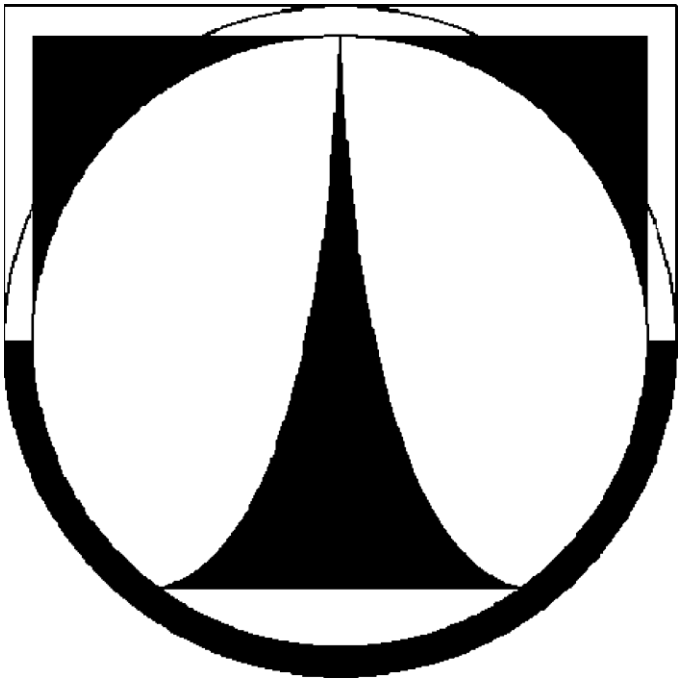
TECHNICKÁ UNIVERZITA V LIBERCI

Ekonomická fakulta



BAKALÁŘSKÁ PRÁCE

2013 **Martin Kraus, DiS**

TECHNICKÁ UNIVERZITA V LIBERCI

Ekonomická fakulta

Studijní program: B 6209 Systémové inženýrství a informatika

Studijní obor: Manažerská informatika

Tvorba archetypu Maven - inicializace J2EE projektu

Creation of Maven Archetype - Inicialization of J2EE Project

BP-EF-KIN-2013-12

Martin Kraus, DiS

Vedoucí práce: Ing. Dana Nejedlová, Ph.D.
Katedra informatiky EF TUL

Konzultant: Ing. David Wegschmied
CCA Group, a.s.

Počet stran: 41 Počet příloh: 0

Datum odevzdání: 10.5.2013

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, zejména §60 - školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

V Plzni, 10.5.2013

ANOTACE

Bakalářská práce se zabývá využitím nástoje Maven v praxi. Práce je rozdělena do dvou částí. V teoretické části jsou vysvětleny základní charakteristiky programovacího jazyka Java, zejména se zaměřením na Java EE. Dále se práce věnuje nástroji Maven od Apache Software Foundation. Zdůrazněny jsou vlastnosti tohoto nástroje a jeho odlišnosti od jiných podobných či dříve používaných nástrojů. Praktická část práce je věnována vytvoření archetypu nástroje Maven a jeho následné použití pro vytvoření budoucích projektů Javy.

KLÍČOVÁ SLOVA

nástroj Maven; programovací jazyk Java; Java EE; Apache Software Foundation; SW archetyp

ANNOTATION

This bachelor‘s thesis deals with the practical use of Maven tool. The thesis is divided into two parts. The theoretical part explains basic characteristics of programming language Java, especially with focus on Java EE. Furthermore, the thesis deals with work in Maven tool of the Apache Software Foundation. It emphasizes the properties of the tool and its differences from other similar or previously used tools. The practical part of the thesis is devoted to the creation of a Maven archetype and its subsequent use by the Maven tool for the creation of future Java projects.

KEY WORDS

Maven tool; programming language Java; Java EE; Apache Software Foundation; SW archetype

Obsah

Seznam obrázků

Seznam použitých zkratek, značek a symbolů

AOP - Aspect-oriented Programming

API - Application Programming Interface

CRUD – Create, Read, Update, Delete

EJB - Enterprise Java Beans

JAXB - Java Architecture for XML Binding

JDK - Java Development Kit

JVM - Java Virtual Machine

MVC - Model-View-Controller

POJO - Plain Old Java Object

POM - Project Object Model

Úvod

Téma Vytvoření archetypu Maven - inicializace J2EE projektu, jsem si vybral, protože na své roční informatické praxi ve společnosti CCA Group, a. s. běžně pracuji s Maveními projekty Java EE, a archetyp, který by dal vzniknout podobným projektům na základě určitých parametrů doposud chybí, přičemž jeho existence by významně zkrátila čas potřebný k vytvoření nového projektu.

Pro vytvoření archetypu Maven je však nutné mít alespoň základní teoretické znalosti v této oblasti. Z tohoto důvodu je práce rozdělena do dvou částí - teoretické a praktické.

V teoretické části se zabývám programovacím jazykem Java, strukturou jeho projektů a především nástrojem Maven, způsobem jeho fungování a filozofií, jenž je vytvářen společností Apache Software Foundation.

Cílem praktické části je vytvořit archetyp nástroje Maven tak, aby byl použitelný pro většinu projektů, pokud možno bez nutnosti jeho úpravy, nebo pouze s minimálními úpravami, s pomocí parametrizace archetypu.

1. Rešerše

Využití nástroje Apache Maven může být v praxi široké, nicméně začátek jeho používání provází určité problémy. Ve své práci Fuhrman a kol. [3] uvádí, že při použití Mavenu je velice důležité strávit nějaký čas tím, že se jeho uživatelé učí s ním zacházet. Funkcionalita jádra Mavenu je velice omezená a rozšiřuje se až použitím mnoha různých pluginů, které mají každý svá specifika konfigurace a použití. Udává, že naučit se a nakonfigurovat Maven pro použití je časově náročné. Nicméně také říká, že použití Mavenu oproti dřívějšímu použití Apache Ant mělo dvě zásadní výhody: 1. automatické rozlišení externích závislostí při překladu či spuštění a 2. mnohem jednodušší deploy programů z čehož vyplývá jednoduché posílání programů dalším osobám. Další výhodu autoři vidí v jednotné struktuře projektů. Celkově autoři považují jejich zkušenosti s Mavenem jako positivní a hodlají jej využívat i nadále.

Autoři Pierce a kol. [1] ve své práci pojednávají o jednom z pluginů Mavenu, Rave, jenž je určen pro práci se sociálními webovými aplikacemi. V práci je uvedeno, že pokud chce nějaký vývojář rozšířit Rave, nemusí mít, díky Mavenu, zdrojové kódy Rave. Stačí pouze vytvořit Maven projekt se závislostí na určité části Rave. Takovýto projekt potom obsahuje pouze upravované či přidané soubory se zdrojovým kódem za předpokladu ovšem, že je dobře nastavena konfigurace závislostí Mavenu.

I Kühner a kol. [2] vyzdvihují výhody nahrazení Apache Ant Mavenem pro větší standardizaci projektů. Dále zmiňují výhody jeho využití pro snadný přístup k mnoha open-source nástrojům a projektům (stačí zadat závislost do *pom.xml*). Dále zdůrazňuje, že používání Mavenu přináší i další výhody. Kontrola kvality kódu, různá měření kódu, statistiky a zprávy o kódu je snadné vygenerovat s použitím různých pluginů. Poukazuje na to, že Maven projekty je snadné udržovat a spravovat a dokumentace k různým pluginům má dostatečnou kvalitu, ovšem je pracné upravit projekt Maven tak, aby vyhovoval jistým netradičním konvencím ve své struktuře a pojmenování.

2. Java EE

Java je objektově orientovaný programovací jazyk vytvořený v roce 1995 a vyznačuje se především tím, že oproti například programovacímu jazyku C, který je čistě kompilovaný a oproti jazyku PHP, který je čistě interpretovaný, jde o jazyk, který se snaží najít kompromis mezi těmito dvěma přístupy. Kompilovaný program je převeden nejprve do strojového kódu, zatímco interpretovaný jazyk je spouštěný instrukci po instrukci za pomoci programu, tzv. interpreteru. Javovský program je převáděn do takzvaného mezikódu, který je interpretován s pomocí JVM. Díky tomu je nezávislý na prostředí, na němž je spouštěn, a zároveň nenastává tak výrazné zpomalení běhu programu jako je typické pro čistě interpretované programovací jazyky. Mimo to jde o jeden z nejpopulárnějších programovacích jazyků v současnosti, mimo jiné také proto, že je od roku 2007 vyvíjen jako open-source a tudíž se stále dynamicky rozvíjí.

Java EE je zkratka pro Java Enterprise Edition. Někdy je označována také jako Java Platform Enterprise Edition, nebo je také známá pod svým starým jménem - J2EE. Jde o součást Javy, a je určena k vývoji a provozu informačních systémů a především podnikových aplikací (proto Enterprise Edition). Jde o nadstavbu nad Javou SE (Standard Edition), od které se pro laika především odlišuje tím, že není určena na vývoj desktopových aplikací. Tím jsou myšleny aplikace běžící na tom samém počítači, na kterém pracuje jejich uživatel, obvykle pracují s lokálními daty. Aplikace Javy EE běží výhradně v rámci aplikačního serveru a jsou určeny k vývoji modulárních, robustních aplikací majících několik vrstev. Následující seznam obsahuje nejčastěji používané aplikační servery:

- Open Source aplikační servery
  - GlassFish
  - Jboss
  - Apache Tomcat
  - Geronimo
- Komerční aplikační servery
  - Oracle WebLogic Server
  - IBM Websphere

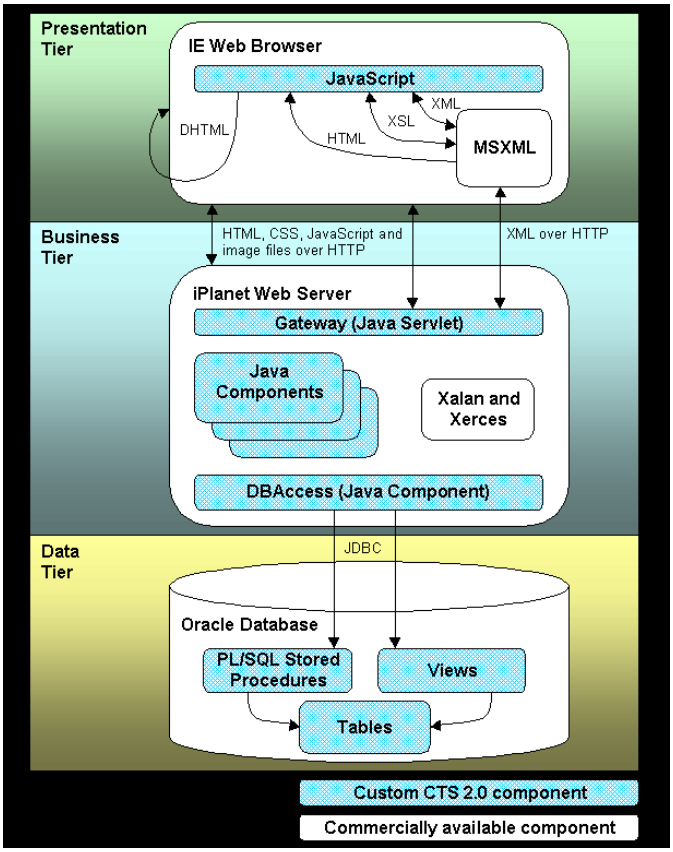
Nutno říci, že v praxi jsou open source aplikační servery mnohem častěji využívané a ty komerční se používají jenom pro opravdu velké aplikace náročné na různé možnosti serveru. Nemalou měrou se o to samozřejmě také zasazuje fakt, že open-source software lze samozřejmě používat bezplatně, zatímco komerční servery mohou být značně drahé, platí se obvykle podle počtu procesorů stroje, na kterém budou běžet.

Programování v Java EE je podobné jako v Javě SE, ale navíc obsahuje technologie jako JavaMail, JAXB (Java Architecture for XML Binding – technologie pro práci s XML dokumenty) , Servlety (a z nich vycházející JSP – Java Server Pages), či EJB (Enterprise Java Beans).

2.1.Struktura projektů J2EE

Java EE aplikace mají několik vrstev. Základní koncept je dvouúrovňový, kde vystupuje klient a server. Pro účely této práce bude použit pro ilustraci tříúrovňový koncept (obrázek č. 1), který pro příklad postačí. Ten má tedy tři základní vrstvy, které nemusí být pohromadě, nemusí dokonce být ani na stejném serveru – například z důvodu bezpečnosti lze datovou vrstvu a/nebo prezentační vrstvu mít na zvláštním serveru.

- Prezentační vrstva
- Business vrstva
- Datová vrstva (persistentní)



Obr. 1 - Tříúrovňová architektura Java EE projektu

zdroj: *http://bristle.com/~fred/XML_Arch.gif*

2.1.1. Prezentační vrstva

Prezentační vrstva sestává z komponent uživatelského rozhraní a jeho procesů, ať už jde o JSP, servlety, webové služby, či jiný způsob komunikace s uživatelem přes síť. Existuje více modelů jak přistupovat k řešení této vrstvy, jako příklad uveďme model MVC, tedy Model-View-Controller. Ten sestává ze tří různých komponent., První z nich (Model) obsahuje stav aplikace a obsahuje data posílaná uživateli, nebo naopak data uživatelem zadaná a zprostředkovává k těmto datům přístup. Druhá komponenta (View) obstarává uživatelské rozhraní a generuje jej podle potřeby uživatele z dat poskytnutých komponentou Model. Jedna instance Modelu může mít více instancí View, neboť každý uživatel může potřebovat jiný formát a jiné uspořádání požadovaných dat. Tato komponenta je obvykle implementována pomocí technologie JSP. Třetí komponenta, Controller, stojí na okraji aplikace a kontroluje tok dat mezi předchozími dvěma

komponentami. Přijímá každou žádost od uživatele (nebo jiného programu) a provádí následující úkony:

- Koordinuje žádosti klientů.
- Updatuje komponentu Model a komponenty View na základě vstupu klienta.
- Plánuje a dohlíží na to, co je třeba vrátit klientovi.
- Volá určený Model a vykoná vložené logické kroky.

2.1.2. Business vrstva

Business vrstva obsahuje vlastní algoritmy programu, zpracování dat, posílání požadavků datové vrstvě a sestává z fasád, entit, komponent a podobně. Jde o programový kód, který zpracovává data s ohledem na určenou business logiku (pravidla). Tato vrstva získá žádost od prezentační nebo nějaké jiné vrstvy (teoreticky může být vrstev mnohem víc než pouhé tři), komunikuje s datovou vrstvou (někdy prostřednictvím další vrstvy) aby získala požadovaná data, která opět zpracuje dle definovaných pravidel a vrátí je určité komponentě. Tato vrstva může být implementována mnoha různými způsoby. Pomocí session EJB, nebo v aplikaci či na webu pomocí POJO (Plain Old Java Object). V závislosti na způsobu implementace business vrstvy je nutné upravit a nastavit datovou vrstvu.

2.1.3. Datová vrstva

Poslední, datová vrstva je vrstva komunikující se zdroji aplikace, například s databází. Jde o kód, který provádí čtení, zápis, mazání, tzv. CRUD (Create, Read, Update, Delete). Tato vrstva získá požadavek od business vrstvy (nebo service vrstvy), komunikuje s databází a vrátí výsledek zpět. V závislosti na způsobu implementace business vrstvy se musí nebo nemusí starat o transakce a transakční zpracování – pokud je implementována pomocí EJB nebo POJO (Plain Old Java Object) tak se o transakční přístup postará business vrstva, ovšem pokud je business logika implementována pomocí POJO a používá AOP (Aspect-oriented Programming) framework jako například široce používaný Spring, tak jsou všechny závislosti na persistenčních mechanismech webové vrstvy odstraněny a díky tomu je lze implementovat tak, aby tento framework sám obstarával transakční chování.

Persistence je definována tak, že jde o konkrétní stav, který přetrvá proces, kterým je vytvořen. Tím je myšleno, že po skončení běhu programu jsou data zapsána do energeticky nezávislé paměti a při dalším spuštění programu je tedy možné je znovu načíst. V našem případě, pokud hovoříme o enterprise aplikacích, je persistence zajištěna databází.

Framework je softwarová struktura, sloužící jako podpora při programování - obsahuje v sobě například podpůrné programy, knihovny, API a podobně a sama o sobě již zajišťuje nějakou funkčnost, kterou může programátor rozšiřovat.

Transakčním chováním se myslí chování takové, které zajišťuje konzistentnost přenášených dat. Například každá změna v databázi je zpracovávána transakčně, což znamená, že pokud se uživatel rozhodne udělat zápis do databáze, nejprve otevře transakci, v jejím rámci udělá změnu v databázi, a pak na závěr provede commit, čímž změny ztrvalí, nebo rollback, čímž změny vrátí a databázi uvede v předchozí stav. Pokud by během takové transakce spadl systém, nebo vypadlo síťové připojení k databázi, tak databáze vrátí změny v původní stav. Transakce buď proběhne celá beze zbytku, nebo vůbec ne. Nemůže nastat situace, kdy provede pouze část zamýšlených změn.

3. Nástroj Apache Maven

Na úvod je nutné vysvětlit rozdíl mezi kompilací a buildingem:- Kompilací se převede zdrojový kód do strojového kódu, a jde o podmnožinu buildovacího procesu, jenž poté ještě provede tzv. linking, kdy strojový kód zkombinuje s potřebnými knihovnami a vznikne samostatně spustitelný soubor. Nástoj Apache Maven je open-source projekt sloužící především k buildování javovských aplikací. (Lze jej použít k buildování třeba C# projektů, Adobe Flex kódu, projektů psaných v C, nebo dokonce i k vytváření technických dokumentací.) Je vytvořen tak, aby co nejvíce zjednodušil buildovací proces a k tomu využívá deklarativní přístup určující strukturu a obsah projektu, což je rozdíl oproti tradičnímu, taskovacímu („taskovacím“ se rozumí přístup, kdy se jednoznačně musí určit každý krok procesu), přístupu. Ten používá například Apache Ant, což je starší univerzální nástroj na buildování. Díky tomu lze sjednotit vývoj v celé společnosti dle jejích standardů a zkrátit čas potřebný pro psaní a udržování buildovacích skriptů.

Příklad jak může vypadat *build.xml* nástroje Apache Ant - je uveden na obrázku č. 2.

```
<project name="my-project" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
    <!-- set global properties for this build -->
    <property name="src" location="src/main/java"/>
    <property name="build" location="target/classes"/>
    <property name="dist"  location="target"/>

    <target name="init">
        <!-- Create the time stamp -->
        <tstamp/>
        <!-- Create the build directory structure used by compile -->
        <mkdir dir="${build}"/>
    </target>

    <target name="compile" depends="init"
            description="compile the source " >
        <!-- Compile the java code from ${src} into ${build} -->
        <javac srcdir="${src}" destdir="${build}"/>
    </target>

    <target name="dist" depends="compile"
            description="generate the distribution" >
        <!-- Create the distribution directory -->
        <mkdir dir="${dist}/lib"/>

        <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
        <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
    </target>

    <target name="clean"
            description="clean up" >
        <!-- Delete the ${build} and ${dist} directory trees -->
        <delete dir="${build}"/>
        <delete dir="${dist}"/>
    </target>
</project>
```

Obr. 2 - Ukázka kódu v Apache Ant

zdroj: *4*

Příklad jak může vypadat *pom.xml*, nástroje Apache Maven, který udělá to samé - je uveden na obrázku č. 3.

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>my-project</artifactId>
    <version>1.0</version>
</project>
```

Obr. 3 - Ukázka kódu v Apache Maven

zdroj: *4*

Na těchto dvou obrázcích je patrné, že v nástroji Apache Ant je nutné každý krok explicitně zadat, nastavit umístění kde co je a kam s tím, přidat časovou značku kdy k buildu došlo, i závěrečné uklizení po úspěšném buildu. Zatímco v nástroji Maven je nutné zadat identifikaci projektu jakou si programátor zvolí a to je vše.

Maven však není určen pouze pro buildování aplikací. Jeho tvůrci jej označují spíše jako „project management tool“ a jako hlavní účel tohoto projektu označují:

- Usnadnění buildovacího procesu
- Zajištění jednotného buildovacího systému
- Zajištění kvalitních projektových informací
- Sestavení zásad pro správné postupy ve vývoji
- Umožnění transparentního přidávání nových vlastností

V praxi Maven značně zjednodušuje buildování standardních projektů, a díky němu je velice snadné přidávání nových knihoven či modulů do již existujících projektů. Mavenní buildy sestávají z deklarací a konfigurací a obejdou se téměř výhradně bez skriptování.

Následující podkapitoly se zabývají filozofií nástroje Maven, principem jeho fungování a způsoby jeho nastavení.

3.1. Convention over configuration

Tento koncept je jednou z hlavních myšlenek Mavenu. Rozumí se jím standardizace projektů a jejich nastavení, což vede k výraznému zmenšení velikosti buildovacích souborů.

Maven se o toto snaží například zavedením rozumných defaultních hodnot pro projekty. Umístění zdrojového kódu je kupříkladu v Mavenu standardně v *${basedir}/src/main/java*, prostředky jsou uloženy v *${basedir}/src/main/resources* a testy v *${basedir}/src/main/test*, zatímco se předpokládá, že výsledek přeloženého programu bude ve formátu JAR v adresáři *${basedir}/target* (*${basedir}* je domovský adresář, kde je Maven nainstalován). Jenom takovéto základní nastavení, pokud by se dělalo třeba v Apache Ant, by nebylo automatické a muselo by se naskriptovat do buildovacího skriptu. Ovšem toto je pouze příklad, Maven jde v tomto směru mnohem dál. Má celý soubor

konvencí pro kompilaci zdrojového kódu, balení distribucí i generování webových stránek o projektu a pro mnoho dalších procesů.

Je však nutné dodat, že naprostá většina předem daných nastavení lze změnit a upravit, pokud to náš projekt vyžaduje. Pokud však je náš projekt velmi netradiční a potřebuje zcela osobitý buildovací přístup, je možné že Maven nebude ideálním nástrojem a bude nutné sáhnout například po Apache Ant, ve kterém lze vše nakonfigurovat a celý postup zadat krok po kroku.

3.2. Pluginy Mavenu

Celý Maven funguje na principu pluginů, které se označují jako artefakty. (Plugin, neboli zásuvný modul, je část programu která nepracuje samostatně, ale jako doplňkový modul jiné aplikace a rozšiřuje funkčnost programu.) Artefakty se starají o vše od kompilování kódu až po publikování webových stránek o projektu. Každý projekt Mavenu musí obsahovat soubor *pom.xml* (POM je zkratka pro Project Object Model), ve kterém jsou mimo jiné napsané všechny závislosti na cizí knihovny a moduly (pluginy). Ty jsou identifikovány pomocí ArtifactId, GroupId a Version. Po spuštění příkazu *mvn install* si Maven všechny tyto závislé knihovny stáhne z centrálního Maven Repository (oficiální a zároveň největší z nich lze nalézt na: *http://mvnrepository.com*), případně z jiných repositories (webových úložišť) a uloží je na disku ve svém lokálním repository. Díky tomu je snadné změnit verzi používané knihovny, nebo přidat novou. Uživatel se nemusí o nic starat, pouze upraví *pom.xml*. Pokud se například rozhodnu, že do svého projektu chci přidat testovací skripty JUnit, jednoduše přidám závislost na danou knihovnu a o víc se nemusím starat a mohu skripty hned používat.

Pluginy mají široké využití. Existují například pluginy pro kompilování kódu, vytváření zpráv, pluginy, které provedou deploy na aplikační server. (Pojmem „deploy“ se rozumí konečné umístění programu na aplikační server kde je připraven k užívání.) Takovýto model umožňuje snadný update projektu, pokud vyjde nová verze určitého pluginu. Není potom potřeba nic měnit uvnitř projektu.

Maven repository lze vytvořit i vlastní, například jednotnou pro celou firmu, jako mezičlánek mezi globální repository a lokální repository na disku. Lze ji umístit dle potřeby - například na firemní server. Pak lze knihovny používané jednotně v organizaci

umístit tam a všichni zaměstnanci je mohou používat a nevzniknou problémy s tím, že by každý programátor používal jinou verzi knihovny, nebo pro danou funkčnost zcela jinou knihovnu.

3.3. Project Object Model (POM)

Project Object Model je základním konceptem nástroje Maven. V něm je deklarována celá struktura projektu, konfigurován build a nastavovány závislosti na jiných projektech nebo knihovnách. Projekt Maven identifikujeme primárně tak, že v něm existuje soubor *pom.xml*.

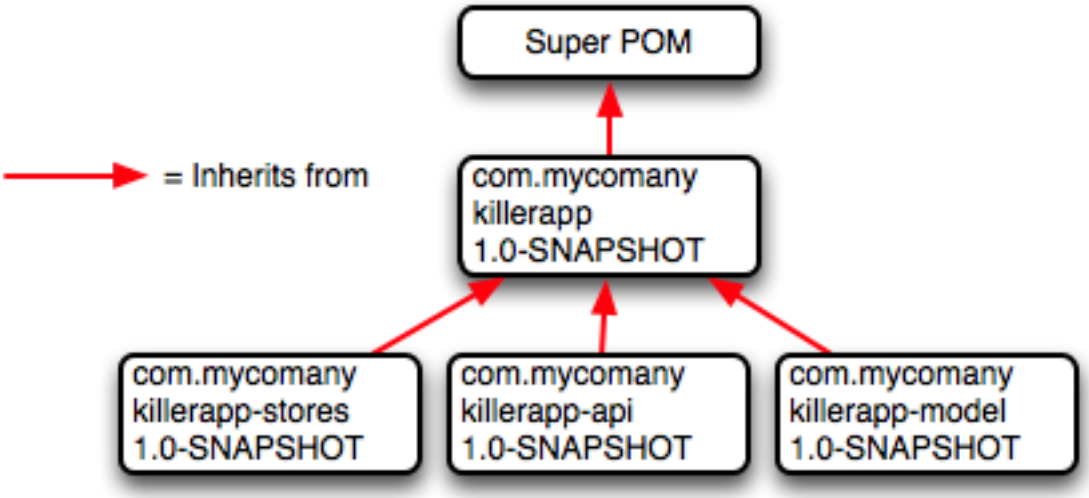
Všechny objekty Mavenu, ať už se jedná o projekty, závislosti, buildy nebo artefakty musí být někde popsány, a to v XML souboru nazvaném POM. Ten říká Mavenu o jaký druh projektu se jedná a jak je nutné upravit základní chování pro generování výsledku ze zdrojového kódu. Je to obdoba souboru *web.xml* Java webové aplikace, nebo *build.xml* nástroje Apache Ant. V souboru *pom.xml* jsou především popisy, kde jsou uvedeny závislosti, umístění repository, nebo kde je zdrojový kód. V žádném případě v POMu nenalezneme konkrétní instrukce, jako je tomu například u výše uvedeného *build.xml* Apache Ant. Filozofie Mavenu a Antu je diametrálně odlišná. (Apache Ant funguje tak, že je nutné každý krok a každé nastavení explicitně zadat).

V POMu najdeme 4 kategorie popisu a konfigurace:

- Obecné informace o projektu
    - Zde je jméno projektu, URL adresa, jméno organizace, kdo jej vyvíjí, licence a podobně.
- Nastavení buildu
    - Zde můžeme upravit základní chování Mavenu při buildování, změnit umístění kódu a testů, přidat cíle pluginů do životního cyklu (toto bude více rozvedeno v kapitole č. 3.6).
- Prostředí buildu
    - Můžeme vytvořit různé profily jako třeba pro vývoj, pro testování, pro produkci, aby například Maven vytvořil build tam, kam je třeba podle fáze ve které se vývoj nachází.

- Vztahy POMu
    - Zde budou všechny artefakty, na kterých projekt závisí, či jejichž POM nastavení dědí.

Všechny POMy našich projektů ovšem rozšiřují základní tzv. Super POM, který nastavuje defaultní hodnoty všech projektů a je součástí instalace Mavenu. Super POM nastavuje standardní proměnné, které dědí všechny ostatní projekty. V Super POMu je tedy především informace o umístění centrální Maven repository, která obsahuje různé Maven pluginy. Také tam jsou defaultní umístění adresářů a defaultní verze základních pluginů Mavenu. Závislosti jednoduché struktury projektu jsou patrné na obrázku 4.



Obr. 4 - Ukázka dědění nastavení u projektů Mavenu

zdroj: *4*

3.4.Nastavení buildu v souboru *pom.xml* a jeho verzování

Tato kapitola pojednává o jednotlivých možnostech nastavení buildovacího procesu v souboru *pom.xml*. Co se týče verzování projektu, tak projekty lze verzovat jakýmkoli způsobem, ovšem pokud budeme dodržovat určitá pravidla, bude možné využívat několik šikovných vlastností Mavenu. Je proto výhodné verzovat ve tvaru např. 1.2.3-beta kde 1 je hlavní verze, 2 je vedlejší verze, 3 je inkrementální verze a „-beta“ je označení. Pokud tedy dodržíme takovýto formát verzování, pak Maven bezpečně pozná novější verzi od starší verze, čehož lze využít například při určování závislostí pro určitý rozsah verzí. Pokud porušíme tento formát, pak k němu bude Maven přistupovat jako k prostému Stringu a tak jej tedy bude porovnávat abecedně (pokud budeme mít verzi 2.7.4beta a vynecháme pomlčku, pak nastane situace, kdy Maven bude verze porovnávat dle abecedy).

3.5.Závislosti projektu

Maven umí pracovat s interními i externími závislostmi. Interní jsou jiné projekty Maven a externí jsou ostatní, jako například framework Spring, nebo třeba jenom naše knihovna .jar kterou používáme v našem projektu pro nějaké úkony.

Každá závislost má určitý z pěti rozsahů (v angličtině jim Maven říká Dependency Scopes), který určuje, ve které classpath bude a která závislost bude přidána k aplikaci. Pojem classpath v jazyce Java vyjadřuje parametr, který určuje, kde se nacházejí jednotlivé třídy zdrojového kódu programu.

- Compile
    - Tento rozsah je defaultní, pokud není rozsah explicitně určen. Compile závislosti jsou přidány do všech classpath a jsou zabaleny k aplikaci.
- Provided
    - Tuto závislost použijeme, pokud očekáváme, že je poskytne nějaká entita zvenčí, například JDK (Java Development Kit - prostředí pro vývoj v programovacím jazyce Java). Tyto závislosti budou v kompilační classpath, ale nebudou v runtime classpath a ani nebudou zabaleny k aplikaci (typický případ je vývoj webových aplikací, kde určité části programu nebudou

v souboru s příponou .war ale na aplikačním serveru nebo v kontejneru servletů).

- Runtime
    - Runtime závislosti jsou potřeba ke spuštění a testu systému, ale nejsou potřeba pro kompilaci.
- Test
    - Test závislosti nejsou potřeba při kompilaci ani při běhu programu, a jsou dosažitelné pouze při testové kompilaci a spuštění.
- System
    - System druh závislosti není obecně doporučován. Je podobný jako Provided závislost, s tím rozdílem že musíme také zadat přesnou cestu k .jar souboru v lokálním systému souborů. Jsou vždy dosažitelné a Maven je nevyhledává v žádné repository. Použít jej lze tehdy, pokud řešíme závislosti, které jsou nyní poskytovány v JDK, ale které byly dříve k disposici jako zvláštní downloady. (Pokud vyvíjíme projekt ve starší verzi Javy, je možné, že to, co je nyní součástí JDK, tehdy ještě nebylo a je to tedy nutné připojit externě.)

Kromě těchto základních druhů závislostí je možné použít i volitelné (optional) závislosti, kdy lze k projektu připojit jenom ten či onen artefakt, podle určitého nastavení. Například můžeme v našem programu nabídnout dva různé druhy logování a každý z nich je závislý na jiném logovacím artefaktu. Pak je oba musíme dát mezi ostatní závislosti, ale označíme je jako volitelné a potom při psaní jiného projektu, který používá náš projekt (je na něm závislý) explicitně zadáme jednu z těch dvou závislostí (podle druhu logování, který si zvolíme) a v balíku pak bude připojená jenom jedna potřebná z těch dvou volitelných závislostí.

3.5.1. Přenosné závislosti projektu

Maven umí snadno pracovat s přenosnými závislostmi, a tedy pokud je projekt A závislý na projektu B, který je zase závislý na projektu C, pak v projektu A stačí zadat závislost na projekt B, a Maven se sám postará o to, aby projekt B měl všechny své potřebné zdroje

přístupné (a to samé platí pro projekt C). Toho je docíleno tím, že Maven si vytváří graf závislostí a automaticky se stará o případné konflikty – pokud Maven vidí dva projekty které potřebují stejný artefakt, ale každý z nich požaduje jeho jinou verzi, automaticky použije tu novější z nich. Obvykle takové řešení funguje správně a ušetří vývojáři spoustu času a práce, ovšem můžou s tím vzniknout problémy, které je zase možné řešit vyloučením závislosti (dependency exclusion). Maven si tyto závislosti hlídá i v pořadí buildování projektů, takže pokud je projekt A závislý na projektu B, pak Maven ví, že jako první musí buildovat projekt B a toto funguje automaticky.

Tím je možné Mavenu říct, aby určitou přenosnou závislost ignoroval, nebo ji ignoroval a nahradil jinou, námi zadanou.

3.6.Životní cyklus projektu

Zatímco závislosti a POM soubor definují projekt, tak životní cyklus určuje, co s takovým projektem má vlastně Maven dělat. Cíle zabalené v Maveních pluginech (v originále goals) jsou svázány do fází a tvoří životní cyklus. Pokud spustíme určitou fázi, nebo určitý cíl nějaké fáze, Maven nejdříve spustí všechny fáze a cíle, které jí v životním cyklu předchází. Životní cyklus se tedy skládá z posloupnosti pojmenovaných fází jako například: připrav zdroje, kompiluj, zabal, instaluj a podobně. Existují fáze určující úkol, který se musí provést před kompilací nebo po určité jiné fázi. Je ovšem možné upravovat tyto existující životní cykly, nebo dokonce vytvářet zcela nové, vlastní. V Mavenu jsou tři standardní životní cykly: clean, default (také build) a site.

3.6.1. Clean

Tento životní cyklus je nejjednodušší. Spouští se příkazem *mvn clean* a má tyto tři fáze:

- Pre-clean
- Clean
- Post-clean

Základním kamenem tohoto životního cyklu je cíl (goal) clean ve fázi clean, což lze napsat jako clean:clean (fáze:cíl). Ten vymaže adresář s buildem projektu (defaultně jde o *${basedir}/target*. Pokud však nespustíme přímo tento cíl, ale celou fázi clean, umožníme

tím Mavenu aby nejdříve spustil všechny cíle fáze pre-clean (například vysvítit varování, že daný adresář bude smazán). Také lze upravit fázi clean, aby kromě souborů .jar smazala i soubory .class a podobně.

3.6.2. Build neboli Default

Toto je nejběžnější životní cyklus. Je to obvyklý model buildovacího procesu pro běžnou aplikaci. Celkem sestává z 21 fází počínaje *validate* a konče *deploy*. Jejich výčet pro tuto práci není podstatný, proto zde nebudou uvedeny. Důležité je však zmínit, že běžný příkaz při vývoji Maven aplikací – *mvn install* spouští právě tento životní cyklus až po předposlední cíl, *install*. Vynechá tedy pouze poslední cíl a to *deploy*.

3.6.3. Site

Tento životní cyklus je určený pro generování projektové dokumentace a zpráv a obsahuje celkem 4 fáze:

- Pre-site
- Site
- Post-site
- Site-deploy

Tento typ životního cyklu není závislý na druhu zabalení (viz kapitola 3.6.4) jako třeba defaultní životní cyklus, neboť tvoří dokumentaci nebo webovou stránku s určitým obsahem a proto to pro něj není relevantní.

3.6.4. Životní cykly v závislosti na druhu zabalení (packaging)

Životní cyklus default si v závislosti na druhu packagingu (například jestli výstupem má být .jar, nebo .war nebo třeba .pom) zvolí rozdílné cíle jednotlivých svých fází. Například projekt s packagingem POM spustí *site:attach-descriptor* během package fáze, zatímco JAR projekt místo toho spustí cíl *jar:jar*. Nejběžnější druhy zabalení projektu jsou například:

- JAR
- POM

- Maven Plugin
- EJB
- WAR
- EAR

Jsou ovšem i jiné, neboť každý projekt nebo plugin může mít vlastní druh výsledného zabalení, z těch běžných je například NAR (native archive), SWF a SWC pro projekty které generují obsah Adobe Flash a Adobe Flex.

3.6.5. Běžné cíle životního cyklu

Většina životních cyklů má společné určité cíle nebo fáze. Například cíle spojené s war nebo jar životními cykly jsou stejné až na jedinou fázi spojenou se zabalením, kde war životní cyklus volá *war:war*, zatímco jar životní cyklus volá *jar:jar*. Fáze pro management zdrojů, testování nebo kompilování zdrojového kódu jsou ovšem v těchto i v mnoha dalších případech společné. Některé z nich, jako například kompilování nebo zpracování zdrojů mají i test protějšky, které fungují ovšem téměř stejně, rozdíly jsou minimální, například ve výstupním adresáři.

3.6.5.1. Zpracování zdrojů

Tato fáze obvykle přemístí zdroje tak, aby byly k disposici přeloženému programu a aby byly případně dalšími fázemi přidány do balíku. Je však možné tento proces upravit, například změnit umístění pro takové soubory, nebo vyfiltrovat různé druhy souborů do různých adresářů (například obrázky do images a xml soubory do adresáře xml), nebo dokonce přidat určitý filtr který může takový soubor upravit nějakou proměnnou, nebo jej zkopírovat v závislosti na jeho obsahu i někam jinam, což se může hodit například tehdy, když vytváříme stejný projekt pro různé platformy (v kombinaci s buildovacími profily).

3.6.5.2. Kompilování

Tato fáze v defaultní formě volá cíl *compile:compile* a ten zkompiluje všechen zdrojový kód a výsledný bytecode zkopíruje do buildovacího adresáře. I ten lze ovšem upravovat podle potřeby, například určením verze zdrojového a cílového kompilačního pluginu.

3.6.5.3. Test

V této fázi většina životních cyklů spouští testovací cíl pluginu Surefire, což je Mavení testovací plugin. Ten v rámci svého defaultního chování vyhledá všechny třídy, jejichž název končí *Test a spustí je jako JUnit testy. Lze jej ovšem nakonfigurovat také, aby spouštěl TestNG testy. Po spuštění *mvn test* Surefire vygeneruje zprávy do zadaného adresáře, kde se objeví na každý test dva soubory: XML dokument obsahující informace o vykonání testu a textový soubor obsahující výstup testu. Pokud některý test selže, základní chování Surefire pluginu je, že ukončí práci Mavenu. Jde jej však nakonfigurovat, aby i po selhání testu pokračoval dál dalšími fázemi.

3.6.5.4. Install

Tato fáze pouze nainstaluje hlavní artefakt projektu (a veškeré jeho závislosti) do lokální repository. Je to jednoduchý proces, ovšem pro fungování Mavenu klíčový.

3.6.5.5. Deploy

Tato fáze uloží artefakt do vzdáleného Mavenního repository, a tento krok učiníme, pokud chceme vydat hotovou verzi projektu. Může to být velmi jednoduché, pouhé zkopírování souborů, nebo i poměrně komplikované, neboť nastavení této fáze musí často obsahovat informace pro naši autorizaci k zpřístupnění tohoto vzdáleného repository (může jít o pouhé jméno a heslo, nebo i třeba o veřejný klíč). Z toho důvodu nastavení této fáze nemůže být obsaženo v souboru POM (ten se ukládá do repository také), ale obvykle bývá v souboru nastavení konkrétního uživatele *settings.xml*, který se defaultně nachází v lokálním repository.

4. Archetyp nástroje Maven

Maven archetype je artefakt Mavenu, který lze připojit jako závislost do projektu. Jde o jakýsi návrhový vzor projektu, podle kterého je možné snadno vygenerovat kostru projektu. Je možné vytvořit vlastní archetyp – buď jej vytvořit z již existujícího projektu, nebo jej napsat ručně. Maven ovšem poskytuje několik základních archetypů pro vytvoření příkladu jednoduché aplikace J2EE, nebo pluginu do Mavenu, nebo projektu Maven atd. Je dokonce možné s pomocí archetypů rozšířit již existující projekt, například spuštěním archetypu vytvářejícího webovou stránku projektu, a takto lze archetypy kombinovat.

S jejich pomocí lze ušetřit značné množství času při vytváření nových projektů. Obzvláště vhodné je to při vývoji v organizaci, kdy se vytváří projekty podle určitých pravidel a s určitou strukturou. U větších projektů je často jejich nastavení a struktura celkem složitá a proto je velice výhodné si všechny tyto společné rysy projektů uchovat v archetypu a při vytváření nového projektu jej vygenerovat pomocí daného archetypu, přičemž lze mnoho věcí vložit do archetypu jako parametr, což ještě zvyšuje flexibilitu archetypů.

4.1. Postup vytvoření archetypu Maven

Vytváření vlastního archetypu je dosti přímočarý proces, který sestává z několika kroků:

1. Vytvoření nového projektu a souboru *pom.xml* pro artefakt archetypu
2. Vytvoření deskriptoru archetypu (jde o soubor *archetype-metadata.xml*, ve kterém jsou uložena metadata archetypu)
3. Vytvoření struktury souborů projektu a vzoru souboru *pom.xml*
4. Instalace archetypu a spuštění Archetype pluginu (který vytvoří kostru projektu dle specifikací v zadaném archetypu).

Alternativou k tomuto postupu je vygenerování archetypu příkazem „*mvn archetype:generate*“ a následnou úpravou adresáře *resources* a přeskočením postupu rovnou na krok č. 4.

5. Vytvoření vlastního archetypu

5.1. Výhody vlastního archetypu v rámci organizace

Pokud společnost vyvíjí několik velkých projektů, je rozhodně výhodné použít nějaký jednotný buildovací nástroj, například Maven. Jednotný způsob vývoje projektů ušetří čas a kapacity vývojářů, což se promítne ve výrazně nižších nákladech. Pokud je navíc buildovací proces prakticky automatický a není potřeba psát další kód aby bylo možné projekt zbuildovat, jde o další významnou úsporu práce a času. Další významné úspory lze dosáhnout tím, že všechny projekty vyvíjené danou společností budou mít stejnou strukturu a budou vyvíjeny stejným způsobem. Pokud si tedy vybereme například Maven či jiný podobný nástroj, je vhodné všechny projekty společnosti převést na Maven a používat jen jeden nástroj, pokud je to možné. Takový převod nemusí a zpravidla ani není jednoduchý a bývá časově náročný, ale z dlouhodobého hlediska se jistě vyplatí.

Vytvořit vlastní archetyp pro Maven v organizaci je další krok k unifikaci vývojových procesů. Takový archetyp dokáže vytvořit na základě určitých parametrů jednotný druh různých projektů s již zapracovanými pojmenováními, vývojovou kulturou společnosti, jednotným způsobem ukládání tříd zdrojového kódu, testovacího kódu a podobně. Potom si všechny nově vytvořené projekty budou poněkud podobné a bude snazší jejich vývoj a pokud nějaký vývojář přejde z jednoho projektu na druhý, velmi snadno se v něm zorientuje a bude moci začít dříve pracovat a vyvíjet.

5.2. Vytvoření obecného archetypu

Nejpřímočařejší postup jak vytvořit archetyp je napsat jej celý ručně. Archetyp sestává z těchto částí:

- Archetype Descriptor
- Prototypové soubory
- Prototyp POMu
- POM archetypu

Nejprve je nutné vytvořit základní projekt a *POM.xml* který bude obsažen v nově vytvořeném archetypu. Následující úryvek kódu je příklad jak může vypadat jednoduchý *POM.xml*. Na obrázku 5 je oficiální ukázka jak tento soubor vypadá.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>my.groupId</groupId>
  <artifactId>my-archetype-id</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
</project>
```

Obr. 5 - Ukázka POM.xml

zdroj: *5*

Tento příklad je sice pouze ukázka, ale stačí v něm vyplnit groupId, artifactId a version a je možné jej hned použít. Pravděpodobně ale bude nutné doplnit i další informace pokud takový archetyp bude třeba skutečně použít pro reálný projekt.

Další krok k vytvoření archetypu touto cestou je vytvoření deskriptoru archetypu. Deskriptor je soubor, který je povinně v tomto umístění: *src/main/resources/META-INF/maven/archetype.xml* (v nových verzích Mavenu se soubor jmenuje *archetype-metadata.xml*). Je to soubor, kam jsou zapsána metadata archetypů. Proto v něm musí být zapsány i údaje o nově vytvářeném archetypu. Takhle vypadá příklad záznamu v deskriptoru pro ukázkový archetyp rychlého startu (quickstart archetype) jak je uveden na webu Apache Maven Project (obrázek 6).

```
<archetype xmlns="http://maven.apache.org/plugins/maven-archetype-
plugin/archetype/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

  xsi:schemaLocation="http://maven.apache.org/plugins/maven-
  archetype-plugin/archetype/1.0.0
  http://maven.apache.org/xsd/archetype-1.0.0.xsd">

  <id>quickstart</id>

  <sources>

    <source>src/main/java/App.java</source>

  </sources>

  <testSources>

    <source>src/test/java/AppTest.java</source>

  </testSources>

</archetype>
```

Obr. 6 - Ukázka deskriptoru archetypu

zdroj: *5*

Ovšem tam kde je tag <id> je potřeba zadat archetypeId uvedený v *pom.xml* zakládaného archetypu. Tag <sources> udává umístění zdrojového kódu projektu, tag <testSources> zase umístění testovacích kódů projektu. Možných tagů je mnoho, můžeme přidat například řádku <allowPartial>true</allowPartial> která řekne že tento archetyp je možné spouštět i do již existujících projektů. Takto je ovšem možné specifikovat pouze konkrétní soubory v určitém umístění, nikoliv prázdné adresáře. Zde je struktura ukázkového quickstart archetypu (obrázek 7).

```
archetype
|-- pom.xml
`-- src
    `-- main
        `-- resources
            |-- META-INF
            |   `-- maven
            |       `--archetype-metadata.xml
            `-- archetype-resources
                |-- pom.xml
                `-- src
                    |-- main
                    |   `-- java
                    |       `-- App.java
                    `-- test
                        `-- java
                            `-- AppTest.java
```

Obr. 7 - Ukázka struktury archetypu

zdroj: *5*

Z této struktury je vidět, že archetype v sobě obsahuje i vytvářený projekt (v adresáři archetype-resources). Do souborů tohoto vytvářeného projektu lze místo napevno zadaných údajů zadat také proměnné, které je nutno deklarovat v souboru *archetype-metadata.xml*. Při generování projektu z takového archetypu se potom Maven na takové proměnné dotáže a nabídne zadanou defaultní hodnotu.

Jak je vidět, je poměrně snadné si takovýmto způsobem upravit strukturu projektu dle požadavků vývojové kultury organizaci či dle svých osobních preferencí.

Potom je už potřeba pouze spustit instalaci nového archetypu příkazem *mvn install* v umístění kde je *pom.xml* nového archetypu a ten se automaticky nainstaluje do lokálního repository počítače a je připraven k použití. Pro jeho použití pak je nutné spustit následující příkaz, který vygeneruje nový projekt podle vytvořeného archetypu, jak je možné vidět na obrázku č. 8.

```
vn archetype:generate                    \
 -DarchetypeGroupId=<archetype-groupId>          \
 -DarchetypeArtifactId=<archetype-artifactId>    \
 -DarchetypeVersion=<archetype-version>          \
 -DgroupId=<my.groupid>                  \
 -DartifactId=<my-artifactId>
```

Obr. 8 – Příkaz pro generování archetypu

zdroj: vlastní

Ze jen nahradíme výrazy v <> identifikačními údaji archetypu a identifikačními údaji nového projektu.

5.3. Vytvoření archetypu z již existujícího projektu

Druhá možnost jak vytvořit archetyp je o mnoho jednodušší, ale je k tomu potřeba nějaký již existující projekt. U takového projektu je potřeba na místě, kde se nachází jeho *pom.*xml spustit jednoduše příkaz *mvn archetype:create-from-project* a vygeneruje se nový archetyp. Ten ovšem obsahuje všecno nastavení, všechny soubory i třídy s kódem daného projektu, proto je potřeba projekt buď připravit před tvorbou archetypu, nebo nově vytvořený archetyp upravit a vyčistit od nežádoucích částí potom. Upravit je třeba obzvláště soubor *archetype.xml* nebo *archetype-metadata.xml* podle toho jakou verzi Mavenu používáme (ať je název souboru jakýkoliv, jde o deskriptor nově vytvořeného archetypu). Ovšem je možné že bude třeba i upravit třídy s kódem programu, podle toho zda je žádoucí aby archetyp vytvářel projekt i s určitým kódem, nebo třeba jen s prázdnými třídami připravenými k vepsání kódu.

Další možností jak si usnadnit tvorbu archetypu je vytvořit jej s pomocí properties souboru. Ten se může jmenovat libovolně, ale je vhodné ho umístit mimo adresářovou strukturu zdrojového projektu, ze kterého je nový archetype vytvářen. Jinak by takový properties soubor byl součástí archetypu. Do něj lze zadat identifikační údaje nového archetypu (groupId, artifactId, version), ale navíc i libovolné proměnné i s defaultními hodnotami. Příkaz pro vytvoření archetypu s pomocí properties souboru vypadá například takto:

mvn archetype:create-from-project -Darchetype.properties=../archetype.properties

V obrázku č. 9 je příklad, jak může vypadat properties soubor:

```
archetype.groupId=my.group.id
archetype.artifactId=archetype-with-properties
archetype.version=2.0

archetype.filteredExtensions=java
archetype.languages=groovy

an_additional_property=my specific value
```

Obr. 9 – Ukázka properties souboru

zdroj: vlastní

Jak je zřejmé, jde o naprosto běžný properties soubor. Záznamy které mají název archetype.<cokoliv> udávají parametry nového archetypu. Ostatní záznamy jsou budoucí

proměnné archetypu. Maven se je pokusí v kódu vyhledat a nahradit. To se bohužel vždy nepovede a proto i při vytváření archetypu z již existujícího projektu i s použitím properties souboru je nutná určitá práce s vytvořeným archetypem.

5.4. Práce s již hotovým archetypem

Základní podoba archetypu je po výše uvedeném postupu dokončena, je ovšem třeba s takovým archetypem pracovat tak, aby zcela splnil svůj zamýšlený účel a aby jeho použití poskytlo maximální možnou úsporu práce a času při jeho použití. Tím je myšlena jeho co největší univerzálnost, aby bylo možné jej použít u co nejvíce nových projektů a samozřejmě je nutné, aby takový archetype vytvořil nový projekt co nejlépe a nejpřesněji, aby následná práce s projektem byla co možná nejjednodušší a co nejvíce věcí bylo vytvořeno a nastaveno archetypem.

Jedou z věcí, které je nutné zkontrolovat případně nastavit jsou proměnné. Jako proměnná jsou samozřejmě vždy nastavené identifikační údaje nového projektu, ale lze tak nastavit i jiné údaje. V obrázku č. 10 je možné vidět nastavené proměnné v souboru *pom.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>


 <groupId>${groupId}</groupId>
 <artifactId>${artifactId}</artifactId>
 <version>${version}</version>
 <packaging>jar</packaging>

 <name>testovaci</name>
 <url>http://maven.apache.org</url>

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

 <dependencies>
  <dependency>
   <groupId>${test_dependencyGroup}</groupId>
   <artifactId>${test_dependencyArt}</artifactId>
   <version>${test_dependencyVersion}</version>
   <scope>test</scope>
  </dependency>
 </dependencies>
</project>
```

Obr. 10 – Ukázka proměnných v pom.xml

zdroj: vlastní

Tučně jsou zvýrazněny proměnné, přičemž text v závorkách je jméno proměnné. Vidíme že identifikace projektu je jako proměnná, ale také je možné vidět že máme proměnnou zadanou groupId, artifactId a verzi jedné závislosti.

Dále je nutné v souboru *archetype-metadata.xml* takové proměnné deklarovat. Na obrázku 11 je ukázka, jak je možné toho docílit:

```
<?xml version="1.0" encoding="UTF-8"?>
<archetype-descriptor
xsi:schemaLocation="http://maven.apache.org/plugins/maven-archetype-
plugin/archetype-descriptor/1.0.0 http://maven.apache.org/xsd/archetype-
descriptor-1.0.0.xsd" name="testovaci"
    xmlns=http://maven.apache.org/plugins/maven-archetype-plugin/archetype-
descriptor/1.0.0
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <requiredProperties>
    <requiredProperty key="test_dependencyArt">
      <defaultValue>junit</defaultValue>
    </requiredProperty>
    <requiredProperty key="test_dependencyVersion">
      <defaultValue>3.8.1</defaultValue>
    </requiredProperty>
    <requiredProperty key="test_dependencyGroup">
      <defaultValue>junit</defaultValue>
    </requiredProperty>
  </requiredProperties>

  <fileSets>
    <fileSet filtered="true" packaged="true" encoding="UTF-8">
      <directory>src/main/java</directory>
      <includes>
        <include>**/*.java</include>
      </includes>
    </fileSet>
    <fileSet filtered="true" packaged="true" encoding="UTF-8">
      <directory>src/test/java</directory>
      <includes>
        <include>**/*.java</include>
      </includes>
    </fileSet>
  </fileSets>

</archetype-descriptor>
```

Obr. 11 – Ukázka deklarace proměnných v archetype-metadata.xml

zdroj: vlastní

Zde je tučně zvýrazněná deklarace proměnné i její defaultní hodnota. Je vidět, že v tomto souboru je deklarovaná i struktura výsledného projektu. Zajímavá a zajisté velmi užitečná vlastnost je, že i do této struktury lze zadat proměnnou, takže je například možné, aby se adresář obsahující zdrojový kód jmenoval podle názvu projektu, což je obzvláště užitečné, pokud máme větší projekt obsahující několik modulů, což je velmi časté například u Enterprise projektů psaných v Javě EE. Toho lze dosáhnout toho, že do *archetype-metadata.xml* do tagu <directory> vložíme odkaz na proměnnou, v tomto případě ale ne ve

tvaru *${proměnná}*, ale takováto proměnná musí být mezi dvěma podtržítky, takto: *__proměnná__*.

5.5. Dokončení archetypu

V novém archetypu Maven jsou tedy všechny požadované proměnné deklarovány, mají nastavené defaultní hodnoty a jsou správně umístěné v kódu. Dále je připravena struktura nového projektu a výchozí závislosti. Je také nastaven požadovaný výchozí zdrojový kód a testovací kód (může jít o určitou společnou základní funkčnost, ale i třeba o pouhý „Hello world!“ program). V takové chvíli je dalším krokem nainstalování archetype jako jakýkoliv jiný artifact Maven přes příkaz *mvn install*. V tom okamžiku se archetype nainstaluje do lokálního repository Maven a je připraven k použití.

V adresáři, kde je třeba vytvořit nový projekt z vytvořeného archetype je možné zadat příkaz *mvn archetype:generate*, který nabídne všechny nalezené archetypy jak v repository Mavenu, tak v našem lokálním repository, a příslušným číslem vybereme námi vytvořený archetype. Je ovšem také možné vytvořit projekt přímo, jediným příkazem, například jak je vidět na obrázku 12:

```
mvn archetype:create -DgroupId=example -DartifactId=myproject
-DarchetypeArtifactId=tellurium-junit-archetype -
DarchetypeGroupId=tellurium
-DarchetypeVersion=1.0-SNAPSHOT
-DarchetypeRepository=<URL nebo jiné umístění repository ve které se
archetype nachází>
```

Obr. 12 – Ukázka přímého vytvoření projektu z archetypu

zdroj: vlastní

Je možné takovýmto způsobem zadat všechny potřebné informace jediným příkazem, konkrétně tedy groupId a artifactId nově vytvářeného projektu, a groupId, artifactId, verzi a umístění zdrojového archetypu. Pokud jsou zadané v archetypu některé požadované parametry, je nutné je zadat, případně potvrdit použití defaultních hodnot a projekt je vytvořen, lze začít rovnou kódovat.

Závěr

Cílem této bakalářské práce bylo vytvořit takový archetype nástroje Maven, který by bylo možné použít pro vytváření více různých projektů majících určitou podobnou strukturu, například pro vytváření více projektů uvnitř jedné organizace. Za tímto účelem bylo nejprve nutné v teoretické části práce představit programovací jazyk Java Enterprise Edition, jeho strukturu a odlišnost od Javy Standard Edition. Dále bylo samozřejmě nutné představit nástroj Apache Maven od Apache Software Foundation. K čemu je určen a jak funguje. Nakonec bylo nutné představit samotný archetype, plugin do nástroje Apache Maven.

V praktické části práce potom už jsou řešeny různé způsoby vytvoření archetypu. Jak takový archetype vytvořit celý ručně, a jak jej vygenerovat z již existujícího projektu. A na samotný závěr práce i jak vytvořený archetype je možné upravovat, jak je možné do něj vložit proměnné, díky nimž pak při tvorbě projektu z archetypu je možné vytvořit přímo projekt, se kterým se již dá bez větších úprav pracovat.

Postup je uveden schválně obecný, aby bylo zřejmé, že takovýmto způsobem je možné vytvořit archetype pro libovolnou organizaci, pro libovolný účel.

Nově vytvořený archetype bezesporu značně usnadní vytváření nových projektů v libovolné organizaci. Z toho plyne poměrně výrazná úspora nákladů neboť čas vývojáře, který stráví konfigurací nově vytvářeného projektu spadá do režijních nákladů organizace a těžko jej lze vyúčtovat zákazníkovi. O jak velkou úsporu jde lze těžko odhadnout, neboť společnost vyvíjející často větší množství nových projektů bude mít velmi značné ekonomické výhody plynoucí z vlastního archetypu, zatímco společnost vyvíjející jeden či dva projekty které neustále zdokonaluje nemusí mít prakticky žádnou ekonomickou výhodu plynoucí z takového archetypu. Zajisté se vyplatí provést analýzu průměrných nákladů spojených se započetím všech nových projektů v organizaci za určité období a pak je potenciální úspora zřejmá.

Seznam použité literatury:

1. ALUR D. CRUPI J. MALKS D. Core J2EE Patterns: Best Practices and Design Strategies, Prentice Hall PTR 2003, edition 2, ISBN-13: 978-0131422469
2. TIMOTHY M. O'BRIEN T. CASEY J. Brian FoxSonatype a kolektiv autorů, Sonatype, Inc. 2010, ISBN-13: 978-0-9842433-1-0
3. O'BRIEN T. Maven: The Definitive Guide, Sonatype, Inc. 2008, ISBN-13: 978-0596517335
4. Maven: The complete reference. *Sonatype.com.* [online]. [cit. 2013-04-15]. Dostupné z: http://www.sonatype.com/books/mvnref-book/reference/
5. Apache Maven project guides. *Apache software foundation.* [online]. [cit. 2013-04-15]. Dostupné z: http://maven.apache.org/guides

Seznam citací

[1] PIERCE M. a kol. Open community development for science gateways with apache rave. In: *GCE '11 Proceedings of the 2011 ACM workshop on Gateway computing environments.* New York, NY, USA, 2011. ACM 2011, s.29-36 ISBN 978-1-4503-1123-6.

[2] KÜHNER G. kol. *Progress on standardization and automation in software development on W7X* In: *Fusion Engineering and Design.* Prosinec 2012, Volume 87, Issue 12, s. 2232-2237. ISSN (2012) 2232 - 2237.

[3] FUHRMAN CH. a kol. *Integrating Tools and Frameworks in Undergraduate Software Engineering Curriculum.* In: *ICSE2012 Proceedings of the 2012 International Conference on Software Engineering*. IEEE Prass Piscataway, NJ, USA: 2012. ISBN 978-1-4573-1067-3.