Vysoká škola ekonomická v Praze
Fakulta informatiky a statistiky
Katedra informačních technologií

Studijní program: Aplikovaná informatika
Obor: Informační systémy a technologie

Popište a porovnejte nástroje Maven a Gradle

DIPLOMOVÁ PRÁCE

Student : Bc. Lucie Tvrdíková
Vedoucí : Ing. Rudolf Pecinovský, CSc.
Oponent : Ing. Vladimír Oraný

2013

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracovala samostatně a že jsem uvedla všechny použité prameny a literaturu, ze které jsem čerpala.

V Praze dne 30. dubna 2013 . . . . . .

Lucie Tvrdíková

Poděkování

Ráda bych poděkovala vedoucímu mé práce Ing. Rudolfu Pecinovskému, CSc. za jeho rady a připomínky k práci. Dále chci poděkovat za spolupráci všem mým kolegům, kteří mi předali své zkušenosti, názory a poznatky z praxe.

Abstrakt

Cílem práce je analýza schopností systémů Maven a Gradle a jejich vzájemné porovnání. V průběhu analýzy vznikly uživatelské příručky v českém jazyce k oběma nástrojům. K dosažení stanovených cílů je využita práce se zdroji, a to konkrétně vydaných uživatelských příruček nebo knih v anglickém jazyce. Uživatelské příručky popisují základy systémů, řízení závislostí, testování a multiprojektové buildy. Porovnání je vytvořeno zejména na základě uživatelských příruček a osobní zkušenosti autorky. Praktická část práce obsahuje soubory pro build aplikace v systému Maven i Gradle.

Klíčová slova

Maven, Gradle, Java, buildovací nástroj

Abstract

The goal of this work is to analyze systems Maven and Gradle and to compare one against the other. Whilst I was analyzing both the systems user guides in Czech language were created. To achieve the set of goals of the thesis I had to use user guides and books that were written in English as a source of information. The user guides describes the system basics, dependency management, testing and multi-project builds. Comparisons between the two systems were created from information from the user guides and personal experience of the author. The practical part of the thesis contains files for build application in Maven and Gradle.

Keywords

Maven, Gradle, Java, build tool

Obsah

1 Úvod 1

1.1 Vymezení tématu práce 1
1.2 Důvod výběru tématu 1
1.3 Cíle práce 1
1.4 Způsoby a metody dosažení cíle 2
1.5 Předpoklady a omezení práce 2
1.6 Struktura práce 2

2 Buildovací nástroje obecně 3

2.1 Buildovací nástroj obecně 3
2.2 Historie buildovacích nástrojů 3
2.3 Systém Maven obecně 4
2.4 Systém Gradle obecně 5

3 Instalace 6

3.1 Instalace systému Maven 6
3.2 Instalace systému Gradle 6

4 Základy systému Maven 7

4.1 Adresářová struktura projektu 7
4.2 POM 7
4.2.1 Proměnné v POM 11
4.3 Životní cyklus 11
4.4 Pluginy 12
4.5 Profily 13
4.6 Logování 14
4.7 Plugin Archetype 14
4.8 Plugin Site 17

5 Základy systému Gradle 18

5.1 Skriptovací soubor build.gradle 18
5.2 Task 18
5.2.1 Hello World v Gradlu 18
5.2.2 Metody objektu DefaultTask 19
5.2.3 Proměnné objektu DefaultTask 20
5.2.4 Typy tasků 21
5.3 Proměnné 23
5.4 Projekt 24
5.5 Práce se soubory a adresáři 25

5.6 Build 25
5.7 Plugin 26
5.7.1 Plugin Java 26
5.8 Gradle wrapper 29

6 Řízení závislostí 30
6.1 Řízení závislostí v systému Maven 30
6.1.1 Úložiště v systému Maven 30
6.1.2 Závislosti v systému Maven 31
6.1.3 Publikování artefaktů v systému Maven 33
6.2 Řízení závislostí v systému Gradle 34
6.2.1 Úložiště v systému Gradle 34
6.2.2 Závislosti v systému Gradle 35
6.2.3 Publikování artefaktů v systému Gradle 37

7 Testování 38
7.1 Testování se systémem Maven 38
7.2 Testování se systémem Gradle 39

8 Multiprojektový build 44
8.1 Multiprojektový build v systému Maven 44
8.2 Multiprojektový build v systému Gradle 46

9 Rešerše dostupné literatury 49
9.1.1 Uživatelské příručky od vydavatelů 49
9.1.2 Vydané knihy o systému Maven 49
9.1.3 Vydané knihy o systému Gradle 50

10 Porovnání systémů Maven a Gradle 51
10.1 Základy systémů 51
10.1.1 Skriptovací soubor 51
10.1.2 Wrapper 51
10.1.3 Logování 51
10.1.4 Šablony projektů 52
10.1.5 Vytvoření dokumentace 52
10.1.6 Build 52
10.1.7 Profily 52
10.1.8 Ostatní 52
10.2 Řízení závislostí 53
10.3 Testování 53
10.4 Multiprojektové buildy 53
10.5 Ostatní 54
10.6 Závěr 54

11 Praktická část 56
11.1 Instalace 56
11.2 Příkazy 56
11.2.1 Spuštění buildu 56
11.2.2 Spuštění Sonar reportu 56
11.3 Úskalí 56
11.3.1 Plugin Maven2Gradle 57
11.3.2 Gradle a Sonar 57
11.3.3 Překlad Groovy kódu 57
11.3.4 Rozsah provided 58
11.3.5 Integrační testy 58
11.3.6 Využití lokálního Maven úložiště v Gradlu 58
12 Závěr 60
Terminologický slovník 61
Seznam literatury 63
Seznam obrázků a tabulek 66
Základní text 66
Seznam obrázků 66
Seznam tabulek 66
Přílohy 67

1 Úvod

1.1 Vymezení tématu práce

Tématem této diplomové práce je analýza schopností a porovnání nástrojů Apache Maven (dále jen Maven) a Gradle. Pro oba systémy je v rámci analýzy vypracován jednoduchý tutoriál s příklady. Schopnosti obou nástrojů jsou poté demonstrovány na reálné aplikaci, kterou ve své diplomové práci Systém pro vedení bakalářských a diplomových prací vypracoval Ing. Petr Bouda.

1.2 Důvod výběru tématu

Toto téma jsem si zvolila, jelikož je mi velice blízké. Již čtyři roky pracuji na pozici Java Developer a během své praxe jsem se setkala nejdříve s nástrojem Maven, který je momentálně asi nejvíce rozšířen, a na současném projektu jsem se seznámila i se systémem Gradle. U obou nástrojů znám a používám ty nejzákladnější funkce, tudíž bych si ráda při psaní této práce rozšířila své dovednosti s těmito nástroji.

1.3 Cíle práce

Hlavním cílem práce je analýza schopností nástrojů Maven a Gradle. V průběhu analýzy vzniknou při popisování vlastností obou systémů uživatelské příručky v českém jazyce s ukázkovými příklady. V současné době je na trhu nebo na internetu několik knih a příruček pro oba nástroje, avšak žádná z nich není v češtině psaná v dostatečném rozsahu.

Dalším cílem je nabídnout čtenáři porovnání mezi oběma nástroji a díky tomu mu pomoci při výběru vhodnějšího nástroje pro jeho projekt. Na internetu, v knihách nebo příručkách jsou sice informace o různých výhodách a nevýhodách, nepodařilo se mi však nalézt žádné kompletní porovnání.

Schopnosti obou nástrojů jsou demonstrovány na reálné aplikaci pomocí konfiguračního skriptu pro Maven i pro Gradle.

1.4 Způsoby a metody dosažení cíle

K dosažení stanovených cílů využiji práci se zdroji. Informace jsou primárně získávány z uživatelských příruček vydávaných přímo distributorem nástroje, využito je i vydaných knih. Jelikož tyto informace nejsou v českém jazyce standardně dostupné v požadovaném rozsahu, jsou tyto údaje přeloženy z angličtiny.

Dále v práci využívám příklady na konkrétním programu a vypracovala jsem konfigurační skripty pro jednu a tu samou aplikaci v Mavenu a Gradlu. Konfigurační soubor pro Maven byl upraven a rozšířen, pro Gradle je skript vytvořen tak, aby prováděl to samé jako Maven.

1.5 Předpoklady a omezení práce

Cílem této práce není podrobné a úplné zmapování a popsání všech funkcí daných nástrojů, nýbrž opravdu jen těch v praxi nejvyužívanějších. Největším omezením této práce je její předepsaný maximální rozsah, protože dovednosti obou nástrojů jsou značně rozsáhlé. V práci nejsou uvedeny ani všechny možné způsoby, jak provést s daným nástrojem jednu operaci. Zejména Gradle má pro mnoho operací alespoň tři různé způsoby realizace.

Práce je určena především pracovníkům a studentům z oblasti vývoje softwaru, kteří se chtějí naučit správně využívat tyto nástroje, nebo těm, kteří chtějí přehledně zjistit, v čem je daný nástroj lepší než ten druhý a naopak.

1.6 Struktura práce

Práce je strukturovaná do dvanácti kapitol. První kapitolou je úvod, následuje vysvětlení pojmu buildovací nástroj, jeho stručná historie a obecný popis systému Maven a Gradle. Ve třetí kapitole popisuji instalaci obou systémů. Následující kapitola je zaměřena na základy systému Maven. Pátá kapitola je věnována základům systému Gradle. Popisu řízení závislostí v obou systémech se věnuji v šesté kapitole. V sedmé kapitole se zabývám využitím obou nástrojů při testování. Osmá kapitola pojednává o popisuje využití nástrojů na multiprojektových projektech. Devátá kapitola obsahuje rešerši dostupné literatury. Desátá kapitola je věnována celkovému porovnání systémů Maven a Gradle. V jedenácté kapitole popisuji praktickou část diplomové práce. Poslední kapitola je zaměřena na celkové shrnutí a zhodnocení přínosů této práce.

2 Buildovací nástroje obecně

V této kapitole vysvětlím, co je v této práci chápáno pod pojmem buildovací nástroj. Zaměřím se na historický vývoj buildovacích nástrojů a obecně představím nástroje Maven a Gradle.

2.1 Buildovací nástroj obecně

Systémy Maven a Gradle se obecně řadí mezi takzvané buildovací nástroje. Pod tímto pojmem rozumíme nástroj pro sestavení (angl. build) a distribuci aplikace. Tento pojem je značně omezený a nezahrnuje všechny dovednosti těchto nástrojů. Někdy se v literatuře lze setkat s příhodnějším pojmenováním nástroj pro řízení projektů. Jaký je mezi nimi rozdíl? Pod pojmem buildovací nástroj rozumíme nástroj, který předzpracovává, kompiluje, zabaluje, testuje a distribuuje. Nástroj pro řízení projektů poskytuje tyto schopnosti, ale jako přidanou hodnotu navíc dokáže vytvářet reporty, generovat webové stránky a usnadnit komunikaci mezi členy týmu (1 s. 1).

Zajímavé je, že Maven v literatuře hodně dbá na používání výrazu nástroj pro řízení projektů, ale setkáme se i s označením buildovací nástroj. Gradle ve své prezentaci a literatuře dbá na označení buildovací systém.

2.2 Historie buildovacích nástrojů

První programátoři, kteří začali používat jazyk Java, neměli mnoho možností, jak kompilovat a buildovat projekty. Bylo třeba nástroje, který by byl nezávislý na platformě. Průkopníkem byl nástroj Apache Ant (Another Neat Tool, dále jen Ant), který vznikl v roce 2000 a původně vychází z programu Make, jenž na platformě závislý byl. Jeho schopnosti byly inspirací pro Ant – zejména kde začít, jaké schopnosti by nástroj měl mít a jaké by naopak mít neměl. Ant poskytl způsob, jak popsat sestavení Java projektů, které bude přenositelné, a zároveň je rozšiřitelný. Měl ale mnoho nevýhod, například byl hodně upovídaný, znovupoužitelnost buildovacích skriptů byla nedostatečná a především měl nedostatečné řízení závislostí (2 s. xi–xii) (3).

Tyto nedostatky vedly ke vzniku dalšího, velice populárního a dnes asi nejpoužívanějšího nástroje Maven. Maven zavedl mnoho konvencí, které se používají dodnes, např. pevnou strukturu projektu, podle které Maven automaticky ví, kde hledat např. zdrojové soubory nebo životní cyklus buildu. V Mavenu díky tomu odpadla nutnost mít konfigurační soubory buildu v mnohém zkopírované, tak jako tomu bylo u Antu, kde se vždy např. muselo nadefinovat, ve kterém adresáři jsou zdrojové soubory projektu (2 s. xi–xii).

Když opadlo prvotní nadšení z Mavenu a zjistilo se, že i tento systém má své nevýhody, otevřela se cesta ke vzniku nových buildovacích nástrojů, jež by kombinovaly to nejlepší z dosavadních nástrojů. Vzniklo mnoho systémů, ale většina z nich se neprosadila, u některých skončil vývoj hned po vydání první verze nebo trvalo několik měsíců, než vyšel další release, který by opravoval chyby, jež byly nalezeny uživateli. Za úspěšný lze považovat až Gradle, který se mezi uživateli rozšířil, neustále se zdokonaluje, přibývají nové funkčnosti a spolu s opravami chyb probíhá každých šest týdnů release (2 s. xi–xii) (3). Více o dalších vzniklých buildovacích nástrojích a jejich hlavní charakteristice viz prezentace (4).

2.3 Systém Maven obecně

Jak již bylo napsáno, lze se setkat s mnoha definicemi pro buildovací nástroje a i s mnoha definicemi, co je Maven. Zde uvádím některé z nich:

„Maven je framework pro řízení projektu“ (5 s. 22).

„Maven je nástroj pro správu, řízení a automatizaci buildů aplikací“ (6).

„Maven je nástroj pro řízení projektu, který zahrnuje POM (Project Object Model), sadu standardů, životní cyklus projektu, systém pro správu závislostí a logiku pro provedení transformací vložených pluginů v definovaných fázích životního cyklu“ (1 s. 1).

Maven je navržen pro usnadnění práce při buildování aplikací a nejčastější agendou s tím spojenou, jako například (8):

- tvorba dokumentace
- tvorba reportů
- řízení závislostí
- release
- distribuce

Tvůrci Mavenu si stanovili při jeho vytváření několik dílčích cílů (9):

- snadný proces buildu aplikace
- stanovení jednotného systému buildů
- poskytování kvalitních informací o projektu
- poskytování osvědčených postupů (angl. best practices)
- povolení transparentního přidávání nových funkcí

Výhodou je podpora ve všech třech nejrozšířenějších IDE (Integrated Development Environment), a to NetBeans, Eclipse a IntelliJ IDEA. Touto podporou se ve své práci nezabývám, více informací lze nalézt v (10).

Maven využívá konvence nad konfigurací, což je velmi jednoduchý koncept výchozích nastavení, která není třeba dále konfigurovat, přitom je ale možné tyto výchozí hodnoty změnit, pokud je to nezbytné. Příkladem konvence je např. standardní adresářová struktura, která bude popsána dále. Existuje mnoho konvencí, které Maven zavedl např. pro kompilování zdrojového kódu, generování webových stránek a mnohé další. To velice usnadňuje práci, protože např. v Antu se všechny tyto konvence musely nastavovat ručně, tj. zapsat do konfiguračního souboru (11 s. 2).

2.4 Systém Gradle obecně

Gradle kombinuje výkon a flexibilitu Antu s řízením závislostí a konvencemi Mavenu do lepšího způsobu, jak buildovat. Gradle ale není pouze kombinace Antu a Mavenu, bere si i to nejlepší z nástrojů Make, Ivy, Rake, Gant, Scons, SBT, Leinengen a Buildr (2 s. xiii).

Gradle je navržen tak, aby byl buildovacím jazykem, ne rigidním frameworkem. Poskytuje podporu pro Java, Groovy, Scala, Web a OSGi projekty. Dále podporuje Ant tasky (česky úkoly, úlohy) a projekty, velice jednoduše lze importovat Ant build. S převodem Maven buildovacího skriptu je to o něco složitější, ale je to také možné. Maven a Ivy úložiště jsou podporována jak pro načtení, tak i publikování závislostí. Gradle je používán na velkých open-sourcových projektech, jako například Spring, Hibernate nebo Grails (12 s. 7–8).

Gradle je také podporován v IDE Eclipse a IntelliJ IDEA. Touto podporou se ve své práci nezabývám, více informací lze nalézt v (12) nebo (13). Existuje i podpora pro NetBeans, která ale není popsaná v oficiální dokumentaci, informace jsou dostupné zde (14).

3 Instalace

V této kapitole popíši instalaci obou nástrojů, postup je stejný, jen je potřeba u každého nástroje nastavit jiné systémové proměnné prostředí. Veškerý postup popsaný v této práci je realizován v operačním systému Windows 7 Home Premium 64bit. Instalací pro jiné operační systémy se zde nezabývám, postup pro jiný operační systém lze snadno dohledat na oficiálních stránkách projektů nebo v literatuře.

Maven i Gradle vyžadují nainstalované JDK Javy, a to verze 1.5 nebo vyšší. Zda máte Javu nainstalovanou a jaké verze, je možné zjistit zadáním příkazu do příkazové řádky `java -version`. V této práci je využito JDK verze 1.6.0_37. Pokud Javu nemáte nainstalovanou, informace, jak ji nainstalovat, naleznete například zde (15).

3.1 Instalace systému Maven

Na oficiálních stránkách projektu Maven (16) je třeba stáhnout aktuální verzi nástroje. V této práci je vše demonstrováno na verzi 3.0.4. Pro účely mé práce postačí stažení binární verze instalace. Stažený soubor je nutno rozbalit a složku umístit na vybrané místo na disku. Do systémových proměnných se poté musí přidat nová proměnná, a to `M2_HOME`, která ukazuje na složku s Mavenem, např. `D:\maven`. Následně je ještě nutné přidat na konec systémové proměnné `Path` cestu k `bin` složce Mavenu - `;%M2_HOME%\bin` (10 s. 8–10).

Pro ověření úspěšné instalace se zadá do příkazové řádky příkaz `mvn -version`. Pokud je vše nastaveno správně, systém vypíše verzi Mavenu, Javy a další informace.

3.2 Instalace systému Gradle

Na oficiálních stránkách projektu Gradle (17) je třeba stáhnout aktuální verzi nástroje. V této práci je vše demonstrováno na verzi 1.3. Pro účely mé práce postačí stažení binární verze instalace. Stažený soubor je nutno rozbalit a složku umístit na vybrané místo na disku. Do systémových proměnných se poté musí přidat nová proměnná, a to `GRADLE_HOME`, která ukazuje na složku s Gradlem, např. `D:\gradle`. Následně je ještě nutné přidat na konec systémové proměnné `Path` cestu k `bin` složce Gradlu - `;%GRADLE_HOME%\bin (2 s. 3–6)`.

Pro ověření úspěšné instalace se zadá do příkazové řádky příkaz `gradle -version`. Pokud je vše nastaveno správně, systém vypíše verzi Gradlu, Javy a další informace.

4 Základy systému Maven

V této kapitole představím základní konvence Mavenu a další základní prvky, na kterých Maven staví.

4.1 Adresářová struktura projektu

Maven definuje základní adresářovou strukturu projektu a obsah těchto adresářů jako jeden ze základních standardů. To velice usnadňuje orientaci uživatelů v projektu, jakmile se tuto strukturu jednou uživatel naučí, dokáže se rychle zorientovat na jiném Maven projektu. Na nejvyšší úrovni, v kořenové složce projektu, se nachází konfigurační soubor buildu `pom.xml`, který bude popsán v následující kapitole. Složka `src`, která je umístěna rovněž v kořenové složce projektu, obsahuje všechny zdroje, které jsou nutné ke kompilaci, testování, dokumentaci a nasazení (11 s. 2). Ty nejzákladnější složky a jejich popis definuje tabulka 1.

Tabulka 1: *Základní Maven adresářová struktura (18)*

Složka	Popis
`src/main/java`	Obsahuje soubory se zdrojovým kódem aplikace s příponou `java`.
`src/main/resources`	Obsahuje ostatní soubory neobsahující zdrojový kód, např. konfigurační soubory s příponou `properties`.
`src/test/java`	Obsahuje soubory se zdrojovým kódem pro testování aplikace s příponou `java`.
`src/test/resources`	Obsahuje ostatní soubory neobsahující zdrojový kód, např. konfigurační soubory určené pro testy s příponou `properties`.
`src/target`	Složka `target` je používána k uložení všech výstupních souborů při buildu.

4.2 POM

POM je XML dokument, který obsahuje deklarativní popis projektu (11 s. 27). Pro základní představu POM pojmenovává projekt, poskytuje sadu identifikátorů projektu, může přizpůsobit chování pluginu nebo buildu a mnoho dalších aktivit (1 s. 19). Pokud není

stanoveno jinak, všechny POM dědí z tzv. `SuperPOM`, které definuje sadu standardních hodnot, např. adresářovou strukturu popsanou výše (1 s. 21).

Nejjednodušší POM, které zabalí zdrojové soubory projektu ze složky `src/java/main` do jar souboru a spustí testy ve složce `src/main/test`, musí obsahovat minimálně následující čtyři povinné elementy uvedené ve výpisu 1.

Výpis 1: Nejjednodušší POM (1 s. 25)

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch08</groupId>
  <artifactId>simplest-project</artifactId>
  <version>1</version>
</project>
```

Element `modelVersion` určuje XML schéma, které popisuje strukturu POM (19). Další tři povinné atributy patří mezi takzvané Maven souřadnice (angl. Maven Coordinates) a jsou popsány v tabulce 2. `GroupId` a `artifactId` musí být unikátní, protože se používají ve jménu jaru a k umístění v úložišti.

Tabulka 2: *Maven souřadnice (11 s. 28–29)*

Jméno atributu	**Popis**
`groupId`	Skupina, společnost, tým, organizace, projekt nebo jiná skupina, která projekt vytvořila. Podobá se balíčkům, např. org.apache.
`artifactId`	Unikátní identifikátor, který reprezentuje projekt.
`version`	Verze projektu. Projekty, které nebyly oficiálně vydány, ale jsou ve vývoji, mají ve verzi označení SNAPSHOT.

Verze aplikace by se ideálně měla skládat z několika čísel oddělených tečkou, případně

i textu podle následujícího schématu:

```
<hlavní verze>.<vedlejší verze>.<přírůstková verze>-<kvalifikátor>
```

Tomuto vzoru odpovídá příklad: 1.3.5-alpha (1 s. 26).

POM obsahuje 4 základní části:

- základní elementy
- nastavení buildu
- metadata projektu
- nastavení prostředí

Ve výpisu 2 uvádím základní strukturu POM a popis elementů, který je částečně převzat z (19), kde je uveden kompletní přehled struktury POM a popis všech elementů. V práci jsou dále postupně detailněji rozebrány ty nejpoužívanější elementy.

Výpis 2: Základní struktura POM a její popis (10 s. 19–20)

```
<!-- Project je kořenový element -->
<project>
    <!-- Definuje verzi XML schématu -->
    <modelVersion>4.0.0</modelVersion>

    <!-- Základní elementy -->
    <groupId>...</groupId>
    <artifactId>...</artifactId>
    <version>...</version>
    <packaging>Typ artefaktu, který projekt produkuje, defaultně jar</packaging>
    <dependencies>Definice závislostí</dependencies>
    <parent>Rodičovské POM</parent>
    <dependencyManagement>...</dependencyManagement>
    <modules>Moduly projektu</modules>
    <properties>Definice vlastních proměnných</properties>

    <!-- Nastavení buildu -->
    <build>...</build>
    <reporting>Konfigurace reportů</reporting>

    <!-- Metadata projektu -->
    <name>Jméno projektu používané v dokumentaci</name>
    <description>Popis projektu použitý v dokumentaci</description>
    <url>URL projektu, použito v dokumentaci</url>
    <inceptionYear>Rok založení projektu</inceptionYear>
    <licenses>Licence projektu</licenses>
    <organization>Organizace</organization>
    <developers>Vývojáři projektu, kteří se podílejí na vývoji
                 přímo</developers>
    <contributors>Přispěvovatelé, kteří se přímo nepodílejí na vývoji
                 kódu</contributors>

    <!-- Prostředí -->
    <issueManagement>...</issueManagement>
    <ciManagement>Management kontinuální integrace</ciManagement>
    <mailingLists>...</mailingLists>
    <scm>Konfigurace SCM</scm>
    <prerequisites>Popis předpokladů prostředí</prerequisites>
    <repositories>Definice úložišť</repositories>
    <pluginRepositories>Definice úložišť pro pluginy</pluginRepositories>
    <distributionManagement>Konfigurace vzdáleného úložiště pro
                              distribuci</distributionManagement>
    <profiles>Profily</profiles>
</project>
```

4.2.1 Proměnné v POM

V POM existují tři druhy proměnných (20):

- prostředí - např. `env.PATH`,
- projektové - každý element v POM může být využitý jako proměnná — např. `project.groupId` nebo `project.build.sourceDirectory`
- nastavení - elementy v `settings.xml` - např. `settings.offline`

Dále lze využít i systémové proměnné, což jsou všechny proměnné dostupné pomocí `java.lang.System.getProperties()`, např. `user.home`, nebo vlastní proměnné definované v `pom.xml` nebo `settings.xml` v sekci `<properties>`. Odkaz na proměnnou se vždy uvádí se znakem dolaru uvnitř složených závorek `${user.home}` (1 s. 28–29), což je demonstrováno ve výpisu 3.

Výpis 3: Využití vlastní proměnné

```
<project>
  ...
  <properties>
    <junit.version>4.11</junit.version>
  </properties>
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
    </dependency>
  </dependencies>
</project>
```

4.3 Životní cyklus

Životní cyklus je organizovaná sekvence fází (1 s. 49). Každá fáze na sebe může mít napojený jeden nebo více cílů (angl. goals). Pokud fáze nemá žádný cíl, není provedena. Cíl je specifický úkol, dá se říci jednotka práce. Cíl může být napojen na jednu nebo více fází životního cyklu. Cíl, který není napojen na žádnou fázi, může být volán přímo (21).

Maven má tři vestavěné životní cykly, a to `default`, `clean` a `site`.

Životní cyklus `default` obsahuje více než 20 fází, v této práci uvádím jen ty nejpoužívanější, kompletní výčet lze nalézt v (1 s. 52–53):

- `validate` - první fáze, ověří informace o projektu — zda jsou dostupné a správné

- `compile` - zkompiluje zdrojový kód
- `test` - spustí testy
- `package` - zabalí zdrojový kód do distribučního formátu — např. jar, war
- `integration-test` - zpracuje distribuční balíček a nasadí ho na integrační prostředí, kde se spustí integrační testy
- `verify` - ověří, že je package validní
- `install` - nainstaluje package do lokálního úložiště
- `deploy` - poslední fáze, nainstaluje package na vzdáleném úložišti

Životní cyklus `clean` zpracovává čištění projektu a má tři fáze (10 s. 21):

- `pre-clean` - provádí procesy před čištěním projektu
- `clean` - odstraní všechny soubory vytvořené v předchozích buildech, tzn. smaže složku s buildem
- `post-clean` - provádí procesy potřebné k dokončení čištění projektu

Životní cyklus `site` generuje dokumentaci. Má následující čtyři fáze (10 s. 21–22):

- `pre-site` - provádí procesy požadované před generací webu
- `site` - generuje stránky projektové dokumentace
- `post-site` - provádí procesy potřebné k dokončení generace webu a připravuje místo pro nasazení
- `site-deploy` - nasadí stránky dokumentace na zadaný webový server

Před spuštěním konkrétní fáze jsou spuštěny všechny předchozí fáze. Pokud tedy spustím fázi test příkazem `mvn test`, provede se nejprve validace, poté kompilace a na závěr se teprve spustí samotné testy. Lze spustit i více fází z různých životních cyklů za sebou, např. příkaz `mvn clean install` nejprve projekt vyčistí a poté nainstaluje (10 s. 21).

4.4 Pluginy

Všechna práce Mavenu je prováděna pluginy. Maven plugin je kolekce jednoho nebo více cílů. Každý cíl reprezentuje nějakou schopnost pluginu. Příkladem pluginu může být plugin Jar, který má cíle pro vytvoření jar souboru. Lze vytvořit i vlastní plugin nejen

v jazyce Java (11 s. 22–23). Tématem vytvoření vlastního pluginu se v této práci nezabývám, více informací lze nalézt v (1).

V příkazové řádce se plugin volá jeho jménem, v příkladu výpis 4 je to `archetype`, následuje dvojtečka a cíl, který se má provést, v příkladu je cíl `generate`. Po zadání příkazu vyhledá Maven v aktuální složce soubor POM, ze kterého získá potřebné informace a provede daný cíl (22).

Výpis 4: Příklad volání pluginu

```
mvn archetype:generate
```

Velice užitečný je cíl `describe` pluginu `help`, který vypíše informace o pluginu, jehož jméno se zadá místo `<název pluginu>` v následujícím příkazu: `mvn help:describe -Dplugin=<název pluginu> -Dfull`. Tento příkaz vypíše nejen informace o pluginu, ale i jeho cíle se stručným popisem, co daný cíl provádí (1 s. 95).

4.5 Profily

Profily umožňují přizpůsobit build určitému prostředí (1 s. 66). Při vývoji aplikací existuje několik prostředí, na které se aplikace nasazuje, jako např. developerské, testovací nebo produkční prostředí. Každé prostředí většinou vyžaduje jiné nastavení, např. pro konfiguraci databáze.

Profily se v POM definují v sekci `<profiles>`, která může obsahovat jeden nebo více elementů `<profile>`, který reprezentuje určitý profil. Každý profil v POM musí mít své unikátní id definované v elementu `<id>`. Profil je aplikován pomocí argumentu v příkazové řádce `-P<profile_id>`, kde se za `<profile_id>` dosadí konkrétní id profilu, např. `mvn install -Pproduction` (1 s. 70). Ty nejvýznamnější elementy, které jsou povolené v sekci `<profiles>`, lze vidět ve výpisu 5.

Profily mohou být definovány v `pom.xml` nebo `settings.xml`. Pro zjištění, kde jsou jaké profily definovány, slouží příkaz `mvn help:active-profiles` (1 s. 75).

Výpis 5: Elementy povolené v sekci `profiles` (1 s. 71)

```
<project>
  ...
  <profiles>
    <profile>
      <id>...</id>
      <build>
        <defaultGoal>...</defaultGoal>
        <finalName>...</finalName>
        <resources>...</resources>
        <testResources>...</testResources>
        <plugins>...</plugins>
      </build>
      <reporting>...</reporting>
      <modules>...</modules>
      <dependencies>...</dependencies>
      <dependencyManagement>...</dependencyManagement>
      <distributionManagement>...</distributionManagement>
      <repositories>...</repositories>
      <pluginRepositories>...</pluginRepositories>
      <properties>...</properties>
    </profile>
  </profiles>
</project>
```

4.6 Logování

Maven poskytuje tři úrovně logování, které se ovládají pomocí následujících parametrů v příkazové řádce (1 s. 87):

- `-e` nebo `--errors` - používá se k diagnostice chyb v Mavenu
- `-X` nebo `--debug` - vypisuje všechny logovací zprávy, používá se při hledání chyb v nastavení Mavenu
- `-q` nebo `--quiet` - vypisuje pouze chyby nebo problémy

4.7 Plugin Archetype

„*Archetype je šablona pro Maven - projekt, která je používána pomocí pluginu Archetype k vytvoření nových projektů*“ (1 s. 209). Pro vytvoření projektu se používá cíl `generate` s následujícími parametry (1 s. 210):

- `groupId`, `artifactId`, `version` - využito v POM
- `packageName` - defaultně se bere `groupId`
- `archetypeGroupId` - `groupId` archetypu, který se má použít k vygenerování projektu
- `archetypeArtifactId` - `artifactId` archetypu, který se má použít k vygenerování projektu
- `archetypeVersion` - verze archetypu, který se má použít k vygenerování projektu
- `interactiveMode` - pokud je interaktivní mód zapnutý, bude uživatel vyzván k zadání výše uvedených parametrů; pokud je nastaven na `false`, vezmou se uvedené hodnoty z příkazové řádky, defaultně je nastaveno `true`

Počet šablon v pluginu neustále roste a bohužel se mění i čísla šablon. Nejzákladnější šablona pro vygenerování jednoduchého projektu, která vytvoří jeden zdrojový soubor, jeden testovací soubor a `pom.xml`, se nazývá `maven-archetype-quickstart` a její číslo je defaultně nabízeno za seznamem dostupných šablon při interaktivním módu. Doporučuji ale zadávat raději všechny údaje v příkazu tak, jak demonstruji ve výpisu 6.

Výpis 6: Příkaz pro vygenerování projektu `maven-archetype-quickstart`

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app
-DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Spuštěním příkazu ve výpisu 6 se vygeneruje složka projektu a následující struktura a soubory (23):

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

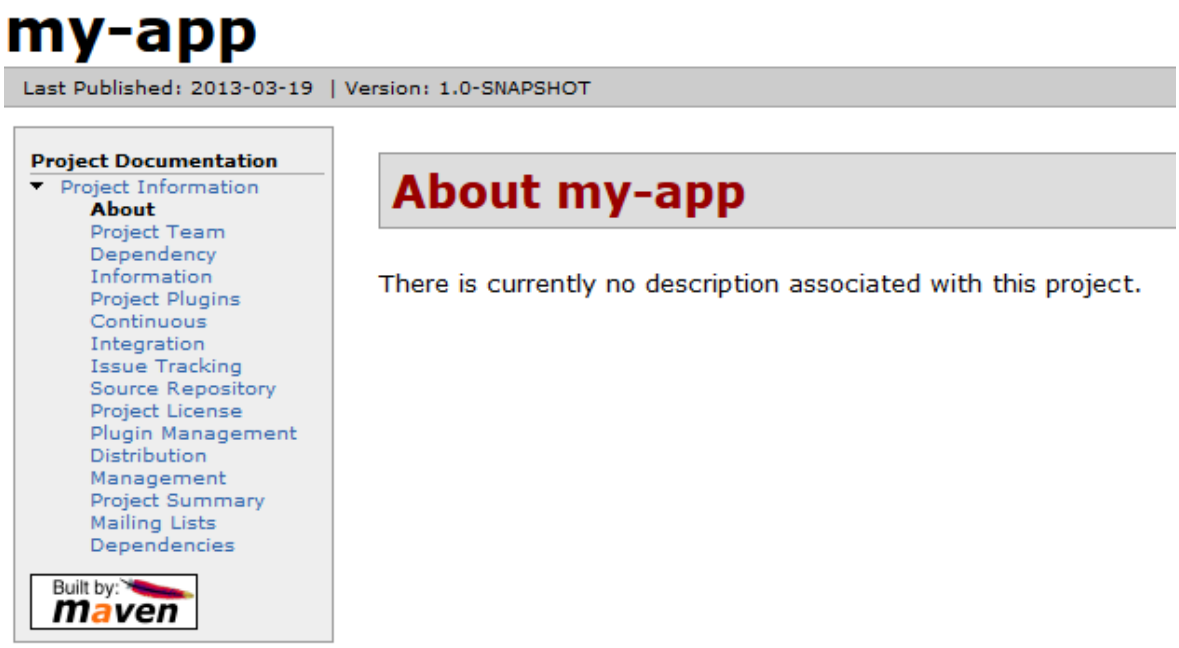
Vygenerované POM ukazuje výpis 7. Je zde nadefinované jen nutné minimum a závislost na JUnit, protože projekt obsahuje testovací třídu.

Výpis 7: POM vygenerované pluginem Archetype

```
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>my-app</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

4.8 Plugin Site

Plugin Site slouží pro tvorbu stránek o dokumentaci projektu. Stránky se vygenerují do adresáře `target/site` jednoduchým příkazem `mvn site`. Stránky lze i nasadit na server, který je nadefinován v sekci `<distributionManagement> <site>`, pomocí příkazu `mvn site-deploy`. Informace pro tyto stránky jsou čerpány z POM. Defaultní vzhled stránek je vidět na obrázku 1. Stránky je možné libovolně upravovat, lze jim např. dát vlastní kaskádní styl nebo přidat obrázky (24). Více informací lze nalézt zde (1).



Obrázek 1:
Náhled vygenerovaných stránek dokumentace projektu

5 Základy systému Gradle

V této kapitole popisuji základy systému Gradle. Gradle využívá konvence Mavenu, a proto zde nebudu znovu popisovat adresářovou strukturu projektu.

5.1 Skriptovací soubor build.gradle

Obdobou konfiguračního souboru `pom.xml` pro Maven je v Gradlu soubor `build.gradle`. Gradle v něm používá DSL (Domain Specific Language), což je flexibilní jazyk založený na Groovy, který může být uživatelem rozšířen. Groovy je dynamický jazyk JVM (Java Virtual Machine), který je v mnoha ohledech i syntaxi podobný Javě, ale s významnými rozdíly. Pro začátky s Gradlem není nutné znát jazyk Groovy, ale postupně, při složitějších operacích, je znalost Groovy výhodou (12 s. 8).

5.2 Task

Konfigurační skript buildu, který se defaultně pojmenovává `build.gradle`, se skládá z projektu a tasků. Projekt má jeden nebo více tasků. Tasky v Gradlu jsou ekvivalentní s targety v Antu (13 s. 27–28). Tasky v Gradlu jsou objekty s metodami a proměnnými. Každý nový task je typu `DefaultTask`, stejně jako je v Javě každý nový objekt potomek `java.lang.Object` (2 s. 16). Metody a atributy objektu `DefaultTask` jsou popsány v dalších kapitolách.

5.2.1 Hello World v Gradlu

Vysvětlení pojmu task je nejlepší na příkladu. Stejně jako ve většině příruček, tak i já zde popisuji jeden ze základních příkladů Hello World. Tento příklad je velice jednoduchý, jeho cílem je na obrazovku vypsat `Hello World!`.

Je třeba vytvořit soubor s názvem `build.gradle` s jediným taskem:

```
task helloWorld << {
  println 'Hello World!'
}
```

Task se spustí zadáním příkazu v příkazové řádce:

```
gradle -q helloWorld
```

Tímto příkazem Gradle začne v aktuální složce hledat soubor s názvem `build.gradle` a z něj provede pouze task s názvem `helloWorld`. Tento task obsahuje pouhé vypsání textu `Hello World!` na konzoli, jelikož `println` je Groovy metoda, která je to samé jako metoda `System.out.println` v Javě. Použitím příkazu `-q` se zabrání Gradlu, aby vypisoval logovací zprávy, s výjimkou chybových zpráv (12 s. 12).

Task lze z příkazové řádky zavolat i pomocí zkratky, což znamená, že se nemusí napsat celý název tasku, ale pouze jeho část, která bude v rámci build souboru unikátní, a Gradle tak bude moci jednoznačně spustit požadovaný task. Např. `gradle hello` místo `helloWorld`. Další způsob, jak zkrátit název tasku, je pomocí takzvané velbloudí notace, která je známá při programování. Při použití jako zkratky v Gradlu je tím ale myšleno to, že se napíšou jen první písmena z názvu tasku, např. `hW` pro spuštění tasku `helloWorld` (12 s. 15).

5.2.2 Metody objektu DefaultTask

Jak již bylo napsáno, `DefaultTask` je předek všech tasků, stejně jako předek všech objektů v Javě je `Object`. `DefaultTask` definuje několik metod, já dále popisuji ty nejvyužívanější.

dependsOn(task)

Pomocí metody `dependsOn` se definuje závislost mezi tasky. Task, na kterém je daný task závislý, proběhne vždy před daným taskem. Task může být závislý i na více tascích (2 s. 17).

Další příklad, výpis 8, demonstruje závislost tasků. První task `hello` definuje vypsání `Hello` na konzoli. Druhý task `world`, který je závislý na prvním tasku, definuje vypsání `World!`. Po spuštění příkazem `gradle -q world` se provede nejprve task `hello`, který vypíše `Hello`, a poté task `world`, který vypíše `World!`. Výsledkem je tedy opět vypsání `Hello World!` do konzole.

Výpis 8: Ukázka závislosti tasků

```
task hello <<{
  print 'Hello '
}
task world(dependsOn: hello){
  println 'World!'
}
```

doFirst(closure)

Metoda `doFirst` přidá blok spustitelného kódu (akci) na začátek tasku. Closure obsahuje kód ke spuštění před taskem. Closure je blok Groovy kódu uvnitř složených závorek. Může se hodit, pokud je task definovaný na jiném místě, kde se nemůže snadno upravit. Přidáním `doFirst` do jiného buildu se při inicializaci doplní o blok `doFirst` původní task (2 s. 18–20).

```
task helloWorld <<{
  println 'World!'
}

helloWorld.doFirst{
  print 'Hello '
}
```

doLast(closure)

Metoda `doLast` je velmi podobná metodě `doFirst`, ale s tím rozdílem, že se přidá blok spustitelného kódu (akci) na konec tasku. Aliasem pro tuto operaci je `<<`, který je uveden i ve výpisu 8 výše (2 s. 20–21).

onlyIf(closure)

Metoda `onlyIf` umožňuje vyjádřit podmínku. Pokud podmínka není splněna, není task proveden. Podmínka je dána v closure, tudíž zde může být jakýkoli Groovy kód (2 s. 21).

5.2.3 Proměnné objektu DefaultTask

Každý task obsahuje vlastní lokální proměnné. Zde uvádím ty nejpoužívanější, zděděné z objektu `DefaultTask` (2 s. 22–26):

- `didWork` - logická proměnná, která indikuje, zda byl task dokončen úspěšně.
- `enabled` - logická proměnná, která určuje, zda bude task proveden. Defaultně je nastavena na `true`. U jakéhokoli tasku je možné nastavit tuto proměnnou na `false`, což způsobí, že task nebude proveden. Pokud je ale task závislý na jiných tascích a tyto tasky mají nastaveno `enabled = true`, závislé tasky se provedou.
- `path` - proměnná typu `String`, která obsahuje plnou cestu k tasku. Defaultně je cesta k tasku jednoduše jméno tasku, před kterým je dvojtečka, např. `:hello`.
- `logger` - reference na vnitřní Gradle logger, který implementuje `org.slf4j.Logger` (SLF4J = Simple Logging Facade for Java) a přidává k němu několik logovacích úrovní navíc. Přehled úrovní logování je vidět v tabulce 3.
- `description` - metadata tasku. Jsou vidět ve výpisu příkazové řádky po zadání příkazu `gradle tasks --all`, který vypíše všechny tasky v buildu.

```
task helloWorld(description: 'Says hello to the world')
```

- `overwrite` - pokud se task překrývá, je třeba nastavit proměnnou `overwrite` na `true`, jinak bude vyhozena výjimka, protože metoda s tímto názvem již existuje (13 s. 83).

- `temporaryDir - File` objekt, který odkazuje na umístění dočasné složky pro build, která je k dispozici pro odkládání dočasných souborů, jež vzniknou během provádění tasku.

Tabulka 3: *Úrovně logování (12 s. 75)*

Úroveň	Použito pro
DEBUG	debug zprávy.
INFO	informační zprávy.
LIFECYCLE	informační zprávy o postupu, defaultní nastavení.
WARNING	varovné zprávy.
QUIET	důležité zprávy.
ERROR	chybové zprávy.

5.2.4 Typy tasků

Kromě objektu `DefaultTask` existují také tasky pro kopírování, archivaci, spuštění programů a mnoho dalších. Je možné vytvářet i vlastní typ tasku. Typ tasku se definuje následujícím způsobem: `task copyFiles(type: Copy).`

Kopírovací task - type: Copy

Kopírovací task kopíruje soubor(y) z jednoho místa na druhé. Pokud je jako zdroj zadána složka, jsou zkopírovány všechny soubory ze zdrojové složky do složky cílové, ale sama složka zkopírovaná není. Je možné zadat omezení, které typy souborů zkopírovat ze zadané složky a které naopak nekopírovat. Pokud cílová složka neexistuje, je vytvořena (12 s. 64). Nejpoužívanější metody tohoto tasku zobrazuje tabulka 4.

Tabulka 4: *Metody kopírovacího tasku (13 s. 95–96)*

Metoda	Popis
from	Parametrem je `String` nebo `java.io.File`, `URL`, `URI`, který udává zdroj ke zkopírování.
into	Parametrem je `String` nebo `java.io.File`, `URL`, `URI`, který udává cíl, kam je zdroj zkopírován.
include	Parametrem je pattern (vzor), který udává, které soubory kopírovat.
exclude	Parametrem je pattern, který udává, které soubory nekopírovat.

Pattern pro metody `include` a `exclude` může obsahovat následující znaky (12 s. 62):

- * - zastupuje jakýkoli počet znaků
- ? - zástupce pro jakýkoli jeden znak
- ** - všechny složky nebo soubory

Pattern pro všechny html soubory by byl `*.html`.

Task Jar - type: Jar

Task Jar vytvoří jar soubor ze zdrojových souborů. Tento typ tasku přináší plugin Java. Do výsledného jaru, který je umístěný do adresáře `build/libs` a pojmenovaný defaultně podle buildu, jsou zabaleny soubory z `main` a `resources` složky. Task Jar dědí z kopírovacího tasku. Obdobně existují i další typy tasků, které vytváří zip, tar, war nebo ear archiv. Všechny tyto typy tasků také dědí z kopírovacího tasku. Jméno archivu je složeno dle následujícího vzorce:

```
[baseName]-[appendix]-[version]-[classifier].[extension]
```

Pokud některý atribut není zadán, je z výsledného jména vypuštěn. Archiv lze nazvat vlastním jménem pomocí atributu `archiveName` (2 s. 27–28) (12 s. 68).

Task JavaExec - type: JavaExec

Tento task spouští Java třídu s `main` metodou (2 s. 28).

Task Javadoc - type: Javadoc

Task Javadoc generuje dokumentaci pro zdrojové třídy v `main` složce do složky `build/docs/javadoc`. Tento task je přidán pluginem Java, který bude popsán dále. Je možno vygenerovat javadoc dokumentaci i pro třídy umístěné jinde upravením tasku Javadoc nebo přidáním nového (12 s. 104).

5.3 Proměnné

V Gradle existují dva typy proměnných – lokální a globální proměnné. Lokální proměnné definuje uživatel přímo ve skriptovacím souboru. Jejich platnost je omezena tím, kde jsou definovány (13 s. 69). Ukázka deklarace a využití lokální proměnné je demonstrována ve výpisu 9.

Výpis 9: Využití lokální proměnné

```
def hello = 'Hello World!'

task helloWorld << {
  println hello
}
```

Druhým typem proměnných jsou takzvané extra proměnné neboli globální proměnné, které se definují v `ext` bloku. Mají širší pole působnosti, na rozdíl od lokálních proměnných jsou vidět i v subprojektech (13 s. 70). Výpis 10 demonstruje definici a využití extra proměnné.

Výpis 10: Využití globální proměnné

```
ext {
  applicationVersion = '1.0-SNAPSHOT'
}

task printVersion << {
  println applicationVersion
}
```

Atributy lze definovat i v externím souboru. Tento textový soubor musí být pojmenován `gradle.properties` a obsahuje název atributu a jeho hodnotu na jednom řádku, jak ukazuje výpis 11. Výpis 12 ukazuje využití proměnných v `build.gradle`. Tento soubor může být umístěn ve složce projektu nebo v `GRADLE_HOME` složce. Atributy ve složce `GRADLE_HOME` přepisují atributy definované v jednotlivých složkách projektu. Proměnné jsou dostupné přes objekt projektu, který je popsán v následující kapitole (3 s. 74).

Výpis 11: Ukázka definice atributů v externím souboru (12 s. 74)

```
version = 4.0
customProperty = Property value from gradle.properties
```

Výpis 12: Ukázka využití proměnných z externího souboru (12 s. 74)

```
task showProperties {
  doLast {
    println "Version: $version"
    println "Custom property: $customProperty"
  }
}
```

5.4 Projekt

Projekt je stejně jako task objekt. V tabulce 5 uvádím přehled nejpoužívanějších atributů projektu a jejich popis.

Tabulka 5: *Atributy projektu (12 s. 70)*

Jméno	Typ	Defaultní hodnota
project	Project	Instance projektu.
name	String	Jméno složky projektu. Pouze pro čtení.
path	String	Absolutní cesta k projektu.
description	String	Popis projektu.
projectDir	File	Složka obsahující skript buildu. Pouze pro čtení.
buildDir	File	Složka se jménem buildu ve složce obsahující skript buildu.
group	Object	Nespecifikováno.
version	Object	Nespecifikováno.

Přes proměnnou `project` lze přistupovat i k taskům skrze jejich jméno, např. `project.hello`, nebo lze k tasku přistoupit pouze přes jeho jméno, jak je vidět na příkladu níže. Dále lze přistupovat i k proměnným tasku, např. `project.hello.name` (13 s. 31).

```
task simple << {
  println "Hello ${message}"
}
simple.message = 'world'
```

5.5 Práce se soubory a adresáři

Pro práci se soubory poskytuje objekt `Project` několik metod. Já zde představím dvě základní metody, které se velice často využívají.

metoda file

Metoda `file` je relativní k aktuálnímu projektu, ne k aktuální složce, protože task se může spustit odkudkoli. Parametrem je `String` nebo `File, URL, URI`, druhým parametrem může být `PathValidation`. V případě, že kontrola vrátí výsledek `false,` je build stopnut (13 s. 90).

```
File configFile = file('src/config.xml')
```

metoda files

Pro práci s kolekcí souborů bere metoda `files` jako argument seznam hodnot typu `String` nebo seznam `File`, lze i kombinovat různé typy parametrů. Lze využít např. i výstup tasku. Kolekce souborů je iterovatelná, což znázorňuje následující příklad:

```
FileCollection collection = files('src/file1.txt', new File('src/file2.txt'))

collection.each {File file ->
  println file.name
}
```

5.6 Build

Gradle má podporu pro inkrementální build. To znamená, že task je v buildu proveden, pouze pokud je to nezbytné. Například je-li nutné sestavit zdrojový kód, Gradle zkontroluje, zda se zdroje od posledního buildu změnily. Pokud se nezměnily, je task označen jako aktuální (angl. UP-TO-DATE) a je přeskočen, tzn. neprovede se. Pokud se změnily, je task proveden. Tento mechanismus je podporován u většiny Gradle tasků, ale je možné ho použít i ve vlastních tascích (12 s. 8).

Gradle opouští striktně definovaný životní cyklus od Mavenu (Ant neměl žádný životní cyklus) a nahrazuje ho defaultní množinou tasků poskytovaných pluginy, např. Java (2 s. 48).

Pokaždé když Gradle provádí build, spouští se tři životní fáze buildu: inicializace, konfigurace a provedení. Provedení je fáze, ve které se tasky provádějí v pořadí daném jejich vzájemnými vztahy. Konfigurace je fáze, ve které jsou objekty tasků sestaveny do interního objektového modelu. Inicializace je fáze, ve které se Gradle rozhodne, které projekty se budou podílet na buildu. To je důležité u multiprojektových buildů (2 s. 16).

Častou chybou je, že se při definování tasku zapomene na `<<`. Poté se z closure stává konfigurační closure a neprovádí se akce. Pokud bychom chtěli provést akci, je třeba ji definovat pomocí `doFirst` nebo `doLast` (12 s. 45).

V Gradlu nezáleží na pořadí tasků a jejich definici. V jakém pořadí se tasky spustí, se vyhodnocuje při konfiguraci, a ne až při provádění. Každý task je proveden pouze jednou. Pokud nastane při provádění tasku chyba, je build zastaven. Toto chování lze vypnout zadáním `--continue` do příkazové řádky. Vyloučit task z provádění lze v příkazové řádce pomocí `-x` (exclude) příkazu. Dalším způsobem, jak task vypnout, je nastavením jeho proměnné `enabled` na `false`. Spuštění tasku může být podmíněné pomocí metody s podmínkou `onlyIf`. Přeskočit task lze i pomocí vyhození výjimky `StopExecutionException`. Tím pádem je task přeskočen a pokračuje se dalším taskem (13 s. 55–56).

5.7 Plugin

V Gradlu lze snadno aplikovat na projekt pluginy. Plugin je zde speciální funkcionalita, která do projektu přidá tasky a proměnné, např. plugin Java přidává tasky pro kompilaci, testování a zabalení zdrojového kódu. Pluginy jsou součástí verze Gradlu, tudíž při jejich aktualizaci nejde stáhnout pouze aktualizaci pluginu, ale je nutno stáhnout celou novou verzi Gradlu. V budoucnu se toto má změnit. Plugin lze napsat i vlastní, který je potom nezávislý na verzi Gradle (12 s. 85).

V Mavenu plugin znamená rozšíření pomocí akce, která je napojená na jednu nebo více fází životního cyklu. V Gradlu může plugin poskytovat konfigurovatelné akce buildu zavedením jednoho nebo více tasků. V podstatě je to ale rozšíření DSL (2 s. 3).

5.7.1 Plugin Java

Plugin Java je založen na konvencích, což znamená, že je již předkonfigurovaný, a pokud se kód řídí těmito konvencemi, není nutné build dále nastavovat, stačí přidat plugin Java

příkazem `apply plugin: 'java'`. Plugin se do kódu přidá velice jednoduše pomocí příkazu `apply plugin:'<jméno_pluginu>'` (13 s. 37).

Plugin Java zavádí do projektu nový koncept, a to `SourceSet`. `SourceSet` je kolekce zdrojových souborů, které jsou kompilovány a prováděny společně. Mohou to být zdrojové soubory Javy nebo zdrojové soubory (např. soubory s příponou `properties`, obrázky atd.). `SourceSet` může být použit k seskupení souborů s určitým významem v projektu, aniž by bylo nutné vytvářet samostatný projekt. Plugin Java dodává bez jakékoli konfigurace `SourceSet` `main` a `test` (13 s. 130). Tyto sourceSety lze konfigurovat, což demonstruje výpis 13, nebo lze definovat vlastní `SourceSet`.

Výpis 13: Konfigurace zdrojových složek sourceSetu (13 s. 134)

```
sourceSets {
  main {
    java {
      srcDir 'src/java'
  }
    resources {
      srcDir 'src/resources'
    }
  }
}
```

Výpis 14 ukazuje, jaké metody jsou dostupné před použitím pluginu Java. Výpis 15 ukazuje metody, které jsou dostupné po aplikování pluginu Java.

Výpis 14: Výpis příkazu `gradle tasks` před použitím pluginu Java

```
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------
Help tasks
----------
dependencies - Displays all dependencies declared in root project 'example'.
dependencyInsight - Displays the insight into a specific dependency in root pro-
ject 'example'.
help - Displays a help message.
projects - Displays the sub-projects of root project 'example'.
properties - Displays the properties of root project 'example'.
tasks - Displays the tasks runnable from root project 'example' (some of the
displayed tasks may belong to subprojects).

To see all tasks and more detail, run with --all.
```

Výpis 15: Výpis příkazu `gradle tasks`, pokud je aplikován plugin Java

```
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------
Build tasks
-----------
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that
dependon it.
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles the main classes.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles the test classes.

Documentation tasks
-------------------
javadoc - Generates Javadoc API documentation for the main source code.

Help tasks
----------
dependencies - Displays all dependencies declared in root project 'example'.
dependencyInsight - Displays the insight into a specific dependency in root pro-
ject 'example'.
help - Displays a help message.
projects - Displays the sub-projects of root project 'example'.
properties - Displays the properties of root project 'example'.
tasks - Displays the tasks runnable from root project 'example' (some of the
displayed tasks may belong to subprojects).

Verification tasks
------------------
check - Runs all checks.
test - Runs the unit tests.

Rules
-----
Pattern: build<ConfigurationName>: Assembles the artifacts of a configuration.
Pattern: upload<ConfigurationName>: Assembles and uploads the artifacts belong-
ing to a configuration.
Pattern: clean<TaskName>: Cleans the output files of a task.

To see all tasks and more detail, run with --all.
```

5.8 Gradle wrapper

Gradle wrapper umožňuje provádět Gradle buildy, i když Gradle není na počítači nainstalován. Díky wrapperu také můžeme vynutit použití určité verze Gradlu pro celý tým. Ukázka použití je ve výpisu 16.

Výpis 16: Použití Gradle wrapperu (12 s. 83)

```
task createGradleWrapper(type: Wrapper) {
  gradleVersion = '1.3'
}
```

Nejprve je nutno spustit task `createGradleWrapper`, který stáhne danou verzi Gradlu do složky `gradle` a vytvoří dávkový soubor `gradlew.bat` pro spouštění příkazů v dané verzi Gradlu. Pomocí tohoto dávkového souboru je nutno spouštět všechny příkazy, takže se např. místo příkazu `gradle tasks` zadá příkaz `gradlew tasks`.

6 Řízení závislostí

Při psaní aplikace je většinou potřeba využít open-source knihovny nebo knihovny třetích stran. Tyto knihovny jsou označovány jako závislosti (angl. dependencies) a jsou uloženy většinou v tzv. úložištích (angl. repositories). Při buildu musí být dostupné na classpath překladače, jinak build zhavaruje. Buildovací nástroje podporují a řídí tyto závislosti. V této kapitole je popsáno, jakým způsobem funguje řízení závislostí v Mavenu i Gradlu.

Java sama o sobě nemá podporu pro práci se závislostmi. Nejpopulárnější nástroje pro řízení závislostí jsou Maven a Apache Ivy (dále jen Ivy). Ivy je nástroj pouze pro řízení závislostí. Oba nástroje podporují úložiště, kde jsou uloženy verzované knihovny spolu s metadaty o těchto knihovnách. Knihovna může být závislá na jiných knihovnách, což je popsáno právě v metadatech. Metadata jsou zapsána v XML popisovači. Ivy podporuje Maven popisovače, ale přidává k nim i další údaje. Gradle využívá pro řízení závislostí Ivy (12 s. 109).

6.1 Řízení závislostí v systému Maven

6.1.1 Úložiště v systému Maven

„Úložiště je kolekce souborů organizovaných podle skupiny, názvu a verze“ (13 s. 46).

„Úložiště je sbírka artefaktů projektu uložených v adresářové struktuře, která přesně odpovídá Maven souřadnicím.“ Artefakt je tedy v úložišti jednoznačně identifikován atributy `groupId`, `artifactId` a `version` (11 s. 30).

Existují dva typy úložišť, a to lokální a vzdálené úložiště. Lokální úložiště uchovává kopii artefaktu staženou ze vzdáleného úložiště a dočasné artefakty, které ještě nebyly publikovány na vzdálené úložiště. Toto úložiště se nachází v adresářové struktuře počítače ve složce `{user_home}\.m2\repository`. Druhým typem úložišť jsou tzv. vzdálená (angl. remote) úložiště, která mohou být dostupná pouze pro firmu nebo mohou být veřejná. Implicitně Maven využívá tzv. Maven Central Repository, které je nastavené v `SuperPOM` a nachází se na adrese: http://repo1.maven.org/maven2 (25). Výpis 17 demonstruje definici úložiště v POM.

Informace o manažerech vzdálených úložišť, jejich výhodách a nevýhodách, instalaci a konfiguraci manažera Nexus od firmy Sonatype lze získat v bakalářské práci Miroslava Králika *Správa úložísk nástroja Maven* (26).

Maven nejprve zkouší najít závislost v lokálním úložišti, teprve v případě neúspěchu se ji pokusí stáhnout ze vzdáleného úložiště. Nejdříve se Maven pokusí knihovnu stáhnout

z centrálního Maven úložiště a potom z dalších úložišť, která jsou definována v POM nebo `settings.xml`.

Výpis 17: Definice centrálního úložiště v `SuperPOM` (1 s. 21-22)

```
<project>
  ...
  <repositories>
    <repository>
      <id>central</id>
      <name>Maven Repository Switchboard</name>
      <layout>default</layout>
      <url>http://repo1.maven.org/maven2</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
</project>
```

Závislost lze do lokálního úložiště nainstalovat i ručně. Tento způsob se využívá v situaci, kdy daná závislost není dostupná na vzdáleném úložišti. Pro minimalizování chyb během ruční instalace se doporučuje využít cíl `install-file` pluginu `install` podle následujícího schématu (27):

```
mvn install:install-file -Dfile=<cesta k závislosti> -DgroupId=<groupId> \
    -DartifactId=<artifactId> -Dversion=<verze> -Dpackaging=<typ balíčku>
```

6.1.2 Závislosti v systému Maven

Maven podporuje od verze Maven2 tranzitivní (angl. transitive) závislosti. To znamená, že pokud je dána závislost, která je závislá na dalších knihovnách, není třeba tyto knihovny ručně definovat do závislostí, ale Maven sám tyto závislosti najde a stáhne (11 s. 31).

Závislosti v Mavenu mají 6 rozsahů (angl. scope), které specifikují míru a okamžik potřeby dané závislosti (1 s. 30–31):

- `Compile` - výchozí rozsah, závislosti jsou dostupné na classpath a jsou zabaleny do archivu, nutné pro kompilaci i běh
- `Provided` - závislosti pro kompilaci i běh, ale knihovna je poskytovaná JDK nebo kontejnerem, a proto není zabalená do archivu, např. Servlet API, závislost není tranzitivní
- `Runtime` - závislosti potřebné při běhu a testech, které ale nejsou nutné pro kompilaci
- `Test` - závislosti pro kompilaci a běh testů

- `System` - podobné `provided`, ale je nutno zajistit cestu ke knihovně na lokálním systému souborů, kde musí být knihovna vždy dostupná, tento rozsah není doporučený
- `Import` - závislosti uvedené v POM prostřednictvím `<dependencyManagement>` elementů, které nahrazují závislosti v POM (28)

Verzi lze určit rozsahem s pomocí kulatých nebo hranatých závorek, tak jako se využívají v matematice. Definice verze [3.8,4.0) tedy znamená, že se má stáhnout závislost od verze 3.8 včetně, ale menší než 4.0. [4.0,) znamená, že požadovaná verze závislosti má být větší nebo rovna verzi 4.0. (,2.0) znamená, že požadovaná verze má být menší než 2.0. (1 s. 33).

Maven používá graf závislostí, s jehož pomocí předchází konfliktům. Pokud je dvakrát vyžadována závislost se stejným `groupId` a `artifactId`, Maven dá vždy přednost novější verzi (1 s. 34).

Definici závislosti demonstruje výpis 18.

Výpis 18: Definice závislosti v POM *(10 s. 32)*

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>...</groupId>
      <artifactId>...</artifactId>
      <version>...</version>
      <scope>...</scope>
    </dependency>
  </dependencies>
</project>
```

Závislosti nadefinované v rodičovském POM jsou dědičné, tudíž jsou i v subprojektech. Pokud je třeba pro subprojekty nadefinovat verzi nějaké knihovně, definuje se tato závislost v sekci `dependencyManagement`, jak ukazuje výpis 19. Díky tomu se v POM potomka nemusí uvádět verze a nevznikne konflikt verzí, což znázorňuje výpis 20.

Výpis 19: Definice závislosti v rodičovském POM (10 s. 33)

```
<project>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.2</version>
      </dependency>
    <dependencies>
  </dependencyManagement>
</project>
```

Výpis 20: Definice závislosti v POM potomka (10 s. 33)

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

6.1.3 Publikování artefaktů v systému Maven

Pro publikování artefaktů slouží plugin Deploy, který ve fázi `deploy` nahraje distribuci projektu na vzdálené úložiště, jež je definováno v sekci `distributionManagement`, jak ukazuje výpis 21.

Výpis 21: Definice úložiště v sekci `distributionManagement`

```
<project>
  ...
  <distributionManagement>
    <repository>
      <id>...</id>
      <name>... </name>
      <url>...</url>
    </repository>
  </distributionManagement>
</project>
```

Pro definici úložiště pro snapshoty slouží sekce `snapshotRepository`, jejíž struktura je vidět ve výpisu 22. Pokud tato sekce není definována, jsou snapshoty nahrány do úložiště v sekci `repository`.

Výpis 22: Definice `snapshotRepository` v sekci `distributionManagement`

```
<project>
  ...
  <distributionManagement>
    <snapshotRepository>
      <id>...</id>
      <name>... </name>
      <url>...</url>
    </snapshotRepository>
  </distributionManagement>
</project>
```

6.2 Řízení závislostí v systému Gradle

6.2.1 Úložiště v systému Gradle

Gradle implicitně nedefinuje žádné úložiště. Předtím než se začnou využívat závislosti, je tedy třeba minimálně jedno úložiště nadefinovat (13 s. 46). V Gradle build souboru lze definovat několik typů úložišť. Gradle poskytuje několik předkonfigurovaných úložišť, které zobrazuje tabulka 6, ale je velmi jednoduché použít vlastní Maven nebo Ivy úložiště.

Tabulka 6: *Přehled předkonfigurovaných úložišť (12 s. 113)*

Typ úložiště	Popis
Maven úložiště	Maven úložiště na vzdáleném PC nebo souborovém systému.
Maven centrální úložiště	Předkonfigurované Maven úložiště k vyhledávání závislostí.
Maven místní úložiště	Předkonfigurované Maven úložiště, které najde závislosti v lokálním Maven úložišti.
Ivy úložiště	Ivy úložiště, které může být na vzdáleném nebo místním PC.
Flat adresář úložiště	Jednoduché úložiště na lokálním systému souborů v počítači nebo nasdílené přes síť.

Většina open-sourcesových projektů umísťuje svoje projekty na centrální Maven úložiště. Přidání tohoto úložiště v Gradlu je velice jednoduché, jak demonstruje následující příklad (12 s. 113–114):

```
repositories {
  mavenCentral()
}
```

Pokud je na počítači používán i Maven, lze využít jeho lokální úložiště pomocí definice v následujícím příkladu (12 s. 114):

```
repositories {
  mavenLocal()
}
```

Pokud je potřeba využít vlastní Maven úložiště, např. v síti intranet, lze to provést následujícími dvěma způsoby (12 s. 114):

```
repositories {
  maven {
    // Name není povinné
    name = 'Main Maven repository'
    url = 'http://intranet/repo'
  }
  mavenRepo(name: 'Snapshot repository', url: 'http://intranet/snapshots')
}
```

Gradle hledá závislost v úložištích podle pořadí, ve kterém jsou úložiště definována (13 s. 47).

6.2.2 Závislosti v systému Gradle

V Gradle build souboru se definují závislosti v konfiguračním tasku. Plugin Java generuje 6 fází (konfigurací), ve kterých může být závislost dostupná. Jejich přehled zobrazuje tabulka 7.

Tabulka 7: *Fáze závislostí (12 s. 112)*

Konfigurace	Dědí z	Použita v tasku	Popis
compile	-	compileJava	Závislosti potřebné ke kompilaci v době překladu.
runtime	compile	-	Závislosti pro běh aplikace, které ale nejsou potřebné při kompilaci.
testCompile	compile	compileTestJava	Závislosti potřebné ke kompilaci testovacích zdrojových kódů.
testRuntime	testCompile	test	Závislosti potřebné ke spuštění testů.
archives	-	uploadArchives	Obsahuje artefakty, které jsou vytvořené v rámci projektu, např. JAR.
default	runtime	-	Výchozí konfigurace, obsahuje všechny runtime závislosti.

Gradle definuje závislosti jednoduše na jednom řádku jako hodnoty atributů `group`, `name` a `version` oddělené dvojtečkou. Lze využít i více upovídanou verzi, kde se vyplní atribut a jeho hodnota. Obě možnosti demonstruje následující příklad:

```
dependencies {
  compile group: 'org.springframework', name: 'spring-core',
                 version: '3.1.1.RELEASE'
  runtime 'org.springframework:spring-aop:3.1.1.RELEASE'
}
```

V Gradlu je možné definovat i dynamickou verzi závislosti, což znamená, že lze definovat např. rozmezí verze závislosti nebo stanovit minimální či naopak maximální verzi. Asi nejčastějším příkladem bude využití minimální verze, jak je vidět na příkladu níže, přičemž Gradle vždy stáhne minimálně verzi 3.1, pokud není dostupná novější vydaná verze větší než 3.1. Není tedy třeba se zajímat o vydání nových verzí závislostí (12 s. 127).

```
dependencies {
  compile group: 'org.springframework', name: 'spring-core', version: '3.1.+'
}
```

Rozmezí verzí lze zadat v hranatých závorkách stejně jako v Mavenu, místo kulaté závorky se ale využívá obrácená hranatá závorka. Pokud chci závislost v rozmezí větší než 1.0, ale menší než 2.0, je zápis následující (12 s. 127):

```
]1.0, 2.0[
```

Je třeba si však uvědomit riziko, které může nová verze pro projekt přinést. Je lepší vždy novou verzi vyzkoušet a otestovat, zda do aplikace nevnese chyby.

Gradle podporuje tranzitivní závislosti. Pokud používáme mnoho závislostí a tyto závislosti mají tranzitivní závislosti, může snadno nastat konflikt. Jedna závislost bude vyžadovat verzi závislosti A 1.0, kdežto druhá závislost bude vyžadovat tutéž závislost A ve verzi 2.0. Gradle defaultně použije novější verzi. Toto implicitní chování lze změnit (12 s. 128).

6.2.3 Publikování artefaktů v systému Gradle

Gradle implicitně nedefinuje žádné úložiště pro publikování artefaktů, a je proto nutné nějaké úložiště nadefinovat. Výpis 23 demonstruje definici úložiště pro publikování artefaktů i snapshotů.

Výpis 23: Ukázka definice Maven úložiště pro publikování artefaktů (13 s. 48)

```
apply plugin: 'maven'

uploadArchives {
  repositories {
    mavenDeployer {
      repository(url: "file://localhost/tmp/myRepo/")
      snapshotRepository(url: "file://localhost/tmp/myRepoSnapshots/")
    }
  }
}
```

7 Testování

Testy jsou neodmyslitelnou součástí projektu. Jejich spuštění je součástí životního cyklu projektu. V této kapitole popisuji testování s pomocí Mavenu i Gradlu.

7.1 Testování se systémem Maven

Spuštění JUnit nebo TestNG testů probíhá ve fázi `test`. Nutností je nastavení závislosti v POM na knihovnu JUnit nebo TestNG. Maven vyhledá a spustí všechny třídy, jejichž název odpovídá vzoru `**/Test*.java`, `**/Test.java` a `**/*TestCase.java`. Ze spuštění testů jsou vyloučeny třídy odpovídající vzoru `**/Abstract*Test.java` a `**/Abstract*TestCase.java`. Je možné přidat i další vzory (10 s. 40).

Příkazem `mvn test` se spustí testy. Pokud některý z testů neskončí úspěšně, Maven automaticky stopne build. Toto defaultní chování lze změnit v POM nastavením ignorování chyb, jak demonstruje výpis 24, nebo při spuštění v příkazové řádce nastavením atributu `maven.test.failure.ignore` (11 s. 56–57):

```
mvn test -Dmaven.test.failure.ignore=true
```

Výpis 24: Nastavení ignorování chyb testů (11 s. 56)

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <testFailureIgnore>true</testFailureIgnore>
  </configuration>
</plugin>
```

Další možností je přeskakovat testy při buildu úplně. To lze pomocí příkazové řádky `mvn install -Dmaven.test.skip=true` nebo opět konfigurací v POM, jak ukazuje výpis 25 (11 s. 57).

Výpis 25: Nastavení přeskakování testů (11 s. 57)

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <skip>true</skip>
  </configuration>
</plugin>
```

Výsledný XML a textový report, který popisuje průběh testů, se po dokončení testů uloží do složky `target/surefire-reports` (11 s. 56). Obrázek 2 zobrazuje výpis testů testovací třídy `ContextTest`. Na výpisu si lze všimnout počtu testů, počtu výjimek, počtu chyb, počtu přeskočených testů a celkový čas testů.

```
-------------------------------------------------------------------------------
Test set: cz.example.ContextTest
-------------------------------------------------------------------------------
Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 5.005 sec
```

Obrázek 2:
Textový report testu v Mavenu

Od verze JUnit 4.7. lze pro testy nastavit paralelní provádění testovacích metod, testovacích tříd nebo obojí. V konfiguraci pluginu Surefire je možné v elementu `parallel` nastavit hodnotu `methods` pro metody, `classes` pro třídy a `both` pro oboje. V sekci konfigurace lze dále nastavit element `threadCount`, který udává počet vláken. Ukázka konfigurace (29) (30):

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <parallel>methods</parallel>
    <threadCount>10</threadCount>
  </configuration>
</plugin>
```

7.2 Testování se systémem Gradle

Gradle podporuje testy pomocí frameworků JUnit a TestNG. Nutností je přidat závislost na daný framework a aplikovat plugin Java. Testy se spustí pomocí příkazu `gradle test`. Gradle zkoumá zkompilované třídy, tedy soubory s příponou `class`. Gradle spustí testy, pokud má třída nebo metoda anotaci `@Test`. Dále Gradle spustí testy, pokud třída dědí

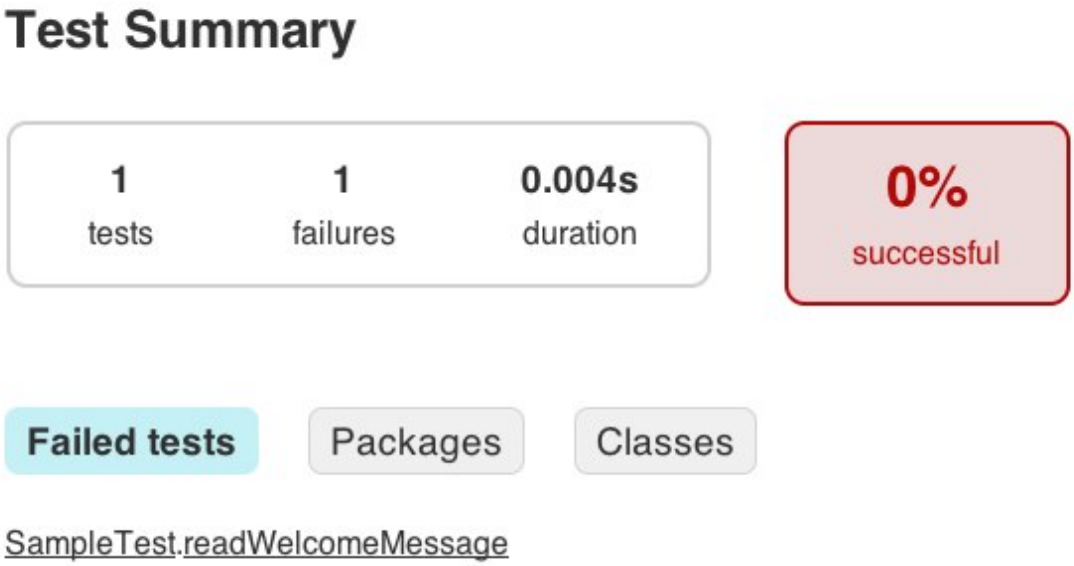
z TestCase nebo GroovyTestCase nebo má anotaci @RunWith. Abstraktní třídy nejsou zkoumány. Toto skenování tříd lze vypnout pomocí nastavení atributu scanForTestClasses na false. Poté se zapne implicitní pravidlo, které do testů zahrne třídy odpovídající vzoru **/*Tests.class a **/*Test.class a vyloučí třídy odpovídající vzoru **/Abstract*.class. Lze přidat vlastní pravidla pro zahrnutí nebo vyloučení tříd, jak ukazuje výpis 26 (12 s. 146).

Výpis 26: Definice vlastních pravidel pro testy (12 s. 146–147)

```
apply plugin: 'java'

test {
  // Vypne automatické skenování
  scanForTestClasses = false

  // Pravidla pro zahrnutí tříd
  include '**/*Test.class', '**/*Spec.class'

  // Pravidla pro vyloučení tříd
  exclude '**/Abstract*.class', '**/Run*.class'
}
```

Pokud některý z testů během buildu neprojde, stopne se celý build. Toto implicitní chování lze vypnout nastavením atributu ignoreFailures na true (12 s. 144). Přeskočení provádění tasků je možné nastavit několika způsoby, které jsem uvedla v podkapitole Build v kapitole Základy systému Gradle. Je nutné nastavit přeskočení tasku test nebo si vytvořit vlastní proměnnou a tu využít v metodě onlyIf, která rozhodne, zda testy spustit.

Výsledný HTML report z testů je umístěn ve složce build/reports/test (12 s. 137). Na obrázku 3 je vidět výsledný report testů, kde lze vyčíst celkový počet testů, celkový počet chyb, čas trvání testů, úspěšnost testů v procentech a výpis testů s chybou.



Obrázek 3:
Shrnutí výsledků testů v Gradlu (12 s. 137)

Obrázek 4 zobrazuje detail jedné neúspěšné testovací třídy. Opět je z reportu vidět počet testů v testovací třídě, počet chybných testů, doba trvání testů ve třídě, úspěšnost testů v procentech a výpis chyby, která v testu nastala.

Class gradle.sample.SampleTest

all > gradle.sample > SampleTest

1	1	0.004s
tests	failures	duration

0% successful

Failed tests | Tests

readWelcomeMessage

```
org.junit.ComparisonFailure: Get text from properties file expected:<Welcome to Gradle[.]> but was:<Welcome to Gradle[!]>
        at org.junit.Assert.assertEquals(Assert.java:125)
        at gradle.sample.SampleTest.readWelcomeMessage(SampleTest.java:15)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:45)
        at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:15)
        at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:42)
        at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:20)
        at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:263)
        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:68)
        at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:47)
        at org.junit.runners.ParentRunner$3.run(ParentRunner.java:231)
        at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:60)
        at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:229)
        at org.junit.runners.ParentRunner.access$000(ParentRunner.java:50)
        at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:222)
        at org.junit.runners.ParentRunner.run(ParentRunner.java:300)
        at org.gradle.api.internal.tasks.testing.junit.JUnitTestClassExecuter.runTestClass(JUnitTestClassExecuter.java:55)
        at org.gradle.api.internal.tasks.testing.junit.JUnitTestClassExecuter.execute(JUnitTestClassExecuter.java:42)
        at org.gradle.api.internal.tasks.testing.junit.JUnitTestClassProcessor.processTestClass(JUnitTestClassProcessor.java:75)
        at org.gradle.api.internal.tasks.testing.SuiteTestClassProcessor.processTestClass(SuiteTestClassProcessor.java:49)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at org.gradle.messaging.dispatch.ReflectionDispatch.dispatch(ReflectionDispatch.java:35)
        at org.gradle.messaging.dispatch.ReflectionDispatch.dispatch(ReflectionDispatch.java:24)
        at org.gradle.messaging.dispatch.ContextClassLoaderDispatch.dispatch(ContextClassLoaderDispatch.java:32)
        at org.gradle.messaging.dispatch.ProxyDispatchAdapter$DispatchingInvocationHandler.invoke(ProxyDispatchAdapter.java:93)
        at $Proxy2.processTestClass(Unknown Source)
        at org.gradle.api.internal.tasks.testing.worker.TestWorker.processTestClass(TestWorker.java:101)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at org.gradle.messaging.dispatch.ReflectionDispatch.dispatch(ReflectionDispatch.java:35)
        at org.gradle.messaging.dispatch.ReflectionDispatch.dispatch(ReflectionDispatch.java:24)
        at org.gradle.messaging.remote.internal.TypeCastDispatch.dispatch(TypeCastDispatch.java:30)
        at org.gradle.messaging.remote.internal.WorkerProtocol.handleIncoming(WorkerProtocol.java:53)
        at org.gradle.messaging.remote.internal.WorkerProtocol.handleIncoming(WorkerProtocol.java:31)
        at org.gradle.messaging.remote.internal.ProtocolStack$ProtocolStage.handleIncoming(ProtocolStack.java:167)
        at org.gradle.messaging.remote.internal.ProtocolStack$BottomStage.handleIncoming(ProtocolStack.java:277)
        at org.gradle.messaging.remote.internal.ProtocolStack$BottomConnection$1.run(ProtocolStack.java:299)
        at org.gradle.messaging.remote.internal.ProtocolStack$ExecuteRunnable.dispatch(ProtocolStack.java:120)
        at org.gradle.messaging.remote.internal.ProtocolStack$ExecuteRunnable.dispatch(ProtocolStack.java:116)
        at org.gradle.messaging.dispatch.AsyncDispatch.dispatchMessages(AsyncDispatch.java:132)
        at org.gradle.messaging.dispatch.AsyncDispatch.access$000(AsyncDispatch.java:33)
        at org.gradle.messaging.dispatch.AsyncDispatch$1.run(AsyncDispatch.java:72)
        at org.gradle.messaging.concurrent.DefaultExecutorFactory$StoppableExecutorImpl$1.run(DefaultExecutorFactory.java:66)
        at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
        at java.lang.Thread.run(Thread.java:680)
```

Obrázek 4:

Detail neúspěšného testu v Gradlu (12 s. 138)

Gradle umí provádět testy paralelně, což znamená, že nastartuje více procesů pro test současně. Každý testovací proces provádí jeden test. Zapnutím paralelního provádění testů může celkový počet prováděných testů v daném okamžiku rapidně vzrůst, pokud máme

mnoho testů. Atribut `maxParallelForks` určuje, kolik procesů testu maximálně paralelně spustit. Defaultní hodnota je jedna, takže testy neběží paralelně (12 s. 145).

Atributem `forkEvery` lze nastavit, po kolika spuštěných testech restartovat testovací proces. Je to alternativa k nastavování velké haldy (angl. heap). Defaultní hodnota je neomezená (12 s. 145). Příklad nastavení testů:

```
test {
  forkEvery = 10
  maxParallelForks = 4
}
```

8 Multiprojektový build

Velké aplikace se obvykle rozdělují do menších projektů, které se nazývají subprojekty nebo moduly. Každý modul má svůj životní cyklus a svůj konfigurační soubor. Dá se říci, že každý modul je samostatný projekt, většinou na jeho vývoji pracuje jeden tým. Všechny tyto moduly ale musí být opět sestaveny dohromady, aby byla aplikace kompletní a schopná distribuce (2 s. 79). V této kapitole popisuji fungovaní multiprojektových buildů v systémech Maven i Gradle.

8.1 Multiprojektový build v systému Maven

V Mavenu je rozdíl mezi projektem, který dědí z rodičovského POM, a projektem, který je modulem. Některé projekty jsou kombinací obou přístupů. Pokud projekt dědí z rodičovského POM, přebírá z něj některé hodnoty. V organizacích existuje většinou organizační POM, ze kterého dědí POM projektu namísto toho, aby dědila ze `SuperPOM` (1 s. 45–46). Z rodičovského POM potomci dědí implicitně následující sekce (1 s. 41–42):

- identifikátory - minimálně `groupId` nebo `artifactId` musí být překryto v potomcích
- závislosti
- sekce `<developers>`
- sekce `<contributors>`
- pluginy
- reporty
- sekce `<plugin>` `<executions>` (`<executions>` se stejným id se spojí)
- sekce `<plugin>` `<configuration>`

Dědění POM je velice silná funkcionalita, která by ale neměla být nadměrně užívána. Pokud jsou projekty rozdílné a nepotřebují sdílet hodnoty z rodičovského POM, neměly by být jeho potomky, ale jeho moduly. A to i v případě, že bude nutno mít v obou modulových POM zkopírované části (1 s. 47–48).

Moduly se definují v kořenovém POM v sekci `<modules>`, toto kořenové POM musí mít nastavený atribut `packaging` na typ `pom`. Výpis 27 ukazuje definici kořenového POM (22).

Výpis 27: Definice kořenového POM (22)

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>my.app</groupId>
  <artifactId>my-parent</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <modules>
    <module>my-app</module>
    <module>my-webapp</module>
  </modules>
</project>
```

Potomek se na rodičovské POM odkazuje v sekci `<parent>`. Pro urychlení hledání rodičovského POM k němu lze nastavit relativní cestu v elementu `relativePath` (22). Definici demonstruje výpis 28.

Výpis 28: Definice relativní cesty k rodičovskému POM

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>my.app</groupId>
    <artifactId>my-parent</artifactId>
    <version>1.0</version>
    <relativePath>../my-parent</relativePath>
  </parent>
  <artifactId>my-project</artifactId>
</project>
```

Maven nejprve načte rodičovské POM a najde všechny moduly. Poté dá všechny tyto modulové POM do komponenty nazývané Maven Reactor, která analyzuje závislosti mezi moduly. Reactor se stará o to, aby nezávislé moduly byly kompilovány a instalovány ve správném pořadí (11 s. 77).

Spustit build lze i pro jeden nebo více konkrétních subprojektů pomocí příkazu `-pl <seznam projektů>` nebo `--projects <seznam projektů>`. Tato možnost se hodí na projekt, který má mnoho subprojektů a celý build projektu by trval dlouho. Dále lze využít možnosti spustit build daného subprojektu a projektů, na kterých je daný subprojekt závislý. V tomto případě je opět třeba uvést pomocí příkazu `-pl` projekt, pro který chci spustit build, a dále uvést příkaz `-am` nebo `--also-make`. Další užitečnou možností je spuštění build subprojektů, které jsou na daném subprojektu závislé, pomocí příkazu `-amd` nebo `--also-make-dependents` a určení subprojektu pomocí příkazu `-pl`. Příklad níže demonstruje použití tohoto příkazu.

```
mvn --projects projectC --also-make-dependents install
```

Velice užitečný je cíl `effective-pom` pluginu `help`, jenž do konzole nebo do souboru vypíše výsledné POM, ve kterém uplatní dědičnost atd. (1 s. 40).

```
mvn help:effective-pom
```

8.2 Multiprojektový build v systému Gradle

Pro multiprojektový build je nutno nadefinovat soubor `settings.gradle` v kořenovém adresáři projektu, do kterého se zapíší subprojekty. V inicializační fázi hledá Gradle soubor `settings.gradle`, ve kterém jsou zapsány podadresáře, jež obsahují subprojekty následujícím způsobem (12 s. 80):

```
include 'my-app', 'my-webapp'
```

Veškerá konfigurace multiprojektového buildu může být soustředěna v hlavním build souboru nebo může být rozdělena pro každý subprojekt v jeho vlastním build souboru. Toto rozdělení je doporučováno pro lepší přehlednost a pochopení zejména nováčkům. Třetí přístup je nazýván hybridní. Slučuje společnou konfiguraci do jednoho souboru a individuální konfiguraci do souboru subprojektu. Každý z přístupů má své výhody (12 s. 79–80).

V `gradle.build` souboru lze přistupovat ke všem projektům. Pomocí sekce `allprojects` je možné nastavit chování pro všechny projekty, např. aplikování pluginu Java. V sekci `subprojects` se nastavuje chování pouze pro subprojekty. Na příkladu níže je demonstrována ukázka přidání tasku `printInfo` do všech projektů, kdy se spuštěním tasku z příkazové řádky v kořenovém adresáři projektu nejprve vypíše jméno hlavního projektu a následně jména subprojektů dle abecedy. Gradle totiž spouští tasky implicitně podle abecedy dle jména subprojektu. V případě, že je toto chování nutné ovlivnit, použije se závislost mezi tasky nebo projekty (12 s. 180–184).

```
allprojects {
  //nastavení pro všechny projekty
  task printInfo << {
    println "Toto je projekt ${project.name}"
  }
}
subprojects {
  //nastavení pouze pro subprojekty
}
```

Gradle povoluje přistupovat k jakémukoli projektu z multiprojektového buildu v libovolném `gradle.build` souboru pomocí objektu `Project`, jak ukazuje příklad níže (13 s. 265).

```
project(':my-app') {
  task hello << {
    println "Hello World!"
  }
}
```

Nevýhodou přístupu z jednoho projektu do druhého v konfigurační a prováděcí fázi je tzv. spojování (angl. coupling) projektů, které zabraňuje Gradlu buildovat projekty paralelně. Ke spojování projektů dochází při modifikaci nebo čtení hodnot objektu `Project` z jiného projektu, ale i při používání konfigurací `allprojects` nebo `subprojects` (13 s. 283–284).

Závislost na jiném projektu se definuje v sekci závislostí následujícím způsobem:

```
apply plugin: 'java'
description = 'Projekt B'
  dependencies {
    compile project(':projectA')
}
```

Pokud je projekt závislý na jiném projektu, Gradle nejprve provede build daného projektu.

Není-li stanoveno jinak, Gradle provádí konfiguraci projektů v abecedním pořadí podle jejich názvu. Pokud je v projektu A využita proměnná, která je definována v projektu B, je nutno zajistit provedení konfigurace projektu B před konfigurací projektu A, aby se daná proměnná nastavila. Toto chování je zaručeno nastavením `evaluationDependsOn(':projectB')` v projektu A.

Jak již bylo řečeno, task lze spustit z jakékoli složky celého projektu, tzn. jak ze složky kořenového projektu, tak i z adresáře subprojektu. Gradle nepoužívá lomítka jako oddělovače, místo toho používá dvojtečku. Kořenový projekt je označován dvojtečkou a nemá žádné jméno. Vzor pro spuštění tasku je následující (13 s. 272–273):

```
:<jméno projektu>:<jméno tasku>
```

Pokud chci spustit task `hello` z kořenového projektu, ale nacházím se v jiné složce, provedu spuštění příkazem `gradle :hello`. Pokud chci ze subprojektu `projectA` spustit task `hello`, ale nacházím se v jiné složce, spustím task příkazem `gradle :projectA:hello` (13 s. 272–273).

Spuštěním buildu ve složce subprojektu provede Gradle build daného subprojektu a projektů, na kterých je daný subprojekt závislý. Pokud chci spustit i testy v těchto závislých projektech, musím spustit `gradle buildNeeded`. Když je potřeba zkontrolovat, zda se testy v jiných projektech nerozbily změnou v projektu, na kterém jsou závislé, je nutné spustit

`gradle buildDependents`. Tento příkaz provede build s testy projektů, které mají na daném projektu závislost.

9 Rešerše dostupné literatury

9.1.1 Uživatelské příručky od vydavatelů

Pro oba nástroje existují uživatelské příručky od jejich vydavatelů. Tyto příručky jsou značně obsáhlé. Jsou dostupné online ve formátu HTML nebo je lze stáhnout ve formátu PDF.

Uživatelská příručka k systému Gradle ve formátu PDF má ve verzi 1.3 (13) již 343 stran a s každou další verzí počet stran narůstá. První část knihy slouží jako zkrácený tutoriál a demonstruje možnosti Gradlu. Druhá část slouží jako kompletní referenční příručka, veškerá funkcionalita je zde podrobně rozebrána. Oproti knihám je zde popsáno i mnoho pluginů.

Uživatelská příručka pro Maven je dostupná na oficiálních stránkách ve formátu HTML (viz rozcestník (31) nebo (32)). Zdaleka není tak dobře vypracovaná jako příručka pro Gradle a rozhodně ji nelze označit za kompletní příručku. Její stažení v PDF verzi v době psaní této práce nefungovalo, a tak bylo nutné projít celé rozsáhlé webové stránky a pracně hledat to, co bylo potřeba vědět.

9.1.2 Vydané knihy o systému Maven

Přehled knih vydaných o systému Maven je k dispozici zde: (33). Pro verzi Maven 3 vyšla zatím jen jedna tištěná kniha, ostatní knihy jsou pro verzi Maven 2 a vznikly před více než 4 lety, proto jsem jejich rešerši vynechala. V poslední době všeobecně ubývá tištěných knih, zato ale vzniká více online příruček. Velice kvalitní jsou příručky od firmy Sonatype, které jsem do rešerše zahrnula, neboť jsou průběžně aktualizovány a na oficiálních stránkách Mavenu jsou na ně uvedeny odkazy.

O systému Maven ve verzi 3 zatím vyšla v anglickém jazyce pouze jedna kniha, a to *Apache Maven 3 Cookbook* (10). Tato kniha obsahuje přes 50 receptů pro zlepšení projektu. Každý recept je jen pro základní představu, jak asi dané zlepšení funguje, takže kniha nezachází hlouběji do problematiky a rozhodně není vhodná pro úplné začátečníky, kteří se chtějí Maven naučit od základů. Kniha obsahuje recepty od základů Mavenu přes tvorbu reportů a dokumentace, podporu Mavenu pro Java, Scala, Groovy a Flex projekty až po integraci s IDE nebo napsání vlastního pluginu.

Online příručka *Maven by Example* (11) od firmy Sonatype provází čtenáře funkcionalitou Mavenu pomocí konkrétních příkladů, které jsou dostupné ke stažení. Prvním příkladem je jednoduchý projekt, ale nechybí zde ani příklad webové aplikace nebo multimodulové enterprise aplikace. Tuto příručku nelze označit jako kompletní, jelikož neobsahuje danou

problematiku do hloubky, avšak dá se podle ní naučit s Mavenem zacházet. Kniha obsahuje i velmi užitečnou kapitolu o optimalizaci a refaktorování POM.

Kompletní referenční příručka od firmy Sonatype *Maven: The Complete Reference* (1) je skvělá pro pochopení celého systému Maven a je taktéž dostupná online. Funkcionalita je zde rozebrána do detailů, a to dokonce lépe než na oficiálních stránkách projektu. Kniha obsahuje rovněž mnoho příkladů. Kniha zahrnuje kromě výše zmíněných témat z jiných knih i kapitolu o podpoře pro vývoj Android aplikací nebo přílohu o nastavení v souboru `settings.xml`.

9.1.3 Vydané knihy o systému Gradle

Přehled o knihách týkajících se systému Gradle je k dispozici zde: (34). O Gradlu vyšly dosud dvě publikace a čtyři knihy se připravují.

První z nich, *Building and Testing with Gradle* (2), vyšla již v roce 2011, kdy nebyla ještě dostupná verze Gradlu 1.0. Kniha obsahuje základní informace, které vysvětluje na ukázkových příkladech. Má pouze 87 stran, takže se problematikou nezabývá do hloubky, ale pouze demonstruje, co Gradle umí, a to především jeho základy. Velice se mi zde líbí, že je přehledně vysvětlen objekt `DefaultTask` a jeho proměnné a metody. Nechybí ani popis využití Ant nebo Maven build skriptů. Chybí však popisy pluginů a integrace s IDE.

Druhou vydanou knihou o Gradlu je *Gradle Effective Implementation Guide* z roku 2012. Tato kniha má bez indexu 356 stran, a tak zachází hlouběji při vysvětlování základů systému Gradle. Bohužel zde většinou chybí přehledné vysvětlení, k jakému objektu patří daná metoda nebo proměnná. Na druhou stranu je zde ale přehledněji vysvětlena na jednom místě podobná funkcionalita, např. jakými způsoby lze vyloučit task z provádění. Velká část knihy je také věnována různým rozšířením, jako například propojení s pluginy na kontrolu a revizi kódu, kontinuální integraci, propojení s IDE a podpoře dalších programovacích jazyků (Groovy, Scala). Tato témata nejsou předmětem této práce.

10 Porovnání systémů Maven a Gradle

V této kapitole kompletně porovnávám nástroje Maven a Gradle. Porovnání je provedeno ve výše popsaných oblastech, ale přidala jsem rovněž několik dalších parametrů, podle kterých lze oba nástroje porovnat. Schopnosti, které popisuji u jednoho nástroje, ale u druhého ne, jsem ověřovala pomocí internetu, zda jimi nedisponuje i druhý nástroj například v podobě pluginu. Pokud jsem našla řešení, vždy jsem ho pro jistotu otestovala, což se ukázalo jako velice dobrý nápad, jelikož některé pluginy nefungovaly. Na závěr uvádím svůj názor na oba systémy.

10.1 Základy systémů

10.1.1 Skriptovací soubor

Největším rozdílem obou systémů je jiný jazyk pro konfigurační soubor. Maven využívá formát XML. Gradle využívá DSL a jazyk Groovy. Pro nováčky je lepší využít ke konfiguraci XML, protože má předdefinovanou strukturu a je zde tím pádem menší prostor pro chyby. Oproti Mavenu a zápisu v XML má zápis v Gradlu méně znaků i řádků.

Gradle je více flexibilní, umožňuje jednoduše naprogramovat nebo upravit funkcionalitu. V Mavenu je nutno napsat nový plugin.

Groovy je nadstavba Javy a ve skriptu lze využívat i Java syntaxi. Je lepší znát alespoň základy Groovy a umět v něm číst, protože pokud bude build napsaný v Groovy, může se stát, že někdo nepochopí, co se děje, a vznikne tak chyba. Jednou z výhod Groovy oproti Javě je, že Groovy kód zabere méně řádků. Je lepší se na začátku projektu domluvit, v jaké syntaxi bude skriptovací soubor zapsán – zda striktně v Javě nebo i v Groovy.

10.1.2 Wrapper

Gradle poskytuje wrapper, díky němuž lze u uživatelů vynutit použití určité verze Gradlu, aniž by jej měli nainstalovaný. Maven takovou funkcionalitu neposkytuje a nepodařilo se mi ji dohledat ani v podobě pluginu.

10.1.3 Logování

Gradle díky přístupu k loggeru umožňuje doplnit si logování kamkoli. V Mavenu není přímý přístup k loggeru, lze jej využít pouze ve vlastním pluginu.

10.1.4 Šablony projektů

Maven nabízí mnoho šablon projektů v pluginu Archetype. Gradle oficiálně takový plugin chystá v budoucích verzích. Momentálně lze využít plugin Templates, který ale obsahuje pouze základní šablony, jež vytvoří pro projekt adresářovou strukturu. Je možné vytvořit i vlastní šablonu, více informací zde: (35).

10.1.5 Vytvoření dokumentace

Plugin Site nabízí v Mavenu jednoduché a rychlé vygenerování projektové dokumentace, kterou je možné nasadit i na server. Gradle obdobnou funkčnost nemá.

10.1.6 Build

Gradle podporuje inkrementální build, který výrazně zkrátí čas buildu. Maven má tuto podporu pouze v pluginu Compiler, ale ani tam není řešena ideálně, více informací lze nalézt zde: (36).

10.1.7 Profily

Maven nabízí jednoduché řešení pro tvorbu profilů. Gradle takovou možnost nenabízí, ale je velice jednoduché si tuto funkci naprogramovat.

10.1.8 Ostatní

Gradle ve svém základu nabízí GUI (grafické uživatelské prostředí, angl. Graphical User Interface), které zobrazuje strom projektů a strom jejich tasků. GUI se vyvolá příkazem `gradle -gui` (12 s. 22–24). Maven ve svém základu GUI nemá, ale existují GUI pluginy, které zobrazují strom závislostí, více zde: (37). Tento strom lze vyvolat i v příkazové řádce, ale ne graficky příkazem `mvn dependency:tree` (38).

V Gradlu lze v příkazové řádce zadat pouze část názvu tasku nebo zkrácený název tasku pomocí velbloudí notace. Maven toto zkracování neposkytuje.

Gradle nabízí jednoduchý způsob, jak zjistit všechny dostupné tasky příkazem `gradle tasks` a všechny dostupné proměnné příkazem `gradle properties`. Maven umožňuje pouze vypsání cílů pro konkrétní plugin příkazem `mvn help:describe -Dplugin=<název pluginu> -Dfull`.

10.2 Řízení závislostí

Tím, že Gradle využívá Ivy, které podporuje Maven popisovače, a zároveň přímo umožňuje využívat Maven úložiště, je řízení závislostí téměř totožné. V obou systémech jsou dva typy úložišť, jejich definice je jednoduchá. Oba systémy podporují tranzitivní závislosti a dovolují definovat dynamické verze závislosti. Oba systémy umožňují i publikovat artefakty.

Výhodou Gradle je zápis závislosti na jeden řádek oproti šesti řádkům v Mavenu.

V Gradlu je plugin jeho součástí a nový je umístěn v další verzi Gradlu. Prozatím to není tak velký problém, protože každých 6 týdnů je vydána nová verze Gradlu. Do budoucna, až se frekvence vydávání prodlouží, by to ale mohlo být kontraproduktivní, a proto Gradle plánuje změnu. Lze předpokládat, že funkcionalita pluginů bude obdobná jako u Mavenu, tj. že bude zřízeno úložiště pro pluginy, kam se budou pluginy a jejich verze umísťovat.

Nevýhodou Mavenu je, že pokud není specifikována verze pluginu, Maven automaticky stáhne z úložiště nejnovější verzi, což může způsobovat problémy, a z praxe vím, že se tím občas rozbije celý build.

10.3 Testování

Gradle nabízí implicitní pravidlo pro vyhledávání testů pomocí anotací nebo zkoumáním rodiče testovací třídy, což Maven neumožňuje. Toto pravidlo je ale možné vypnout, čímž se aktivuje pravidlo, které testy vyhledává podle názvu, stejně jako je tomu v Mavenu. Další pravidla lze přidat v obou systémech.

Pokud nastane chyba, je build automaticky stopnut v obou systémech, což lze také v obou systémech vypnout. Oba nástroje taktéž umožňují testování při buildu přeskočit.

Rozdíl výsledných reportů testů je v obou systémech patrný. Zatímco Maven v základu disponuje pouze reportem v XML, Gradle nabízí pro uživatele přívětivější report v HTML. V Mavenu je také možné vygenerovat reporty testů v HTML, ale je k tomu třeba nakonfigurovat plugin Surefire Report, o kterém lze zjistit více informací zde: (39).

Oba nástroje podporují paralelní testování.

10.4 Multiprojektové buildy

Gradle umožňuje použít jeden skriptovací soubor, ve kterém je nastavení všech projektů, a to bez opakování konfigurace, což u Mavenu nejde.

Výhodou Mavenu je možnost zobrazení výsledného POM, ve kterém se projeví dědičnost pomocí příkazu `mvn help:effective-pom`. Gradle nemá možnost zobrazit výsledný soubor `build.gradle` pro konkrétní projekt.

10.5 Ostatní

Gradle provádí každých 6 týdnů release nové verze, ve kterém jsou opravy chyb, ale i přidání nové funkcionality. Maven neprovádí release pravidelně, mezi posledními dvěma vydáními je prodleva více než rok. Je to dáno tím, že Maven je už stabilní a nové verze pluginů se vydávají zvlášť.

Oba nástroje za sebou mají velmi silnou komunitu přívrženců.

Pokud se vyskytne problém, je možné, že jeho řešení již existuje na nějakém serveru, kde se řeší problémy v programování. Zde v počtu dotazů jednoznačně vede Maven, jelikož je to starší nástroj. Např. server stackoverflow.com po zadání dotazu Gradle nalezne ke dni 5. 4. 2013 2052 výsledků. Pro Maven nalezne 56 879 výsledků.

Gradle reaguje na svém fóru http://forums.gradle.org na dotazy během několika dnů.

Celkově musím zhodnotit dokumentaci dostupnou ke Gradlu jako lepší. S každým vydáním nové verze vydá Gradle i aktualizovanou příručku, ve které jsem našla většinu informací. Maven žádnou takovou příručku nevydává a dokumentace dostupná na stránkách není zrovna řešena nejlépe. Dostat se na rozcestník dokumentace je velice složité a nepřehledné. Sonatype se snaží své dvě příručky průběžně aktualizovat s novými verzemi Mavenu, ale i zde jsem si všimla drobných chybiček, např. popis využití souboru `profiles.xml` k definování profilů, avšak tento soubor není od verze 3.0 v Mavenu nadále podporován. Dále mi v dokumentacích chyběla zmínka o tom, že Maven umožňuje paralelní testování.

10.6 Závěr

Cílem této diplomové práce není určit, který z nástrojů je lepší. Oba nástroje jsou vhodné pro malý i velký standardní projekt. Pokud je třeba vybrat nástroj pro malý projekt, kde nebude třeba řešit nestandardní záležitosti, je výběr nástroje jen čistě otázkou vkusu. Věřím, že výše uvedeným porovnáním ulehčím výběr vhodného nástroje.

Pro větší projekty je vhodnější Gradle, jelikož je při buildu rychlejší a tím šetří čas. Ale použít i zde Maven není rozhodně chybou. Pokud předem víte, že na projektu bude třeba řešit nestandardní situace, které by se v Mavenu řešily napsáním vlastního pluginu, je lepší využít Gradle.

Gradle v mnoha aspektech logicky vyhrává nad Mavenem, protože jeho cíl byl od začátku následovat konvence Mavenu, vzít si z něj a jiných buildovacích nástrojů to nejlepší a naopak to, co nebylo vyhovující v Gradlu, zlepšit. Jedním z mott Gradlu je: *„Udělat nemožné možným, možné jednoduchým a jednoduché elegantním“* (40 s. 6).

Přechod na Gradle bude asi jednodušší pro uživatele Antu. Některým uživatelům Mavenu vadí volnost, kterou Gradle vývojáři nabízí, jelikož jsou zvyklí na striktní konvence Mavenu. U mě osobně vítězí Gradle, asi nejvíce na něm oceňuji možnost inkrementálního buildu a celkově rychlejší build, poněvadž často pracuji na velkých projektech, kde build trvá několik minut.

11 Praktická část

11.1 Instalace

Pro praktickou část diplomové práce je třeba mít nainstalovanou Javu verze 1.7, Maven a Gradle, jak je popsáno v kapitole 3 Instalace. Doporučuji si nainstalovat Gradle verze 1.5.

Dále je nutné mít nainstalovaný aplikační server, postačí server Apache Tomcat (dále jen Tomcat). Já jsem pracovala s verzí 6.0.33. Na server je třeba nasadit war soubor aplikace Sonar, který jsem umístila do přílohy této diplomové práce a je verze 3.5.1. Do složky `webapp` serveru Tomcat je potřeba umístit toto war. Defaultně je poté adresa pro Sonar localhost:8080/sonar. Tuto adresu je možné změnit v buildovacích souborech obou systémů umístěných v kořenové složce.

11.2 Příkazy

11.2.1 Spuštění buildu

V Mavenu se spustí build i s integračními testy obou modulů v kořenové složce příkazem `mvn clean install`. Výsledné war `thesisapp.war` je umístěno ve složce `target`. V Gradlu se spustí build taktéž s integračními testy v kořenové složce příkazem `gradlew clean build`. Výsledné war je ve složce `build/libs`.

11.2.2 Spuštění Sonar reportu

V Mavenu se spustí report Sonaru příkazem `mvn sonar:sonar`. V Gradlu se spustí report příkazem `gradlew sonarRunner`.

11.3 Úskalí

Během vývoje jsem se setkala s několika problémy, na které bych chtěla upozornit.

11.3.1 Plugin Maven2Gradle

Plugin Maven2Gradle slouží k převodu Maven konfiguračního souboru POM na skriptovací soubor `build.gradle`. V literatuře jsem několikrát četla o tom, jak je tento převod jednoduchý (13 s. 207–208). V realitě tomu tak ovšem není. V mém případě se tento plugin dal použít hlavně pro převod závislostí, kde funguje velice dobře. Dále je plugin schopný převést Maven souřadnice – `groupId`, `artifactId` a `version` a nastavení verze Javy pro překladač. Úroveň tohoto pluginu mě velice zklamala, předpokládala jsem i další základní funkčnost, jako např. pokud je nastaveno `packaging` na war, aplikuje se v Gradle plugin War nebo převod nastavení pro Sonar. Nutno podotknout, že tento plugin je stále ve vývoji.

11.3.2 Gradle a Sonar

Do verze 1.5 využíval Gradle pro práci se Sonarem plugin Sonar. I já jsem tento plugin vyzkoušela, avšak při pokusu o vygenerování reportu skončil Gradle chybou. Tento plugin fungoval, ale postupem času se přišlo na to, že nefunguje pro verzi Sonaru 3.4 a vyšší. Já používám verzi 3.5.1. Opravu vydal Gradle s verzí 1.5, kdy vznikl nový plugin Sonar Runner. Tento plugin již funguje i pro vyšší verzi Sonaru a v příručce pro verzi Gradlu 1.5 již autoři doporučují využívat plugin Sonar Runner namísto pluginu Sonar a doplňují i vysvětlení proč. Jelikož používám verzi 1.3, na které mi vše fungovalo, vytvořila jsem v kořenové složce projektu pro Gradle wrapper s verzí 1.5. Použití wrapperu je vysvětleno v kapitole 5.8 Gradle wrapper (v příloze je již u projektu přiložena složka s wrapperem verze 1.5). Využití wrapperu je nutné, protože jak jsem již zmínila dříve, Gradle vydává aktualizace pluginů vždy s novou verzí, Gradle verze 1.3 tedy neumí využít plugin Sonar Runner, který je až ve verzi 1.5.

11.3.3 Překlad Groovy kódu

Pro překlad Groovy kódu používá Maven plugin Groovy-Eclipse. Gradle oficiálně tento plugin nenabízí, nalezla jsem ale projekt v Gitu, který by měl poskytovat funkčnost tohoto pluginu pro Gradle, více zde: (41). Tento plugin nebyl oficiálně vydán, tudíž je třeba si projekt z Gitu stáhnout a provést build. Zde ovšem vznikl problém, protože v závislostech je uvedeno Gradle API:

```
compile gradleApi()
```

Tento projekt vznikl před třemi lety a vznikal pro minimální verzi Gradlu 0.9-rc-2. Při buildu jsou třeba třídy z Gradle API, které už ale současná verze Gradlu, kterou využívám, tedy 1.3, neposkytuje. Při buildu tak vznikne několik chyb a build je neúspěšný. Tento problém jsem se rozhodla vyřešit opět použitím wrapperu verze 1.0, 0.9 i 0.9-rc-2, ale chy-

by v buildu se vyskytly znovu nebo se objevily nové. Naposledy bylo do tohoto pluginu něco přidáno v Gitu před dvěma lety. Já si myslím, že je škoda, že mnoho pluginů v Gradlu bylo někým vytvořeno, ale plugin nadále není udržován funkční i s novými verzemi Gradlu. Jelikož mi toto zkoušení zabralo dost času, rozhodla jsem se pokračovat bez tohoto pluginu a ukázalo se, že pro překlad kódu stačí aplikovat plugin Groovy.

11.3.4 Rozsah provided

V Gradlu není mezi základními rozsahy závislostí rozsah `provided`, který je znám z Mavenu a znamená, že daná závislost nebude ve výsledném archivu. Pokud na projekt aplikujeme plugin War, přidají se nám dva rozsahy `providedCompile` a `providedRuntime`. Tyto rozsahy jsou stejné jako rozsahy `compile` a `runtime` popsané dříve s tím rozdílem, že nejsou přidané do výsledného war souboru. Problém s rozsahem `provided` nastává ve chvíli, kdy je potřeba, aby subprojekt byl zabalen do jar archivu. V Gradlu jsou rozsahy konfigurace a lze přidat vlastní konfiguraci jako v následujícím příkladu:

```
configurations {
  provided
}

sourceSets {
  main {
    compileClasspath += configurations.provided
  }
}
```

Konfiguraci je poté ještě nutno přiřadit na classpath rozsahu konkrétního sourceSetu. V příkladu je konfigurace přiřazena sourceSetu `main` a classpath rozsahu `compile`, závislost s rozsahem `provided` tedy v příkladu nebude v `runtime` rozsahu dostupná.

11.3.5 Integrační testy

Integrační testy jsou v Mavenu realizované pomocí frameworku Arquillian a pluginu Failsafe. Gradle nenabízí obdobu tohoto pluginu a testy lze spustit bez něj.

11.3.6 Využití lokálního Maven úložiště v Gradlu

V praxi, ale i při psaní této diplomové práce jsem se setkala s problémem využívání lokálního Maven úložiště, pokud je na počítači používán Maven i Gradle zároveň. Pokud se nejprve spustí build v Gradlu, jsou závislosti uloženy v lokálním Gradle úložišti. Je-li následně spuštěn build v Mavenu, stáhnou se závislosti do lokálního Maven úložiště, jelikož Maven nepodporuje a tudíž ani nevyužívá lokální Gradle úložiště. Pokud poté opět spustím

build Gradlu, nastane konflikt, neboť závislosti jsou v obou úložištích. V příkazové řádce Gradle vypíše následující chybu:

```
Artifact 'commons-collections:commons-collections:3.2.1@jar' not found.
```

Tato chyba je Gradlu nahlášena a v budoucnu by se měla vyřešit, bohužel momentálně ale nemá prioritu. Řešením je vymazání daných závislostí v lokálním Gradle úložišti. Toto řešení je ale velice nepraktické, a proto je lepší nevyužívat v Gradlu lokální Maven úložiště. I když ani toto řešení není zcela optimální, protože některé závislosti se budou nacházet v obou úložištích a tím pádem budou zbytečně zabírat místo na pevném disku.

12 Závěr

Hlavním cílem této diplomové práce byla analýza schopností obou nástrojů. Popis vlastností byl koncipován tak, aby ho bylo možné použít jako uživatelskou příručku s ukázkovými příklady k oběma nástrojům. Od každého systému jsem přečetla oficiální uživatelskou příručku a další knihy. Každá kniha vysvětlovala systém svým způsobem a z trochu jiného úhlu pohledu, což mi poskytlo nadhled nad danou problematikou. Hlavní přínos vidím v tom, že výsledná příručka je složena z informací z několika příruček uvedených v rešerši. Ne všechny knihy byly kvalitně napsány a moje příručka obsahuje všechny nejdůležitější informace ve zkrácené verzi.

Dalším cílem bylo nabídnout čtenáři porovnání mezi oběma nástroji a díky tomu mu pomoci při výběru vhodnějšího nástroje pro jeho projekt. Pro tuto kapitolu není k dispozici dostupná literatura, v některých knihách bylo např. porovnání jednoho aspektu s druhým systémem, ale nikde nebylo dostupné kompletní porovnání. Rozhodla jsem se tedy oba tyto nástroje porovnat zejména ve funkčnostech, které popisuje moje uživatelská příručka, a dále přidat několik dalších důležitých aspektů, jako např. frekvenci vydávání nové verze. Funkčnosti, které nejsou popisovány v příručkách, ale druhý systém jimi disponuje, jsem dohledávala pomocí internetu a ověřovala, že daná funkčnost v systému opravdu funguje.

Cílem praktické části této diplomové práce bylo demonstrovat schopnosti obou nástrojů na reálné aplikaci pomocí konfiguračního skriptu pro Maven i Gradle. Konfigurační skript pro systém Maven již byl k aplikaci přiložen, já jsem ho rozšířila a doplnila. Pro Gradle byl skriptovací soubor vytvořen tak, aby uměl vše, co je nastavené v Mavenu. Během praktické části jsem narazila na některá úskalí, která v práci popisuji a tím ostatní uživatele upozorňuji, jakou cestou se vydat, nebo naopak nevydávat.

Terminologický slovník

Termín	Zkratka	Význam [zdroj]
Integrated Development Environment	IDE	Vývojové prostředí je software, který je zaměřen na usnadnění práce vývojáři. Většinou je zaměřené na jeden programovací jazyk, např. pro Javu Eclipse, NetBeans. [Wikipedia]
Java		Objektově orientovaný programovací jazyk, který vyvinula společnost Sun. [autor]
Groovy		Objektově orientovaný programovací jazyk, lze využít syntaxi Javy. [autor]
framework		Framework je softwarová struktura, která slouží jako podpora při programování a vývoji a organizaci jiných softwarových projektů. Může obsahovat podpůrné programy, knihovnu API, návrhové vzory nebo doporučené postupy při vývoji. [Wikipedia]
open-source		Otevřený software, který díky licenční politice umožňuje využití i změnu daného softwaru. [autor]
buildovací nástroj		Nástroj pro sestavení a distribuci aplikace. [Sonatype]
multiprojekt		Projekt složený z více modulů neboli subprojektů. [autor]
subprojekt		Modul projektu. [autor]
metadata		Strukturovaná data o datech. [Wikipedia]
release		Oficiálně vydaná konkrétní verze softwaru. [autor]
plugin		Zásuvný modul, software, který nepracuje samostatně, ale jako doplňkový modul jiné aplikace, a rozšiřuje tak její funkčnost. [Wikipedia]
classpath		Proměnná, která definuje cestu k souborům s příponou class. [autor]

Termín	Zkratka	Význam [zdroj]
closure		Prvek programovacího jazyka Groovy, blok kódu nebo také ukazatel metody. [Codehaus]
Java Virtual Machine	JVM	Nástroj na spuštění programů napsaných na platformě Java. [autor]
Java Development Kit	JDK	Produkt, který obsahuje soubor základních nástrojů pro vývoj aplikací pro platformu Java. [Wikipedia]
Domain-specific language	DSL	Doménově specifický jazyk je programovací jazyk, který je prostřednictvím vhodné abstrakce a výrazového slovníku zaměřen na omezenou, konkrétní problémovou doménu. [Wikipedia]
snapshot		Verze softwaru, která nebyla oficiálně vydaná a je označená příponou SNAPSHOT. [autor]
refaktorování		Disciplinovaný proces provádění změn v softwarovém systému takovým způsobem, že nemají vliv na vnější chování kódu, ale vylepšují jeho vnitřní strukturu s minimálním rizikem vnášení chyb. [Fowler]

Seznam literatury

1. O'BRIEN, Tim, Manfred MOSER, John CASEY, Brian FOX, Jason Van ZYL, Eric REDMOND a Larry SHATZER. *Maven: The Complete Reference* [online]. 2011. S.l.: Sonatype, Inc. [vid. 26. leden 2013]. Dostupné z: http://www.sonatype.com/books/mvnref-book/pdf/mvnref-pdf.pdf.

2. BERGLUND, Tim a Matthew MCCULLOUGH. *Building and testing with Gradle*. Sebastopol, CA: O'Reilly Media, Inc., 2011. ISBN 978-1-449-30463-8.

3. *Apache Ant* [online]. S.l.: s.n., 2013. [vid. 3. březen 2013]. Dostupné z: http://cs.wikipedia.org/w/index.php?title=Apache_Ant&oldid=9029010.

4. SADOGURSKY, Baruch. Your Next Successful Build. In: [online]. S.l. [vid. 3. březen 2013]. Dostupné z: http://prezi.com/ngwwf1wcfn7n/copy-of-your-next-successful-build/.

5. MASSOL, Vincent, John CASEY, Brett PORTER, Carlos SANCHEZ a Jason van ZYL. *Better Builds With Maven* [online]. S.l.: s.n., 2008 [vid. 15. leden 2013]Dostupné z: http://www.maestrodev.com/wp-content/uploads/2012/03/betterbuildswithmaven-2008.pdf

6. *Apache maven* [online]. S.l.: s.n., 2012. [vid. 15. leden 2013]. Dostupné z: http://cs.wikipedia.org/w/index.php?title=Apache_maven&oldid=9475834.

7. Maven. *FI WIKI* [online]. 2012 [vid. 26. únor 2013]. Dostupné z: http://kore.fi.muni.cz/wiki/index.php/Maven

8. Maven - Maven Getting Started Guide. *Apache Maven Project* [online]. [vid. 3. březen 2013]. Dostupné z: http://maven.apache.org/guides/getting-started/index.html

9. Maven - What is Maven? *Apache Maven Project* [online]. [vid. 3. březen 2013]. Dostupné z: http://maven.apache.org/what-is-maven.html

10. SRIRANGAN. *Apache Maven 3 Cookbook: over 50 recipes towards optimal Java software engineering with Maven 3*. Birmingham: Packt Publishing Ltd., 2011. ISBN 978-1-84951-244-2.

11. O'BRIEN, Tim, John CASEY, Brian FOX, Jason Van ZYL, Juven XU, Thomas LOCHER, Dan FABULICH, Eric REDMOND a Bruce SNYDER. *Maven by Example* [online]. 2011. S.l.: Sonatype, Inc. Dostupné z: http://www.sonatype.com/books/mvnex-book/pdf/mvnex-pdf.pdf.

12. IKKINK, Hubert Klein. *Gradle Effective Implementation Guide: Empower yourself to automate your build*. Birmingham: Packt Publishing Ltd., 2012. ISBN 978-1-84951-810-9.

13. DOCKTER, Hans a Adam MURDOCH. *Gradle User Guide* [online]. 2012. S.l.: Gradleware, Inc. [vid. 10. leden 2013]. Dostupné z: http://www.gradle.org/docs/1.3/userguide/userguide.pdf.

14. Gradle - NetBeans Plugin detail. *NetBeans* [online]. [vid. 11. březen 2013]. Dostupné z: http://plugins.netbeans.org/plugin/41776/gradle

15. Instalace Javy pro vývoj programů. *java.vse.cz* [online]. [vid. 3. březen 2013]. Dostupné z: http://java.vse.cz/Java/Instalace

16. Maven - Download Apache Maven. *Apache Maven Project* [online]. [vid. 3. březen 2013]. Dostupné z: http://maven.apache.org/download.cgi

17. Gradle - Downloads. *Gradle* [online]. [vid. 3. březen 2013]. Dostupné z: http://www.gradle.org/downloads

18. Maven - Introduction to the Standard Directory Layout. *Apache Maven Project* [online]. [vid. 4. březen 2013]. Dostupné z: http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

19. Maven - POM. *Apache Maven Project* [online]. [vid. 11. březen 2013]. Dostupné z: http://maven.apache.org/ref/3.0.4/maven-model/maven.html

20. Maven - POM Reference. *Apache Maven Project* [online]. [vid. 30. březen 2013]. Dostupné z: http://maven.apache.org/pom.html

21. Maven - Introduction to the Build Lifecycle. *Apache Maven Project* [online]. [vid. 16. březen 2013]. Dostupné z: http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

22. Maven - Introduction to the POM. *Apache Maven Project* [online]. [vid. 16. březen 2013]. Dostupné z: http://maven.apache.org/guides/introduction/introduction-to-the-pom.html

23. Maven - Maven in 5 Minutes. *Apache Maven Project* [online]. [vid. 16. březen 2013]. Dostupné z: http://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

24. Maven - Guide to creating a site. *Apache Maven Project* [online]. [vid. 16. březen 2013]. Dostupné z: http://maven.apache.org/guides/mini/guide-site.html

25. Maven - Introduction to Repositories. *Apache Maven Project* [online]. [vid. 23. březen 2013]. Dostupné z: http://maven.apache.org/guides/introduction/introduction-to-repositories.html

26. KRÁLIK, Miroslav. *Správa úložísk nástroja Maven* [online]. 2010. S.l.: s.n. Dostupné z: https://isis.vse.cz/auth/lide/clovek.pl?zalozka=7;id=53533;studium=52730;download_prace=1.

27. Maven - Guide to installing 3rd party JARs. *Apache Maven Project* [online]. [vid. 30. březen 2013]. Dostupné z: http://maven.apache.org/guides/mini/guide-3rd-party-jars-local.html

28. Maven - Introduction to the Dependency Mechanism. *Apache Maven Project* [online]. [vid. 10. březen 2013]. Dostupné z: http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

29. Maven Surefire Plugin - Fork Options and Parallel Test Execution. *Apache Maven Project* [online]. [vid. 5. duben 2013]. Dostupné z: http://maven.apache.org/surefire/maven-surefire-plugin/examples/fork-options-and-parallel-execution.html

30. Maven Surefire Plugin - Using JUnit. *Apache Maven Project* [online]. [vid. 5. duben 2013]. Dostupné z: http://maven.apache.org/surefire/maven-surefire-plugin/examples/junit.html#Running_tests_in_parallel

31. Maven - Users Centre. *Apache Maven Project* [online]. [vid. 3. březen 2013]. Dostupné z: http://maven.apache.org/users/index.html

32. Introduction. *Apache Maven Project* [online]. [vid. 3. březen 2013]. Dostupné z: http://maven.apache.org/ref/3.0.4/

33. Maven - External Resources on Maven. *Apache Maven Project* [online]. [vid. 3. březen 2013]. Dostupné z: http://maven.apache.org/articles.html

34. Gradle - Books. *Gradle* [online]. [vid. 3. březen 2013]. Dostupné z: http://www.gradle.org/books

35. townsfolk/gradle-templates · GitHub. *GitHub* [online]. [vid. 8. duben 2013]. Dostupné z: https://github.com/townsfolk/gradle-templates

36. JetBrains TeamCity Blog » » Incremental Building with Maven and TeamCity. *JetBrains TeamCity Blog* [online]. [vid. 8. duben 2013]. Dostupné z: http://blogs.jetbrains.com/teamcity/2012/03/14/incremental-building-with-maven-and-teamcity/

37. maven-dependency-browser - GUI browser for maven 2.0 - Google Project Hosting. *Google Code* [online]. [vid. 7. duben 2013]. Dostupné z: http://code.google.com/p/maven-dependency-browser/

38. Maven Dependency plugin - Resolving conflicts using the dependency tree. *Apache Maven Project* [online]. [vid. 7. duben 2013]. Dostupné z: http://maven.apache.org/plugins/maven-dependency-plugin/examples/resolving-conflicts-using-the-dependency-tree.html

39. Maven Surefire Report Plugin - Introduction. *Apache Maven Project* [online]. [vid. 2. duben 2013]. Dostupné z: http://maven.apache.org/surefire/maven-surefire-report-plugin/

40. MUSCHKO, Benjamin. Gradle in Action MEAP v3 - Chapter 2 - GradleinA_MEAP_ch2.pdf. *Mannin Publications co.* [online]. [vid. 10. duben 2013]. Dostupné z: http://www.manning.com/muschko/GradleinA_MEAP_ch2.pdf

41. gradle-groovy-eclipse-compiler. *GitHub* [online]. [vid. 21. duben 2013]. Dostupné z: https://github.com/adammurdoch/gradle-groovy-eclipse-compiler

Seznam obrázků a tabulek

Základní text

Seznam obrázků

Obrázek 1: 17
Obrázek 2: 39
Obrázek 3: 41
Obrázek 4: 42

Seznam tabulek

Tabulka 1: Základní Maven adresářová struktura (18) 7
Tabulka 2: Maven souřadnice (11 s. 28–29) 8
Tabulka 3: Úrovně logování (12 s. 75) 21
Tabulka 4: Metody kopírovacího tasku (13 s. 95–96) 22
Tabulka 5: Atributy projektu (12 s. 70) 24
Tabulka 6: Přehled předkonfigurovaných úložišť (12 s. 113) 34
Tabulka 7: Fáze závislostí (12 s. 112) 36

Přílohy

1. Zdrojové kódy programu s konfiguračními soubory Mavenu
2. Zdrojové kódy programu se skriptovacími soubory Gradlu
3. War soubor aplikace Sonar