VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PODPORA PRO UŽITÍ JAZYKA PYTHON PRO VÝVOJ ZÁSUVNÝCH MODULŮ SERVERU JENKINS

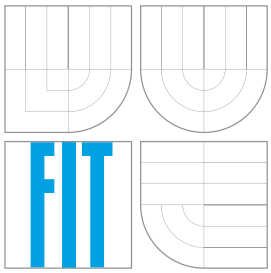
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ BAMBAS

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PODPORA PRO UŽITÍ JAZYKA PYTHON PRO VÝVOJ ZÁSUVNÝCH MODULŮ SERVERU JENKINS

SUPPORT FOR JENKINS PLUGIN DEVELOPMENT IN PYTHON LANGUAGE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. TOMÁŠ BAMBAS
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR MÜLLER
SUPERVISOR

BRNO 2014

Abstrakt

Server pro průběžnou integraci Jenkins CI umožňuje rozšiřovat svou funkcionalitu pomocí zásuvných modulů. Tyto moduly lze programovat v jazycích Java a Ruby. Podpora pro jazyk Python chybí, přestože se jedná o jeden z nejpopulárnějších programovacích jazyků současnosti. Implementovali jsme proto vývojářské nástroje, které umožňují programovat moduly v jazyce Python a tyto nástroje jsme začlenili do projektu Jenkins CI. K nástrojům byla zveřejněna uživatelská dokumentace. Programátoři mohou teď díky této práci implementovat moduly do Jenkins CI v jazyce Python.

Abstract

Jenkins CI, the continuous integration server, enables to extend its functionality by plug-ins. These plug-ins can be written in Java and Ruby. The support for Python language is missing although it is one of the most popular programming languages. Therefore we have implemented the SDK for Python plug-in development and this SDK has been integrated into the Jenkins CI community repository. The documentation for the plug-in development in Python has been also published. Thanks to that work, developers can now implement plug-ins in Python programming language.

Klíčová slova

InstallShield, Jenkins, Jython, průběžná integrace, Python, Ruby, vývojářský nástroj, zásuvný modul

Keywords

continuous integration, InstallShield, Jenkins, Jython, plug-in, Python, Ruby, SDK

Citace

Tomáš Bambas: Podpora pro užití jazyka Python pro vývoj zásuvných modulů serveru Jenkins, diplomová práce, Brno, FIT VUT v Brně, 2014

Podpora pro užití jazyka Python pro vývoj zásuvných modulů serveru Jenkins

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval zcela samostatně pod vedením pana Ing. Petra Müllera. Další informace mi poskytl technický konzultant Ing. Vojtěch Juránek.

.....
Tomáš Bambas
18. května 2014

Poděkování

Rád bych poděkoval svému vedoucímu, Ing. Petru Müllerovi, za pomoc s praktickými věcmi a technickému konzultantovi, Ing. Vojtěchu Juránkovi, za věcné připomínky, rady a nápady.

Obsah

Kapitola 1

Úvod

Průběžná integrace je jedním z moderních přístupů ve vývoji softwaru. Jedná se o metodu vývoje softwaru, která klade důraz na průběžné začleňování a testování kódu všech vývojářů. Programovou podporou průběžné integrace jsou servery zajišťující činnosti průběžné integrace, které se dají automatizovat. Jedním z nejznámějších serverů je i Jenkins CI. Jenkins je naprogramován v jazyce Java a je také zaměřen hlavně na vývoj softwaru v tomto jazyce. O pojmu průběžná integrace a serveru Jenkins pojednává kapitola 2.

Systém Jenkins podporuje instalaci a užití zásuvných modulů. Ve skutečnosti je většina funkcionality nad softwarovými projekty prováděna právě zásuvnými moduly. Popisem systému zásuvných modulů v Jenkins CI se zabývá kapitola 3.

Vývoj modulů probíhá v Jazyce Java, za použití programu Maven, nástroje pro správu softwarových projektů v tomto jazyce. Dále jsou k dispozici nástroje pro podporu vývoje modulů v jazyce Ruby. O vývoji modulů v jazycích Java a Ruby pojednává kapitola 4.

Kapitola 5 popisuje návrh nástrojů pro podporu vývoje zásuvných modulů v jazyce Python. Návrh vychází ze získaných znalostí o vývoji modulů v jazycích Java a Ruby a poskytuje návod pro implementaci nových nástrojů.

Kapitoly 6–8 pak dokumentují implementaci navržených nástrojů a modulů včetně způsobu jejich testování.

Poslední kapitola 9 popisuje zveřejnění jednotlivých součástí práce včetně uživatelské dokumentace a shrnuje celkové výsledky, jichž bylo dosaženo.

Kapitoly 2–5 byly převzaty ze semestrálního projektu vypracovaném na Fakultě informačních technologií a odevzdaném v lednu 2014.

Kapitola 2

Jenkins

Obsah této práce se bezprostředně týká systému Jenkins, serveru pro průběžnou integraci. Z tohoto důvodu je třeba popsat praktický účel systému, jeho funkcionalitu a architekturu.

2.1 Průběžná integrace

Průběžná integrace (angl. continuous integration) je praxe ve vývoji softwaru, kdy každý vývojář průběžně začleňuje změny, které provedl v kódu, do centrálního repozitáře, alespoň jednou denně, a tyto změny jsou automaticky otestovány a v případě nalezených chyb je vývojáři oznámena tato skutečnost. Důležitý je fakt, že je v tomto případě softwarový produkt překládán centrálně na takzvaném „build serveru“ a tato činnost, která bývá u velkých softwarových projektů časově náročná, je tedy delegována na výkonný server. Součástí praktik softwarové integrace není pouze průběžné zavádění změn a provádění testů (např. jednotkových testů nebo integračních testů) nad produktem, ale také průběžné dodávání spustitelných verzí testerům pro komplikovanější testy, případně koncovým uživatelům nebo zákazníkovi. Hlavní výhoda průběžné integrace tedy spočívá v delegaci časově náročných překladů a testů výkonnějšímu stroji, dále ve včasném odhalování chyb v kódu a v průběžné zpětné vazbě od zákazníka, popřípadě koncových uživatelů. [16]

2.2 Systémy pro průběžnou integraci

Průběžná integrace je z velké části automatický proces a tuto automatickou část, nebo spíše podporu průběžné integrace zajišťují specializované programy, tzv. servery pro průběžnou integraci (angl. CI servers). Hlavním úkolem těchto serverů je sledovat změny v kódu v repozitářích systémů pro správu verzí (např. SVN nebo Git), překládat pomocí různých nástrojů (make, ant, bash atd.) vyvíjený produkt na základě nalezených změn v kódu nebo podle časového rozvrhu, testovat zdrojový kód, popřípadě již přeloženou spustitelnou verzi a poskytovat poslední úspěšně přeložený produkt různým zájmovým skupinám (např. testerům). Jako další funkce těchto serverů můžeme uvést notifikace o očekávaných i neočekávaných událostech (úspěšné či neúspěšné překlady či testy) různými komunikačními kanály a síťovými protokoly (SMTP, XMPP, RSS), možnou integraci v některých IDE (Eclipse, Visual Studio) nebo distribuci prováděných úkonů na další build servery.

2.3 Jenkins CI

Jenkins CI[1], nebo zkráceně Jenkins, je server pro průběžnou integraci napsaný kompletně v jazyce Java a je vyvíjen jako komunitní projekt pod svobodnou licencí MIT[2]. Jenkins je multiplatformní aplikace a může být spouštěn na jakémkoliv operačním systému podporujícím běh JRE verze 6 nebo vyšší. Jenkins se chová jednak jako java servlet, což znamená, že může být spuštěn v rámci jakéhokoliv Java aplikačního serveru (např. Jetty nebo Tomcat), a jednak sám obsahuje zabudovaný aplikační server Winstone, a tudíž může být spuštěn samostatně, využívaje právě tento server.

Zdrojové kódy projektu Jenkins jsou umístěny na serveru GitHub a správa verzí je tedy logicky vykonávána nástrojem Git. Dále je nutné zmínit, že se projekt Jenkins odštěpil roku 2011 od projektu Hudson. Oba projekty jsou nyní vyvíjeny souběžně. Některé třídy nebo balíčky ve zdrojových kódech projektu Jenkins proto stále obsahují ve svém názvu, kvůli zpětné kompatibilitě, slovo Hudson.

2.3.1 Uživatelské rozhraní

Uživatelské rozhraní systému Jenkins je primárně webové; standardně, pokud není určeno jinak, se Jenkins naváže na TCP port 8080, přes který je možno přistupovat k jeho webovému rozhraní. Dále je možno Jenkins ovládat pomocí konzolové aplikace Jenkins CLI[3], která ovšem také komunikuje se serverem přes uvedené webové rozhraní. Na obrázku 2.1 je vykreslena úvodní obrazovka uživatelského rozhraní. Vlevo nahoře můžeme vidět hlavní menu, kde se nachází například odkaz do konfigurační nabídky nebo odkaz pro přidání nové úlohy (viz dále). Největší část obrazovky, napravo, zabírá seznam nakonfigurovaných úloh z nějakého pohledu (zde konkrétně: všechny úlohy). Vlevo uprostřed se nachází výčet sestavení (viz dále), která čekají na své provedení ve frontě, a konečně vlevo dole jsou aktuálně prováděná sestavení.

Obrázek 2.1: Úvodní obrazovka uživatelského rozhraní serveru Jenkins.

[1] Jenkins CI – http://jenkins-ci.org
[2] The MIT License (MIT) – http://opensource.org/licenses/mit-license.php
[3] Jenkins CLI – https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+CLI

2.3.2 Úlohy

Úlohy (angl. Jobs), nebo také projekty, jsou hlavními objekty, se kterými server a uživatel operují. Jednotlivé úlohy jsou definovány umístěním zdrojových kódů (např. přístupem do repozitáře nějakého systému pro správu verzí), spouštěčem sestavení (např. podle nějaké časové frekvence nebo změnou kódu v repozitáři), kroky sestavení (angl. build steps), které mohou být například vykonáním nějakého skriptu nebo sestavovacího nástroje, jako je make nebo ant, dodatečných akcí, které mají být vykonány po překladu (např. notifikace vývojářům nebo zveřejnění výsledků JUnit testů), a dalším nastavením.

Pokud je úloha nakonfigurována a povolena a je aktivován její spouštěč, jsou provedeny automaticky všechny kroky sestavení a dodatečné akce. Jenkins dále indikuje, zda úloha skončila s úspěchem či bez a s touto skutečností dále pracuje. Typické provedení úlohy sestává ze stažení aktuální verze zdrojových kódů z repozitáře do lokálního pracovního prostoru, překladu zdrojových kódů do spustitelné podoby, provedení testů a rozeslání notifikací.

2.3.3 Další funkcionalita

Jenkins nabízí standardně, bez dodatečných zásuvných modulů, převážně nástroje pro práci s jazykem Java. Pro sestavení projektu lze užít nástrojů ant nebo maven, které se typicky zaměřují na Javu. Jednotlivé kroky sestavení lze také popsat shell skriptem (v případě unixových OS) nebo dávkou (v případě Windows OS). Mezi akce, které lze vykonat po sestavení, patří zveřejnění výsledků JUnit testů a Javadoc dokumentace, spuštění jiné úlohy (takto lze řetězit úlohy) nebo zaslání e-mailů definovaným příjemcům.

Jenkins dále umožňuje prohlížet pracovní prostor úloh přes webové rozhraní (pokud nemá uživatel přímý přístup k serveru, tak se k souborům ani jinak nedostane), spravovat zásuvné moduly (instalovat, odstraňovat) za běhu systému, zobrazovat zátěžové statistiky a logovací výstup, spravovat uzly (pomocné build servery), spouštět Groovy[4] skripty nad instancí serveru nebo spravovat uživatele pomocí lokální či externí databáze včetně jejich práv. Jenkins samozřejmě nabízí spoustu dalších funkcí, jejichž popis je však nad rámec této práce. Přestože se Jenkins ve standardní instalaci zaměřuje na jazyk Java, obrovské množství dodatečné funkcionality lze získat instalací nejrůznějších zásuvných modulů.

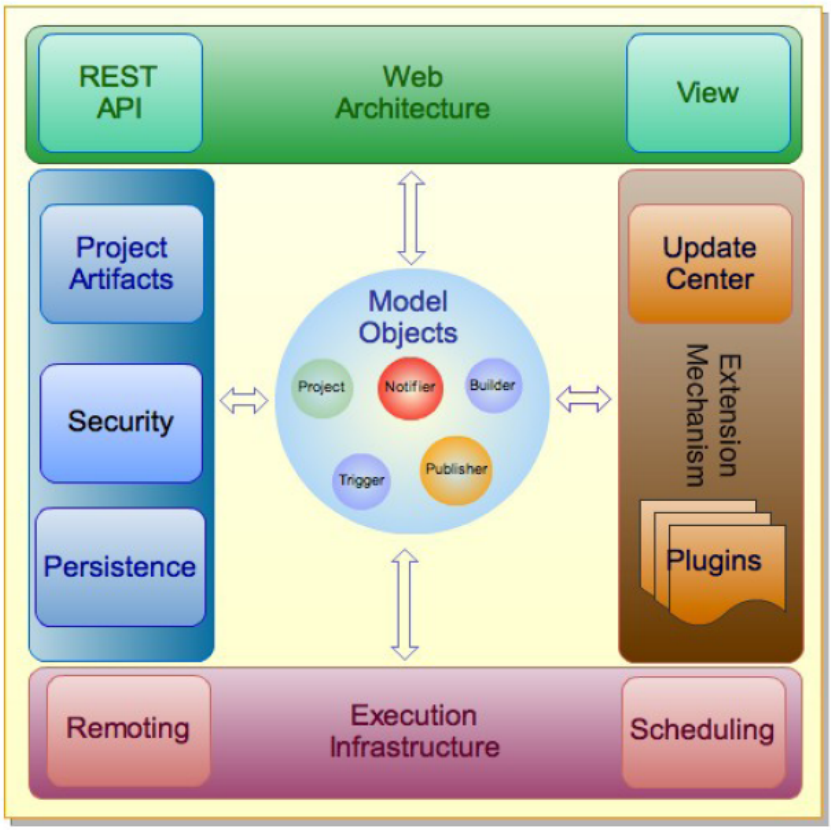
2.3.4 Architektura serveru Jenkins

Jenkins je rozšiřitelná webová aplikace, nelze ji však označit za informační systém. Přehled architektury je znázorněn na obrázku 2.2. Jádrem aplikace je objektový model, přičemž kořenovým objektem je objekt `Hudson`. Funkční metody jsou součástí objektů, nejedná se proto o doslovnou MVC (z angl. Model View Controller) architekturu. [14]

Architektura kromě objektového jádra dále obsahuje infrastrukturu, která zajišťuje plánování a vykonávání nakonfigurovaných a povolených úloh v systému, včetně správy vzdálených podřízených strojů, kterým lze delegovat práci na prováděných úlohách. Jenkins se může chovat jako distribuovaný systém, v tom případě vystupuje instance Jenkinse jako hlavní (angl. master) uzel, který vzdáleně nebo lokálně ovládá podřízené (angl. slave) uzly. Komunikace mezi uzly funguje na bázi TCP/IP protokolu, podřízené uzly se tedy mohou nacházet kdekoliv v síti nebo na stejném stroji jako hlavní uzel. [12]

Další důležitou částí aplikace je webová nebo také view architektura. Každá stránka webového rozhraní je pevně navázána na nějaký objekt z instance aplikace. Například koře-

[4] Groovy – http://groovy.codehaus.org



Obrázek 2.2: Architektura serveru Jenkins. Zdroj: [12].

nová stránka „/“ je pevně navázána na kořenový objekt `Hudson`. O navázání URL požadavků na příslušné objekty v instanci aplikace se stará framework Stapler. Kupříkladu HTTP požadavek „`POST /project/jaxb/testResult HTTP/1.1`“ je vyhodnocen následovně:

1. Na kořenový singleton třídy `Hudson` je zavolána metoda `getProject(,,jaxb‘‘)`.
2. Na vrácený objekt reprezentující úlohu je zavolána metoda `getTestResult()`.
3. Získané view obsahující výsledky testů je vráceno prohlížeči.

Samotné pohledy (angl. views) jsou definovány ve speciálním formátu jelly[5]. Jedná se o XML formát popisu uživatelského rozhraní. Tyto soubory, typicky uložené ve složce `resources` projektu Jenkins nebo některého ze zásuvných modulů, jsou dynamicky převáděny na HTML stránky. [15]

Velmi důležitou součástí architektury jsou prvky zajišťující správu a běh zásuvných modulů. Tato funkcionalita je podrobně popsána v kapitole 3.

Mezi další stavební prvky aplikace patří například součásti obstarávající zabezpečení serveru a práva uživatelů nebo prvek zajišťující serializaci objektů, což umožňuje uchovávání nejrůznějších stavů a nastavení. Popis dalších prvků architektury je nad rámec této kapitoly.

[5] Jelly: Executable XML – http://commons.apache.org/proper/commons-jelly/

2.3.5 Maven

Projekt Jenkins stejně tak jako všechny jeho zásuvné moduly využívá Maven[6], nástroj pro správu a sestavování projektů v Javě. Maven je obdoba známějšího nástroje pro překlad programů v Javě, nástroje Ant[7]. Cílem projektu Maven je zjednodušit proces překladu aplikace, poskytnout jednotný systém pro překlad softwaru, poskytovat užitečné informace o projektu, poskytovat návod pro osvědčené postupy ve vývoji softwaru a umožnit průhlednou migraci k novým funkcím. [17]

Maven umožňuje snadno definovat závislosti projektu na jiných projektech a automaticky přibalit všechny závislosti do výsledného balíčku při překladu. K tomuto účelu je třeba pouze definovat závislosti v definici projektu a repozitáře, kde lze tyto závislosti nalézt. Systém Jenkins je velice komplexní aplikace a je závislý na spoustě jiných knihoven a aplikačních prostředích. Z tohoto důvodu je pro Jenkins Maven nejlepším řešením.

Vlastnosti projektu spravovaného nástrojem Maven jsou definovány v souboru `pom.xml`. Tento soubor by měl být umístěn v kořenovém adresáři projektu. Akronym POM znamená „Project Object Model“ a obsahem souboru je intuitivní popis projektu ve formátu XML. Typickými složkami, které se dále nachází v kořenovém adresáři projektu, jsou `src` se zdrojovými kódy projektu a `resources` s dalšími soubory, které by měly být součástí výsledného balíčku. V příkladu 2.1 se nalézá jednoduchá ukázka souboru `pom.xml`.

Příklad 2.1

```
<project xmlns=,,...‘‘ xmlns:xsi=,,...‘‘ xsi:schemaLocation=,,...‘‘>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>
  <name>Simple Application</name>
  <url>http://mycompany.com</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Tento soubor nám říká, že projekt patří do balíčku (skupiny) `com.mycompany.app`, identifikační jméno projektu je `my-app`, verze aplikace je `1.0`, výsledným balíčkem je formát `JAR` a aplikace je závislá na balíčku `junit`, který se nachází v centrálním repozitáři projektu Maven.

Maven se ovládá pomocí takzvaných cílů a fází. Fáze jsou pouze pokrytím několika cílů, přičemž to, které cíle fáze začleňuje, záleží na nastavení projektu. Například příkaz „`mvn package`“ říká, že je třeba přeložit a zabalit projekt. Pokud je položka `packaging` nastavena na `jar`, vytvoří se v této fázi soubor ve formátu JAR.

[6] Apache Maven – http://maven.apache.org/
[7] Apache Ant – http://ant.apache.org/

Maven umožňuje implementovat vývojářům zásuvné moduly rozšiřující jeho funkcionalitu. Použití konkrétního zásuvného modulu určíme prefixem odděleným od cíle nebo fáze dvojtečkou. Například příkaz „`mvn archetype:generate`“, říká, že má Maven spustit cíl `generate` pomocí standardního zásuvného modulu `archetype`. Tento příkaz pouze vytvoří v pracovním adresáři šablonu pro nový projekt. Projekt Jenkins využívá vlastního zásuvného modulu do aplikace Maven pro snadný vývoj svých zásuvných modulů. Užití tohoto modulu je popsáno v kapitole 4.

Kapitola 3

Zásuvné moduly v systému Jenkins

Tato kapitola popisuje význam zásuvných modulů v systému Jenkins, architekturu a způsob jejich začlenění.

3.1 Zásuvný modul

Zásuvný modul (angl. plugin) je software, který nepracuje samostatně, ale pouze rozšiřuje funkcionalitu nějakého jiného softwaru. Zásuvný modul má tedy význam pouze v rámci nějaké aplikace, pro niž byl napsán. Mnoho aplikací poskytuje aplikační rozhraní pro psaní zásuvných modulů a umožňuje tak programátorům snadno vytvářet dodatečnou funkcionalitu, aniž by se přímo účastnili vývoje jádra aplikace.

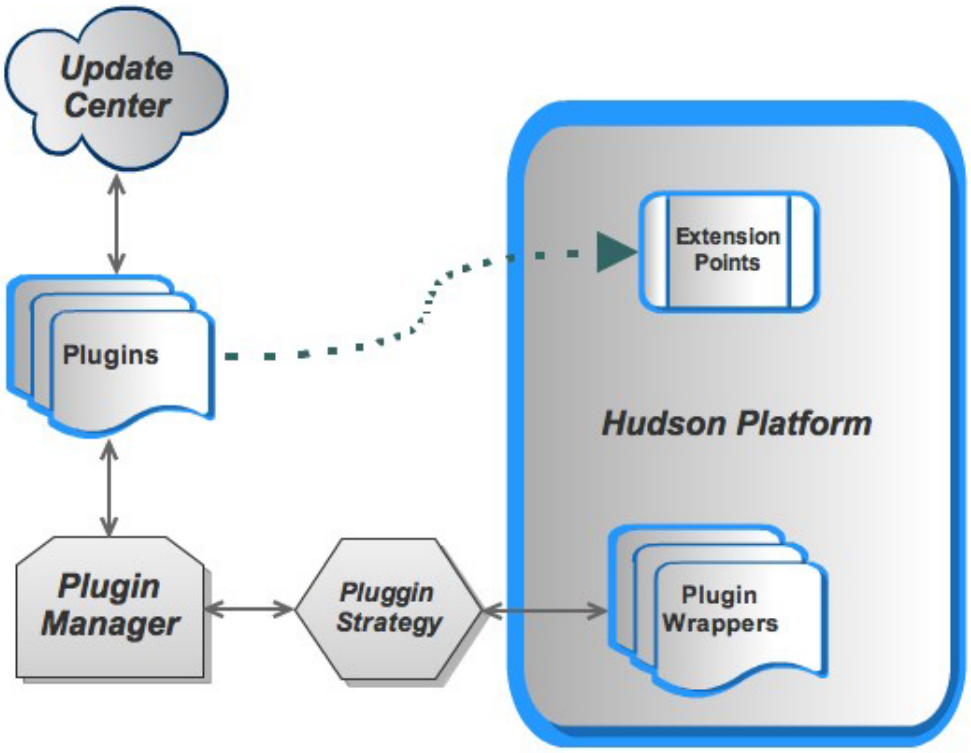
Zásuvné moduly poskytují dvě zásadní výhody oproti začlenění funkcionality do jádra aplikace. Modularita softwaru je výhodná jednak pro vývojáře aplikace, a to z toho důvodu, že se všelijaká funkcionalita vyvíjená vně hlavní jádro aplikace nestává přítěží a neznečišťuje architekturu softwaru, a jednak pro uživatele, protože jim nevnucuje veškerou možnou funkcionalitu, ale dává jim pouze volbu doinstalovat ji a přizpůsobit si tak aplikaci ke své potřebě.

3.2 Zásuvné moduly v Jenkins CI

Jenkins nabízí jednoduché aplikační rozhraní pro vývoj zásuvných modulů v Javě. Komunita a firmy kolem projektu Jenkins se aktivně podílejí na vývoji mnoha modulů, které rozšiřují jeho funkcionalitu. V době psaní tohoto textu jich bylo k dispozici více než 1000. Ve skutečnosti většinu práce v systému Jenkins odvádějí právě moduly. Například i práci se Subversion repozitáři nebo překlad projektů pomocí Maven a Ant nástrojů zajišťují moduly, které jsou dodávány společně s instalací serveru Jenkins. Mezi typické činnosti, které zajišťují moduly, patří překlad projektů, notifikace, spouštění překladů, sestavování zpráv o překladech, správa zdrojového kódu nebo například správa vzdálených překladových (angl. build) serverů. [6]

Architektura zásuvných modulů v systému Jenkins je znázorněna na obrázku 3.1. Skládá ze tří základních komponent:

1. Manažer zásuvných modulů (angl. Plugin Manager)
2. Centrum aktualizací (angl. Update Center)



Obrázek 3.1: Architektura zásuvných modulů v aplikaci Jenkins. Zdroj: [13].

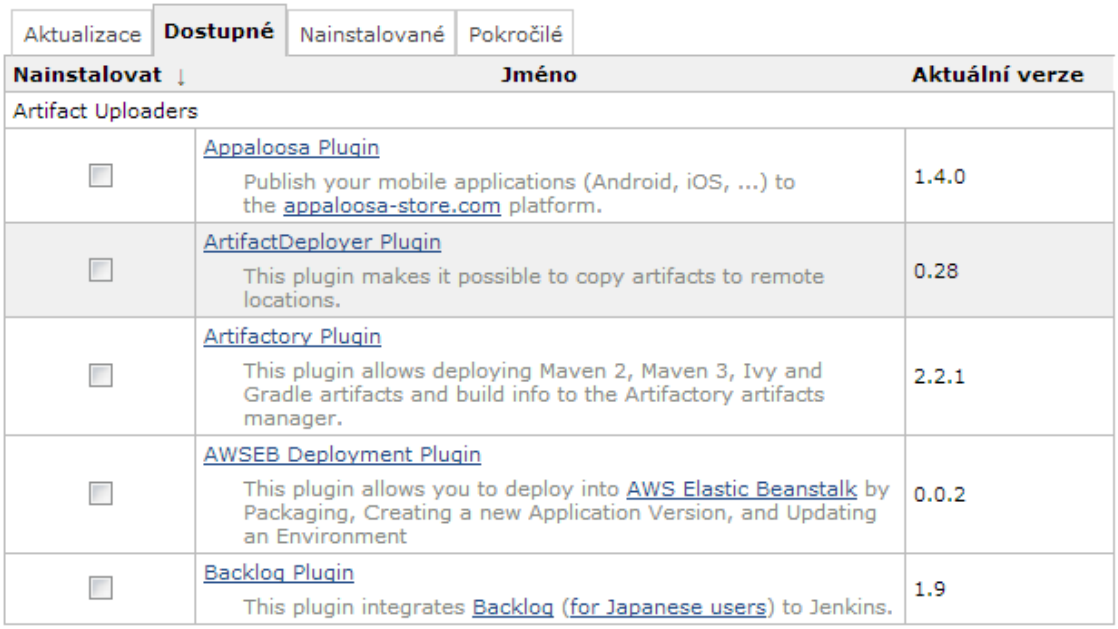
3. Obálka modulu a Strategie modulu (angl. Plugin Wrapper & Plugin Strategy)

3.2.1 Manažer zásuvných modulů

Manažer zásuvných modulů, který je implementován v třídě `PluginManager`, je hlavní řídící jednotkou, která se stará o zásuvné moduly. Je to služba jádra, která je zodpovědná za načítání modulů přibalených k základnímu balíku stejně jako modulů z aktualizačního centra. Manažer umožňuje instalaci modulů, jejich aktualizaci a také konfiguraci centra aktualizací nebo nastavení připojení přes proxy server. Manažer dále uchovává seznam nainstalovaných, dostupných a nefunkčních modulů. Snímek uživatelského rozhraní manažera se nachází na obrázku 3.2. Rozhraní obsahuje čtyři záložky. V první záložce je seznam dostupných aktualizací, druhá a třetí záložka obsahují seznam nainstalovaných a dostupných (instalovatelných) modulů. Poslední záložka zprostředkovává možnosti nastavení centra aktualizací. Přístup k uživatelskému rozhraní manažera zásuvných modulů je možný přes odkazy `[Administrace]/[Spravovat pluginy]` z hlavní nabídky webového rozhraní. Manipulovat s moduly mohou pouze uživatelé s příslušnými právy. [13]

3.2.2 Centrum aktualizací

Schéma centra aktualizací se nachází na obrázku 3.3. Centrum aktualizací reprezentují dva objekty (třídy) v modelu objektů systému Jenkins, a to `UpdateCenter` a `UpdateSite`. Tyto objekty poskytují informace manažeru zásuvných modulů o dostupných modulech. Jedná se



Obrázek 3.2: Uživatelské rozhraní manažera zásuvných modulů.

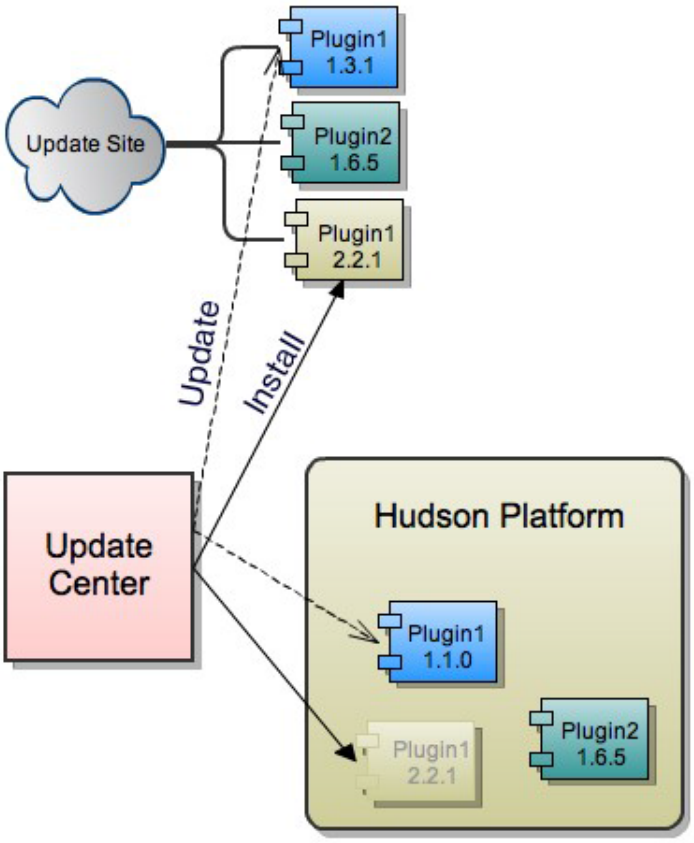
jednak o nové moduly, jednak i o aktualizace již nainstalovaných modulů. Jenkins umožňuje nastavit a použít více aktualizačních míst. Manažer zásuvných modulů využívá tyto objekty také pro zobrazení dostupných modulů ve svém uživatelském rozhraní, které bylo popsáno v sekci 3.2.1. Objekt centra aktualizací je vlastníkem různých úloh, jako například úloh stahování nebo úloh instalace. Tyto úlohy jsou vykonávány asynchronně, jako důsledek požadavků z uživatelského rozhraní manažera modulů.

3.2.3 Obálka modulu

Třída `PluginWrapper` obaluje každý správně nainstalovaný modul. Úkolem objektu třídy `PluginWrapper` je zajišťovat povolení/zakázání modulu a takzvané přišpendlení modulu, které umožňuje zachovat uživateli určitou verzi integrovaného modulu při přechodu systému na novější verzi. Obálka nese také další metadata o modulu, jako například informaci o souboru s modulem, manifest obsahující charakteristiky modulu, URL, kde se nachází veřejné informace o modulu, nebo například informaci o tom, zda je modul integrovaný v základním balíku Jenkins CI nebo zda je dodatečně doinstalovaný. [3]

3.2.4 Strategie modulu

Třída `ClassicPluginStrategy` jež je implementací rozhraní `PluginStrategy`, zajišťuje načítání a rozbalování jednotlivých modulů. Jakákoliv třída implementující `PluginStrategy`, tedy i výchozí `ClassicPluginStrategy`, vytváří na požádání (na zavolání metody s parametrem typu `File`) objekt typu `PluginWrapper`. [3]



Obrázek 3.3: Schéma centra aktualizací v aplikaci Jenkins. Zdroj: [13].

3.2.5 Postup zavádění modulů

Ze všeho nejdříve jsou zavedeny integrované moduly. Integrované moduly se nachází v adresáři `WEB-INF/plugins` balíku Jenkins. Z tohoto adresáře jsou všechny moduly zkopírovány do lokální kořenové složky se zásuvnými moduly (v unixových operačních systémech se standardně jedná o složku „`~/.jenkins/plugins`“).

Třída `InitStrategy` slouží ke zjištění, kde všude se nachází zásuvné moduly. Na požádání vrátí seznam souborů s nalezenými moduly. Hledá v kořenovém adresáři modulů a také v umístěních definovaných systémovou vlastností `hudson.bundled.plugins`. Následně je pro každý modul pomocí objektu `PluginStrategy` vytvořen obal, objekt typu `PlugginWrapper`, což mimo jiné zahrnuje:

1. načtení manifestu modulu
2. nalezení všech knihoven a tříd obsažených v modulu
3. nalezení závislých modulů
4. vytvoření objektu `ClassLoader` (slouží k dynamickému načítání tříd v Javě) pro vlastní třídy modulu
5. vytvoření objektu `ClassLoader` pro závislé třídy

Dále se připraví seznam aktivních modulů a zkontroluje se jejich možná duplicita. Nakonec se načtou všechny aktivní moduly pomocí `PluginStrategy`. V průběhu načítání modulů se také identifikují chybějící a cyklické závislosti.

Kapitola 4

Vývoj zásuvných modulů

Tato kapitola popisuje jakým způsobem jsou vyvíjeny zásuvné moduly do systému Jenkins CI. Mimo jiné pojednává i o formátu zásuvných modulů a jakým způsobem jsou distribuovány uživatelům.

4.1 Formát zásuvného modulu

Zásuvný modul do systému Jenkins je prostý JAR archiv s příponou `hpi` nebo `jpi`. V tomto formátu je distribuován i uchováván (po instalaci). Obsah archivu musí dodržovat určité konvence, jak je znázorněno v příkladu 4.1.

Příklad 4.1

```
nazev.jpi
 +- META-INF
 |   +- MANIFEST.MF
 +- WEB-INF
 |   +- classes
 |   +- lib
 +- (statické zdroje)
```

- Název souboru před `jpi/hpi` příponou reprezentuje krátké jméno zásuvného modulu. Toto jméno jednoznačně identifikuje modul v systému.
- `MANIFEST.MF` je soubor, který obsahuje metadata o obsahu JAR archivu. V případě použití JAR archivu jako zásuvného modulu do systému Jenkins, obsahuje `MANIFEST.MF` kromě obvyklých položek také několik speciálních záznamů:
  - `Short-Name`: Krátký název zásuvného modulu, který slouží pro vnitřní identifikaci v systému Jenkins.
  - `Plugin-Developers`: Tento atribut obsahuje informace o vývojářích, kteří se podíleli na tvorbě modulu.
  - `Url`: URL adresa na wiki stránku projektu Jenkins CI obsahující informace o tomto modulu.
  - `Jenkins-Version`: Verze aplikace Jenkins CI, pro kterou byl modul vytvořen. Tento atribut tedy značí minimální verzi systému Jenkins, pod kterou lze modul provozovat.

  - `Long-Name`: Atribut `Long-Name` je volitelný delší název pro zásuvný modul. Pokud je uveden, zobrazuje se uživatelům systému tento název modulu. Pokud není uveden, použije se krátký název, který jinak slouží pouze pro vnitřní identifikaci modulu v Jenkins CI.
  - `Plugin-Dependencies`: Tento volitelný atribut obsahuje seznam čárkou oddělených názvů modulů a jejich verzí, které jsou vyžadovány pro běh tohoto modulu. Třídy a knihovny takto definovaných modulů jsou načteny příslušnou instancí `ClassLoader` modulu, takže je může modul využívat.

- `WEB-INF/classes` může obsahovat soubory s třídami, ze kterých se skládá modul. Dále může obsahovat soubory s `.jelly` soubory, které definují uživatelské rozhraní. Alternativně mohou být některé nebo všechny soubory zabaleny do JAR archivu a umístěny ve složce `WEB-INF/lib`.

- `WEB-INF/lib` obsahuje JAR archivy, které jsou načteny spolu se soubory ze složky `WEB-INF/classes` pomocí `ClassLoader` objektu příslušného modulu. Typicky tato složka slouží pro umístění různých knihoven, potřebných pro běh modulu.

- Statické zdroje modulu se umísťují do kořenové složky archivu. Může se jednat například o obrázky, HTML soubory, CSS šablony, JavaScript soubory a podobně. Takto zabalené zdroje jsou dostupné přes webové rozhraní v rámci relativní adresy. Pokud je v archivu například soubor `abc/def.png` a Jenkins běží na adrese `http://localhost`, URL s umístěním obrázku je `http://localhost/plugin/nazev/abc/def.png`.

4.2 Maven a zásuvné moduly

Pro vývoj zásuvných modulů se doporučuje využívat nástroj Maven, pomocí kterého je i vyvíjeno jádro systému Jenkins. Pro účely vývoje modulů byl vytvořen zásuvný modul do nástroje Maven, který se nazývá „Maven Jenkins Plugin“ a identifikuje se jako „maven-hpi-plugin“. Modul poskytuje různé cíle pro usnadnění vývoje modulů:

- `hpi:create` – Vytvoří šablonu pro nový zásuvný modul. Lze nastavit jméno a verzi modulu. Podobným způsobem funguje webový generátor šablon pro nové moduly, který se nachází na adrese `http://plugin-generator.jenkins-ci.org/`.

- `hpi:hpi` – Přeloží projekt a sestaví hpi balíček.

- `hpi:hpl` – Funguje podobně jako hpi:hpi, ale vytvoří hpl balíček. Jedná se o ladící verzi balíčku hpi s odkazem na umístění souborů v lokálním souborovém systému. Tvorba balíčku je rychlejší, ale funguje pouze na stroji, kde se vyvíjí.

- `hpi:run` – Přeloží projekt a spustí lokální verzi Jenkins (v rámci aplikačního serveru Jetty) s přeloženým modulem. Maven si v tomto případě sestaví sám spustitelnou verzi systému Jenkins, vývojář tedy ani nemusí mít Jenkins nainstalován v operačním systému. Domovská složka systému Jenkins je zvolena lokálně v rámci projektu modulu, takže tento způsob ladění nemůže poškodit obsah standardní domovské složky Jenkins, pokud existuje. [4]

4.3 Způsob rozšiřování funkcionality

Ve zdrojovém kódu projektu Jenkins se nachází abstraktní třídy, které implementují rozhraní `ExtensionPoint`. Samotné rozhraní je prázdné, jeho implementace abstraktní třídou však indikuje, že třídy, které dědí z této abstraktní třídy a nachází se v zásuvném modulu, realizují příslušné rozšíření. V zásuvném modulu je třeba takovouto třídu, která dědí z jakékoliv abstraktní třídy implementující rozhraní `ExtensionPoint`, označit anotací `@Extension`. Systém zásuvných modulů takto správně identifikuje a aplikuje příslušné rozšíření. [6]

Konstruktor třídy v modulu také může obsahovat anotaci `@DataBoundConstructor`, která systému říká, že tento konstruktor přebírá parametry uživatelského rozhraní.

Jádro systému Jenkins obsahuje přes jedno sto abstraktních tříd, které implementují rozhraní `ExtensionPoint`. Jedná se o třídy s různorodou funkcionalitou. Jako příklad lze uvést třídy `Builder` (implementující překlad projektů) nebo `Notifier` (implementující notifikace).

Zásuvné moduly mohou obsahovat další třídy implementující `ExtensionPoint` a poskytnout tak dalším modulům možnost jejich realizace, čehož je hojně využíváno.

4.4 Projekty využívané pro chod systému Jenkins

Systém Jenkins využívá pro svůj běh několik důležitých knihoven nebo nástrojů, které jsou přímo využívány i při tvorbě zásuvných modulů. Nástroje sloužící k jednoduché tvorbě uživatelského rozhraní v Jenkins jsou Stapler[1] a Jelly[2]. Pro trvalé uchovávání dat (tzv. serializaci) je použita knihovna XStream[3].

4.4.1 Stapler a Jelly

Stapler je knihovna, která provádí intuitivní překlad URL požadavků na volání metod objektů v rámci architektury webové aplikace. Stapler každé klientem volané URL rekurzivně zpracovává na volání metod, od kořenového objektu systému počínaje, až do té doby, než dostane odpověď v podobě takzvané akční metody nebo statického pohledu. Způsob tohoto zpracování URL požadavků byla vysvětlena v sekci 2.3.4. Stapler byl vyvinut na míru systému Hudson, ale funguje jako oddělená knihovna, takže jej mohou využívat i jiné projekty. Stapler také umožňuje předávat jednotlivým metodám parametry, jako například řetězce nebo čísla, které mohou být součástí URL požadavků.

Jelly je oproti tomu nástrojem pro specifikaci a interpretaci takzvaných „spustitelných XML souborů“. Jelly obecně umožňuje psaní skriptů v podobě XML souborů, avšak v projektu Jenkins je využíván pro specifikaci uživatelského rozhraní. Uživatelské rozhraní zásuvných modulů je tedy také definováno v těchto skriptech, které mají podobu XML souborů s příponou `.jelly` a typicky se nachází ve složce `resources` v projektu zásuvného modulu. Pomocí jelly skriptů lze snadno definovat formuláře, jejichž uživatelem vyplněný obsah je pak předáván konstruktorům a metodám objektů, které tvoří zásuvný modul. Stapler přímo vyhledává tyto soubory. Skripty s příponou `.jelly` jsou tedy těmito již zmíněnými statickými pohledy, ke kterým se Stapler snaží dostat při rekurzivním zpracování URL požadavků.

[1] Stapler – http://stapler.kohsuke.org/
[2] Jelly: Executable XML – http://commons.apache.org/proper/commons-jelly/
[3] XStream – http://xstream.codehaus.org/

Jenkins poskytuje více knihoven značek pro jelly, které obsahují prvky použitelné pro tvorbu uživatelského rozhraní jak v jádře, tak i v zásuvných modulech. Nejdůležitějšími knihovnami jsou `/lib/form` a `/lib/hudson`. Knihovna `/lib/form` obsahuje formulářové prvky pro komunikaci s uživatelem a `/lib/hudson` definuje prvky obsahující informaci o aktuálním stavu běžící instance systému Jenkins. Užitečné prvky nabízí také knihovna `/lib/layout`, která obsahuje základní rozvržení stránek webového rozhraní Jenkins. Tyto knihovny lze v jelly skriptech používat tak, že deklarujeme jmenný prostor knihovny na začátku skriptu a pak jednoduše vkládáme XML elementy z knihovny s příslušným prefixem identifikujícím deklarovanou knihovnu. Pokud například deklarujeme použití knihovny `/lib/form` zápisem `xmlns:f=“/lib/form“`, můžeme následně použít ve skriptu elementu `textbox` zápisem `<f:textbox/>`.

4.4.2 XStream

Knihovna XStream slouží pro serializaci objektů v Javě. Serializace je převedení vnitřní reprezentace dat do podoby datového streamu, s kterým je dále možno libovolně nakládat. XStream serializuje data do podoby XML souborů. Takto serializovaná data jsou tudíž snadno čitelná i lidem. XStream samozřejmě nabízí také možnosti zpětného načtení serializovaných dat. Jenkins využívá knihovnu XStream pro serializaci různých nastavení systému, což zahrnuje i globální nastavení zásuvných modulů nebo například nastavení projektů, do jejichž vlastností přidávají prvky i zásuvné moduly. V systému Jenkins jsou mechanismy serializace nastavení zcela automatizovány. Pokud však programátor nechce, aby se nějaká data zbytečně serializovala, stačí příslušný atribut třídy označit vlastností `transient`. Takto označený atribut nebude serializován. Serializovaná data s nastavením ukládá Jenkins do domovského adresáře instance systému (standardně se jedná o složku `~/.jenkins`).

4.5 Stavba jednoduchého modulu

V sekci 4.1 je popsána stavba přeloženého modulu, připraveného k instalaci. V této sekci je popsána stavba zdrojového kódu jednoduchého modulu. Jako příklad byl zvolen modul vytvořený online generátorem modulů, o kterém se psalo v sekci 4.2. Modul byl nazván `novy-plugin`. Složky a soubory modulu jsou znázorněny v příkladu 4.2.

V kořenovém adresáři se nachází soubor `pom.xml`, v němž jsou zapsány instrukce pro překladový nástroj Maven. Obsahuje informace o modulu, jak ho přeložit, co je rodičovským projektem a kde se nachází repozitáře s potřebnými moduly do programu Maven.

Složka `src/main/java` obsahuje zdrojové kódy modulu hierarchicky uspořádané ve standardním stylu zápisu programů v jazyce Java (složky jsou uspořádány podle názvů balíčků). Vzorový modul obsahuje jednu hlavní třídu `HelloWorldBuilder`, která je rozšířením třídy `Builder`, jenž slouží pro vykonávání překladu projektů (úloh). Samotný překlad úlohy je vykonáván metodou `perform()`. Tato třída dále obsahuje podtřídu `DescriptorImpl`, která je rozšířením třídy `BuildStepDescriptor`. Třída je určena pro uchování nastavení úlohy související s modulem a globálním nastavením modulu. Vzorový modul při překladu úlohy pouze vypíše logovací výstup.

Složka `src/main/resources` obsahuje soubory definující uživatelské rozhraní modulu. Soubor `index.jelly` obsahuje jednoduchý popis modulu, který se zobrazuje v manažeru modulů. Ve složce `HelloWorldBuilder` se nachází soubory tvořící uživatelské rozhraní pro třídu `HelloWorldBuilder`. XML soubor `config.jelly` popisuje rozhraní pro nastavení překladu úlohy a `global.jelly` popisuje rozhraní pro globální nastavení modulu. Dále jsou

v tomto adresáři pomocné soubory HTML obsahující uživatelskou nápovědu pro vyplňování formulářů.

Modul se přeloží příkazem `mvn hpi:hpi`, jak bylo popsáno v sekci 4.2. Příkaz vytvoří složku `target` s balíčkem `novy.hpi`.

Příklad 4.2

```
.
|-- pom.xml
|-- src
    |-- main
        |-- java
        |   |-- org
        |       |-- jenkinsci
        |           |-- plugins
        |               |-- novy
        |                   |-- HelloWorldBuilder.java
        |-- resources
            |-- index.jelly
            |-- org
                |-- jenkinsci
                    |-- plugins
                        |-- novy
                            |-- HelloWorldBuilder
                                |-- config.jelly
                                |-- global.jelly
                                |-- help-name.html
                                |-- help-useFrench.html
```

4.6 Způsob distribuce modulů

Zásuvné moduly jdou jednoduše distribuovat jako `jpi` balíčky. Modul v podobě lokálně uloženého balíčku `jpi` (nebo `hpi`) lze instalovat do systému Jenkins pomocí pokročilých voleb manažera zásuvných modulů. Pokud si vývojář zásuvného modulu přeje, aby byl jeho modul dostupný všem uživatelům aplikace Jenkins, může požádat o jeho zveřejnění v repozitáři modulů[4]. Zveřejnění modulu probíhá v následujících krocích:

1. vytvoření účtu na webu jenkins-ci.org
2. vytvoření účtu na serveru GitHub
3. požádání komunity o udělení práv editovat zdrojový kód v rámci komunitního účtu `github.com/jenkinsci`
4. nahrání zdrojového kódu modulu do nového GitHub repozitáře v rámci účtu `jenkinsci`
5. založení informační wiki stránky o modulu na serveru jenkins-ci.org
6. úprava `pom.xml` souboru tak, aby obsahoval správné informace o umístění repozitáře se zdrojovým kódem, autorovi modulu a wiki stránce projektu

[4] Jenkins Plugins Repository – http://repo.jenkins-ci.org/releases/org/jenkins-ci/plugins/

7. sestavení a odeslání aktuální verze modulu do repozitáře modulů pomocí Maven příkazu „`mvn release:prepare release:perform`“

Pokud vše proběhne správně, měl by být takto zveřejněný modul instalovatelný do aplikace Jenkins přes internet pomocí centra aktualizací.

4.7 Vývoj modulů v jazyce Ruby

Vzhledem k tomu, že je server Jenkins naprogramován v jazyce Java, využívá se standardně pro vývoj zásuvných modulů tentýž jazyk. Ne každému vývojáři, který chce nebo musí implementovat nějaký modul, však tento jazyk vyhovuje, a proto vznikl projekt, jehož hlavním cílem je umožnit vývojářům vyvíjet snadno moduly v jazyce Ruby. Projekt se nazývá jenkins.rb a zdrojové kódy jsou umístěny na serveru GitHub[5].

4.7.1 Cíle projektu jenkins.rb

Projekt jenkins.rb je stále ve vývoji, přesto už je ve stavu schopném nabídnout komfortní alternativu k vývoji modulů v jazyce Java. Komunita kolem projektu si vytyčila několik cílů, kterých chtějí dosáhnout nebo již dosáhli:

- *Vývoj s použitím výhradně jazyka Ruby*
  Výhradně znamená v tomto případě naprogramování kompletního modulu bez použití jediné řádky kódu v jazyce Java.
- *Struktura zdrojových kódů v Ruby stylu*
  Projekty v jazyce Java, moduly do systému Jenkins nevyjímaje, mají rozmístěné soubory podle balíčků, do kterých patří. Při vývoji v Ruby fungují jiné zvyklosti a složitá hierarchie adresářů je pouze přítěží.
- *Vývoj bez použití nástroje Maven*
  Nástroj Maven je vytvořen na míru pro jazyk Java. Programátoři v jazyce Ruby jsou zvyklí na jiné nástroje, jako například *RubyGems*. Moduly v jazyce Ruby jsou tedy spravovány pomocí tohoto nástroje a nástroje *Jenkins Plugin Tool* (viz dále).
- *Správa životního cyklu modulů*
  Zásuvný modul hpi do nástroje Maven obstarává správu kompletního životního cyklu vyvíjeného modulu. Jelikož však projekty v Ruby nepoužívají Maven, byl pro pro tyto účely vytvořen jiný nástroj, s názvem *Jenkins Plugin Tool*.
- *Psaní kódu bez importování nativních Java tříd*
  Tento cíl zatím nebyl splněn. Podle dostupných informací je zatím vytvořeno Ruby API pouze pro blíže nespecifikované nízké procento `ExtensionPoint` tříd. Většinu tříd je tedy nutno importovat přímo z Java API.

Projekt si tedy neklade za cíl pouze umožnit vývoj modulů v Ruby, ale umožnit ho také co nejkomfortněji. V době psaní této zprávy se v repozitáři s moduly nacházelo celkem deset modulů napsaných v jazyce Ruby. Vzhledem k celkovému počtu modulů je to sice zanedbatelné číslo, přesto ale mohou být tyto moduly pro některé uživatele nepostradatelné. [11]

[5] Jenkins.rb – https://github.com/jenkinsci/jenkins.rb

4.7.2 Obsah projektu

Projekt jenkins.rb zahrnuje 4 části, které dohromady tvoří komplexní podporu programátorům v Ruby:

- *java-runtime*
  Jedná se o běhové prostředí zajišťující načítání a běh modulů napsaných v Ruby. Po kompilaci tvoří modul s názvem ruby-runtime.jpi (nezaměnit s následující součástí, ruby-runtime), na němž jsou všechny moduly napsané v Ruby závislé. Modul ruby-runtime.jpi je sám závislý na knihovnách/nástrojích:
  - JRuby – interpret jazyka Ruby pro JVM
  - JRuby-Rack – nástroj pro spouštění Rack aplikací (webové aplikace napsané v Ruby) uvnitř Java servletů
  - JRuby-XStream – knihovna umožňující serializaci Jruby tříd pomocí XStream
  - JRuby-Stapler – knihovna umožnující práci knihovny Stapler s Ruby objekty

  Knihovny JRuby-XStream a JRuby-Stapler jsou vytvořeny přímo pro účely projektu jenkins.rb.

- *ruby-runtime*
  Obsahuje skripty napsané v Ruby, jenž tvoří rozhraní mezi třídami v Ruby, které obsahuje terminální modul, a třídami systému Jenkins naprogramovanými v Javě. Tato součást je obsažena ve všech modulech napsaných v Ruby (jako gem balíček). [2]

- *Jenkins Plugin Tool (zkráceně jpi)*
  Tento nástroj byl vytvořen pro účely automatizované správy projektů modulů v Ruby (nahrazuje Maven pro Java moduly). Nástroj umožňuje vytvořit šablonu projektu nového modulu, vytvořit šablonu implementace vybrané `ExtensionPoint` třídy, přeložit modul a sestavit jpi/hpi balíček, spustit Jenkins server s poslední verzí modulu a nahrát modul do repozitáře modulů.

- *Command-line interface (zkráceně CLI)*
  CLI je konzolové rozhraní pro ovládání serveru Jenkins. Tato součást je nepodstatná z hlediska zaměření této práce, byla vytvořena hlavně pro možnost automatizace ovládání Jenkins serveru (toho využívá jpi). [2]

Podle dostupných zdrojů byly již dříve v jádře Jenkins provedeny změny umožňující načítání metadat o třídách napsaných v Ruby. [10]

4.7.3 JRuby

JRuby je implementace jazyka Ruby pro Java Virtual Machine. JRuby může být používán jako samostatný interpret jazyka Ruby. Podstatnější však je, že umožňuje za běhu spouštět skripty napsané v jazyce Ruby v rámci vykonávání kódu napsaného v jazyce Java, čehož je využíváno v projektu jenkins.rb. Způsob takového volání externího kódu v Ruby je založen na třídě `ScriptingContainer`, která uchovává běhové prostředí pro vykonávaný kód. Nejprve je vytvořen objekt této třídy, na který je následně zavolána metoda `parse()` s argumentem určujícím, kde se nalézá Ruby kód, který má být načten. Takto načtený

kód může být následně vykonán. Třída `ScriptingContainer` má mnoho dalších metod pro práci s běhovým prostředím. Lze například nastavovat proměnné, zjišťovat jejich hodnotu nebo přímo spouštět kód předaný řetězcem. Nespornou výhodou je, že takto vykonávaný Ruby kód může přímo pracovat s Java třídami. [7]

4.7.4 Načítání a běh Ruby modulů

Projekt jenkins.rb bohužel postrádá programovou dokumentaci, nicméně zdrojové kódy jsou dostupné, je tedy možno vyčíst základní funkcionalitu správy modulů napsaných v jazyce Ruby. Zásuvný modul ruby-runtime.jpi (zkompilovaná část java-runtime projektu jenkins.rb) obsahuje mimo jiné třídy `RubyPlugin` a `RubyExtensionFinder`, které jsou jádrem procesu načítání a vykonávání Ruby modulů. Třída `RubyExtensionFinder` je rozšířením třídy `ExtensionFinder` (`ExtensionPoint`). Má za úkol vyhledávání modulů napsaných v Ruby. Třída `RubyPlugin` (rozšiřující třídu `Plugin`) je zase obálkou pro samotný modul. Uchovává instanci třídy `ScriptingContainer`, která obsahuje načtený kód Ruby modulu a dále zprostředkovává volání metod objektů tohoto modulu.

Kapitola 5

Návrh SDK

Tato kapitola se zabývá návrhem nástrojů pro usnadnění vývoje zásuvných modulů pro Jenkins v jazyce Python. Jsou zde představeny cíle kladené na výstupní aplikaci, součásti navržené aplikace a princip napojení modulů v Pythonu na Jenkins Java API. Projekt byl nazván *jenkins.py* po vzoru projektu jenkins.rb.

5.1 Cíle projektu

Hlavním cílem projektu jenkins.py je poskytnout podporu pro vývoj modulů v jazyce Python, podobně jako je tomu v případě projektu jenkins.rb pro jazyk Ruby. Do stanovených cílů ovšem nepatří zbavení se závislosti na nástroji Maven, strukturování projektů výhradně ve stylu programů v Pythonu ani zbavení se nutnosti přímé práce s Java třídami systému Jenkins. Cíle projektu lze shrnout v následujících bodech:

- *Snadná implementace funkčních částí modulů pomocí jazyka Python*
  Moduly by však měly obsahovat také kód v Javě kvůli snadnému napojení na systém zásuvných modulů.
- *Automatická tvorba šablon pro nové zásuvné moduly*
  Programátor neznalý jazyka Java by tak měl dostat pomocnou ruku při zakládání nového projektu.
- *Funkční práce s moduly pomocí nástroje Maven*
  Všechny operace Maven modulu *hpi* by měly být funkční také při práci s Python moduly.
- *Vytvoření manuálu k programování Python modulů*
  Manuál by měl být srozumitelný i programátorům bez hlubších znalostí jazyka Java.

5.2 Jython

Projekt jenkins.rb využívá k provádění kódu v Ruby knihovnu/nástroj JRuby. Analogicky se pro jazyk Python nabízí nástroj Jython[1]. Jython je interpret jazyka Python pro Java Virtual Machine. Stejně jako JRuby lze i Jython používat jednak jako nezávislý interpret jazyka Python a jednak jako integrovaný interpret v rámci nějaké Java aplikace. Běhové

[1] Jython – http://www.jython.org/

prostředí Python kódu se v druhém případě uchovává ve třídě `PythonInterpreter`. Kód programu lze interpretu zadat například metodou `execfile()` s jedním argumentem určujícím cestu ke skriptu. Hodnotu proměnných lze nastavovat metodou `set()`, podobně je to se zjišťováním jejich hodnot. Je také možno přímo vyhodnocovat výrazy v rámci běhového prostředí metodou interpretu `eval()`. Takto lze tedy i volat funkce obsažené v načteném skriptu.

Podobně jako JRuby umožňuje Jython pracovat v rámci Python kódu s dostupnými třídami jazyka Java. To se týká standardních tříd JRE i tříd samotné Java aplikace (v případě užití integrovaného interpretu). [8]

5.3 Součásti projektu jenkins.py

Navržená struktura projektu sestává ze tří hlavních částí:

1. *Automatický generátor wrapperů*
   Cílem je vytvořit spustitelnou aplikaci, která projde zdrojový kód aktuální verze serveru Jenkins a vytvoří pro každý `ExtensionPoint` takzvaný *wrapper* (z angl. wrapper – obal), který bude fungovat jako rozhraní mezi kódem modulu psaným v Pythonu a aplikačním rozhraním systému Jenkins. Funkce wrapperů je vysvětlena v sekci 5.5.

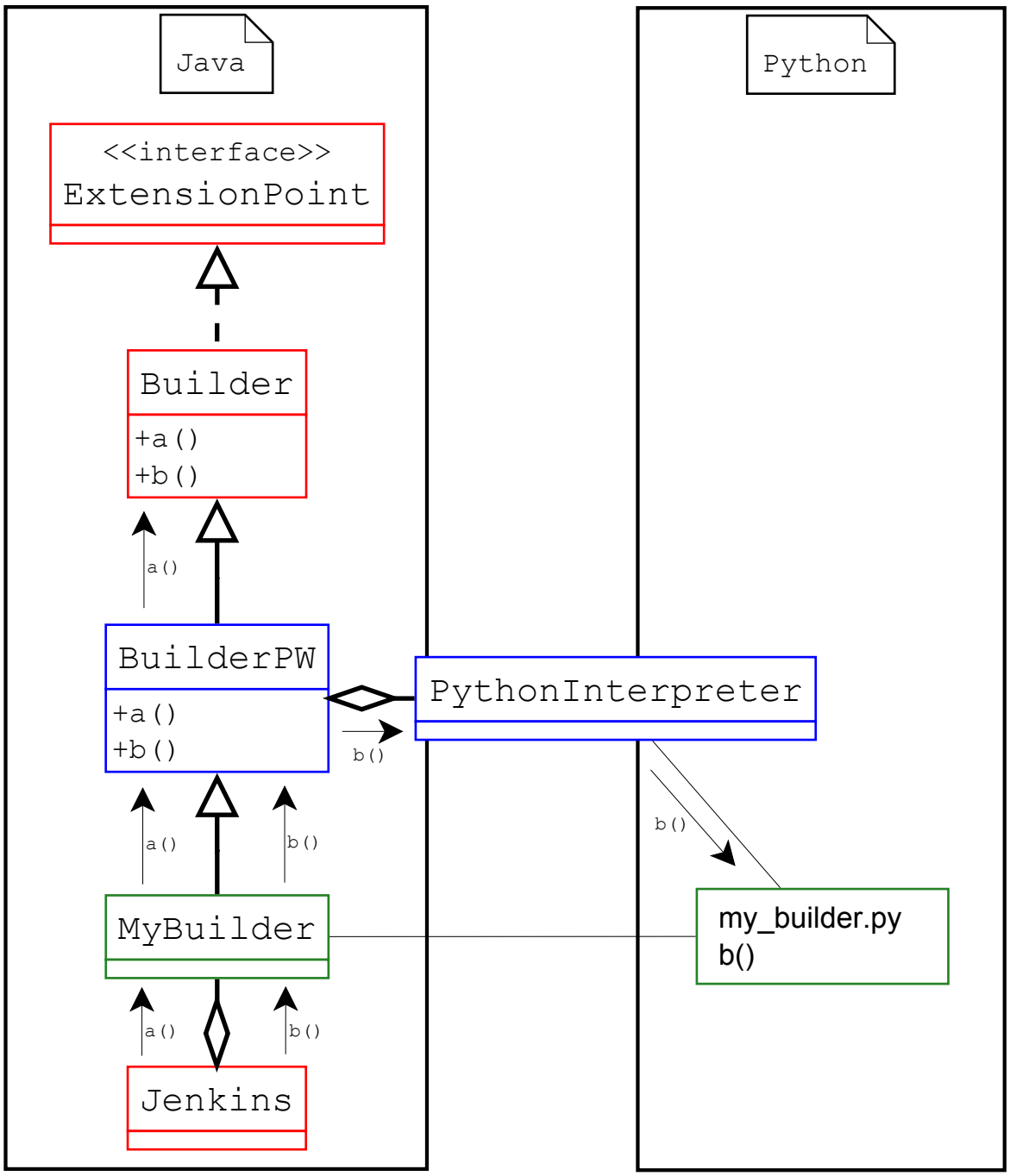
2. *Pomocné knihovny*
   Bude třeba vytvořit pomocné funkce na převod objektů typu `PyObject` (které vrací volání `eval()` nad vestavěným interpretem Jython) na standardní datové typy jazyka Java. Dále bude třeba zajistit funkci rozbalování Python souborů z balíčků `JPI`, tak aby mohly být následně volány wrappery. Standardně se všechny Java třídy modulu pouze přebalí do balíčku `JAR` a neexistuje tedy přímý přístup k souborům typu `.class`. Pro Python skripty obsažené v modulech bude rozbalování nezbytné. Všechny ostatní funkce nutné pro správný chod wrapperů budou také obsaženy v této knihovně.

3. *Generátor šablon modulů*
   Jedná se o nástroj, který umožní programátorům vygenerovat kostru nového Python modulu pro `ExtensionPoint Builder`. Stejně jako u generátoru wrapperů se bude jednat o spustitelnou aplikaci. Takto vytvořená kostra nového modulu by měla programátorovi pomoci s vývojem jednoduchého builderu v Pythonu nebo alespoň naznačit způsob stavby modulů v Pythonu v případě, že chce programátor rozšířit jiný `ExtensionPoint` než je `Builder`.

5.4 Modul python-wrapper

Výstup z generátoru wrapperů spolu s pomocnými knihovnami budou tvořit modul, na kterém budou závislé všechny moduly programované v Pythonu. Modul bude nazván *python-wrapper*. Zde nejde použít analogie z projektu jenkins.rb, protože název python-runtime by neodpovídal rozsahu, jakého dosahuje modul ruby-runtime.

Generátor šablon modulů tedy musí zohledňovat závislost na modulu python-wrapper (v souboru pom.xml).



Obrázek 5.1: Znázornění práce Python wrapperu `BuilderPW` pro třídu `Builder`.

5.5 Princip napojení na Java API

Na obrázku 5.1 je znázorněn princip funkce Python wrapperů obsažených v modulu python-wrapper.jpi pro `ExtensionPoint` třídy. Pro příklad byla zvolena třída `Builder` a programátorem modulu vytvořená implementace `MyBuilder`. Červeně jsou vyznačeny třídy, které náleží samotnému systému Jenkins. Modře jsou zvýrazněny třídy, které bude obsahovat modul python-wrapper.jpi. Zeleně jsou pak vyznačeny součásti, které vytvoří programátor modulu. Postup volání metod objektu zásuvného modulu `MyBuilder` systémem Jenkins sestává z následujících kroků:

1. `MyBuilder` sám neimplementuje žádnou metodu kromě konstruktoru a pomocných delegačních metod. Je tedy zavolána metoda rodičovské třídy `BuilderPW` (akronym PW znamená Python Wrapper).

2. `BuilderPW` se podívá přes instanci `PythonInterpreter` do skriptu `my_builder.py`, zda je zde obsažena příslušná funkce se správným počtem parametrů.

3. Pokud je ve skriptu `my_builder.py` obsažena příslušná funkce, je zavolána (v příkladu jde o metodu `b()`). V opačném případě je výpočet delegován abstraktní nadřazené třídě `Builder` (v příkladu metoda `a()`).

4. V případě, že volaná metoda vrací nějakou hodnotu a výpočet je delegován skriptu `my_builder.py`, převede se odpověď na správný datový typ a vrátí se výsledek.

Jediný kód v Javě obsažený v modulu by tedy měla být třída `MyBuilder` s anotací `@Extension`, která pouze deleguje výpočet příslušnému wrapperu. Tato třída musí být v modulu obsažena už kvůli správnému nalezení a identifikaci modulu. Samotná funkcionalita modulu je však obsažena v Python skriptu `my_builder.py` a případně dalších skriptech nebo knihovnách, které jsou importované v rámci skriptu `my_builder.py`. Tímto je zajištěno napojení Python skriptů obsažených v modulu na Java API poskytované systémem Jenkins.

5.6 Proveditelnost návrhu

V rámci návrhu byl implementován jednoduchý zásuvný modul, který dokazuje, že navržené řešení je implementovatelné (angl. Proof of Concept, zkr. PoC). Modul implementuje jednoduchý překladač (angl. builder), tedy rozšíření implementující `ExtensionPoint Builder`, který pouze zaznamenává svou činnost pomocí objektu `Logger`. Důležité je, že metoda `perform()`, která provádí hlavní činnost překladače (v tomto případě pouze zaznamenává svou účast), byla implementována pomocí funkce `perform()` obsažené v přiloženém skriptu napsaném v jazyce Python.

PoC modul obsahuje jednak třídy, které by měly být později součástí modulu python-wrapper (patří sem pomocné třídy a třídy implementující zjednodušené Python wrappery `BuilderPW` a `BuildStepDescriptorPW`), a jednak součásti, které budou patřit do konečného zásuvného modulu. Rozdělení součástí do dvou modulů je prokazatelné díky dříve popsané možnosti definování závislostí napříč moduly v systému Jenkins.

V souvislosti s implementací PoC modulu bylo objeveno několik problémů, s kterými je třeba při vývoji projektu jenkins.py počítat:

- Kromě tříd, které dědí z třídy implementující `ExtensionPoint` rozhraní, jsou nedílnou součástí většiny modulů takzvané deskriptory. Deskriptor je třída, která dědí z třídy `Descriptor` nebo z některé z jejích podtříd a uchovává a zpracovává globální data společná pro všechny instance rozšiřující třídy. Například pro třídu `Builder` je k dispozici deskriptor `BuildStepDescriptor`. Modul implementující překladač pak dědí z těchto dvou tříd, přičemž třída, která dědí z třídy `BuildStepDescriptor`, je vnitřní statickou třídou třídy odvozené z třídy `Builder`.
  Z potřeby přítomnosti deskriptorů v modulech bude třeba generovat Python wrappery také pro všechny deskriptory. Jedině tak bude možné implementovat veškerou užitečnou funkcionalitu modulů v Pythonu.

- Třídy modulů bývají často součástí serializace pomocí knihovny XStream. Serializace se týká také všech rodičovských tříd. Vygenerované wrappery by tedy měly u všech svých atributů uvádět klíčové slovo `transient`, aby tak indikovaly knihovně XStream, že není třeba dané atributy serializovat. V opačném případě by docházelo k chybám při pokusu o uložení dat patřících modulu.

- Jak bylo již vysvětleno, Jenkins využívá pro napojení funkcionality tříd na webové rozhraní projekt Stapler. Stapler hledá v jednotlivých třídách rozšíření přítomnost metod s prefixem do a get. Jelikož úplné názvy těchto metod nejsou dopředu známy (jejich přítomnost záleží na konkrétní implementaci), nemůžou wrappery automaticky delegovat činnost těchto metod skriptům v Pythonu. *Musí být proto k dispozici mechanismus pro ruční delegaci vykonávání funkcionality Python skriptům.*
  Stejná situace může nastat, pokud programátor implementuje v rámci své Java třídy nějaké rozhraní. Takovéto chování také nemůže být dopředu známo, a proto bude programátor muset ručně delegovat, pomocí stejného mechanismu, vykonávání metod daného rozhraní Python skriptu.

- V rámci skriptů napsaných v Pythonu bude třeba přistupovat pomocí nějakého mechanismu k instanci Java objektu, z kterého byl delegován výpočet v daném skriptu. Tato instance může být využita například k volání metod rodičovských tříd nebo k nastavení atributů, které mají být uloženy pomocí knihovny XStream. Skripty v Pythonu můžou implementovat vlastní mechanismus ukládání dat, ovšem využití rodičovského Java objektu a knihovny XStream je nejjednodušší řešení.

- Většina rozšiřitelných (`ExtensionPoint`) tříd obsahuje *abstraktní metody*, jejichž implementace je kontrolována v průběhu překladu modulu (standardní chování jazyka Java). Jelikož je však vykonávání těchto metod delegováno Python skriptům, wrappery musí tyto metody implementovat a přímo delegovat jejich výpočet (neexistuje rodičovská implementace v případě neexistence implementace v Pythonu). Modul python-wrapper by se měl tedy postarat o kontrolu implementace daných funkcí v Pythonu a v případě chybějících implementací srozumitelně informovat programátora.

- V případě, že je modul testován pomocí příkazu „`mvn hpi:run`“, je s ním nakládáno jinak než při standardním načtení systémem Jenkins z balíčku `jpi`. V prvním případě nejsou soubory modulu vůbec zabaleny do archivu a jsou pouze zkopírovány do

dočasného adresáře. V druhém případě se nachází soubory tříd i statické zdroje v archivu JAR. Knihovna modulu python-wrapper proto musí správně vyhodnotit, kde se nachází skripty v Pythonu a v případě potřeby je vybalit z archivu do nadřazeného adresáře.

- Jazyk Java má mírně odlišné bázové typy než jazyk Python. Všechny hodnoty získané z interpretu `PythonInterpreter` navíc mají v rámci kódu v jazyce Java typ `PyObject`. Modul python-wrapper proto musí obsahovat funkce pro převod mezi jednotlivými datovými reprezentacemi oběma směry.

- Modul python-wrapper bude obsahovat jedinou závislost, a to závislost na knihovně Jython. Tuto závislost bude třeba indikovat v souboru `pom.xml` a zajistit přítomnost balíčku Jython v Maven repozitáři projektu Jenkins CI.

5.7 Výhody a nevýhody řešení

Vzhledem k tomu, že se navržený projekt jenkins.py odlišuje svými možnostmi od existujícího projektu jenkins.rb, nastává otázka, jaké výhody a nevýhody nabízí projekt jenkins.py oproti projektu jenkins.rb (pokud pomineme fakt, že je použit jiný jazyk pro realizaci modulů) a proč byla zvolena jiná cesta pro řešení podobné úlohy.

5.7.1 Výhody

- *Nebude třeba implementovat podporu XStream a Stapler pro Jython.* Volání Stapler metod bude delegováno prostřednictvím Java objektů. Podobně může být využit Java objekt pro serializaci (ukládání) dat pomocí knihovny XStream.

- *Netřeba nahrazovat nástroj Maven a jeho modul maven-hpi-plugin.* Téměř všechny operace nástroje Maven využívané při vývoji modulů v Javě budou využitelné i pro moduly v Pythonu (opět díky přítomnosti Java tříd). Patří sem příkazy:

  - „`mvn hpi:run`“
  - „`mvn package`“ (alias „`mvn hpi:hpi`“)
  - „`mvn release:prepare release:perform`“

- Přítomnost Java tříd v Python modulech dále umožní *využít stávající mechanismus serveru Jenkins pro vyhledávání a načítání modulů.* Jinak řečeno, modul python-wrapper nebude muset implementovat vlastní `ExtensionFinder` a `ClassLoader` pro Python moduly.

- *Možnost implementace pouze některých částí modulu v Pythonu.* Knihovnu modulu python-wrapper budou moci využít i moduly naprogramované v Javě a realizovat tak část své funkcionality v Pythonu. Tento kombinovaný přístup k vývoji je využitelný, pokud chce programátor modulu použít nějakou knihovnu napsanou v Pythonu, ale zároveň potřebuje, aby byl modul napsán v Javě (například využívá i jiných knihoven v Javě nebo není dostatečně znalý jazyka Python).

5.7.2 Nevýhody

- *Bude třeba naprogromavat jednoúčelový nástroj pro tvorbu wrapperů* pro všechny třídy realizující `ExtensionPoint` nebo odvozené ze třídy `Descriptor`.
- *Programátor modulu bude muset zvládat alespoň minimální základy jazyka Java.* Přítomnost Java tříd v Python modulech, přinášející nejvíce výhod, je zároveň zřejmou nevýhodou řešení.

Jelikož výhody návrhu značně převažují nad jeho nevýhodami, byla zvolena tato cesta realizace projektu jenkins.py.

Kapitola 6

Knihovna modulu python-wrapper

Tato kapitola se zabývá implementací knihovny, která je součástí modulu python-wrapper.

6.1 Stavba modulu python-wrapper

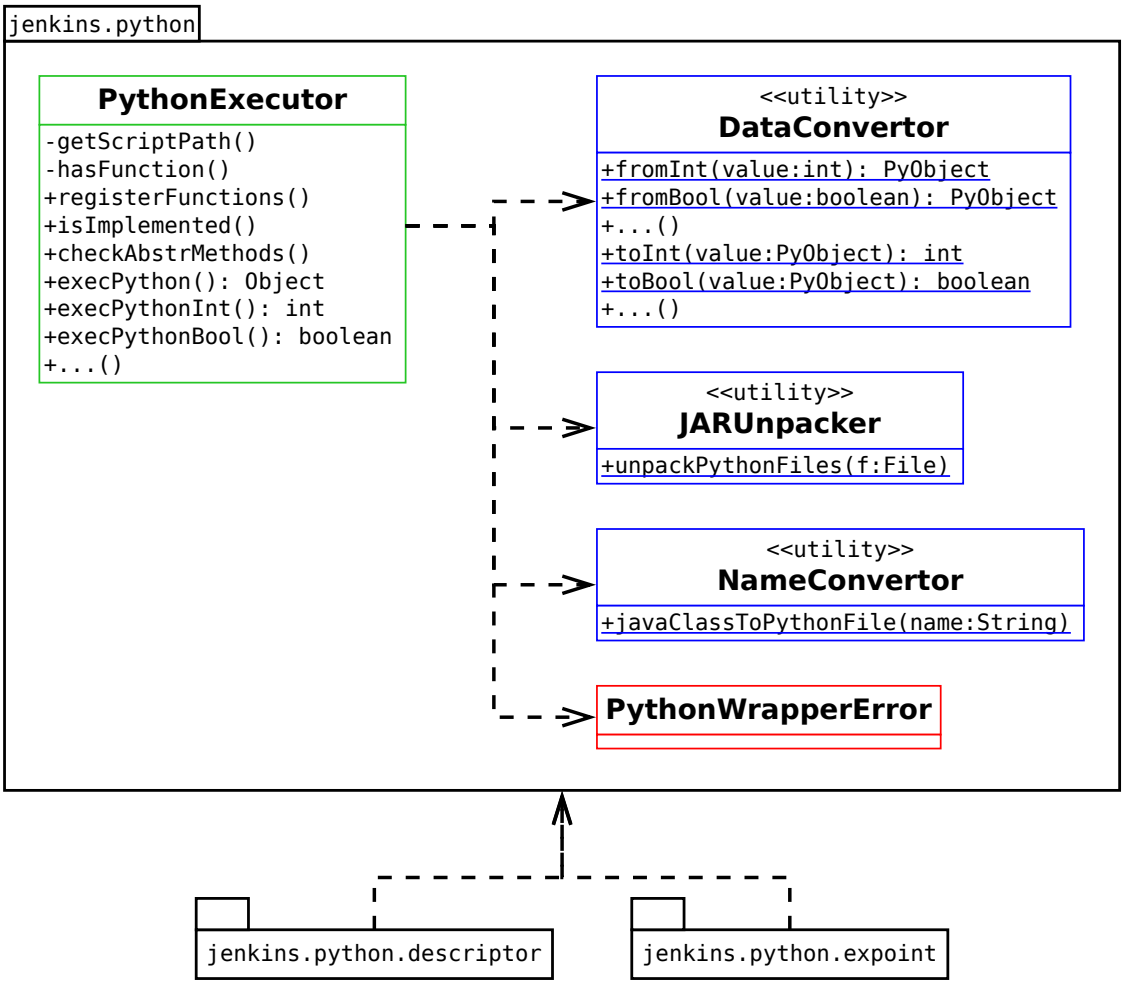
Modul python-wrapper se skládá z knihovny pro podporu spouštění skriptů v Pythonu, jejíž funkce je vysvětlena v této kapitole, a z obalových tříd (wrapperů) pro rozšiřitelné třídy a deskriptory systému Jenkins. Struktura a funkce obalových tříd je vysvětlena v následující kapitole 7. Knihovna je umístěna v balíčku `jenkins.python`, zatímco wrappery jsou umístěny v balíčcích `jenkins.python.descriptor` a `jenkins.python.expoint`. Stavba modulu je znázorněna na obrázku 6.1.

6.2 Struktura knihovny

Knihovna modulu python-wrapper sestává z několika spolupracujících tříd:

- `PythonExecutor`
  Jedná se o hlavní třídu knihovny. Obaluje objekt typu `PythonInterpreter` a na žádost vykonává funkce obsažené v načteném skriptu v Pythonu.

- `DataConvertor`
  Implementuje nástroj pro převod mezi datovými typy Pythonu a Javy.
  Pokud se v rámci kódu v Pythonu pracuje s třídami v Javě, datové typy jsou převáděny automaticky oběma směry. Programátor tvořící skripty v Pythonu tedy ani nemusí rozeznat, že pracuje s třídami jazyka Java. Tyto automatické převody jsou zajištěny interními operacemi interpretu Jython. Pokud se však v rámci Java kódu pracuje s třídami nebo obecně skripty v Pythonu, všechna data získaná/zasílaná z/do Python skriptů jsou zabalena v objektu typu `PyObject`.
  Pro převod objektů typu `PyObject` na bázové typy a objekty v Javě (a stejně tak opačným směrem) slouží nástroj `DataConvertor`. Vzhledem k tomu, že má jazyk Python mírně odlišné bázové typy než jazyk Java, byla sestavena převodní tabulka mezi jednotlivými typy v obou jazycích. Odpovídající reprezentace jsou definovány v tabulce 6.1.



Obrázek 6.1: Diagram tříd modulu python-wrapper.

Java	Python
`boolean`	`bool`
`double`	`float`
`float`	`float`
`long`	`long`
`int`	`int`
`short`	`int`
`byte`	`int`
`char`	`str` (první znak)
`[]`	`array.array`
`String`	`str`
`Object`	`object`

Tabulka 6.1: Převodní tabulka mezi typy jazyků Java a Python.

- `NameConvertor`
  Tato třída implementuje jednoduchý nástroj pro převod názvů tříd jazyka Java na odpovídající jména skriptů v jazyce Python. Například pro třídu s názvem `MojeTrida` zformuluje jméno skriptu `moje_trida.py`.

- `JARUnpacker`
  Nástroj slouží pro rozbalování Python skriptů z archivů formátu JAR. V zásadě je rozbalena celá složka `python` v kořenovém umístění v archivu do stejného adresáře, kde se archiv nachází.

- `PythonWrapperError`
  Jedná se o výjimku deklarující běhovou chybu, která nastala při načítání skriptu, při volání funkcí nebo při neexistující implementaci abstraktních metod.

6.3 Princip užití knihovny

Příklad 6.1

```
<build>
  <resources>
  ...
    <resource>
      <directory>src/main</directory>
      <includes>
        <include>**/*.py</include>
      </includes>
    </resource>
  ...
  </resources>
</build>
```

Vstupním rozhraním pro spouštění Python skriptů v modulech systému Jenkins je třída `PythonExecutor`. Aby bylo možno spouštět v rámci nějakého objektu jazyka Java skripty v Pythonu, musí být dodržen následující postup:

1. Všechny Python skripty, včetně potřebných knihoven, musí být umístěny v projektu ve zdrojovém podadresáři `src/main/python`.

2. V souboru s nastavením projektu nástroje Maven `pom.xml` musí být označeny všechny soubory s příponou `.py` jako statické zdroje. Tím je zajištěno, že se při překladu projektu pomocí nástroje Maven zabalí všechny skripty do balíčku `HPI` a později se, při instalaci modulu systémem Jenkins, přebalí do balíčku `JAR` s ostatními statickými zdroji a třídami jazyka Java. Příslušné nastavení je uvedeno v příkladu 6.1.

3. Soubor `pom.xml` musí obsahovat informaci o závislosti na modulu python-wrapper.

4. Objekt typu `PythonExecutor` musí být inicializován objektem, ze kterého má být spouštěn kód v Pythonu. Konstruktor třídy `PythonExecutor` přebírá jako jediný argument právě tento inicializační objekt. Podle názvu třídy inicializačního objektu je pak dohledán přidružený skript v Pythonu, neboli název souboru s Python skriptem musí odpovídat názvu třídy inicializačního objektu převedeného pomocí nástroje

třídy `NameConvertor`.
Při inicializaci objektu `PythonExecutor` je nejprve vytvořen nový interpret jazyka Python (objekt typu `PythonInterpreter`), jemuž je následně předána cesta ke skriptu, který má být načten. V případě nutnosti je ještě před načtením skriptu rozbalena složka `python` z archivu JAR pomocí nástroje `JARUnpacker`. Po načtení skriptu je nastavena globální proměnná `extension`, která odkazuje na inicializační objekt v Javě. Nakonec je zavolána funkce `init_plugin()` v načteném skriptu (pokud existuje) a tím je inicializace dokončena.

5. Volání funkcí v načteném Python skriptu je pak prováděno pomocí metod objektu `PythonExecutor` s prefixem `execPython`. Metody očekávají jako první argument řetězec s názvem volané funkce a následně seznam argumentů, které mají být funkci předány. Pokud jsou argumenty bázového typu, musí být převedeny pomocí nástroje `DataConvertor`. Podle zvolené varianty metody je pak vrácen výsledek volání funkce. Například metoda `execPythonFloat()` navrácí výsledek typu `float` (a proto i volaná funkce v přidruženém skriptu musí navracet výsledek typu `float`).

Při určování cesty k přidruženému skriptu v Pythonu je rozlišován stav, kdy je modul instalován standardním způsobem do systému Jenkins (skripty se musí nejprve vybalit z JAR archivu modulu) a stav, kdy je pouze testován pomocí příkazu „`mvn hpi:run`“ (skripty nejsou zabaleny v archivu, ale nachází se spolu s ostatními statickými zdroji v dočasném adresáři). To, v jakém stavu se spuštěný modul nachází, lze snadno zjistit podle cesty k Java třídě inicializačního objektu. V druhém případě obsahuje cesta koncovku `.jar`.

Třída `PythonExecutor` (tedy i knihovna modulu python-wrapper) může být využita dvojím způsobem. Jednak může programátor modulu použít třídu přímo podle výše uvedeného postupu a kontrolovat tak volání funkcí ve skriptu pomocí `execPython` metod (to je vhodné především pro kombinovaný vývoj Java/Python) a jednak lze použít wrappery, které zaobalují práci s objektem `PythonExecutor` a posouvají tak vývoj modulů blíže k Pythonu. O wrapperech pojednává kapitola 7.

6.3.1 Registrace funkcí

Třída `PythonExecutor` disponuje možností ověření existence funkcí v načteném skriptu. Existence funkcí je zjištěna metodou `registerFunctions()`, která přebírá pole názvů funkcí a pole s počty argumentů jednotlivých funkcí. Existence každé funkce je tímto ověřena a zaznamenána do statického pole. Metodou `isImplemented()` je pak možno dotázat se na existenci konkrétní funkce. Metoda přebírá jediný argument, a to ID funkce, které odpovídá umístění funkce v poli názvů funkcí zaslané metodě `registerFunctions()`. Tato funkcionalita je určena primárně pro wrappery, ale je možno ji aplikovat i při přímém užití objektu `PythonExecutor`.

6.3.2 Kontrola abstraktních metod

Pro kontrolu implementace abstraktních metod rodičovské rozšiřitelné třídy je možno využít veřejnou metodu `checkAbstrMethods()`. Použití metody je určeno primárně pro wrappery. Metoda přebírá pole s názvy funkcí v konvenci jazyka Python, pole s názvy metod v konvenci jazyka Java a pole s poli objektů typu `Class` (meta třída určující typ). Metoda kontroluje implementaci daných metod/funkcí ve třídě inicializačního objektu a v přidruženém skriptu. Pokud neexistuje implementace některé z metod ve třídě inicializačního objektu

ani implementace ekvivalentní funkce v přidruženém skriptu v Pythonu, je vyvolána běhová chyba typu `PythonWrapperError` se srozumitelnou zprávou. Programátorovi je tak jasně řečeno, jaká implementace chybí, a musí být bezpodmínečně doplněna.

6.4 Testování

Modul python-wrapper byl otestován automatickými testy poskytovanými modulem maven-hpi-plugin, které se spouští při překladu vyvíjeného modulu. Dále byl modul otestován přímo, využitím jeho funkcionality při vývoji modulu InstallShield do systému Jenkins. O modulu InstallShield pojednává kapitola 8. Díky testování bylo nalezeno množství kosmetických a několik kritických chyb, které byly následně opraveny.

Kapitola 7

Nástroj PWM

Tato kapitola se zabývá implementací nástroje pro generovaní obalových tříd pro rozšiřitelné třídy systému Jenkins a jejich deskriptory. Obalové třídy byly nazvány *wrappery* (z angl. wrapper – obal) a implementovaný nástroj pak Python Wrapper Maker (čes. tvůrce obalů pro Python), zkráceně PWM.

Nejdříve je vysvětlena struktura wrapperů a jejich funkce a následně je zdokumentován nástroj PWM, který slouží k tvorbě těchto tříd.

7.1 Wrappery

Wrappery jsou v kontextu této technické zprávy třídy, které dědí z rozšiřitelných tříd (tříd implementující rozhraní `ExtensionPoint`) systému Jenkins nebo z jejich deskriptorů. Jejich úlohou je poskytnout vrstvu pro třídy obsažené v modulech, které chtějí implementovat daný `ExtensionPoint` s možností realizace funkčních metod v jazyce Python. Wrappery se nazývají stejně jako jejich rodičovské třídy, ale obsahují sufix `PW` (z angl. Python Wrapper). Například pro rozšiřitelnou třídu systému Jenkins `Notifier` je k dispozici wrapper `NotifierPW`. Pokud chce tedy programátor modulu využít možnosti implementace metod v jazyce Python, musí jeho implementované rozšíření dědit z třídy `NotifierPW` namísto třídy `Notifier`. Pokud tak učiní, musí vložit do projektu skript v jazyce Python s názvem ekvivalentním implementované třídě, avšak v konvenci jazyka Python. Způsob pojmenování přidružených Python skriptů byl vysvětlen v kapitole 6.

Wrappery pro rozšiřitelné třídy se nacházejí v balíčku `jenkins.python.expoint` a wrappery pro deskriptory tříd v balíčku `jenkins.python.descriptor`. Všechny wrappery jsou součástí modulu python-wrapper, jehož struktura a funkce byla vysvětlena v kapitole 6.

7.1.1 Struktura obecného wrapperu

Pokud se podíváme blíže na strukturu a obsah obecného wrapperu, lze jej charakterizovat v několika bodech:

- Každý wrapper přímo dědí z `ExtensionPoint` třídy nebo z deskriptoru. Pro každý `ExtensionPoint` a deskriptor existuje právě jeden wrapper.
- Všechny wrappery jsou abstraktní. Nelze je tedy inicializovat, slouží pouze jako mezivrstva pro třídy v terminálních modulech.
- Wrapper obsahuje jediný atribut, a to objekt typu `PythonExecutor`.

- Všechny zděděné metody, které jsou označeny v rodičovských třídách modifikátorem `public` nebo `protected`, wrapper přepisuje. Vlastní funkcionalita metod však není implementována, wrapper pouze kontroluje výskyt implementace ekvivalentní funkce v přidruženém Python skriptu (pomocí metody `isImplemented()` objektu `PythonExecutor`) a pokud taková implementace existuje, je zavolána. Pokud implementace funkce v přidruženém skriptu neexistuje, je namísto ní zavolána metoda rodičovské třídy.
  Všechny argumenty předané metodě jsou pouze delegovány funkci nebo rodičovské implementaci metody. Pokud je však volána funkce v přidruženém skriptu v Pythonu s argumenty bázového typu, jsou tyto argumenty převedeny pomocí nástroje `DataConvertor` na objekty typu `PyObject`. Stejně tak výsledek volání metody nebo funkce je pouze vrácen zpět volajícímu objektu.
  Vzhledem k odlišné konvenci pojmenování metod/funkcí v jazycích Python a Java odlišují i wrappery názvy Java metod od názvů Python funkcí. Pokud například wrapper přepisuje metodu s názvem `mojeMetoda()`, hledá pak v přidruženém Python skriptu výskyt funkce s názvem `moje_metoda()`.

- Většina rozšiřitelných tříd a všechny deskriptory jsou *abstraktními třídami* a taktéž deklarují některé metody jako abstraktní. Všechny abstraktní metody wrappery implementují, a to tím způsobem, že volají ekvivalentní funkci v přidruženém Python skriptu, aniž by kontrolovaly její výskyt (neexistuje totiž rodičovská implementace, která může být zavolána v případě neexistence funkce). Implementace všech původně abstraktních metod v terminálním modulu je kontrolována hromadně v metodě `init_python()` (viz dále).

- Přístup k původním implementacím metod rodičovských tříd je zajištěn veřejnými metodami, jejichž název začíná prefixem `super` a končí původním názvem volané metody. Tyto metody byly do wrapperů přidány proto, aby mohly být snadno volány původní rodičovské metody z externích objektů (kterými jsou i objekty typu `PythonInterpreter`). Pakliže by tyto metody nebyly definovány, přístup k rodičovským implementacím metod z Python skriptů by se mohl odehrávat pouze složitým způsobem pomocí tříd z balíčku `java.lang.reflect` ze standardní knihovny jazyka Java.

- Wrapper dále obsahuje veřejné metody, které obalují volání `execPython` metod privátního objektu `PythonExecutor`. Jejich účelem je poskytnout podtřídám možnost ruční delegace volání funkcí v Python skriptu. Tato vlastnost se hodí zejména, pokud třída v terminálním modulu implementuje nějaké rozhraní nebo definuje metodu kvůli nějakému volání z uživatelského rozhraní pomocí Stapler knihovny (tyto metody nemohou být wrapperem obaleny, jelikož není dopředu znám jejich název).

- Konstruktory wrapperu pouze předávají své argumenty ekvivalentnímu konstruktoru nadřazené třídy. Pokud programátor vyžaduje, aby se v konstruktoru třídy v terminálním modulu odehrávala nějaká činnost, musí tak učinit voláním `execPython()` metody.

- Před jakoukoli činností wrapperu související s přidruženým Python skriptem se zavolá privátní metoda `init_python()`. V této metodě se nejprve inicializuje objekt třídy `PythonExecutor`, následně se ověří implementace všech abstraktních metod voláním

`checkAbstrMethods()` a nakonec se registrují všechny abstraktní, veřejné a chráněné metody voláním `registerFunctions()`.

7.1.2 Tvorba wrapperů

Při obecném pohledu na wrappery vyvstává otázka, proč je jednoduše nevytvořit ručně a proč místo toho implementovat nástroj, který je generuje automaticky. Vývoj nástroje pro tvorbu wrapperů se vyplatí z několika důvodů:

1. Strukturu každého wrapperu je možné automaticky odvodit pro každou rozšiřitelnou třídu i pro všechny deskriptory. Tvorba wrapperů se tedy stává jednotvárnou prací, kterou lze automatizovat.

2. Počet potřebných wrapperů a jejich objem je dosti velký na to, aby se vyplatila implementace nástroje, který obstará tuto práci namísto lidských zdrojů. Aktuální verze modulu python-wrapper obsahuje 114 wrapperů pro rozšiřitelné třídy a 36 wrapperů pro třídy deskriptorů. Dohromady dosahují všechny wrappery objemu více než 45 tisíc řádků kódu v jazyce Java. Oproti tomu nástroj PWM, popisovaný dále v této kapitole, obsahuje pouze dva tisíce řádků kódu. Časová úspora je tedy značná.

3. Nástroj je znovu využitelný. Tvorba wrapperů je jednorázovou prací pouze zdánlivě. S každou novou verzí serveru Jenkins mohou vznikat, a také často vznikají, nové rozšiřitelné třídy. Zároveň může docházet k aktualizaci stávajících rozšiřitelných tříd (při zachování zpětné kompatibility), typicky přidáním nových metod. Wrappery se tedy budou muset pravidelně aktualizovat s rozumným časovým odstupem, například pro každou verzi serveru Jenkins s rozšířenou podporou (angl. Long Term Support, zkráceně LTS), a nástroj PWM bude tudíž využíván pravidelně.

7.2 Nástroj PWM

Nástroj pro tvorbu wrapperů byl nazván PWM (z angl. Python Wrapper Maker) a byl implementován v jazyce Java. PWM je samostatnou spustitelnou aplikací s jedinou závislostí, a to na knihovně Eclipse JDT. Implementační jazyk Java byl zvolen z toho důvodu, že hlavní operace programu zahrnují práci s třídami jazyka Java, ať už v konkrétní nebo abstraktní podobě, k čemuž se hodí samotná Java nejlépe. Nástrojem pro definici projektu byl zvolen Maven, protože umožňuje snadnou deklaraci závislosti aplikace na knihovnách jazyka Java a komunita kolem projektu Jenkins je s tímto nástrojem sžita.

Aplikace rozeznává tři argumenty. Prvním je argument „`-v`“, který aplikaci říká, že má běžet v „upovídaném režimu“ (angl. verbose). Druhým argumentem je „`-h`“, který pouze navádí aplikaci, aby vytiskla nápovědu (angl. help) na standardní výstup a ukončila svou činnost. Třetím a zároveň jediným povinným argumentem je „`-i`“, za nímž musí následovat cesta ke složce (angl. input directory) se zdrojovým kódem serveru Jenkins. Všechny argumenty mají své doslovné synonymum („`--verbose`“, „`--help`“ a „`--input-dir`“).

7.2.1 Eclipse JDT

Eclipse Java development tools[1] (zkráceně Eclipse JDT) je knihovna určená pro zásuvné moduly do aplikace Eclipse[2], vývojového prostředí pro jazyk Java. Knihovna nabízí mimo

[1] Eclipse JDT – http://www.eclipse.org/jdt
[2] Eclipse – http://www.eclipse.org

jiné i třídy pro práci s abstraktními syntaktickými stromy (angl. *abstract syntax trees*, zkr. AST) jazyka Java, jejichž činnost je potřebná a využívaná v nástroji PWM. Přestože je knihovna určena primárně pro zásuvné moduly do prostředí Eclipse, lze ji využít i samostatně, bez závislosti na jádru projektu Eclipse, čehož se užívá i v nástroji PWM. Eclipse JDT se nachází v centrálním repozitáři projektu Maven, a proto je indikace závislosti na této knihovně otázkou několika řádků v konfiguračním souboru `pom.xml`. Tento oddíl čerpá z článku [9] a dokumentace [18].

Možnosti knihovny Eclipse JDT, které jsou využívané nástrojem PWM, můžeme rozdělit do tří kategorií:

1. *Převod kódu v jazyce Java na abstraktní syntaktické stromy (syntaktická analýza)*
2. *Zkoumání a úprava AST*
   Úprava a průzkum abstraktních syntaktických stromů zahrnuje přidávání a odstraňování uzlů ve stromě, modelování kompletně nových stromů a uzlů, úpravu atributů jednotlivých uzlů, změnu pořadí uzlů v rodičovských uzlech, vzájemné porovnávání podstromů a listových uzlů mezi sebou, vyhledávání uzlů ve stromě podle různých parametrů, kopírování listových uzlů nebo celých podstromů apod…
3. *Převod abstraktních syntaktických stromů na řetězce kódu jazyka Java (generování kódu)*
   Knihovna umožňuje mimo jiné i nastavovat formátování výsledného kódu.

Výčet nejdůležitějších tříd knihovny Eclipse JDT použitých v programu PWM zahrnuje jednak funkční třídy a jednak třídy datové, reprezentující uzly v AST:

- `AST`
  Třída `AST` reprezentuje abstraktní syntaktický strom jako celek. Objekt typu `AST` je vstupním prvkem k dalším operacím nad abstraktním syntaktickým stromem.
- `ASTParser`
  Tato funkční třída slouží pro převod kódu (syntaktickou analýzu, angl. *parse*) jazyka Java na abstraktní syntaktický strom.
- `ASTMatcher`
  Pro posuzování shody listových uzlů, ale i celých podstromů slouží objekt třídy `ASTMatcher`.
- `Document`
  Reprezentuje abstraktní dokument obsahující kód v jazyce Java. Mimo prostředí Eclipse může být inicializován prostým textovým řetězcem a stejně tak je možné z něho získat konkrétní obsah ve formě textového řetězce.
- `TextEdit`
  Objekt typu `TextEdit` obsahuje zaznamenané změny, které se odehrály v AST reprezentujícím dokument. V průběhu života objektu třídy `AST` jsou tyto změny zaznamenávány a posléze mohou být objektem typu `TextEdit` zpětně aplikovány do dokumentu.
- `ASTNode`
  `ASTNode` je abstraktní třída, která reprezentuje obecný uzel v AST. Všechny třídy reprezentující nějaký typ uzlu musí dědit z této třídy.

- `CompilationUnit`
  Jedná se o kořenový prvek AST. Každý korektní dokument (kód) je převeden objektem typu `ASTParser` na objekt třídy `AST`, z něhož lze získat kořenový prvek stromu, uzel typu `CompilationUnit`. Objekt `CompilationUnit` tedy reprezentuje kód celého dokumentu (zdrojového souboru jazyka Java) včetně deklarace balíčku, importů a typu.

- `TypeDeclaration`
  Tato třída reprezentuje deklaraci typu, tedy definici třídy jazyka Java. Každý validní uzel typu `CompilationUnit` musí obsahovat jednu kořenovou definici třídy.

- `MethodDeclaration`
  Reprezentuje definici metody uvnitř definice třídy. Každý uzel typu `TypeDeclaration` může vlastnit seznam definic metod, což ostatně vyplývá z gramatiky jazyka Java.

- `PackageDeclaration`
  Deklarace balíčku, v jakém se definovaná třída nachází, je uchovávána v objektu typu `PackageDeclaration`. Uzel typu `CompilationUnit` musí obsahovat právě jeden uzel typu `PackageDeclaration`.

- `ImportDeclaration`
  Reprezentuje uzel deklarující užití příkazu import. Každý uzel typu `CompilationUnit` může obsahovat více uzlů typu `ImportDeclaration`.

- `Type`
  Abstraktní třída reprezentující užití nějakého typu v kódu se nazývá `Type`. Potomci této třídy jsou neabstraktní třídy reprezentující užití konkrétního druhu datového typu. Například se jedná o třídu `PrimitiveType` deklarující užití primitivního typu.

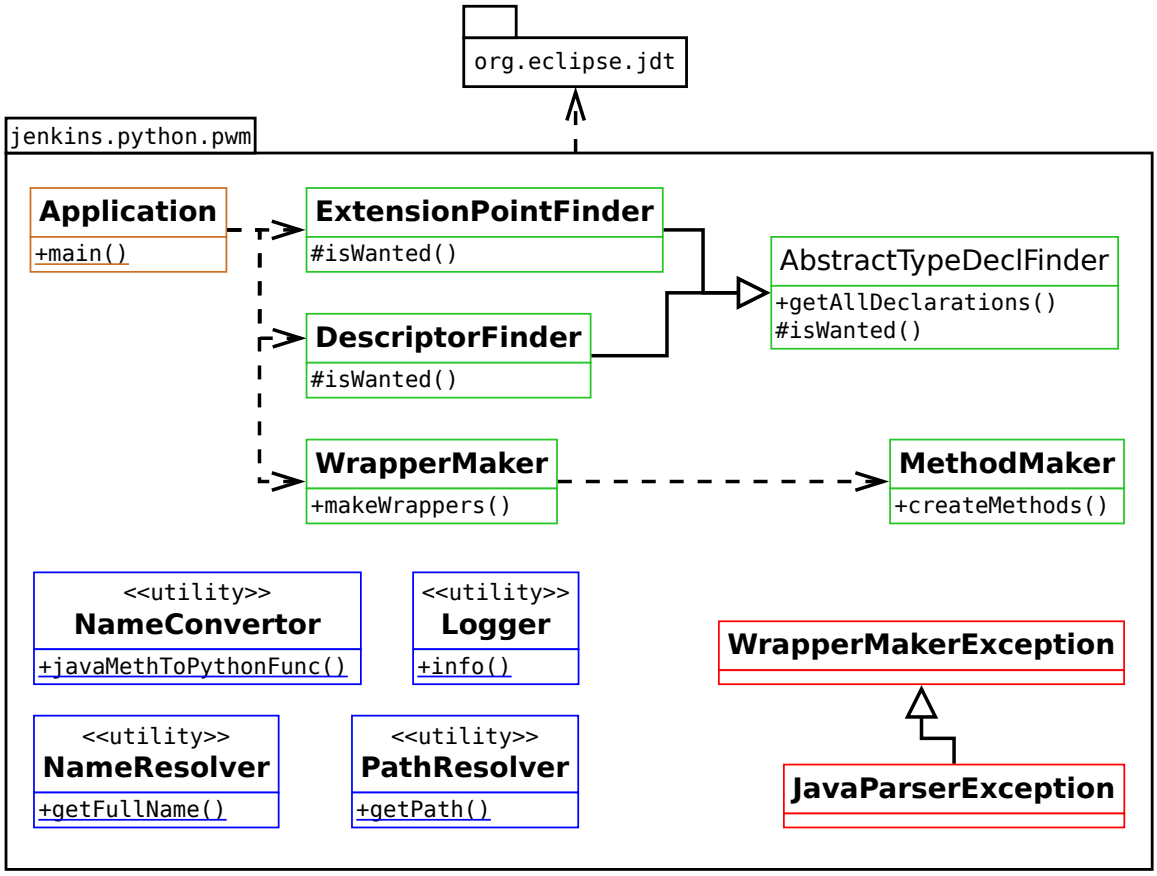
- `Name`
  Třída `Name` je abstraktní typ reprezentující použití nějakého jména v kódu. Instance potomků této třídy jsou listovými uzly AST (například objekty typu `SimpleName`).

- `Expression`
  Abstraktní třída `Expression` reprezentuje obecný výraz v jazyce Java.

- `Statement`
  Typ `Statement` uchovává příkaz jazyka Java (jedná se také o abstraktní třídu). Příkazy jsou základním stavebním kamenem kódu v jazyce Java a můžou jimi být mimo jiné i výrazy.

7.2.2 Struktura aplikace PWM

Všechny třídy nástroje PWM jsou umístěny v balíčku s názvem `jenkins.python.pwm`. V balíčku jsou pouze funkční třídy, užitkové třídy a třídy výjimek. Veškerá data zpracovávaná programem jsou uchovávána v objektech tříd z balíčku `org.eclipse.jdt`, případně ve standardních kontejnerech jazyka Java. Struktura aplikace je zachycena v diagramu na obrázku 7.1.



Obrázek 7.1: Diagram tříd nástroje PWM.

Užitkové třídy

Užitkové třídy (angl. utility classes) jsou znovupoužitelné třídy pouze se statickými metodami. Třídy tohoto druhu mají typicky jednoúčelový a přesně definovaný význam. Slouží funkčním třídám v rámci celé aplikace při řešení konkrétních problémů nebo zajišťují specifickou činnost, která může být využita v průběhu celého běhu programu.

- `Logger`
  Tato třída zajišťuje protokolování činnosti napříč celou aplikací. Její chování je do jisté míry ovlivněno vstupním parametrem aplikace – `verbose`. Nabízí několik statických metod pro zaznamenávání různých druhů informací:
  - `info()`
    Zaznamenání obecné informace o činnosti programu zajišťuje metoda `info()`.
  - `verbose()`
    Používá se pro zaznamenání podrobnějších informací o činnosti programu. Informace se zaznamená, pouze pokud je nastaven příznak daný parametrem `verbose`.
  - `warning()`
    Varovná zpráva o nebezpečné, ale nekritické události je předána této metodě.

  - `error()`
    Varovná zpráva o události, která zabraňuje další činnosti programu a vynucuje jeho ukončení je předána metodě `error()`.

- `NameConvertor`
  Slouží pro převod názvů metod v konvenci jazyka Java na názvy funkcí nebo metod v konvenci jazyka Python. Například pro název metody `nejakaMetoda()` vytvoří jméno `nejaka_metoda()`.

- `NameResolver`
  Třída vyhodnotí plně kvalifikované jméno třídy zapsané v konvenci jazyka Java pro objekty typu `Name`, `Type` a `TypeDeclaration`. Celá jména tříd jsou důležitá jednak pro jednoznačnou identifikaci a jednak pro dohledání zdrojového souboru, kde je třída definována. Každý uzel v AST obsahuje odkaz na kořenový uzel typu `CompilationUnit`. Z tohoto důvodu lze tedy zjistit deklaraci balíčku a všechny importované třídy a balíčky v souboru, ve kterém se uzel, jehož celé jméno se vyhodnocuje, nachází. Z deklarace balíčku lze zjistit celé jméno definice typu (objekt třídy `TypeDeclaration`). Ze seznamu importovaných tříd a balíčků lze pak vyhodnotit celé jméno pro použité typy v kódu programu (objekty tříd `Name` a `Type`).
  Například pro typ `Hudson` použitý v kódu programu se vyhodnotí jeho celý název `hudson.model.Hudson`.

- `PathResolver`
  Vyhodnocuje úplnou cestu k definičnímu souboru pro předané plně kvalifikované názvy tříd. Například pro třídu s názvem `hudson.model.Hudson` vyhodnotí cestu `/home/.../hudson/model/Hudson.java` (na platformě GNU/Linux). Třída umí vyhodnocovat i jména vnitřních tříd (jako je např. `jenkins.Trida.VnitrniTrida`).

Funkční třídy

- `Application`
  Třída je vstupním bodem aplikace, obsahuje statickou metodu `main()`, jejím účelem je pouze kontrola parametrů, tisk nápovědy a delegace řízení ostatním funkčním třídám. Jméno této třídy jako vstupního bodu musí být deklarováno v konfiguračním souboru `pom.xml`.

- `AbstractTypeDeclFinder`
  Tato abstraktní třída definuje obecnou funkční třídu, která realizuje vyhledávač nějaké deklarace typu ve zdrojovém kódu systému Jenkins. Třída obsahuje schopnost rekurzivně procházet adresář se zdrojovým kódem jádra serveru Jenkins a správně identifikovat nějaký druh definice třídy. To, zda je nalezená třída ta správná a chtěná, určuje abstraktní metoda `isWanted()`, kterou musí všechny podtřídy implementovat. `AbstractTypeDeclFinder` umí také vyhodnotit a nalézt všechny předky nalezené třídy.

- `ExtensionPointFinder`
  Dědí z abstraktní třídy `AbstractTypeDeclFinder` a implementuje vyhledávač rozšiřitelných tříd (tříd realizujících rozhraní `ExtesionPoint`).

- `DescriptorFinder`
  Stejně jako `ExtensionPointFinder` dědí z abstraktní třídy `AbstractTypeDeclFinder`.

`DescriptorFinder` ovšem implementuje vyhledávač deskriptorů (deskriptorem je třída `Descriptor` a dále všechny třídy, které jsou z této třídy odvozeny).

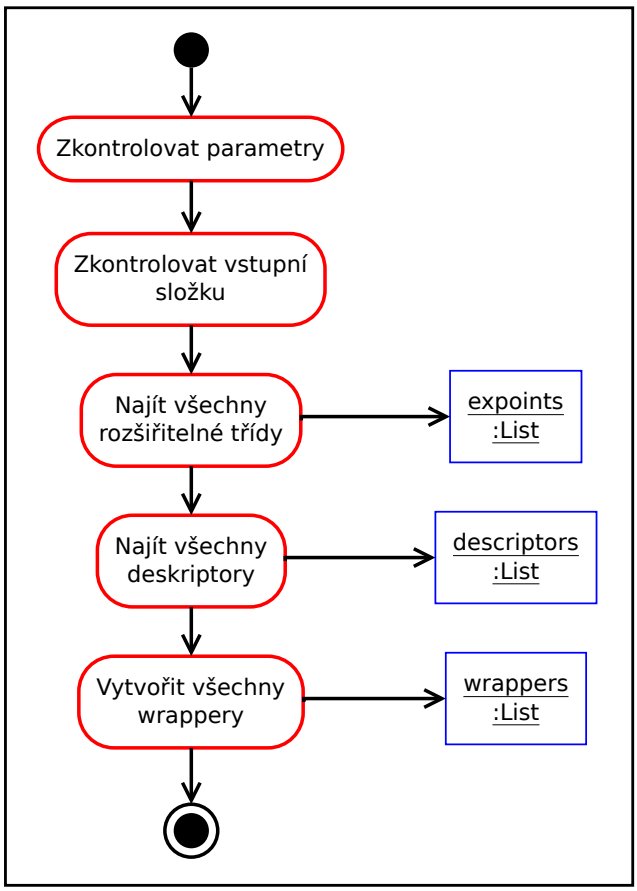
- `WrapperMaker`
  Tato třída slouží k tvorbě wrapperů k nalezeným třídám. Má na starosti sestavení kostry wrapperu (přesněji sestavení celého wrapperu, kromě jeho metod).

- `MethodMaker`
  Tato třída vytváří metody pro konkrétní wrapper.

Výjimky

- `WrapperMakerException`
  Tato výjimka deklaruje obecnou chybu v programu PWM.

- `JavaParserException`
  Dědí ze třídy `WrapperMakerException` a deklaruje výjimečnou událost, která nastala při syntaktické analýze (například může jít o chybný soubor nebo špatně nastavená přístupová práva k souboru).



Obrázek 7.2: Diagram aktivit znázorňující běh programu PWM.

7.2.3 Proces tvorby wrapperů

Aplikace PWM sestává z několika zřetězených úloh a realizuje tak proces tvorby wrapperů. Vstupem procesu je složka se zdrojovým kódem systému Jenkins. Výstupem jsou pak wrappery pro všechny deskriptory a rozšiřitelné třídy. Zjednodušený proces zahrnující nejdůležitější úlohy aplikace je znázorněn na obrázku 7.2. Sestává z následujících fází:

1. *Kontrola parametrů*
   Aplikace zkontroluje parametry předané uživatelem. Pokud uživatel zadal přepínač žádající o tisk nápovědy, aplikace pouze vytiskne nápovědu k aplikaci a skončí. V opačném případě se kontroluje přítomnost povinného parametru určujícího cestu ke složce se zdrojovým kódem serveru Jenkins.

2. *Kontrola vstupní složky*
   Aplikace zkontroluje přítomnost podadresáře „`./core/src/main/java/`“ ve složce zadané uživatelem. V tomto podadresáři se nachází zdrojový kód jádra aplikace Jenkins, kde jsou mimo jiné i zdrojové soubory rozšiřitelných tříd a deskriptorů v jazyce Java.

3. *Nalezení všech rozšiřitelných tříd*
   K nalezení rozšiřitelných tříd je využita funkční třída `ExtensionPointFinder`. Aplikace rekurzivně prochází složku se zdrojovým kódem jádra a hledá soubory s příponou `.java`. Pokud je takový soubor nalezen, je nad ním provedena syntaktická analýza pomocí objektu `ASTParser`. Pokud je ve zpracovaném AST obsažena třída (uzel typu `TypeDeclaration`), ať už kořenová nebo vnitřní, která implementuje rozhraní `ExtensionPoint`, je přidána do seznamu. K této třídě jsou následně rekurzivně dohledáni a zpracováni do AST také všichni její předci. Výstupem této fáze je tedy seznam objektů typu `TypeDeclaration`, které definují jednak rozšiřitelné třídy a jednak jejich předky.

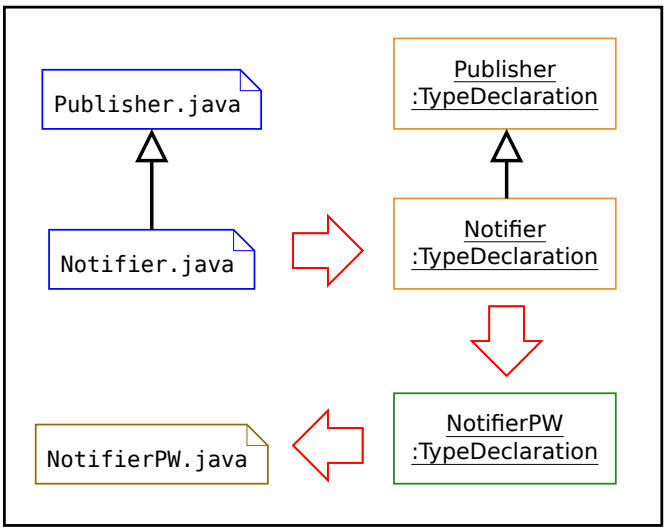
4. *Nalezení všech deskriptorů*
   Tato fáze je podobná té předchozí s tím rozdílem, že jsou vyhledány třídy, které dědí ze třídy `Descriptor`. Do výsledného seznamu je zahrnuta i třída `Descriptor` a to z toho důvodu, že terminální moduly mohou přímo využívat tuto třídu pro implementaci deskriptoru a proto musí být vytvořen wrapper i pro ni.

5. *Vytvoření wrapperů*
   V této fázi se ke každé nalezené rozšiřitelné třídě a ke každému deskriptoru vytvoří příslušný wrapper pomocí tříd `WrapperMaker` a `MethodMaker`. Nejprve je vytvořena od základu reprezentace wrapperu v AST a tato reprezentace je následně převedena na kód v jazyce Java a uložena do souboru s příponou `.java`. Soubory jsou ukládány do podadresářů `expoint` a `descriptor` v aktuálním pracovním adresáři.

Tvorba jediného wrapperu v rámci celého výše definovaného procesu je znázorněna na obrázku 7.3. Jedná se o vertikální průřez procesu. Jako příklad posloužila rozšiřitelná třída `Notifier`.

Nejprve je nalezen soubor `Notifier.java`, který je následně převeden syntaktickou analýzou na AST. V souboru je nalezena kořenová definice typu `Notifier`, která implementuje rozhraní `ExtensionPoint`. Typ je tedy správně identifikován jako rozšiřitelná třída. K této třídě jsou následně nalezeny všichni předci a taktéž převedeni na AST (na schématu se nachází z úsporných důvodů pouze jediný předek, `Publisher`). Ze zřetězených objektů typu

`TypeDeclaration` je vytvořen nový objekt reprezentující wrapper `NotifierPW`, a to včetně rodičovského AST uzlu typu `CompilationUnit`. Tento wrapper je následně převeden na kód v jazyce Java a uložen do souboru `NotifierPW.java`.



Obrázek 7.3: Proces tvorby wrapperu `NotifierPW.java`.

Tvorba kostry wrapperu

Při tvorbě nového wrapperu pomocí objektu třídy `WrapperMaker` je nejprve vytvořen objekt typu `CompilationUnit` z prázdného dokumentu. Prvotně se tedy jedná o reprezentaci prázdného kódu v jazyce Java. Do tohoto objektu je přidán uzel deklarace balíčku, v němž se wrapper nachází (`jenkins.python.expoint` nebo `jenkins.python.descriptor`). Následně jsou zkopírovány z rozšiřitelné třídy (nebo deskriptoru) a jejích předků všechny uzly reprezentující import tříd a balíčků (duplicitní importy jsou odstraněny). Nakonec je vytvořena třída (uzel typu `TypeDeclaration`), která reprezentuje nový wrapper, a přidána do kořenového uzlu typu `CompilationUnit`. Tato třída dědí z původní třídy, je abstraktní a obsahuje jediný atribut, objekt typu `PythonExecutor`. Název nové třídy obsahuje sufix `PW`.

Tvorba metod wrapperu

K tvorbě metod wrapperu se využívá třídy `MethodMaker`. Nejprve jsou nalezeny všechny metody původní třídy a jejích předků, které mají modifikátor přístupu `public` nebo `protected` a všechny konstruktory původní třídy. Poté jsou vytvořeny konstruktory, které pouze volají deklarativně ekvivalentní konstruktory původní třídy. Deklarace nalezených metod jsou zkopírovány do wrapperu, přičemž je přidána anotace `@Override`. Způsob definice těchto metod však závisí na tom, zda byly původně abstraktní či nikoli. Dále jsou vytvořeny metody s prefixem `super`, které volají rodičovskou implementaci, a metody s prefixem `execPython`, obalující volání objektu `PythonExecutor`. Nakonec je vytvořena od základu metoda `initPython()`, včetně příkazů pro kontrolu implementace abstraktních metod a registrace funkcí. Definice všech metod odpovídá struktuře wrapperu popsané v oddíle 7.1.1.

Přenos typových parametrů

Některé rozšiřitelné třídy a všechny deskriptory jsou generickými typy. Automatické kopírování deklarací metod z původních generických tříd do wrapperů vedlo k chybně definovaným typům. To bylo zapříčiněno konkretizací typových parametrů v některých rozšiřitelných třídách a deskriptorech. Deklarace metod zkopírované z rodičovských tříd totiž obsahovaly obecnější typy, než jaké byly definovány v rozšiřitelných třídách a terminálních deskriptorech. To vedlo k chybám při kompilaci wrapperů. Například obecný deskriptor `Descriptor` obsahuje typový parametr `T`, kdežto deskriptor `NodeDescriptor` konkretizuje tento typový parametr na typ `Node`. Deklarace metod zkopírované do wrapperu `NodeDescriptorPW` ze třídy `Descriptor` pak obsahovaly generický typ `T`, přestože už byl tento typ konkretizován třídou `NodeDescriptor` na třídu `Node`.

Z tohoto důvodu byl implementován mechanismus pro přenesení konkretizovaných typových parametrů do rodičovských tříd. Tento mechanismus se aplikuje automaticky ihned po vyhledání všech předků nalezené třídy. Veškeré použití parametrického typu v rodičovských třídách je nahrazeno konkretizovaným typem, který definuje finální rozšiřitelná třída nebo deskriptor. Deklarace metod zkopírované z takto upravených tříd už obsahují správné typy a k typovým chybám při překladu wrapperů nadále nedochází.

7.3 Testování

Výstupy, které produkuje nástroj PWM, byly otestovány zároveň s modulem python-wrapper. To je umožněno tím, že jsou všechny wrappery součástí tohoto modulu. Testování modulu python-wrapper je popsáno v oddílu 6.4. Díky testům výstupu aplikace PWM bylo odhaleno množství chyb v samotné aplikaci. Všechny nalezené chyby byly následně opraveny.

7.4 Ruční úprava wrapperů

Přestože nástroj PWM generuje wrappery automaticky, před jejich začleněním do modulu python-wrapper jsou nutné další ruční úpravy. Bez těchto úprav neproběhne překlad wrapperů správně. Tyto korekce se týkají pouze menšího množství wrapperů a většinou nejdou aplikovat genericky, takže by se automatizace těchto úprav buď nevyplatila, nebo by zavedla do kódu nástroje příliš mnoho specificity. Všechny úpravy lze snadno identifikovat, protože bez jejich aplikace nahlásí překlad modulu python-wrapper chybu.

Použitá verze knihovny Eclipse JDT obsahuje chybu, která zapříčiňuje nenalezení některých statických importů při syntaktické analýze. Tyto importy jsou pak použity bez klíčového slova `static`, což mírně změní jejich význam. Knihovna dále vrací metody vnitřního `enum` typu, pokud takový vnitřní typ třída obsahuje, jako vlastní metody třídy. Deklarace těchto metod jsou následně zkopírovány do wrapperu, což zapříčiňuje chybu při překladu.

Další ruční úpravy jsou například důsledkem importu různých typů se stejným jménem. V případě použití tohoto jména pak překladač nepozná, o který typ se jedná. Celková ruční úprava wrapperů však nepřesáhne dvě hodiny čistého času.

Kapitola 8

Nástroj PPSM a InstallShield modul

Tato kapitola se věnuje implementaci nástroje PPSM, což je skript na tvorbu šablon nových modulů v jazyce Python, a dále popisuje implementaci modulu InstallShield do systému Jenkins.

8.1 Nástroj PPSM

Pokud vývojář zvolí pro implementaci nového modulu jazyk Java, má k dispozici nástroj Maven a jeho zásuvný modul maven-hpi-plugin pro rychlou tvorbu šablony nového projektu (příkaz „`mvn hpi:create`“). V případě užití jazyka Python však vygenerovaná šablona vyžaduje další dodatečné generické úpravy, než je možné začít vyvíjet samotný modul. Z tohoto důvodu byl implementován skript *PPSM* (z angl. Python Plugin Skeleton Maker – Tvořič kostry Python modulu), který nahrazuje příkaz „`mvn hpi:create`“ pro vývojáře, kteří chtějí vyvíjet modul v Pythonu.

Nástroj PPSM byl implementován v jazyce Python verze 3. Skript je interaktivní a vyžaduje od uživatele zadání jména nového modulu. Po získání jména od uživatele skript zkopíruje generickou šablonu modulu do aktuálního pracovního adresáře. Následně je jméno zadané uživatelem vepsáno do šablony v různých formách. Upraveno je například jméno Python skriptu, jméno názvu Java třídy, jméno balíčku nebo název modulu v konfiguračním souboru `pom.xml`. V závislosti na jméně modulu jsou upraveny i další položky v souboru `pom.xml` (například adresa umístění repozitáře na serveru GitHub). Takto upravená šablona modulu je pak připravena k okamžitému zabalení nebo testování pomocí nástroje Maven.

Samotný modul v šabloně implementuje jednoduchý vzorový překladač (angl. builder), který pouze protokoluje svou činnost. Z uživatelského rozhraní je implementováno jednoduché globální nastavení a nastavení překladu pro konkrétní úlohu. Funkcionalita překladače je tedy ekvivalentní s překladačem, který implementuje vzorový modul vygenerovaný příkazem „`mvn hpi:create`“ s tím rozdílem, že funkční metody jsou implementovány v Pythonu.

8.2 InsttallShield modul

Kvůli ověření funkčnosti implementovaných nástrojů pro vývoj modulů v jazyce Python bylo nutné vytvořit užitečný modul, který tyto nástroje využívá (a je tedy naprogramován

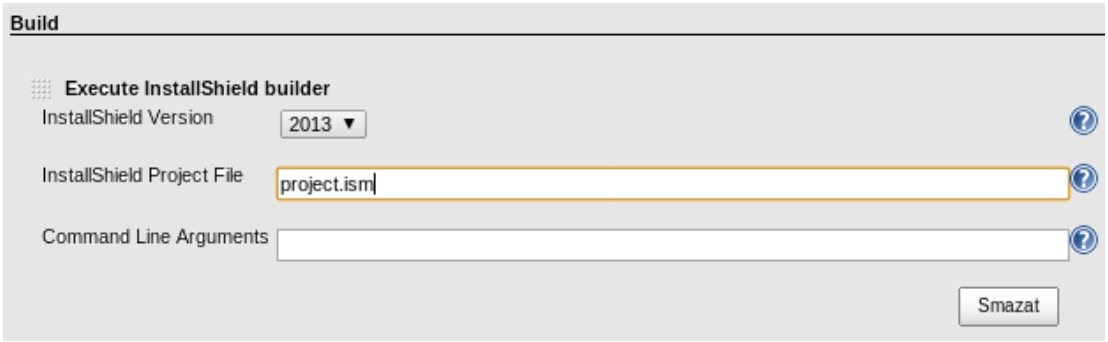
v jazyce Python). Z tohoto důvodu byl vznesen dotaz na uživatelském fóru[1] serveru Jenkins, jaký modul komunitě schází a jaké požadavky by na něj měly být kladeny. Z vícero nápadů, o něž se uživatelé podělili, byl vybrán modul, který má za úkol sestavovat projekty nástroje InstallShield a tento modul byl následně implementován.

8.2.1 Nástroj InstallShield

Nástroj InstallShield[2] je komerční aplikace pro vývojáře od společnosti Flexera Software[3], která slouží pro tvorbu instalátorů aplikací pro operační systém MS Windows. InstallShield umožňuje tvorbu instalátorů jednoduchým způsobem pomocí intuitivního grafického uživatelského rozhraní. Projekty nástroje InstallShield lze ukládat do binárních souborů s koncovkou `.ism` nebo `.ise`.

Kromě grafického uživatelského rozhraní obsahuje InstallShield také sadu řádkových nástrojů. Jedním z těchto nástrojů je i `ISCmdBld.exe`, který je navržen pro neinteraktivní automatickou tvorbu instalátorů. Jeho použití však předpokládá předem připravený projekt ve formátu `.ism` nebo `.ise`. Nástroj je určen pro integraci do jiných systémů pro vývoj softwaru nebo pro začlenění do automatických skriptů. Jediným povinným argumentem nástroje je „`-p`“, za kterým musí následovat relativní nebo absolutní cesta k projektovému souboru `.ism` nebo `.ise`. Jeho činnost je však možno řídit také dalšími argumenty.

Přestože je InstallShield komerční aplikací, nabízí firma Flexera Software ke stažení i zkušební časově omezenou verzi, jejíž funkcionalita je ekvivalentní verzi placené. Tato zkušební verze byla využita při vývoji modulu InstallShield do serveru Jenkins. [1]



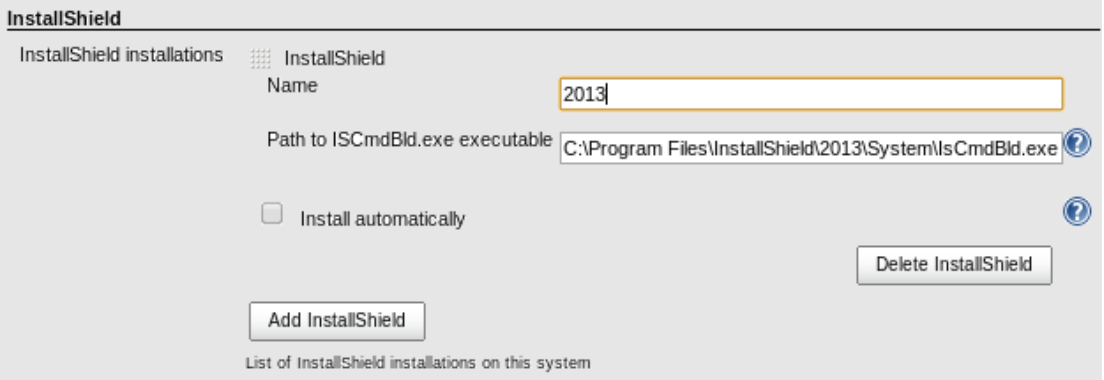
Obrázek 8.1: Užavatelské rozhraní modulu InstallShield (překlad projektu).

8.2.2 Implementace modulu

InstallShield modul do serveru Jenkins byl nazván installshield-plugin a jeho funkční metody byly implementovány v Pythonu. Grafické uživatelské rozhraní modulu se nachází na obrázcích 8.1 a 8.2. Hlavní funkcí modulu je překlad projektů aplikace InstallShield, k čemuž využívá nástroje `ISCmdBld.exe`. Modul dále obsahuje tyto funkce:

- *Definice cesty k projektovému souboru*
  Uživatel modulu může zadat cestu k souboru s InstallShield projektem. Cesta je relativní ke kořenovému adresáři úlohy, nad kterou je prováděn překlad.

[1] Jenkins Users: Google Groups – https://groups.google.com/forum/#!forum/jenkinsci-users
[2] InstallShield – http://www.installshield.com
[3] Flexera Software – http://www.flexerasoftware.com/company



Obrázek 8.2: Globální nastavení modulu InstallShield.

- *Automatické nalezení projektového souboru*
  V případě, že uživatel nezadá cestu k projektovému souboru, modul se tento soubor pokusí automaticky nalézt.
- *Možnost zadání dalších argumentů nástroje ISCmdBld.exe*
  Uživatel má možnost definovat pro každou úlohu další volitelné argumenty nástroje `ISCmdBld.exe`.
- *Indikace úspěchu překladu*
  Modul správně určí, zda při překladu došlo chybě či zda vše proběhlo v pořádku a tuto informaci korektně zprostředkuje uživateli.
- *Práce s více verzemi nástroje InstallShield na jednom uzlu (počítači)*
  Projektové binární soubory nástroje InstallShield nejsou zpětně kompatibilní. Z tohoto důvodu umožňuje modul definovat více nainstalovaných verzí nástroje v globálním nastavení a každé úloze přiřadit správnou verzi, která se má při překladu použít.
- *Expanze proměnných systému Jenkins*
  Uživatel může použít v nastavení u jednotlivých úloh (v nastavení cesty k projektovému souboru a v definici volitelných argumentů) proměnné systému Jenkins. Tyto proměnné jsou pak při samotném překladu převedeny na svoji stávající hodnotu.

Při vývoji modulu byly využity wrappery `ToolInstallerPW` a `BuilderPW` a jejich příslušné deskriptory `ToolDescriptorPW` a `BuildStepDescriptorPW`. Funkcionalita modulu tak mohla být snadno implementovaná v jazyce Python.

8.3 Testování

Nástroj PPSM byl otestován přímým použitím při tvorbě InstallShield modulu. Samotný InstallShield modul byl pak otestován na předem připravených InstallShield projektech s využitím nástroje InstallShield verze „2013 Express Edition Free Trial“.

Kapitola 9

Závěr

Tato kapitola rámcově uzavírá technickou zprávu. Popisuje způsob zveřejnění modulů, použité nástroje při tvorbě této práce a možnosti rozšíření projektu jenkins.py. Na konci kapitoly se nachází celkové shrnutí dosažených výsledků.

9.1 Zveřejnění modulů

Zveřejnění modulů probíhalo dle návodu popsaném v sekci 4.6. Mírná odlišnost od návodu spočívala v tom, že projekt jenkins.py neobsahuje pouze zdrojový kód modulu python-wrapper, ale i nástroje PWM a PPSM. Přesto však postup i v případě tohoto projektu fungoval bezproblémově. Moduly mohou být snadno staženy a nainstalovány do serveru Jenkins pomocí manažera modulů, jehož funkce je popsána v oddílu 3.2.1. Pro oba projekty jenkins.py[1] a installshield-plugin[2] byly vytvořeny repozitáře v rámci komunitního účtu `jenkinsci` na serveru GitHub. Dále byly pro moduly python-wrapper[3] a installshield-plugin[4] založeny Wiki stránky na serveru jenkins-ci.org.

Pro vývojáře, kteří mají zájem o vývoj modulů v jazyce Python, byly připraveny dva návody v anglickém jazyce.[5] První návod dokumentuje ruční užití třídy `PythonExecutor`, které mohou využít vývojáři pro začlenění Python skriptů do svých modulů napsaných v Javě. Druhý návod popisuje jak užívat wrappery a implementovat pomocí nich moduly v jazyce Python.

Projekt jenkins.py i modul installshield-plugin byly přijaty komunitou *bez výhrad*. Modul installshield-plugin dosáhl dle statistik za měsíc duben minimálně 10 uživatelských instalací. Modul python-wrapper pak 75 instalací. Vzhledem k tomu, že je modul installshield-plugin závislý na modulu python-wrapper, samostatných instalací modulu python-wrapper je jen 65. Nutno dodat, že statistiky zachycují instalace modulů pouze u neurčitého počtu instancí aplikace Jenkins, v nichž uživatelé ručně nastavili anonymní sběr dat pro tyto účely. [5]

9.2 Použité nástroje

Na vývoj nástrojů a modulů byl použit textový editor Geany, nástroj Maven, JDK verze 7 a Python verze 3.3. Jako správce verzí zdrojového kódu byl zvolen nástroj Git, kvůli nutnému

[1] jenkins.py – https://github.com/jenkinsci/jenkins.py
[2] installshield-plugin – https://github.com/jenkinsci/installshield-plugin
[3] Python Wrapper – https://wiki.jenkins-ci.org/display/JENKINS/Python+Wrapper+Plugin
[4] InstallShield Plugin – https://wiki.jenkins-ci.org/display/JENKINS/InstallShield+Plugin
[5] jenkins.py documentation – https://github.com/jenkinsci/jenkins.py/wiki

začlenění projektů do repozitářů na serveru GitHub.

Projekt jenkins.py byl vyvíjen a testován na operačních systémech MS Windows 7 64-bit a openSUSE 12.2 32-bit. Vzhledem k tomu, že virtuální stroj jazyka Java a interpret jazyka Python poskytují stejné běhové prostředí pro oba operační systémy, žádné rozdíly mezi funkcionalitou projektu na obou systémech nebyly zaznamenány. Modul InstallShield mohl být vyvíjen a testován pouze na operačním systému MS Windows, jelikož je aplikace InstallShield určena pouze pro tento systém.

Tato textová dokumentace byla vysázena pomocí nástroje LaTeX. Diagramy a schémata byly nakresleny v aplikacích Dia a Inkscape.

9.3 Možnosti rozšíření

Jednou z možností vylepšení projektu jenkins.py by mohl být mechanismus, který by umožňoval automatickou delegaci Stapler metod skriptům v Pythonu. Aktuálně musí vývojáři delegovat vykonávání těchto metod ručně. Uvažovat by šlo například o využití metody `doDynamic()` u jednotlivých wrapperů. Další možností by bylo implementovat serializaci objektu třídy `PythonExecutor` pro knihovnu XStream, která by tak obstarávala převod globálních proměnných v přidruženém Python skriptu. Vývojáři by pak nemuseli využívat Java objekty k uchovávání nastavení.

Zajímavým vylepšením by také bylo, kdyby se vytvářel pouze jeden statický objekt typu `PythonExecutor` pro každou třídu rozšíření, a ne pro každou její instanci. Takové řešení by mohlo urychlit běh modulů. V takovém případě by však bylo nutné předávat instanci rodičovského Java objektu jiným způsobem, než globální proměnnou `extension`, jak je tomu doposud. Vhodnou úpravou nástroje PWM by zřejmě bylo, pokud by u rozšiřitelných tříd a deskriptorů dohledával, jaká rozhraní implementují. Pokud by dané třídy neimplementovaly některou z metod rozhraní, mohl by program PWM nakládat s těmito metodami, jako by se jednalo o metody abstraktní a vyžadovat tedy jejich implementaci v terminálním modulu v rámci volání metody `checkAbstrMethods()`.

Současný návrh projektu jenkins.py má své výhody i omezení, které byly popsány v oddíle 5.7. Pokud by byl zájem o implementaci SDK pro Python na úrovni projektu jenkins.rb, jednalo by se konceptuálně o jiný projekt, než je současný jenkins.py. V takovém případě by ovšem musely být nejprve implementovány všechny potřebné knihovny, které jsou nutné pro načítání a běh samostatných Python modulů uvnitř systému Jenkins.

9.4 Shrnutí

V technické zprávě byly popsány základní vlastnosti serveru pro průběžnou integraci Jenkins CI a jeho systému zásuvných modulů. Dále jsou podrobně zdokumentovány možnosti vývoje zásuvných modulů v jazycích Java a Ruby. Z této dokumentace pak vychází návrh nástrojů pro podporu vývoje modulů v jazyce Python. Navržené řešení bylo následně úspěšně implementováno a schváleno komunitou. Příslušné nástroje a moduly byly zveřejněny a jsou tak k dispozici vývojářům, kteří mají zájem implementovat moduly v jazyce Python. V technické zprávě pak byla tato implementace zdokumentována.

Literatura

[1] Flexera Software LLC: InstallShield 2013 Help Library [online]. Dostupné z: http://helpnet.installshield.com/installshield20helplib/installshield20helplib.htm, [cit. 11. května 2014].

[2] GitHub Wiki: Jenkins.rb Home Page [online]. Dostupné z: https://github.com/jenkinsci/jenkins.rb/wiki, [cit. 3. ledna 2014].

[3] Jenkins CI: Jenkins Documentation [online]. Dostupné z: http://javadoc.jenkins-ci.org, [cit. 3. ledna 2014].

[4] Jenkins CI: Maven Jenkins Plugin [online]. Dostupné z: http://jenkins-ci.org/maven-hpi-plugin, [cit. 3. ledna 2014].

[5] Jenkins Wiki: Plugin Installation Statistics [online]. Dostupné z: https://wiki.jenkins-ci.org/display/JENKINS/Plugin+Installation+Statistics, [cit. 11. května 2014].

[6] Jenkins Wiki: Extend Jenkins [online]. Dostupné z: https://wiki.jenkins-ci.org/display/JENKINS/Extend+Jenkins, [cit. 3. ledna 2014].

[7] JRuby: JRuby Documentation [online]. Dostupné z: http://jruby.org/documentation, [cit. 3. ledna 2014].

[8] Jython: Jython Documentation [online]. Dostupné z: http://www.jython.org/docs, [cit. 3. ledna 2014].

[9] KUHN, T.; THOMANN, O.: Eclipse Corner Article: Abstract Syntax Tree [online]. Dostupné z: http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html, 2006 [cit. 11. května 2014].

[10] LOWELL, C.: JRuby Branch merged! [online]. Dostupné z: http://jenkins-ci.org/content/jruby-branch-merged, 2011 [cit. 3. ledna 2014].

[11] LOWELL, C.: What it takes to bring Ruby to Jenkins [online]. Dostupné z: http://blog.thefrontside.net/2011/05/12/what-it-take-to-bring-ruby-to-jenkins, 2011 [cit. 3. ledna 2014].

[12] PRAKASH, W.: Hudson Execution and Scheduling Architecture [online]. Dostupné z: http://hudson-ci.org/docs/HudsonArch-Execution.pdf, 2010 [cit. 3. ledna 2014].

[13] PRAKASH, W.: Hudson Plugin Architecture [online]. Dostupné z: http://hudson-ci.org/docs/HudsonArch-Plugin.pdf, 2010 [cit. 3. ledna 2014].

[14] PRAKASH, W.: Hudson View Architecture [online]. Dostupné z: http://hudson-ci.org/docs/HudsonArch-View.pdf, 2010 [cit. 3. ledna 2014].

[15] PRAKASH, W.: Hudson Web Architecture [online]. Dostupné z: http://hudson-ci.org/docs/HudsonArch-Web.pdf, 2010 [cit. 3. ledna 2014].

[16] SMART, J. F.: *Jenkins: The Definitive Guide*. O'Reilly Media, Inc., 2011, ISBN 978-1-4493-0535-2.

[17] The Apache Software Foundation: Apache Maven Project: Documentation [online]. Dostupné z: http://maven.apache.org/guides/index.html, [cit. 3. ledna 2014].

[18] The Eclipse Foundation: Eclipse documentation [online]. Dostupné z: http://www.eclipse.org/documentation, [cit. 11. května 2014].