Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Tvorba a správa scénářů pro testování komponentových aplikací

Prohlášení

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

V Plzni dne 8.5.2013,

Ondřej Kouba

Poděkování

Chtěl bych poděkovat zejména vedoucímu této práce Ing. Richardovi Lipkovi PhD. za jeho čas, trpělivost a skvělé vedení, dále firmě Certicon a.s. za zkušenosti a znalosti, které mi pomohly vypracovat tuto diplomovou práci, a nakonec Petře Hinnerové za jazykovou korekturu a toleranci mých cholerických záchvatů.

Abstract

The main goal of my thesis is to create a tool for automatic scenario generation for SimCo framework, which is a tool designed for testing OSGI component based application developed at the Department of Computer Science and Engineering at University of West Bohemia. A SimCo user can choose which bundles shall be tested and which should be replaced by their proper simulation – a mock up bundle. The simulated bundles and whole scenario had to be created manually in order to use the SimCo so far, which was time demanding and uncomfortable. This thesis creates the simulation bundles from real bundles and fabricates the scenario skeleton for the user saving his time and letting him to focus more on the testing than on writing boiler plate code. This thesis could also be used outside the SimCo framework as a tool for Rapid Application Development as a Bundle stub generator.

Obsah

1 Úvod

Na začátek bych rád čtenáře uvedl do problematiky testování OSGI komponentových aplikací. Mějme aplikaci složenou z komponent. Stejně jako v opravdovém životě i ve světě IT můžeme problém dekomponovat do jednotlivých funkčních celků. Jako příklad z reálného světa si můžeme vzít osobní automobil. Ten se skládá ze spousty vzájemně spolupracujících komponent. Jistě obsahuje volant, motor, kola, ale také rádio, klimatizaci, stěrače, ostřikovače a další součástky. Pro funkčnost auta samozřejmě nepotřebujeme všechny díly, postačí nám jenom ty základní - motor, kola a volant. Podobná je situace u komponentových aplikací, ty se také skládají z různých částí, které spolu mohou a nemusí souviset. Pro správnou funkci programu spolu musí komponenty interagovat stejně jako u osobního automobilu. Říkáme, že spolu kompontenty komunikují. Pokud je pro správnou funkci vyžadována další komponenta, říkáme, že na sobě závisí. Stejně jako v případě osobního automobilu pro běh aplikace také nepotřebujeme všechny komponenty, ale stačí jistý základ, jádro aplikace. Platí tedy, že ne všechny komponenty spolu komunikují, a ne všechny komponenty jsou na sobě závislé.

Pokud budeme uvažovat testování takovéto komponentové aplikace, opět můžeme vycházet z příkladu automobilu. Každá automobilka musí provádět takzvané crash testy. Ty mají otestovat, jak se automobil bude chovat v případě nárazu. V tomto případě je vozidlo zdemolováno a tudíž je snaha o dosažení co nejrychlejšího a nejlevnějšího testování. Např. místo motoru jistě můžeme použít nefunkční maketu, stejně tak místo palubního počítače. Určitě bychom těžko hledali dobrovolníka, který bude sedět ve vozidle, zatímco s ním nabouráme velkou rychlostí do zdi, proto použijeme dobrou aproximaci pomocí anatomické figuríny. Jediné, co musí zůstat pravé a funkční, je karoserie a airbagy, protože právě ty chceme otestovat. Pokud chceme otestovat funkčnost motoru, nepotřebujeme nutně sedačky pro cestující, rádio, ostřikovače, možná ani karoserii.

Pokud tedy testujeme komponentovou aplikaci, nic nám nebrání použít místo opravdových komponent, které například čtou data z databáze, starají se o jejich integritu nebo počítají nějakým složitým algoritmem zajímavé statistiky, komponenty falešné (v angličtině označované jako „Mock up“ nebo „Stub“), které vždy vrátí stejný záznam, stejnou statistiku, ale ve výrazně kratším čase. Tento přístup si samozřejmě můžeme dovolit (stejně jako v předchozím příkladu s crash testem) pouze v případě, že netestujeme komponentu, kterou nahrazujeme falešnou, v tomto případě databázi nebo generování statistik.

Cílem této práce je napsat program generující scénáře pro testování komponentových aplikací. Komponentová aplikace je specifická v tom, že jednotlivé kusy aplikace jsou rozděleny do komponent, které obsahují samotné implementační prvky. V našem případě se jedná o třídy. Některé jsou použitelné zvenku, jsou to třídy, které poskytujeme ostatním komponentám. Kromě těchto tříd komponenta obsahuje také třídy, které používáme vnitřně, a jsou nutné pouze pro naši funkci.

Pokud chceme testovat komponentovou aplikaci, můžeme využít výhody tohoto rozdělení. Komponentu, kterou budeme chtít testovat, obalíme stuby, které imitují požadované chování okolních komponent. Pomocí nich poté můžeme simulovat všechny možné situace

a ověřit reakci testované komponenty. Třídy, které je potřeba stubovat, jsou pouze viditelné navenek. Tak můžeme snadno otestovat funkcionalitu v námi vyžadovaných scénářích, aniž bychom museli řešit, jak fungují okolní komponenty. A právě generováním těchto stubů celých komponent se zabývá tato diplomová práce.

1.1 Motivace

V současné době je vyvíjen framework SimCo, který poskytuje zajímavé možnosti pro testování komponentových aplikací. Pro jeho použití je ale potřeba napsat stuby komponent, což vyžaduje zbytečnou stereotypní práci, kterou je možné zautomatizovat. Místo psaní boiler-plate[1] kódu se tedy tester může soustředit na psaní toho, co chce opravdu otestovat.

Cílem této diplomové práce je tedy generování právě těchto simulačních komponent. Jejich využití je vhodné nejenom pro testování pomocí SimCo frameworku, ale tento nástroj by měl být také schopný vytvořit zdrojové texty komponenty z již zkompilovaných souborů. Automatické generování balíku také můžeme použít v rámci extrémního programování, kdy balíkem nasimulujeme očekávané chování, které někdo právě vyvíjí.

Motivací je tedy snaha ušetřit čas, zbytečné úsilí a stereotypní práci při psaní testovacích balíků.

[1] Jako boiler plate kód nebo také boilerplate je označován kód, který se používá na více místech téměř s nulovou úpravou. Tento termín se také používá v případě, že programátor musí napsat zbytečně mnoho řádek kódu k implementování jednoduché funkcionality.

2 OSGI a Eclipse Equinox

OSGI je specifikace komponentového frameworku běžícího nad Javou, spravována a vyvíjena neziskovou korporací OSGi Alliance, která byla založena v roce 1999. Členové OSGI Alliance spravují API a poskytují nápady, podněty a inovaci na poli této komponentově orientované architektury. Mezi firmy, podílející se na vývoji, patří i takoví giganti ve světě IT jako Adobe, Hitachi, IBM, Oracle, Red Hat nebo Siemens.

Vzhledem k tomu, že OSGI Alliance vytváří pouze rozhraní, existuje více implementací. V této práci bude použita implementace Equinox od Eclipse.

2.1 Historie

Na začátku oba projekty, OSGI i Eclipse, běžely nezávisle na sobě, téměř bez vzájemného ovlivňování. OSGI Alliance byla volným sdružením prodejců embedded a telekomunikačních systémů. Jejich specifikace komponentového rozhraní vznikla velice rychle a postupem času se objevovalo čím dál více implementací. OSGI Alliance se dále zaměřovala na embedded systémy a právě ty mají hlavní zásluhu na tom, jak OSGI vypadá dnes.

Nadace Eclipse vznikla jako volné sdružení výrobců softwarových nástrojů určených zejména pro vývoj integrovaných vývojových prostředí, anglicky IDE (Integrated Development Environment). Jejich technologie se brzy rozšířily a zabraly významné procento trhu. Velké softwarové společnosti začaly používat Eclipse platformu pro vývoj svých hlavních produktů, zejména kvůli dvěma faktorům – modularitě a otevřené komunitě Eclipse. Jak se postupem času platforma Eclipse rozrůstala z důvodů standardizace a lepší podpory dynamické modularity, bylo potřeba změnit jádro celého projektu. Tak vznikl projekt Equinox v roce 2003.

OSGI nebylo automatickou volbou, ale do tohoto projektu bylo vybráno na základě vypracované studie proveditelnosti. Vzhledem k tomu, že OSGI je pouze specifikace, bylo potřeba rozhodnout, jaká konkrétní implementace bude použita pro Eclipse platformu. V té době neexistovalo tolik open source řešení jako dnes, jediné použitelné byl Felix od Apache. Dále v úvahu připadalo řešení od IBM – Service Management Framework (SMF). Pro výběr IBM hrál fakt, že se jednalo o vyzkoušenou technologii nasazenou v několika velkých komerčních projektech. Rozhodování velice ulehčilo také to, že se IBM rozhodlo věnovat zdarma kód SMF Equinox týmu. Na základě této vzájemně výhodné dohody vzniklo například vývojové prostředí Rational Application Development od IBM, které je právě postaveno na Equinox platformě. [1]

Nadace Eclipse přispěla do OSGI specifikace několika důležitými koncepty. Lazy aktivací balíků, fragmenty, sémantikou jmen, verzí balíků a závislostmi mezi nimi. Závislostmi mezi komponentami se do velké míry zabývá tato diplomová práce.

Právě díky Equinoxu a IBM je Eclipse nadace hrdým členem OSGI Aliance.

2.2 Architektura

Technologie OSGI je soubor specifikací, které definují dynamický komponentový systém pro programovací jazyk Java. Tyto specifikace umožňují vývoj aplikací, jež jsou dynamicky složeny z mnoha znovupoužitelných komponent. OSGI specifikace umožňuje odstínit implementaci jednotlivých komponent použitím služeb, což jsou objekty s definovaným rozhraním sdílené mezi komponentami aplikace. [2]

Nápadů pro využití komponentově orientovaného návrhu bylo více, ale OSGI je první opravdu úspěšná specifikace, která našla skutečně masivní použití v praxi.

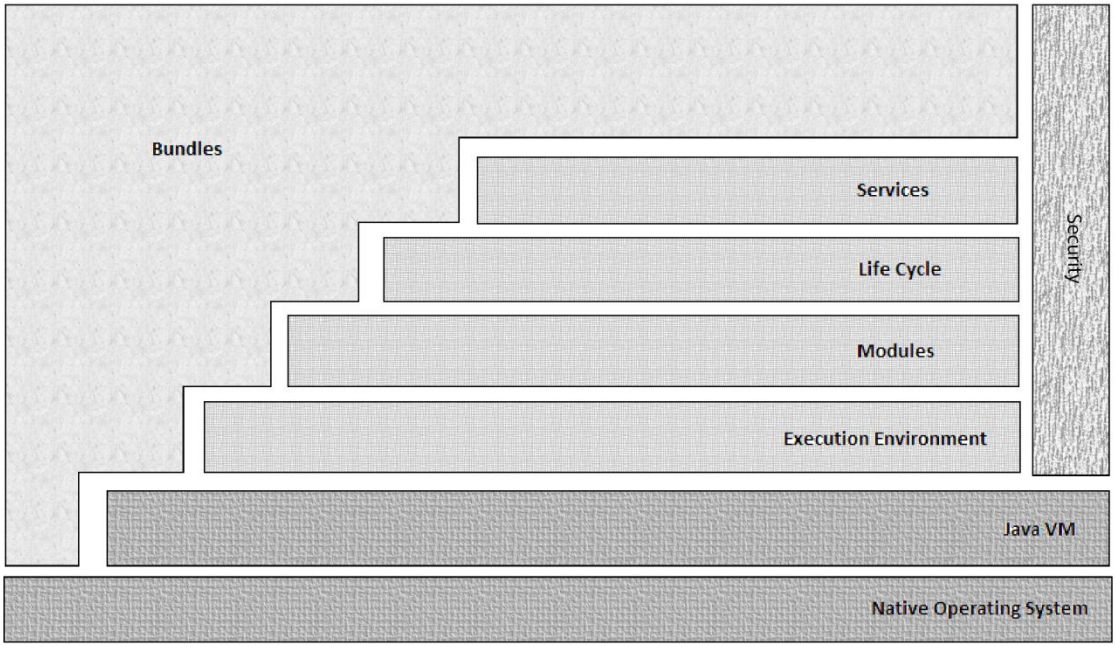
Hlavními výhodami OSGI jsou:

- **Dynamičnost** - jednotlivé kusy aplikace(komponenty) je možné dynamicky přidávat za běhu
- **Modulárnost** - aplikace se skládá z mnoha samostatných částí, které spolu komunikují předem definovaným způsobem(službou)
- **Bezpečnost** - balík sám definuje, co přesně mohou ostatní balíky použít.
- **Snadná orientace v závislostech** – komponenta přesně definuje, jaké další prvky potřebuje ke svému běhu.

Z těchto benefitů jednoznačně vyplývá i snadná testovatelnost, neboť každý balík lze otestovat samostatně. Přínosem je také usnadněný vývoj. Komponenty mohou být vyvíjeny nezávisle na sobě.

2.2.1 Vrstvy

OSGI používá architekturu vrstev vystavených nad Java virtuálním strojem, která je znázorněna na následujícím diagramu:



Obrázek 1 - Osgi vrstvy [2]

Následující seznam obsahuje vysvětlení jednotlivých vrstev:

- **Bundles** - jedná se o OSGI komponenty napsané vývojáři.
- **Services** - vrstva služeb, která dynamicky spojuje jednotlivé komponenty napsané vývojáři. Každý balík může požádat OSGI Framework o službu, kterou definoval jiný balík.
- **Life Cycle** - stará se o životní cyklus komponent – instalaci, spuštění, zastavení, aktualizaci a odinstalování komponent z Frameworku.
- **Modules** - určuje, jak komponenta může importovat a exportovat kód.
- **Execution Environment** - definuje, jaké třídy a metody jsou k dispozici v určité platformě.
- **Security** - stará se o bezpečnostní stránky Frameworku a dbá na to, aby bezpečnost nebyla porušována.

2.2.2 Komponenta, její struktura a životní cyklus

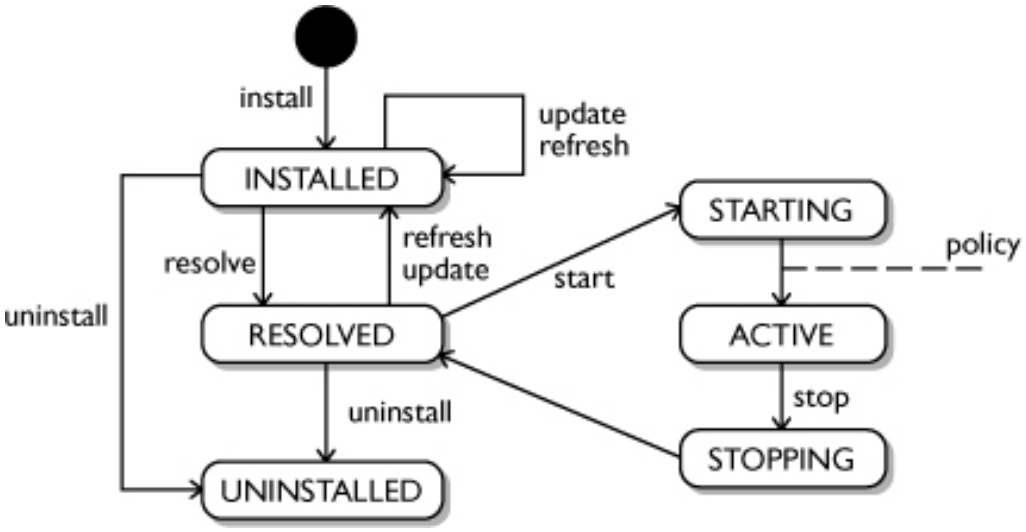
Komponenty jsou základním stavebním kamenem aplikace. Jsou označovány anglickým slovem bundle nebo, v terminologii Equinoxu, také jako zásuvný modul, anglicky plug-in. V této práci bude dále pro označování komponent používáno slovo bundle.

Každá komponenta obsahuje soubor, popisující jaké ostatní bundly vyžadujeme pro správnou funkcionalitu, tedy závislosti na ostatních prvcích aplikace. Dále zde také definujeme, jakou minimální verzi Javy vyžadujeme, jakou funkcionalitu poskytujeme navenek, jak se komponenta jmenuje a jakou má verzi. Tento soubor se nazývá manifest a je mu věnována speciální kapitola 2.3.

Komponenta dále obsahuje zdrojové soubory rozdělené pro lepší orientaci do jednotlivých adresářů. Tyto složky zdrojových kódů se jmenují balíčky, anglicky **package**. Bezpečnost OSGI kontroluje, zda každý, kdo přistupuje k našemu zdrojovému kódu, má správná práva. Ta jsou opět specifikována v manifestu. Jedná se o balíčky, které používáme, neboli importujeme, anglicky tedy **imported packages,** a dále ty, které poskytujeme navenek, neboli exportujeme, anglicky tedy **exported packages**.

Komponentová aplikace používající Equinox má tyto komponenty uloženy v jar souborech, kde se nachází přeložené zdrojové soubory a manifest. Volitelně se zde mohou nacházet i zdrojové soubory. Pokud aplikace není open source, většinou je možné použít nějaký šifrovací algoritmus znemožňující jednoduchou dekompilaci přeložených souborů.

Každá komponenta má svůj životní cyklus, který je zobrazen na stavovém diagramu, viz Obrázek 2.



Obrázek 2 - Životní cyklus OSGI komponenty, bundlu. Zdroj [1]

Pokud bychom chtěli začít používat komponentu, musíme ji nejprve nainstalovat do frameworku. To je možné pomocí metody `installBundle` `BundleContextu`, kterou poskytuje LifeCycle vrstva OSGI. Při instalaci je potřeba ověřit, že Framework již obsahuje všechny požadované komponenty. Pokud je tomu tak, je komponenta přesunuta do stavu **resolved**; pokud ne, je vyhozena výjimka a zůstává ve stavu **installed**. Ze stavu **resolved** je možné přejít do stavu **starting**. Diagram stavů **starting**, **active** a **stopping** je nakreslen záměrně mimo instalaci a odinstalaci bundlu, protože tuto funkcionalitu má na starosti třída **Activator**. Tuto třídu si můžeme sami napsat, a tak můžeme ovlivnit, jaká činnost se má vykonat při zavedení komponenty do frameworku. Více o **Activátorech** naleznete v části 2.3 zabývající se manifestem, protože zde je **Activator** deklarován. Samozřejmě bundle můžeme také odinstalovat a tím se dostane do stavu **uninstalled**.

2.2.3 Služby

Vrstva služeb umožňuje lepší sdílení prostředků mezi komponentami. Ve světě objektového programování se pro tento účel využívá návrhový vzor Factory. Tento návrhový vzor má mnoho výhod, ale také několik zásadních nevýhod. Factory je pasivní, nemůže nijak oznámit, že je k dispozici, a volající nemůže procházet možné implementace a vybrat si tu, která mu nejvíce vyhovuje. Implementace ovšem také není dynamická – jakmile ji Factory jednou poskytne, nemůže si ji vzít zpět nebo ji změnit. Tyto neduhy se snaží OSGI vyřešit vrstvou služeb a jejich registrací.

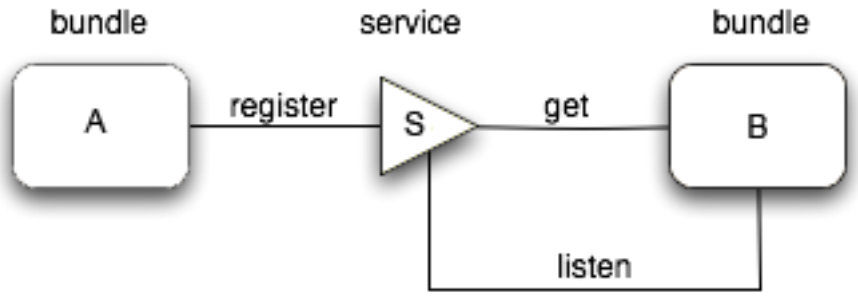
Komponenta může zaregistrovat službu ve frameworku popsanou jedním nebo více rozhraními. Ostatní bundly mohou přistoupit k repozitáři služeb a procházet seznam služeb poskytujících určité rozhraní.

Nechť existují bundly **A**, **B** a služba **S**. Bundle **A** poskytuje službu **S** tak, že ji zaregistruje ve frameworku. Bundle **B** chce používat službu **S**, může to udělat dvěma způsoby:

1. Může požádat framework o službu poskytující rozhraní služby S, a následně dostat seznam všech služeb poskytujících dané rozhraní. Z toho seznamu si pak může vybrat nejvhodnější rozhraní.

2. Může čekat, než se objeví kýžená služba. Framework následně notifikuje komponentu, až bude služba k dispozici.

Oba přístupy demonstruje Obrázek 3.



Obrázek 3 - Služby, registrace, vyzvednutí služby, čekání na notifikaci, že služba je k dispozici. [2]

OSGI poskytuje možnost registrovat nebo odregistrovat služby pod stejným rozhraním. Co se stane, pokud chceme získat určitou službu, a ve frameworku pro ni existuje více implementací? Odpovědí jsou tzv. property soubory, které přesně popisují zaregistrovanou službu. OSGI framework poskytuje filtrovací jazyk, který využívá těchto souborů k umožnění výběru služby, která nás zajímá. Vzhledem k tomu, že v této práci bude použito rozšíření OSGI o funkce aspektově orientovaného programování pomocí Spring Gemini Blueprint projektu, je další popisování těchto property souborů nad rámec vymezeného tématu. V případě zájmu lze více o tomto tématu nalézt v odkazované literatuře. [1]

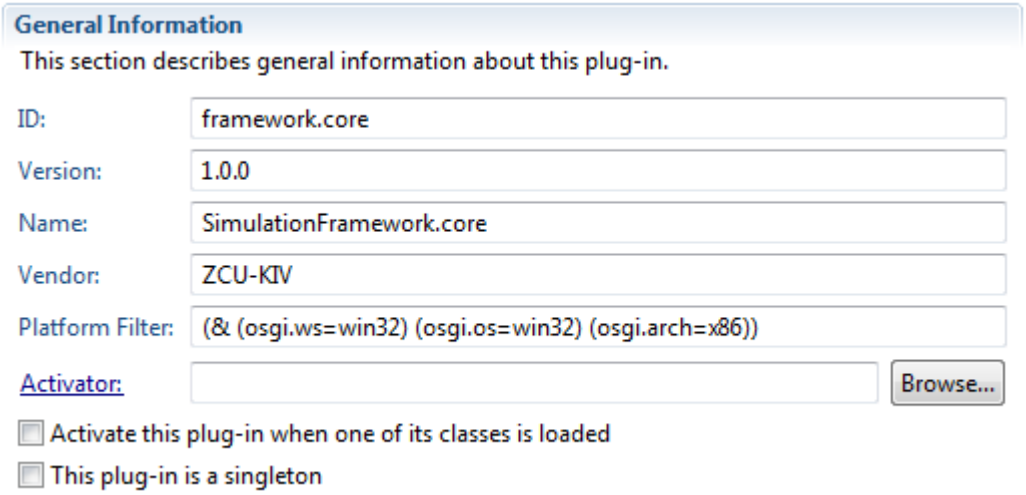
2.3 Manifest a Activator

Jak již bylo řečeno v předchozí kapitole, důležitou součástí komponenty je její popis, tedy její manifest. Jedná se o soubor Manifest.mf, který se nachází ve složce META-INF, což je soubor nutný pro zavedení komponenty do frameworku. Manifest se skládá z hlavičky a těla, přičemž hlavička obsahuje nutné parametry **Bundle-SymbolicName** a **Bundle-Version**.Tyto dva údaje jednoznačně identifikují komponentu, proto dvě komponenty se stejným symbolickým jménem nemohou být najednou ve frameworku. Pokud se pokusíme nainstalovat dvě komponenty se stejným jménem, je vyhozena výjimka. Základní manifest může vypadat například takto:

```
Bundle-SymbolicName: framework.core
Bundle-Version: 1.0.0
```

Ukázka 1 – Nutná součást hlavičky manifestu.

Tento manifest obsahuje opravdu jen to nejnutnější z hlavičky manifestu. Pro psaní tohoto souboru existuje rozšíření do Eclipse, které poskytuje grafické rozhraní pro snadnou správu manifestu. Zde je programátor upozorněn, pokud neuvedl nutnou informaci o komponentě nebo jeho manifest obsahuje chybu. Dále zde velice snadno může upravovat informace o dané komponentě. Na obrázku 4 můžete vidět, jak vypadá nastavování základních údajů o bundlu.



Obrázek 4 - Grafické uživatelské rozhraní pro nastavování základních informací o komponentě. Eclipse Juno

Nyní si rozebereme jednotlivé položky tohoto uživatelského rozhraní

ID – nutný údaj, pod tímto jménem bude komponenta známa ve frameworku, a pomocí tohoto údaje můžeme z frameworku dostat referenci na komponentu.

Version – druhý nutný údaj, popisuje verzi komponenty. Ostatní bundly mohou vyžadovat bundle pouze v určité verzi. Verze jsou 3 čísla, která mají následující sémantiku: První číslo značí tzv. major verzi. Pokud změníme toto číslo, značíme tím, že se zásadně změnila struktura komponenty, a tudíž není zpětně kompatibilní s předchozí major verzí. Druhé číslo je nazýváno minor verzí a značí lehkou nekompatibilitu, API by se nemělo zásadně změnit. Třetí číslo značí micro verzi, pokud se změní, musí platit, že komponenta API je zpětně kompatibilní s předchozí verzí. Zvýšení této verze se obvykle používá u opravení chyb. Dále může verze obsahovat nepovinný identifikátor tzv. qualifier. Ten obsahuje čas, kdy byla komponenta naposledy aktualizována. Např. tedy 201303031200, což znamená, že komponenta byla naposledy aktualizována 3. 3. 2013 v 12:00.

Name – nepovinný parametr značící jméno komponenty. Může obsahovat libovolný textový řetězec.

Vendor – značí identifikaci firmy nebo subjektu, který vyrobil tuto komponentu. V tomto případě je vendorem Západočeská Univerzita, konkrétně katedra informatiky a výpočetní techniky. V praxi zde bývá celý název firmy např. Bosch Automotive a.s.

Platform Filter – zde je definováno LDAP filtrem, na jaké platformě je schopna tato komponenta běžet. Obsahuje následující platné parametry:

- **osgi.nl** –nastavení jazyka platformy
- **osgi.os** –operační systém platformy
- **osgi.arch** –architektura platformy
- **osgi.ws** – systém použitý pro vytváření grafického rozhraní

V našem příkladu může komponenta běžet pouze pod operačním systémem Windows s procesorem s x86 architekturou.

Activator – zde je uvedena cesta k Activatoru komponenty. Activator je speciální třída, která je zavolána při zavedení bundlu do frameworku. Activator je třída, která musí implementovat rozhraní **BundleActivator**, které obsahuje metody `start` a `stop`. V metodě `start` obvykle chceme získat instance služeb, které potřebujeme, v metodě `stop` většinou tyto služby uvolňujeme, aby mohly být dealokovány Garbage Collectorem. Metody `start` a `stop` obsahují bundle context, ve kterém jsou uloženy všechny ostatní komponenty, a skrz který můžeme instalovat nové komponenty. Příklad jednoduchého Activatoru najdete v Ukázce 2 na další stránce.

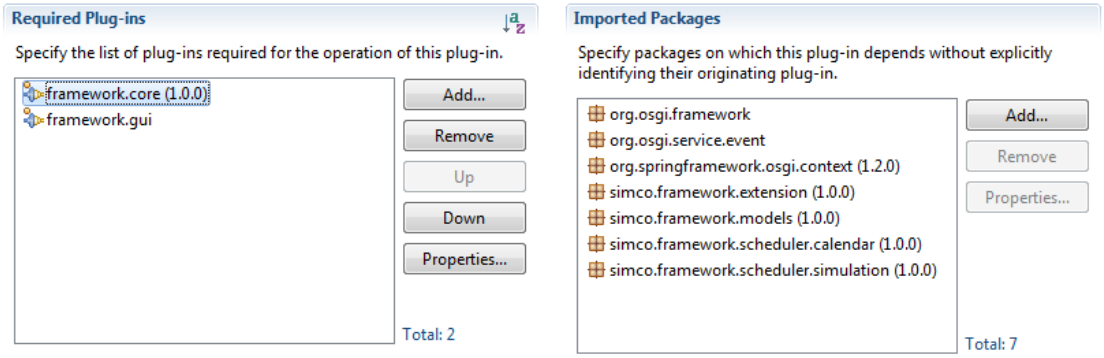
```
public class Activator implements BundleActivator
{
	private static BundleContext context;
	public static BundleContext getContext()
	{
		return context;
	}

	@Override
	public void start(BundleContext bundleContext) throws Exception
	{
		Activator.context = bundleContext;
	}

	@Override
	public void stop(BundleContext bundleContext) throws Exception
	{
		Activator.context = null;
	}
}
```

Ukázka 2 - Implementace obyčejného Activatoru

V této práci aktivátory nebudou použity, protože jejich funkci zastane framework Spring.

2.3.1 Závislosti

V manifestu můžeme kromě základních informací definovat závislosti mezi jednotlivými komponentami. A to jak bundly a balíčky, které vyžadujeme – required bundles a imported packages, tak ty balíčky, které poskytujeme – exported packages. Jako v případě hlavičky manifestu, i zde můžeme využít funkcionalitu Eclipse a speciální uživatelské rozhraní. Pro komponentu použitou v předchozím příkladu vypadá následovně:

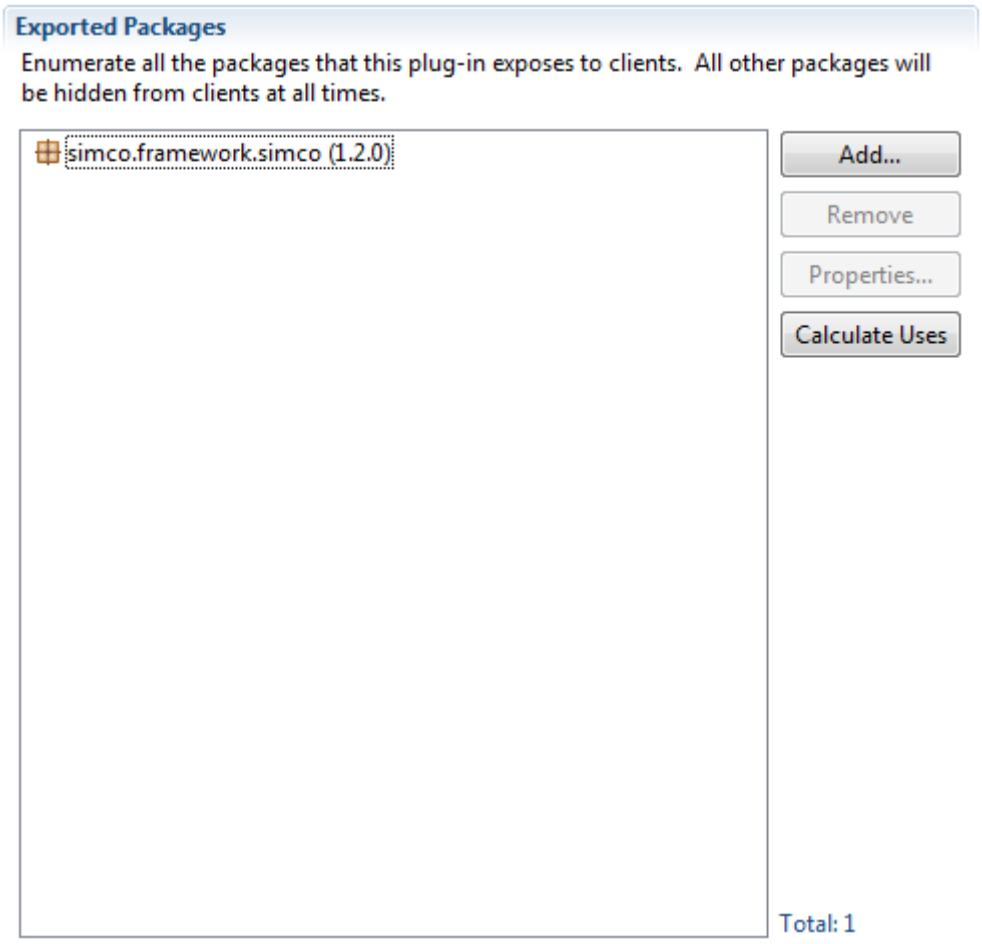


Obrázek 5 - Editor závislostí Eclipse Juno

V levé části máme specifikováno, jaké komponenty vyžadujeme k běhu. Pokud nebudou nalezeny, bundle se nikdy nedostane do stavu **resolved**. Tlačítkem add můžeme přidávat závislosti. Závislost může být volitelná - optional, v tom případě komponenta může být spuštěna bez přítomnosti požadovaného bundlu. Dále můžeme specifikovat, jakou přesnou verzi komponenty požadujeme (to značí číslo uvedené v závorce u framework.core). Můžeme zadat i rozsah verzí. Pokud není nalezena komponenta s požadovanou verzí, závislost není splněna.

V pravé části je varianta, kterou většina lidí v praxi příliš nedoporučuje, jedná se o import balíčků. Jak bude uvedeno dále, komponenta může poskytovat svoje zdrojové kódy navenek jako balíčky. Zde specifikujeme, jaké balíčky budeme používat. Framework poté hledá v seznamu exportovaných balíčků a snaží se požadovanou závislost splnit. Pokud nenalezne žádaný balíček, závislost není splněna a komponenta nemůže být spuštěna. Nevýhodou tohoto principu je horší orientace v tom, jaké komponenty vlastně požadujeme. Programátor musí balíčky hledat v komponentách, místo toho, aby hledal pouhý jar soubor. Výhodou je možnost použít pouze část komponenty. I zde platí, že můžeme specifikovat verzi balíčku, stejně jako v předchozím případě u komponenty.

Zatím bylo nastíněné jak deklarovat komponenty, na kterých je bundle závislý, nyní je čas demonstrovat, jak sdělit frameworku, že určitý kód poskytujeme ostatním balíčkům. K tomu slouží již zmíněná část manifestu – Exported Packages. Pokud někdo definuje závislost na našem balíčku, dostane se pouze ke kódu, který je označen jako viditelný, tedy exportovaný. Exportovat můžeme pouze celou složku, balíček, anglicky package. Pokud je tedy úkolem vytvořit falešnou komponentu, pak jediné, co nás bude zajímat, budou balíčky volně přístupně ostatním komponentám, tedy právě tyto exported packages. Jako v předchozím případě, i zde existuje pomoc v podobě uživatelského rozhraní Eclipse, jmenuje se Exported Packages viz Obrázek 6.



Obrázek 6 - Poskytování zdrojového kódu ostatním balíčkům. Exported Packages

Opět je zde tlačítko add, přes které můžeme přidat balíček zdrojových kódu. Můžeme zde také nastavit verzi balíčku, který exportujeme. Ta bude porovnána s required version uvedenou v komponentě, která chce používat námi poskytnutou funkcionalitu.

2.3.2 Ukázka kompletního manifestu používaného ve frameworku SimCo

Na následujícím příkladu bych rád ukázal, jak vypadá manifest vytvořený uživatelským rozhraním Eclipse. Jedná se o textový soubor, který je nastavený přesně podle předchozích příkladů. Je zde vidět i nepovinná položka hlavičky, popis komponenty - **Bundle-Description**.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Simco Plug-in
Bundle-SymbolicName: framework.simco
Bundle-Description: This bundle actions in SIMULATION component and INTERMEDIATE
component.
Bundle-Version: 1.0.0
Import-Package: simco.framework.scheduler.calendar;version="1.0.0",
 simco.framework.scheduler.simulation;version="1.0.0",
 simco.framework.models;version="1.0.0",
 simco.framework.extension;version="1.0.0",
 org.osgi.framework,
 org.springframework.osgi.context;version="1.2.0",
 org.osgi.service.event
Export-Package: simco.framework.simco;version="1.2.0";
  uses:="simco.framework.models,
   simco.framework.extension,
   org.springframework.osgi.context,
   simco.framework.scheduler.simulation,
   simco.framework.scheduler.calendar,
   org.osgi.service.event,
   org.osgi.framework"
Require-Bundle: framework.gui,
 framework.core;bundle-version="1.0.0"
```

Ukázka 3 - Ukázka manifestu pro framework.simco

3 Gemini Blueprint, Spring DM

Gemini Blueprint přináší do světa OSGI prvky aspektově orientovaného programování. Tento projekt začal v roce 2006 jako Spring OSGI projekt. Později byl přejmenován na Spring DM projekt, kde DM značí Dynamic Bundles. V roce 2009 se Spring DM sloučil s Gemini projektem a tak vznikl Gemini Blueprint, který je nyní pod kontrolou nadace Eclipse.

Díky této nadstavbě lze používat veškeré výhody (i nevýhody) frameworku Spring v prostředí OSGI. Díky Gemini Blueprint můžeme v OSGI použít následující prvky:

- **IOC** – Inversion of Control, označované také jako Dependency Injection
- **Spring Resource rozšíření**
- **AOP** – kompletní podpora pro aspektově orientované programování

Nejzajímavější z hlediska použití je pro tuto práci IOC, protože tento princip využívá Framework SimCo, tudíž musí být použit i v této práci, i když zdaleka ne v takovém množství.

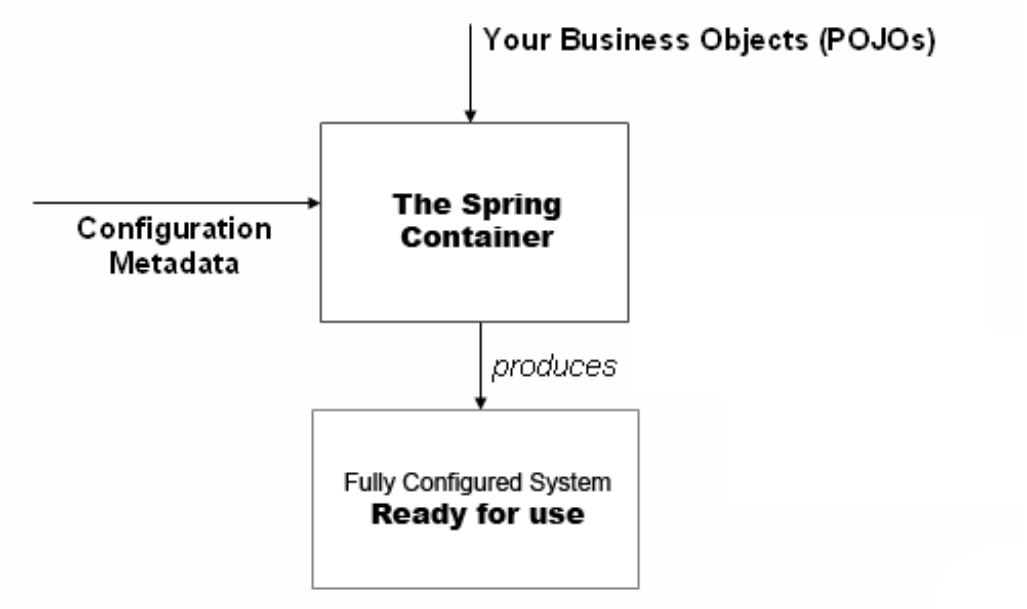
3.1 IOC Inversion of Control a DI Dependency Injection

Definice IOC podle Ralpha E. Johnsona a Briana Foota z roku 1988:

Důležitou charakteristikou frameworku je, že metody definované uživatelem pro nastavení frameworku budou volány zevnitř, místo toho aby je volal uživatel ve svém kódu. Tím pádem framework často přebírá úlohu hlavního programu v koordinování aktivity aplikace. Toto přebrání kontroly dává frameworku možnost fungovat jako rozšířitelná kostra aplikace. [3]

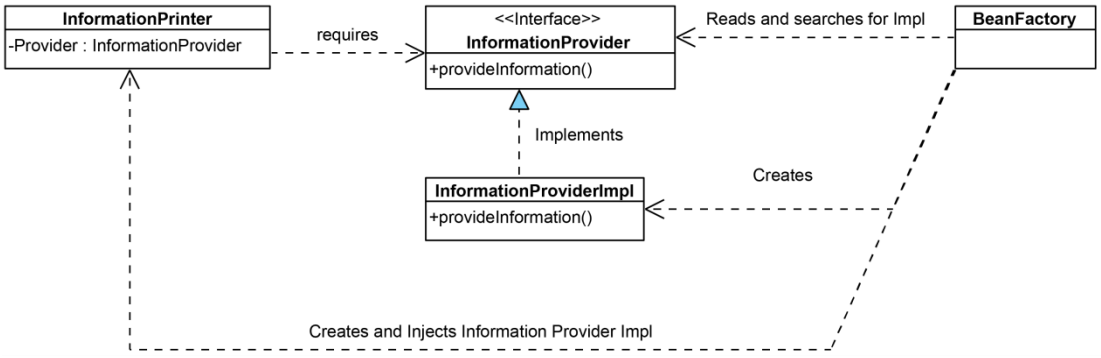
Podle této definice je hlavní úkolem programovací techniky Inversion of Control přesunout vytvoření aplikačního modelu z kódu uživatele do frameworku. Obvykle k tomuto účelu slouží prostředník, který vytvoří instance požadovaných objektů, a ty poté „vstříkne“ (anglicky inject) do těch míst kódu, které definují závislost na daném objektu. Realizaci tohoto „vstříknutí“ se říká Dependency Injection, zkratka DI. Prostředník logicky musí vědět o závislostech mezi objekty, obvykle je získá z anotací ve zdrojovém kódu nebo ze speciálních konfiguračních souborů. Při startu aplikace framework načte všechny konfigurační soubory nebo anotace a vytvoří z nich strom závislostí, aby mohla být aplikace zkompilována. Strom prochází od listů ke kořenu a postupně vytváří nové instance jednotlivých objektů.

Spring používá pro tento účel **ApplicationContext**, ve kterém jsou uloženy jednotlivé objekty. V terminologii springu jsou nazývány **bean**, v češtině se používá slangový výraz „beany“. **ApplicationContext** je vytvořen třídou **BeanFactory**, která se stará o parsování konfiguračních souborů nebo anotací, vytváří podle získaných dat instance jednotlivých objektů a poskytuje jim podle definovaných závislostí odkazy na vyžadované instance. Úložiště těchto objektů se nazývá Spring Container. Princip vytvoření takového úložiště demonstruje Obrázek 7.



Obrázek 7 - Spuštění aplikace používající Spring framework. Zdroj : [4]

Tento přístup má několik výhod. Největší spočívá v tom, že závislosti jsou definovány vně programu. Mějme jednoduchou funkcionalitu definovanou rozhraním. Představme si následující problém, který popisuje Obrázek 8



Obrázek 8 Jednoduchý diagram tříd

Máme tedy třídu, která potřebuje jednu instanci daného rozhraní. Klasickou programovací technikou je poskytnout v hlavní metodě instanci **InformationPrinteru** odkaz na instanci **InformationProviderImpl**. V zásadě s tím nemusí být žádný problém, dokud je kód ve stejných balíčcích nebo komponentách. Mějme komponentovou aplikaci. Definice rozhraní je v komponentě A, třída **InformationPrinter** je definována v komponentě B a třída **InformationProviderImpl** je definována v komponentě C. Komponenta B je závislá na A i na C. S použitím Springu, který zde znázorňuje třída **BeanFactory**, tato závislost odpadá. Nemusíme tedy deklarovat závislost na konkrétní implementaci, protože framework ji sám vytvoří a poskytne komponentě, která ji vyžaduje.

Aby tato funkcionalita byla zajištěna, musíme však napsat konfigurační soubory. Zde přichází hlavní nevýhoda Springu. Pokud je projekt opravdu veliký, počítejme zhruba přes 80 tisíc tříd, každá třída, pokud chce využívat benefitů Springu, by měla mít svoje místo v konfiguračním souboru. Pokud bychom měli jeden soubor obsahující celou aplikaci, údržba by se stala opravdovou noční můrou. Tento problém má samozřejmě

několik řešení. Triviálním řešením je rozdělení souboru na dílčí celky. Lepším řešením je použít spolu se Springem OSGI a rozdělit tak konfigurační celky do jednotlivých komponent, kde každá komponenta obsahuje jeden konfigurační soubor Springu, o kterém pojednává kapitola 3.2.

Samozřejmě je nutné dávat pozor na to, aby velikost komponenty zůstala únosných mezích. Pokud použijeme analogii s osobním vozidlem, jediná komponenta by neměla zastupovat celý vůz. Stejně jako v každém programovacím stylu, je i v komponentově orientované architektuře velice důležitý refaktoring.

3.2 Konfigurace Springu pro OSGI

Od verze 2.0 umožňuje Spring konfiguraci také pomocí anotací. V této práci však používáme standardní xml konfigurační soubory. Každá komponenta by měla mít právě jeden konfigurační soubor, je ale možné do nich vkládat dílčí konfigurační soubory. V této diplomové diplomové práci operujeme právě s jedním konfiguračním souborem, který bude umístěn ve stejném adresáři jako manifest OSGI, tedy v adresáři META-INF, v podadresáři spring.

Konfigurační soubor, jako každý správný xml soubor, obsahuju hlavičku a značku beans. Ten zahrnuje definici jmenných prostorů, anglicky namespace. Existuje plugin do Eclipse, který poskytuje kontextovou nápovědu, čímž se psaní konfiguračních souborů výrazně zjednodušuje. Do xml souboru stačí správně naimportovat dané jmenné prostory, a pak je možné přistupovat k veškerým kontextovým nápovědám, které významně usnadňují práci. Základní konfigurační soubor demonstruje Ukázka 4.

```
<?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">

       <bean id="SpringMocker"
             class="org.kiv.zcu.cz.sandcrew.mocking.SpringMocker"
             init-method="execute">
       </bean>

</beans>
```

Ukázka 4 - Jednoduchý Spring konfigurační soubor.

Jak je vidět z předchozího příkladu, v tagu beans může být libovolné množství tagů bean. Ty označují instanci, která bude vytvořena. V tomto případě Spring vytvoří instanci třídy SpringMocker. Pokud bychom chtěli konstruktoru předat nějaké parametry, můžeme tak učinit vnořeným tagem constructor-arg. Tato instance bude známa pod id SpringMocker, a poté, co bude úspěšně vytvořena, se zavolá její metoda execute. Pokud bychom chtěli předat konkrétní parametry konstruktoru, vypadalo by to následovně:

```
<bean id="SpringMocker" class="org.kiv.zcu.cz.sandcrew.mocking.SpringMocker"
      init-method="execute">
            <constructor-arg value="RandomString"/>
            <constructor-arg value="123"/>
            <constructor-arg ref="SpringMocker"/>
</bean>
```

Ukázka 5 - Vytvoření instance s parametry

V tomto případě by dále byla vyrobena instance pomocí konstruktoru, který by akceptoval String, Integer a instanci SpringMockeru. Z příkladu je zřejmé, že konstruktoru jde předat jakýkoliv parametr, ať už primitivního typu, kdy jeho hodnotu předáme rovnou jako text, nebo komplexního typu, kdy vytvoříme instanci a pomocí klíčového slova ref ji předáme kýženému objektu. Tato instance ovšem může být definována v libovolném balíčku. Pokud Springu nenastavíme klíčové slovo prototype, vytvoří vždy pod jedním jménem jednu instanci. Díky tomuto řešení se programátor nemusí starat, kde je deklarována instance implementace rozhraní, kterou hodlá používat. Stačí mu vědět, že někde bude zaregistrován příslušný objekt pod kýženým jménem. Tento objekt může být použit na více místech, ve více třídách nebo jejich instancích. Programátor však stále musí importovat rozhraní, aby byl překladač schopný program přeložit. Díky tomuto přístupu již dále není nutné používat návrhový vzor Singleton, protože Spring v základním nastavení Singletony vytváří.

3.2.1 Možnosti konfigurace OSGI

Spring v kombinaci s OSGI umožňuje konfiguraci vztahů mezi jednotlivými komponentami pomocí služeb. Službu je nejprve potřeba v komponentě zaregistrovat. Následně je třeba v komponentě, která ji hodlá používat, deklarovat referenci na požadovanou službu. Příklad deklarace OSGI služby zobrazuje následující ukázka konfiguračního souboru.

```
<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" />
```

Ukázka 6 - Registrace OSGI služby

Na předchozím příkladu je vidět, že je nejprve potřeba vytvořit instanci služby, a tu poté zaregistrovat. Pro registraci je potřeba zadat rozhraní, které služba poskytuje, a referenci na instanci služby. Konzument služby, tedy komponenta využívající službu, by poté měla následující konfiguraci:

```
<osgi:reference id="coreService" interface="simco.framework.core.ICore"/>
```

Ukázka 7 - Konfigurace konzumenta služby

S touto konfigurací je možné pomocí Dependency Injection vložit do libovolného objektu implementaci služby z libovolné komponenty. Ta pak může používat instanci služby, aniž by věděla, odkud služba je, a jak se k ní dostala. Službu můžeme předat jako parametr konstruktoru, jako property (v tomto případě se zavolá setter daného objektu), a nebo o ní stále můžeme požádat OSGI, čímž bychom se ale připravili o všechny benefity Springu.

```
<osgi:service id="CoreServiceOsgi" ref="CoreService"
                    interface="simco.framework.core.ICore" />
<!-- Constructor approach -->
<bean id="SpringMocker"
      class="org.kiv.zcu.cz.sandcrew.mocking.SpringMocker"
      init-method="execute">
      <constructor-arg ref="CoreServiceOsgi"/>
</bean>
<!-- Property approach -->
<bean id="AnotherMocker" class="org.kiv.zcu.cz.sandcrew.mocking.SpringMocker">
      <property name="coreService" ref="CoreServiceOsgi"/>
</bean>
<!-- Factory approach -->
<bean id="AnotherFactoryMocker" factory-bean="SomeFactory"
      factory-method="FactoryMocker">
      <constructor-arg ref="CoreServiceOsgi"/>
</bean>
```

Ukázka 8 - Možné postupy pro vytvoření instance objektu s implementací OSGI služby pomocí Springu

Ukázka 8 demonstruje možnosti konfigurace nové instance používající službu definovanou v jiné či stejné komponentě. Je zde použit ještě jeden způsob, jak vytvářet objekty pomocí Springu, a to je použití vlastní implementace návrhového vzoru Factory. Spring potom zavolá metodu registrovanou v atributu factory-method. Tento přístup bude podrobněji popsán v kapitole o transparentní proxy (7.8), kde bude tento princip použit pro realizaci vyžadované funkcionality. Aby Spring mohl vytvořit instanci přes factory z jiné komponenty, musí k ní mít přístup. Balíček, kde se nachází zdrojový kód factory, musí být exportován.

4 SimCo

SimCo je projekt Katedry informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity v Plzni. Zkratka SimCo pochází z anglického **Si**mulation **o**f **Co**mponent, česky simulace komponent. Jádro aplikace vypracoval jako diplomovou práci Ing. Tomáš Kabíček, kde SimCo definoval jako „*framework, který umožňuje simulaci některých komponent*“ [4].

Tento framework používá OSGI implementaci Equinox v kombinaci se Spring DM. Cílem celé aplikace je umožnit simulaci na principu diskrétní událostní simulace. [4]. Jakákoliv aplikace z principu neběží spojitě, ale diskrétně, kdy nejmenší jednotkou času je tik procesoru. SimCo vnímá aplikaci na vyšší vrstvě abstrakce. Nejmenší časová jednotka pro SimCo je volání metody. To je umožněno tím, že framework pohlíží na objekty jako na black-box, nezajímá jej tedy, jak jsou jednotlivá volání realizována, ale pouze jejich návratová hodnota. Za událost v terminologii SimCo frameworku budeme považovat volání metody v programovacím jazyce Java.

Nyní je potřeba definovat, jaký je rozdíl mezi jednotkovým testováním (Unit testy) a aplikačním testováním.

Aplikační testování by se dalo chápat jako množina scénářů, které vymyslí tester podle specifikace požadavků nebo jiného dokumentu. Testování je poté provádění stále stejných kroků, u kterých je očekáváno, že přinesou stále stejný, negativní nebo pozitivní, výsledek. V ideálním případě jsou testy automatizované. U aplikačního testování je důležité, že se jedná o pohled **shora**, tedy z uživatelského rozhraní, a většinou běží nad **kompletní** aplikací.

Jednotkové testování je naopak prací programátora. Ten také vytváří scénáře, které bývají automatizovány. Jedná se však o pohled **zdola**, protože programátor testuje aplikaci jednotlivými voláními kódu. Test většinou běží pouze nad **jednou** jednotkou testu, zato ji obvykle testuje důkladněji.

Ambicí simulačního frameworku SimCo je otestovat program s využitím benefitů obou přístupů. To je umožněno konfiguračním souborem, který obsahuje scénář celého testovacího běhu. V tomto konfiguračním souboru je uloženo, které metody se zavolají. Může se tak jednat jak o metody grafického uživatelského rozhraní, tak o metody na úrovni databáze atd. Konfigurační soubor dále obsahuje souhrn komponent, které budou použity pro simulační běh. Tyto komponenty mohou, ale nemusí, být všechny, které program obsahuje.

SimCo může aplikaci testovat jak **shora,** tak **zdola.** Může také testovat **kompletní** aplikaci nebo pouze tu **část**, která nás zajímá. Navíc se snaží přidat roli náhody tak, aby testovací scénář nebyl pokaždé stejný. K tomu využívá generování náhodných čísel a statistická rozdělení.

4.1 Diskrétní událostní simulace

Simulované systémy lze podle jejich vlastností rozdělit do dvou skupin. První skupinu tvoří spojité systémy, jejichž parametry se mění v čase spojitě. Naopak diskrétní systémy jsou systémy, jejichž parametry se nemění spojitě, ale pouze v diskrétní množině bodů časové osy. S diskrétními systémy je spojen pojem událost. Tímto pojmem míníme každou změnu některé z uvažovaných vlastností v konkrétním časovém bodě, dále odebrání nebo přidání prvku ze systému nebo do něj. Určitá událost může být příčinou dalších událostí v budoucích časových bodech [5].

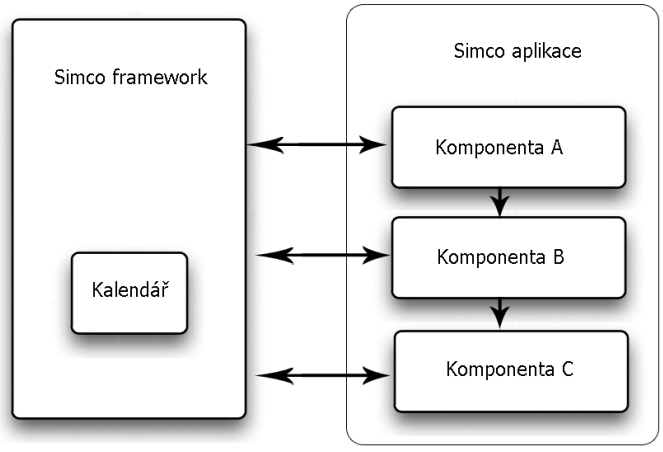
Běh aplikace by se dal chápat jako množina událostí s daným časem. Události obecně netrvají stejně dlouho. Pokud se odehrají, jejich výskyt může, ale nemusí být pravidelný, což záleží na náhodné veličině přidělené události. Z těchto důvodů je potřeba mít seznam událostí, kde budou všechny uloženy. Tomuto seznamu budeme říkat kalendář, který slouží jako hlavní kontrolní prvek simulace, vyvolává události, ví o všech událostech, které se staly. Díky kalendáři můžeme simulaci zastavit nebo krokovat.

Abychom mohli používat události, potřebujeme hodiny. Tyto hodiny neměří reálný čas, ale čas simulační, ve kterém nastanou jednotlivé události. Po vykonání události se přesuneme v kalendáři na další nebo předchozí událost. Pokud je potřeba, smažeme ji z kalendáře a hodiny nastavíme na simulační čas nové události.

Hlavní smyčka SimCo frameworku vypadá následovně:

1) Nastavit simulační čas na první nevykonanou událost.
2) Vykonat událost (a případně odstranit ze seznamu).
3) Aktualizovat statistiky a provést ostatní potřebné úkony [4].

Každá událost bude mít přiřazenou činnost v SimCo frameworku, volání metody mezi komponentami. Pokud vznikne takové volání, které není v Kalendáři, framework jej odchytí, vytvoří pro něj událost a vloží ji do kalendáře. Tak je zajištěno, že kalendář ví o veškeré interakci, která se děje mezi komponentami aplikace. K tomu používá tři druhy komponent, které budou rozebrány v kapitole 4.2. Ukázku komunikace můžete vidět na obrázku Obrázek 9.



Obrázek 9 Kalendář a komunikace s komponentami. Zdroj [4]

4.2 Komponenty

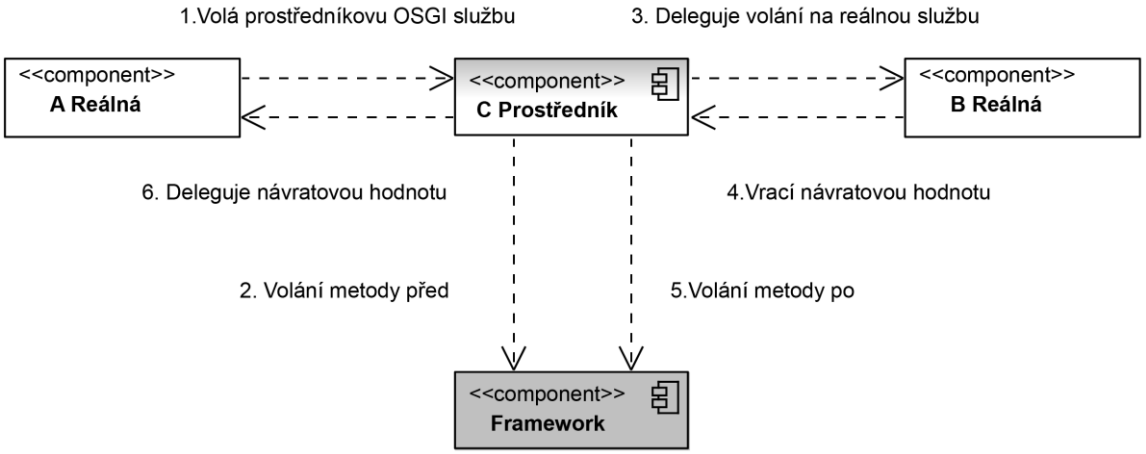
Simco používá a rozlišuje čtyři základní druhy komponent:

a) Reálná
b) Simulační
c) Prostředník
d) Framework

Všechny tyto komponenty jsou Equinox bundly využívající Spring. Komponenty frameworku obstarávají komunikaci mezi ostatními typy komponent. Řídí celou simulaci a mají přehled o tom, jakého typu jsou ostatní komponenty v aplikaci. Cílem aplikace je, aby reálné komponenty nerozeznaly, že běží v simulačním prostředí, aby nemusely být nijak upraveny pro běh v simulaci a vykazovaly stejné chování a vlastnosti, jako když běží mimo framework.

Aby SimCo bylo schopné zachytit volání mezi dvěma reálnými komponentami, používá komponentu typu prostředník, kterou je potřeba vložit mezi každé dvě reálné komponenty tak, aby každé volání z komponenty **A** do komponenty **B** šlo přes prostředníka, komponentu **C**.

Prakticky to znamená, že komponenta **C** bude nabízet stejné služby jako komponenta **B**. Požadavky komponenty **A** bude přeposílat dále na komponentu **B**, přičemž může po přijetí požadavku a před přeposláním vykonat libovolnou akci (např. oznámit frameworku, že komponenta **A** požaduje určitou službu komponenty **B**). Komponenta **C** dále deleguje volání na komponentu **B** a čeká na její odpověď. Po jejím přijetí může opět vykonat libovolnou akci a výsledek přepošle komponentě **A**. Tento princip je znázorněn na následujícím diagramu:



Obrázek 10 - Zachycení komunikace mezi dvěma komponentami

Posledním typem komponent jsou komponenty simulované, což jsou vhodně vytvořené komponenty poskytující stejné služby jako komponenty reálné, které jsou jejich předlohou. Rozdíl mezi reálnou a simulační komponentou je v tom, že reálná komponenta nad daty používá nějaký algoritmus nebo je její návratová hodnota výsledkem určitého měření. Simulační komponenta se snaží imitovat chování reálné komponenty tak, aby to bylo nějakým způsobem výhodné, ať z důvodu úspory času, nebo nedostupnosti služby, nebo neexistence implementace bundlu, nebo z jiných důvodů.

4.3 Konfigurační soubor.

Konfigurační XML soubor je jednou z nejdůležitějších částí celého Simco frameworku. Jeho hlavním obsahem je definice událostí typu REGULAR a CASUAL, čímž je vytvořen jakýsi scénář celého průběhu simulace. Proto se tomuto souboru říká *scénář* (scenario) [4].

Scénář je očekáván v kódování UTF-8 a zatím se musel psát ručně. XML soubor by se pomyslně dal rozdělit na tři díly. V prvním jsou definovány obecná nastavení aplikace, ve druhém události a v třetím komponenty. Základní konfigurační soubor tedy vypadá následovně:

```
<?xml version="1.0" encoding="UTF-8"?>
<simCoScenario>
        <appSettings>
        <!-- Zde budou nastaveni -->
        </appSettings>
        <components>
        <!-- Zde budou komponenty -->
        </components>
        <events>
        <!-- Zde budou události -->
        </events>
</simCoScenario>
```

Ukázka 9 - Ukázka prázdného konfiguračního souboru

4.3.1 Nastavení aplikace

Nastavení aplikace je uloženo v tagu **appSettings**. V současné době jediný podporovaný a parsovaný tag je **stepLenght**, který určuje, jak dlouho bude trvat krok aplikace. Zadává se v jednotkách, podporované hodnoty jsou **sec**, vteřiny, a **msec** milisekundy. Konfigurace pak vypadá následovně:

```
<appSettings>
      <stepLenght unit="msec">500</stepLenght>
</appSettings>
```

Ukázka 10 - Nastavení kroku na 500 milisekund, tedy půl vteřiny.

4.3.2 Nastavení komponent

Informace o komponentách, které bude aplikace používat, jsou uloženy v tagu **components**. Uvnitř může obsahovat libovolné množství komponent, které jsou identifikovány tagem **component**. Každá komponenta má definovaný atribut **type**, který přesně odpovídá typům komponent. Dále obsahuje dva vnořené elementy **symbolicName** a **settingsFile**. **SymbolicName** obsahuje jméno, pod kterým je bundle znám v OSGI frameworku, a musí se shodovat se stejnou informací v manifestu komponenty. Komponenta podle tohoto tagu bude hledána v `BundleContextu`. Pokud nebude nalezena, simulační scénář nebude moci být načten. Druhá informace **settingsFile** obsahuje plnou cestu ke konfiguračnímu souboru dané komponenty. Možné typy komponent jsou podle předem uvedeného dělení následující:

- SIMULATION – simulační komponenta
- REAL – reálná komponenta
- INTERMEDIATE – komponenta typu prostředník
- FRAMEWORK – komponenta frameworku

Jednoduchý konfigurační soubor s komponentami vypadá následovně:

```
<components>
   <component type="REAL">
      <symbolicName>Calculator</symbolicName>
      <settingsFile>c:/workspace/Calculator/calculatorSettings.xml</settingsFile>
      <intermediateSymbolicName>CalculatorIntermediate</intermediateSymbolicName>
   </component>
   <component type="REAL">
      <symbolicName>Presenter</symbolicName>
     <settingsFile>c:/workspace/Presenter/presenterSettings.xml</settingsFile>
   </component>
   <component type="SIMULATION">
      <symbolicName>SimcoGPS</symbolicName>
      <settingsFile>c:/workspace/SimcoGPS/gpsSettings.xml</settingsFile>
   </component>
   <component type="SIMULATION">
      <symbolicName>SimcoAccelerometer</symbolicName>
      <settingsFile>
      c:/workspace/SimcoAccelerometer/accelerometerSettings.xml
      </settingsFile>
   </component>
      <component type="INTERMEDIATE">
      <symbolicName>CalculatorIntermediate</symbolicName>
   </component>
</components>
```

Ukázka 11 - Jednoduchý konfigurační soubor s komponentami

4.3.3 Události a jejich konfigurace

Poslední dosud nezmíněnou vlastností aplikace, kterou je možné nastavit v konfiguračním souboru, jsou Události, anglicky **events**. Podle typu výskytu je dělíme na události **opakované**, regular, a **náhodné**, casual. U událostí opakovaných stačí nastavit periodu, s jakou se bude opakovat. Perioda představuje počet kroků, které simulace vykoná. Číslo 10 tedy znamená, že se událost znovu vykoná po deseti dalších událostech. Zajímavější je náhodná událost typu casual. U té je možné nastavit, podle jakého rozdělení se bude náhodná veličina generovat. V nabídce jsou zatím dvě základní – normální a exponenciální. Pro dané rozdělení je potřeba určit jeho parametry. Generované číslo má opět stejný význam jako u opakované události. Jediný rozdíl spočívá v tom, že číslo je generováno náhodně podle parametrů rozdělení. Posledním způsobem je přímo zadat, s jakou pravděpodobností událost nastane.

Událost dále obsahuje (1) **zdroj**, který ji vyvolává, tedy symbolické jméno komponenty, (2) jméno **služby**, která je volána, (3) její **metodu** a (4) **parametry**, se kterými je metoda volána. Konfigurace událostí pak může vypadat zhruba následovně:

```
<events>
   <event type="REGULAR">
      <source>SimcoAccelerometer</source>
      <serviceName>simco.application.accelerometer.IAccelerometer</serviceName>
      <methodName>setVector</methodName>
      <methodParameters>
        <parameter dataType="java.lang.Integer" value="10" />
        <parameter dataType="java.lang.Integer" value="10" />
        <parameter dataType="java.lang.Integer" value="10" />
      </methodParameters>
      <eventDetails>
         <detail key="period" val="10" />
      </eventDetails>
   </event>
   <event type="CASUAL">
      <source>SimcoAccelerometer</source>
      <serviceName>simco.application.accelerometer.IAccelerometer</serviceName>
      <methodName>getVector</methodName>
      <methodParameters/>
      <eventDetails>
         <detail key="distribution" val="gauss" />
         <detail key="mean" val="10" />
         <detail key="standardDeviation" val="1" />
      </eventDetails>
   </event>
</events>
```

Ukázka 12 - Konfigurace událostí aplikace

5 Mockovací frameworky

Mockovací frameworky vznikly díky jednotkovým testům, ty jsou jednou ze základních praktik používaných na jakémkoliv dobře řízeném velkém projektu. Díky jednotkovým testům máme jistotu, že funkcionalita pokrytá logickou jednotkou, obvykle třídou, je správně naimplementována. Jednotkové testy také poskytují schopnost regresivního testování, kdy běží, většinou přes noc, na tzv. Overnight Buildu. Programátoři ráno obvykle vědí[2], zda se jim podařilo ten den něco „rozbít". Většinou existuje požadavek na co nejvyšší pokrytí kódu automatickými testy. Zde ovšem začínají vznikat následující problémy:

- Musíme zasahovat do kódu, který testujeme – proměnná nebo metoda je private, final.
- Je nutné napsat neúnosně velký stub, přitom potřebujeme jenom jednu metodu.
- Pokud refaktorujeme kód, musíme také refaktorovat jednotkové testy.

S těmito problémy se už zajisté potýkal každý programátor, který kdy psal jednotkové testy.

5.1 MockObjects

Mock Objects není sám o sobě testovací framework, jedná se o strategii, kterou můžeme použít v případě jednotkového testování. Jde vlastně o postup, jak psát stuby a testovat výkonný kód. Z těchto doporučení a strategií vzniklo velké množství testovacích Frameworků, které generují Mock Objects místo vývojářů, takže nejsou zatíženi psaním stubů a mohou se plně soustředit přímo na testování. Mock Objects jsou tak duchovním otcem všech dále zmíněných nástrojů.

Termín Mock a mockovat zavedl Peter Marks [6]. Mock ve světě programování označuje imitaci daného objektu, třídy nebo rozhraní. Většinou vrací požadované hodnoty potřebné pro řádné otestování jiného objektu v jednotkovém testu. Cílem mockingu je obalit třídu pro jednotkový test těmito objekty, abychom se mohli soustředit pouze na testování požadované jednotky. Hlavní myšlenky, terminologii a postupy jsou uvedeny v článku [7]. Zde jsou shrnuty zkušenosti autorů s jednotkovými testy, z nichž je navrženo názvosloví a základní principy mockování, tedy testování jednotkovými testy s pomocí mocků. Základní poznatky se dají shrnout do čtyř bodů:

- Je třeba soustředit se na jednu funkcionalitu v jednom testu. K tomu je nutné vědět, co přesně testujeme a kde jsou problémy. Testovaný kód by měl s okolím komunikovat co nejjednodušeji. Při tom ale mohou vznikat problémy při nastavení okolí, nebo to dokonce není možné z důvodu zapouzdření [7].

- Zkušenost je, že s použitím Mock Objects získáme silnější testy a lepší strukturu jak testovaného programu, tak testů. Testy napsané pomocí této strategie mají daný formát, který dává vývojovému týmu společný základ a konvenci [7].

[2] Pokud se podívají na výsledky Overnight Buildu. Existují projekty, kde jsou tyto výsledky ignorovány, dokud není situace kritická.

- Věříme, že kód by měl být napsán tak, aby byl snadno testovatelný. Mock Objects je dobrá strategie, jak toho dosáhnout [7].
- Zjistili jsme, že při refaktorování kódu nám Mock Objects dokáží pomoci [7].

Jak je vidět tento článek poukazuje na všechny palčivé problémy jednotkového testování uvedené na předchozí straně. A nabízí řešení pomocí následující filozofie.

- Mock Object nesmí volat další Mock Object, zřetězení je tedy zakázáno [7].
- Pokud je psaní Mock Objectů příliš složité, autor testovaného kódu by měl začít přemýšlet o refaktoringu [7].
- Izolací testovaného objektu od ostatních Mocků zajistíme, že testujeme opravdu to, co chceme. Této izolace docílíme tak, že místo výkonného kódu vložíme do testovaného objektu Mock pomocí konstruktoru. Tuto techniku nazýváme endotesting, protože takto testujeme komponenty zevnitř [7].

5.1.1 Použití

1. Nastavit očekávané hodnoty
2. Nastavit počet volání dané metody, například metoda bude zavolána jednou, dvakrát …
3. Pustit produkční kód.
4. Ověřit, že očekávaná hodnota se rovná aktuální, že počet volání se rovná očekávanému.

5.1.2 Příklad

Zmiňovanou strategii můžeme ukázat na následujícím příkladu. Pokud používáme strategii Mock Objects musíme nejprve vytvořit třídu `MockWidget`, která bude nám vyhovující implementací nějakého `Widgetu`. Budeme očekávat, že dostane odpověď `DEVICE_READY`. Tímto je splněn bod číslo jedna.

Očekáváme, že metoda bude zavolána pouze jednou, nastavíme tedy počet volání na jedna. V těle Mocku musí být řešeno, kolikrát je metoda zavolána. Je-li to vícekrát než očekávaný počet Mock skončí výjimkou, například `AssertionFailedException`.

Dále pustíme produkční kód. V tomto příkladu třída `Application` pošle registrační kód našemu `MockWidgetu`.

Nakonec zavoláme metodu `verify`, která zkontroluje, zda všechny metody byly zavolány přesně tolikrát, kolikrát jsme chtěli, a zda obdržené hodnoty odpovídají našim očekávaným. Kód mocku demonstruje Ukázka 13 a kód testu Ukázka 14.

```
public void testPollCount()
{
	myMockWidget.setResponseCode(DEVICE_READY);
	myMockWidget.setExpectedPollCount(1);
	myApplication.sendRegistrationKey(myMockWidget);
	myMockWidget.verify();
}
```

Ukázka 13 - Kód testu

```
public class MyMockWidget implements Widget
{
	private int myPollCount;
	private int myExpectedPollCount;
	private ResponseCode myResponseCode, actualResponseCode;

	public ResponseCode getDeviceStatus()
	{
		myPollCount++;
		if (myPollCount > myExpectedPollCount)
		{
			junit.framework.Assert.fail("Polled too many times");
		}
		return myResponseCode;
	}

	public void setResponseCode(ResponseCode expected)
	{
		myResponseCode = expected;
	}

	public void setExpectedPollCount(int expected)
	{
		myExpectedPollCount = expected;
	}

	public void verify()
	{
		junit.framework.Assert.assertTrue("Actual response code doesn't match
		expected", myResponseCode == actualResponseCode);
	}
}
```

Ukázka 14 - Kód jednoduchého Mock Objectu

Více o Mock Objects a jejich strategii lze zjistit v dokumentu Endotesting[3]. Díky tomuto dokumentu vznikly téměř všechny mockovací frameworky. Základní motivací pro jejich vznik byla úvaha: „Pokud pořád dokola píšeme ty samé třídy, nedalo by se pomocí nástrojů pro generování kódu vyhnout psaní mocků a dynamicky je generovat?“. Tato motivace je velice zajímavá a silně připomíná motivaci pro náš problém. Proto je vhodné v této práci uvést, jaké mockovací strategie existují, jak fungují a jak se pomocí nich napíše jednoduchý test.

[3] Dostupné online na http://stalatest.googlecode.com/svn/trunk/Literatur/mockobjects.pdf

5.2 JMock

JMock je open source framework, který poskytuje pohodlné a výkonné API pro mockování rozhraní, specifikování očekávaného volání a definování očekávaného chování. JMock shrnuje zkušenost za několik let, která vznikla používáním Mock Objects strategie jejími tvůrci [8].

Použití Mock Objects trpělo jedním neduhem. Programátor musel během psaní testu přestat a napsat mock implementaci netestovaných objektů, viz Ukázka 1. JMock se snaží tento neduh odstranit dynamickým generování kódu mocků při běhu testu. Nabízí tak API, kterým je možné dynamicky vytvářet mocky se všemi vlastnostmi shrnutými v předchozí kapitole [8].

Vstupním bodem JMock frameworku je třída `MockObjectTestCase`, která dědí od klasického JUnitového `TestCase`. Pokud vytváříme test, můžeme od této třídy dědit. Tím získáme veškeré možnosti testovacího frameworku, včetně automatické validace po skončení testu [8].

Mocky jsou vytvářeny metodou `mock(Class<?>)`, která očekává jako vstup rozhraní, jehož implementaci pak vrátí jako Mock object. Ten může být přetypován na mockovaný typ a poslán do produkčního kódu, aby mohl být otestován. V nové verzi je mock automaticky přetypován na poskytnutý interface. Místo psaní těla třídy stačí napsat pouze:

```
Widget myWidget = mock (Widget.class);
```

Ukázka 15 - Snadné vytváření mocků s JMock 2.6.

Dále je potřeba vytvořit takzvaná očekávání. V první verzi tohoto frameworku se vytvářela pomocí objektu Mock, který poskytoval metody nad vytvořeným MockObjectem. Náš příklad by tak vypadal následovně:

```
public void testPollCount()
{
	Mock widget = mock (Widget.class);
	widget.expects(once()).method("getDeviceStatus").
	will(returnValue(DEVICE_READY));
	myApplication.sendRegistrationKey((Widget) widget);
}
```

Ukázka 16 - JMock1 kód testu

Jak je vidět na ukázce kódu, Framework se používá velice podobně jako například SQL, jazyk testu je tedy velmi podobný přirozenému jazyku. Mock Object očekává, že bude zavolána metoda `getDeviceStatus` právě jednou a vrátí specifikovanou hodnotu, zde jakousi konstantu `DEVICE_READY`. Před ukončením testu každý vygenerovaný Mock Object sám zvaliduje splnění podmínek. Z ukázky kódu je jasně vidět, že díky frameworku není nutné vytvoření celé třídy. Třicet řádek kódu bylo nahrazeno jednou řádkou. Zdrojový kód testu zůstává stále stejný. Obecně tedy můžeme říct, že použitím frameworku si ušetříme psaní stubů a navíc získáme další možnosti.

JMock totiž nabízí následující funkcionalitu:

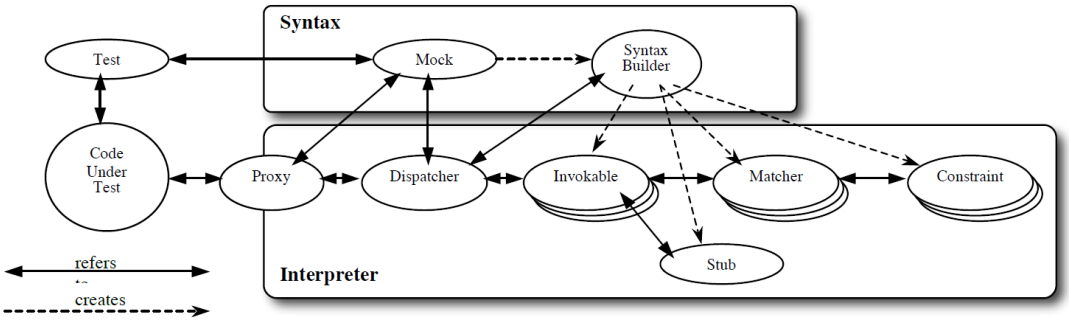
```
public void testExpectations()
{
	Mock mock = mock (Widget.class);
	mock.expects(exactly(7)).
	method("methodName").
	with(null,null).
	after(mock, "previousMethod").
	match(exactly(4)).
	will(returnValue(7)).
	id("invocation 1");
}
```

Ukázka 17 - Možnosti JMock1

Na předchozí ukázce kódu jsem chtěl demonstrovat možnosti testovacího frameworku JMock. Nyní je potřeba jednotlivá klíčová slova rozebrat a vysvětlit jejich význam a použití.

- **Expects** - klíčové slovo očekávání. Pomocí něj specifikujeme počet očekávaných volání metody. Použitelné hodnoty jsou `once` – právě jednou, `never` – nikdy, `atLeastOnce` – alespoň jednou, `atMostOnce` – maximálně jednou. Nic nám ale nebrání si naimplementovat vlastní `InvokeCountMatcher`.
- **Method** – jméno metody. Metoda musí existovat, je hledána reflexí (java.util.reflex.*).
- **With** – parametry metody, s jakými bude zavolána. V tomto případě to budou dva parametry null a null. Obecně to může být libovolná hodnota.
- **After** – po jaké metodě bude tato metoda zavolána. Obsahuje informace o objektu (mock) a jméně metody (previousMethod), která musí být volána před touto metodou. Zde platí stejné pravidlo jako u klíčového slova method.
- **Match** – co musí splňovat metoda after. Zde jde o čtvrté volání metody after.
- **Will** – co se bude dít po zavolání této metody. Zde je obvykle implementována nějaká návratová hodnota. Nic nebrání testerovi si naimplementovat vlastní návratovou hodnotu použitím třídy `Stub`.

5.2.1 Jak to funguje



Obrázek 11 Schéma frameworku jMock - zdroj [JMock2]

Pokud si pozorně prohlédnete předchozí obrázek, zjistíte, že celá funkčnost frameworku je vytvořena použitím Java Proxy. Přes ní je implementována výkonná část mocku, tedy ta, která nás zajímá. Syntaktickou část demonstruje Ukázka 17. Výkonná část komunikuje s Proxy. Každý mock má svou Proxy a svůj Dispatcher. V něm jsou zaregistrovány všechna chování metod. Ta mají rozhraní Invokable. Mock dostane požadavek na metodu, tu odešle proxy, která se snaží najít chování pomocí Dispatcheru. Pokud nenajde odpovídající chování, můžeme očekávat výjimku.

Test správně proběhne pouze v tom případě, že všechny metody mají správně naimplementované chování. To nemusí úplně vyhovovat v případě, že chceme komponentu použít pouze jako stub, nad kterým bychom chtěli vyvíjet nějakou funkcionalitu. Pokaždé před použitím metody stubu bychom museli napsat kód, který metoda vrátí, i když krátký.

5.2.2 Shrnutí

JMock je velice výkonný a užitečný nástroj, pro naše účely je ale nepoužitelný z několika důvodů:

- Kód vlastně vůbec negeneruje, v nové verzi dokonce nepoužívá ani žádný Code Generation Framework. Vše zvládne reflexí a proxy.
- Nepotřebujeme stanovovat, kolikrát má nebo může být metoda volána, to za nás řeší SimCo.
- Nepotřebujeme řešit pořadí volání, to za nás opět zařídí SimCom.
- Bez použití knihovny cglib umí mockovat pouze rozhraní. S jejím použitím JMock neumí mockovat final třídy a metody. Kód, který je generován knihovnou cglib, je nepoužitelný pro naše účely, protože vygenerovaná třída je pouze kolekce řetězců a návratových hodnot.

Můžeme se ovšem poučit z nastíněných problémů a jejich řešení. Knihovna totiž jinak nabízí spoustu zajímavých funkcionalit pro jednotkové testování.

- Celá knihovna je Thread Safe, pokud se použije správná threading policy. JMock poskytuje základní podporu pomocí třídy `Synchronizer`. Viz [9]
- Pomocí Proxy-Dispatcher patternu je možné implementovat Transparentní Proxy, viz 7.8.
- Knihovna má propracované, inteligentní a důkladné logování, kterým by se mohl inspirovat testovací Framework SimCo.
- Nutí programátora psát Interface a správně dekomponovat program. Kvůli tomuto pedantickému přístupu musí programátor psát správně strukturovaný kód, což v budoucnu jistě ocení.

Osobní hodnocení

Mně osobně se více líbila starší verze 1.x, protože psaní testů bylo přímočaré a efektivní. Ve verzi 2 se autoři snaží o vytvoření vlastního jazyka a výsledek, dle mého názoru, není příliš použitelný. Tento pocit je ale silně subjektivní, funkční jsou obě verze. Kterou z nich použít, to už je otázka osobních preferencí.

5.3 EasyMock

EasyMock je prvním frameworkem, který začal dynamicky generovat mocky. Filosofií je velmi podobný JMocku, protože vychází ze stejného článku [7]. Poprvé byl prezentován na OOPSLA konferenci v roce 2001. [10]

Aktuální verze frameworku je 3.1. V následujících kapitolách zjistíme, že se zase tolik neliší od frameworku JMock.

5.4.1 Filosofie

Jak bylo řečeno v předchozí kapitole, použití JMocku se dá rozdělit do čtyř kroků. EasyMock má podobnou filosofii, jen ji nazývá jinak. Porovnání postupu použití frameworku najdete v Tabulka 1.

Tabulka 1 - Porovnání postupu testování. JMock vs EasyMock

JMock	EasyMock
Vytvořit Mocky	**Vytvořit Mocky ve stavu nahrávání**
Nastavit Očekávání	**Nahrát Chování**
	Přepnout Mocky do stavu přehrávání
Pustit produkční kód	**Pustit produkční kód**
Validovat (děje se automaticky)	**Validovat (můžeme manuálně i automaticky)**

EasyMock obsahuje jeden krok navíc, a tím je přepnutí mocku, který zde má dva stavy. Po jeho vytvoření je v takzvaném stavu nahrávání, kdy každé zavolání metody je nahráváno. Očekává se, že po zavolání metody se nastaví její návratový typ. Zavoláním metody `replay` se Mock přesune do stavu přehrávání a může být použit v jednotkových testech. Můžeme vytvořit více typů mocků podle přísnosti validace. Tyto typy rozdělujeme na :

- **Nice** – nezáleží na pořadí nahraných metod, metody můžou být volány, i když nebyly nahrány.
- **Normal** – nezáleží na pořadí metod. Metody vyhodí výjimku, pokud nebyly nahrány.
- **Strict** – záleží na pořadí metod, jak byly nahrány. Vyhodí výjimku, pokud nebyly nahrány.

Míra přísnosti testu odpovídá zvolenému typu mocku. Logicky platí: Čím přísnější test, tím více se musí upravovat, pokud dojde ke změně kódu. Pokud ovšem nastane změna v logice následkem nerelevantní změny v kódu, anglicky side effect of change, strict mock nás na to upozorní tím, že testy začnou padat. Obecně platí, že úroveň přísnosti testů s sebou přináší vyšší režii v podobě nutnosti úpravy testů pro každou změnu. Zároveň ale přináší benefit v podobě vyšší kontroly zdrojového kódu a zachycení nežádoucích účinků změn, které ani zdánlivě nemusejí souviset se zmiňovanou funkcionalitou.

5.4.2 Vytváření mocků

EasyMock poskytuje dvě možnosti, jak používat tuto knihovnu. Jedna je pomocí statického importu, kdy se do libovolné třídy importem přidá funkcionalita pro vytváření, nahrávání a validaci mocků a nebo použitím poskytované třídy **`EasyMockSupport`**. Druhá varianta je zvláštní v tom, že třída **`EasyMockSupport`** nedědí od JUnit **`TestCase`**, ale očekává, že uživatel bude používat JUnitové anotace. V jednoduché ukázce v této práci bude knihovna využita pomocí statických importů. Výhodou toho přístupu je jednoduchost, nevýhodou pak, že všechny mocky se musí validovat deklarativně, u každého vytvořeného mocku musí být zavolána metoda `validate`. Pro začátek je nutné uvést statický import, jak demonstruje Ukázka 18.

```
import static org.easymock.EasyMock.*;
```

Ukázka 18 - Statický import EasyMock knihovny

Díky tomuto jednořádkovému importu je možné využívat v testu veškerou funkcionalitu knihovny. Kód celého triviálního testu by vypadal následovně:

```
public void testWidget()
{
	Widget mock = createMock(Widget.class);//Mock-Creation
	expect(mock.getDeviceStatus()).andReturn(DEVICE_READY);//Assertion
	replay();
	myApplication.sendRegistrationKey(mock);//Invocation
	verify();
}
```

Ukázka 19 - Naivní test v EasyMock

Z ukázky kódu je patrná inspirace stejnou filosofií MockObjects. Proto jsou zdrojové kódy testů pro obě knihovny až na specifika EasyMock (metody `replay` a nutnost manuální validace) téměř identické.

5.4.3 Vlastnosti knihovny

EasyMock se nesnaží být čistý za každou cenu. Hlavním cílem tohoto frameworku je usnadnění práce uživateli. To znamená například, že od základu umožňuje mockovat i třídy pomocí frameworků **Cglib** a **Objenesis**. Kde se **JMock** snaží programátora donutit používat best practices, **EasyMock** nabízí způsob, jak otestovat i kusy kódu, které by měly být pro testování refaktorovány. Mockování tříd má ovšem následující limity:

Final metody – nemohou být namockovány, bude volán jejich kód i v mocku. To je možné obejít napsáním vlastního Unfinalizeru[4], ale už jen z principu věci se to silně nedoporučuje. Existují i frameworky řešící toto dilema (Power Mock), ale to už je mimo rozsah této práce.

[4] Nástroj, který změní byte kód třídy tak, že z něj smaže klíčová slova final. Poté je třída dynamicky načtena přes `ClassLoader` a testována. Není to bohužel pouze teoretická úvaha. Tato obskurnost se v praxi opravdu používá.

Private metody – také nemohou být namockovány, z principu by neměly být nikdy volány, protože mock by měl fungovat jako black-box. Veškeré implementační detaily by měly zůstat pro volajícího neviditelné. Vzhledem k tomu, že EasyMock nabízí funkcionalitu částečného mockování, jsou i privátní metody dostupné v mocku, ale fungují podobně jako final, bude zavolán původní kód.

Metody třídy Object – metody `equals`, `toString`, `hashCode` a `finalize` nemohou být mockovány. Je to považováno za výhodu, protože uživatel se nemusí zabývat implementačními detaily pro tyto metody.

Podporované JVM – pouze následující virtuální stroje jsou podporovány a odzkoušeny pro **Objenesis**. Aktuálně podporované stroje lze nalézt na stránkách knihovny **Objenesis** a na stránkách **EasyMock** frameworku. V době psaní této práce to byly tyto:

- **Sun Hotspot** VM, verze 1.3, 1.4, 1.5, 1.6 a 1.7
- **Dalvik** (Ice Cream Sandwich a Jelly Bean)
- **BEA JRockit** verze 7.0 (1.3.1), 1.4.2 a 1.5
- **GCJ** verze 3.4.4 (testováno na Windows/Cygwin)
- **Aonix PERC** (nefunkční serializace mocků), testováno na verzi 5.0.0667 [10]

Tyto limitace je možné obejít napsáním vlastního nástroje pro generování mocků. Knihovna je snadno rozšiřitelná, stačí implementovat rozhraní `IClassInstantiator` a jeho instanci zaregistrovat staticky voláním `ClassInstantiatorFactory.setInstantiator`. Toto volání je statické, a proto je doporučeno na konci testu vrátit zpět původní instanci metodou `setDefaultInstantiator`.

EasyMock také podporuje takzvané částečné mockování, ačkoliv oficiální stanovisko autorů knihovny je, že pokud musíte použít tuto funkcionalitu, měli byste nejdříve přemýšlet o refaktoringu. Touto knihovnou je tedy možné mockovat pouze některé metody a ostatní nechat nezměněné. Zdrojový kód vzorového testu s použitím částečného mockování by vypadal pak následovně:

```
public void testWidget()
{
	Widget mock = createMockBuilder(Widget.class).
	addMockedMethod("getDeviceStatus").createMock();//Mock creation
	expect(mock.getDeviceStatus()).andReturn(DEVICE_READY);//Assertion
	replay();
	myApplication.sendRegistrationKey(mock);//Invocation
	verify();
}
```

Ukázka 20 - Ukázka částečného mockování

5.4.4 Shrnutí

Klady

- Jednoduchost
- Rozšiřitelnost
- Možnost částečného mockování
- Podpora pro Android
- Mockování tříd
- Stručná a přehledná dokumentace
- Nabízí veškerou důležitou funkcionalitu JMocku

Zápory

- Nutnost nahrávání a přehrávání
- Možnost obcházet best practices
- Při selhání vyhazuje obyčejné výjimky
- Logování by mohlo být lepší
- JMock nabízí víc funkcionality

5.5 Mockito

Mockito je další mockovací framework, který vymyslel Szczepana Faber z důvodu nespokojenosti s ostatními mockovacími frameworky. [11] Tento framework je relativně mladý a snaží si vzít to nejlepší ze svých předchůdců **EasyMocku** a **JMocku**. Autor Mockita měl pro jeho zavedení následující důvody.

- Ať je to jednoduché – není důvod vymýšlet nějaký složitý a sofistikovaný jazyk pro volání metod v Javě, pro to již existuje jazyk, jmenuje se Java. [11]
- Čím méně specifického jazyka, tím lépe. Interakce ať je jenom volání metod. Volání metod je jednoduché, učit se nový jazyk je složité. [11]
- Nepoužívat řetězce pro metody. Strávil jsem více času čtením zdrojového kódu než psaním nového. Proto chci, aby zdrojový kód byl čitelný. [11]
- Nepoužívat anonymní třídy – přináší více složených závorek, více odsazení, více kódu, více znaků. Už jsem zmínil, že jsem strávil více času čtením než psaním? [11]
- Jednoduchý refaktor – změna jména metody, by neměla rozbít test. [11]

Původně se jednalo o „hackování“ kódu **EasyMock** frameworku a upravování jej k obrazu svému. Postupem času se z **Mockita** stal autonomní projekt, který je nyní absolutně samostatným a oblíbeným mockovacím frameworkem.

5.5.1 Filosofie

Mockito používá podobnou filosofii jako **EasyMock**, ale jsou zde jisté rozdíly. Všechny mocky jsou **nice,** viz kapitola 5.4.1, nevyhazují tedy výjimku, pokud byla volána metoda, která nebyla očekávána. Mocky se nejprve nastavují, v terminologii **Mockita** stubují, až poté se volá produktivní kód. Po jeho skončení přichází verifikace. Ta zde probíhá po každé metodě. Kontrolují se pouze ty metody, které programátor explicitně napíše. Je tedy možné testovat i vybrané kusy kódu, nikoliv pouze celek. Cílem **Mockita** je snaha poskytnout programátorovi nástroj, kterým co nejrychleji otestuje svůj kód bez zbytečné frustrace nebo nastavování mocku.

Mockito si také klade za cíl co nejvíce zjednodušit výpis chyb. Cílem je, aby výpis chyby, anglicky stack trace, byl pokud možno co nejjednodušší a programátor se v něm co nejlépe orientoval. Další snahou je jednoduché porovnávání testovaných hodnot. Cílem není vymyslet vlastní jazyk, ale použít co nejčitelnější metody pro porovnávání. Dalo by se říci, že **Mockito** se snaží vzít si to nejlepší z **JMocku** a **EasyMocku**, smíchat to a výsledný koktejl nabídnout programátorovi, bez nevýhod těchto známých mockovacích frameworků. [12]

5.5.2 Příklad

Abychom knihovnu mohli použít, musíme ji přilinkovat do projektu. To je opět možné statickým importem. Tato funkcionalita je vypůjčena z **EasyMocku**. Po přilinkování můžeme začít se samotným kódem testu. **Mockito** nenabízí speciální třídu, kterou by bylo potřeba dědit. Vše je řešeno jedním statickým importem.

```
import static org.mockito.Mockito.*;
```

Ukázka 21- Statický import pro použití knihovny Mockito

Mocky se zde dají vytvořit pouze jedním způsobem, a to voláním metody `mock`. Dále je možné nastavit, jaké hodnoty budou vráceny pro příslušné volání pomocí metody `when`. Poté můžeme specifikovat, jaká hodnota bude vrácena. Pokud ji nespecifikujeme, vrátí se přednastavená hodnota, stejná jako v Javě pro neinicializovaný objekt. Pro objekty je to null, pro String prázdný řetězec atp. Mockito se ale snaží dělat **Nicer** [12] mocky, což znamená, že pokud očekáváme kolekci, dostaneme prázdnou kolekci, ne null, pokud očekáváme pole, dostaneme prázdné pole atp. Zdrojový kód triviálního testu bude vypadat následovně:

```
public void testWidget()
{
	Widget mock = mock(Widget.class);//creation
	when(mock.getDeviceStatus()).thenReturn(DEVICE_READY);//stubbing
	myApplication.sendRegistrationKey(mock);//Invocation
	verify(mock).getDeviceStatus();
}
```

Ukázka 22 - Triviální test Mockito

Kód testu je tedy až na klíčové slovíčko `replay` de facto identický. Část očekávání spíše připomíná **JMock1**, ale není tak komplexní. Vzhledem k tomu, že **Mockito** vychází z **EasyMocku**, není příliš velký problém refaktorovat kód z jednoho frameworku na druhý.

5.5.3 Limitace

Mockito má následující limitace:

- Vyžaduje Javu verze 1.5.
- Nelze mockovat statické metody.
- Nelze mockovat final metody.
- Nelze mockovat metody `equals` a `hashCode`. Ty by se neměli vůbec mockovat, protože Mockito počítá s jejich specifickou implementací u mocků, proto by modifikace těchto metod mohla rozbít celý Framework.
- K mockování tříd využívá knihovnu Objenesis, proto platí stejná limitace pro virtuální stroje jako pro EasyMock.
- Nepodporuje částečné mockování.

Mockito zároveň obsahuje skvělou vlastnost a tou je samo-validace. Pokud programátor špatně použije některou metodu frameworku, je upozorněn výjimkou přesně na té řádce, kde je špatně použita. Například pokud verifikujeme volání před tím, než jej nastavíme, knihovna nás upozorní, že jsme nenastavili očekávání.

5.5.4 Shrnutí

Klady

- Jednoduchost
- Rozšiřitelnost
- Podpora pro Android
- Mockování tříd a Nicer Mocky
- Stručná a přehledná dokumentace
- Nabízí veškerou důležitou funkcionalitu JMocku
- Odstraňuje neduhy EasyMocku
- Samovalidující
- Možnost nevalidovat všechna volání
- Snadno čitelný
- Dobře trackovatelný Stack Trace

Zápory

- Není tak pedantický
- JMock nabízí víc funkcionality
- Validace se musí psát pro každé volání

6 QDox

QDox je velmi rychlá a malá open source knihovna na parsování zdrojových souborů, je designovaná speciálně pro účely generátorů kódu nebo nástrojů pro tvorbu dokumentace. [13]

QDox využívá knihovny JFlex a BYacc/J pro parsování nejdůležitějších informací ze zdrojových souborů v jazyce Java. Čte pouze ty nejdůležitější informace jako importy, definice tříd a rozhraní, JavaDoc a atributy tříd. Parser úplně ignoruje implementace metod, tedy veškerý kód, který je napsán ve složených závorkách. Díky tomu je tato knihovna velice rychlá a používaná ve spoustě dalších projektů.

Obrovskou výhodou QDoxu je, že dokáže zpracovat jak zkompilované soubory, tak i zdrojové soubory. Bohužel u zkompilovaných souborů nedokáže extrahovat JavaDoc, protože v byte kódu neexistuje, je odstraněn při kompilaci. QDox je open source. Zdrojové soubory jsou volně přístupné na internetu a tím pádem mohou být libovolně modifikovány.

Vzhledem k tomu, že QDox dokáže pracovat jak se zdrojovými soubory, tak se zkompilovanými soubory, jeví se jako ideální kandidát pro generování stubů pro SimCo.

7 Návrh a implementace nástroje pro generování mockupů

V teoretickém úvodu byly vysvětleny technologie, které využívá framework SimCo. Nyní je potřeba definovat problém, dekomponovat jej na funkční celky, navrhnout strategii jejich implementace, ověřit zda realizace problému je skutečně funkční.

7.1 Definice problému

V kapitole 4.2 zabývající se komponentami je uvedeno, že ve frameworku SimCo existují čtyři druhy komponent. Tato diplomová práce bude používat pouze tři – simulované, reálné a typu prostředník.

Cílem této práce je vytvořit základ pro funkční scénář z poskytnutých reálných komponent. Uživatel dostane na výběr, které komponenty chce simulovat a které budou použity bez jakékoliv změny. Aplikace pak následně analyzuje závislosti mezi komponentami a mezi každé dvě reálné komponenty vloží komponentu typu prostředník. Vzhledem k tomu, že prostředník připomíná návrhový vzor proxy, budeme jej označovat také jako transparentní proxy.

Všechny komponenty, které uživatel označil jako simulované, budou prozkoumány a budou z nich vytvořené komponenty nové, které by se daly označit za stubové komponenty.

Práci tak můžeme rozdělit na dva dílčí problémy :

1) Vytvoření simulačních komponent – mockupů
2) Vytvoření spojení mezi reálnými komponentami – transparentní proxy

7.2 Použité technologie

Cílem projektu je vyrobit rozšíření do již existujícího programu. Aby aplikace byla kompatibilní s frameworkem, je logickým a nejjednodušším krokem použít stejné technologie, které používá framework SimCo, i když v novějších verzích:

- Java 1.7 JDK verze 13 (minimální verze)
- Eclipse 4.2.1 Juno
- Eclipse Equinox
- Spring DM Release 4

Aplikace se bude snažit o platformě nezávislý kód, hypoteticky by měla jít spustit na libovolném operačním systému s kompatibilním virtuálním strojem. Vyvíjena však bude pod operačním systémem Windows 7, stejně jako byl vyvíjen framework SimCo.

Dále bude vyžadovat knihovny k vytváření simulovaných komponent a k vytváření prostředníků mezi reálnými komponentami. Po dlouhých úvahách, které budou popsány dále, byla vybrána pouze následující knihovna:

- QDox verze 1.12.1

Tato knihovna je poskytována jako open source pod licencí Apache. Bude použit její zdrojový kód, který bude modifikován.

7.3 Analýza problému a dekompozice

V kapitole 7.1 jsme definovali dva základní problémy. Tyto problémy spolu nijak nesouvisí, týkají se každý jiného typu komponenty. Logickým východiskem je rozdělit aplikaci na dvě na sobě nezávislé komponenty. Vzhledem k tomu, že komponenty jsou rozšířením funkcionality frameworku SimCo, měly by tak být i označeny. Jména komponent budou začínat `simco.framework.extension.` První se bude zabývat tvorbou simulovaných komponent, nazveme ji `mockupcreator`, druhá se bude zabývat tvorbou prostředníků mezi reálnými komponentami a nazveme ji `transparentproxy.`

7.3.1 Strategie vytváření simulovaných komponent

Pokud bychom chtěli vytvářet automaticky simulované komponenty, máme dvě možnosti. Buď použít mockovací framework k dynamickému vytváření objektů z nějakých konfiguračních souborů, nebo fyzicky vytvořit třídy a do nich generovat implementační kód. Každá z variant má svoje pozitivní i negativní stránky, jedná se asi o stejné dilema, jako zda použít aspektově orientované programování nebo ne.

7.3.2 Dynamický přistup k vytváření simulačních komponent

Hlavní nevýhodou dynamického přístupu je ztížení refactoringu a nutnost psát dlouhé konfigurační soubory. Tento problém hezky shrnul Szczepan Faber [11], a byl vlastně hlavním důvodem, proč vyvinul Mockito. Pokud bychom totiž k problému přistupovali dynamicky, museli bychom si někam ukládat, jaké metody mají vracet které hodnoty. Pro každou třídu, kterou bychom chtěli použít, bychom museli vytvořit konfigurační soubor. Jistě by tento konfigurační soubor časem začal enormně narůstat, nehledě na fakt, že psát jména metod bez jakékoliv kontextové nápovědy je pro programátora neskutečně otravné a deprimující [11].

Další zásadní nevýhodou je pak refaktoring. Pokud se změní jméno metody nebo její signatura, budeme muset tuto informaci aktualizovat v konfiguračním souboru třídy. Představme si nyní, že změníme rozhraní, které implementuje tisíc tříd, a pokaždé chceme, aby vracelo různou hodnotu. Budeme tak muset změnit tisíc konfiguračních souborů.

Nevýhodou je také fakt, že definování složitějšího chování metod je nutné pomocí frameworku. Pokud bychom chtěli vyrobit nějaké výrazně složitější chování, mohli bychom narazit na limitace mockovacího frameworku, který použijeme. Podle známého pravidla „*There is no silver bullet*“ [14], neexistuje mockovací framework, který by svým použitím nepřinesl potíže, které by nebylo nutné řešit.

Nespornou výhodou je však možnost celý balík obsáhnout v jednom konfiguračním souboru, který by mohl být dynamicky generován. Otázkou je, jak složitá by byla tato konfigurace? Jak složité by muselo být uživatelské rozhraní? Jaké by byly limity?

7.3.3 Statický přístup k vytváření simulačních komponent

Hlavní nevýhodou statického přístupu je nutnost po každé změně kódu psát znovu implementaci metod. Otázkou je, zda by nestačilo zkopírovat chování z předchozí verze simulované komponenty. Příprava scénáře by obecně byla obtížnější. Aplikace by generovala zdrojové soubory simulované komponenty. Ty by musely být vloženy do nějakého IDE jako projekt, doplněny o kýžené kusy kódu a přeloženy.

Výhodou je, že lze okamžitě poznat, zda se aplikace změnila nebo kde je chyba. Aplikace buď nepůjde přeložit, nebo při spuštění spadne s hláškou, která nebude obsahovat nadbytečné informace z mockovacího frameworku.

Další výhodou je rychlost spuštění aplikace, kdy režie na vytvoření testované komponenty je pouze při vytvoření. Pokud bychom použili dynamický přístup, museli bychom při každém spuštění dynamicky vytvořit celou simulovanou komponentu, zatímco staticky vytvořená simulovaná komponenta je pokaždé zkompilovaná a připravená v jar souboru.

Za hlavní výhodu lze považovat fakt, že simulovanou komponentu můžeme rozšířit o libovolně složitou strukturu tříd, knihoven. Limitace chování simulované komponenty vyplývá pouze z použitého programovacího jazyka a programátorových schopností s ním pracovat.

Pro statický přístup jsem se rozhodl po důkladném zvážení obou metod. Hlavními důvody ovlivňujícími moje rozhodnutí byly: (1) Dynamickým přístupem se zabývá jiná diplomová práce. (2) Mockovací framework limituje možnosti návratových hodnot a zvyšuje složitost testování komponent.

7.3.4 Dekompozice

Cílem dekompozice bude nalézt kandidáty na třídy a tyto kandidáty umístit do balíčků. Balíčky budou vždy začínat informací, kdo je vyrobil a do jaké organizace patřil. `Org.kiv.zcu.cz.sandcrew` bude prefix, který jednoznačně identifikuje kód, který jsem pro SimCo poskytnul. Nejprve začnu s dekompozicí prvního problému, poté budu pokračovat s problémem druhým.

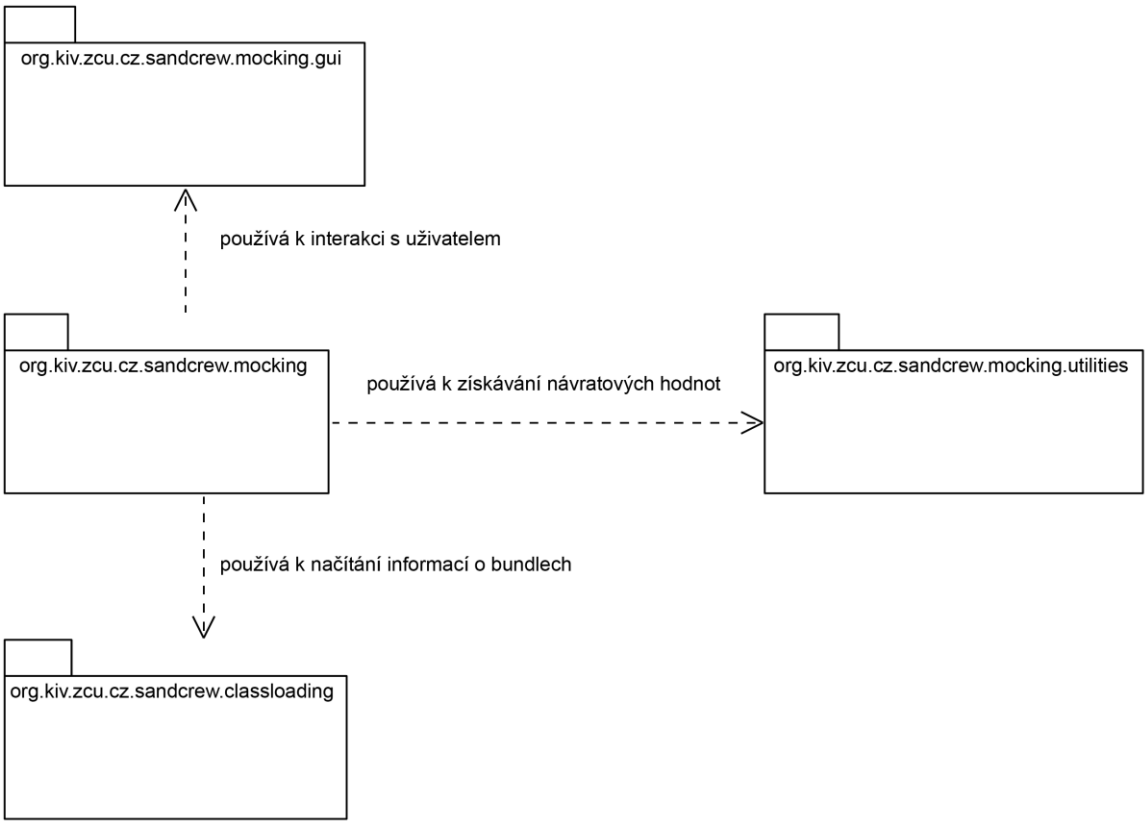
S dekompozicí problému začnu od vstupu. Jako vstup jsou očekávány jar soubory nebo již nainstalované bundly v repozitáři OSGI. Pro čtení informací o zdrojových nebo zkompilovaných souborech z komponenty bude definována třída `JarReader`, která bude schopna poskytnout načtené informace aplikační vrstvě. Bude se starat o přečtení všech kýžených informací z dané komponenty. Následně bude poskytovat informaci o tom, jaké konkrétní třídy, v kterém balíčku, je třeba simulovat. Balíček poskytující tyto služby nazveme `classloading`.

Pokud máme načtenou komponentu, budeme potřebovat vytvořit pro každou třídu její simulovanou implementaci. Vzhledem k tomu, že budeme chtít číst přeložené i zdrojové soubory, vytvoříme abstraktní třídu `AbstractMocquer`, která bude poskytovat základní služby pro vytváření mockupů tříd. Její specializací potom budou konkrétní implementace `ClassMocquer` a `JavaSourceMocquer`, které budou řešit specifika vytváření mockupů z již přeložených nebo zdrojových souborů. Po delší úvaze jsem se rozhodl umístit do stejného balíčku i třídu `SpringMocker`, která bude třídou řídící spolupráci ostatních komponent. Její odpovědností bude interakce mezi uživatelem a programem, interakce s datovou vrstvou a vytváření specializovaných objektů pro tvorbu simulovaných tříd. Balíček obsahující tyto třídy nazveme `mocking.`

Z předchozí analýzy jádra mockování je jasná potřeba vytvořit balíček pro uživatelské rozhraní. Třída `GuiHandler` bude zodpovědná za interakci s uživatelem, získání místa odkud se budou číst komponenty. Jejím úkolem bude také zjistit, které komponenty se budou simulovat a které ne, a kam se budou zapisovat. Uživatelské rozhraní bude v balíčku `mocking.gui`.

Poslední funkcionalitou potřebnou pro správné vytváření simulovaných tříd budou pomocné třídy pro vytváření návratových hodnot. Uživateli bude poskytnuta možnost zaregistrovat pro třídu defaultní návratovou hodnotu. K tomu bude sloužit singleton utility třída `MockingUtils` v balíčku `mocking.utils`. Funkcionalita kolem návratových hodnot má vlastní balíček z důvodu budoucí rozšiřitelnosti.

Na diagramu dekompozice na jednotlivé balíčky a vazby je vidět, že je použitá třívrstvá architektura. Kde `classloading` balíček je datová vrstva, `mocking` a `mocking.utilities` aplikační a `mocking.gui` prezentační.



Obrázek 12 - Diagram balíčků pro komponentu MockCreator

Situace pro transparentní proxy bude výrazně jednodušší. Komponenta `TransparentProxy` bude sloužit pouze k tomu, aby zaregistrovala **OSGI** službu, která bude řešit propojení reálných komponent pomocí **Springu**. Úpravu konfiguračních souborů reálných komponent bude mít na starosti opět `SpringMocker`. O realizaci se bude starat **OSGI** služba deklarovaná v komponentě `TransparentProxy`. Pro deklaraci služby je potřeba rozhraní, proto jej vytvoříme a nazveme `ITransparentProxyFactory`, které bude mít na starosti vytváření proxy objektů mezi reálnými komponentami.

7.4 Čtení komponent – Datová vrstva

Ke čtení informací o komponentách slouží třída `JarReader`. Z komponenty potřebujeme získat informace o tom, jaké balíčky exportuje, a z nich získat všechny třídy, které jsou přístupné zvenčí, tedy public. Komponenta teoreticky může obsahovat jak zdrojové, tak zkompilované soubory. Proto budou potřeba dvě mapy balíčků na kolekce zkompilovaných tříd – `Class`, nebo adresářů zdrojových souborů – `File`. Tyto dvě informace jsou drženy, jako atributy této třídy viz Ukázka 23.

```
private Map<ExportedPackage, Collection<Class<?>>> classMap;
private Map<ExportedPackage, File> sourceFilesMap;
```

Ukázka 23 – Atributy třídy JarReader

Třída potřebuje pouze informaci o tom, který jar soubor má načíst. Místo ručního čtení informací o komponentě, lze požádat OSGI, aby nám je poskytlo. K tomu slouží služba `PackageAdmin` a její metoda `getExportedPackages`. K jejímu použití je však potřeba, aby komponenta již byla v OSGI kontejneru a abychom měli referenci na ni.

`BundleContext` obsahuje informace o všech bundlech, které aktuálně běží v OSGI kontejneru. Nejprve je nutné zjistit, zda požadovaná komponenta není již nainstalována. Ukázka 2 demonstruje jak získat instanci `BundleContextu`. Přesně tento `Activator` používá **MockupCreator**. `BundleContext` poskytuje metodu `getBundles`, která vrací pole všech komponent, které aktuálně běží v OSGI kontejneru. Mezi nimi stačí najít podle jména kýženou komponentu. Tuto funkcionalitu bude poskytovat metoda `getBundleByFilename`. Pokud OSGI zatím neobsahuje kýženou komponentu, zkusíme ji nainstalovat. K tomu slouží metoda `installBundle(String absolutePath)`, kterou opět poskytuje `BundleContext`. Ta vrací požadovanou referenci na komponentu. Zdrojový kód této operace můžete vidět v následující ukázce:

```
BundleContext bc = Activator.getContext();
Bundle loaded = getBundleByFilename(file.getName(), bc);
if(loaded == null)
{
	loaded = bc.installBundle(file.getAbsolutePath());
	loaded.start();
}
```

Ukázka 24 – Získání instance komponenty v OSGI

S referencí na komponentu lze požádat službu `PackageAdmin` o informaci, jaké balíčky jsou exportovány. Toto volání poskytuje pole tříd `ExportedPackages`. S pomocí reference na komponentu a pole exportovaných balíčků jde získat informaci o tom, jaké třídy obsahuje balíček. K tomu slouží metoda `addPackages`, s jejíž pomocí je sestavena informace o tom, které třídy je potřeba vytvořit pro který balíček. Postup získání exportovaných balíčků demonstruje Ukázka 25.

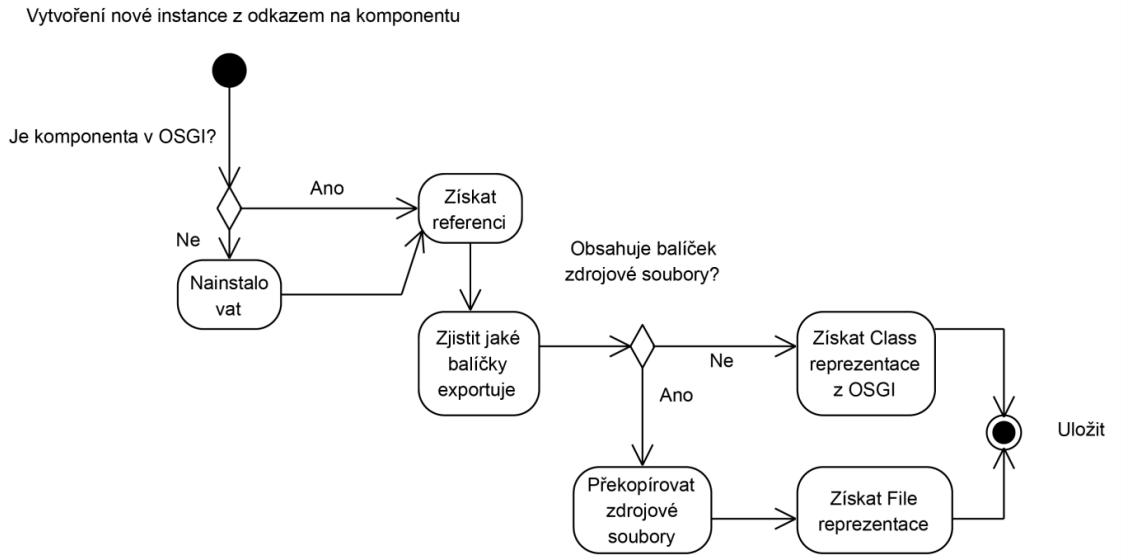
```
ServiceReference ref = bc.getServiceReference(PackageAdmin.class.getName());
PackageAdmin pa = (PackageAdmin) bc.getService(ref);
ExportedPackage[] ep = pa.getExportedPackages(loaded);
addPackages(ep, loaded);
```

Ukázka 25 – Získání informace o exportovaných balíčcích

Nyní je nutné získat z komponenty informace o třídách, které je potřeba simulovat. Komponentu můžeme požádat, aby našla všechny soubory daného typu na dané cestě. K tomu slouží metoda `findEntries` třídy `Bundle`. Cestu lze získat poměrně snadno. Pro zdrojové soubory je to */src/název balíčku/*.java*, pro přeložené soubory je to */bin/název balíčku/*.class*. Situaci ale komplikuje fakt, že přeložené a zdrojové soubory obecně nemusí být ve složce bin nebo src. Toto se stává například v případě exportu balíčku z Eclipse. Aplikace se zdrojové soubory pokusí najít nejprve v adresáři src, a pokud je nenalezne, zkusí to o adresář výš. Prioritu vždy budou mít zdrojové soubory, protože obsahují informaci o JavaDocu. Tato informace se bohužel při překladu ztrácí a neexistuje způsob jak ji získat zpět. Proto simulované komponenty vytvořené z již přeložených souborů nebudou obsahovat dokumentaci v podobě JavaDoc informace.

Metoda `findEntries` vrací výčet URL kýžených souborů. V případě přeložených souborů toužíme získat Class reprezentaci dané třídy. Zde nám opět pomůže OSGI. Třída `Bundle` obsahuje metodu `loadClass`, která pro zadané celé jméno třídy vrátí její `Class` reprezentaci. Celé jméno získáme jako kombinaci názvu balíčku a názvu třídy. Například třída `JarReader` je definována jedinečně svým celým jménem `org.kiv.zcu.cz.sandcrew.mocking.JarReader`.

Situace je podobná pro zdrojové soubory. Zde je potřeba získat přímo soubor jako třídu `File`. Zde jsem bohužel narazil na chybu v Javě, kdy pro `URI` získané z URL metodou `toURI`, není možné získat referenci na soubor. `URI` je neplatné. Tuto nemilou chybu jsem byl nucen vyřešit zkopírováním souborů pomocí `InputStreamu`, který třída `URL` poskytuje. Poté je už možné získat `File` reprezentaci jednotlivých tříd. Celý princip datové vrstvy by se dal trivializovat a shrnout následujícím diagramem:



Obrázek 13 - Proces čtení informací z poskytnutého jar souboru

7.4.1 Čtení Springových Konfiguračních souborů

K Implementaci komponent typu prostředník je nutné znát vazby mezi reálnými komponentami. Ty jsou definovány ve Spring konfiguračních souborech, které jsou umístěny dle konvence na cestě `/META-INF/spring/*.xml`.

K jejich načtení dojde, když se spustí komponenta, která to má ve Springu na starosti. Jmenuje se `org.springframework.osgi.extender`. Jejím úkolem je projít všechny komponenty v OSGI kontejneru a načíst jejich konfigurační soubory tak, aby mohl být vytvořen SpringContext. Více informací bylo již řečeno v kapitole 3, kde jsou také uvedeny příklady konfiguračních souborů. Nejdůležitější jsou pro tento problém ukázky registrace a reference na OSGI službu.

Obecně jakákoliv komponenta může zaregistrovat službu pod jednoznačným identifikátorem `id` a rozhraním `interface`. Komponenta, kterou nazveme konzument služby, ji může obdržet pod tímto rozhraním. Pokud ale obě komponenty budou reálného typu, nebude mít framework tušení o tom, že mezi nimi probíhá komunikace. Proto přesně mezi ně musíme vložit prvek, který může framework kontrolovat. Ano jde o transparentní proxy.

V následující triviální ukázce komponenta A registruje službu, kterou komponenta B konzumuje. Po prozkoumání konfiguračního kódu zjistíme, že v komponentě A se nejprve vyrobí instance implementace daného rozhraní služby – `CalendarImpl`, a to se poté zaregistruje jako služba `CalendarServiceOsgi` poskytující rozhraní `ICalendar`. Konzumentovi služby se pak stačí pouze odkázat na službu jejím rozhraním a může používat všechny metody rozhraní, které jsou nad službou definovány.

```
<!-- Komponenta A Producent -->
<bean id="CalendarService"
class="simco.framework.scheduler.calendar.CalendarImpl">
       <constructor-arg ref="EventAdmin"/>
       <constructor-arg ref="SimCoContextService" />
       <constructor-arg ref="HistoryWriter" />
</bean>
<osgi:service id="CalendarServiceOsgi" ref="CalendarService"
    interface="simco.framework.scheduler.calendar.ICalendar" />

<!-- Komponenta B Konzument -->
<osgi:reference id="CalendarServiceOsgi"
interface="simco.framework.scheduler.calendar.ICalendar"/>
```

Ukázka 26 - Producent a konzument služby

Simco potřebuje mezi tyto dvě komponenty vložit komponentu typu prostředník. Toho lze docílit pomocí návrhového vzoru Factory a jeho podpory ve Springu. Nechť existuje služba, která bude schopna vytvořit takovou transparentní proxy. Služba bude zaregistrována pod symbolickým jménem `TransparentProxyGeneration` a bude poskytovat factory metodu `newTransparentProxy`. Touto službou budeme chtít vyrobit identickou službu, která bude delegovat volání na tu původní.

Přesně tímto způsobem fungují mockovací frameworky, až na to, že většinou neposílají informaci původní službě. Nabízí se zde použití mockovacího frameworku EasyMock, který podporuje částečné mockování, které můžeme použít pouze jako transparentní proxy. Také můžeme použít přímo implementaci Java Proxy a vytvořit si vlastní delegaci. Nevýhodou je, že nebudeme schopni udělat transparentní proxy nad ničím jiným než nad rozhraním. Otázkou je, zda potřebujeme něco více? Ukázka 27 demonstruje tento princip. Zobrazen bude pouze kód producenta, protože u konzumenta se vůbec nic nezmění.

```
<bean id="CalendarService"
   class="simco.framework.scheduler.calendar.CalendarImpl">
       <constructor-arg ref="EventAdmin"/>
       <constructor-arg ref="SimCoContextService" />
       <constructor-arg ref="HistoryWriter" />
</bean>
<!-- Deklarace používání transparentní proxy -->
<osgi:reference id="SimCoContextService"
   interface="simco.framework.extension.ISimCoContext" />
<!-- Deklarace používání transparentní proxy -->
<bean id="CalendarServiceProxy" factory-bean="TransparentProxyGeneration"
   factory-method="newTransparentProxy">
       <constructor-arg ref="CalendarService"/>
       <constructor-arg value="LOGGING"/>
       <constructor-arg value="simco.framework.scheduler.calendar.ICalendar" />

</bean>
<osgi:service id="CalendarServiceOsgi" ref="CalendarServiceProxy"
    interface="simco.framework.scheduler.calendar.ICalendar" />
```

Ukázka 27 – Nahrazení OSGI služby proxy objektem.

Teď již stačí tento princip automatizovat. Bude použit podobný trik jako u komponenty JarReader, kde získáme opět instanci komponenty `Bundle`, kterou požádáme o poskytnutí manifestu na kýžené lokaci. Manifest poté prohledáme na deklarace a reference služeb a vytvoříme seznam deklarovaných a konzumovaných služeb. Pak stačí projít seznam konzumovaných služeb, najít komponentu, která ji deklaruje, a v ní provést úpravy zmíněné v předchozím odstavci. Všechny změněné konfigurační soubory poté uložíme.

7.5 Mockování tříd

Po úspěšném načtení informací z komponenty je potřeba vytvořit simulované komponenty, k tomu slouží třída `AbstractMocquer` a její potomci. Protože padlo rozhodnutí, že se třídy budou vyrábět jako statické java soubory (viz kapitola 7.3), bude pro tento účel použit parsovací framework QDox.

Výhodou QDoxu je, že umí pracovat jak s přeloženými soubory, tak se zdrojovými soubory. Jediný rozdíl je v absenci JavaDocu. Způsob, kterým se ale používá je trochu odlišný, proto jsem vytvořil abstraktní implementaci, která pracuje až s produktem čtení souborů. Tento produkt je třída QDoxu `JavaClass`. `AbstractMocquer` zajišťuje veškerou funkcionalitu spojenou s třídou `JavaClass` z prostého důvodu: Výstup ze zdrojových i přeložených souborů je pak stejný a na jednom místě ve zdrojovém kódu.

7.5.1 QDox a zdrojové soubory

Preferovaným dokumentem jsou zdrojové soubory, kvůli již zmíněnému JavaDocu. `JarReader` má uloženou množinu kombinací balíčku a adresáře se zdrojovými soubory. Následující příklad jasně vysvětlí, proč byla vybrána právě tato kombinace.

```
JavaDocBuilder builder = new JavaDocBuilder();
builder.addSourceTree(value);
for(JavaClass clazz : builder.getClasses())
{
	for(JavaMethod method : clazz.getMethods())
	{
		method.setSourceCode(null);
		handleAMethod(method, clazz);
	}
}
```

Ukázka 28 – Výroba simulovaných komponent ve třídě JavaSourceMocquer

Nejprve je potřeba vytvořit instanci třídy JavaDocBuilder, které předáme třídy podle konkrétního balíčku. Knihovna projde rekurzivně celý adresář a přidá všechny nalezené třídy. To by ale mohl být problém v případě, kdy složka obsahuje další složku se zdrojovými soubory. Proto bude třeba upravit knihovnu tak, aby nečetla zdrojové soubory rekurzivně, ale pouze z předaného adresáře. Tato úprava bude aplikována v metodě `addSourceTree`.

Po úspěšném naparsování lze požádat QDox o všechny třídy, které byly nalezeny metodou `getClasses`. V tuto chvíli máme k dispozici kostru všech tříd v daném balíčku. Každou třídu je potřeba projít a vytvořit jí kýžené tělo. K tomu slouží metoda `handleAMethod`, kterou implementuje `AbstractMocquer`. Po úspěšném vyplnění těl metod je nutné uložit vygenerovanou třídu jako soubor. K tomu slouží utility třída `MockingUtils`, která poskytuje metody pro práci se soubory a je schopna vytvořit metodou `createBufffer BufferedWriter`, který zapíše vytvořenou třídu na patřičné místo.

7.5.2 QDox a zkompilované soubory

Složitější situace vzniká, pokud chceme vytvořit zdrojové soubory z již přeložených. QDox poskytuje možnost použít třídu `ClassLibrary`, které se předá `ClassLoader` kýženého balíčku. Z toho důvodu musí být balíček nainstalován, jinak by `ClassLoader` nebyl schopen poskytnout potřebné třídy. `ClassLibrary` s předaným `ClassLoaderem` použije opět `JavaDocBuilder`. V tomto případě se však musíme ptát na každou třídu zvlášť, je tedy potřebné sestavit kompletní jméno třídy. `JarReader` již však obsahuje načtené Class reprezentace tříd. Z `JavaDocBuilderu` tak jednoduchým způsobem získáme `JavaClass`, což je úplně stejná třída jako v případě parsování zdrojových souborů. Následující ukázka demonstruje výše popsaný postup.

```
ClassLibrary lib = new ClassLibrary(clazz.getClassLoader());
JavaDocBuilder doc = new JavaDocBuilder(lib);
JavaClass aClazz = doc.getClassByName(clazz.getName());
```

Ukázka 29 – Parsování knihovnou QDox z přeložených souborů.

Hlavní rozdíl mezi zdrojovými a přeloženými soubory spočívá v tom, že pro přeložené soubory je nutné získávat naparsované reprezentace `JavaClass` pro každou třídu zvlášť voláním `getClassByName`, zatímco pro zdrojové soubory je dostaneme všechny najednou metodou `getClasses`. V obou případech specializované třídy získají instanci třídy `JavaClass`, která se dále zpracovává stejně. Pokud je získaný soubor rozhraní, práce je u konce. Pokud se jedná o třídu, je potřeba vytvořit těla metod tak, aby byla celá komponenta vytvořena ideálně bez syntaktických chyb a mohla být přeložena překladačem.

7.6 Mockování metod

Jistě by bylo vhodné, aby tento nástroj nabízel podobné vlastnosti jako Mockito a vytvářel **nicer** objekty. Viz kapitola 5.5. Také by bylo vhodné, aby kód vytvořený tímto nástrojem byl přeložitelný a použitelný pokud možno bez jakýchkoliv úprav. Tento problém rozhodně není triviální. I člověk občas dokáže těžko odvodit, jaký je účel metody podle jejího jména, a jak by se daná metoda měla chovat[5]. Pokud programátor používá správně konvence jazyka Java, můžeme snadno odvodit chování takzvaných „getrů" a „setrů". U ostatních metod bohužel lze pouze spekulovat o tom, jaký je jejich záměr, a jejich implementace bude ponechána uživateli tohoto nástroje.

Tento program se tedy pokusí řešit následující problémy:

- **Návratové hodnoty metod stubů** - každá metoda bude vracet nějakou hodnotu, kterou uživatel může specifikovat poskytnutým rozhraním.
- **Správná funkce getrů a setrů** – program bude kontrolovat, zda existuje proměnná, se kterou getr nebo setr pracuje, a pokusí se zrekonstruovat funkci těchto metod.
- **Upozornit uživatele, že metoda potřebuje doplnit implementaci** – v případě, že program není schopen vytvořit tělo metody, vyplní komentář s klíčovým slovem **TO-DO**, kterým informuje uživateli o tom, že je potřeba doplnit implementaci.

K tomu je potřeba vytvořit předem definované návratové hodnoty. Pokud metoda deklaruje návratovou hodnotu, musí za každé okolnosti nějakou hodnotu vrátit.

7.6.1 Návratové hodnoty

V Javě rozlišujeme datové typy na primitivní a referenční. Pro referenční typy je problematika návratových hodnot velice jednoduchá, stačí, aby metoda vrátila jako návratovou hodnotu **null**. V případě primitivních datových typů je situace trochu komplikovanější, protože nejsou reference, nemají prázdnou referenci, a tím pádem ani společnou návratovou hodnotu. Proto je potřeba definovat návratové hodnoty pro všechny skupiny primitivních datových typů – číslo, logickou proměnnou a znak. Základní návratové hodnoty tedy budou následující:

- Číslo – int, byte, double, float, long – 0
- Logická proměnná – boolean – false
- Reference - null

Tyto základní návratové hodnoty neposkytují příliš mnoho funkcionality, proto by bylo vhodné umožnit uživateli definovat svoje vlastní návratové hodnoty pro datové typy. K tomuto účelu bude opět sloužit třída `MockingUtils` a rozhraní `IReturnDataTypeMatcher`. `MockingUtils` bude poskytovat metody `addMatcher` a `removeMatcher`, kterými uživatel může přidávat vlastní návratové hodnoty. První zaregistrovaná hodnota pro datový typ bude vrácena. Pokud nebude nalezena, program zapíše základní návratovou hodnotu uvedenu výše. Zdrojový kód demonstruje Ukázka 30.

[5] Obvzlášť pokud programátor nedodržuje konvence a nazývá svoje metody foo nebo a,b,c,d.

```
for(IReturnDataTypeMatcher matcher : matchers)
{
	String returnCode = matcher.getReturnCodeFor(type);
	if(returnCode != null) return returnCode;
}
if(type.isPrimitive())
{
	writeDefaultPrimitive(type);
}
return "return null;";
```

Ukázka 30 - Získání zaregistrované hodnoty pro datový typ

Jak je vidět z ukázky, rozhraní `IReturnDataTypeMatcher` má za úkol poskytnout zdrojový kód pro typ návratové hodnoty metodou `getReturnCodeFor`. Důležité je si uvědomit, že tato metoda musí vracet přímo zdrojový kód i s klíčovým slovíčkem **return** a středníkem na konci. Zdrojový kód může být obecně jakkoliv složitý, ale měl by být přeložitelný. Kompilační chyby vytvořené špatnou návratovou hodnotou uživatelem definovaných tříd nemohou být dávány za vinu této aplikaci.

V metodě je použita třída QDoxu `JavaType`. Tato třída obsahuje informaci o typu, který je zde uložen jako celé jméno třídy. Pro `String` je to tedy `java.lang.String`.

Tato aplikace používá svoji implementaci této funkcionality pro vracení řetězců - String. Ukázka 31 demonstruje, jak vypadá jednoduchá implementace tohoto rozhraní.

```
public class MyDefaultMatcher implements IReturnDataTypeMatcher
{
   private static Map<Type, String> returnMapping = new HashMap<Type,String>();
   static
   {
       returnMapping.put(new Type("java.lang.String"),
      "return \"Mocked String\";");
   }

   @Override
   public String getReturnCodeFor(Type type)
   {
      return returnMapping.get(type);
   }

}
```

Ukázka 31 – Implementace vlastních návratových hodnot.

7.6.2 Getry a Setry

V Javě existují metody, které se konvenčně píší vždy stejně. Pokud programátor dodržuje konvenci při psaní svého zdrojového kódu, může pak například používat aplikační frameworky s daleko menším úsilím, než kdyby svůj kód psal bez využívání jakýchkoliv standardů. Většina moderních frameworků se spoléhá na to, že jejich uživatel dodržuje základní konvence v psaní zdrojových kódů. Například **Spring** se spoléhá při nastavování objektů na to, že pro objekt s daným názvem bude existovat setr, tedy metoda set pokračující názvem objektu. Na stejnou konvenci se bude spoléhat i tento nástroj.

Podle Java konvence existují prefixy, které značí getr a setr. Jedná se o následující předpony:

- get – getr. Mějme metodu `getTime`, očekáváme, že bude vracet proměnou **`time`**.
- is – getr. Stejný případ jako get ale pro logické proměnné.
- has – getr. Ten samý význam jako is.
- set – setr. Mějme metodu `setTime`, očekáváme, že bude mít jeden parametr, který se bude jmenovat **`time.`** Právě ten uložíme do proměnné **`time`**.

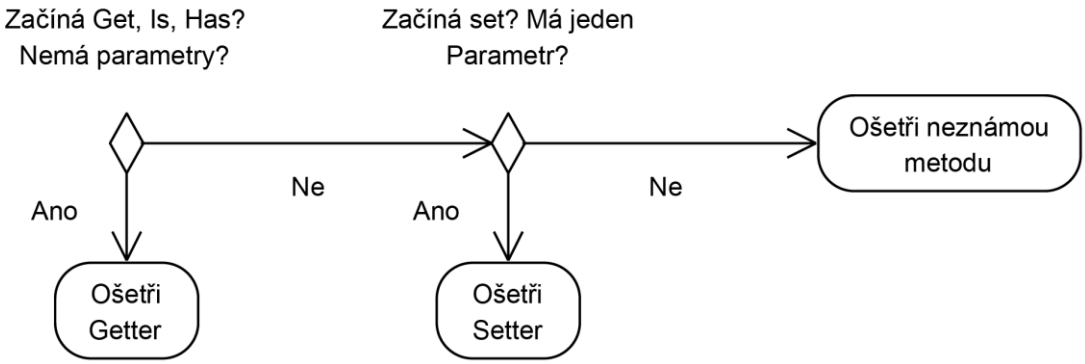
Za getr bude aplikace považovat metodu začínající get, is a has. Metoda nesmí mít žádné parametry. V tomto případě vezme následující znaky za klíčovým slovem a zjistí, zda existuje proměnná se stejným jménem. Pokud ano, vytvoří kód, který vrátí její hodnotu. Pokud ne, vrátí základní návratovou hodnotu pro návratový typ metody.

Obdobná situace bude v případě setru, klíčové slovo je zde ale pouze jedno – set. V tomto případě aplikace očekává, že metoda bude mít právě jeden parametr. Tento parametr bude mít stejné jméno jako následující znaky za klíčovým slovem set. Dále je třeba zkontrolovat, jestli existuje i stejnojmenná proměnná. Vstupní parametr i proměnná musí mít stejný typ. Pokud jsou splněny všechny podmínky, je vložen kód nastavení proměnné na hodnotu poskytnutou jako vstupní parametr. Pokud ne, je vypsána hláška pro uživatele.

7.6.3 Neznámý typ metody

Posledním typem metody jsou všechny metody, které nepatří do předchozích kategorií. Zvláštním případem je ještě konstruktor, který aplikace ignoruje a implementaci nechává na uživateli. Očekává se, že v konstruktoru může být obecně libovolný kód. Je nemožné odhadnout, jak by mělo vypadat tělo takovýchto metod, proto aplikace vyplní do těla metody značku TO-DO tak, aby uživatel věděl, co je potřeba doplnit, a doplní návratovou hodnotu, pokud metoda nějakou má.

Na Obrázku 14 je znázorněno shrnutí, jak jsou metody rozdělovány a řešeny.



Obrázek 14 - Rozdělení metod podle typu

7.7 Vytvoření komponenty

Podle předchozích kroků dokážeme vytvořit stuby pro všechny třídy, které potřebujeme. Stuby mají chování, které očekáváme. Přesto se zatím nejedná o komponentu. K tomu, aby se o ni jednalo, musíme přidat platné manifesty. OSGI manifesty není potřeba měnit, protože funkcionalita komponenty navenek se nesmí změnit jak u simulované, tak u reálné komponenty. V případě Springu se opět manifest u simulované komponenty nebude měnit, u reálné komponenty však bude upraven.

Poté, co se nám podaří vytvořit všechny zdrojové soubory, je třeba zkopírovat soubor Manifest.MF, a pokud jde o simulovanou komponentu i soubor Configuration.xml. U reálné komponenty je situace trochu složitější. Pokud byl její konfigurační soubor změněn, je potřeba uložit jeho novou podobu, pokud ne, jeho stávající. Za tuto funkcionalitu je zodpovědná třída `SpringManifestReader`.

Určitě by bylo vhodné, aby nově vytvořená komponenta šla načíst přímo v Eclipse. K tomu jsou potřeba další soubory, které je nutné buď získat, nebo vytvořit. Jedná se o následující soubory: **build.properties**, **.project** a **.classpath**.

Tyto soubory mohou být teoreticky velice složité. Mohou, ale nemusí, být součástí komponenty. V případě přeložené komponenty do jar souboru většinou není tato informace přístupná. Aplikace se pokusí vytvořit základní typy těchto souborů. Jsou to stejné soubory jako vyrobí Eclipse, pokud vytvoříte nový projekt. Tyto soubory budou uloženy v adresáři **resources** komponenty MockCreator.

Build.properties obsahuje informaci o tom, kde se nachází zdrojové soubory, kde přeložené soubory, a další spousty informací, jejichž přesnou podobu lze nalézt v dokumentaci Eclipse [15]. Záměrně je vynechána informace o tom, jaká verze Javy bude použita. K tomu by sloužilo nastavení `jre.compilation.profile`.

Ukázka 33 demonstruje základní soubor, který je pro naše účely dostačující.

```
source.. = src/
output.. = bin/
bin.includes = META-INF/,\.
```

Ukázka 32 – Základní nastavení v souboru build.properties.

Obdobným problémem je soubor **.classpath**, který obsahuje informace o tom, jaké soubory budou k dispozici pro sestavení projektu. V našem případě nám bude opět stačit nejtriviálnější soubor .classpath. Opět se snažíme vyhnout informaci o přesné verzi Javy.

```
<classpath>
	<classpathentry kind="src" path="src"/>
	<classpathentry kind="con" path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
	<classpathentry kind="con" path="org.eclipse.pde.core.requiredPlugins"/>
	<classpathentry kind="output" path="bin"/>
</classpath>
```

Ukázka 33 – Základní .classpath soubor

U posledního souboru **.project** je situace komplikovanější. Ten jediný bude potřeba modifikovat. Obsahuje totiž informaci o tom, jak se projekt jmenuje. Ukázka 34 zobrazuje takovýto základní soubor.

```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
	<name>framework.extension</name>
	<comment></comment>
	<projects>
	</projects>
	<buildSpec>
		<buildCommand>
			<name>org.eclipse.jdt.core.javabuilder</name>
			<arguments>
			</arguments>
		</buildCommand>
		<buildCommand>
			<name>org.eclipse.pde.ManifestBuilder</name>
			<arguments>
			</arguments>
		</buildCommand>
		<buildCommand>
			<name>org.eclipse.pde.SchemaBuilder</name>
			<arguments>
			</arguments>
		</buildCommand>
	</buildSpec>
	<natures>
		<nature>org.eclipse.pde.PluginNature</nature>
		<nature>org.eclipse.jdt.core.javanature</nature>
	</natures>
</projectDescription>
```

Ukázka 34 – Základní .project soubor s vyplněným jménem pro reálnou komponentu framework.extension.

7.8 Transparentní Proxy

Druhá komponenta, o které pojednává tato práce, se jmenuje TransparentProxy. Jejím úkolem je zaregistrovat a zprostředkovat službu, která bude vkládat transparentní proxy objekty mezi dvě reálné komponenty.

Vzhledem k tomu, že úkolem této služby bude poskytovat pouze factory metodu pro vytvoření proxy objektu, bude k vytvoření OSGI služby nejprve potřeba definovat rozhraní. K vytvoření proxy je potřeba vědět, pro jaký objekt vytváříme proxy, jaký interface služba používá a jaký typ proxy chceme použít. Zdrojový kód rozhraní bude demonstruje Ukázka 35.

```
public interface TransparentProxyFactory
{
	public Object newTransparentProxy(Object bean, final ProxyType type,
		Class<? extends Object> clazz);
	public enum ProxyType
	{
		LOGGING, WAITING
	}
}
```

Ukázka 35 – Rozhraní služby na vytváření transparentní proxy.

Obrázek 10 v kapitole 4.2 zřetelně demonstruje, jak bude vypadat funkčnost proxy. Aby mohl uživatel vytvářet vlastní implementace transparentní proxy, poskytuje aplikace rozhraní `IProxyHandler`, které obsahuje metody `invokeBefore`, `invokeAfter` a `invokeAfterException`. Je tak možné vytvořit vlastní kód, který bude vykonávat nějakou činnost před vyvolanou metodou, po vyvolané metodě a reagovat na výjimku. Poskytované rozhraní zobrazuje Ukázka 36.

```
public interface IProxyHandler
{
	public void invokeBefore(Method m);
	public void invokeAfter(Method m);
	public Object invokeAfterException(Exception e, Method m)
	 throws Exception;
}
```

Ukázka 36 – Rozhraní pro vytváření vlastních Proxy implementací

Důležité je si uvědomit, že volání metod rozhraní jsou blokující. Dokud neskončí metoda `invokeBefore`, nebude delegováno skutečné volání. Dokud neskončí volání `invokeAfter`, nebude předána návratová hodnota. Toho lze využít pro zpomalení průběhu programu nebo pro implementaci komunikačního zpoždění mezi komponentami. Normální návratová hodnota bude předána v případě, že nebude odchycena výjimka. V tomto případě se vyvolá metoda `invokeAfterException`, kde se uživatel může rozhodnout, jestli výjimku pošle dále, nebo vrátí nějakou rozumnou návratovou hodnotu.

Dříve byl tento problém v SimCu řešen psaním komponent typu prostředník, což byla de facto statická implementace proxy. S použitím dynamické proxy odpadá tento problém, stačí napsat pouze vhodnou implementaci rozhraní `IProxyHandler`, které bude vykazovat stejnou funkcionalitu jako celá komponenta typu Prostředník nebo jako více různých komponent tohoto typu.

Problém přichází s grafickým rozhraním frameworku SimCo, který s touto funkcionalitou nepočítá. Aktualizace frameworku tak, aby zobrazoval správně vložené proxy objekty, by ale neměla být nikterak složitá a není součástí této práce.

7.8.1 Jak vytvořit vlastní ProxyHandler

V předchozí kapitole bylo řečeno, jak se vytváří transparentní proxy a jak funguje. Očekává se, že uživatel si bude chtít napsat implementaci metod, které se volají před delegováním volání, po něm a v případě kdy během delegování dojde k výjimce. Stačí tedy vhodně naimplementovat metody zobrazené v Ukázce Ukázka 36. Pokud máme napsanou implementaci `ProxyHandleru`, musíme ji zaregistrovat. Všechny dostupné implementace reprezentuje `Enum ProxyType`, který je součástí definice `TransparentProxyFactory`. V něm musíme definovat nový typ, viz Ukázka 37.

```
public enum ProxyType
{
	LOGGING, WAITING, THENEWONE
}
```

Ukázka 37 – Registrace vlastní transparentní proxy do enumu.

Dále musíme doplnit informaci o nové proxy do switche uvnitř implementace služby. Moje implementace služby se nazývá `DelegationFactory` a obsahuje metodu, která podle hodnoty enumu ProxyType vrátí instanci požadované proxy. Tato metoda se jmenuje `loadProxyHandlerByType` a zaregistrování nové proxy demonstruje Ukázka 38.

```
private IProxyHandler loadProxyHandlerByType(ProxyType type)
{
	switch(type)
	{
		case LOGGING : return new LoggingHandler();
		case WAITING : return new WaitingHandler();
		case THENEWONE : return new TheNewProxy();
		default : return new LoggingHandler();
	}
}
```

Ukázka 38 – Registrace vlastní transparent proxy ve službě.

Poslední možností je nastavit novou transparentní proxy jako defaultní. K tomu stačí změnit hodnotu `PROXY_TYPE` ve třídě `SpringManifestReader`. Typ proxy se dá také snadno změnit ve Spring konfiguračním souboru, stačí do argumentu typ zadat název proxy v enumu. V tomto případě tedy místo `LOGGING` hodnotu `THENEWONE`.

7.9 Uživatelské rozhraní

Komponenta `MockCreator` obsahuje základní uživatelské rozhraní, které se snaží být minimalistické a snadno použitelné. Jedná se o tři jednoduché dialogy, které umožní nastavit uživateli, s jakými komponentami má pracovat, odkud je má načítat a kam je má ukládat.

Grafické uživatelské rozhraní se spouští samo ihned po nainstalování komponenty do OSGI kontejneru. To umožňuje framework Spring specifikací `init` metody v konfiguračním souboru. Zaregistrována je metoda `execute` třídy `SpringMocker`. Tato metoda by se dala nazvat jako jádro celého programu. Její příklad najdete v Ukázce 39.

```
File readPath = getPathFromDialog(SEARCH_PATH);
if(readPath == null)
{
	return;
}
readDirs(showTable(readPath));
if(saveDialog())
{
	mockIt();
}
```

Ukázka 39 - Hlavní metoda třídy SpringMocker

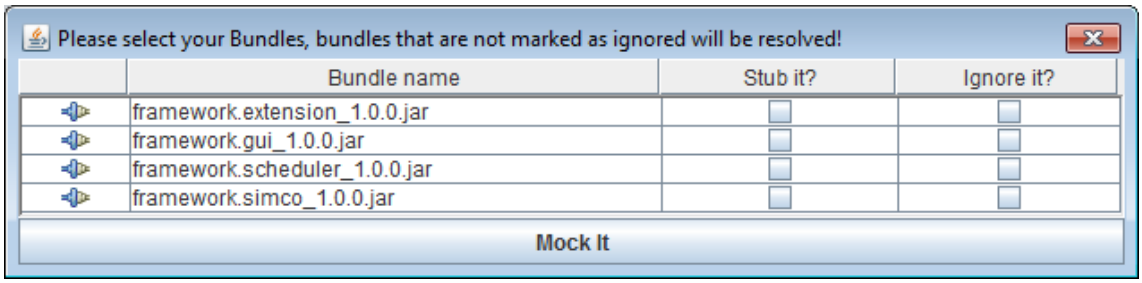
Jak je z kódu patrno, je potřeba od uživatele získat tři zásadní informace:

1. Odkud máme načíst komponenty.
2. Jaké komponenty se mají simulovat, jaké mají být reálné, případně jaké se mají ignorovat.
3. Kam se mají vytvořené komponenty uložit.

Pro první a třetí úkol slouží jednoduché dialogy, které poskytuje třída `JFileChooser`, protože očekáváme jako vstup adresář, je dialog zapnut pouze v adresářovém režimu.

Zajímavější komponentou je tabulka, která vzniká načtením informací v poskytnutém adresáři. Aplikace načte všechny jar soubory a pomocí třídy JarReader hledá jejich reprezentaci v OSGI kontejneru nebo se je pokusí nainstalovat. Více v kapitole 7.4. Teoreticky je tedy možné použít prázdné jar soubory se správným symbolickým jménem. Aplikace primárně získává data z OSGI kontejneru.

Pokud je načtena množina komponent, je zobrazena tabulka, v níž uživatel může vybrat, co se s načtenou komponentou bude dít dále. Existují tři možnosti – komponenta se buď bude simulovat, nebo budou vloženy transparentní proxy, nebo bude ignorována. Tyto možnosti lze nastavit v přehledné tabulce viz Obrázek 15.



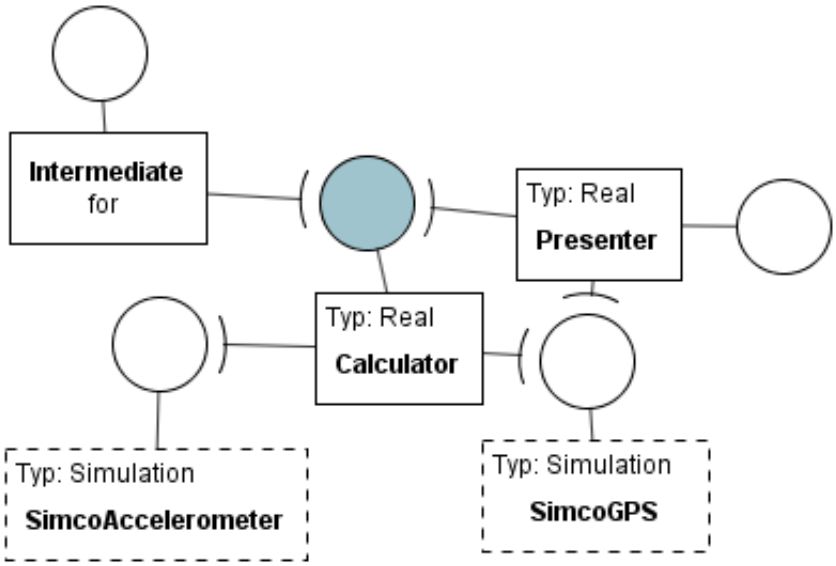
Obrázek 15 - Tabulka pro nastavení komponent

Tabulka kontroluje volbu uživatele, takže není možné zaškrtnout pro jednu komponentu obě políčka zároveň. Sloupeček „Stub it?“ označuje, že se komponenta bude simulovat. Pouze kód, který je použitelný zvenku, bude exportován. Možnost „Ignore it?“ značí, že s komponentou nebude prováděna žádná operace, tudíž pro ni nebude vytvořen projekt, ani nebude uložena na disk. Komponenta, která nemá zaškrtnutu ani jednu z možností, je považována za komponentu reálnou. Pro ni bude zjištěno, zda poskytuje službu, kterou používá jiná reálná komponenta. Pokud ano, bude mezi ně vložena transparentní proxy, budou tedy změněny její manifesty.

Po kliknutí na tlačítko „Mock it“ se uživateli zobrazí dialog s žádostí o udání složky, kam se vytvořené projekty uloží. Po poskytnutí této informace aplikace vytvoří na daném místě adresářovou strukturu, kde jedna složka odpovídá jednomu projektu, jedné komponentě. Uživatel nyní může vytvořené komponenty naimportovat jako existující projekty do IDE Eclipse.

8 Ověření Implementace – uživatelský manuál

Pro ověření funkčnosti implementace jsem využil data, která vytvořil Tomáš Kabíček ve své diplomové práci. Scénář zachycuje Obrázek 16.

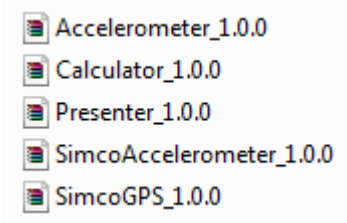


Obrázek 16 - Scénář z diplomové práce Tomáše Kabíčka

Scénář obsahuje dvě komponenty reálné, dvě komponenty simulované a jednu komponentu typu prostředník. Simulaci lze ověřit tak, že použijeme komponenty poskytnuté Tomášem Kabíčkem a zkusíme je znovu vygenerovat tímto nástrojem. Pokud aplikace dokáže načíst podobný scénář, bude považována za správně fungující. Na scénáři vytvořeném touto aplikací nebude komponenta typu prostředník, protože ta je nyní implementována dynamicky, nikoliv staticky. Komponenta Intermediate bude tím pádem chybět na obrázku úplně.

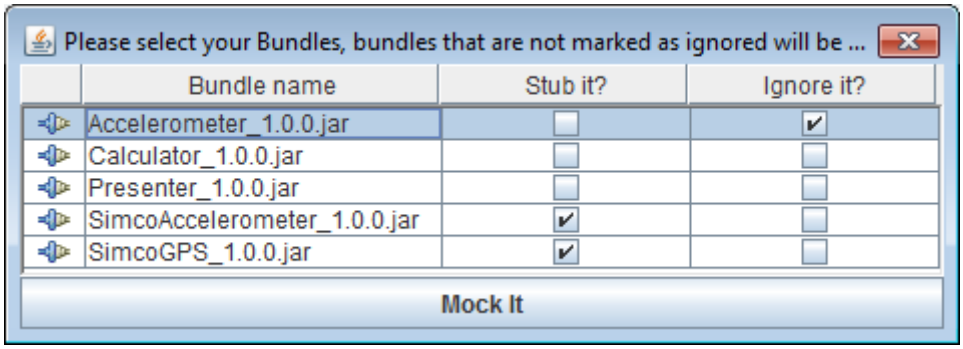
Jak lze ověřit, že funguje transparentní proxy? Aplikace zatím poskytuje obyčejnou logovací proxy, která pokaždé, když je zavolána nějaká metoda, vypíše informaci před a po jejím zavolání. Pro ověření funkčnosti se stačí podívat do konzole, zda obsahuje tuto informaci. Dalším ověřením je použití debug perspektivy v Eclipse. Pro tuto práci se spokojíme s výstupem do konzole.

Pro začátek exportujeme potřebné knihovny do jar souborů. Jedná se o komponenty Calculator, Presenter, SimcoAccelerometr a SimcoGPS. Tyto jar soubory uložíme do definovaného adresáře, který bude pak vypadat zhruba jako na Obrázku 17.



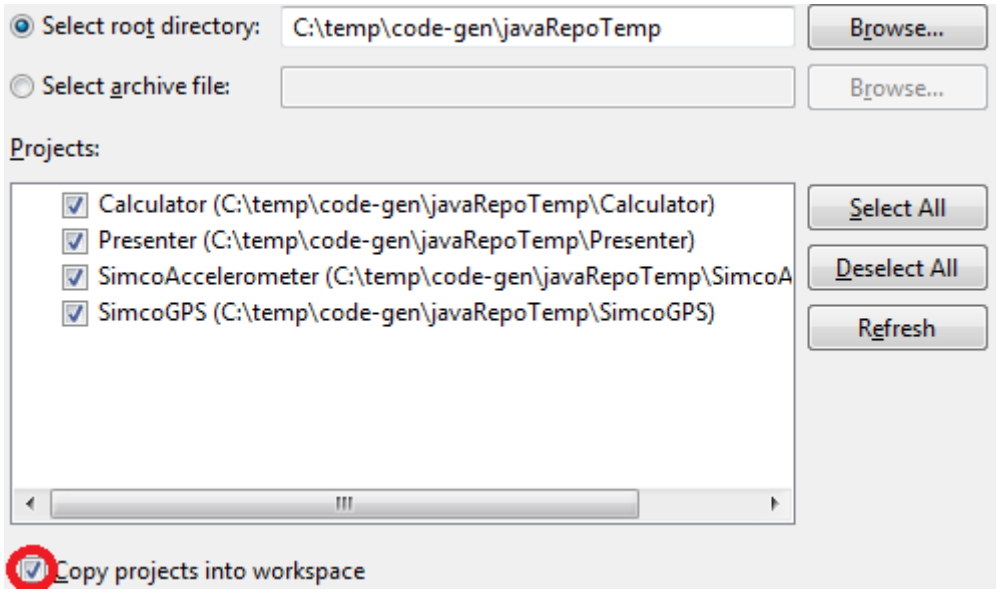
Obrázek 17 - Adresář s komponentami testovacího scénáře

Dále spustíme aplikaci a zajistíme, aby všechny zmíněné komponenty byly v launch konfiguraci. V dialogu vybereme složku, kam jsme uložili komponenty. V tabulce nastavíme checkboxy podle námi požadovaného scénáře jako na Obrázku 18.



Obrázek 18 - Nastavení komponent scénáře

Stisknutím tlačítka „Mock It“ vyvoláme ukládací dialog, kde vybereme cestu, kam se uloží komponenty vytvořené aplikací. Po skončení aplikace můžeme naimportovat do IDE vytvořený scénář. Projekty lze importovat pouze v tom případě, že ještě neexistují v aktuálním prostředí. Proto doporučuji pro testování použít nový workspace. Stisknutím **File → Import → Existing projects to workspace** získáme dialog, kde zadáme cestu k adresáři, který jsme poskytli aplikaci jako výstupní. Příklad úspěšného importu můžete vidět Obrázku 19. Silně doporučuji zaškrtnout možnost **copy projects into workspace**, pokud nepoužíváte adresář, kam jste exportovali data jako workspace.



Obrázek 19 - Import vygenerovaných projektů do Eclipse

Po importu projektů zjistíme, že aplikace nejde přeložit. To je způsobeno chybějícím zdrojovým kódem uvnitř metod. Nástroj upozorňuje uživatele komentáři, kde je potřeba doplnit zdrojový kód. Typicky jde o komentáře a o metody, které nástroj nezná. Dobrým příkladem je metoda getVector, kde aplikace předpokládá, že tato metoda vrátí hodnotu atributu vector. Ta však ve třídě neexistuje, proto vrátí null a očekává, že uživatel nepíše vhodnější chování, viz Ukázka 40.

```
public Accelerometer(simco.framework.scheduler.calendar.ICalendar calendar,
simco.framework.extension.ISimCoContext simcoContext,
simco.framework.simco.IEventCreator eventCreator)
{
	//TO-DO fill a construct
}

public java.util.Vector getVector()
{
	//WARNING NO FIELD LIKE vector found, you are breaking convention
	return null;
}

public void setVector(int x, int y, int z)
{
	//TO-DO: Fill me in
}
```

Ukázka 40 - Vygenerovaný kód třídy Accelerator

Scénář od Tomáše Kabíčka bohužel používá k realizaci transparentní proxy LDAP filtry [4]. Tímto způsobem vlastně vytvářel transparentní proxy. Pokud bychom bez úprav spustili aplikaci, služby by vůbec nebyly nalezeny, protože jsou deklarovány v komponentě CalculatorIntermediate. Tím pádem musíme změnit konfigurační soubory springu, kde se deklarují OSGI služby. Smazáním atributů filter docílíme toho, že aplikace bude fungovat správně s dynamickou transparentní proxy. Proto smažeme v souboru presenterDefinition.xml atribut filter, viz Ukázka 41.

```
<osgi:reference id="Calculator"
interface="simco.application.calculator.ICalculator" filter="(simul=true)" />
```

Ukázka 41 Pro funkčnost dat od Tomáše Kabíčka je potřeba smazat LDAP filter.

Po doplnění potřebného kódu můžeme spustit aplikaci SimCo. Pokud jsme správně naimplementovali požadované metody, neměla by vyskočit žádná výjimka. Již během startu aplikace uvidíme v konzoli, že služba Calculator probíhá přes proxy. Komponenta presenter totiž periodicky vypisuje aktuální stav metodou `refreshDisplay`. K tomu používá právě službu `Calculator`, kterou poskytuje komponenta `Calculator`. Vzhledem k tomu, že jsou obě komponenty reálného typu, služba `Calculator` byla vytvořena jako proxy mezi oběma objekty. Pokud se tedy podíváme do konzole, uvidíme výpis proxy, viz Ukázka 42.

```
This was invoked before the method getWalkedDistance
This was invoked after the method getWalkedDistance
Walked distance [m]: 1716343
This was invoked before the method getStepsOnMeter
This was invoked after the method getStepsOnMeter
Steps on one meter: 1.42
This was invoked before the method getBurnedKCal
This was invoked after the method getBurnedKCal
Burned Kilo Calories: 171636
```

Ukázka 42 – Potvrzení že služba Calculator je za transparentní proxy

Pokud se podíváme na zdrojový kód služby Presenter, zjistíme, že opravdu všechna volání jsou zachycena proxy. Pokud by vznikla nějaká chyba ve službě Calculator, logovací proxy ji zachytí.

9 Závěr

V rámci této diplomové práce se podařilo vytvořit nástroj, který ulehčí práci uživatelům frameworku SimCo. Aplikace není všemocná, a ani to není její ambicí, na rozdíl od snahy být co nejvíce rozšiřitelná a platformě nezávislá. Jistě by bylo možné tuto práci rozšířit, aby vytvářela kód simulovaných komponent tak, aby uživatel nemusel dopisovat ručně prakticky nic, ale podle mého názoru by vynaložená práce vysoce převýšila uspořený čas pro uživatele. Zdrojový kód v metodě může být de facto jakýkoliv a původní sémantika autora také, proto vytvořit univerzální nástroj pro rekonstrukci zdrojového kódu je dle mého názoru nemožné.

Jistě existují chyby a limitace v této aplikaci. QDox například neumí pracovat s generikami, tudíž generovaný kód obsahuje varování. Stejně tak píše celé jméno třídy místo krátkých jmen, čímž činí zdrojový kód méně přehledným. O těchto problémech ale vývojáři QDoxu ví a mohly by být opraveny ve verzi 2.0.

Rizikem této aplikace je opět QDox, protože pokud tento projekt skončí, veškeré úpravy parseru pro novou funkcionalitu, například při upgradu na novou verzi Javy, budou muset být naprogramovány uživateli SimCa.

Aplikace nemusí být použita pouze pro generování scénářů, také by se dala použít jako komponentový API dekompilátor a pomocník při agilních metodikách.

Služba generující transparentní proxy by mohla být implementována frameworkem EasyMock, konkrétně jeho částečným mockováním. Tím pádem by instance služby mohla být přetypována na konkrétní implementaci. V této práci je použita klasická třída Proxy, kterou obsahuje Java, protože OSGI specifikace jasně říká, že služby by se měly používat jako rozhraní a neměly by být přetypovávány na konkrétní implementaci. [1]

Myslím si, že aplikace splnila očekávání, se kterými byla vyvíjena. Nástroj je použitelný, funguje, a tím pádem bylo splněno zadání diplomové práce.

Bibliografie

[1] J. McAffer, P. VanderLey a S. Archer, OSGI and Equinox: Creating Highly Modular Java™ Systems, Boston: Addison Wesley, 2010.

[2] OSGI Alliance, „The OSGi Architecture," OSGI Alliance, 1 1 2013. [Online]. Available: http://www.osgi.org/Technology/WhatIsOSGi. [Přístup získán 15 4 2013].

[3] E. R. Johnson a B. Foote, „Designing Reusable Classes," *Journal of Object-Oriented Programming,* pp. 22-35, 1988.

[4] T. Kabíček, Simulační systém softwarových komponent, Plzeň: Západočeská Univerzita v Plzni, Fakulta Aplikovaných Věd, Katedra Informatiky a Výpočetní Techniky, 2011.

[5] S. Racek, Pravděpodobnostní modely počítačů, Plzeň: Vydavatelství ZČU, 2002.

[6] S. Freeman, „Brief History of mock objects," 8 09 2009. [Online]. Available: http://www.mockobjects.com/2009/09/brief-history-of-mock-objects.html. [Přístup získán 2013 4 13].

[7] T. Mackinnon, S. Freeman a P. Craig, „Endo-testing: unit testing with mock objects," *Extreme programming examined,* pp. 287 - 301, 2001.

[8] S. Freeman, T. Mackinnon, N. Pryce and J. Walnes, "Mock Roles, objects," *OOPSLA '04 Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications,* pp. 236-246, 2004.

[9] S. Freeman, T. Mackinnon, N. Pryce, M. Talevi a J. Walnes, „JMock," JMock, 19 12 2012. [Online]. Available: http://jmock.org. [Přístup získán 13 4 2013].

[10] T. Freese a H. Tremblay, „EasyMock Documentation," 1 1 2001-2012. [Online]. Available: http://www.easymock.org/Documentation.html. [Přístup získán 13 4 2013].

[11] S. Faber, "Mockito," 14 Leden 2008. [Online]. Available: http://monkeyisland.pl/2008/01/14/mockito/. [Accessed 15 Duben 2014].

[12] Mockito, „mockito Simple & Better mocking," 6 Červen 2012. [Online]. Available: http://code.google.com/p/mockito/. [Přístup získán 15 Duben 2013].

[13] Thoughtworks, „Overview," Thoughtworks, 28 Listopad 2010. [Online]. Available: http://qdox.codehaus.org. [Přístup získán 15 Duben 2013].

[14] F. P. Brooks, „No Silver Bullet — Essence and Accident in Software Engineering," *Proceedings of the IFIP Tenth World Computing Conference,* 1986.

[15] Eclipse Foundation, „Eclipse documentation - Current Release," Eclipse Foundation, [Online]. Available: http://help.eclipse.org/juno/index.jsp. [Přístup získán 15 4 2013].

[16] R. Johnson, J. Hoeller, A. Arendsen a S. Colin, „The Spring Framework - Reference Documentation," Spring Source Community, [Online]. Available: http://static.springsource.org/spring/docs/2.5.x/reference/. [Přístup získán 13 Duben 2013].

[17] N. Pryce a S. Freeman, „Evolving an Embedded Domain-Specific Language in Java," sv. 1, č. 1, 2006.

Seznam ukázek a obrázků

Seznam ukázek zdrojových kódů

Seznam použitých obrázků

Slovníček pojmů

Bundle – plugin, komponenta v OSGI.

OSGI – akronym pro Open Services Gateway initiative, komponentový framework, viz [2]

JAR – java archive. Bundly v OSGI bývají uloženy právě v tomto formátu. Je to komprimovaný soubor v zip formátu.

ZIP – bezztrátový kompresní formát.

XML – Extensible Markup Langugage. Lidsky čitelný formát pro výměnu dat mezi aplikacemi.

IOC – Inversion of Control. Úzce spjato s Dependency Injection. Hlavní stavební kámen aspektově orientovaného programování a frameworku Spring, viz kapitola 3.1.

DI – Dependency Injection. Realizace IOC, viz kapitola 3.1.

API – Application Programming Interface. Rozhraní, které poskytuje aplikce.

JDK - Java Development Kit (Soubor základních nástrojů pro vývoj aplikací pro platformu Java)

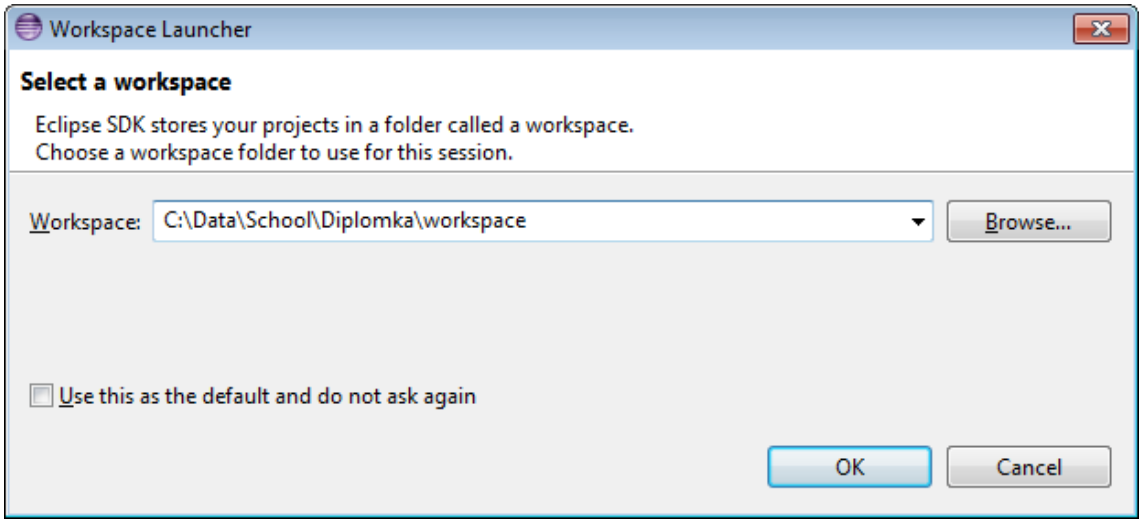
LDAP - Lightweight Directory Acces Protocol (Protokol pro přístup k datům)

DVD – Optický disk s větší kapacitou než cd. Obvykle 4.2 GB. K této práci je přiloženo právě jedno DVD.

Uživatelská příručka

Vzhledem k tomu, že aplikace ke svému spuštění potřebuje poměrně složité prostředí, na DVD, přiloženém k této práci, je poskytnuto celé vývojové prostředí včetně workspace. Poskytnuta bude jak 32 bitová, tak 64 bitová Eclipse. Předpokladem je, že uživatel má správně nastavené cesty k JDK 1.7.

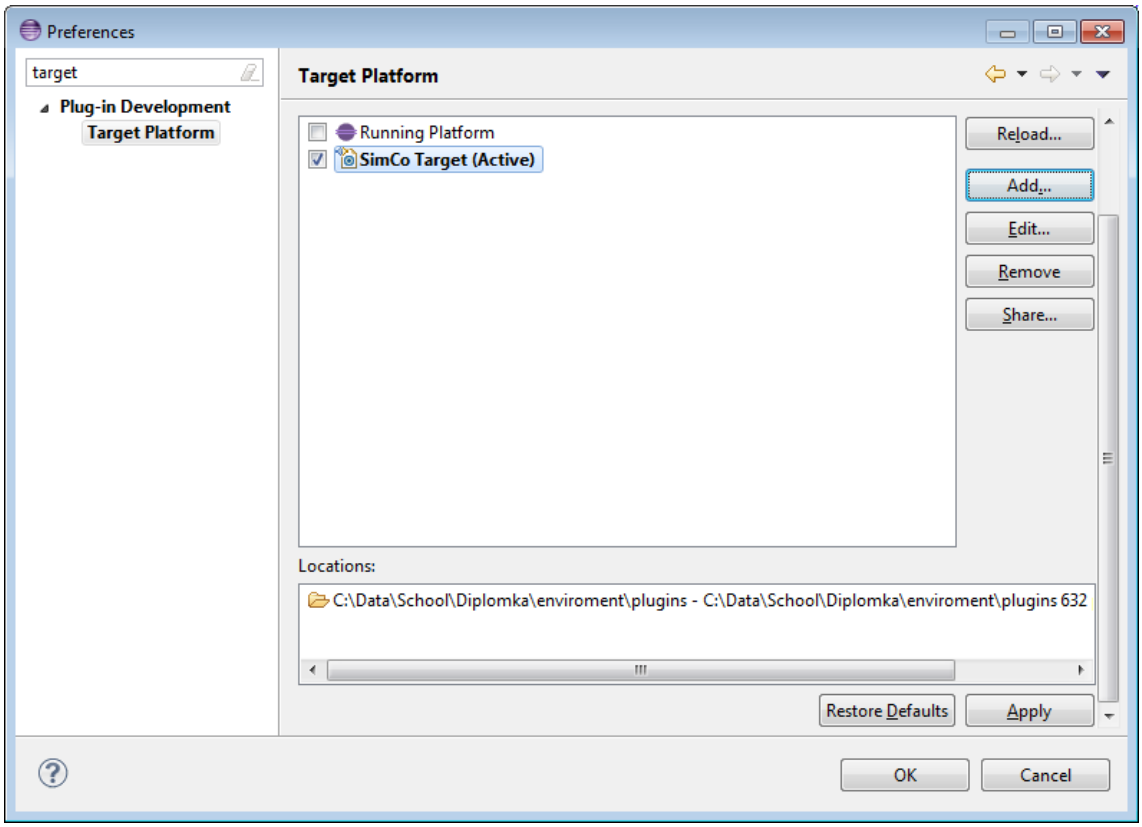
Eclipse spustíme souborem **Eclipse.exe** umístěným v adresáři **Environment32bit** nebo **Environment64bit**. Při spuštění Eclipse vybereme cestu k workspace. Jak demonstruje Obrázek 20.



Obrázek 20 - Vybrání workspace

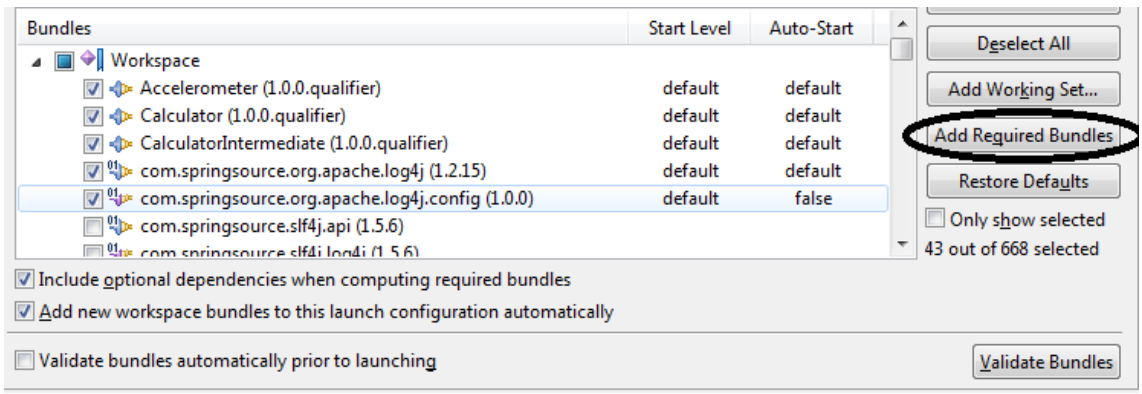
Dále je potřeba ověřit, že je nastavena správná platforma. Informaci o platformě nalezneme po kliknutí na **Window → Preferences → Plug-in Development → Target Platform**. Zde je potřeba nastavit cestu k adresáři **target platform**, poskytnutém na přiloženém DVD.

Obrázek 21 demonstruje jak správně nastavit platformu. Nová platforma musí obsahovat pouze jednu složku a to právě tu, která je poskytnuta na DVD, přiloženém k této práci.



Obrázek 21 - Nastavení Target Platform

Dále je potřeba nastavit takzvanou Launch Configuration. Eclipse obsahuje tlačítko pro spuštěni v normálním nebo debug módu. Po kliknutí na šipku obsahuje možnost **Run Configuration** nebo **Debug Configuration**. Po kliknutí na konfiguraci se objeví dialog, ve kterém je potřeba nastavit jaké komponenty budou k dispozici pro běh programu. Důležité je zaškrtnout všechny komponenty začínající **simco** a dále všechny komponenty, které jsou nutné pro běh scénáře. Plugin org.springframework.osgi.extender je také nutný pro běh aplikace, jinak nebudou přečteny konfigurační soubory Springu. Poté stačí kliknout na tlačítko **Add Required Bundles** a budou přidány všechny komponenty, které jsou požadovány vybranými komponentami, viz Obrázek 22.



Obrázek 22 - Vybrání komponent v Launch Configuration

Nyní je vše nakonfigurováno pro spuštění programu. Všechny další potřebné informace naleznete v kapitole 8.