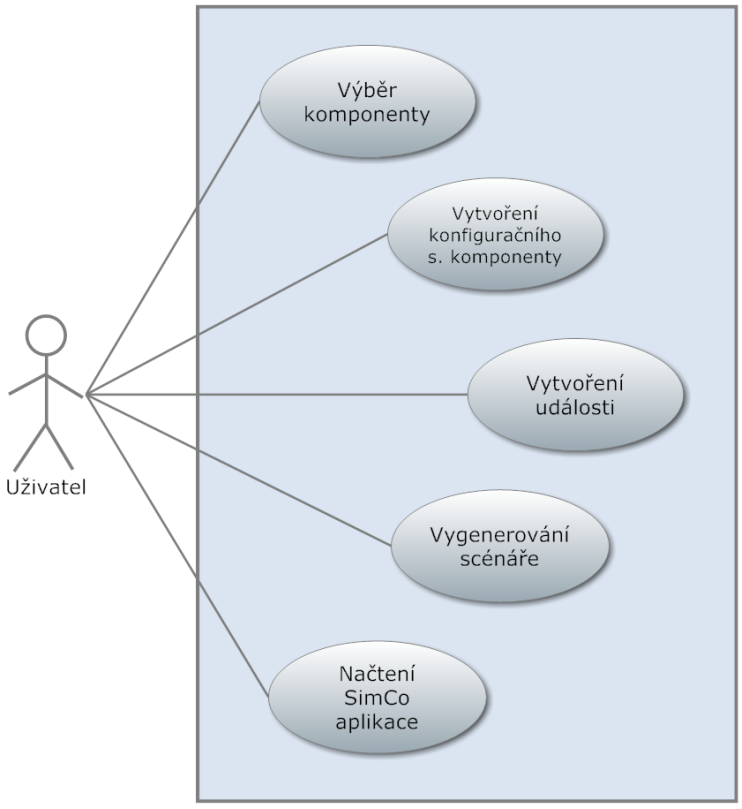
V *SimCo aplikaci* musí mít každá reálná a simulační komponenta konfigurační soubor. Aplikace umožní uživateli vytvořit základní kostru tohoto konfiguračního souboru tak, aby bylo možné načíst komponentu do *SimCo frameworku*.

Načtení SimCo aplikace

Aplikace umožní rovněž načítání dříve vygenerovaných scénářů, aby nebylo nutné při požadovaných úpravách vytvářet scénář od začátku.

Automatizovaná práce s EFP

Aplikace bude umět pracovat s mimofunkčními charakteristikami komponent, přesněji jednotlivých metod. Automaticky načte definované hodnoty mimofunkčních charakteristik testovaných komponent a tyto hodnoty nabídne uživateli při definování vlastností vytvářené události.



Obr. 5.1: Use-case diagram

Použitelnost

Aplikace bude mít grafické uživatelské rozhraní a bude uživateli umožňovat rychlé a intuitivní ovládání.

5.2 Typy komponent

SimCo framework při simulaci pracuje se třemi typy komponent, z nichž každá má svoje zvláštnosti. Vytvořená aplikace musí mezi těmito typy rozlišovat a umožnit nastavení všech potřebných parametrů. Níže jsou rozepsány jednotlivé typy a k nim náležící parametry.

REAL

Reálná komponenta provádí reálný výpočet, ke kterému byla vytvořena. Pro to, aby mohla být načtena *SimCo frameworkem*, musí mít vytvořený a nastavený konfigurační soubor. Generování základní kostry tohoto souboru je jedna z požadovaných funkcí na vyvíjenou aplikaci. Kromě toho může mít volitelně nastavenu intermediate komponentu. Ta se vkládá mezi dvě reálné komponenty, aby mohl *SimCo framework* zachytávat probíhající komunikaci.

SIMULATION

Simulační komponenta má za úkol simulovat činnost okolí testovaných komponent. Stejně jako reálná komponenta musí mít nastaven konfigurační soubor. Oproti reálným komponentám pro ni ovšem nemá smysl definovat intermediate komponentu.

INTERMEDIATE

Jak již bylo výše uvedeno, intermediate komponenta má funkci prostředníka (proxy) mezi dvěma reálnými komponentami. Kromě funkce přeposílání zpráv může danou komunikaci ovlivňovat z časového hlediska. Tím se rozumí časové zpoždění, které může simulovat například komunikaci přes síť.

Lze zadat tři typy zpoždění:

- `REGULAR`
  - Pravidelné zpoždění pevné délky.
  - Jeden číselný parametr
- `CASUAL_FIXED`
  - Nahodilé zpoždění s pevnou délkou. Oproti prvnímu typu se vyskytuje jen se zadanou pravděpodobností.
  - Dva číselné parametry.

- `CASUAL_GEN`
  - Nahodilé zpoždění s generovanou délkou i pravděpodobností výskytu.
  - Dva parametry reprezentované generátory náhodných čísel.

Poznámka: V *SimCo frameworku* se zpoždění `CASUAL_FIXED` i `CASUAL_GEN` označují shodně `CASUAL`. Liší se pouze typem parametrů.

5.3 Typy událostí

SimCo framework uživateli umožňuje definovat tři typy událostí simulace. Ty mají společné to, že jsou voláním metody nějaké komponenty *SimCo aplikace*. Z toho vyplývá, že je potřeba správně nastavit parametry této metody. Zároveň je možné událost doplnit o očekávanou návratovou hodnotu, aby mohl *SimCo framework* verifikovat správnost poskytovaných výsledků. Důležitá je také možnost zadat mimofunkční požadavky na volání dané metody, což simulátoru umožní ověřit kvalitu poskytovaných služeb.

Výše uvedené parametry události jsou společné pro všechny tři typy. Odlišnosti jsou ve způsobu, jakým se události vkládají do kalendáře diskrétní událostní simulace.

Události z pohledu simulace:

- `SINGULAR`
  - Jednorázová událost.
  - 1 parametr – simulační čas vykonání události.
- `REGULAR`
  - Opakovaná událost s pevnou velikostí periody opakování.
  - 3 parametry – čas prvního vyvolání události, čas ukončení vyvolávání události, čas mezi jednotlivými vyvoláními události.
- `CASUAL`
  - Opakování událostí s proměnnou velikostí periody opakování.
  - 3 parametry – čas prvního vyvolání události, čas ukončení vyvolávání události, identifikace generátoru náhodných čísel pro generování času mezi jednotlivými vyvoláními události.

5.4 Možnosti zadání parametrů

Aby nedošlo k nesprávné interpretaci, zdůrazněme, že parametrem je v této podkapitole chápán parametr metody[3].

[3] Nikoliv parametr události, jako jsou časy vykonání události či mimofunkční charakteristiky.

Jednotlivé parametry lze zadat následujícími způsoby:

- *Hodnotou*
  - Parametr má jednu pevně danou hodnotu. Při opakovaném vyvolání události v simulaci je vždy použita stejná hodnota.
- *Výčtem*
  - Je dán výčet hodnot. V simulaci se používá u opakovaných událostí typu `REGULAR` a `CASUAL`, kdy *SimCo framework* pro parametr metody postupně bere hodnoty z definovaného výčtu.
- *Vstupem ze souboru*
  - Je dán soubor, ze kterého *SimCo framework* načte konkrétní hodnoty pro parametr události. Dále se postupuje jako v předchozím případě.
- *Generátorem*
  - Parametr má přiřazen generátor, který při simulaci generuje konkrétní hodnoty parametrů. Obecně se může jednat o jakýkoliv generátor – nejen generátor náhodných čísel, ale například pro parametr typu String lze použít generátor řetězců.

S parametry úzce souvisí návratová hodnota metody. Ta se často mění se změnou hodnot parametrů metody. Z toho vyplývá, že návratové hodnoty lze zadávat ve srovnání s parametry analogicky – hodnotou, výčtem a vstupem ze souboru. Výjimkou je použití generátoru náhodných hodnot, které by v případě návratové hodnoty nemělo rozumné využití.

5.5 Získání informací o komponentách

Aby bylo možné vytvořit *SimCo aplikaci* (vybrat komponenty a definovat události), je nutné mít o každé komponentě (přesněji OSGi bundlu) následující informace:

1) jméno bundlu (základní identifikátor bundlu),
2) verze bundlu,
3) typ komponenty (z pohledu *SimCo frameworku* – viz 5.2),
4) poskytované služby (OSGi služby, které daný bundle poskytuje),
5) metody poskytovaných služeb,
6) typy parametrů metod,
7) typ návratové hodnoty metody.

Nabízejí se čtyři možné způsoby, jak získat potřebné údaje – ručním zadáním, přes OSGi API, ze zdrojových souborů, Java reflexí. V následujících podkapitolách je provedena analýza (jaké jsou možnosti, výhody a nevýhody) a výběr vhodného způsobu.

5.5.1 Ruční zadání

Ruční zadání je nevyhnutelné u bodu 3 – typ komponenty, neboť se jedná o údaj určený výhradně pro *SimCo framework*, který si sebou jednotlivé komponenty nenesou.

Pro zbylých šest bodů by ruční zadání kladlo vysoké nároky na uživatele, neboť by musel veškeré informace kompletovat z různých zdrojů – zdrojových souborů, konfiguračních souborů, případně přímo z kódu[4]. To by zbytečně komplikovalo používání aplikace, proto bylo rozhodnuto, že ruční zadání bude nutné pouze v případě typu komponenty. Veškeré ostatní údaje se budou určovat automaticky s minimálními nároky na uživatele.

5.5.2 OSGi API

Nejprve je nutné uvést důležitý předpoklad pro získání informací přes OSGi API. Zkoumaný bundle musí být nainstalován do stejného OSGi kontejneru jako aplikace, jejíž vytvoření je předmětem této práce. To je stejný OSGi kontejner, ve kterém je nainstalován *SimCo framework*. Tento předpoklad však není výrazným omezením, neboť daný bundle by stejně bylo nutné dříve nebo později[5] do tohoto kontejneru nainstalovat.

Každému bundlu v OSGi je při startu předána reference na `BundleContext`, který zprostředkovává interakci s OSGi Frameworkem. Poskytované API nabízí metody pro získání všech aktuálně nainstalovaných bundlů. Z těch lze získat jméno a verzi bundlu a poskytované služby bundlu v podobě reference na objekt poskytující službu (viz 2.2.5). Přestože je služba registrována přes rozhraní, OSGi neumožňuje získat informace o metodách (a jejich parametrech) v rozhraní.

Výhodou je přímočaré zjištění informací přes nabízené API. Nevýhoda spočívá v nutnosti mít již zkoumaný bundlu nainstalován v OSGi kontejneru, což ovšem není komplikací (viz 1. odstavec).

Tento způsob byl proto zvolen pro zjištění jména a verze bundlu a poskytovaných služeb.

5.5.3 Zdrojové soubory

Ze zdrojových a konfiguračních souborů lze získat veškeré potřebné informace. Konkrétně jméno a verze bundlu jsou definovány v konfigurační souboru OSGi (`MANIFEST.MF`). Náročnější je zjištění nabízených služeb – ty mohou být deklarovány

[4] V případě, kdy jsou služby registrovány přes API OSGi.

[5] Nejpozději před spuštěním simulace v *SimCo frameworku*.

v XML souboru, nebo registrovány přes OSGi API přímo ve zdrojovém kódu. Naproti tomu jména metod a typy parametrů a návratových hodnot lze získat snadno parsováním zdrojových souborů. Velkou výhodou je možnost získat nejen typy parametrů, ale i jejich jména. K tomu lze přidat i čtení *javadoc* komentářů parametrů i metod. Jména parametrů a komentáře metod a parametrů sice nejsou pro *SimCo aplikaci* potřebné, ale uživateli mohou významně ulehčit práci při vytváření událostí, neboť při použití neznámé komponenty nebude muset hledat informace v dokumentaci.

Podstatnou nevýhodou je ale nutnost mít k dispozici zdrojové soubory, což zejména v případě komponent třetích stran mnohdy není zaručeno.

5.5.4 Java reflexe

Reflexe je nástroj Javy, který umožňuje získat za běhu informace o prvcích třídy. Z našeho pohledu je důležité, že lze získat jména metod a typy parametrů a návratové hodnoty metod. Tyto informace poskytuje neměnná instance třídy `java.lang.Class`, kterou vytvoří JVM (*Java virtual machine*) pro každý objekt. Otázkou tedy je, jak získat objekt představující službu zkoumaného bundlu. Odpovědí je OSGi API a jeho metody pro získání služby, což představuje předání reference na objekt reprezentující danou službu. Malou nevýhodou je skutečnost, že reflexí nelze[6] získat jména parametrů. Ty ale nejsou pro definování událostí pro *SimCo framework* potřebné.

5.5.5 Shrnutí

Z výše uvedené analýzy vzešlo následující řešení: jméno a verze bundlu budou získány přes OSGi API společně s poskytovanými službami, z nichž budou následně reflexí získány jména metod a typy parametrů a návratových hodnot. Jediný údaj, který bude vyžadovat ruční zadání uživatele, bude typ komponenty. Zároveň bude aplikace umožňovat volitelně načíst další informace ze zdrojových souborů - jména metod a parametrů a k nim náležící *javadoc* komentáře.

5.6 Ukládání a načítání SimCo aplikace

Jedním s požadavků je možnost načíst dříve vytvořenou *SimCo aplikaci* (včetně definovaných událostí). Pro realizaci tohoto požadavku se nabízí dvě možnosti:

- načítání ze souboru scénáře,
- specifikovat nový formát souboru pro ukládání a následné načítání *SimCo aplikace*.

[6] Při defaultním nastavení překladače Javy.

5.6.1 Načítání ze scénáře

Výhodou by v tomto případě byla nepotřeba specifikovat a následně používat další soubor.

Významnou nevýhodou je ovšem malé množství informací o bundlech v tomto souboru. Ty by se po načtení musely znovu získat přes OSGi API a reflexí. Zároveň by nebylo možné uchovávat údaje doplněné uživatelem (např. poznámky k metodám a parametrům či informace načtené ze zdrojových souborů).

5.6.2 Vlastní soubor pro ukládání SimCo aplikace

Specifikace vlastního souboru by umožnila uložit a poté načíst veškeré potřebné údaje o bundlech i událostech.

5.6.3 Zvolené řešení

Z uvedených možností bylo vybráno vytvoření vlastního souboru. V něm budou definovány události simulace analogickým způsobem jako v souboru scénáře. Navíc bude obsahovat kompletní popis bundlů *SimCo aplikace* – ke každému bundlu seznam poskytovaných služeb, ke službám seznam metod a jejich parametrů a návratových hodnot, včetně poznámek uživatele.

5.7 Formát uložení dat

Vzhledem k tomu, že soubor scénáře je ve formátu XML, nabízí se použít stejné řešení i pro soubor s uloženou *SimCo aplikací*. Výhodou je především snadné strojové zpracování, nevýhoda plynoucí z bobtnání ukládaných dat pro nás není relevantní, jelikož se očekávají soubory o velikosti maximálně v řádu jednotek megabajtů[7].

Java obsahuje jako součást standardního JDK[8] několik technologií pro ukládání a čtení XML souborů. Mezi hlavní patří SAX, StAX, DOM a JAXB [HER07].

Důležitým aspektem při výběru z uvedených technologií je fakt, že pro běh vyvíjené aplikace nebude nejvhodnější uchovávat data v analogické struktuře, jako budou ukládána v XML souboru. Z toho důvodu není vhodným řešením použití DOM nebo JAXB, jejichž výhoda tkví právě ve vytváření objektové reprezentace XML souboru. V případě DOM se vytváří tzv. *Document Object Model*, v případě JAXB uchovávají data objekty tříd vygenerovaných z XSD k danému XML souboru. Při použití těchto

[7] Spíše však desítek až stovek kilobajtů.

[8] Všechny uvedené technologie jsou v Java Core API od JDK 1.6.

technologií by tudíž bylo stejně nutné provést přemapování načtených dat. Ze zbylých dvou alternativ umožňuje zápis do XML pouze StAX, a bude proto použit při generování XML souborů. A pro svou přímočarost bude použit rovněž při načítání XML souborů.

5.8 Práce s EFP

Jedním z účelů *SimCo frameworku* je testování komponent s cílem ověřit mimofunkční charakteristiky. Vyvíjená aplikace musí pochopitelně umožnit uživateli zadat ověřované mimofunkční charakteristiky. Současně je vhodné, aby bylo toto zadávání co nejvíce automatizované.

5.8.1 Měřitelné EFP

Problémem u mimofunkčních charakteristik je jejich měření a ověřování. Například v případě charakteristik vycházejících z uživatelských požadavků (použitelnost, podpora, atd.) je obtížné už určení metrik pro jejich popis. U výkonnostních požadavků (doba odezvy, paměťové nároky atd.) je určení metrik jednodušší, ale simulace ověřující konkrétní charakteristiky může být obtížně realizovatelná.

SimCo framework v současné době umožňuje měření jediné mimofunkční charakteristiky – doby odezvy. Je ale potřeba počítat s tím, že se v budoucnu mohou rozšířit možnosti *SimCo frameworku* o měření dalších charakteristik. Dá se ale předpokládat, že měřené charakteristiky budou vždy vázány přímo ke konkrétním metodám, nikoliv pouze ke komponentě jako celku.

5.8.2 Načítání EFP

Z předchozích odstavců vyplývá, že každá komponenta, přesněji její metody, může mít definované mimofunkční charakteristiky. Otázkou je, kde a v jakém formátu budou tyto charakteristiky uloženy a jak se budou načítat. Nabízejí se dvě možnosti uložení – do vlastních souborů (které vytvoří uživatel), nebo využití souborů s metadaty z CRCE.

Z vlastních souborů

Při specifikaci vlastního formátu souborů by uživatel musel testovaným komponentám přiřadit mimofunkční charakteristiky, což je hlavní nevýhoda tohoto přístupu. Na druhou stranu by tento proces stačilo vykonat pouze jednou, v případě dalších testování by již přiřazené charakteristiky existovaly. Výhodou je rovněž možnost specifikovat vlastní formát tak, aby odpovídal definicím mimofunkčních charakteristik v *SimCo frameworku*.

Z metadat poskytovaných CRCE

Druhou možností je využít již existující přiřazení mimofunkčních charakteristik ke komponentám. Pro tento účel slouží úložiště CRCE (viz kapitola 4.4), které pro uložené komponenty uchovává soubory s popisnými metadaty, ve kterých jsou kromě jiného i přiřazené mimofunkční charakteristiky.

Výhodou by byly menší nároky na uživatele, který by nemusel provádět přiřazování charakteristik ke komponentám.

Nevýhodou je nutnost provést mapování z formátu používaného v CRCE na popis EFP pro *SimCo framework*. Předpokladem je rovněž skutečnost, že se daná komponenta v CRCE úložišti nachází.

Shrnutí

I přes zmíněné nevýhody byl nakonec vybrán druhý způsob – načítání z metadat CRCE úložiště. Jedná se především o řešení s výhledem do budoucna, neboť v současné době není CRCE úložiště v provozu. Zároveň zatím neexistuje jasná specifikace formátu metadat a dotažené není ani programové vybavení, které tato metadata může zpracovávat.

Nicméně předpokládá se, že v budoucnu by *SimCo framework* měl sloužit právě pro ověřování mimofunkčních charakteristik definovaných pro komponenty v CRCE úložišti. To je hlavním důvodem, proč bylo zvoleno řešení s „napojením“ na CRCE. V současnosti ale bude nutné mít soubory s metadaty uložené na vlastním počítači. Poté, co se do CRCE doimplementují webové služby na poskytování metadat k vybrané komponentě, bude možné upravit vytvořenou aplikaci, aby si sama automaticky vyhledávala mimofunkční charakteristiky k testovaným komponentám.

Z pohledu vyvíjené aplikace bude práce s mimofunkčními charakteristikami následující. Při výběru komponenty uživatelem se aplikace pokusí automaticky najít EFP ke komponentě, přesněji k metodám komponenty. V případě úspěchu tyto charakteristiky přiřadí ke komponentě. Uživatel pak bude moci prohlížet definované charakteristiky a při vytváření událostí bude mít možnost použít definované hodnoty těchto charakteristik.

6 IMPLEMENTACE

6.1 Architektura vyvíjené aplikace

6.1.1 Okolí aplikace

Jelikož má být vyvíjená aplikace podporou pro *SimCo framework*, bylo již při prvotní analýze rozhodnuto, že poběží ve stejném prostředí – v OSGi kontejneru rozšířeném o Spring DM. Tím je dáno, že se jedná o komponentovou aplikaci složenou z bundlů.

Výslednou aplikaci tvoří 2 bundly:

- SimcoAppManager,
- EfpProvider.

EfpProvider zprostředkovává práci s mimofunkčními charakteristikami – načítá informace z metadat z CRCE. Dále poskytuje služby pro přiřazení načtených charakteristik ke komponentě a pro vizualizaci všech načtených mimofunkčních charakteristik. Blíže je tento bundle popsán v kapitole 6.9.

Zbytek kapitoly 6 pojednává o hlavní části aplikace – bundlu *SimcoAppManager*, který uživateli umožňuje vytvářet *SimCo aplikace* z komponent a událostí a generovat scénáře pro *SimCo framework*.

6.1.2 Konfigurace

Jako každý OSGi bundle má i *SimcoAppManager* konfigurační soubor `MANIFEST.MF` ve složce `META-INF`. Kromě klasických hlaviček, například pro nastavení jména a verze bundlu, je zde obsažena hlavička `Bundle-ClassPath`, pomocí níž jsou na *classpath* přidány použité knihovny třetích stran. Jedná se o:

- *qdox-1.9.jar* – pro parsování zdrojových souborů,
- *miglayout-4.0-swing.jar* – pokročilý layout manažer pro Swing.

Použití obou knihoven je popsáno v pozdějších kapitolách.

Zároveň je *SimcoAppManager* Spring DM bundlem, což znamená, že je potřeba druhý konfigurační soubor – pro Spring. Ten se nachází ve složce `META-INF/spring` a je v něm nakonfigurováno získávání a registrování služeb, aby se o tyto úkony postaral Spring. Hlavní část tohoto konfiguračního souboru je na ukázce kódu č. 6.1. Nejprve je pomocí elementu `reference` získána reference na službu poskytovanou *EfpProviderem*. V druhé části je vytvořena beana `CoreService` – ta je implementací

služby, kterou bude poskytovat *SimcoAppManager*. Přes konstruktor jsou injektovány reference na `bundleContext` (pro interakci s OSGi frameworkem) a na získanou službu *EfpProvidera*. Poslední částí je zaregistrování služby `CoreService` pod rozhraním `simco.framework.appmanager.ISimcoAppManager`. K tomu slouží element `service`.

```
<osgi:reference id="efpProvider"
      interface="simco.framework.efp.IEfpProvider" />

<bean id="CoreService"
      class="simco.framework.appmanager.core.SimcoAppManager">
      <constructor-arg ref="bundleContext" />
      <constructor-arg ref="efpProvider" />
</bean>

<osgi:service id="SimcoAppManagerOsgi" ref="CoreService"
      interface="simco.framework.appmanager.ISimcoAppManager" />
```

Ukázka kódu 6.1: Konfigurační soubor *SimcoAppManageru*

6.1.3 Poskytovaná služba

SimcoAppManager poskytuje v rámci OSGi službu registrovanou pod rozhraním `ISimcoAppManager`. Toto rozhraní obsahuje jednu metodu:

`startApp()`.

Zavoláním této metody se vytvoří grafické uživatelské rozhraní *SimcoAppManageru*.

Pro integraci do *SimCo frameworku*, přesněji do jeho grafického uživatelského rozhraní, tudíž stačí připsat do patřičného konfiguračního souboru získání služby `simco.framework.appmanager.ISimcoAppManager` a následně získanou referenci vložit do atributu patřičné třídy. Poté již stačí přidat nové tlačítko (položku menu) a v jeho `ActionListeneru` zavolat `startApp()` nad zmíněným atributem s referencí na získanou službu.

6.1.4 Rozdělení aplikace

Architektura aplikace *SimcoAppManager* vychází ze vzoru *Model-view-controller* – je rozdělena fyzicky do 9 balíků a logicky do 3 vrstev následujícím způsobem:

- *Datová vrstva* – veškeré třídy uchovávající data jsou obsaženy v balíku `simco.framework.appmanager.data` (hlavní třídy a jejich vztahy jsou blíže popsány v kapitole 6.2).

- *Prezentační vrstva* – třídy zajišťující interakci s uživatelem prostřednictvím GUI jsou v balíku `simco.framework.appmanager.gui`.
- *Aplikační vrstva* – všechny ostatní balíky obsahují třídy aplikační logiky nebo podpůrné třídy aplikace.

Stručný popis obsahu všech balíků je na následujících řádcích.

Seznam balíků aplikace

`simco.framework.appmanager.core`

Obsahuje jádro aplikace a třídu `SimcoAppManager`, která je implementací poskytované služby.

`simco.framework.appmanager.services`

V tomto balíku jsou důležité třídy aplikační logiky – pro získávání informací o komponentách, načítání projektů, validaci uložených dat a další.

`simco.framework.appmanager.gui`

Pro tvorbu grafického uživatelského rozhraní byla použita knihovna *Swing*. Důvodem je skutečnost, že samotný *SimCo framework* má implementováno GUI právě ve Swingu. V tomto balíku se nachází komponenty pro GUI – okna, panely a dialogy.

Podrobně jsou třídy z tohoto balíku popsány v kapitole 6.4.

`simco.framework.appmanager.data`

Obsahuje datové třídy pro uchovávání informací o komponentách, událostech a pomocné "obálky" pro informační i chybové zprávy.

Hlavní datové třídy a jejich vztahy jsou popsány v kapitole 6.2.

`simco.framework.appmanager.control`

V tomto balíku jsou třídy pro obsluhy událostí uživatelského rozhraní – implementace *listenerů* pro komponenty Swingu.

`simco.framework.appmanager.io`

V tomto balíku jsou třídy zajišťující vstupy a výstupy. To obnáší ukládání a načítání dat z XML souborů.

`simco.framework.appmanager.types`

Obsahuje výčtové typy pro prvky *SimCo aplikace* (typy komponent, typy událostí) i pro typy zpráv používaných v *SimcoAppManageru* mezi *observer* a *observable* třídami.

simco.framework.appmanager.utils

V tomto balíku jsou pomocné třídy pro logování, lokalizaci a konfiguraci aplikace.

simco.framework.appmanager.excpetions

Tento balík obsahuje vlastní implementace výjimek včetně zpráv používaných pro informování o nastalých chybových stavech v aplikaci.

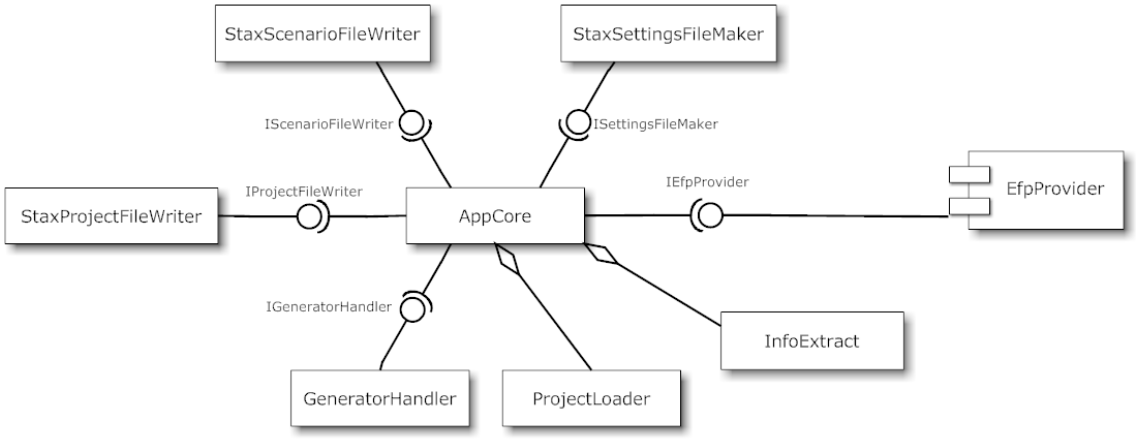
AppCore

Jádrem *SimcoAppManageru* je třída `AppCore`. Ta je implementována jako *singleton* s privátním konstruktorem, z čehož plyne, že existuje jediná instance, která se vytvoří při zavedení třídy. Referenci na jádro pak lze získat prostřednictvím statické metody `getInstance()`.

Vzhledem k tomu, že se jedná o centrální bod celé aplikace, jsou zde reference na objekty tříd aplikační logiky. Zejména se jedná o:

- `infoExtract` – získává informací o komponentách.
- `projectLoader` – načítá projekty (*SimCo aplikace*).
- `projectFileWriter` – ukládá projekty do souboru.
- `scenarioExporter` – generuje scénáře pro *SimCo framework*.
- `generatorHandler` – zprostředkovává práci s generátory.

Dále obsahuje referenci na službu poskytovanou druhým bundlem – *EfpProviderem*. Na obrázku 6.1 je diagram zobrazující výše popsané skutečnosti.



Obr. 6.1: Diagram tříd aplikační logiky

Zároveň se uchovává reference na instanci třídy `ProjectComp`, která představuje vstupní bod datového modelu.

Hlavní metodou jádra (a zároveň jedinou metodou nepřímo dostupnou mimo bundle) je `startApp()`. Její hlavní činností je vytvoření grafického uživatelského rozhraní (`Gui`) a jejího *controlleru* (`GuiController`), který obsahuje obsluhy událostí vyvolaných uživatelem.

Pro úplnost uveďme, že druhou třídou v balíku `core` je `SimcoAppManager` implementující rozhraní `ISimcoAppManager`, z čehož plyne, že je zároveň implementací služby poskytované celým bundlem. Tato třída v konstruktoru získá referenci na jádro, jemuž následně předá Springem injektované reference na `BundleContext` a `EfpProvider`. Jediná metoda služby – `startApp()` – volá stejnojmennou metodu jádra, která byla popsána v předchozím odstavci.

6.2 Datové modely

V této kapitole jsou popsány datové struktury – třídy sloužící pro uchovávání dat v *SimcoAppManageru*. Z analýzy v kapitole 5 je zřejmé, že je potřeba udržovat informace především o *SimCo aplikaci* – komponentách a událostech. Mimo to je nutné vytvořit datové třídy pro mimofunkční charakteristiky, které lze připojit k definované události, a pro generátory, které je možné použít pro generování hodnot různých parametrů. Uvedené třídy jsou v balíku `simco.framework.appmanager.data`.

6.2.1 Projekt

Hierarchicky nejvýše stojí třída `ProjectComp`. Ta obsahuje veškeré náležitosti *SimCo aplikace*:

- seznam všech komponent,
- seznam všech událostí pro *SimCo framework*.

Kromě toho jsou zde atributy týkající se nastavení simulátoru, které je možné ukládat do souboru scénáře. Jedná se o délku simulačního kroku a adresář pro ukládání výsledků simulace.

6.2.2 Komponenty

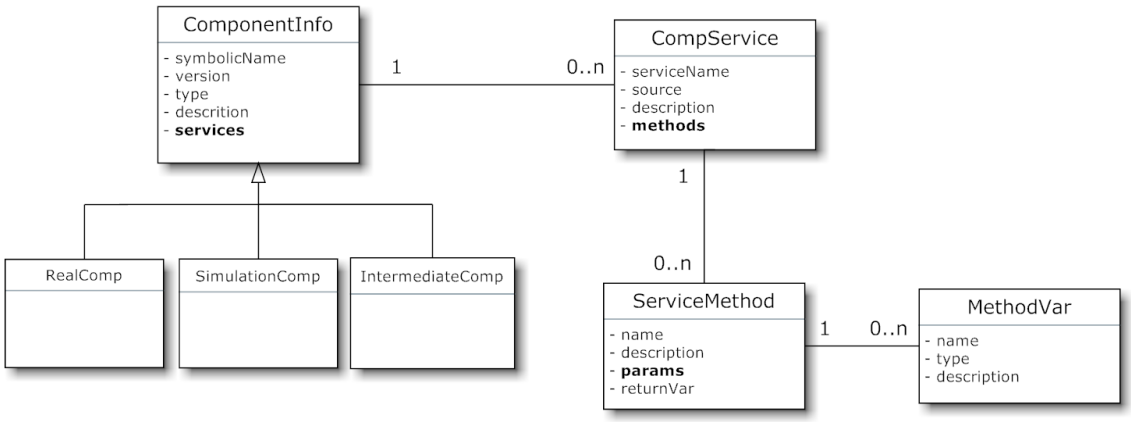
Stěžejní částí *SimCo aplikace* jsou komponenty, v prostředí OSGi označované termínem bundle. Pro uložení všech potřebných informací (popsaných v kapitole 5.5) slouží celkem 7 tříd, jejichž vzájemný vztah je pro lepší pochopení znázorněn na obrázku 6.2.

Základem je třída `ComponentInfo` reprezentující jeden bundle. Tomu odpovídají obsažené atributy – symbolické jméno a verze bundlu a především seznam poskytovaných služeb (`services`). Každý bundle může poskytovat *0-n* služeb.

Z pohledu *SimCo frameworku* je důležitý atribut `type`, který určuje, zda se jedná o reálnou, simulační, nebo intermediate komponentu. Od třídy `ComponenInfo` jsou zděděny třídy `RealComp`, `SimulationComp` a `IntermediateComp`, jež obsahují implementaci pro specifika daného typu komponenty. Například v případě `IntermediateComp` se jedná o možnost zadat zpoždění při předávání zpráv mezi reálnými komponentami.

Jednotlivé služby jsou popsány třídou `CompService`. Atributy vycházejí ze skutečnosti, že v OSGi je službou objekt, který je zaregistrován prostřednictvím svého rozhraní. Jméno rozhraní (a zároveň jméno služby) je dáno atributem `serviceName` a seznam poskytovaných metod atributem `methods`.

Pro popis metody slouží třída `ServiceMethod` se jménem metody, seznamem parametrů a návratovou hodnotou. Pro definice jednotlivých parametrů i návratové hodnoty je použita třída `MethodVar`.



Obr. 6.2: Diagram tříd komponenty

6.2.3 Události

Druhou důležitou částí *SimCo aplikace* jsou události. Ty jsou reprezentovány třídou `EventInfo`. Základním údajem je typ události (`REGULAR` / `SINGULAR` / `CASUAL`) v atributu `type`. Podle specifik daného typu události jsou vyplněny atributy pro první vyvolání, konec a periodu vyvolávání události. Důležitým atributem je seznam parametrů volané metody. Každý parametr je definován třídou `EventParameter`, přičemž hodnota parametru může být zadána jednou hodnotou, výčtem, vstupem ze souboru nebo generátorem (viz kapitola 5.4). Třída `EventInfo` může dále obsahovat očekávanou návratovou hodnotu metody a seznam mimofunkčních požadavků pro verifikaci.

Vzhledem k požadavku řadit události ve výpisu podle času prvního vyvolání, implementuje třída `EventInfo` rozhraní `Comparable`. Z toho plyne, že obsahuje implementaci metody `compareTo()`, což umožní jednoduché řazení voláním metody `sort()` nad kolekcí obsahující seznam událostí.

6.2.4 EFP

Jeden mimofunkční požadavek je představován třídou `EfpParameter`, která obsahuje definici požadavku (jméno, jednotka, datový typ hodnoty) a přidělenou hodnotu. Výše popsaná událost typu `EventInfo` má seznam těchto požadavků o velikosti *0-n*.

6.2.5 Generátory

Již v několika případech bylo zmíněno, že pro určení konkrétních hodnot v průběhu simulace může sloužit generátor. Jedná se o tyto případy:

- hodnota parametru metody události (obecně jakýkoliv generátor),
- perioda vyvolávání události typu `CASUAL` (generátor náhodných čísel),
- délka a pravděpodobnost výskytu zpoždění `INTERMEDIATE` komponenty (generátor náhodných čísel).

Obecná reprezentace takového generátoru je dána třídou `Generator`. Každý generátor má své označení, atribut `seed` (pro inicializaci generátoru), určení typu generovaných hodnot a obecně libovolné množství parametrů (`GeneratorParameter`) generátoru ve tvaru *klíč-hodnota*.

6.3 Získání informací o komponentách

Chce-li uživatel přidat do *SimCo aplikace* novou komponentu, je úkolem *SimcoAppManageru* poskytnout seznam bundlů nainstalovaných v OSGi a po výběru konkrétního bundlu automaticky získat všechny potřebné informace a vytvořit datové struktury popsané v kapitole 6.2.2. Pro získání požadovaných údajů je použito OSGi API ve spojení s Java reflexí. Volitelně je možné pro podrobnější popis načíst navíc jména parametrů a `javadoc` komentáře ze zdrojových souborů. Podrobně jsou jednotlivé kroky rozepsány v následujících podkapitolách.

6.3.1 Využití OSGi API

Každému bundlu nainstalovanému do OSGi je v metodě `start()` třídy `BundleActivator` předána reference na `BundleContext`. Při použití Spring DM je možné využít Springem vytvořenou beanu `bundleContext` a prostřednictvím

Dependency injection ji vložit na požadované místo [SDM14]. `BundleContext` je rozhraní umožňující interakci s OSGi frameworkem a díky tomu lze například registrovat a získávat služby nebo získat seznam všech nainstalovaných bundlů v OSGi frameworku.

V našem případě je při výběru komponenty uživatelem nejprve volána metoda `getBundles()` třídy `BundleContext`, vracející pole referencí na objekty typu `Bundle`. Jméno bundlu lze získat jednoduše voláním `getSymbolicName()` nad konkrétním bundlem. Seznam všech takto získaných bundlů je zobrazen uživateli, který následně vybere požadovaný bundle. Ten je pro další zpracování předán metodě `createCompInfoFromBundle()` třídy `InfoExtract`.

V první fázi se získá verze bundlu. OSGi však neposkytuje[9] metodu, která by toto přímočaře umožňovala. Je potřeba využít skutečnosti, že verze bundlu je definována v souboru manifestu pod hlavičkou `Bundle-Version`. `Bundle` objekt pak poskytuje metodu `getHeaders()` vracející mapu, která umožňuje přes klíč (jméno hlavičky) získat hodnoty všech hlaviček ze souboru manifestu.

Dalším krokem je zjištění všech služeb poskytovaných vybraným bundlem. K tomu je nutné nejprve získat reference na služby typu `ServiceReference` voláním metody `getRegisteredServices()` nad `Bundle` objektem. Samotná reference na službu ale z našeho pohledu neposkytuje žádné užitečné informace. Je však důležitá jakožto parametr metody `getService()` třídy `BundleContext`, která již vrací přímo objekt implementující danou službu.

V tuto chvíli máme bundle, služby poskytované tímto bundlem a objekty implementující tyto služby. To jsou veškeré, z našeho pohledu, důležité informace, které lze získat přes OSGi API. Pro zjištění podrobností o metodách je využita Java reflexe.

6.3.2 Využití Java reflexe

Reflexe je mocný nástroj Javy, pomocí něhož je možné za běhu prozkoumávat třídy, rozhraní, atributy a metody. Navíc lze vytvářet nové instance zpracovávaných tříd, volat jejich metody nebo nastavovat hodnoty atributů.

Vstupním bodem pro všechny operace spojené s reflexí je objekt `java.lang.Class`. Máme-li k dispozici instanci nějaké třídy, lze voláním metody `getClass()` nad danou instancí získat požadovaný `Class` objekt popisující třídu, ze které je prozkoumávaná instance vytvořena.

[9] V použité verzi OSGi. Od verze 4.2 již rozhraní Bundle obsahuje metodu getVersion().

Pro začátek se hodí zrekapitulovat, co již máme a jaké informace ještě potřebujeme získat. V předchozím kroku jsme prostřednictvím OSGi API dostali:

- vybraný bundle v podobě `Bundle` objektu,
- služby poskytované bundlem v podobě pole `ServiceReference` objektů,
- objekty implementující dané služby.

Z uvedených objektů potřebujeme zjistit:

- jména metod jednotlivých služeb,
- typy parametrů a návratových hodnot daných metod.

Je potřeba si uvědomit, že není možné reflexí získat a zpracovat všechny metody z objektu implementujícího danou službu. Tento objekt totiž může obsahovat i metody, jež nepatří do rozhraní, přes které je objekt zaregistrován jako služba. Při tomto postupu by tudíž byly uživateli mylně (jako metody služby) nabízeny i metody, které nejsou z vnějšku bundlu dostupné.

Z toho důvodu se metody nezískávají přímo ze třídy objektu, jenž implementuje službu, ale z jeho rozhraní. Vzhledem k tomu, že třída může obecně implementovat libovolný počet rozhraní, je nejdříve nutné najít to rozhraní, přes které je registrována služba. To se provede v metodě `findServiceInterface()` třídy `InfoExtract`. Zde se nejprve získá `Class` objekt z objektu implementujícího službu a nad ním se zavolá metoda `getInterfaces()`, která vrátí opět `Class` objekty, tentokrát reprezentující rozhraní výchozího objektu. Následně se porovnávají jména získaných rozhraní se jménem rozhraní registrované služby – to je obsaženo v `ServiceReference` objektu získaném přes OSGi API. Odpovídající rozhraní je metodou vráceno v podobě `Class` objektu.

V dalším kroku se v metodě `getRuntimeInfo()` získá seznam metod voláním `getDeclaredMethods()` nad `Class` objektem rozhraní. Nad konkrétními metodami se pak zavolá `getGenericParameterTypes()` vracející typy parametrů a `getReturnType()` vracející typ návratové hodnoty.

Výsledkem výše popsaného procesu je kompletní datová struktura popsaná v kapitole 6.2.2 se všemi nezbytnými údaji pro vytváření událostí.

6.3.3 Načtení ze zdrojových souborů

Pokud bude mít uživatel k dispozici zdrojové soubory, může doplnit informace získané reflexí o jména parametrů a `javadoc` komentáře. To může ulehčit práci při vytváření událostí.

Pro parsování zdrojových souborů je použita open source knihovna *QDox* (šířená pod licencí Apache 2.0), která umožňuje získat definice tříd, rozhraní, metod a parametrů doplněné o `javadoc` komentáře [QDX14]. Bez jakýchkoliv úprav tak plně dostačuje

našim potřebám, a není proto nutné psát na zpracovávání zdrojových souborů vlastní kód.

Z pohledu *SimcoAppManageru* obstarává zpracování zdrojových souborů metoda `getSourceInfo()` třídy `InfoExtract`, jíž se jako parametr předá objekt typu `CompService`, do něhož se pomocí knihovny QDox doplní informace ze zdrojového souboru. Nejprve se vytvoří vstupní bod knihovny QDox – `JavaDocBuilder`. Jeho zpracováním se získá objekt typu `JavaClass`, což je reprezentace třídy při práci s knihovnou QDox. Jednotlivé metody se z tohoto objektu získají podle signatury voláním `getMethodBySignature()`. Výsledkem jsou `JavaMethod` objekty, z nichž lze získat voláním `getParameters()` seznam parametrů a následné volání `getName()` pro konkrétní položku vrátí jméno parametru.

`Javadoc` komentáře třídy, případně metody lze získat jednoduše voláním `getComment()` nad příslušným objektem. Komplikovanější je situace v případě parametrů, neboť QDox neumí podle jména svázat `javadoc` komentář s odpovídajícím parametrem. Je však možné nad objektem typu `JavaMethod` volat `getTagsByName("param")`, který vrací pole komentářů za značkou `@param`. Za touto značkou při správném zápisu následuje jméno a popis parametru. Získaný text je proto rozdělen na jméno a popis a podle jména je následně přidělen odpovídajícímu parametru.

6.4 Prezentační vrstva

Prezentační vrstva zajišťuje interakci s uživatelem a má podobu grafického uživatelského rozhraní, přičemž implementace využívá knihovnu Swing. Veškeré komponenty pro vizualizaci jsou v balíku `simco.framework.appmanager.gui`.

Pro zopakování uveďme hlavní funkce nabízené uživateli:

- Vytvoření SimCo aplikace.
  - Přidání komponenty – bundlu z OSGi kontejneru.
  - Vytvoření konfiguračního souboru pro bundle.
  - Vytvoření události na základě metody služby poskytované vybraným bundlem s možností přiřadit mimofunkční požadavky.
- Uložení *SimCo aplikace* (projektu) do souboru.
- Načtení *SimCo aplikace* ze souboru.
- Generování souboru scénáře.

V následujících podkapitolách je popsán návrh a implementace GUI, které poskytne uživateli výše popsanou funkcionalitu.

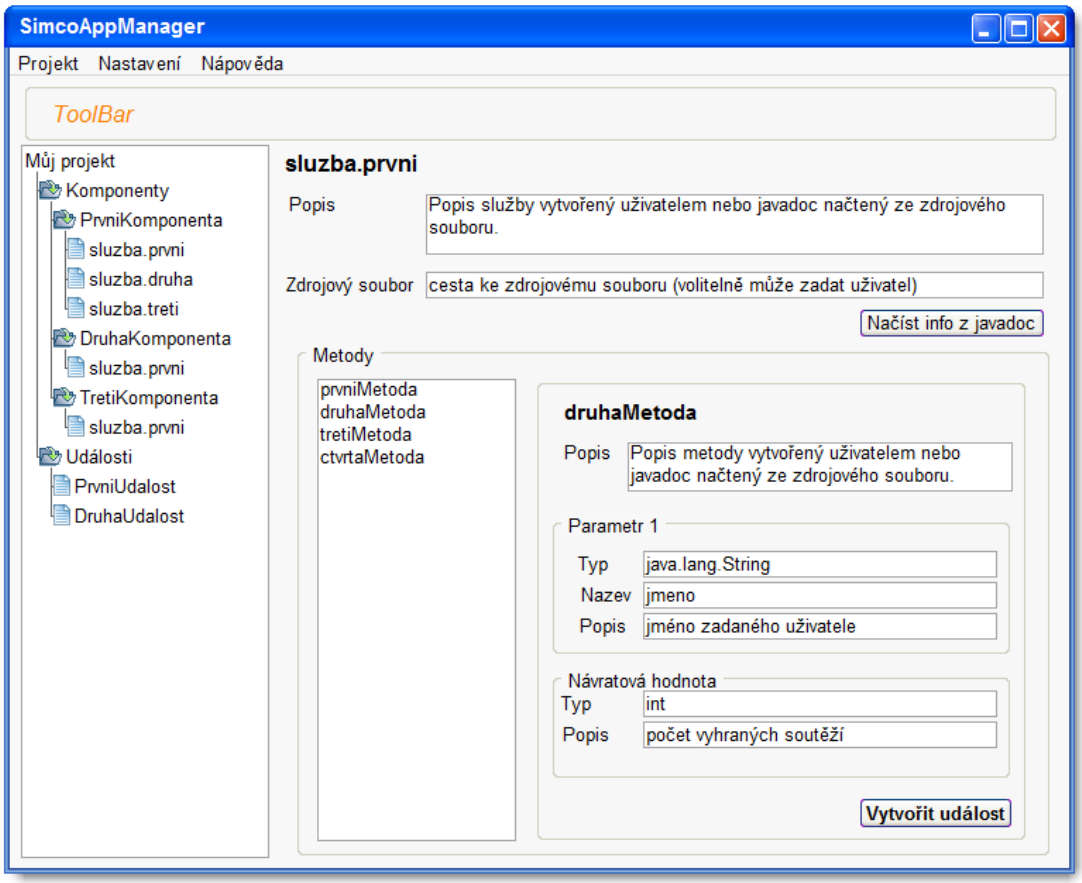
6.4.1 Návrh GUI

Prvním krokem je návrh GUI, což představuje výběr ovládacích prvků a rozvržení jednotlivých komponent.

V kapitole 6.2 jsou popsány datové struktury pro projekt, komponenty a události. Z uvedeného popisu je patrné, že se jedná o stromovou strukturu. Hlavním ovládacím prvkem aplikace byl proto zvolen strom, jehož kořenovým uzlem je projekt (*SimCo aplikace*) s dvěma následníky – seznamem komponent a událostí. Jednotlivé komponenty (bundly) mají jako následníky registrované služby, které představují objekty s metodami.

Uživateli se po výběru uzlu stromu (projekt, komponenta, služba, událost) zobrazí panel s informacemi závislými na vybraném objektu s možností úpravy editovatelných údajů. Navíc může provádět další úkony spojené s vybraným objektem – např. v případě služby lze vytvořit událost.

Návrh GUI byl vytvořen pro situaci, kdy je z prvků stromu vybrána služba. To znamená, že je zobrazen panel poskytující veškeré informace o službě včetně všech metod a jejich parametrů. Pro návrh vzhledu GUI byl použit nástroj *MockupScreens* [MUS14], přičemž výstup je na obrázku 6.3.



Obr. 6.3: Návrh GUI

6.4.2 Layout manager

Vzhledem k faktu, že uživatelské rozhraní obsahuje větší počet oken/panelů, v nichž jsou netriviálně rozmístěny další komponenty, bylo potřeba vhodně zvolit tzv. *layout manažer(y)*. Úkolem layout manažera je správné rozmístění grafických komponent, jejich ukotvení a změna velikosti v závislosti na změně velikosti nadřazeného kontejneru. Knihovna Swing poskytuje několik základních layout manažerů, nicméně možnosti jejich nastavení jsou velmi omezené. Například rozvržení komponent podle obrázku 8 by vyžadovalo kombinaci nemalého množství standardních layout manažerů. To by bylo komplikované pro implementaci, navíc by to mělo negativní vliv na přehlednost kódu.

Z uvedených důvodů byla pro zajištění funkce layout manažerů zvolena open source knihovna *MigLayout* (dostupná pod licencí BSD) [MIG14]. Jedná se o pokročilý a flexibilní layout manažer s širokými možnostmi nastavení, pomocí nichž lze jednoduše definovat rozmístění komponent a jejich reakce na změnu velikosti okna.

6.4.3 Hlavní okno

Vytvoření hlavního okna aplikace zajišťuje třída `Gui`. V konstruktoru jsou volány metody, které inicializují *menu* a *toolbar* a rozdělují zbylý obsah okna na dvě části použitím `JSplitPane` :

- *Levá část* – panel se stromem (`JTree`) reprezentující *SimCo aplikaci*.
- *Pravá část* – panel `infoP` pro zobrazení informací týkající se vybrané položky stromu.

Vytvořenému stromu je přidán *ActionListener*, jenž podle vybrané položky (uzlu) stromu vytvoří odpovídající panel (viz 6.4.5) a ten následně zobrazí v pravé části okna na informačním panelu `infoP`. Zároveň vytvoří odpovídající *controller* s obsluhami událostí pro daný panel.

Jelikož `Gui` implementuje rozhraní `Observer`, může být její instance registrována jako posluchač objektu zděděného od třídy `Observable`. V tomto případě jsou to datové třídy, které tímto způsobem informují o změně datového modelu. `Gui` na tyto změny reaguje v metodě `update()`, jejíž parametrem je „messenger“ `ProjChangeMessage` s dvěma atributy – typ zprávy (přidání/odstranění/změna komponenty nebo události) a objekt (komponenta/událost), jehož se změna týká. Podle uvedených atributů se provedou změny v modelu stromu s následným překreslením.

Ve třídě `ProjectTreeCellRenderer` je upravená implementace defaultního `TreeCellRendereru`, jehož úkolem je vykreslení stromu. Úprava se týká nastavení ikon pro uzly v závislosti na objektech, které reprezentují.

Důležité pro ovládání aplikace jsou rovněž *toolbar* a *menu* – obsahují tlačítka pro:

- vytvoření nového projektu,
- přidání komponenty do projektu,
- uložení projektu,
- načtení projektu,
- export do scénáře.

Výkonný kód reagující na stisk jednotlivých tlačítek obsahuje třída `GuiController` v balíku simco.`framework.appmanager.control`.

6.4.4 Přidání komponenty

Jednou z událostí obsluhovaných ve třídě `GuiController` je přidání komponenty do projektu – metoda `addComp_AP()`. Zde je prostřednictvím jádra získán seznam všech bundlů nainstalovaných v OSGi a následně je vytvořen dialog, jehož obsahem je panel `AddCompPanel` zobrazující seznam získaných bundlů.

Pro úplnost uveďme[10], že poté, co uživatel vybere konkrétní bundle a určí jeho typ, se volá metoda jádra, která prostřednictvím třídy `InfoExtract` zjistí veškeré informace o bundlu (viz kapitola 6.3), přičemž výsledkem jsou třídy datového modelu reprezentující komponentu *SimCo aplikace*.

6.4.5 Panely pro jednotlivé objekty

V kapitole 6.4.3 bylo uvedeno, že při výběru položky stromu se v pravé části okna zobrazí panel týkající se objektu reprezentovaného touto položkou. Popis panelů jednotlivých objektů je v této kapitole.

Projekt

Panel `ProjectPanel` uživateli umožňuje prostřednictvím dvou textových polí nastavit volitelné parametry pro *SimCo framework*.

Ve spodní polovině panelu je tabulka obsahující přehled všech definovaných událostí a jejich hlavních parametrů. Podle těchto parametrů lze seznam událostí v tabulce řadit. Problémem je skutečnost, že některé parametry mohou nabývat různých datových typů – např. perioda je pro `SINGULAR` a `REGULAR` události celým číslem, zatímco pro `CAUSAL` událost je reprezentována generátorem. Proto je implementována třída `MixedComparator`, umožňující řazení hodnot různých datových typů – seřadí to, co seřadit lze (co lze převést na číslo), zbytek zařadí na konec seznamu.

[10] I když se to netýká přímo prezentační vrstvy.

Komponenta

Již bylo zmíněno, že existují 3 typy komponent odlišující se svými atributy. Podle toho vypadá obsah panelu `CompPanel`. Např. pro intermediate komponentu lze nastavit realizované zpoždění zadáním hodnot nebo generátorů. V případě reálné a simulační komponenty je možné vytvořit kostru konfiguračního souboru (viz kapitola 6.7), který je nutný pro vygenerování scénáře.

Služba

Návrh panelu služby je vidět již na obrázku 8 v pravé části okna. Jeho implementaci představuje třída `ServicePanel`. Ta je rozdělena na 3 části:

- Horní – textová pole pro popis služby a cestu ke zdrojovému souboru.
- Levá dolní – seznam s metodami služby.
- Pravá dolní – podle metody vybrané ze seznamu je zobrazen panel s popisem parametrů a návratové hodnoty.

Obsluha tlačítek na panelu je definována třídou `ServiceController`. Důležité je především tlačítko pro vytvoření nové události, jehož obsluhu představuje metoda `createEvent_AP()`. V té se vytvoří nová instance třídy `AddEvent` – okno s panelem `EventEditPanel`, které umožní zadat parametry události.
Opět pro doplnění uveďme, že zpracování zadaných parametrů události provádí `AddEventController`. Ten nejprve provede s využitím třídy `EventValidator` validaci zadaných hodnot a následně (při úspěšné validaci) vytvoří novou událost a tu uloží do datového modelu.

Událost

Informace o události jsou na panelu `EventPanel`. V horní části jsou zobrazeny základní needitovatelné informace – zdrojová komponenta a metoda, typ události. Pod nimi je vložen `EventEditPanel` umožňující editaci parametrů události. Validaci a uložení provedených změn zajišťuje `EditEventController`.

6.4.6 Vytvoření a editace události

Na předchozích řádcích byl zmíněn `EventEditPanel`. Ten umožňuje zadat nebo editovat veškeré parametry týkající se události, přičemž je rozdělen na 5 částí:

- určení typu události,
- zadání parametrů události (čas prvního volání, perioda volání atd.),
- zadání hodnot parametrů metody události,
- možnost zadat očekávanou návratovou hodnotu,
- možnost připojit mimofunkční požadavky.

Hodnoty parametrů metody a očekávanou návratovou hodnotu metody lze zadat různými způsoby. Podle vybraného způsobu se zobrazí `JTextField` (pro jednu hodnotu nebo vstup ze souboru), `JTextArea` (pro výčet hodnot), či `GeneratorPanel` (pro zadání generátorem).

6.4.7 Panel pro generátor

Generátory mohou být použity pro definice různých parametrů (parametr metody, perioda události simulace atd.). Vyskytují se tudíž na různých místech aplikace, a proto byla vytvořena nová grafická komponenta `GeneratorPanel` reprezentující generátor.

Tato komponenta obsahuje seznam dostupných generátorů[11] a textová pole pro zadání parametrů, která se překreslují podle vybraného generátoru.

Navenek poskytuje metodu `getSelectedGenerator()`, která vrací vybraný generátor v podobě objektu třídy `Generator` s vyplněními parametry.

6.5 Vstupy a výstupy aplikace

6.5.1 Ukládání a načítání projektu

Ukládáním projektu je export objektů reprezentujících *SimCo aplikaci* do XML souboru tak, aby při následném načtení nebylo potřeba opět získávat informace o komponentách přes OSGi API a Java reflexi. Samotné generování XML je definováno třídou `StaxProjectFileWriter`, která implementuje rozhraní `IProjectFileWriter`. To obsahuje metodu `writeToFile()` s parametrem typu `ProjectComp`.

Načítání projektu z XML souboru zajišťuje `ProjectLoader` z balíku `simco.framework.appmanager.services`. Ten nejprve načte data z XML voláním metody `loadFromFile()` třídy `StaxProjectFileReader` a následně provede validaci událostí v načteném objektu typu `PojectComp` prostřednictvím třídy `EventValidator`.

Ukázka části XML souboru s uloženým projektem je v příloze B.

6.5.2 Generování scénáře

Hlavním výstupem aplikace je soubor scénáře pro *SimCo framework*. Obsahuje základní identifikaci použitých komponent a úplné definice událostí. Generování zajišťuje `StaxScenarioFileWriter` implementující rozhraní `IScenarioFileWriter`

[11] Poskytovaných *GeneratorHandlerem.*

s metodou `exportScenario()`. Té se kromě jména výstupního souboru předá, stejně jako v případě ukládání projektů, objekt typu `ProjectComp`. Ještě před vygenerováním souboru se provede kontrola, zda jsou správně vyplněné veškeré údaje, které má soubor scénáře obsahovat. Například každá reálná nebo simulační komponenta musí mít nastavenu cestu ke konfiguračnímu souboru pro *SimCo framework*. Tuto kontrolu provádí `ScenarioValidator`.

Ukázka jednoduchého XML souboru scénáře je v příloze C.

6.6 Práce s generátory

Již několikrát bylo vysvětleno používání generátorů v aplikaci.

SimcoAppManager poskytuje uživateli jen generátory implementované v *SimCo frameworku*. Definice těchto generátorů obsahuje XML soubor `generators.xml` v adresáři `resources` (viz ukázka kódu č. 6.2).

```
<?xml version="1.0" encoding="UTF-8"?>
   <simCoGenerators>
      <generator>
         <type name="gauss" csName="gaussovský" />
         <param name="mean" csName="střední hodnota" />
         <param name="standardDeviation" csName="směr. odchylka" />
         <output type="int" />
      </generator>
</simCoGenerators>
```

Ukázka kódu 6.2: Konfigurační soubor pro generátory

V uvedeném souboru je definován jeden generátor. Ten nese označení "gaussovský", má 2 parametry (střední hodnotu a standardní odchylku) a generované hodnoty jsou typu `int`.

Z pohledu implementace *SimcoAppManageru* poskytuje dostupné generátory `GeneratorHandler`. Ten při inicializaci s využitím `StaxGeneratorsFileReader` načte definované generátory ze souboru a uloží je do seznamu `generators`. Prostřednictvím metody `getGenarators() pak` poskytuje načtené generátory zbytku aplikace v podobě seznamu objektů třídy `Generator`.

Při implementování nového generátoru do *SimCo frameworku* stačí pouze přidat definici do výše uvedeného konfiguračního souboru. Poté může uživatel nový generátor používat na všech místech v *SimcoAppManageru*, kde je možné zadat parametr prostřednictvím generátoru.

6.7 Konfigurační soubory komponent

Každá komponenta musí mít pro použití v *SimCo frameworku* vytvořen konfigurační XML soubor. Ten musí obsahovat jména tříd implementující všechny služby poskytované komponentou. Dále může být prostřednictvím tohoto souboru zadáno zpoždění, které má nastat při volání vybrané metody.

SimcoAppManager umožňuje generování tohoto konfiguračního souboru, přičemž jména tříd jsou zjištěna automaticky voláním metody `getImplementClassName()` třídy `InfoExtract`. Následně je vytvořeno okno `SettFileMaker`, které zobrazuje seznam tříd a umožňuje přidat nastavení doby výpočtu pro metodu.

Datový model této konfigurace představuje třída `CompSettingsClass` s atributy pro jméno služby a jméno třídy, která tuto službu implementuje. Navíc má seznam objektů typu `CompSettingsMethod`, které nesou nastavení zpoždění (představující simulovanou dobu výpočtu) pro jednu konkrétní metodu třídy. V uvedeném seznamu nejsou objekty typu `CompSettingsMethod` pro všechny metody služby, ale pouze pro ty, jež mají nastaveno zpoždění, protože pouze ty se ukládají do konfiguračního souboru[12].

Generování datové reprezentace do souboru definuje `StaxSettingsFileMaker`.

Příklad konfiguračního souboru je v příloze D.

6.8 Podpůrné třídy

V balíku `simco.framework.appmanager.utils` jsou podpůrné třídy především pro lokalizaci, logování a vytváření dialogů.

6.8.1 Lokalizace

Všechny používané textové řetězce jsou umístěny na jednom místě v souboru `messages.properties` ve složce `resource/local`. Třída `LocalizationUtil` poskytuje `ResourceBundle`, který na základě zadaného klíče vrátí lokalizovaný řetězec. To umožňuje případnou snadnou lokalizaci aplikace do jiného jazyka.

6.8.2 Logování

Pro logování je použit bundle `com.springsource.org.apache.log4j`, ve kterém je zabalena knihovna *Log4j*. Pro podporu práce s touto knihovnou je vytvořena třída

[12] Pokud nejsou nastavena žádná zpoždění, obsahuje konfigurační soubor pouze seznam tříd.

`Logging`, která inicializovaný objekt „loggra“ poskytuje zbytku aplikace. Přes tento objekt je možné do souboru snadno zapisovat informační i chybové zprávy podle nastavené šablony.

6.8.3 Dialogy

Třída `Dialogs` poskytuje statické metody pro vytváření dialogů všech typů. Nejedná se pouze o jednoduché dialogy informačního nebo chybového charakteru. Jsou zde rovněž metody pro dialogy typu `JFileChooser`. Pokud je v aplikaci potřeba vybrat soubor (načíst, uložit, získat jméno), stačí zavolat metodu `chooseFile()` třídy `Dialogs`. Ta se postará o zobrazení a nastavení dialogu pro výběr souboru a jako návratovou hodnotu vrátí vybraný soubor.

6.9 Komponenta EfpProvider

Předchozí části kapitoly 6 se vztahovaly k implementaci bundlu *SimcoAppManager*. Tato podkapitola popisuje druhý vytvořený bundle – *EfpProvider*. Jedná se o nezávislý, samostatný bundle poskytující službu pro práci s mimofunkčními charakteristikami. *EfpProivder* je využíván bundlem *SimcoAppManager*.

6.9.1 Základní popis

EfpProvider zajišťuje načítání mimofunkčních charakteristik přiřazených ke komponentám (s využitím metadat z CRCE úložiště). Prostřednictvím služby poskytuje seznam všech mimofunkčních charakteristik definovaných pro *SimCo framework*, přičemž při zadání konkrétní komponenty může předvyplnit hodnoty těchto charakteristik na základě dat z CRCE. Dále umožňuje prostřednictvím GUI zobrazit mimofunkční charakteristiky načtené z CRCE metadat.

Komponenta je členěna na 5 balíků, přičemž nejdůležitější jsou:

- `simco.framework.efp.data`
  - Datového třídy.
- `simco.framework.efp.io`
  - Obsahuje třídy zajišťující načítání z metadat a z konfiguračního souboru.
- `simco.framework.efp.efpviewer`
  - Třídy pro vytvoření grafického uživatelského rozhraní.

6.9.2 Uložení dat

Na základě analýzy (viz kapitola 5.8) bylo rozhodnuto, že hodnoty mimofunkčních charakteristik přiřazené ke komponentám budou načítány z metadat poskytovaných

CRCE úložištěm. Velmi jednoduchý příklad takového souboru je v ukázce kódu č. 6.3. Element `resource` obsahuje identifikaci bundlu jako hodnotu atributu `crce:id` ve tvaru „`symbolicName_version`“. V tomto elementu může být libovolné množství elementů `capability`. Pro naše použití jsou důležité ty elementy, které obsahují element `attribute` s atributem „`name=API_method`“. V takovém případě obsahuje atribut `value` kompletní identifikaci metody ve tvaru „`service.method`“. Obsahuje-li tento element `property` elementy s definovanými `name`, `value`, `type` a `unit`, je tato `property` brána jako jedna mimofunkční charakteristika pro metodu reprezentovanou elementem `capability`. *EfpProvider* proto vytvoří a uloží odpovídající datové struktury (viz kapitola 6.9.3).

```
<?xml version="1.0" encoding="UTF-8"?>
<repository name='CRCE Repository' increment='13582741'
      xmlns="TBD-CRCE-METADATA-XSD-URI" xmlns:crce="http://crce">

  <resource crce:id="Calculator_1.0.0.qualifier" id="123456">
    <capability namespace="some.namespace">
      <attribute name="API_method"
             value="simco.application.ICalculator.getCountOfSteps"/>
      <property namespace='crce.metric'>
        <attribute name="name" value="invocation-response" />
        <attribute name="value" value="100" />
        <attribute name="type" value="int" />
        <attribute name="unit" value="msec" />
      </property>
    </capability>
  </resource>
</repository>
```

Ukázka kódu 6.3: Soubor metadat ke komponentě z CRCE

V uvedeném příkladu je definována mimofunkční charakteristika pro metodu „*getCountOfSteps*“ služby „*simco.application.calculator.ICalculator*“ se jménem „*invocation-response*“ a hodnotou 100 milisekund.

Načítání zajišťuje `EfpAssignedLoader`, který prochází specifikovaný adresář s metadaty z CRCE. Čtení jednotlivých souborů zajišťuje `EfpAssignedFileReader`. Zde je potřeba opět připomenout, že v současné době zatím není v CRCE implementována funkčnost (webová služba), která by poskytovala metadata k uloženým komponentám – proto jsou čteny z určené složky na disku. Po doimplementování CRCE by bylo možné upravit `EfpAssignedLoader` tak, aby metadata nenačítal z disku, ale získával je prostřednictvím webových služeb CRCE.

Z výše uvedeného vyplývá, že jsou *EfpProviderem* načteny všechny capability odpovídajícího tvaru. To může být obecně celá řada nejen mimofunkčních charakteristik dané metody. Ovšem *SimCo framework* umí měřit pouze některé

charakteristiky [13], je proto nutné vytvořit konfigurační soubor s definicemi EFP používaných SimCo frameworkem. Jednoduchý příklad tohoto konfiguračního souboru je v ukázce kódu č. 6.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<efpDefs>
  <efpRequirement>
    <name>maxDuration</name>
    <crcName>invocation-response</crcName>
    <csName>max. doba běhu</csName>
    <unit>msec</unit>
    <type>int</type>
  </efpRequirement>
</efpDefs>
```

Ukázka kódu 6.4: Soubor s definicemi EFP

EfpProvider prostřednictvím metody `getEfpsForComponentMethod()` vrací pouze EFP definované v tomto souboru, přestože z CRCE metadat může mít načteno široké spektrum mimofunkčních charakteristik. Pokud by do *SimCo frameworku* bylo přidáno měření dalších EFP, stačí pouze přidat definici tohoto EFP do konfiguračního souboru. *EfpProvider* bude následně poskytovat i odpovídající data z CRCE souborů.

6.9.3 Datový model

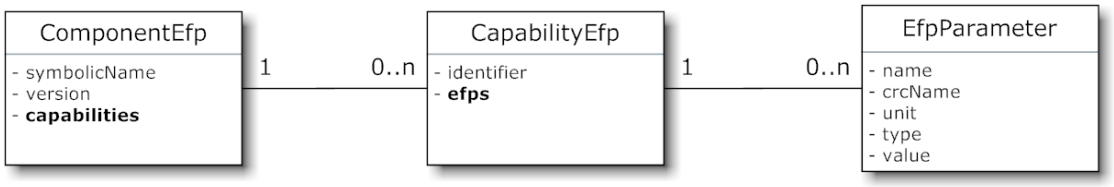
EfpProvider uchovává dvě kolekce:

- seznam definic EFP – `efps`,
- mapa přiřazující EFP k bundlům – `assignedEfps` (s klíčem ve tvaru „symbolicName_version“).

V prvním případě se jedná o kolekci `List` s položkami typu `EfpParametr`. V případě, kdy SimCo framework umožňuje měřit pouze jednu mimofunkční charakteristiku, obsahuje tento seznam pouze jednu položku.
Ve druhém případě je struktura dat složitější. Odpovídá uložení v souborech metadat z CRCE – každá komponenta může mít libovolný počet definovaných *capability*, které mohou mít libovolný počet definovaných *property*. V našem případě *property* představují mimofunkční charakteristiky, proto jsou reprezentovány třídou `EfpParameter`. Ta kromě definice obsahuje i konkrétní hodnotu dané charakteristiky. Nadřazená třída `CapabilityEfp` obsahuje kolekci typu `Map`, jejímž klíčem je jméno EFP a hodnotou objekt typu `EfpParameter`. Na vrcholu je třída `ComponentEfp` obsahující kolekci `capabilities`. Opět se jedná o mapu, kde klíčem je identifikátor

[13] V současné době pouze jednu – dobu běhu.

ve tvaru „service.method“ a hodnotou objekt typu `CapabilityEfp`. Uvedené vztahy jsou patné na obrázku číslo 6.4.



Obr. 6.4: Diagram tříd EFP přiřazených k metodě komponenty

Návrh datového modelu umožňuje snadné zjištění mimofunkčních charakteristik přiřazených ke konkrétní metodě komponenty. Jméno a verze bundlu se použije jako klíč do mapy `assignedEfps` pro získání instance třídy `ComponentEfp`, identifikátor služby a metoda se použijí jako klíč do mapy `capabilities` pro získání instance třídy `CapabilityEfp`. V její mapě `efps` jsou již konkrétní mimofunkční charakteristiky.

6.9.4 Prohlížeč EFP

EfpProvider umožňuje data načtená z CRCE souborů prohlížet, k čemuž slouží třída `EfpViewerGui` v balíku `simco.framework.efp.efpviewer`.

Jelikož mají načtená data stromovou strukturu (viz obrázek 8), je prostředkem zobrazení strom (`JTree`), přičemž listy stromu jsou atributy třídy `EfpParameter`.

Aby bylo prohlížení načtených mimofunkčních charakteristik pro uživatele přívětivější, lze vyhledávat v zobrazených datech podle identifikátoru bundlu nebo služby (včetně metody). To zajišťuje textové pole s přidaným *DocumentListenerem*, implementovaným vnitřní třídou `FilterTFListener`. Díky tomu je při změně obsahu textového pole volána metoda `updateTree()`, která upraví `TreeModel` zobrazeného stromu tak, aby odpovídal zadání uživatele. Tento filtr tudíž umožňuje zobrazit pouze EFP vybrané metody, nebo bundlu.

6.9.5 Poskytovaná služba

Služba poskytovaná bundlem *EfpProvider* je zaregistrována pod rozhraním `IEfpProvider`. To obsahuje 4 metody:

- `getEfps()`
  - Vrací seznam definic dostupných EFP.
- `getEfpsForComponentMethod(bundle, service, method)`
  - Vrací seznam dostupných EFP s vyplněními hodnotami pro konkrétní metodu.

- `viewAssignedEfps()`
  - Vytvoří GUI pro prohlížení EFP načtených z CRCE metadat.
- `viewAssignedEfps(filterType, filterText)`
  - Stejně jako v předchozím případě vytvoří prohlížeč EFP, zobrazí ovšem pouze EFP pro bundly nebo metody (podle typu filtru) odpovídající zadanému textu.

7 TESTOVÁNÍ

7.1 Testované komponenty

Pro otestování funkčnosti *SimcoAppManageru* byl použit komponentový souborový manažer, konkrétně jeho komponenty pro kompresi souborů do ZIP archivu a pro prohlížení souborů v HEXA podobě. Byly vytvořeny a otestovány celkem 3 *SimCo aplikace* skládající se ze tří komponent, přičemž každá aplikace obsahovala komponenty *SimcoFileDirManager a LocalFSAcces*. Třetí komponenty jednotlivých *SimCo aplikací* a účel testování jsou rozepsány v následujících odrážkách.

1) *ZipCompression2*
    - Test kompresní komponenty – volá se metoda zajišťující kompresi souborů, přičemž je nastavena očekávaná návratová hodnota a mimofunkční požadavek na dobu běhu.
2) *HEXAViewer*
    - Test HEXA prohlížeče, který načítá najednou celý soubor. Postupně se otevírají a zavírají okna prohlížeče pro různě velké soubory.
3) *FastHEXAViewer*
    - Test HEXA prohlížeče, který načítá soubory dynamicky podle právě prohlížené části. Události stejně jako v předchozím případě představují otevírání oken pro různě velké soubory.

Při testování se prošlo celým procesem od importu testovaných bundlů do simulačního prostředí až po vygenerování výsledků simulace v *SimCo frameworku*.

7.2 Vytvoření SimCo aplikace

7.2.1 Import bundlů do simulačního prostředí

Prvním krokem je import bundlů do prostředí Eclipse, ve kterém běží *SimCo framework* a *SimcoAppManager*. Jelikož máme testované bundly k dispozici v podobě projektu pro Eclipse, stačí pouze importovat tyto projekty do „workspacu“ Eclipse. Aby byly všechny importované bundly nainstalovány ve spuštěném OSGi frameworku, je nutné je vybrat ve spouštěcí konfiguraci. To se provede prostřednictvím *Run dialogu* na záložce *Bundles*. Po spuštění OSGi frameworku jsou již testované bundly souborového manažeru nainstalované v prostředí simulace, což umožňuje jejich načtení *SimcoAppManagerem*.

7.2.2 Tvorba scénáře

V tuto chvíli již přichází na řadu vytvořená aplikace *SimcoAppManager*. Následující popis se vztahuje k první *SimCo aplikaci*, která testuje komponentu *ZipCompression2*. Pro zbylé *SimCo aplikace* je postup analogický.

Kroky vedoucí k vygenerování testovacího scénáře pro *SimCo framework* jsou následující:

1) Vytvoření nového projektu s názvem *FileManagerCompression*.
2) Přidání dvou reálných komponent *LocalFSAcces* a *ZIPCompression2* přes dialog „Přidat komponentu“.
3) Přidání simulační komponenty *SimcoFileDirManager* přes dialog „Přidat komponentu“.
4) Automatické vygenerování konfiguračního souboru pro obě reálné komponenty (*LocalFSAcces* a *ZIPCompression2*). K tomu je využit dialog „Vytvořit konfigurační soubor“ dostupný z panelu dané komponenty.
5) Přes panel služby `filemanager.utils.ICompression` je vytvořena událost nad metodou `zipFSNode()` s nastavením:
   - Typ události: `REGULAR`.
   - Čas začátku nastaven na 1, čas konce na 1000 a perioda na 100.
   - Parametr metody typu `String` byl zadán hodnotou – obsahuje jméno souboru určeného ke komprimaci.
   - Byla zadána očekávaná návratová hodnota volané metody – *true*.
   - Byl připojen mimofunkční požadavek na kontrolu doby běhu nastavený na 20 milisekund.
6) Byly vytvořeny 4 kopie události z bodu 5, přičemž byl editován čas začátku a konce a parametr metody (jméno souboru).
7) V tuto chvíli je možné pro vybrané komponenty a definované události vygenerovat testovací scénář přes tlačítko „Exportovat scénář“.

7.2.3 Testy dalších funkcí

Kromě kroků potřebných k vygenerování požadovaných scénářů byly testovány i další funkce *SimcoAppManageru*:

- Odstranění komponenty ze *SimCo aplikace*.
- Odstranění události ze *SimCo aplikace*.
- Uložení celého projektu se *SimCo aplikací* do souboru.
- Načtení projektu ze souboru.
- Načtení dalších informací ke komponentám ze zdrojových souborů.
- Prohlížení EFP načtených z metadat z CRCE.

- Editace parametrů již vytvořených událostí.
- Přidání komponenty typu intermediate a zadání zpoždění.

7.3 Spuštění v SimCo frameworku

XML soubor se scénářem získaný v předchozím kroku je nyní použit jako vstup pro *SimCo framework*. Po načtení scénáře je spuštěna simulace. Výstupem je soubor s popisem událostí simulace včetně informací upozorňujících na nesplněné mimofunkční požadavky nebo očekávané návratové hodnoty.

7.4 Výsledky

7.4.1 ZIP komprese

Pro první *SimCo aplikaci*, která testovala komponentu *ZipCompression2*, je část výstupního souboru simulace na obrázku 7.1. Je vidět záznam dvou událostí, přičemž v prvním případě nebyla splněna očekávaná návratová hodnota (očekáváno `true`, vráceno `false`) a v druhém případě nebyl splněn mimofunkční požadavek (požadována doba běhu 20 ms, naměřeno 20,9547 ms).

```
service: filemanager.utils.ICompression
method: zipFSNode
parameterList: file004.dat (java.lang.String)
time: 3901
duration: 16,7311 ms
failed return value: true expected, false returned

service: filemanager.utils.ICompression
method: zipFSNode
parameterList: file05.dat (java.lang.String)
time: 4001
duration: 20,9547 ms
failed EFP - 20 ms expected, 20,9547 ms measured
```

Ukázka kódu 7.1: Část výstupního souboru SimCo frameworku

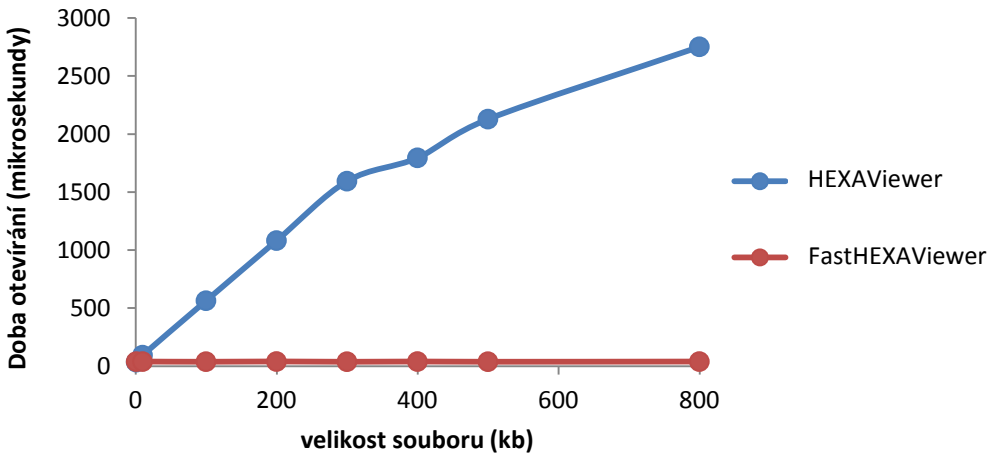
7.4.2 HEXA prohlížeč

V případě druhé a třetí *SimCo aplikace*, které testovaly dva HEXA prohlížeče, je zajímavější porovnat doby běhu. Komponenta *HEXAViewer* načítá celý zobrazovaný soubor, zatímco *FastHEXAViewer* načítá jen první část souboru.

Z výstupu SimCo frameworku byla vytvořena tabulka 7.1 ukazující závislost načtení souboru na jeho velikosti.

velikost souboru (kb)	doba otevírání (mikrosekundy)	
	HEXAViewer	FastHEXAViewer
1	34,003	38,536
10	94,063	38,875
100	563,372	37,971
200	1081,478	39,182
300	1593,445	37,903
400	1794,481	39,028
500	2128,229	37,945
800	2751,651	38,859
1024	5126,544	38,677

Tabulka 7.1: Závislost HEXA prohlížečů na velikosti souboru



Obr 7.1: Graf závislosti HEXA prohlížečů na velikosti souboru

Z uvedené tabulky je patrné, že otevření okna prohlížeče *FastHEXAViewer* je nezávislé na velikosti souborů, zatímco doba otevírání okna *HEXAVieweru* s velikostí souboru narůstá. Vše dokresluje graf na obrázku 7.1.

7.5 Zhodnocení

Testování prokázalo, že vyvinutá aplikace je funkční a umožňuje tak vytvoření *SimCo aplikace* a následné vygenerování souboru scénáře, který je zpracovatelný *Simco frameworkem*. Testování navíc ukázalo možnosti *SimCo frameworku* při ověřování mimofunkčních charakteristik.

8 ZÁVĚR

Cílem práce byla tvorba nástroje, který uživateli umožní jednoduché vytváření scénářů pro testování komponent v *SimCo frameworku* za účelem ověření mimofunkčních charakteristik.

Vyvíjená aplikace byla od začátku koncipována tak, aby nesloužila jen jako generátor scénářů, ale aby umožnila pokročilejším způsobem spravovat *SimCo aplikace*. Výsledná aplikace se skládá ze dvou komponent – *SimcoAppManager* a *EfpProvider*. *EfpProvider* poskytuje podporu pro práci s mimofunkčními charakteristikami – načítá data z metadata CRCE úložiště a ta poskytuje *SimcoAppManageru*. Zároveň umožňuje procházet veškerá načtená data pomocí jednoduchého prohlížeče. *SimcoAppManager* nabízí GUI pro výběr testovaných komponent (včetně vytváření jejich konfiguračních souborů) a definování událostí. Pro automatické získání požadovaných informací o komponentách je použita kombinace OSGi API, Java reflexe a služeb *EfpProvidera* (případně zdrojové soubory, jsou-li k dispozici). Při návrhu GUI byl kladen důraz především na jednoduché a intuitivní ovládání.

Výsledná aplikace uživatelům *SimCo frameworku* usnadní práci tím, že nebudou muset manuálně vytvářet testovací scénáře v podobě XML souborů, což znamenalo psaní množství režijního kódu a vyžadovalo znalost formátu XML souboru scénáře.

Při testování se prošlo celým procesem od importu bundlu do OSGi až po výsledek simulace *SimCo frameworku*. K tomu byly použity komponenty souborového manažeru, přičemž byly testovány různé možnosti nastavení událostí tak, aby bylo při simulaci dosaženo úspěšné i neúspěšné verifikace. Všechny body zadání tím byly splněny.

Aplikace může běžet samostatně, díky poskytované OSGi službě je možné ji rovněž snadno integrovat do grafického uživatelského rozhraní *SimCo frameworku*. Dále umožňuje jednoduché rozšíření o nové generátory a mimofunkční charakteristiky, pokud by byly doimplementovány do *SimCo frameworku*. Co se týká budoucích vylepšení, po nasazení CRCE úložiště do provozu je možné upravit *EfpProvider*, aby soubory s metadaty nenačítal z lokálního disku, ale aby je získával prostřednictvím webových služeb z CRCE úložiště.

PŘEHLED ZKRATEK

API	Application Programming Interface
CBSE	Component-Based Software Engineering
CRCE	Component Repository supporting Compatibility Evaluation
DOM	Document Object Model
EFFCC	Extra-Functional Property Featured Compatibility Checks
EFP	Extra-functional Properties
GUI	Graphical User Interface – grafické uživatelské rozhraní
JAXB	Java Architecture for XML Binding
OSGi	Open Service Gateway initiative
SAX	Simple API for XML
StAX	Streaming API for XML
XML	Extensible Markup Language

LITERATURA A ZDROJE

[BAC00] BACHMANN, F.: ***Technical Concepts of Component-Based Software Engineering***, [online]. Carnegie Mellon University, 2000 [cit. 2014-02-09]. Dostupné z: http://www.sei.cmu.edu/reports/00tr008.pdf

[HOL11] HOLÝ, L.: ***Komponentové modely a architektury***. [online]. 2011 [cit. 2014-02-09].
Dostupné z: http://wiki.kiv.zcu.cz/uploads/UvodDoKomponent

[OSG14] OSGi. [online]. [cit. 2014-02-10]. Dostupné z: http://www.osgi.org

[CAL12] DE CASTRO ALVES, A.: ***OSGi in Depth***. New York: Manning Publications Co., 2012.

[EDG13] EDSTROM, J., GOODYEAR J.: ***Instant OSGi starter***. Birmingham: Packt Publishing, 2013.

[WAL09] WALLS, C.: ***Modular Java***. Raleigh: Pragmatic Bookshelf, 2009.

[PRO11] PROKOP, M.: ***Vizualizace simulace komponent***, diplomová práce ZČU-KIV, 2011.

[KAB11] KABÍČEK, T.: ***Simulační systém softwarových komponent založený na komponentách***, diplomová práce ZČU-KIV, 2011.

[JEZ12] JEŽEK, K.: ***Extra-Functional Properties Support For a Variety of Component Models***, dizertační práce ZČU-KIV, 2012.

[GLI07] GLINZ, M.: ***On Non-Functional Requirements***. [online]. 2007 [cit. 2014-03-25].
Dostupné z:
http://courses.cs.ut.ee/2010/sem/uploads/Main/07RE-reading-nfr.pdf

[KUC11] KUČERA, J.: ***Úložiště komponent podporující kontroly kompatibility***, diplomová práce ZČU-KIV, 2011.

[RUZ12] RŮŽIČKA, Z.: ***Rozšíření frameworku pro ověřování kompatibility softwarových komponent***, bakalářská práce ZČU-KIV, 2012.

[OBR14] OSGi Bundle Repository (OBR) - Apache. [online]. [cit. 2014-04-20]. Dostupné z:
http://felix.apache.org/site/apache-felix-osgi-bundle-repository.html

[SDM14] Spring Dynamic Modules Reference Guide [online]. [cit. 2014-04-25]. Dostupné z: http://docs.spring.io/osgi/docs/1.2.1/reference/html/

[QDX14] QDox [online]. [cit. 2014-04-25].
Dostupné z: http://qdox.codehaus.org/

[MUS14] MockupScreens [online]. [cit. 2014-04-27].
http://www.mockupscreens.com/

[MIG14] MigLayout - Java Layout Manager [online]. [cit. 2014-04-27].
http://www.miglayout.com/

[HER07] HEROUT, P.: ***Java a XML***. 1. vydání, České Budějovice: Kopp, 2007.

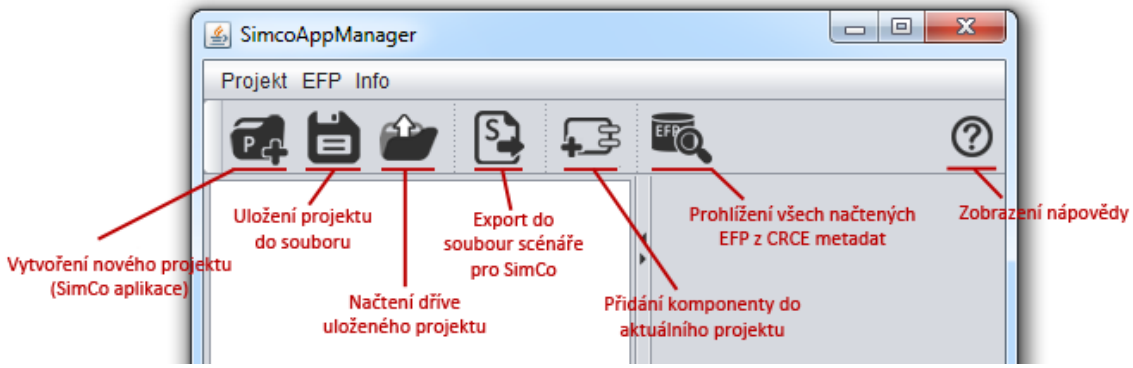
PŘÍLOHA A – UŽIVATELSKÁ PŘÍRUČKA

Instalace a spuštění

Běhovým prostředím aplikace je OSGi framework se Spring DM bundly. Pro jednoduché spuštění je na přiloženém CD platforma Eclipse, ve které je již vše nakonfigurováno. Pro spuštění stačí kliknout na ikonu *Run* (konfigurace se jmenuje „SimcoAppManager“).

Ovládání aplikace

Po spuštění se otevře hlavní okno aplikace. Ovládání je možné přes menu, ikony toolbaru nebo klávesové zkratky. Popis ovládacích prvků na toolbaru je na obrázku A.1.



Obr. A.1: Základní ovládací prvky

Vytvoření / uložení / načtení projektu

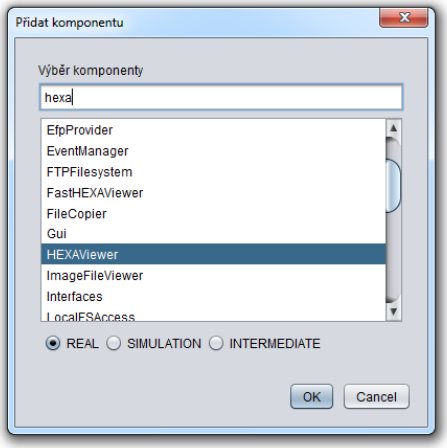
SimCo aplikace je v *SimcoAppMangeru* nazývána *projekt*. Prvním krokem je vytvoření nového projektu, do kterého je následně možné přidávat komponenty a vytvářet události. Vytvořený projekt lze uložit do souboru. Projekt lze rovněž načíst ze souboru kliknutím na příslušnou ikonu, nebo použitím *Drag And Drop* přetáhnout soubor s projektem ze souborového manažeru do levé části hlavního okna.

Výběr komponenty

Do vytvořeného (načteného) projektu lze přidávat komponenty, které jsou nainstalovány v běžícím OSGi[14]. Po kliknutí na „Přidat komponentu“ se zobrazí dialog

[14] Testované komponenty je proto nutné nejprve nainstalovat do OSGi. Buď přes dostupnou OSGi konzoli, nebo importem projektu do workspace Eclipse a nastavením jejich spuštění přes *Run Dialog*.

umožňující provést výběr z komponent nainstalovaných v OSGi a určit typ pro *SimCo framework*. Uvedený dialog je vidět na obrázku A.2.



Obr. A.2: Přidání komponenty

Hlavní okno

Na obrázku A.3 je vidět hlavní okno aplikace – je rozděleno na 2 části:

- *Levá část*
  - Strom reprezentující projekt – obsahuje komponenty a události. Jednotlivé komponenty mají pod sebou poskytované služby. Při výběru položky stromu se v pravé části zobrazí odpovídající panel.
- Pravá část
  - Může být zobrazen panel pro projekt, komponentu, službu nebo událost. Každý panel obsahuje údaje, které je možné editovat – provedené změny je potřeba potvrdit tlačítkem „Uložit změny“.

Panel OSGi služby

Na obrázku A.3 je vidět situace, kdy je vybrána (a tím pádem v pravé části zobrazena) služba poskytovaná bundlem.

V horní části je možné vybrat zdrojový soubor pro načtení dalších informací. Jsou zde tří tlačítka:

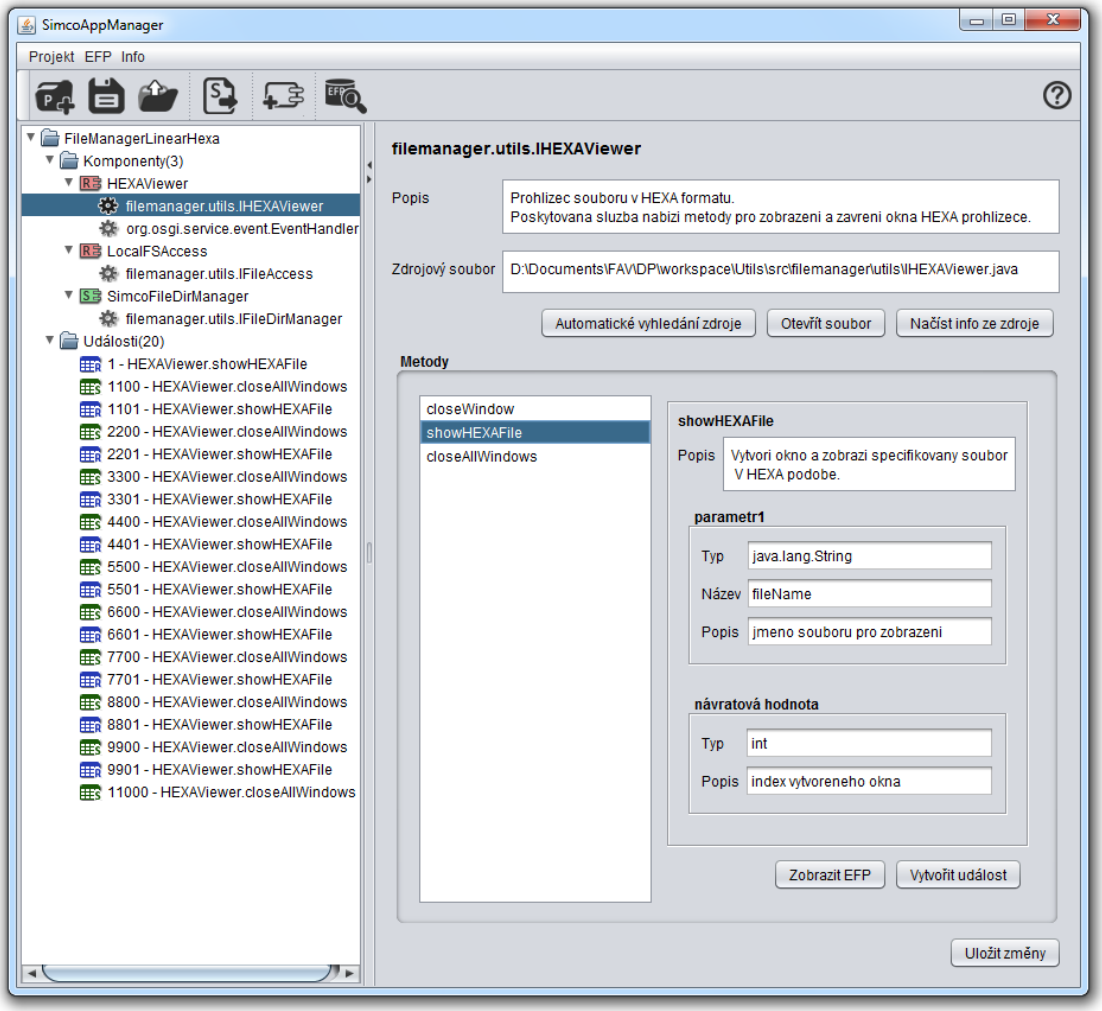
- *Automatické vyhledání zdroje* – pokud je bundle jako projekt importovaný ve workspace Eclipse, je možné využít tuto funkci, která prohledává workspace. Jedná se ovšem pouze o pomocnou funkci, která nezaručuje úspěšné nalezení souboru[15].

[15] Například registruje-li bundle službu pod rozhraním importovaným z jiného bundlu, algoritmus zdrojový soubor nenalezne.

- *Otevřít soubor* – otevře zdrojový soubor v textovém editoru.
- *Načíst info ze zdroje* – načte jména parametrů a *javadoc* komentáře z vybraného zdrojového souboru.

V dolní části panelu je seznam metod dané služby. Po výběru metody se zobrazí parametry a návratová hodnota metody. Po kliknutí na „Zobrazit EFP“ se zobrazí okno s mimofunkčními charakteristikami načtenými k této metodě z CRCE metadat.

Důležité je tlačítko „Vytvořit událost“. To zobrazí okno pro zadání parametrů a následné vytvoření události.



Obr. A.3: Hlavní okno se zobrazeným panelem služby

Vytvoření události

Po kliknutí na „Vytvořit událost“ se objeví okno (viz obrázek A.4) pro zadání všech parametrů události. Předně je to typ události (*casual* / *regular* / *singular*). Následně se zadají časové údaje k události – „Detail události“.

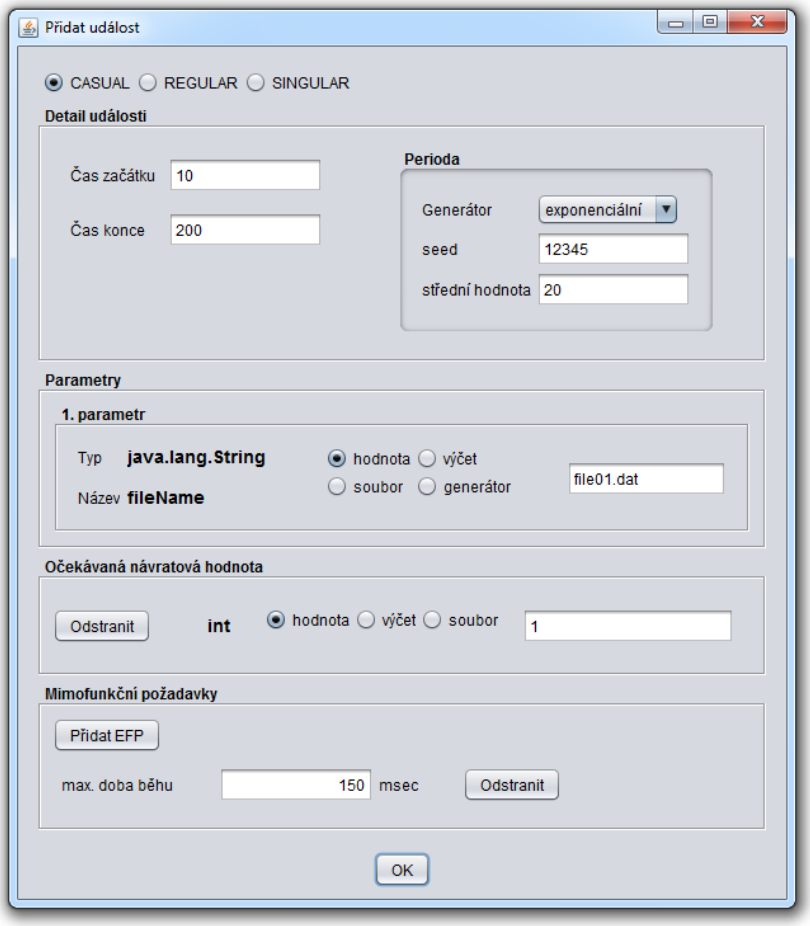
Další nastavení se týká parametrů metody – konkrétní hodnoty lze zadat:

- jednou hodnotou,
- výčtem hodnot,
- souborem obsahujícím hodnoty,
- generátorem.

Podle vybrané možnosti se zobrazí textové pole nebo panel s generátorem.

Analogickým způsobem je možné rovněž zadat očekávanou návratovou hodnotu. Pokud zadána není, v *SimCo frameworku* se nekontroluje.

Poslední možností je přidání mimofunkčních charakteristik, které se mají testovat.



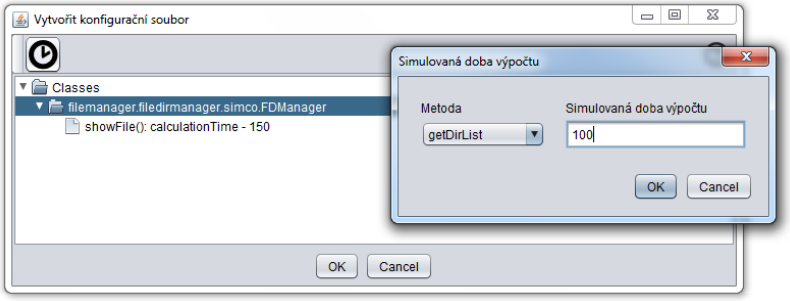
Obr. A.4: Vytvoření události

Stejně vypadající panel (jako na obr. A.4) je na panelu události – to umožňuje pozdější editaci všech parametrů události. Na tomto panelu lze rovněž vytvořit kopii události.

Vytvoření konfiguračního souboru komponenty

Každá komponenta musí mít vytvořen konfigurační soubor. Zároveň musí mít nastavenu cestu k tomuto souboru. Obojí lze provést na panelu komponenty. Po kliknutí

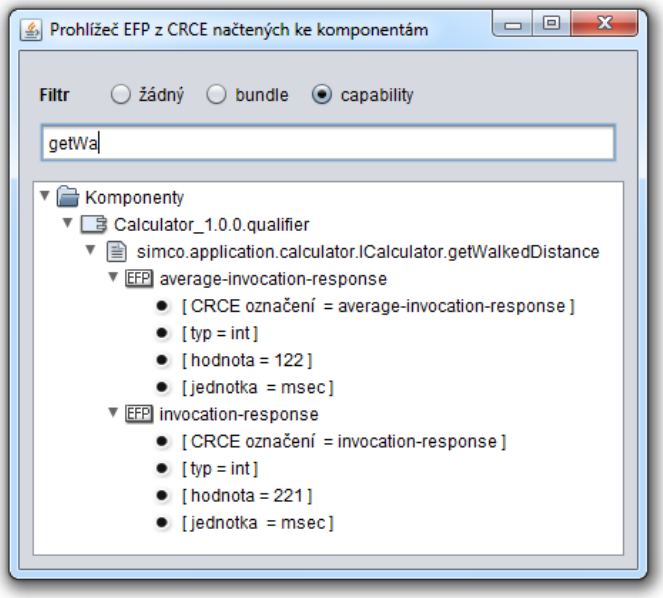
na „Vytvořit konfigurační soubor“ se zobrazí dialog, jenž je ukázán na obrázku A.5. Zde je seznam všech tříd implementujících služby daného bundlu. Je možné nastavit simulovanou dobu výpočtu pro vybrané metody zobrazených tříd – výběrem položky ve stromu a kliknutím na ikonu hodin. Kliknutím na „OK“ se vytvoří konfigurační soubor.



Obr. A.5: Dialog pro vytvoření konfiguračního souboru komponenty

Prohlížení EFP z CRCE

Přes ikonu toolbaru „Prohlédnout EFP“ lze prohlédnout veškerá data načtená z CRCE metadat. Zobrazená struktura má hierarchii bundle – capability – EFP – atributy. Je možné použít filtrování a tím například zobrazit jen konkrétní bundle nebo konkrétní metodu. Na obrázku A.6 je použití filtru pro výpis EFP k metodě „getWalkedDistance“.



Obr. A.6: Prohlížeč EFP načtených z CRCE metadat

Export scénáře

Pro export vytvořeného projektu do souboru scénáře pro *SimCo framework* stačí kliknout na ikonu „Exportovat scénář“.

PŘÍLOHA B – UKÁZKA ČÁSTI SOUBORU S PROJEKTEM

Na ukázce kódu B.1 je část souboru s uloženým projektem. Je zde jeden element `component`, který obsahuje informace o komponentě. Celý soubor pak obsahuje v elementu `components` analogický popis pro všechny komponenty v projektu. Dále jsou v souboru popisy všech definovaných událostí v elementu `events`. Ty mají stejný formát jako v souboru scénáře (viz příloha C).

```
<component type="REAL">
   <symbolicName>HEXAViewer</symbolicName>
   <version>1.0.0.qualifier</version>
   <description>Zobrazuje soubory v hexa formátu</description>
   <settingsFile>hexaConfig.xml</settingsFile>
   <service>
      <interface>filemanager.utils.IHEXAViewer</interface>
      <description>Prohlizec souboru v HEXA formatu.
                   Poskytovana sluzba nabizi metody pro otereni a zavreni
                   okna HEXA prohlizece.</description>
      <sourceFile>D:\filemanager\IHEXAViewer.java</sourceFile>
      <method>
         <name>closeWindow</name>
         <description>Zavre specifikovane okno.</description>
         <parameter>
            <name>index</name>
            <type>int</type>
            <description>index okna, jez se ma zavrit</description>
         </parameter>
         <returnVar>
            <type>void</type>
            <description/>
         </returnVar>
      </method>
      <method>
         <name>showHEXAFile</name>
         <description>Vytvori okno a zobrazi specifikovany soubor v HEXA
                      podobe.</description>
         <parameter>
            <name>fileName</name>
            <type>java.lang.String</type>
            <description>jmeno souboru pro zobrazeni</description>
         </parameter>
         <returnVar>
            <type>int</type>
            <description>index vytvoreneho okna</description>
         </returnVar>
      </method>
      <method>
         <name>closeAllWindows</name>
         <description>Zavre vsechna otevrena okna.</description>
         <returnVar>
            <type>void</type>
            <description/>
         </returnVar>
      </method>
   </service>
<component>
```

Ukázka kódu B.1: Ukázka elementu „component“ ze souboru projektu

PŘÍLOHA C – UKÁZKA SOUBORU SCÉNÁŘE

Na ukázce kódu C.1 je kompletní soubor jednoduchého scénáře vygenerovaného *SimcoAppManagerem* pro *SimCo framework*.

```
<?xml version="1.0" encoding="UTF-8"?>
<simCoScenario>
   <components>
      <component type="REAL">
         <symbolicName>Calculator</symbolicName>
         <settingsFile>calculatorSettings.xml</settingsFile>
         <intermediateSymbolicName>CalculatorIntermediate
                                    </intermediateSymbolicName>
      </component>
      <component type="INTERMEDIATE">
         <symbolicName>CalculatorIntermediate</symbolicName>
         <delay type="CASUAL">
            <probability>
               <generator>
                  <name>gauss</name>
                  <seed>12345</seed>
                  <param key="mean">70</param>
                  <param key="standardDeviation">20</param>
               </generator>
            </probability>
            <lenght>
               <generator>
                  <name>gauss</name>
                  <seed>12345</seed>
                  <param key="mean">15</param>
                  <param key="standardDeviation">3</param>
               </generator>
            </lenght>
         </delay>
      </component>
      <component type="REAL">
         <symbolicName>Presenter</symbolicName>
         <settingsFile>presenterSettings.xml</settingsFile>
      </component>
      <component type="SIMULATION">
         <symbolicName>SimcoGPS</symbolicName>
         <settingsFile>gpsSettings.xml</settingsFile>
      </component>
      <component type="REAL">
         <symbolicName>GPS</symbolicName>
         <settingsFile>gpsSettings.xml</settingsFile>
      </component>
      <component type="REAL">
         <symbolicName>Accelerometer</symbolicName>
         <settingsFile>accelerometerSettings.xml</settingsFile>
      </component>
      <component type="SIMULATION">
         <symbolicName>SimcoAccelerometer</symbolicName>
         <settingsFile>simcoAccelerometerSettings.xml</settingsFile>
      </component>
   </components>
```

```
    <events>
        <event type="CASUAL">
            <source>Presenter</source>
            <serviceName>simco.application.presenter.IPresenter</serviceName>
            <methodName>refreshDisplay</methodName>
            <methodParameters/>
            <eventDetails>
                <detail key="start" val="1"/>
                <detail key="stop" val="2000"/>
                <detail key="period">
                    <generator>
                    <name>exponencial</name>
                        <seed>12345</seed>
                        <param key="mean">250</param>
                    </generator>
                </detail>
            </eventDetails>
            <efpRequirements>
                <efpRequirement>
                    <name>maxDuration</name>
                    <value unit="msec" type="int">10</value>
                </efpRequirement>
            </efpRequirements>
        </event>
        <event type="REGULAR">
            <source>CalculatorIntermediate</source>
            <serviceName>simco.application.calculator.ICalculator</serviceName>
            <methodName>testMethod</methodName>
            <methodParameters>
                <parameter dataType="int" order="1">
                    <generator>
                        <name>exponencial</name>
                        <seed>12345</seed>
                        <param key="mean">30</param>
                    </generator>
                </parameter>
                <parameter dataType="java.lang.String" order="2">
                    <value order="1">walk</value>
                    <value order="2">bike</value>
                </parameter>
            </methodParameters>
                <expectedResults>
                    <result dataType="int">
                    <value order="1">0</value>
                </result>
            </expectedResults>
            <eventDetails>
                <detail key="start" val="10"/>
                <detail key="stop" val="200"/>
                <detail key="period" val="20"/>
            </eventDetails>
            <efpRequirements>
                <efpRequirement>
                    <name>maxDuration</name>
                    <value unit="msec" type="int">10</value>
                </efpRequirement>
            </efpRequirements>
        </event>
    </events>
</simCoScenario>
```

Ukázka kódu C.1: Soubor scénáře

Příloha D – Ukázka konfiguračního souboru komponenty

Na ukázce kódu D.1 je konfigurační soubor komponenty pro SimCo framework. Na uvedeném příkladu je komponenta poskytující jednu službu - má jednu implementující třídu. K jedné z metod služby (`getDirList()`) je přidána simulovaná doba výpočtu s hodnotou 100.

```
<?xml version="1.0" encoding="UTF-8"?>
<componentSettings name="SimcoFileDirManager">
    <classes>
        <class name="filemanager.filedirmanager.simco.FDManager">
            <methodSettings>
                <method name="getDirList">
                    <calculationTime value="100"/>
                </method>
            </methodSettings>
        </class>
    </classes>
</componentSettings>
```

Ukázka kódu D.1: Konfigurační soubor komponenty