Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vývoj aplikácií v prostredí Spring Framework

Bakalárska práca

2013

Filip Marek

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vývoj aplikácií v prostredí Spring Framework

Bakalárska práca

Študijný program: Informatika
Študijný odbor: 2508 Informatika
Školiace pracovisko: Katedra informatiky FMFI
Vedúci práce: Mgr. Jaroslav Budiš

Bratislava, 2013
Filip Marek

62382975

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Filip Marek
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

Názov: Vývoj aplikácií v prostredí Spring Framework.

Cieľ: Preskúmať možnosti moderných frameworkov nad jazykom Java. Využitím týchto technológií navrhnúť a implementovať aplikáciu na vizualizáciu výsledkov RNA-Seq analýz.

Vedúci: Mgr. Jaroslav Budiš
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 08.10.2012

Dátum schválenia: 11.10.2012 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie

Ďakujem môjmu vedúcemu Mgr.Jaroslavovi Budišovi za jeho pomoc a čas, ktorý mi venoval. Taktiež za dobrý nápad na prácu, vďaka ktorej som si mohol vyskúšať veľa praktických vecí v jazyku Java.

Abstrakt

Práca sa zaoberá skúmaním možností využitia vybraných moderných frameworkov nad jazykom Java. Opíšeme základné princípy fungovania týchto frameworkov, ako aj výhody, ktoré nám poskytujú.

Táto práca má za cieľ pomôcť programátorom s voľbou vhodných podporných frameworkov pre jazyk Java na vývin komplexnej aplikácie s GUI. Tiež im má pomôcť pri prvotnej implementácii a pri zoznámení sa s danými nástrojmi.

Praktickou časťou bakalárskej práce je vývoj aplikácie na vizualizáciu výsledkov RNA-Seq analýz, ktorá poslúži ako základ pre ďalší vývoj väčšieho bioinformatického programu.

Kľúčové slová: Java, framework, Spring, Maven

Abstract

The present thesis deals with the possibilities of the use of modern frameworks above the Java programming language. We will describe basic principles of functioning of these frameworks, as well as the advantages provided by them.

The purpose of this thesis is to help programmers with the choice of appropriate supportive frameworks for the Java programming language for development of a complex application with GUI. Besides that, it should help them with the primary implementation and familiarization with the given tools.

The practical part of the paper aims at development of an application for visualization of the results of RNA-Seq analyses. It will serve as a basis for the further development of a superior bioinformatics program.

Keywords: Java, framework, Spring, Maven

Obsah

Zoznam obrázkov

Úvod

Jedným z najpoužívanejších programovacích jazykov dnešnej doby je objektovo orientovaný jazyk Java. Pre uľahčenie a zefektívnenie vývoja softvérových aplikácií bolo pre programátorov vytvorených veľa rôznych podporných prostriedkov - frameworkov.

Cieľom tejto bakalárskej práce je spoznať výhody a nevýhody vybraných technológií a frameworkov. Poskytneme ich popis v takej miere, aby bolo možné porozumieť ich princípom a základným funkčným vlastnostiam. Okrem toho ukážeme reálne fungovanie týchto frameworkov pri implementácii aplikácie na vizualizáciu výsledkov RNA-Seq analýz z oblasti bioinformatiky.

Táto práca bude slúžiť ako hlavný základ pre ďalšiu tvorbu väčšieho bioinfromatického softvéru. Dôležitou úlohou bude zistiť, ktoré z preskúmaných frameworkov sa dajú použiť aj v budúcnosti. Ich vhodným výberom si môžeme veľmi uľahčiť a zrýchliť prácu, no na druhej strane sa môže stať, že zlou voľbou si vytvoríme veľa zbytočných problémov. Potom aj zmena použitého frameworku nemusí byť úplne jednoduchá.

V prvej kapitole si definujeme základné bioinformatické pojmy a požiadavky na náš vyvíjaný softvér. Vďaka tomu sa potom budeme vedieť rozhodnúť, ktoré nástroje budú pre nás užitočné. V druhej kapitole si rozoberieme, ako začať s prípravou na väčší projekt a uvidíme, ako nám môžu UML diagramy uľahčiť prácu. Ďalšia kapitola hovorí o spolupráci viacerých programátorov. V štvrtej kapitole ukážeme pridávanie iných tried, knižníc alebo frameworkov pomocou závislostí. V piatej časti predstavíme niektoré z hlavných súčastí Spring frameworku. Popíšeme jeho koncept kontajnera a vytvárania závislostí medzi triedami a tiež oddeľovanie a pridávanie rôznych vrstiev aplikácie. V poslednej kapitole prezentujeme použitie najnovšieho Java frameworku na tvorbu grafického rozhrania.

Kapitola 1

Popis problematiky

Spolupráca biológov a informatikov je čoraz viac výraznejšia a dôležitejšia. Obzvlášť veľké množstvá dát majú biológovia z oblasti genetiky. Pri ich spracovávaní počítačom je potrebné používať rýchle a efektívne algoritmy.

Genetické informácie popisujú celý živý organizmus - jeho stavbu tela, veľkosť, sfarbenie. Túto informáciu uchováva deoxyribonukleová kyselina (DNA). Na jednom z vlákien DNA vzniká počas transkripcie ribonukleová kyselina (RNA), ktorá je jej zjednodušenou kópiou. Vzniknutá RNA môže niesť informáciu o stavbe proteínov, regulovať génové expresie alebo transportovať aminokyseliny k ribozómu.

Štruktúru DNA tvorí dlhá sekvencia dusíkatých báz: adenín (A), cytozín (C), tymín (T) a guanín (G). Môžeme ju teda reprezentovať ako slovo vytvorené z abecedy $\{A, C, T, G\}$. Z hľadiska biológie je zaujímavé zobrazenie tejto dlhej sekvencie ($\sim 10^9$ báz) a kratších úsekov - readov ($\sim$ 100 báz) poukladaných a zarovnaných nad ňou. Miesta, kde nastáva transkripcia, sú určené anotáciami. V nich sú zaznamenané hranice transkripčného úseku na sekvencii.

1.1 Návrh aplikácie

Vytvorili sme podklad pre väčší projekt, pomocou ktorého načítavame veľké objemy dát sekvencií a k nim príslušné ready a anotácie. Takto načítané a spracované vstupné údaje vizualizujeme v grafickom rozhraní.

Pri prípravách na tvorbu našej aplikácie sme prešli viacero požiadaviek, ktoré sa pokúsime počas tvorby zohľadniť:

1. Využitím frameworkov chceme uľahčiť a zrýchliť vývoj projektu.

2. Chceme umožniť jednoduché rozšírenia programu, najmä pridávanie nových formátov vstupných súborov.

3. Na projekte bude občas spolupracovať viac ľudí, a preto je dôležité navrhnúť taký koncept, aby si navzájom nezasahovali do svojej práce.

4. Potrebujeme oddeliť vizualizáciu (grafické rozhranie) od dátového modelu.

5. Chceme mať možnosť pridávať do aplikácie rôzne podporné vrstvy bez toho, aby sme museli zasahovať do hlavného jadra.

6. Bude nutné vytvoriť efektívne spracovávanie vstupných súborov, keďže objem týchto dát bude veľký ($\sim$ 2 - 5 GB).

7. Po vytvorení tried jadra aplikácie ich chceme sprístupniť na ďalšie použitie aj pre iných programátorov.

Aplikáciu sme rozdelili do viacerých častí, aby sme ich neskôr mohli používať v iných projektoch aj samostatne. V časti "fmfiuk.biotools.core" sme vytvorili hlavné triedy na načítavanie a spracovávanie súborov pre rôzne očakávané vstupy. Pripojili sme "fmfiuk.biotools.services", kde tieto triedy jadra pospájame a pripravíme na použitie. Ako poslednú časť sme pridali "fmfiuk.biotools.gui", teda grafické prostredie pre používateľov na výsledné zobrazovanie sekvencií, anotácií a zarovnaní.

Kapitola 2

UML Lab

2.1 UML (Unified Modeling Language)

"Modelovací jazyk UML je grafickým jazykom na návrh a zobrazenie štruktúry programu. UML je súhrnom predovšetkým grafických notácií (diagramov) k vyjadreniu analytických a návrhových modelov."[1] Tieto diagramy sú ďalej využívané ako podporný nástroj pri komunikácii medzi architektmi a programátormi pri tvorbe programov a spresňovaní požiadaviek. Vizuálnym návrhom dokážeme veľakrát lepšie pripraviť koncept celého programu. Ľahko ním totiž definujeme dedičnosť, polymorfizmus alebo zapuzdrenie.

Vďaka presnému vyjadrovaniu pri návrhoch a voľbe vhodného modelovacieho nástroja môžeme potom priamo z diagramov generovať štrukturovaný kód v podobe tried a interfacov, ich konštruktorov, metód a vzťahov medzi nimi.

2.2 Modelovacie prostredie

Pri prípravách na väčší projekt je vhodné začať práve s návrhom pomocou UML schém. Na našu prácu sme si preto vybrali UML Lab, ktorý je komplexným softvérom pre programátorov a architektov na vývoj a modelovanie takýchto UML diagramov. Tento produkt sa dá pridať do vývojového prostredia Eclipse. Štandardne je možné generovať kód do jazykov Java a PHP. UML LAB ponúka aj možnosť vytvoriť si vlastné štýly.

Používanie tohto programu nám prinieslo viacero výhod:

1. Z diagramov sa generujú základné zdrojové kódy pre všetky triedy a ich metódy.

2. Medzi diagramami (triedami) sa dajú vytvárať vzťahy.

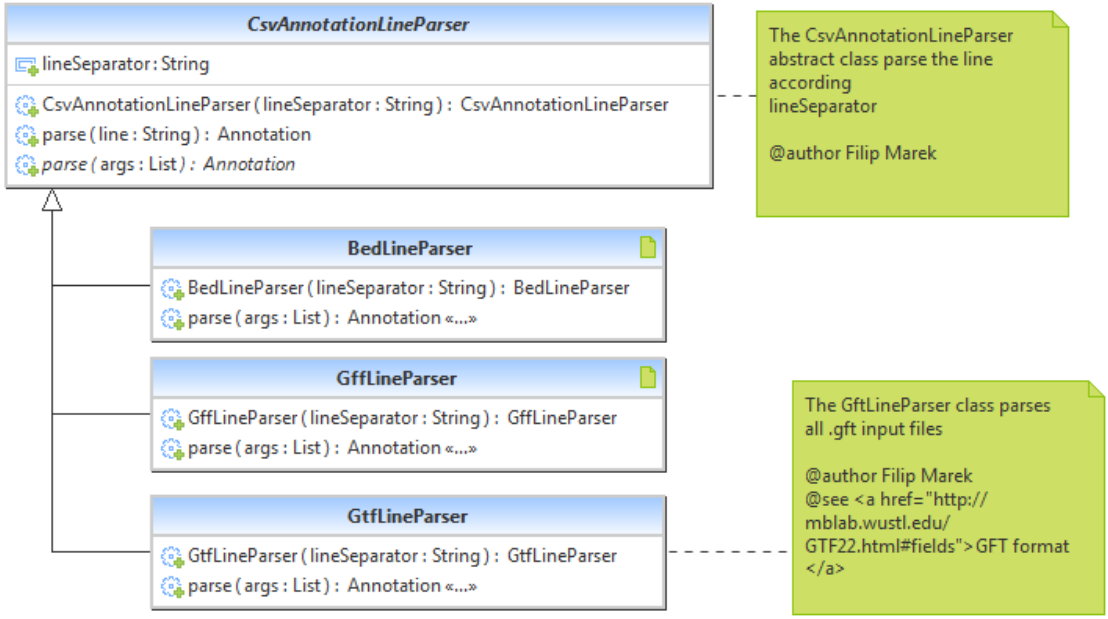
3. Pre všetky objekty a ich metódy sa dá dopĺňať dokumentácia.

4. Atribútom a metódam sa dá nastavovať viditeľnosť (public, protected, private) a modifikátory (static, abstract).

5. Je podporovaný spätný proces - vytváranie UML diagramov z kódu.

Každú z tried jadra našej aplikácie sme najprv vymodelovali a tak sme si jednoduchšie premysleli potrebnú funkcionalitu týchto objektov. Na načítavanie anotácií zo súborov sme vytvorili abstraktnú triedu *CsvAnnotationLineParser*. Táto trieda mala definovanú abstraktnú metódu *parse*. K tomu sme pridali tri triedy pre načítavanie z konkrétnych typov súborov podľa ich prípon (.bed, .gff, .gtf), ktoré rozširujú abstraktnú triedu (obr. 2.1).

Obr. 2.1: UML diagram pre načítavanie podľa súborov



Z diagramov sme vygenerovali štandardné časti zdrojových kódov aj s dokumentáciou (obr. 2.2) a vyhli sme sa tak neveľmi zaujímavej práci - písaniu deklarácií tried. Do týchto tried sme potom implementovali funkcionalitu podľa našich potrieb.

Obr. 2.2: Vygenerovaný kód objektu GtfLineParser

```
/**
 * The GftLineParser class parses all .gft input files
 *
 * @author Filip Marek
 * @see <a href="http://mblab.wustl.edu/GTF22.html#fields">GFT format</a>
 */
public class GtfLineParser extends CsvAnnotationLineParser {

    public GtfLineParser(String lineSeparator) {
        super(lineSeparator);
    }

    /*
     * (non-Javadoc)
     *
     * @see
     * fmfiuk.biotools.core.parser.annotion.CsvAnnotationLineParser#
     * parse(java.util.List)
     */
    @Override
    public Annotation parse(java.util.List<String> args) {
        return null;
    }

}
```

Kapitola 3

Verziovací systém

3.1 Nevyhnutnosť spolupráce

Keď pracujú na projekte viacerí programátori, spolupráca a vzájomná komunikácia sú jednou z kľúčových oblastí, na ktoré si musia dať pozor. Zároveň narážajú na dva veľké problémy. Prvým problémom je, ako udržiavať program v konzistentnom a funkčnom stave, no pritom umožniť prácu na programe viacerým programátorom naraz. Druhým problémom je uchovávanie histórie. Potrebovali by sme teda, aby sa dalo prezerať všetky zmeny počas celého vývoja projektu. Preto bol vyvinutý verziovací systém, ktorý tieto dva problémy dokáže efektívne riešiť.

3.2 Repozitár

Základnou časťou celého verziovacieho systému je repozitár. Je vytvorený rovnako ako štandardný súborový systém so stromovou štruktúrou. Navyše však zaznamenáva všetky vykonané zmeny v priebehu času. Jeho verejným umiestnením získavame, že ktokoľvek z programátorov sa môže k nemu pripájať a v uložených súboroch čítať a zapisovať. Vykonávané zmeny sú dostupné pre všetkých.

Používaním verziovacích nástrojov získavame veľkú výhodu oproti bežnému zdieľanému priečinku v sieti. Vďaka konceptu repozitára sa tu dá ľahko zistiť, ako presne vyzeral daný súbor pred nejakým konkrétnym časom. Môžeme sa tiež pozrieť, ktorý programátor vykonal aké zmeny (obr. 3.1).

Na súboroch funguje princíp pracovnej kópie. Teda ktokoľvek, kto chce súbory čítať alebo do nich zapisovať, si vytvorí svoju lokálnu pracovnú kópiu a na nej môže pracovať bez obmedzení. Po skončení svojej práce súbor nahrá naspäť do repozitára [2].

Obr. 3.1: História repozitára

https://fmfiuk-biotools.googlecode.com/svn/fmfiuk.biotools.core

Revision	Date	Changes	Author	Comment
***99**	**10.1.2013 11:05**	**11**	**filip**	**detection of correct parser**
95	31.12.2012 11:51	3	filip	Predicate<File> first experiment implementation
93	6.12.2012 10:26	33	filip	CorrectLineParser created and implemented
92	3.12.2012 16:18	2	jaro	name modified
91	3.12.2012 16:12	15	filip	name modified
89	29.11.2012 14:46	1	jaro	add contructor NucleotideSequence(String)
85	28.11.2012 14:17	2	filip	type of param in getCombinations changed...
84	27.11.2012 23:52	2	filip	AmbiguousNucleotidr - getCombination

3.3 Modely verziovania - zdieľanie súborov

Dôležitou úlohou verziovacích systémov je nielen uchovávanie všetkých zmien v súboroch, ale je to aj možnosť editovať a zdieľať súbory medzi viacerými programátormi naraz. Problémom však stále zostáva, ako dosiahnuť, aby viac ľudí nezasahovalo na to isté miesto v súbore naraz.

Kým sme v situácii, že do súborov nikto nechce zapisovať, nenastávajú žiadne problémy. Z repozitára si vezmeme najnovšiu verziu, spravíme potrebné úpravy a pozmenený súbor nahráme späť. Ak sa nájde niekto ďalší, kto by chcel čítať súbor, uvidí už aktuálnu verziu. Ak sú však viacerí ľudia, ktorí chcú editovať rovnaký súbor naraz, môže nastať konflikt.

Ak by boli dvaja, čo idú pracovať naraz v jednom súbore, spravia si kópiu súboru podľa najnovšej verzie a začnú na nej pracovať. Akonáhle prvý z nich dokončí prácu, nahrá súbor do repozitára a ostatní už vidia jeho upravenú verziu. Keď skončí svoju prácu druhý z nich, tiež nahrá svoj súbor. Tým ale odstráni zmeny vykonané prvým programátorom, keďže obaja vychádzali z rovnakého pôvodného súboru. Pritom možno vôbec nemenili rovnaké časti súborov. Keď teda obaja skončia svoju prácu, očakávame, že v najnovšej verzii repozitára bude súbor s úpravami od oboch používateľov.

V súčasnosti poznáme dve základné riešenia problému zdieľania súborov. Keďže tieto dva prístupy sú odlišné, je dôležité rozumieť obom a vybrať si jeden z nich podľa aktuálnych potrieb daného projektu.

3.3.1 Zámky

Jedným zo základných riešení, ako predísť problémom pri zdieľaní súborov, je používanie zámkov. Repozitár dovoľuje upravovať súbor iba jednému programátorovi.

Predtým, ako začne prvý z nich pracovať, zamkne súbor. Potom už môže zapisovať, čo potrebuje. Po skončení práce súbor nahrá do repozitára a odomkne ho. Keď je súbor odomknutý, iný programátor si ho môže zamknúť a robiť svoje úpravy. Riešenie zdieľania súborov pomocou zámkov je častokrát dosť obmedzujúce a prináša viacero nevýhod:

1. Zamykanie môže spôsobovať zbytočný problém čakania. Navyše sa ľahko môže stať, že niektorý programátor zabudne súbor odomknúť.

2. Riziko vzájomného uviaznutia. Ak prvý programátor zamkne súbor A, druhý zamkne súbor B a potom obaja potrebujú zamknúť ešte ten opačný súbor, nastáva uviaznutie a musia sa dohodnúť, ktorý z nich ustúpi zo svojich požiadaviek.

3. Bezdôvodná serializácia vzniká za predpokladu, že každý z programátorov by chcel upravovať inú časť z jedného súboru. Jeden z nich teda musí čakať na uvoľnenie zámku, aj keď by si v práci nezavadzali.

4. Zámky iba vytvárajú dojem bezpečnosti. Keď programátori začnú upravovať dva na sebe závislé súbory, po ich odomknutí môžu byť v nekonzistentnom alebo aj nekompilovateľnom stave.

3.3.2 Zlučovanie

Iný model verziovania je založený na paralelných úpravách a následnom zlučovaní modifikovaných súborov. Každý z programátorov pracuje na svojej lokálnej kópii. Pri nahrávaní súboru na repozitár musí v prípade konfliktu svoj súbor zlúčiť s aktuálnou verziou. Konflikt nastane, ak má vo svojom súbore také úpravy, ktoré sa niektorou časťou prekrývajú s aktuálnou verziou a túto verziu v repozitári vytvoril niekto iný z používateľov. Pri zlučovaní musí manuálne prejsť cez všetky rozdiely a rozhodnúť, ktorú časť chce zachovať. Je však potrebné dbať na niekoľko dôležitých vecí:

1. Ak dvaja programátori po sebe upravovali rovnaké časti súborov, druhý z nich sa pri zlučovaní dostáva do konfliktu. Sám si potom musí vybrať, ktorá zo zmien bude zapísaná vo verzii repozitára.

2. Samotné zlučovanie vyžaduje prácu navyše. Verziovacie nástroje často umožňujú automatické zlučovanie súborov, ale napriek tomu je dobré prejsť kritické miesta manuálne.

3. Pri zlučovacej metóde je veľmi dôležitá komunikácia. Bez nej by zlučovanie konfliktných častí súborov bolo veľmi ťažké.

3.4 Subversive SVN

Na prácu v našom projekte sme si vybrali systém Subversive SVN. Jednou z výhod bolo pre nás to, že je priamo podporovaný nami používaným vývojovým prostredím Eclipse. Zároveň je pomerne jednoduchým a vhodným nástrojom na to, aby sme pochopili verziovací systém.

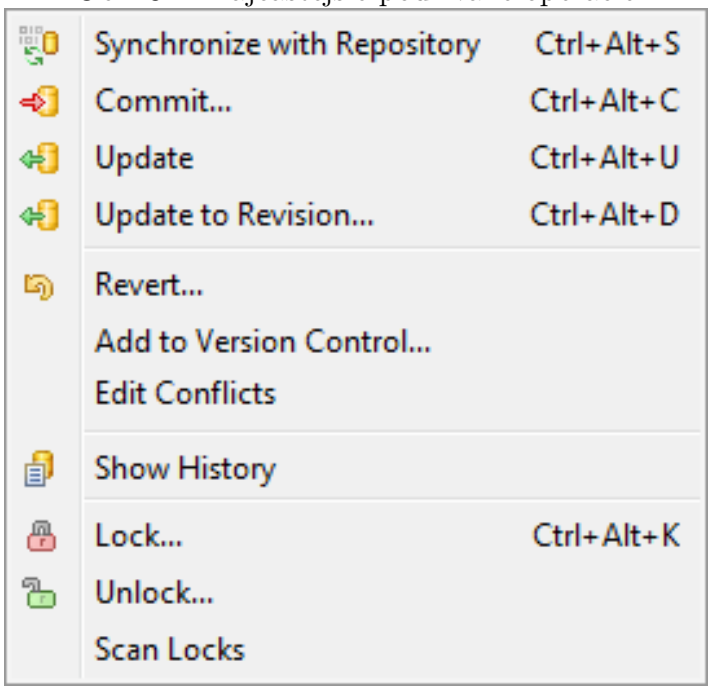
Tento systém ponúka na výber obe možnosti verziovania súborov. Model zámkov sa používa častejšie, pokiaľ v tíme pracuje málo ľudí alebo sú vykonávané zmeny rýchlo a na malých úsekoch. Pre naše potreby by teda stačilo používanie zámkov. Napriek tomu sme používali metódu zlučovanie súborov. Vyžadovalo to niekedy viac práce, no na druhej strane sme si mohli vyskúšať, ako reálne funguje zlučovanie.

Verziovací systém sme využívali počas každej práce na projekte, no dokázali sme využiť len niekoľko základných operácií (obr. 3.2):

1. Add to Version Control: Po vytvorení nového súboru sme ho takto nahrali na repozitár.
2. Update: Vždy pred začatím práce na projekte sme vykonali update a tak sme videli, či niekto iný vykonal zmeny. Získali sme tak najnovšiu verziu projektu.
3. Show History: Pre ľahšie zorientovanie sa vo vykonaných zmenách sme si prezreli históriu, kde boli všetky zmeny popísané v komentároch (obr. 3.1).
4. Edit Conflicts: Používaním zlučovacieho riešenia problému zdieľania súborov sme občas narazili na rozdiely, ktoré sme museli manuálne zlúčiť. Sami sme teda rozhodli, ktorú z verzií (lokálnu alebo v repozitári) chceme uložiť ako najnovšiu.
5. Commit: Takto sme potvrdili naše zmeny a nahrali ich spolu s pridaným komentárom zmien do repozitára.

Celkovo sa so systémom SVN pracovalo veľmi dobre. Keď sme dodržiavali štandardný postup a pridávali sme aj komentáre pri nahrávaní do repozitára, dalo sa ľahko orientovať v tom, čo bolo kedy pridané a zmenené.

Obr. 3.2: Najčastejšie používané operácie



Kapitola 4

Maven

Maven je nástroj na vykonávanie rutinných programátorských činností ako napríklad kompilácia, testovanie a generovanie zbalených zdrojových kódov do .jar súboru. "Umožňuje navyše pokročilú správu projektov, čo zahŕňa vytvorenie objektového modelu projektu, súbor noriem, životný cyklus projektu, správu závislostí a logiku pre vykonávanie cieľov definovaných v jednotlivých fázach vývojového cyklu."[3]

4.1 Výstavba projektu

Celý projekt je definovaný pomocou projektového objektového modelu (POM). V hlavnom súbore pom.xml (dodatok A) sú popísané všetky dôležité informácie o projekte a aj konfigurácii pluginov. Je tu tiež uvedená cesta ku zdrojovým súborom, testom a cieľovým adresárom [4]. Tento súbor je vytvorený v jazyku XML.

Najpodstatnejšou časťou objektového modelu sú informácie o samotnom projekte. Túto časť (obr. 4.1) musí obsahovať každý POM.

1. modelVersion: Verzia objektového modelu, ktorý používa POM.
2. name: Meno projektu, ktoré sa používa v dokumentácii a v iných projektoch, ktoré ho používajú.
3. groupId: Unikátny identifikátor organizácie alebo skupiny, ktorá projekt vytvorila.
4. artifactId: Unikátny názov projektu. Názov finálneho zbaleného projektu sa po skončení štandardne vytvára ako [artifactId]-[version].jar
5. version: Aktuálna verzia projektu.

Obr. 4.1: Povinná časť nášeho objektového modelu

```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
     http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <name>RNAViewer</name>

    <groupId>fmfiuk.biotools</groupId>
    <artifactId>RNAViewer</artifactId>
    <version>0.0.1-SNAPSHOT</version>
      ⋮
</project>
```

Týmto spôsobom je zabezpečená unikátnosť jednotlivých projektov vytvorených cez Maven. Možno ich teda podľa týchto údajov neskôr medzi sebou spájať - vytvárať závislosti 4.2.2.

4.2 Využitie v praxi

Maven framework sme použili preto, lebo v našom projekte by sme robili viacero štandardných činností opakovane. Toto bola možnosť, ako si prácu zľahčiť. Jeho integrácia do projektu bola veľmi intuitívna a podarila sa nám bez väčších komplikácií. Vďaka nemu sme boli odbremenení od formálnych záležitostí a tak sme mohli venovať viac času vývoju a implementácii. Využívali sme ho teda priebežne pri viacerých operáciách.

4.2.1 Archetypy

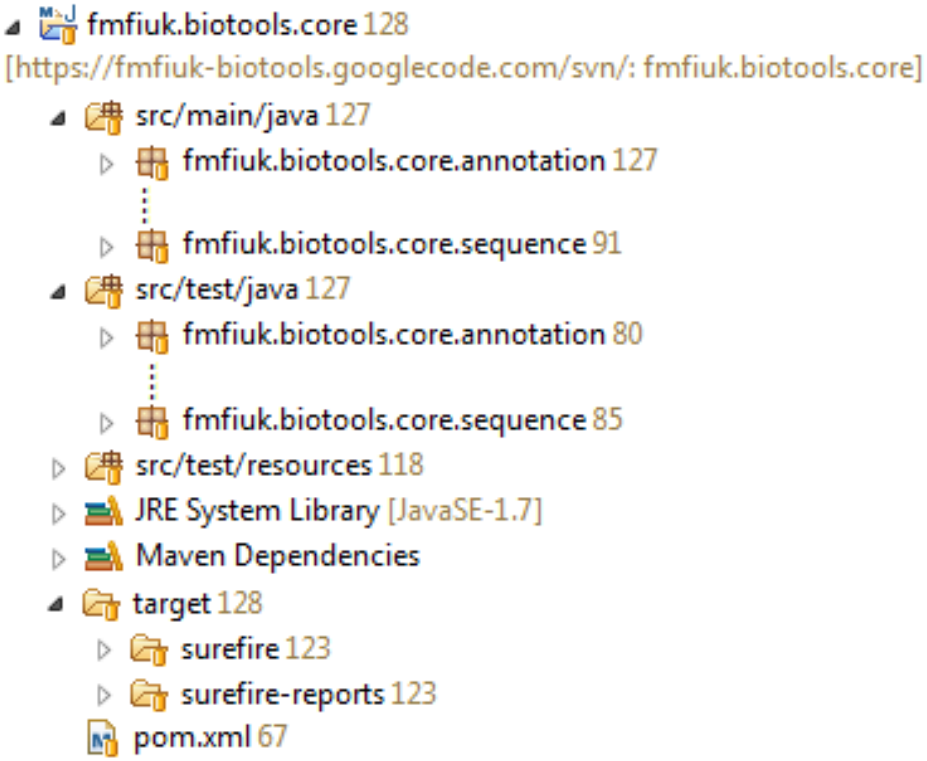
Jednou zo základných vecí pri tvorbe väčších projektov je voľba vhodnej organizácie zdrojových súborov v adresároch. Pri tejto voľbe Maven ponúka viacero rôznych riešení - archetypov. Archetyp je šablóna, podľa ktorej dokáže Maven generovať základnú projektovú štruktúru. Môže byť využitý aj na dodržiavanie rovnakých štandardov počas práce na viacerých projektoch. Je tiež možné vytvoriť si vlastný návrh štruktúry a následne ho používať na všetky projekty za pomoci Mavenu.

Rôzne archetypy generujú odlišnú štruktúru. Napríklad maven-archetype-site vytvorí pripravenú štruktúru ako webovú stránku. Zahrnuté sú súbory na lokalizáciu textov v anglickom a francúzskom jazyku, hlavná stránka (index) a tiež stránka s častými

otázkami (FAQ). Preto treba vedieť, akú formu projektu pripravujeme, aby sme vedeli vybrať, ktorý archetyp použiť.

Na vytvorenie našej štruktúry pre RNAViewer sme využili základný archetyp maven-archetype-quickstart. Je to veľmi jednoduchý template, ktorý automaticky rozlišuje adresáre na hlavné a testovacie. Súbory sa potom ukladajú do adresárov src/main/java a src/test/java. V nich sme si potom sami priebežne vytvárali balíčky podľa toho, čo sme aktuálne potrebovali 4.2. Maven nám taktiež pripojil JUnit framework na testovanie. Okrem toho sa samozrejme generuje základný konfiguračný súbor pom.xml pre spravovanie celého projektu počas vývoja. Ten je vytváraný každým archetypom.

Obr. 4.2: Vygenerovaná štruktúra podľa maven-archetype-quickstart



4.2.2 Závislosti

V projektoch často používame aj iné frameworky alebo triedy, na ktoré sa v našich triedach odkazujeme. Tieto triedy je zväčša potrebné niekde získať (stiahnuť) a pridať ich do knižnice na kompilovanie. Výhodou je, že my ako vývojári sa nemusíme zaťažovať s ich sťahovaním, ale zabezpečí to Maven. Potrebujeme iba definovať požadovanú závislosť v pom.xml súbore. Definujeme ich pomocou modelVersion, artifactId a groupId, čo slúži ako jedinečné označenie pre ktorýkoľvek iný framework alebo triedu.

Pre rýchlejšie spracovanie si Maven udržuje lokálny repozitár (home/.m2/repository), kde si priebežne sťahuje všetky definované vzťahy. Počas kompilácie sa skontrolujú definované závislosti v lokálnom priečinku a ak sa tam nenachádzajú, Maven sa pokúsi stiahnuť ich z verejne dostupných repozitárov na internete (napr. `www.mvnrepository.com`). Vo verejných repozitároch sa dajú nájsť všetky dostupné knižnice a frameworky vytvorené pre jazyk Java. Slúžia ako verejné úložisko, kde ich ktokoľvek môže nájsť a stiahnuť na použitie pre vlastné potreby.

Maven zo závislostí vytvára hierarchie. Väčšina použitých knižníc má tiež svoje závislosti. Tie ale v konfiguračnom súbore nemusíme definovať. Vďaka Mavenu sa sťahujú automaticky (obr. 4.3). Jedným zápisom do XML dostaneme všetky definované triedy priamo pripravené na použitie, a tiež výhodu, že hneď, ako bude vytvorená nová verzia niektorej z definovaných závislostí, automaticky sa stiahnu aj všetky ostatné.

Obr. 4.3: Porovnanie definovaných a automaticky pridaných závislostí

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>3.2.2.RELEASE</version>
  </dependency>
  <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>3.1.2.RELEASE</version>
  </dependency>
</dependencies>
```

```
junit : 4.8.2 [test]
spring-core : 3.2.2.RELEASE [compile]
  commons-logging : 1.1.1 [compile]
spring-context : 3.1.2.RELEASE [compile]
  spring-aop : 3.1.2.RELEASE [compile]
  spring-beans : 3.1.2.RELEASE [compile]
    spring-core : 3.1.2.RELEASE
    (omitted for conflict with 3.2.2.RELEASE) [compile]
  spring-core : 3.1.2.RELEASE
  (omitted for conflict with 3.2.2.RELEASE) [compile]
  spring-expression : 3.1.2.RELEASE [compile]
    spring-core : 3.1.2.RELEASE
    (omitted for conflict with 3.2.2.RELEASE) [compile]
  spring-asm : 3.1.2.RELEASE [compile]
```

V našom projekte sme využili napríklad JUnit framework, pomocou ktorého budeme testovať naše triedy. Pridali sme aj spring-context a spring-core a Maven sám pridal ďalšie potrebné knižnice (spring-aop, spring-beans, spring-expression,...).

4.2.3 Projektový cyklus

Celý vývoj projektu môže opakovane prechádzať viacerými fázami. Medzi tie základné patrí:

1. validate: Kontrola konfigurácie a korektnosti obsahu POM.
2. compile: Kompilácia zdrojových kódov projektu.

3. test: Vykonanie všetkých unit testov.

4. package: Zbalenie zdrojových kompilovaných súborov do archívu pripraveného na použitie ([artifactId]-[version].jar).

5. integration-test: Proces nasadenia archívu do testovacieho prostredia.

6. verify: Overenie správnosti potrebných údajov na používanie v iných projektoch pomocou Mavenu.

7. install: Inštalácia do lokálneho repozitára na ďalšie použitie.

8. deploy: Odovzdanie výsledného projektu do verejného Maven repozitára na používanie pre iných vývojárov.

Tieto kroky sú štandardne vykonávané v uvedenom poradí. Preto ak vykonáme fázu package a nedefinujeme výnimku, vykonajú sa aj všetky fázy pred ňou - teda validate, compile a test.

Počas prác na našom projekte sme veľmi často používali spoločné testovanie všetkých unit testov (obr. 4.4). Takýmto spoločným testovaním sme si boli istí, že všetky naše triedy pracujú správne aj po jednoduchých úpravách.

Obr. 4.4: Vykonanie spoločných testov cez Maven

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running fmfiuk.biotools.core.annotation.AnnotationSetTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.062 sec
Running fmfiuk.biotools.core.annotation.AnnotationTest
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running fmfiuk.biotools.core.annotation.RangeTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.015 sec
    ⋮
    ⋮
Running fmfiuk.biotools.core.sequence.SingleNucleotideTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec

Results :

Tests run: 50, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.295s
[INFO] Finished at: Sat Mar 23 21:15:28 CET 2013
[INFO] Final Memory: 9M/152M
[INFO] ------------------------------------------------------------------------
```

Po ukončení prác na jadre programu sme využili fázu install. Jej vykonaním sme získali v lokálnom repozitári celé jadro pripravené na použitie (napríklad pridaním ako závislosti) v iných častiach systému. Takto vytvorené a otestované jadro sme sprístupnili pre ďalších používateľov (`http://code.google.com/p/fmfiuk-biotools/downloads`).

Kapitola 5

Spring Framework

"Spring je aplikačný framework s otvoreným zdrojovým kódom, ktorý je určený na podporu pri riešeniach z mnohých oblastí, ako napríklad: web, databázový prístup, cloud, bezpečnosť aplikácií či vývoj na mobliné platformy."[5] Navyše umožňuje pokročilé návrhové prístupy ako aspektovo orientované programovanie (aspect oriented programming, AOP) a pridávanie závislostí (dependency injection, DI).

Keďže aplikácie vytvorené za pomoci Springu je možné spúšťať všade, kde aj obyčajné Java programy (na rôznych platformách), zabezpečená je ľahká prenositeľnosť. Zároveň sa využívaním množstva priložených štandardných tried zvyšuje produktivita, lebo je možné venovať viac času na vývoj tried špecifických pre konkrétnu aplikáciu.

Z množstva ponúkaných súčastí Springu sme využili pridávanie závislostí a aspektovo orientované programovanie. Okrem toho sme zistili, ako funguje v Springu mapovanie objektov na XML a naopak.

5.1 Aplikačný kontext

V aplikáciách založených na Springu sú objekty (beany) zahrnuté v Spring kontajneri (dodatok B). Tento kontajner objekty vytvára, spája dokopy, konfiguruje a spravuje počas celého ich životného cyklu od vytvorenia až po vymazanie z pamäti (obr. 5.1). Na to, aby objekty mohol prepájať a riadiť, sa používa koncept pridávania závislostí.

Aplikačný kontext je Spring kontajner definovaný pomocou ApplicationContext interface. Ako jeho implementáciu sme využili FileSystemXmlApplicationContext. V aplikačnom kontexte definujeme použité objekty ako beans. Takto definované beany potom v programe získavame pomocou metódy getBean(String name) (obr. 5.2). De-

Obr. 5.1: Konfiguračný súbor aplikačného kontextu a beanov

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="bedLineParser"
     class="fmfiuk.biotools.core.parser.BedLineParser">
        <constructor-arg value="\t" />
    </bean>
      ⋮
</beans>
```

finovali sme napríklad objekt triedy BedLineParser (obr. 5.1), ktorý je singletonom a môžeme ho vytiahnuť kdekoľvek v programe.

Obr. 5.2: Získanie referencie na objekt zo Spring kontajnera

```
public static void main(String[] args) {
    ApplicationContext context = new FileSystemXmlApplicationContext(
            "src/main/config/spring/applicationContext.xml");

    BedLineParser bedLineParser = (BedLineParser) context
            .getBean("bedLineParser");
    bedLineParser.getLineSeparator();
}
```

5.1.1 Bean

"Java bean sú také triedy, ktoré majú svoje osobitné pomenovanie a k ich inštanciám pristupujeme pomocou týchto referencií."[5] Aby bolo možné použiť v Springu niektorú triedu ako bean, musí spĺňať niekoľko podmienok. Daný objekt musí obsahovať defaultný public konštruktor (bez argumentov) a premenné, ktoré chceme používať, musia byť prístupné pomocou get a set metód. Výhody použitia beans:

1. Vzťahy medzi objektami definujeme na jednom mieste.
2. Jednu inštanciu je možné použiť na viacerých miestach.

Používanie beanov prináša aj nevýhody:

1. Pre nutnosť používania množstva getterov a setterov sa stávajú triedy menej prehľadnými.
2. Použitím konštruktora bez argumentov často vznikajú nepoužiteľné triedy, v ktorých je nutné následne nastaviť ďalšie premenné.

5.2 Pridávanie závislostí

Pridávanie závislostí je pomerne novou programátorskou technikou a stalo sa jednou zo základných častí Springu.

Štandardný návrh aplikácie obsahujúcej viacero tried musel mať tieto triedy navzájom prepojené. Každý z objektov teda musel mať referencie na ostatné objekty, s ktorými potreboval spolupracovať. Navyše bolo potrebné poznať konkrétnu implementáciu referencovaného objektu. Takto bola inicializácia objektov zahrnutá v rôznych triedach.

Takýto prístup viedol k priveľmi previazanému programu. Bolo pomerne zložité takéto triedy využiť na niečo iné a tiež vytvoriť jednotné testy. Lepším spôsobom je previazanie tried pomocou interface, čo umožňuje jednoduché zmeny samotnej implementácie.

Pri používaní DI sú vzťahy medzi objektami definované deklaratívne v XML súbore, čo dovoľuje zmenu ich implementácie bez prekompilovania kódu. Bežne sú používané dva spôsoby pridávania závislostí - pomocou konštruktorov alebo setterov.

5.2.1 Konštruktor

Použitie konštruktora je štandardným spôsobom, ako vytvoriť v Spring kontajneri bean. Je možné využiť defaultný konštruktor, no dá sa používať aj taký, ktorý obsahuje argumenty. Na ich pridávanie sa v konfiguračnom súbore pri definícii objektu pridá element <constructor-arg> (obr. 5.3). Na definovanie hodnôt sa používa atribút value="". Pomocou neho môžeme zadať konkrétnu hodnotu. Ak ale chceme použiť ako argument iný objekt, používa sa atribút ref="" a v ňom id daného beanu. Keď tento element nie je použitý, berie sa základný konštruktor bez argumentov.

5.2.2 Set metódy

Vlastnosti beanov sú prístupné pomocou get a set metód, pričom v springu sa dajú konfigurovať pomocou elementu <property>. Tento element je funkcionalitou a používaním veľmi podobný konštruktorom, avšak jeho použitím sa závislosti pridávajú cez set metódy (obr. 5.4). Konkrétna metóda sa definuje v atribúte name="" a jej hodnota pomocou value="".

Obr. 5.3: Použitie konštruktora pri definovaní beanu

```
<bean id="bedLineParser"
 class="fmfiuk.biotools.core.parser.BedLineParser">
    <constructor-arg value="\t" />
</bean>

<bean id="gffLineParser"
 class="fmfiuk.biotools.core.parser.GffLineParser">
    <constructor-arg value="\t" />
</bean>

<bean id="gtfLineParser"
 class="fmfiuk.biotools.core.parser.GtfLineParser">
    <constructor-arg value="\t" />
</bean>
```

Obr. 5.4: Vytvorenie objektu použitím konštruktor a property

```
<bean id="bedCsvAnnotationFileParser"
 class="fmfiuk.biotools.core.common.CsvAnnotationFileParser">
    <constructor-arg ref="bedLineParser" />
    <property name="canParsePredicate" ref="correctFilePredicateBed" />
</bean>
```

Častokrát nám viac záleží na vzájomnom prepojení tried medzi sebou. Vtedy využívame atribút ref="", rovnakým spôsobom ako pri konštruktoroch. Práve v tomto sa však prejavuje výhoda používania Springu a jeho beanov. Definované závislosti môžeme ľahko zmeniť a definovať nové. Toto všetko sa dá urobiť robiť pri každom spúšťaní aplikácie nanovo, bez toho, aby bola nutná opätovná kompilácia.

Spring takisto podporuje pridávanie viacerých beans do jednej vlastnosti (ak je typu List, Collection, Map, ...). Závislosti sa dajú pridávať cez elementy:

1. <list>: Závislosti sú pridané ako java.util.List a povolené sú aj rovnaké hodnoty.
2. <set>: Pridáva sa java.util.Set a zaručené je, že duplikáty sa nedajú pridať.
3. <map>: java.util.Collection pridáva páry názov-hodnota, pričom môžu byť ľubovoľného typu.
4. <props>: java.util.Collection sa používa pri dvojiciach stringov ako názov-hodnota.

Tieto rôzne spôsoby DI je možné kombinovať (obr. 5.5). Do konštruktora beanu AnnotationService sme vložili dva parsery na rôzne druhy vstupných súborov (.gff, .bed). Táto trieda má za úlohu vybrať podľa vstupného súboru anotácií taký parser, ktorý vie

Obr. 5.5: List iných beanov použitých v konštruktore

```
<bean id="annotationService"
 class="fmfiuk.biotools.services.structure.AnnotationService">
    <constructor-arg>
        <list>
            <ref bean="gffCsvAnnotationFileParser" />
            <ref bean="bedCsvAnnotationFileParser" />
        </list>
    </constructor-arg>
</bean>
```

daný typ súborov spracovať. Po nájdení správneho z nich, bude daný parser pokračovať vo vykonávaní.

Pri viacerých častiach projektu sa nám oplatilo používať prístup pridávania závislostí. Príkladom sú tiež parsery na rôzne formáty vstupných súborov s RNA anotáciami (obr. 5.3). Týmto prístupom sme získali veľkú flexibilitu pre nás aj iných používateľov. Nám sa vďaka jednej zmene beanu ľahko menia použité parsery. Okrem toho prístupom Java kontajnera šetríme miestom v pamäti, keďže definované beans sa dajú použiť aj na inom mieste, a preto netreba vytvárať viacero inštancií rovnakého objektu.

Pridanie nového parsera je jednoduchšie a navyše pritom nie je potrebný zásah do existujúceho kódu. Stačí naprogramovať nový parser (tak aby dedil z abstraktnej triedy CsvAnnotationLineParser) podľa iného typu vstupného súboru a definovať ho v konfiguračnom súbore. V našom kóde by takto nebolo potrebné nič iné meniť a ani ju kompilovať.

5.3 Aspektovo orientované programovanie

"Aspektovo orientované programovanie je definované ako technika, ktorá pomáha rozdeliť úlohy v rámci daného softvérového systému."[5] Každý dobre navrhnutý softvérový produkt pozostáva z viacerých komponentov, pričom každý z nich má inú funkcionalitu. Problémom však je, že moduly často plnia aj iné úlohy oproti ich základnému poslaniu. Býva v nich zahrnuté napríklad logovanie, správa transakcií či bezpečnosť.

Týmto prístupom sa do našich komponentov pridáva zložitosť na viacerých úrovniach.

1. Pridaný kód v triedach je častokrát rovnaký na viacerých miestach. To znamená, že keď je treba niektorú časť pozmeniť, musíme prezrieť všetko, kde sa daná časť

používala. A aj keby sme mali definované iba jedno volanie špeciálnej funkcie, musela by sa nachádzať na viacerých miestach, čo by bolo len zbytočné duplikovanie.

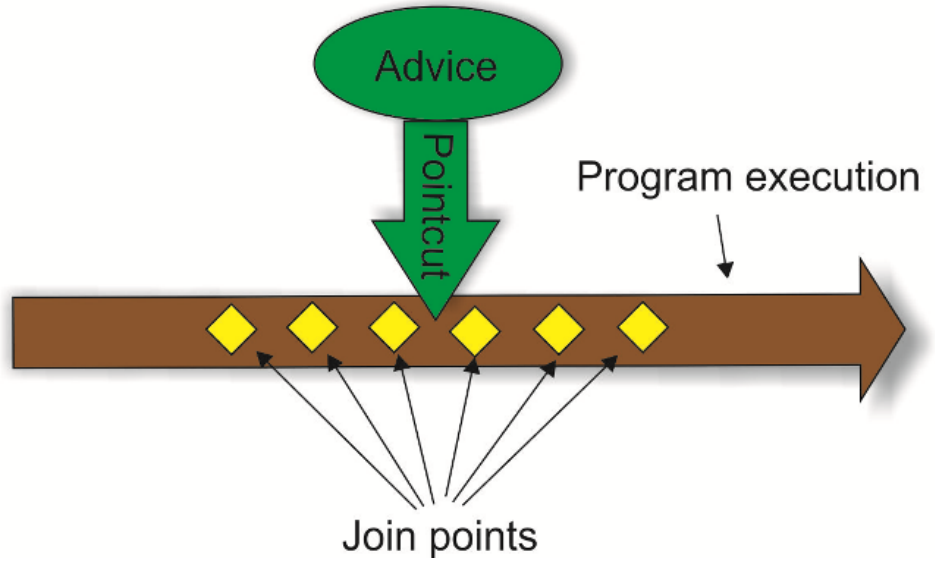
2. V našich komponentoch by sa takýmto spôsobom práce nachádzalo príliš veľa kódu, ktorý vôbec nezodpovedá ich pôvodnému určeniu.

Vďaka AOP sa dajú vytvoriť rôzne ďalšie vrstvy aplikácie (logovacia, bezpečnostná), ktoré deklaratívne pripájame na komponenty z jadra bez toho, aby to priamo ovplyvnilo ich činnosť. Vďaka tomu sú všetky tieto komponenty súdržné a zároveň sa zameriavajú len na svoju úlohu.

5.3.1 Princíp

Princíp fungovania aspektu je pomerne jednoduchý (obr. 5.6). V programe definujeme miesta, na ktoré sa aspekt viaže. Potom sa na týchto miestach počas behu programu vykonávanie preruší a vloží sa funkcionalita aspektu. Pridávanie rôznych vrstiev

Obr. 5.6: Vykonávanie funkcie aspektu v programe



pomocou aspektov prináša čiastočne nový pohľad na spoluprácu objektov. Používaním AOP sa zavádza viacero nových pojmov:

1. aspect: Je to modul, ktorý obsahuje metódy na spracovávanie požiadaviek z iných častí aplikácie.

2. advice: Práca, ktorú vykonáva aspekt, sa nazýva Advice. Konkrétnejšie definuje, čo a kedy sa vykonáva.

3. join point: Je to bod vo vykonávaní aplikácie, kde môže byť zapojený aspekt. Týmto bodom môže byť metóda, ktorá bude zavolaná, alebo aj výnimka, čo môže byť vyhodená, či iná udalosť. Je to miesto, kde môžeme vložiť Java kód aspektu do štandardného behu metód z komponentov, a tak pridať novú funkcionalitu. V Springu rozlišujeme päť možných časov, kedy aspekt dokáže niečo vykonať:
   (a) before: Časť vykonaná aspektom sa robí pred danými metódami.
   (b) after: Časť vykonaná aspektom sa robí po tom, ako skončia dané metódy.
   (c) after-returning: Ak dostaneme očakávaný výsledok z metódy, spustí sa vykonávanie aspektu.
   (d) after-throwing: Aspekt je použitý po vyhodení výnimky.
   (e) around: Metódy sú obklopené funkciou aspektu - teda pred aj po jej vykonaní.
4. pointcut: Definuje, kde sa spúšťa aspekt. Týmto spôsobom popíšeme, pri ktorých balíčkoch, triedach alebo metódach z komponentov aplikácie sa vykonávajú metódy nášho aspektu.

5.3.2 Deklarácia aspektov

Použité aspekty deklarujeme v konfiguračnom súbore, najčastejšie na rovnakom mieste ako sú definované aj beans alebo iné spring komponenty. Všetky súčasti aspektu definujeme pomocou XML tagov.

1. <aop:config>
2. <aop:aspect>
3. <aop:advisor>
4. <aop:pointcut>

Pre advice sa používajú elementy ako <aop:before>, <aop:after-returning>,... Obsahujú atribút method="", ktorý obsahuje meno vykonávanej metódy a atribút pointcut-ref="". Ten má odkaz na zoznam tried a metód, pri ktorých sa bude metóda aspektu spúšťať.

Obr. 5.7: Logger s výpisom do konzoly

```
public class ConsoleLogger implements MethodBeforeAdvice, AfterReturningAdvice,
        ThrowsAdvice {

    public void before(Method method, Object[] args, Object target)
            throws Throwable {
        System.out.println("Start " + target.getClass().getName() + "."
                + method.getName());
    }

    public void afterReturning(Object returnValue, Method method,
            Object[] args, Object target) throws Throwable {
        System.out.println("Finished " + target.getClass().getName() + "."
                + method.getName());
    }

    public void afterThrowing(Method method, Object[] args, Object target,
            Exception exception) {
        System.out.println(exception.getLocalizedMessage());
    }
}
```

5.3.3 Logger

V aplikácii sme vytvorili jednoduchý logger (5.7) s metódami before a after, ktoré vypisujú na konzolu "start" a "end" a príslušný názov spustenej metódy. Tento konzolový logger sme pridali ako bean v konfiguračnom súbore (obr. 5.8). V XML deklarácii

Obr. 5.8: AOP deklarácia loggera

```
<bean id="logger"
  class="fmfiuk.biotools.services.logging.ConsoleLogger" />

<aop:config>
    <aop:pointcut id="selectAll"
     expression="execution(* fmfiuk.biotools.core..*.*(..))" />
    <aop:advisor advice-ref="logger" pointcut-ref="selectAll" />
</aop:config>
```

na pripojenie loggera ako aspektu sme využili skrátený zápis. Celý aspekt definujeme v elemente <aop:config>. V ňom sú popísané všetky ostatné časti potrebné na to, aby logger pracoval správne. V atribúte expression="" elementu <aop:pointcut> deklarujeme, pri ktorých volaniach metód z balíkov sa vykonávajú funkcie aspektu. Druhý element <aop:advisor> hovorí o tom, ktorá trieda (bean) je použitá ako advice. V našom prípade to je definovaný ConsoleLogger.

5.4 Objekt/XML mapovanie

Spring poskytuje veľa tried na často sa opakujúce úlohy. V našej aplikácii sme chceli urýchliť načítavanie vstupných súborov. Využili sme preto Spring konvertovanie XML dokumentov na objekty a naopak. Túto činnosť prevodu na XML označujeme ako XML Marshalling alebo XML Serializácia. Spätný proces sa nazýva XML Unmarshalling.

Použitie marshallingu nám pomohlo vyhnúť sa viacnásobnému načítavaniu tých istých vstupných súborov. Keďže sú naše súbory veľmi veľké, ich prvotné spracovanie môže trvať pomerne dlho. Po ich prvom načítaní máme informácie o súbore uložené v objekte. Pri ukončení aplikácie si tieto informácie uložíme pomocou marshallera do XML súborov, odkiaľ ich pri nasledujúcom používaní programu ľahko načítame opäť do pamäte.

Štandardné interfacy pre triedy umožňujúce XML serializáciu v Spring frameworku sú org.springframework.oxm.Marshaller a org.springframework.oxm.Unmarshaller. V nich sú zahrnuté metódy marshal a unmarshal. Táto abstrakcia umožňuje jednoduchú zámenu konkrétnych tried určených na Marshalling.

Konfigurácia je vytváraná klasickým použitím Java beans v aplikačnom kontexte aplikácie. Pri používaní ktorýchkoľvek mapovacích metód je navyše pridaná XmlMappingException výnimka, takže všetky chybové správy sú zachytávané pomocou nej.

Interface Marshaller má metódu marshal(), ktorá konvertuje každý java.lang.Object na javax.xml.transform.Result. Konkrétne použité implementácie Result interface-u sú zväčša realizované formou XML výstupu (obr. 5.9).

Obr. 5.9: Objekt prevedený do súboru XML

```
<?xml version="1.0" encoding="UTF-8"?>
<array-list>
    <sequence-file-index xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:type="java:fmfiuk.biotools.core.parser.sequence.SequenceFileIndex">
        <sequence-index-map xsi:type="java:org.exolab.castor.mapping.MapItem">
            <key xsi:type="java:java.lang.String">&gt;Bm_scaf2</key>
            <value xsi:type="java:fmfiuk.biotools.core.parser.sequence.FixedLineSequenceIndex">
                <line-length>4</line-length>
                <line-separator-length>2</line-separator-length>
                <sequence-length>8</sequence-length>
            </value>
        </sequence-index-map>
        <position-period>1000</position-period>
        <sequence-file>../fmfiuk.biotools.core/src/test/sequence/silkworm.fa</sequence-file>
    </sequence-file-index>
</array-list>
```

V poli ArrayList<SequenceFileIndex> sme mali uložené informácie o načítanom vstupnom súbore a na tento objekt sme použili Marshaller. Trieda SequenceFileIndex nesie informácie o indexe vytvorenom podľa vstupného súboru a má ich uložené v premenných String sequenceFile, BufferedCharReader reader, Map<String, SequenceIndex> sequenceIndexMap a Integer positionPeriod. Všetky tieto údaje zostali zachované aj vo výstupe.

V zložitejších prípadoch je možné vytvoriť konfiguračný súbor, ktorým sa definujú vzťahy medzi vstupným objektom a výsledným XML súborom. Teda to, ktoré vlastnosti objektu budú mapované na aké elementy, prípadne atribúty. Takýto mapovací súbor dostane ako vlastnosť daný marshaller.

Metóda unmarshal() má opačnú funkcionalitu. Funkcia ako vstup (zvyčajne formou XML) dostane javax.xml.transform.Source a načíta z neho informácie do pamäte, do nejakého objektu.

5.4.1 Implementácie Marshaller

Na výber máme viacero rôznych tried, ktoré implementujú Marshaller interface.

1. jaxb2-marshaller: JAXB kompilátor môže prekladať W3C XML schému do niekoľkých tried.
2. xmlbeans-marshaller: Je obmedzený tým, že dokáže prekladať len triedy typu XmlObject.
3. castor-marshaller: Funguje na podobných princípoch ako JAXB.
4. jibx-marshaller: V premennej tergetClass potrebuje mať nastavené meno triedy, ktorú bude spracovávať.

Kapitola 6

JavaFX

"JavaFX je sada grafických a mediálnych balíkov vyvinutých firmou Oracle. Umožňuje návrh, testovanie a ladenie grafických používateľských prostredí, ktoré fungujú rovnako na rôznych platformách."[6] Tento framework, ako nástroj na tvorbu GUI, je poslednou časťou pri celkovom vývoji našej aplikácie.

6.1 Architektúra

Súčasná verzia JavaFX (2.2) je plne zahrnutá v Java SE 7 a v JDK (Java Development Kit). Pretože JDK je podporované na všetkých hlavných platformách, JavaFX aplikácie skompilované do JDK 7 tiež fungujú tam, kde JDK. Oracle prisľúbil synchronizované vydávanie nových verzií a aktualizácií pre všetky platformy. Aj vďaka tomu táto multiplatformová kompatibilita pomáha udržiavať rovnakú verziu aplikácií a je výhodná pre programátorov, ale aj používateľov.

Framevork JavaFX má viacero kľúčových vlastností:

1. Java API: JavaFX je napísaná ako Java API, preto aplikácie môžu použiť referencie na ktorékoľvek iné Java knižnice.

2. FXML a Scene Builder: FXML je deklaratívny značkovací jazyk založený na XML, používaný na dizajn GUI. Programátor môže vytvárať celkový dizajn v súbore pomocou FXML alebo môže použiť interaktívny JavaFX Scene Builder. Tento builder potom samostatne exportuje výsledný FXML súbor.

3. CSS: Vzhľad JavaFX aplikácií môže byť prispôsobený pomocou kaskádových štýlov (CSS). Vďaka tomuto konceptu je oddelený ľahko meniteľný design od samotnej funkčnosti jednotlivých prvkov (tlačidiel, polí, tabuliek,...).

4. Swing: Framework swing bol doteraz často používaným nástrojom v Jave na tvorbu designu, a preto ešte aj dnes je veľa aplikácií vytvorených pomocou neho. JavaFX dokáže tieto aplikácie aktualizovať do novej podoby.

5. Web: Webový komponent, ktorý využíva technológiu WebKitHTML pomáha vkladať do JavaFX aplikácií webové stránky. Podporuje sa tiež volanie JavaScriptov.

6.2 Použitie

V našej aplikácii sme využili na dizajn užívateľského rozhrania program Scene Builder. Výsledné GUI sme exportovali do FXML súboru, ktorý sme pripojili do programu (obr. 6.1). Podobne sme pridali aj CSS súbor Style, ktorý slúži na grafickú úpravu našej

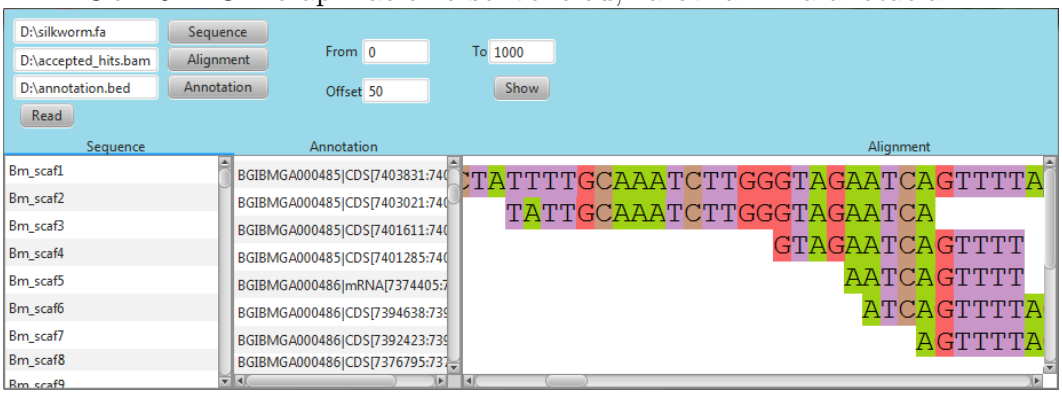
Obr. 6.1: Pripojenie FXML a CSS

```
public class RNAViewer extends Application {

    public static void main(String[] args) {
        Application.launch(RNAViewer.class, args);
    }

    @Override
    public void start(Stage stage) throws Exception {
        stage.setTitle("RNA Viewer");
        AnchorPane root = FXMLLoader
                .load(getClass().getResource("Viewer.fxml"));
        root.getStylesheets().add("Style.css");
        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

aplikácie.

Naša výsledná aplikácia - RNAViewer dokáže načítať sekvenciu a k nej zarovnanie a anotácie. Po načítaní súborov sa vytvorí zoznam sekvencií. Z neho si môžeme vybrať, ktorú sekvenciu chceme zobraziť a aj to, ktorú časť z nej. Ak sú v danej sekvencii aj anotácie, zobrazia sa v druhom zozname. Vybraná sekvencia aj s jej zarovnaniami sa zobrazia v textovom okne. Vo vrchnom riadku sa nachádza sekvencia a pod ňou usporiadané zarovnania (obr. 6.2).

Obr. 6.2: Okno aplikácie zo sekvenciou, zarovnaním a anotáciami



Záver

V tejto práci sme popísali základné princípy vybraných frameworkov v jazyku Java. Ukázali sme, ako s ich pomocou vytvoriť komplexnú aplikáciu, a v čom nám ich použitie môže uľahčiť jej vývoj.

Celú aplikáciu sme rozdelili do viacerých častí, ktoré sme previazali pomocou Spring frameworku. Vďaka tomuto riešeniu sa nám neskôr podarilo jednoducho pridávať nové vrstvy do aplikácie, ako napríklad logovanie.

Využili sme Maven, ktorý nám pomohol vytvoriť prehľadnú štruktúru celej aplikácie. Tento framework nám poslúžil pri definovaní závislostí na externých triedach a knižniciach. Použili sme ho aj na spoločné testovanie a vytváranie samostatného archívu na ďalšie použitie.

Pre lepšiu spoluprácu programátorov sme vyskúšali použiť osvedčený verziovací systém SVN. Vďaka nemu môžu na projekte pracovať súčasne aj viacerí programátori a aplikácia pritom zostáva v konzistentnom stave. Výsledné časti aplikácie sme zverejnili v úložisku na stránke projektu (`http://code.google.com/p/fmfiuk-biotools`).

Všetky frameworky, ktoré sme popísali, sa dajú využiť aj pri implementácii väčších aplikácií. Ponúkajú veľa možností, a preto je dobré zoznámiť sa s nimi a vybrať si podľa vlastných požiadaviek.

Dodatok A

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
    w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.
    apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>fmfiuk.biotools</groupId>
  <artifactId>services</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>fmfiuk.biotools.services</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>3.2.2.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>3.1.2.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-aop</artifactId>
      <version>3.1.2.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>fmfiuk.biotools</groupId>
      <artifactId>RNAViewer</artifactId>
      <version>0.0.1-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>net.sf.ehcache</groupId>
```

```
      <artifactId>ehcache-core</artifactId>
      <version>2.6.2</version>
    </dependency>
    <dependency>
      <groupId>aspectj</groupId>
      <artifactId>aspectjweaver</artifactId>
      <version>1.5.4</version>
    </dependency>
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjtools</artifactId>
      <version>1.7.2</version>
    </dependency>
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjrt</artifactId>
      <version>1.7.2</version>
    </dependency>
    <dependency>
      <groupId>cglib</groupId>
      <artifactId>cglib</artifactId>
      <version>2.2</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-simple</artifactId>
      <version>1.7.3</version>
    </dependency>
  </dependencies>
</project>
```

Dodatok B

applicationContext.xml

```
<?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:p="http://www.springframework.org/schema/p"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:oxm="http://www.springframework.org/schema/oxm"
  xmlns:cache="http://www.springframework.org/schema/cache"
  xsi:schemaLocation="http://www.springframework.org/schema/aop http://
      www.springframework.org/schema/aop/spring-aop-3.1.xsd http://www.
      springframework.org/schema/oxm http://www.springframework.org/schema
      /oxm/spring-oxm.xsd http://www.springframework.org/schema/beans http
      ://www.springframework.org/schema/beans/spring-beans.xsd http://www.
      springframework.org/schema/cache http://www.springframework.org/
      schema/cache/spring-cache-3.1.xsd">

<!-- annotations -->
<bean id="bedLineParser" class="fmfiuk.biotools.core.parser.BedLineParser
    ">
  <constructor-arg value="\t" />
</bean>

<bean id="gffLineParser" class="fmfiuk.biotools.core.parser.GffLineParser
    ">
  <constructor-arg value="\t" />
</bean>

<bean id="gtfLineParser" class="fmfiuk.biotools.core.parser.GtfLineParser
    ">
  <constructor-arg value="\t" />
</bean>

<bean id="correctFilePredicateBed" class="fmfiuk.biotools.core.common.
    CorrectFilePredicate">
  <constructor-arg value="bed" />
</bean>

<bean id="correctFilePredicateGff" class="fmfiuk.biotools.core.common.
    CorrectFilePredicate">
  <property name="fileName" value="gff" />
</bean>

<bean id="correctFilePredicateGtf" class="fmfiuk.biotools.core.common.
    CorrectFilePredicate">
  <property name="fileName" value="gtf" />
</bean>

<bean id="bedCsvAnnotationFileParser" class="fmfiuk.biotools.core.common.
    CsvAnnotationFileParser">
```

```
  <constructor-arg ref="bedLineParser" />
  <property name="canParsePredicate" ref="correctFilePredicateBed" />
</bean>

<bean id="gffCsvAnnotationFileParser" class="fmfiuk.biotools.core.common.
    CsvAnnotationFileParser">
  <constructor-arg ref="gffLineParser" />
  <property name="canParsePredicate" ref="correctFilePredicateGff" />
</bean>

<bean id="gtfCsvAnnotationFileParser" class="fmfiuk.biotools.core.common.
    CsvAnnotationFileParser">
  <constructor-arg ref="gtfLineParser" />
  <property name="canParsePredicate" ref="correctFilePredicateGtf" />
</bean>

<bean id="annotationService" class="fmfiuk.biotools.services.structure.
    AnnotationService">
  <constructor-arg>
    <list>
      <ref bean="gffCsvAnnotationFileParser" />
      <ref bean="bedCsvAnnotationFileParser" />
      <ref bean="gtfCsvAnnotationFileParser" />
    </list>
  </constructor-arg>
</bean>

<!-- nucleotideSequences -->
<bean id="nucleotideSequenceService" class="fmfiuk.biotools.services.
    structure.NucleotideSequenceService">
</bean>

<!-- Sequence -->
<bean id="sequenceService" class="fmfiuk.biotools.services.structure.
    SequenceService">
  <property name="sequenceFileIndexManager" ref="sequenceFileIndexManager
      " />
</bean>

<!-- alignments -->
<bean id="correctFilePredicateBai" class="fmfiuk.biotools.core.common.
    CorrectFilePredicate">
  <constructor-arg value="bai" />
</bean>

<bean id="correctFilePredicateBam" class="fmfiuk.biotools.core.common.
    CorrectFilePredicate">
  <constructor-arg value="bam" />
</bean>

<bean id="bamAlignmentFileReader" class="fmfiuk.biotools.core.parser.
    alignment.BamAlignmentFileReader">
  <property name="canParsePredicate" ref="correctFilePredicateBam" />
</bean>

<bean id="alignmentService" class="fmfiuk.biotools.services.structure.
    AlignmentService">
  <constructor-arg>
    <list>
      <ref bean="bamAlignmentFileReader" />
    </list>
  </constructor-arg>
</bean>

<!-- Storing information into XML files with Spring Marshalling -->
<bean id="marshaller" class="org.springframework.oxm.castor.
    CastorMarshaller" >
```

```
  <property name="mappingLocation" value="d:/Programs/fmfiuk.biotools.
      services/src/main/config/spring/mapping.xml" />
</bean>

<bean id="sequenceFileIndexManager" class="fmfiuk.biotools.core.parser.
    sequence.SequenceFileIndexManager">
  <constructor-arg value="settings.xml" />
  <property name="marshaller" ref="marshaller" />
  <property name="unmarshaller" ref="marshaller" />
</bean>
<!-- Spring AOP Logging -->
<bean id="logger" class="fmfiuk.biotools.services.logging.ConsoleLogger"
    />
  <aop:config>
    <aop:pointcut id="selectAll" expression="execution(* fmfiuk.biotools.
        core..*.*(..))" />
    <aop:advisor advice-ref="logger" pointcut-ref="selectAll" />
  </aop:config>
<beans>
```

Literatúra

[1] KANISOVÁ Hana a MÜLLER Miroslav. *UML srozumitelně.* Computer Press, 2004.

[2] COLLINS-SUSSMAN Ben a FITZPATRICK Brian W. a PILATO C. Michael. *Version Control with Subversion.* O'Reilly Media, 2004.

[3] O'BRIEN Tim a kol. *Maven: The Definitive Guide.* O'Reilly Media, 2009.

[4] MASSOL Vincent a kol. *Better Builds with Maven.* Library Press, 2008.

[5] WALLS Craig. *Spring in Action.* Manning Publications Co., 2011.

[6] Oracle. Javafx architecture. `http://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.htm`, máj 2013.

[7] Tutorialspoint. Spring applicationcontext container. `www.tutorialspoint.com/spring/spring_applicationcontext_container.htm`, apríl 2013.

[8] SpringSource. Spring marshalling. `http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/oxm.html`, apríl 2013.