VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# NÁVRH A IMPLEMENTACE TESTOVACÍHO SYSTÉMU NA ARCHITEKTUŘE GRID

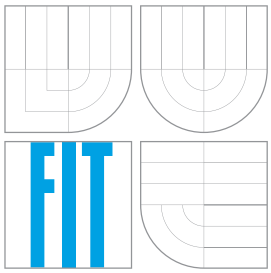
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. FILIP HUBÍK

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# NÁVRH A IMPLEMENTACE TESTOVACÍHO SYSTÉMU NA ARCHITEKTUŘE GRID

DESIGN AND IMPLEMENT GRID TESTING SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. FILIP HUBÍK
AUTHOR

VEDOUCÍ PRÁCE — Ing. RADEK KOČÍ, Ph.D.
SUPERVISOR

VEDOUCÍ PRÁCE Z RED HAT — JIŘÍ PECHANEC
SUPERVISOR

BRNO 2013

*Zadání diplomové práce/11021/2012/xhubik01*

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav inteligentních systémů Akademický rok 2012/2013

# Zadání diplomové práce

Řešitel: **Hubík Filip, Bc.**
Obor: Počítačové a vestavěné systémy
Téma: **Návrh a implementace testovacího systému na architektuře GRID**
**Design and Implement Grid Testing System**
Kategorie: Softwarové inženýrství

Pokyny:

1. Prostudujte dostupné implementace architektury GRID v Javě.
2. Seznamte se s metodou průběžné integrace aplikací, s nástrojem pro průběžnou integraci Jenkins a testovacími frameworky TestNG a JUnit.
3. Navrhněte architekturu softwaru, který umožňuje distribuci testovacích úloh na uzly zapojené v architektuře GRID a jejich paralelní běh na těchto uzlech. Při návrhu uvažujte následující možné postupy, případně jejich kombinaci: rozšíření jádra nástroje Jenkins, plug-in do nástroje Jenkins, nebo samostatná aplikace komunikující s nástrojem Jenkins přes definované rozhraní.
4. Navrhněte algoritmy pro rozdělení testovacího souboru a jeho distribuci na uzly GRID podle závislostí jednotlivých testů.
5. Navrženou architekturu a algoritmy implementujte.
6. Diskutujte dosažené výsledky a možná rozšíření.

Literatura:

- J. F. Smart. Jenkins: The Definitive Guide. O'Reilly Media, 2011
- Extend Jenkins. Dostupné na https://wiki.jenkins-ci.org/display/JENKINS/Extend+Jenkins, říjen 2012

Při obhajobě semestrální části diplomového projektu je požadováno:

- První tři body zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kočí Radek, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 17. září 2012
Datum odevzdání: 22. května 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
*vedoucí ústavu*

# Abstrakt

Tento diplomový projekt se zabývá problematikou paralelizace sestavování a testovaní aplikací tvořených v jazyce Java. Navrhuje software, který s využitím metod postupné integrace, paralelizace a distribuce výpočetně náročných úloh na architekturu grid napomáhá ke zrychlení vývoje softwarových produktů a automatizaci částí jejich vývojového cyklu.



# Abstract

This project addresses parallelization of building and testing projects written i Java programming language. It proposes software that uses methods of continual integration, parallelization and distribution of computationally intensive tasks to grid architecture. Suggested software helps to accelerate the development of software product and automation of its parts.



# Klíčová slova

grid Jenkins Hudson Java Red Hat open source MapReduce Maven TestNG Junit Infinispan GridGain Apache Hadoop Classloader JVM javac paralelizace testování compilace distribuce SwitchYard ModeShape



# Keywords

grid Jenkins Hudson Java Red Hat open source MapReduce Maven TestNG Junit Infinispan GridGain Apache Hadoop Classloader JVM javac parallelization testing compiling distribution SwitchYard ModeShape



# Citace

Filip Hubík: Návrh a implementace testovacího systému na architektuře GRID, diplomová práce, Brno, FIT VUT v Brně, 2013

# Návrh a implementace testovacího systému na architektuře GRID

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pánů Ing. Radka Kočího, Ph.D., zástupce FIT VUT Brno a Jiřího Pechance, zástupce firmy Red Hat

.....
Filip Hubík
21. května 2013

## Poděkování

Oběma vedoucím děkuji za konzultace a pomoc při vedení a vypracování práce. Díky také společnosti Red Hat za hardware zapůjčený pro testování.

# Obsah

# Seznam obrázků

# Kapitola 1

# Úvod

V dnešní době je většina softwarových produktů, určených pro použití v praxi, psána užitím univerzálních programovacích jazyků (Java, C++, C#). Univerzálnost takových jazyků je velice výhodná, protože umožňuje velmi rychlou tvorbu produktu, tj. není potřeba implementovat základní funkcionalitu, která je již dostupná prostřednictvím knihoven daného jazyka. Při vývoji softwaru se užívají metodiky, které jsou složeny z různě dlouhých vývojových částí od návrhu, přes implementaci, až po odevzdání produktu zákazníkovi.

Softwarové projekty se v praxi vyznačují narůstajícím časem průchodu životním cyklem celého projektu s jeho velikostí. Tato charakteristika rozhodně není žádoucí. Proto se vyvinuly postupy, jak takovou tendenci optimalizovat. Jedním z nich může být paralelizace sestavování („build“ procesu), avšak tato metoda může být značně omezena možnostmi daného programovacího jazyka. Například sestavování produktu v programovacím jazyce C lze paralelizovat díky definicím v hlavičkových souborech, jazyk Java však s tímto konceptem ve svých původních verzích nativně nepočítal. Nabízí se otázka, jak tedy zrychlit průchod životním cyklem softwaru, či kterou další jeho část lze zefektivnit. Jednou z možností je zaměřit se na testování. V praxi je doba sestavování velkých softwarových produktů v jazyce Java zanedbatelná oproti době testování. Optimalizací, správných rozdělením a výslednou paralelizací testovacího procesu lze docílit značného urychlení tvorby SW produktů.

Softwarové firmy, vyvíjející produkty v jazyce Java, provozují nástroje pro průběžnou integraci, které se starají zejména o verzování, sestavování a testování vyvíjeného softwaru. Tyto nástroje ale mají své nedostatky, např. provádění některých potencionálně paralelizovatelných částí životního cyklu sekvenčně. Je třeba se také zamyslet, jaká architektura je pro paralelizované úlohy vhodná. U rostoucích modulárních projektů přestává stačit výpočetní výkon jednotlivých počítačů, ať už jednojádrových či vícejádrových. Nabízí se varianty distribuovaného počítání, tj. Cluster a Grid computing nebo užití superpočítačů. Superpočítače s velkým počtem výpočetních jednotek nejsou pro firmy cenově výhodné, naopak počítačové clustery jsou zase špatně škálovatelné. V dnešní internetové době, kdy je dosahováno stále větších výkonností počítačů či rychlostí připojení mezi různými místy světa, se však stává pro firmy nejvýhodnější alternativou Grid computing, který se jeví hlavně jako vhodný přístup pro vývoj velkých modulárních softwarových produktů.

Vezmeme-li v potaz potencionální možnost paralelizace testovacích úloh, kdy si představíme testovací soubor celého projektu jako množinu menších testovacích modulů, můžeme spojením integračního software a gridové architektury dosáhnout paralelního otestování celku a tím zefektivnit jednu z nejproblematičtějších částí životního cyklu vyvíjeného softwaru.

## 1.1 Cíle práce

Cílem práce je implementace plugin modulu do programátory oblíbeného nástroje pro průběžnou integraci Jenkins (větev nástroje Hudson). Software bude podporovat framework Maven, Junit a TestNG a bude přesunovat výpočetně náročné operace sestavování a testování softwarových produktů na architekturu grid. Budou diskutovány, navrhovány a vybrány nejvhodnější postupy zrychlení či paralelizace testování (tedy i sestavovaní) robustnějších multimodulárních projektů v jazyce Java. Práce je vypracovávána ve spolupráci s QA oddělením firmy Red Hat a se snahou o co největší využití vhodných existujících open source technologií pro usnadnění řešení dílčích problémů.

## 1.2 Obsah práce

Nejprve bude diskutována problematika sestavování a testování projektů v jazyce Java (kap. 2), budou navrhnuty také postupy paralelizace sestavovacího a testovacího procesu. V této kapitole bude také rozebrán současně nejčastěji používaný framework ke správě životního cyklu Maven. Budou také podrobně srovnány frameworky Junit a TestNG s důrazem na rozdíly mezi nimi. Dále bude rozebrán nejpoužívanější gridový software (kap. 3), zejména využívající algoritmu MapReduce. Budou srovnány výhody a nevýhody nejrozšířenějších implementací, které by mohly sloužit jako část navrhovaného software. Následovat bude rozbor systému Jenkins (kap. 4). V kapitolách 5 a 6 budou diskutovány jednotlivé aspekty návrhu a implementace. V závěru práce bude demonstrováno použití implementace na reálných příkladech (kap. 7).

# Kapitola 2

# Prostředky jazyka Java

V následující kapitole stručně popsán jazyk Java, dále budou diskutovány zejména detaily sestavovacího, spouštěcího a testovacího procesu a budou uvedeny možnosti paralelizace těchto procesů. Nejedná o podrobný popis jazyka, ale o popis nejdůležitějších částí, které budou důležité pro dosažení cílů této práce. Místo bude věnováno i frameworkům Maven, TestNG a Junit a jejich souvislostem s možnostmi paralelizace. Cílem je ukázat teoretické řešení paralelní kompilace a testování Maven modulů.

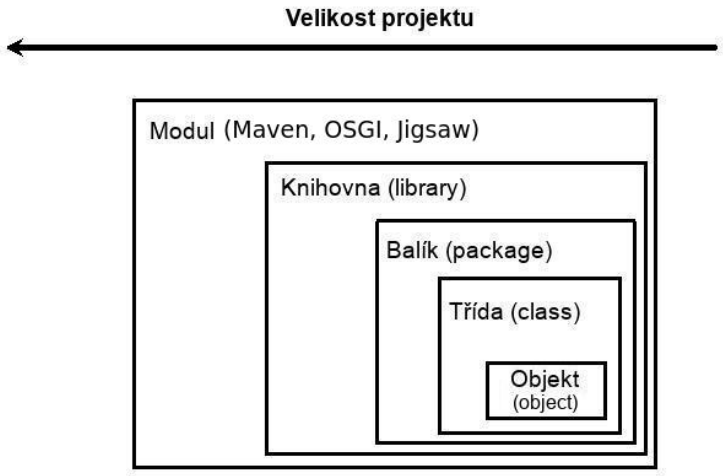
## 2.1 Základní charakteristika a struktura jazyka

Jazyk Java je částečně interpretovaný, částečně kompilovaný, objektově orientovaný (OO), vysokoúrovňový, univerzální programovací jazyk vyvinutý firmou Sun, v současné době spravován firmou Oracle. Jednou z jeho hlavních výhod je velmi dobrá přenositelnost kódu, tedy jednotlivý program v jazyce Java je možno spustit na libovolné platformě s funkčním prostředím Java (tzv. Java runtime environment), ať už se jedná o operační systém na vestavěném zařízení či na osobním počítači.

Jelikož se jedná o OO jazyk, je základním stavebním prvkem jazyka objekt, který vzniká instanciací z jeho mateřské třídy. Třídy jsou organizovány do větších celků, tzv. balíků (package), které zaručují oddělený jmenný prostor. Jmenný prostor balíku je tvořen hierarchicky a je zapisován v tečkové notaci, např. „org.java.projekt.trida“. Třídy jsou organizovány do zmíněných balíků, více balíků je organizováno do knihoven, obojí často zapouzdřeno do archivů (JAR, WAR, EAR,atd.). Takové archivy mohou obsahovat také doprovodná metadata v souboru MANIFEST.MF, která obsahují informace o vstupní metodě, o verzi archivu či autorovi.

Entitou zapouzdřující knihovny či balíky mohou být moduly. Větší projekty mohou být organizovány do modulů pomocí speciálních frameworků (Maven, OSGi, Jigsaw), kde modul zpravidla přidává další metadata k JAR archivu, jako jsou verzovací informace, repozitáře modulů, závislosti mezi moduly apod. Moduly tedy mohou prezentovat samy sebe a jsou připraveny na okamžitou a snadnou distribuci. Doprovodná metadata mohou být prezentována i prostřednictvím textového souboru či souboru typu XML.

Struktura logické organizace tříd v závislosti na velikosti projektu je znázorněna na obrázku 2.1. Zatím co menší a střední projekty jsou převážně v poslední fázi vývoje distribuovány jako archivy reprezentující knihovny či balíky, u velkých projektů je velmi výhodné, až nutné, využít přídavných vlastností modulů jako zmíněné verzování či spolupráce s integračními nástroji.




Obrázek 2.1: Logická organizace projektu v jazyce Java

## 2.2 Nejdůležitější součásti architektury jazyka

Nejdůležitější součásti jazyka Java jsou kompilátor a prostředí JRE (Java runtime environment), zapouzdřující Java virtual machine (JVM, obr. 2.2). JVM obsahuje důležité části jako „just-in-time“ kompilátor (JIT), interpreter, garbage collector či další důležitou komponentu, tzv. classloader subsystém. Kompilátor převádí program v textové podobě do nikoliv nativního, ale do speciálního kódu, který je dále přiveden na vstup JVM. V JVM jsou spuštěny optimalizace takového kódu a „just-in-time“ překlad do kódu nativního. JVM je načtena kompletně v paměti a stará se jak o zpracování a spuštění kódu, tak o správu paměti. K účelu správy paměti slouží garbage collector, který alokuje a uvolňuje objekty závisle na toku programu. Nejdůležitější komponenty budou popsány podrobněji v následujících sekcích.

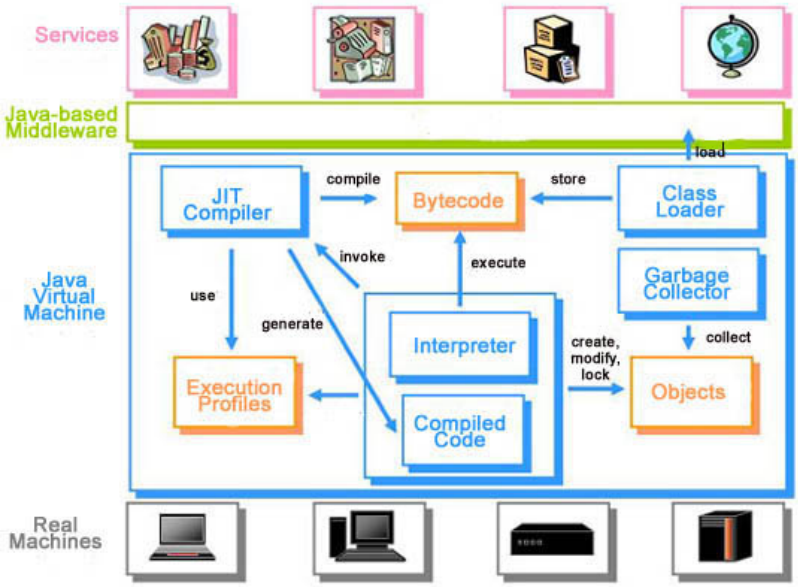
### 2.2.1 Statická kompilace

Vstupem základního procesu statické kompilace v jazyce Java jsou zdrojové soubory s příponou „.java“ reprezentující třídy, vytvářené programátorem. Zpravidla každá třída je umístěna v samostatném souboru. Tyto třídy jsou následně překladačem transformovány do souborů s příponou „.class“, které obsahují tzv. bytekód, tj. kód, který je vykonáván JVM. Ovšem existují i komplikovanější postupy, jak překládat zdrojový kód (např. kompilace za běhu programu, či načtení zkompilovaných zdrojových souborů z nelokálního úložiště). Soubory s příponou „.class“ je dále také možné pro snazší distribuci zabalit do archivu a přidat doplňující informace.

Syntaxe kompilátoru *javac* pro kompilaci obecného jednoduchého kódu je uvedena v úryvku 2.1 Za zmínku stojí velice důležitý parametr „-cp“ (tj. proměnná prostředí CLASSPATH), který slouží pro připojení existujících knihoven či dalších zdrojových souborů k procesu kompilace či parametr „-d“, který slouží ke změně umístění zkompilovaných souborů.

Úryvek kódu 2.1: Kompilátor jazyka Java

```
javac <options: -cp, -d ...> <source files>
```




Obrázek 2.2: Architektura Java [5]

Při kompilaci je ve vstupním souboru či souborech také provedena kontrola referencí. Je-li nalezena reference na objekt či metodu, která není v aktuálním souboru, je hledání provedeno v dalších vstupních zdrojových souborech (parametr „source files“). Pokud ani v nich není cíl reference nalezen, proběhne hledání v domovském balíku odkazující třídy, tedy pokud je správně nastaven parametr CLASSPATH. Parametr CLASSPATH často odkazuje na více umístění zdrojových souborů, které jsou také zařazeny do vyhledávacího a (v případě potřeby) kompilačního procesu.

### 2.2.2 JVM

Spuštění programu, nebo lépe řečeno interpretace bytekódu, je proces výhradně v režii JVM (obr. 2.2). Bytekód je načten z „class“ souborů pomocí tzv. **„classloader subsytému“**. Tento subsytém je velice důležitý, jedná se o klíčový prostředek k využití modularity za běhu programu. JVM načte do paměti prostřednictvím classloaderu bytekód třídy až v případě, kdy je požadována její přítomnost (tzv. „on-demand“ přístup). Bytekód je dále verifikován, profilován a optimalizován. Interpreter jej vykonává dle toku programu a spolupracuje s JIT kompilátorem, který jeho příkazy překládá do nativního kódu. Kód přeložený do instrukcí cílové platformy je poté spuštěn. V průběhu interpretace programu je také komunikováno s garbage-collectorem, který alokuje místo pro nové objekty a uvolňuje místo po objektech, na které už v programu neexistuje ukazatel či je objekt vyhodnocen jako dealokovatelný.

Pro načtení zkompilovaného kódu do JVM, jeho interpretaci a překlad do kódu nativního se v jazyce Java využívá program *java*. Obdobně jako u kompilátoru je třeba často uvést parametr „-cp“, specifikující v tomto případě cestu ke zkompilovaným knihovnám, souborům či archivům „jar“. Nesmí chybět ani přesný název a umístění vstupní metody programu. Přesná syntaxe je následující:

Úryvek kódu 2.2: Spuštění programu v jazyce Java

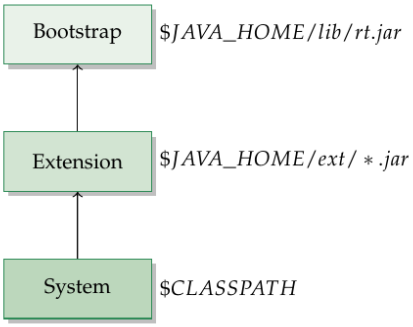
```
java [-options] class [args...]
```

```
java [-options] -jar jarfile [args...]
```

### Classloader subsystém

Tento velice důležitý podsystém JVM je zodpovědný za načtení potřebných souborů formou bytekódu do paměti. V omezené míře je součástí JRE od verze 1.0, značných úprav se dočkal v Java 1.2. Struktura systému je hierarchická ve formátu rodič-potomek a sestává z různých druhů systémových classloader jednotek (dále jen CL, obr. 2.3). Do hierarchie však mohou být zavedeny i CL uživatelské. Systémové CL slouží k načtení důvěrných systémových souborů z lokálního disku nebo k načtení již zkompilovaných a připravených tříd.

První z nich, Bootstrap neboli Primordial CL, načte z disku zkompilované třídy Java JRE, potřebné pro zavedení základního prostředí, zpravidla uložené v souboru rt.jar (Oracle Java 6). Dále je třeba načíst doplňkové knihovny jazyka přes Extension CL a v neposlední řadě třídy určené parametrem „-cp“ nebo proměnnou prostředí CLASSPATH pomocí System CL.




Obrázek 2.3: Struktura CL v Java 6 [15]

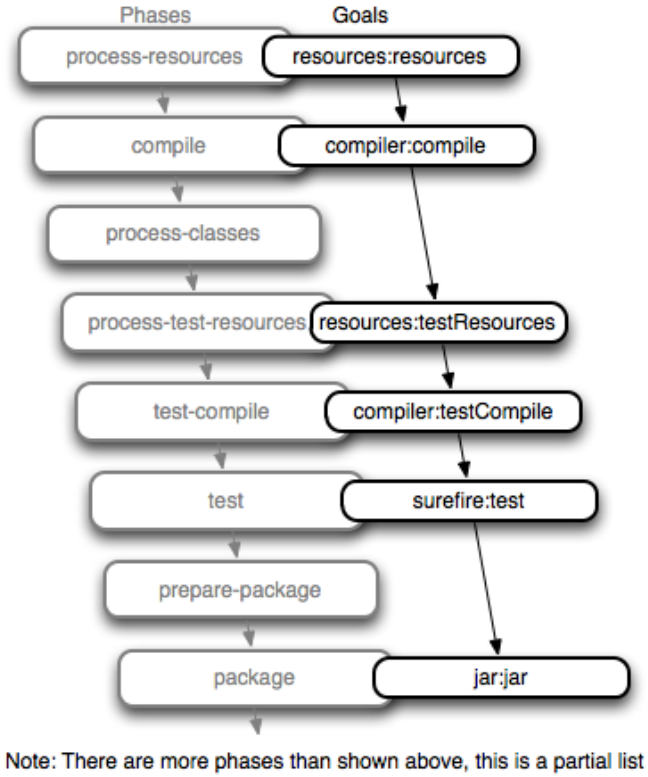
Prostřednictvím uživatelských CL je možno provádět dynamickou kompilaci či načítání zkompilovaných tříd za běhu aplikace a tím rozšiřovat její funkcionalitu, což otevírá rozsáhlé možnosti, ale také má svá bezpečnostní rizika.

Uživatelské CL je navíc možno vytvářet a rušit za běhu programu, čímž připomínají architekturu, jejíž jádro je zavedeno v paměti trvale a přídavná funkcionalita je zaváděna pomocí plugin modulů (analogie s jádrem systému Linux a jeho moduly). Co se týče lokality dat v souvislosti s načítáním bytekódu tříd, je možno prostřednictvím uživatelských CL načítat bytekód z nejen různých umístění na disku, ale například pomocí síťových protokolů ze vzdálených umístění (na takovém mechanismu jsou postaveny Java applety ve webových prohlížečích). Je třeba zmínit, že CL umožňují i oddělení jmenného prostoru jednotlivých tříd, čili jedna třída může existovat v paměti několikrát, což by standardně nemohla, ale pouze v jiných jmenných prostorech.

## 2.3 Framework Maven a modularita

U rozsáhlých projektů je tedy od určitého bodu nutno rozdělit projekt jako celek do modulů, které lze do určité míry považovat za samostatné entity. Apache Maven je framework sloužící pro zavedení konceptu modularity, automatizaci sestavování, testování a distribuci

softwarových produktů v jazyce Java, avšak podporuje i jazyky další (Ruby, Scala). V současné době nahrazuje stále méně používaný sestavovací nástroj Ant. Velice používaný je i jako nástroj pro správu závislostí v projektu. Pracuje na základě znovupoužitelnosti zaužívaných metod pro práci se softwarem v jeho vývojové fázi. Jsou definovány tři základní životní cykly: default, site a clean. První cyklus se stará o generování kódu, druhý o prezentaci výsledků nebo dokumentaci. Cyklus clean se stará o čištění adresáře od generovaných souborů a uvedení projektu do původního stavu. V rámci nejčastěji používaného životního cyklu - default - jsou definovány některé nejdůležitější fáze vývojového procesu (lifecycle phases) na obrázku 2.4.




Obrázek 2.4: Fáze a cíle v Maven [1]

Každá fáze může mít asociováno nula a více cílů (goals), každý cíl může být spojen s nula a více fázemi. Cíl je vstupní bod pro použití určité funkcionality, implementované v Maven v drtivé většině formou zásuvných modulů (dále jen plugin modulů). Je-li zavolána některá z fází, jsou spuštěny všechny fáze jí předcházející včetně, což znamená, že jsou také vykonány příslušné cíle. Cíle, které nejsou přiřazeny žádné z fází a jsou tak vytrženy z posloupnosti exekuce po sobě jdoucích fází a je možno je pouze volat externě přes příkazovou řádku. Naopak fáze, které nemají asociován žádný cíl, není možno spustit.

Příkazová řádka Maven se skládá z volitelných parametrů, možných cílů a/nebo fází životního cyklu.

mvn [parametry] [jméno_plugin_modulu]:[cíl] [cíle] [fáze]

[1] Obrázek převzat z http://books.sonatype.com/mvnex-book/reference/simple-project-sect-simple-core.html

Je možné cíle i fáze a libovolně kombinovat. Maven určí chování zjištěním příslušnosti fází k cyklům a cílů k fázím a příkaz provede. Lze také volat přímo plugin modul identifikován jeho jménem, který vykonává požadovanou funkci v rámci Maven. Plugin moduly lze do prostředí importovat pouhou instalací do lokálního repozitáře, poté je lze vyvolat uvedeným způsobem.

Lokální repozitář ukládá nainstalované plugin moduly, které nejsou součástí instalace Maven, dále také produkty fáze install cyklu default a celkově veškeré závislosti potřebné pro průchody životními cykly. Nejčastěji se jedná o balíky zkompilovaných zdrojových souborů ve formátu JAR, mohou to být ale také archivy EAR, WAR či nekomprimovaný konfigurační soubor ve formátu POM a další. Repozitář je zpravidla dostupný v různých umístěních v závislosti na operačním systému. V Linux a MAC OS jej najdeme ve složce *∼/.m2*, v MS Windows pod *C:\Documents and Settings\<username>\.m2*. Pokud není závislost nalezena v lokálně, je stažena online z Maven repozitáře. Pokud není nalezena ani tam, hledá se v implicitně nebo explicitně (v konfiguračním souboru Maven settings.xml) specifikovaných repozitářích. V případě opětovného nenalezení dojde k chybě při běhu některé fáze nebo cíle.

### 2.3.1 Struktura modulu a závislosti

Základní jednotka práce v Maven je „project object model“, reprezentovaná implicitně souborem pom.xml, lze však definovat i jiný zdrojový soubor (dále jen POM). Maven zapouzdřuje Java projekty do tzv. modulů, každý modul má asociován právě jeden POM soubor, kde jsou definována v podstatě metadata o celém projektu či modulu. Minimální struktura souboru je uvedena v úryvku 2.3.

Úryvek kódu 2.3: Minimální soubor POM

```
<project>
<modelVersion>1.0.0</modelVersion>
<groupId>org.myproject.project</groupId>
<artifactId>myproduct</artifactId>
<version>1</version>
</project>
```

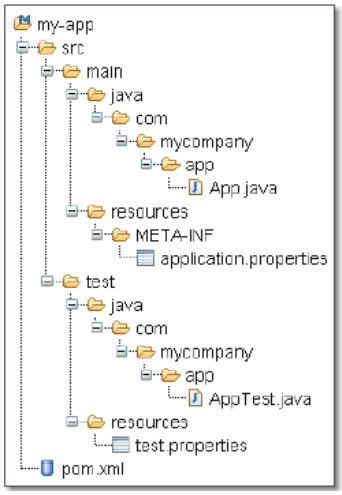
POM může obsahovat různé druhy informací, údaje o vývojáři nebo o vyvíjeném software, může definovat různé relace mezi moduly, specifikovat vzdálené repozitáře, použité plugin moduly, „resource“ objekty apod.

Orientované relace mezi Maven moduly jsou znázorněny na obrázku 2.6 formou šipek od modulu k modulu. Sémantika takové relace, například na obrázku označené písmenem „r“, se dá vyložit slovy takto: modul B závisí na modulu A a zároveň modul A je potřeba mít zkompilovaný, aby mohl být kompilován modul B. Souhrnně tyto relace nazýváme závislosti mezi moduly určitého typu. Tyto závislosti (ve formě orientovaných hran) mohou společně s moduly jako vrcholy tvořit orientovaný acyklický graf (může být souvislý i nesouvislý), každý typ závislostí má však vlastní graf a závislosti se mohou překrývat. Nejdůležitější je však graf závislostí kompilace („compile scope graph“) a graf závislostí testování („test scope graph“). Maven (verze 3) definuje 6 druhů závislostí:

- compile - závislost je potřebná pro kompilaci aktuálního modulu, v případě potřeby si Maven před kompilací zajistí dostupnost potřebných závislostí z repozitáře

- provided - v tomto případě je závislost přítomna pouze při kompilaci, ale není zahrnuta do výsledného sestavení - očekává se, že bude nějakým způsobem poskytnuta za běhu aplikace
- runtime - závislost není potřeba ke kompilaci, ale k běhu aplikace
- test - závislost potřebná pro otestování modulu
- system - podobné jako provided, jen je potřeba explicitně specifikovat lokální cestu k archivu zdrojových souborů, taková závislost není vyhledávána v repozitářích
- import - speciální závislost, která se nepodílí na tvorbě tranzitivních relací

Maven také přesně definuje hierarchickou strukturu modulu. Adresářová struktura pro ukázkovou aplikaci *App.java* je zobrazena na obrázku 2.5, může však být explicitně změněna.



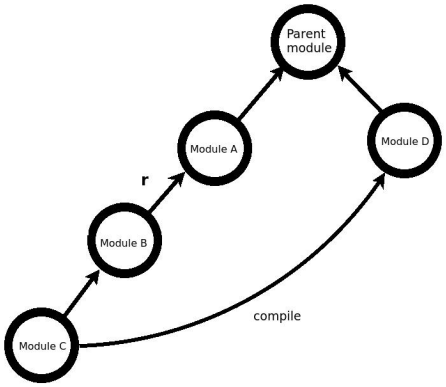

Obrázek 2.5: Ukázková adresářová struktura Maven modulu [22]

Adresář „src“ je určen pro umístění zdrojových souborů, adresář „test“ pro testovací třídy. V každém z nich je umístěna konvenční struktura balíků v jazyce Java. I moduly mohou být organizovány do hierarchických struktur a mohou být definovány závislosti mezi nimi. Takovým projektům se říká *„multimodulární projekty“* a práce s nimi je jedním z impulzů vzniku tohoto textu a doprovodné implementace.

### 2.3.2 Multimodulární projekty

Multimodulární projekty jsou nejčastějším typem logické organizace rozsáhlých aplikací v jazyce Java vyvíjených pomocí Maven. Díky tomu, že sestávají z menších modulů, je možné lépe organizovat či rozdělovat práci v týmu. Maven při vykonávání cílů nad takovými projekty využívá zásuvný modul „Maven Reactor Plugin“, který vytváří graf závislostí mezi moduly, vypočítá sekvenční pořadí jejich kompilace či testování a provede zadaný cíl (od Maven verze 3 je možno experimentálně provádět paralelně). Dále může být zkompilován, otestován a integrován celý multimodulární projekt nebo jeho části, může být pokračováno v zadaném procesu v případě chyby od posledního úspěšného bodu.

Důležitým pojmem v oblasti tohoto typu projektů je tzv. **kořenový modul** („root module“), který zapouzdřuje celý multimodulární projekt v Maven. Ve své podstatě většinou neobsahuje data na úrovni zdrojového kódu, slouží jako jakýsi kontejner a zdroj metadat o modulech obsažených v projektu. Často sestává jen z jednoho POM souboru, který se v adresářové struktuře nachází na vstupní úrovni zdrojových souborů projektu. Tento POM soubor je možno archivovat či lokálně nainstalovat, může se také na jeho umístění odkazovat množství POM souborů modulů v projektu obsažených.




Obrázek 2.6: Ukázka grafu modulů

Projekty tohoto typu mohou obsahovat velké množství závislostí. Existují mechanismy usnadňující distribuci závislostí, parametrů a vlastností modulu grafem modulů. Jsou to dědičnost a agregace. Mechanismus dědičnosti definuje nutné zadání rodiče uvnitř POM submodulu potomka, tento potomek pak dědí veškeré závislosti rodiče a může přidávat své vlastní. Tento mechanismus je vhodný pro moduly se stejnou či podobnou konfigurací napříč hierarchií. Agregace naopak definuje v rodičovském POM jména všech modulů potomků, moduly pak zmiňovat rodiče nemusí. Pokud je spuštěna např. kompilace nad rodičem, je poté spuštěna i nad všemi potomky. Oba přístupy lze kombinovat.

### 2.3.3 Předpoklady pro modularitu

Cílem zavedení modularity je zajistit možnosti spouštění cíle nad celým projektem nebo jeho součástmi flexibilně, proto je také třeba mít přesně definovány vzájemné závislosti modulů v projektu.

Pokud je třeba spustit například kompilaci některého dílčího modulu z projektu, vloží Maven do proměnné CLASSPATH prostředí Java již přeložené zdrojové soubory závislostí, které definuje konfigurační soubor POM aktuálního modulu. Toto jsou nejen závislosti na modulech projektu, který aktuální (sub)modul zastřešuje, ale také na archivech či modulech třetích stran. U závislostí třetích stran se očekává jejich přítomnost v lokálním repozitáři, není-li tomu tak, provede se jejich stažení a instalace.

Kompilace projektu, jehož struktura závislostí modulů obsahuje cykly skončí s chybou, protože Maven detekuje cyklické závislosti již před startem procesu. Toto ovšem platí pro spuštění dané operace nad kořenovým konfiguračním souborem celého projektu pomocí Maven Reactor Plugin, nikoliv na úrovni modulu. Pokud je Maven spuštěn na úrovni modulu, nedetekuje cyklycké závislosti v rámci celého projektu a prostřednictvím Reactor je spuštěna kompilace podstromu s kořenem na úrovni aktuálního modulu. Cyklické závislosti jsou zdrojem chyb a je třeba se jich vyvarovat, u rozsáhlých projektů však mohou být pro-

blémem, který je třeba řešit některými nástroji pro visualizaci stromové struktury modulů, či použít existující zásuvné moduly Maven (např. Maven Dependency Plugin).

Projekty musí samozřejmě mít strukturu vhodnou pro paralelní zpracování, tzn. projekt nesmí být velmi malého rozsahu (dostatečně rozsáhlý aby paralelizace měla smysl) a musí obsahovat více než 1 modul. Nesmí také obsahovat reflexivní relace závislostí.

### 2.3.4 Definice pojmů

Jak již bylo popsáno v sekci 2.3.1, Maven může disponovat několika druhy závislostí. Pro hlubší poznatky a práci s grafy závislostí modulů určitého typu v Maven je třeba zavést jednotku, která bude vzájemně odlišovat jednotlivé moduly nebo popisovat některé jejich významné vlastnosti.

Nabízí se pojem nejkratší cesty, zavedené v teorii grafů jako minima z délek cest mezi vrcholy grafu. Tato veličina ale nebude stačit, protože orientace závislostí Maven modulů směřuje ke kořenovému modulu (obr. 2.7) a taková informace o vrcholu (modulu) v grafu by pak byla téměř nicneříkající. Je tedy třeba naopak zjistit maximální vzdálenost mezi libovolným modulem a kořenovým modulem projektu jako vrcholy grafu. Tato informace pak může pomoct při určování počtu potřebných modulů, které musejí být připraveny jako prerekvizita pro spuštění Maven nad takovým nekořenovým modulem.

Mějme orientovaný nesouvislý konečný neprázdný graf bez reflexivních relací závislostí a smyček G, G = $<U, H>$, $H \subseteq \{(x,y)|(x,y) \in U \times U, x \neq y\}$ kde U je množina Maven modulů v projektu a H množina orientovaných hran, které reprezentují relaci závislosti jednoho typu mezi nimi. Tomuto speciálnímu grafu budeme říkat **Maven graf** určitého typu podle příslušné relace závislosti, zkráceně Maven graf. Takový graf bývá také někdy označován jako „Directed acyclic graph“ (DAG). Ukázku grafu takového typu lze vidět na obrázku 2.7a. Takto definovaný graf samozřejmě může být i stromem, v praxi je však jen ojediněle.

Dále každý souvislý podgraf takového grafu, neboli komponenta, má klíčový prvek, tzv. **kořenový uzel**. Kořenový uzel grafu reprezentuje kořenový modul projektu (kap. 2.3.2). Pokud ohodnotíme všechny uzly Maven grafu jejich výstupním stupněm (výstupní valencí), tj. funkcí $deg_{out}(x)$ $x \in U$, kde

$$deg_{out}(x) = num \text{ , kde } num \text{ je počet hran vycházejících z uzlu } x \tag{2.1}$$

můžeme pro kořenový uzel $root \in U$ psát

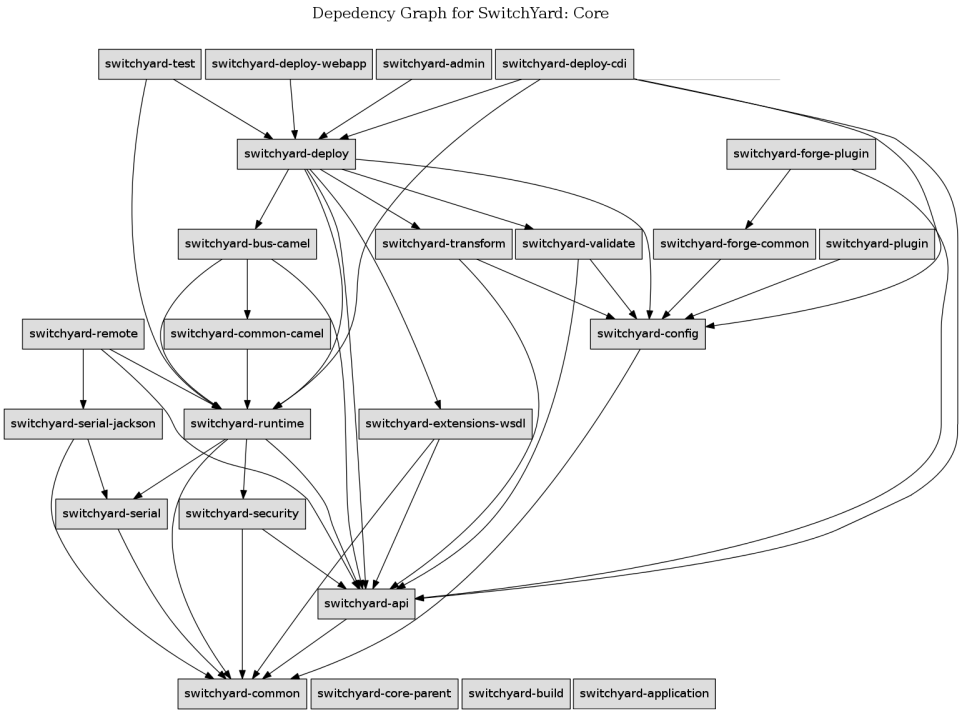
$$deg_{out}(root) = 0 \tag{2.2}$$

Tvrzení: kořenový uzel má každá komponenta v Maven grafu právě jeden. Důkaz: z definice komponenty, jako úplného souvislého podgrafu a předpokladu, že v Maven grafu nejsou chybně zavedeny cykly mohou nastat dva případy.

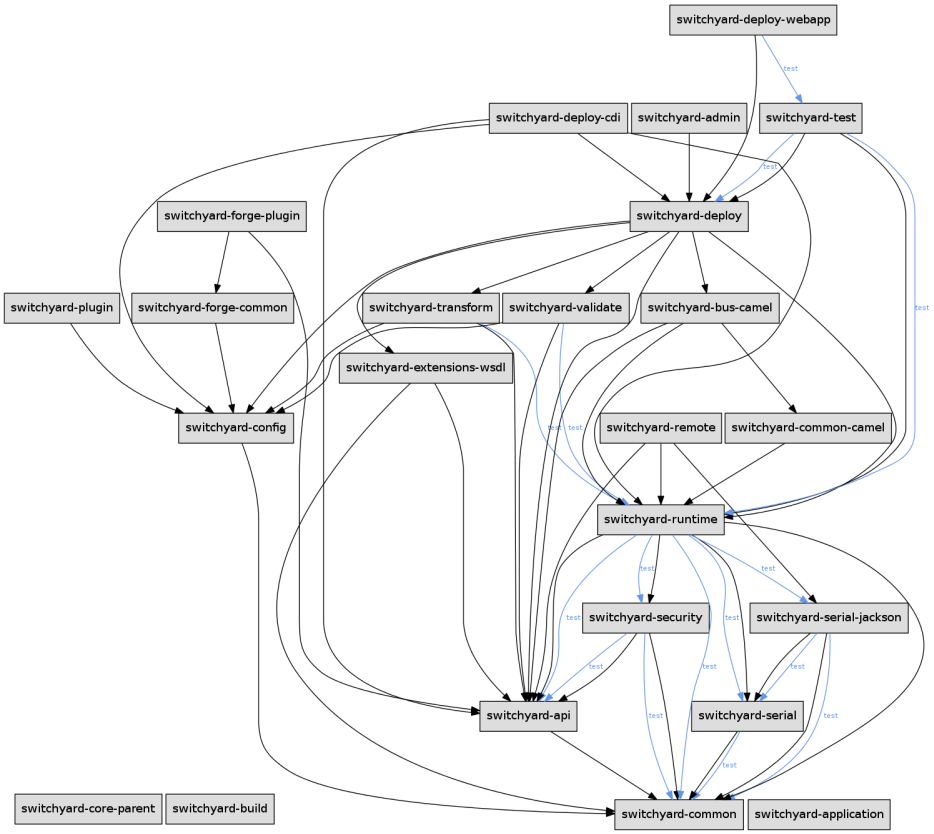
1) V grafu existuje komponenta, kterou tvoří právě jeden uzel V, pak $deg_{out}(V) = 0$.

2) Existuje-li komponenta mající více uzlů a v orientovaném Maven grafu a neexistuje cyklus, musí existovat uzel W, kde $deg_{out}(W) = 0$. Důkaz v literatuře [12].

Máme-li definován kořenový uzel komponenty a vybereme množinu vrcholů v komponentě $V \subseteq U$, lze tedy definovat funkci **vzdálenosti dvou uzlů** $maxdist(x,y)$ v Maven




a)




b)

Obrázek 2.7: Graf závislostí projektu SwitchYard Core 0.6.0.Final obsahující 4 komponenty a) závislosti typu „compile“ b) globální graf závislostí

grafu [2].

$$maxdist(x, y) = val, x, y \in V, \text{ kde } val \text{ je délka } \textbf{nejdelší} \text{ možné cesty mezi uzly } x \text{ a } y, \tag{2.3}$$

**Úrovní uzlu (modulu)**[2] v Maven grafu bude nyní modifikovaná funkce vzdálenosti $level(x)$, která určuje vzdálenost mezi kořenovým uzlem příslušné komponenty Maven grafu a libovolným uzlem komponenty $x \in V$.

$$level(x) = maxdist(root, x) \tag{2.4}$$

**Globálním Maven grafem** nazveme takový Maven graf, který může obsahovat závislosti jakéhokoliv typu, definované v kap. 2.3.1. Pokud dále nebude uveden typ Maven grafu, předpokládejme, že se jedná o globální Maven graf obsahující právě jednu komponentu. Ukázka takového grafu v praxi je na obrázku 2.7b. Graf obsahuje 4 komponenty, pořadí zpracování jednotlivých komponent není důležité a je určeno zpravidla pomocí pořadí v textu POM souboru, ať už se jedná o jakýkoliv cíl.

**Hloubka** *depth(T)* Maven grafu $T =< U, H >$ bude maximem ze všech úrovní uzlů ve všech komponentách v Maven grafu

$$depth(T) = \max\{level_T(x) | x \in U\} \tag{2.5}$$

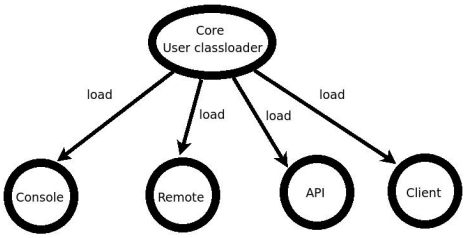
### 2.3.5 Paralelizace kompilace a testování modulů s Maven

Jak bylo řečeno na úvod, jazyk Java nativně nepodporuje koncept paralelní kompilace, a je potřeba při kompilaci jednotlivých zdrojových tříd dohledat veškeré potřebné závislosti. S využitím „Java Reflection API“ a classloader subsystému lze tento zastaralý způsob obejít.

Pokud je rozsáhlejší multimodulární projekt psán s jejich využitím, umožňuje udržovat minimální vazbu mezi moduly. Tato vazba vzniká až za běhu programu, kdy classloader dynamicky načte do jádra předem zkompilované moduly (analogie s plugin moduly), čili v době kompilace není potřeba rešit závislosti na prostředcích rodiče. Rozdělením do modulů (viz obr. 2.8) jsou tedy vytvořeny entity v době kompilace na sobě nezávislé, a tím dávají možnost k jejich paralelní kompilaci. Vždy musí existovat rodičovský modul, tzv. jádro (na obrázku „core“), které funkcionalitu CL implementuje a udržuje tak minimální dynamickou vazbu mezi moduly. Je potřeba také dodat, že CL by neměly sloužit k dynamické kompilaci celých modulů, což by vedlo k razantnímu poklesu výkonu běžící aplikace, ale pouze k nahrání již zkompilovaných modulů do paměti. Musí být také striktně definováno a přesně dodrženo API jednotlivých modulů. Tento koncept je hojně využíván pro své nesporné výhody, je však náročnější na implementaci a čitelnost kódu a musí se dodržovat určitá pravidla struktury projektu.

Paralelní testování je o něco méně problematické, protože závisí na úspěšném procesu kompilace. Paralelizace testovacího souboru jednotlivých modulů je triviálně realizovatelná v případě, že testovací třídy nemají závislosti mimo testovaný balík. V praxi toto však neplatí, jsou třeba závislosti nejen třetích stran, ale také závislosti na některém rodičovském modulu. Pak se musí zajistit přítomnost veškerých potřebných závislostí před vlastním testováním, např. prostřednictvím načtení z repozitáře.

[2] Pro jednoduchost bude pojem vzdálenosti a úrovně dvou uzlů v komponentě Maven grafu dále v práci označovat obecně tyto pojmy v rámci celého Maven grafu, myšleno tím bude však komponenty, do které uzel náleží.




Obrázek 2.8: Načítání paralelně kompilovaných modulů za běhu aplikace

## 2.4 Další implementace modularity

Jak bylo v úvodu autorem předesláno, tato práce se bude zabývat výhradně frameworkem Maven. Je ale vhodné přiblížit si také alternativy či trendy v oblasti modularity softwaru programovaném v jazyce Java. Některá z těchto alternativ se může do budoucna ukázat jako lepší řešení modularity. Již dnes některé alternativy Maven dokáží odstranit jeho nedostatky, zejména problematiku tzv. „Dependency Hell“.

Zavedení modularity slouží například k naprostému oddělení jmenného prostoru či potřeby přehledného systému závislostí. V současné době existují, nebo jsou v přípravě, kromě zmíněného frameworku Maven, dvě další implementace modularity v jazyce Java:

1. OSGi
2. Jigsaw

### „Dependency Hell“

Jedná se o problém se závislostmi na jiných verzích stejné knihovny. Máme-li projekt A, který závisí na projektu B, dále projekt C závislý na projektu B, ale také na projektu A, ovšem jiné verze. Java, čili ani Maven, tento problém nedokáží vyřešit, OSGi a Jigsaw však ano. Classloader subsystém je schopen načíst obě verze projektu A reprezentované pomocí speciálního archivu „bundle“ a spuštění či kompilace takové aplikace tedy není problém.

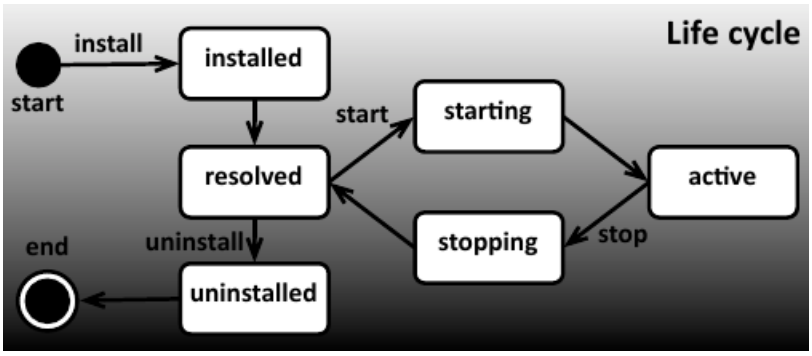
### 2.4.1 OSGi

OSGi je standard, organizující balíky do tzv. „bundle“, které jsou ekvivalentem archivů v jazyce Java. Každý bundle může přidávat do souboru META-INF/MANIFEST.MF (úryvek 2.4) uvnitř svého archivu informace o své verzi, aktivátoru (třídě, která zavádí bundle do systému), symbolickém jménu atd. Důležitou součástí souboru jsou také informace o závislostech, které jsou určeny řetězci Export-Package a Import-Package. Bundle samy o sobě by nemohou žádnou funkcionalitu poskytovat, proto jsou v OSGi přítomny součásti pro práci s nimi. Jedná se o doprovodné služby, rozhraní, manažery životního cyklu bundle, bezpečnostní prvky či podporu lokalizace. Každý bundle vstupuje do systému pomocí svého aktivátoru, jeho životní cyklus je znázorněn na obrázku 2.9.

Úryvek kódu 2.4: Metadata v MANIFEST.MF

```
...
Bundle-ManifestVersion: 3.0
Bundle-Name: Service
Bundle-SymbolicName: org.service.module
```

```
Bundle-Activator: org.service.module.Activator
Import-Package:   org.http.request;version="2.0.0"
Export-Package:   org.service.module.core;version="3.0.0"
```




Obrázek 2.9: Životní cyklus bundle[3]

Oproti Maven nebo Jigsaw je managment závislostí spuštěn až za běhu prostředí. OSGi umožnuje souběžný běh aplikací v rámci jednoho VM, ale udržuje izolaci mezi aplikacemi. Reimplementuje také classloader subsystém, který má v Javě některé nedostatky, jako například obtížná správa CLASSPATH z více aplikací nebo nemožnost mít více verzí jednoho balíku uvnitř jedné JVM. Pomocí uvedených vlastností a systému závislostí také řeší OSGi problém „Dependency Hell“. Doposud tři nejznámější implementace standardu OSGi jsou Apache Felix, Eclipse Equinox a Knopflerfish.

### 2.4.2 Jigsaw

Projekt Jigsaw je implementace modularity, která je vyvíjena jako budoucí komponenta jazyka Java (minimálně implementace OpenJDK). Aktuální informace hovoří o zařazení do Javy verze 9. Celkově je cílem projektu odlehčení aplikací od použití veškerých součástí robustního prostředí JVM formou odstranění nutnosti importu nepotřebných rozhraní, zrychlení prostředí, defragmentace platformy, zavedení modulů, správy závislostí, jejich distribuce atd. Zavádí do jazyka nová klíčová slova, která umožní koncept modularity rozvést. Ukázku navrhované syntaxe je zobrazena v úryvku 2.5.

Úryvek kódu 2.5: Moduly s Jigsaw

```
module m1 {
        requires m2 @>= 1.0;
        requires optional m3;
        class foo.Main;
}

module m2 @ 1.0 {
        exports m2;
}
```

V tomto případě máme modul *m2*, který exportuje veškeré veřejné typy uvnitř modulu pro použití mimo něj. U modulu je také udána jeho verze. Dále existuje modul, které tyto typy importuje, avšak klade si podmínku, že verze importovaného modulu musí být větší

nebo rovna číslu 1.0, což je splněno. Dále modul *m1* explicitně určuje vstupní bod, kde bude hledána tradiční vstupní metoda main(). Jako volitelná je deklarována také závislost *m3*. Pomocí zavedení explicitních verzí u modulů je Jigsaw také schopen řešit problém Dependency Hell.

## 2.5 Frameworky Junit a TestNG

Junit je framework umožňující psát jednotkové testy tříd („unit testy“) v jazyce Java. Je distribuován formou JAR archivu, skládajícího se z knihovny tříd, definic anotací a dalších prostředků potřebných pro testování. Framework se rozšířil i do jiných jazyků, jako jsou Perl, Python, C++ atd. TestNG je mladší framework vycházející z Junit, snažící se odstranit některé jeho nedostatky jako nelogičnost některých anotací či malou granularitu testovacích možností na úrovni sdružování testů do skupin. Cílem testování je tvorba testovacích tříd ke třídám existujícím nejlépe tak, aby testovaná funkcionalita v testovací třídě pokrývala maximum funkcionality třídy implementované.

Co se týče syntaxe, oba frameworky jsou si velice podobné. Ukázkou může být jednoduchý test ke třídě Adder.java, který lze zapsat z užitím některého z frameworků jako v úryvku 2.7.

Úryvek kódu 2.6: Adder.java

```
...
public class Adder{
        private int value = 0;

        public void sum(int sum) {
                result += sum;
        }

        public int getResult() {
                return value;
        }
}
```

Úryvek kódu 2.7: Testovací třída pro Adder.java v Junit a TestNG

```
...
public class TestAdder {
        @Test
        public void sum() {
                Adder add = new Adder();
                add.sum(10.0);
                add.sum(12.0);
                assertEquals(22.0, add.getResult());
        }
}
```

Pro takto jednoduchý příklad nelze vidět rozdíly mezi oběma frameworky, rozdíly přicházejí až s komplexnějšími potřebami. Junit i TestNG také podporují celou řadu anotací,

které jsou jedním z prostředků, kterým je v podstatě definována funkcionalita frameworku a šíře testovacích schopností. Soubor anotací TestNG je širší:

| Feature | JUnit 4 | TestNG |
|---|---|---|
| test annotation | @Test | @Test |
| run before all tests in this suite have run | – | @BeforeSuite |
| run after all tests in this suite have run | – | @AfterSuite |
| run before the test | – | @BeforeTest |
| run after the test | – | @AfterTest |
| run before the first test method that belongs to any of these groups is invoked | – | @BeforeGroups |
| run after the last test method that belongs to any of these groups is invoked | – | @AfterGroups |
| run before the first test method in the current class is invoked | @BeforeClass | @BeforeClass |
| run after all the test methods in the current class have been run | @AfterClass | @AfterClass |
| run before each test method | @Before | @BeforeMethod |
| run after each test method | @After | @AfterMethod |
| ignore test | @ignore | @Test(enable=false) |
| expected exception | @Test(expected = ArithmeticException.class) | @Test(expectedExceptions = ArithmeticException.class) |
| timeout | @Test(timeout = 1000) | @Test(timeout = 1000) |

Obrázek 2.10: Anotace Junit a TestNG [1]

**Functionality - JUnit 4 vs TestNG**

| | Annotation Support | Exception Test | Ignore Test | Timeout Test | Suite Test | Group Test | Parameterized (primitive value) | Parameterized (object) | Dependency Test |
|---|---|---|---|---|---|---|---|---|---|
| **TestNG** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **JUnit 4** | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ |

Obrázek 2.11: Obecné srovnání Junit a TestNG [1]

Obecné srovnání obou frameworků je na obrázku 2.11. Nevýhod Junit je tedy oproti TestNG hned několik. Junit testovací třída musí obsahovat anotace @BeforeClass a @AfterClass, dále nemá možnost využití skupin testů („groups“), musí často dědit od nějaké předem definované třídy, omezuje jména metod, neexistují závislosti testů na sobě, nepodporuje paralelní testování. Framework TestNG se jeví tedy jako flexibilnější.

Pro testování menších aplikací se tedy rozdíly obou stírají, u komplexnějších však začne vynikat větší flexibilita TestNG, zejména v problematice kategorizace a spojování testů do balíků.

### 2.5.1 Kategorizace a balíky testů

Oba frameworky definují prostředky pro vytváření skupin testů. Je možné vytvořit tzv. testovací balíky („test suites“), v TestNG je také možné navíc organizovat testy do testo-

vacích tříd („test groups“), což by se dalo vyložit jako kategorizace testů. Ukázka kódu pro spuštění balíku testů v Junit je v úryvku 2.9.

Úryvek kódu 2.8: Junit test suite

```
...
@RunWith(Suite.class)
@Suite.SuiteClasses({
        Junit1.class,
        Junit2.class
})
public class MyTestSuite {
}
```

Anotace říkají frameworku, že pokud bude spuštěno testování třídy *MyTestSuite*, bude také spuštěno testování tříd *Junit1* a *Junit2*, čímž dochází ke spojení testů do balíku. V TestNG není nutno zasahovat do kódu, ale je možné předat jako vstup procesu testování XML soubor, ve kterém mohou být explicitní informace o skupinách, balících a procesu jejich testování. Ukázka souboru testng.xml je v úryvku 2.9.

Úryvek kódu 2.9: XML soubor s metadaty o procesu testování v rámci TestNG

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="MySuite">
  <test name="MyTest">
    <groups>
      <run>
      <include name="performanceTestLabel"/>
      <exclude name="optimalizationTestLabel"/>
      </run>
    </groups>
    <classes>
      <class name="org.tests.classTest"/>
        <methods>
          <include name="myTestMethod" />
        </methods>
    </classes>
    <packages>
        <package name="org.tests.performanceTests.*"/>
    </packages>
  </test>
</suite>
```

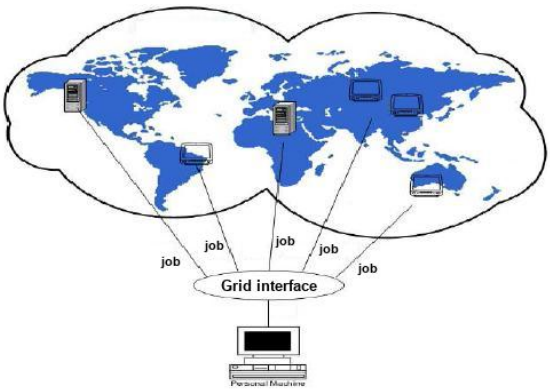
Sekce „group“ XML souboru říká, které skupiny testů se mají nebo nemají zahrnout do procesu testování. Zahrnuty tedy budou testy, které jsou anotovány pomocí *@Test(groups="performanceTestLabel")*. Sekce *classes* naopak do testování zahrne třídu *classTest*, ne však celou, ale jen její metodu *myTestMethod()*. Nakonec jsou připojeny také veškeré testy z Java balíku *org.tests.performanceTests*. Konfigurační soubor může být široce modifikován, může obsahovat různé inkluze a exkluze testů, metod, skupin či Java balíků a nastavovat jejich parametry. Zájemci najdou podrobný popis v manuálu TestNG[2].

Z uvedeného příkladu lze tedy jasně vidět vetší flexibilitu TestNG, alespoň v organizaci testovacího souboru. Oba frameworky se však neustále vyvíjí a snaží se přidávat nové prostředky pro zvětšení flexibility testování a celkové použitelnosti.

# Kapitola 3

# Architektura grid

Architektura grid je definována jako soubor heterogenních, různorodě geograficky rozložených výpočetních prvků (uzlů), spolupracujících různým dílem na dosažení výsledků společného cíle, který je následně prezentován uživateli jako výstup jednoho komplexního výpočetního systému. Jednotlivým výpočetním jednotkám se říká uzly. Výpočet na jednotlivých uzlech probíhá pro uživatele transparentně, je však možnost zřídka k jednotlivým uzlům přistupovat i individuálně.




Obrázek 3.1: Distribuovaná grid architektura [7]

Tato architektura je tedy velmi vhodná pro výpočty paralelního charakteru (paralelní algoritmy), u kterých potřebujeme rozdělit výpočet (program) na více nezávislých částí a na závěr zredukovat výsledek do jednotného výstupu, což je přesně případ paralelizace testovacích úloh v jazyce Java. Tento algoritmus řešení rozdělení problému a následné agregace dat se nazývá MapReduce. Běží-li testování jako sekvenční proces na jednotlivém výpočetním prvku, může se jednat o velice zdlouhavý proces. Je tedy třeba vhodně rozdělit testovací soubor, ten distribuovat na jednotlivé uzly grid architektury, provést paralelní otestování sestaveného produktu a kompletaci výsledků testů. Výhodou je v tomto případě heterogennost gridové architektury, které nahrává vlastnostem jazyka Java a to možnosti spuštění aplikace nezávisle na použité architektuře.

Testovací soubor produktu tedy může být nejen prováděn na uzlech se specifickou architekturou, nýbrž také může být dělen na jednotlivé části spustitelné výhradně na požadované platformě (např. otestování specifických testů pro určitý operační systém pouze na uzlu s tímto systémem). Takové flexibilita je tedy cestou ke zkvalitňování softwarového

produktu již při jeho vývoji.

Jak již bylo uvedeno v úvodu, jako integrační software bude použit nástroj Jenkins. Vyvstává otázka, zda-li bude vůbec potřeba s nástrojem Jenkins integrovat nějaký další software reprezentující gridovou vrstvu. Tedy jestli přidání gridové vrstvy přinese některé výhody, nebo zda-li nepřistupovat k jednotlivým přiděleným strojům přímo přes integrační software bez podobné vrstvy. Je třeba srovnat výhody a nevýhody jednotlivých existujících implementací, frameworků, middleware či knihoven, které mohou dopomoci ke zefektivnění, zkvalitnění či vylepšení návrhu řešení testovacího softwaru, který je předmětem této diplomové práce.

## 3.1 Open source implementace grid architektury

Pojmem open source myslíme filosofii vývoje software, jehož kódy jsou veřejně dostupné, upravitelné a volně distribuovatelné. Open source implementace mají řadu výhod. Není potřeba zajišťovat specifické licence softwaru na počty uzlů či procesorů s rostucí infrastrukturou, je možno si software upravovat dle libosti, spolupracovat s komunitou apod. Nevýhodou je například absence firemní podpory, kterou naopak autoři často nabízejí komerčně. Obecně lze druhy gridových implementací rozdělit následovně:

- Knihovna (library) - implementace určitých znovupoužitelných entit (objektů, funkcí) programovacího jazyka
- Framework - soubor knihoven, nástrojů a modulů pro cílenou implementaci problému pomocí osvědčených řešení, důraz na rychlost vývoje, zpravidla směruje nebo omezuje vývojáře funkcionalitou, výkonem, způsobem programování apod.
- Middleware - je zpravidla software reprezentující vrstvu mezi vyvíjeným produktem a dalšími komponentami, například fyzickou vrstvou, přilehlé vrstvy spojuje, nezávislý na platformě
- Toolkit - soubor nástrojů a postupů pro tvorbu specificky zaměřeného software

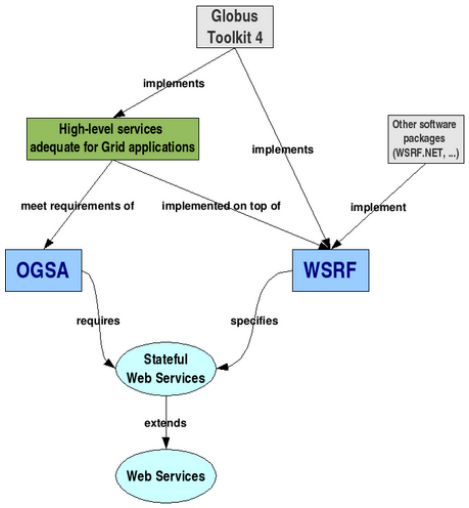
## 3.2 Globus Toolkit

Jedná se o komplexní softwarový produkt sdružení The Globus Alliance psaný v jazycích Java a C, umožnující uživatelům využívat a sdílet výpočetní prostředky, data, databáze a nástroje bezpečně bez ohledu na firemní, institucní a geografická omezení a beze ztráty lokální autonomie. Spravuje softwarové služby, knihovny pro jejich monitorování, vytváření, zabezpečení a lokaci. Zahrnuje také software pro datový, komunikační, bezpečnostní a informační managment, dále software pro detekci chyb, pro zajištění přenositelnosti či QoS. Je zaměřen na zastření rozdílů mezi organizacemi, často využívajícími jiných prostředků pro provoz vlastních gridových infrastruktur (různě geograficky umístěných) a sdružování těchto organizací do komplexnějších celků se sdílením vzájemných zdrojů [21].

Gloubus definuje řadu standardizací, na kterých je toolkit postaven. Vzájemné vztahy jsou ilustrovány na obrázku 3.2.

- WSRF - Web Services Resource Framework - specifikace pro tvorbu tzv. „statefull web services“. Jedná se o stavové služby identifikované jednoznačným URI (analogie s webovým URL), poskytované gridem koncovým uživatelům. Specifikace je vyvíjena sdružením OASIS, je nástupcem starší specifikace OGSI.

- OGSA - Open Grid Services Architecture - standard definující architekturu poskytování gridových služeb a popisující rozhraní pro práci s nimi. Vyžaduje mimo jiné použití „Web services“ poskytovaných WSRF frameworkem. Snaží se o jednotnou definici veškerých užívaných služeb ve společném souboru definic.



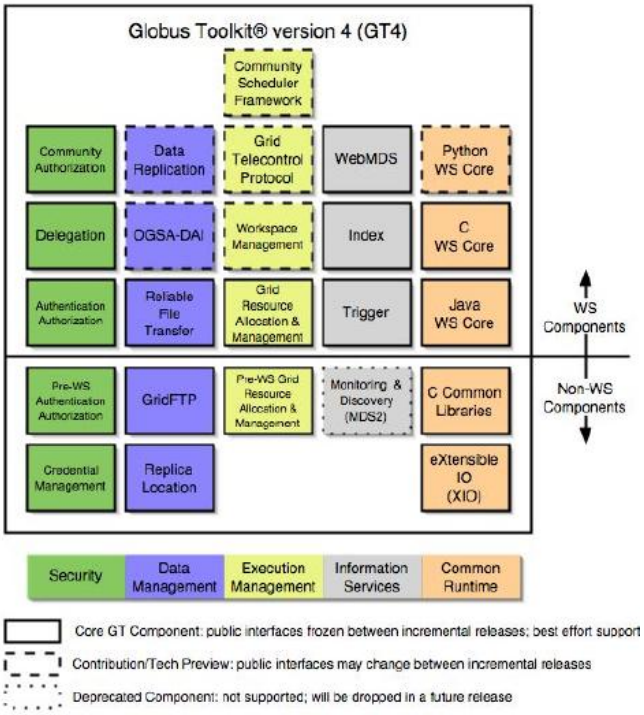

Obrázek 3.2: Vztahy Globus specifikací [17]

Nejdůležitější části struktury toolkitu jsou na obrázku 3.3. Struktura projektu je dělena na dvě hlavní části. První část jsou již zmíněné „web services“, která obsahuje služby tvořené pomocí WSRF. Druhá část obsahuje doprovodnou funkcionalitu pro chod gridu, jako je služba datového úložiště (GridFTP) či plánovač úloh GRAM (Grid Resource Alocation & Managment), který zajišťuje přistup, monitoring a správu služeb. Slouží také jako unifikované rozhraní pro plánování úloh a managment služeb. Bezpečnost je obecně zajišťena vrstvou Grid Security Interface (GSI, na obrázku vrstva Security), která je založena na mechanismu proxy certifikátů, na kryptografii privátních a veřejných klíčů a digitálních podpisech. Nedílnou součástí je i MDS - Monitoring and Discovery Service, která slouží pro správu a monitorování prostředků.

Celkově je tedy Globus toolkit určen pro vědecká HPC prostředí, superpočítače či velké nadnárodní gridové infrastruktury. Není nutno však použít celý toolkit jako takový, je možno využívat jen určité jeho součásti. Využití našel například v obrovských vědeckých výpočetních a datových projektech v CERN, Southern California Earthquake Center, TeraGrid, NASA, DARPA [20].

Výhody:

- Bezpečnost
- Široká škála poskytovaných služeb
- Vysoká modularita a konfigurovatelnost
- Stabilní, praxí ověřený produkt
- Podpora Unix, MAC OS, Linux




Obrázek 3.3: Struktura Globus Toolkitu verze 4 [17]

Nevýhody:

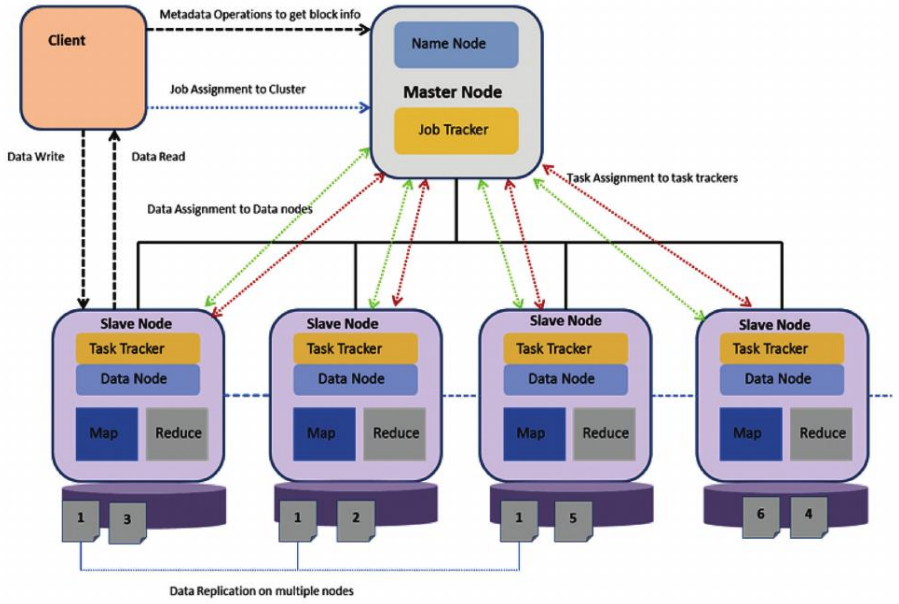
- Robustnost
- Náročné na administraci
- Omezená podpora Windows
- Částečná omezení jazyka Java
- Omezené možnosti integrace

## 3.3 Apache Hadoop

Velmi významným gridovým řešením je od firmy Apache framework Hadoop. Je psán kompletně v jazyce Java a distribuován pod Apache v2 licencí. Je zaměřen na distribuované výpočty nad velmi velkými objemy dat.

Jádrem systému je algoritmus MapReduce, který nejprve rozdělí vstupní data na několik stejných dílů a rozdistribuuje je na všechny výpočetní uzly. Každý jednotlivý uzel provede operaci MAP nad svou množinou dat a tyto výstupy se agregují operací REDUCE a uloží do sdíleného souborového systému. Tento přístup je ideální pro námi řešenou úlohu, pokud si představíme dělenou množinu jako celý soubor testů vyvíjeného softwarového produktu.

Hadoop poskytuje nativní sdílené úložiště formou svého souborového systému HDFS. Tento souborový systém slouží jako úložiště pro veškeré soubory dat po celou dobu výpočtu. Ukládá jak vstupní data, tak jejich jednotlivé rozdělené části. Je třeba poznamenat, že HDFS je systém tzv. „write-once-read-many“ typu, kdy je možné na disk soubor zapsat,




Obrázek 3.4: Struktura Apache Hadoop [6]

ale už jej nelze měnit. Toto omezení umožňuje zvýšit úroveň souběžnosti zpracování dat. Soubory jsou ukládány formou bloků, které jsou distribouvány skrze více uzlů. Dále je třeba zmínit optimalizaci na velmi velké soubory a odolnost proti chybám pomocí replikace dat.

Architektura je typu master/slave a je rozdělena na několik částí dle obrázku 3.4. Na master uzlu najdeme komponenty Namenode a Job Tracker. První zmiňované slouží jako úložiště metadat systému HDFS a řídí I/O operace slave uzlů. Namenode je místem možných potencionálních poruch - dojde-li k jeho poruše, systém selže - i proto je dobré využít možnosti zavést do systému sekundární Namenode. Job Tracker je místem plánování úloh, řídí celou operaci MapReduce popř. provádění zotavení se z chyb. Běží zpravidla vždy na master uzlu. Task Tracker slouží k lokálnímu vykonávání zadaných operací od Job Trackeru. Reaguje na události během výpočtu a informuje jej o průběhu. Poslední komponenta, Datanode, spolupracuje s HDFS a spravuje veškeré datové procesy slave uzlu, stará se i o replikace dat. Komunikuje také s Namenode, kterého informuje o své činnosti.

Hadoop se využívá na zpracování mohutných záznamů (logů), analýzu webových stránek, práci s vyhledávacími indexy či pro strojové učení - obecně tam, kde je třeba načíst data v celém svém objemu a nad nimi provádět operace. Používají jej Google, Amazon, Yahoo!, Facebook, apod.

Výhody:

- Automatická reakce na změny topologie a detekce chyb
- Vhodný pro zpracování velkých datových objemů jako celek
- Streamování dat
- Lineární škálovatelnost
- Vlastní MapReduce, lokální rychlé zpracování

- Vlastní spolehlivý souborový systém HDFS
  - Redundance i na levnějším hardware, není potřeba přídavný HW (RAID)
  - Výborná schopnost zotavení se z chyb
  - Vysoká propustnost pro velké množiny dat
- Podpora Linux, Unix, MAC OS

Nevýhody:

- HDFS vrstva způsobuje narůstání času přístupu k datům (oproti paměťovém gridu)
- Částečná podpora Windows
- Náročná konfigurace
- Master uzel může být slabým místem

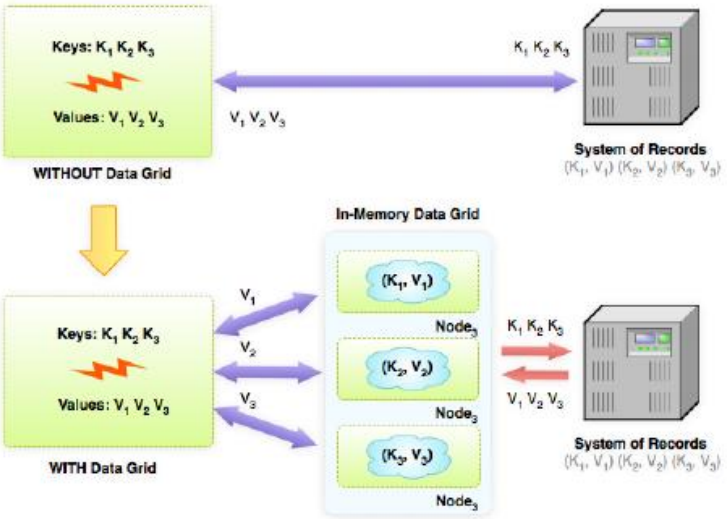
## 3.4 GridGain

GridGain je implementací výpočetního gridu, který lze také kombinovat z některou implementací datových gridů do jednoho celku. Je distribuován pod licencí LGPL. Jedná se o middleware kompletně v jazyce Java, je určen pro náročné distribuované výpočty v reálném čase. Tvůrci jej označují podtitulem „In-Memory Data Platform“, který napovídá o dalším odlišném přístupu k implementaci grid architektury, než jeho dva předchůdci. Oproti Hadoop neuvažuje ve svém návrhu nativní úložiště, ale snaží se operovat co nejvíce s daty v paměti, což se musí logicky nepříznivě projevit na ceně infrastruktury (ovšem ne v ceně na 1GB RAM). Naopak je tedy možné dosahovat vyšší výkonnosti výpočtů. Zakládá si na využití algoritmu MapReduce, jako úložiště dovede využít například HDFS systém z Hadoop. Dokáže přehledně spravovat topologii pomocí rozsáhlého rozhraní SPI (Service provider interface). Struktura GridGain je na obrázku 3.5.

GridGain je navíc určen pro Java developery a podporuje řadu pro ně určených technologií, jako jsou JBoss, Spring, JGroup či vývojářská IDE (Netbeans, Eclipse). Disponuje řadou dalších výhod jako podpora stremovaného MapReduce, distribuovaných úkolů, mechanismy load-balancing, „Zero deployment“ apod. Pro své výhody je tento framework velmi rozšířený, nejznámější je jeho použití v projektu SETI (Search for Extraterrestrial Intelligence), dále jej používají firmy jako Apple, Cannon, Sony. Je použit v oblastech jako jsou analýza investičního rizika, bioinformatika, výzkum medicíny či online vzdělávání.

Výhody:

- 100% zaměřeno na jazyk Java
- Jednoduchost a produktivita
- Automatická škálovatelnost a reakce na změny topologie
- Vylepšený load balancing
- Dynamické a adaptivní MapReduce (podpora i pro streamované)
- Zero deployment




Obrázek 3.5: GridGain „In-memory“ grid [4]

- Rozsáhlé možnosti integrace (JBoss, Spring, JGroups, Webspere, ...)
- Distribuovaná a/nebo replikovaná efektivní cache
- Podpora Linux, Unix, MAC OS i Windows

Nevýhody:

- Mladý projekt
- Občasné problémy s nestabilitou

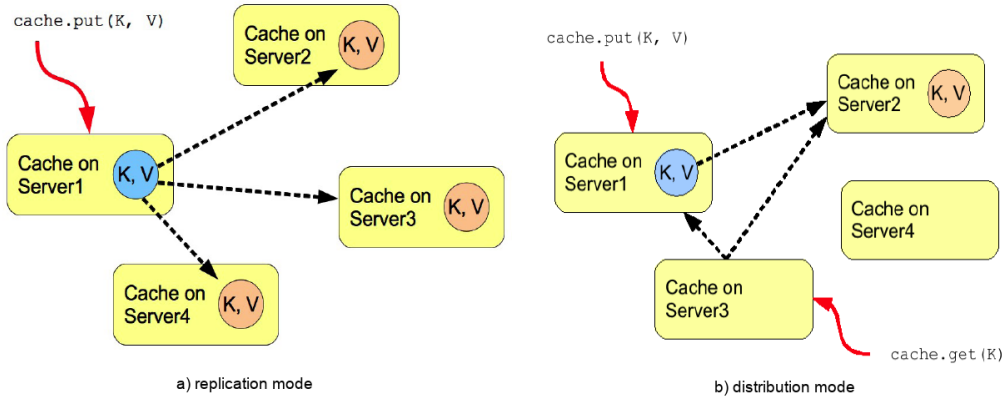
## 3.5 Infinispan

Infinispan je framework pro tvorbu datového gridu vyvíjený firmou Red Hat. Funguje jako distribuovaná datová cache, která ukládá záznamy ve tvaru $\langle klic, hodnota \rangle$. Jakýkoliv vložený záznam je pak dostupný z kteréhokoli uzlu v gridu. Obdobně jako GridGain, jsou veškerá data uchovávána v paměti. Tzv. „State-of-the-art“ jádro Infinispanu je optimalizováno pro vysokou úrovně souběžnosti procesů, zejména na vícejádrových procesorech. Snaží se klást důraz na neblokující algoritmy a minimum synchronizačních sekcí. Je také optimalizován pro použití jako obrovská efektivní paměť napříč stovkami serverů. Datový grid je také velmi dobře škálovatelný, teoreticky neexistuje horní limit počtu uzlů, a to díky minimalizaci komunikace mezi uzly pomocí peer-to-peer připojení. Komunikace v gridu je omezena pouze na zjišťování přítomnosti nových uzlů a rovnoměrnou distribuci dat napříč gridem.

Celý systém může pracovat ve dvou základním módech, replikace či distribuce, viz obrázek 3.6. Replikační mód kopíruje data mezi všechny uzly, avšak je vhodný pro malé velikosti gridu (do 10 strojů), a to kvůli narůstajícímu množství síťové komunikace. Data jsou pak rychle dostupná každému uzlu lokálně, lze tak například sdílet stavové komponenty. Distribuční mód je vhodný pro větší počty uzlů v gridu, je možno jej kombinovat s replikací,

která v tomto případě tvoří kompromis mezi výkonností a odolností dat proti chybám. Obrovská výhoda tohoto přístupu je lineární škálovatelnost.

Využití našel Infinispan jako sekundární paměť cache pro Hibernate framework, HTTP cache v JBoss AS 6 a 7 či úložiště pro indexační nástroje (Lucene).




Obrázek 3.6: Módy běhu Infinispan [18]

Výhody:

- Škálovatelnost
- Vlastní MapReduce implementace
- Velmi malé přístupové doby do paměti
- Schopnost adresovat velké množství paměti skrze mnoho uzlů
- Připraveno pro integraci do Cloud prostředí
- Serverový modul spolupracující s dalšími jazyky (C, PHP, Ruby, ...)
- Managment velkého množství strojů
- Podpora Linux, Unix, MAC OS i Windows

Nevýhody:

- Pouze datový grid - omezená šíře použitelnosti

## 3.6 Zhodnocení

Tato kapitola porovnává gridové implementace a nastiňuje možnosti pro návrh a celkovou implementaci vyvíjeného systému. Zhodnocuje klady a zápory jednotlivých softwarových produktů, konečný výběr vhodné alternativy bude proveden až v kapitole zabývající se návrhem (5).

# Kapitola 4

# Integrační prostředí Jenkins

Téma následující kapitoly je patrno z názvu, je potřeba však informovat, že nástroj Jenkins je velmi komplexní a zde budou rozebrány jen důležité části spojené s předmětem této práce. Popis značné části software je možno nastudovat z [16].

## 4.1 Průběžná integrace

Pokud bylo dříve při vývoji softwarového projektu užíváno vodopádového či iteračního modelu, nastával často problém při integraci projektu jako celku. Integrací se myslí proces, kdy je kód od každého vývojáře přesunut například do společného umístění a je týmem nebo jednotlivcem integrován do celkové implementace. Takový proces může trvat měsíce a je třeba zpětně dohledávat implementační detaily či spojovat obrovské kusy kódu, což rozhodně není příspěvkem ke zvýšení produktivity.

Průběžná integrace (Continuous integration - CI) je metoda vývoje softwaru, kdy se v krátkých, nejlépe periodicky se opakujících intervalech celý software integruje formou tzv. „buildů“. Jedná se o vývojové verze software, které prošly sestavováním, testováním, kontrolou kvality kódu a distribučním procesem. V podstatě CI software monitoruje uživatelský či sdílený repozitář a v případě detekce změny vyprodukuje transparentně na programátorovi build. Dojde li k chybě, je programátor neprodleně informován. Proces může být spouštěn nezávisle na změnách, ale periodicky v určitých časových intervalech. Nejedná se tedy jen pouze o kompilaci projektu, velice důležité jsou i následné kroky. Metoda se vyvinula jako část tzv. Extreme programming (XP), která klade důraz na rychlou reakci na změny v požadavcích zadavatele nebo koncového uživatele. XP je agilním modelem vývoje softwaru, oproti vodopádovému modelu klade důraz na časový rámec ve vydávání verzí projektu, zpravidla dochází k integraci několikrát denně. CI obecně poskytuje tedy celou řadu užitečných prostředků či služeb:

- Automatizace vývoje
- Okamžitý přístup k poslední verzi aplikace
- Spolupráce s verzovacími systémy
- Zrychlení vývoje
- Zkvalitnění kódu
- Rychlé nalezení chyb ve zdrojovém kódu a metriky

- Automatická kontrola kódu
- Šetření času a výpočetních kapacit
- Časované plánovaní

CI může být i určitou formou komunikace se zákazníkem. Zákazník má neustálý přehled o vývojové fázi produktu, navíc jsou dostupné jeho spustitelné verze, takže může produkt vyzkoušet a reflektovat změny či nové požadavky. Celkově při vývoji software narůstá výpočetní náročnost na integraci projektu jako celku s počtem chyb v kódu, počtem komponent (modulů) a času od poslední integrace projektu. Nástroje pro průběžnou integraci však dokáží ty problémy do značné míry minimalizovat.

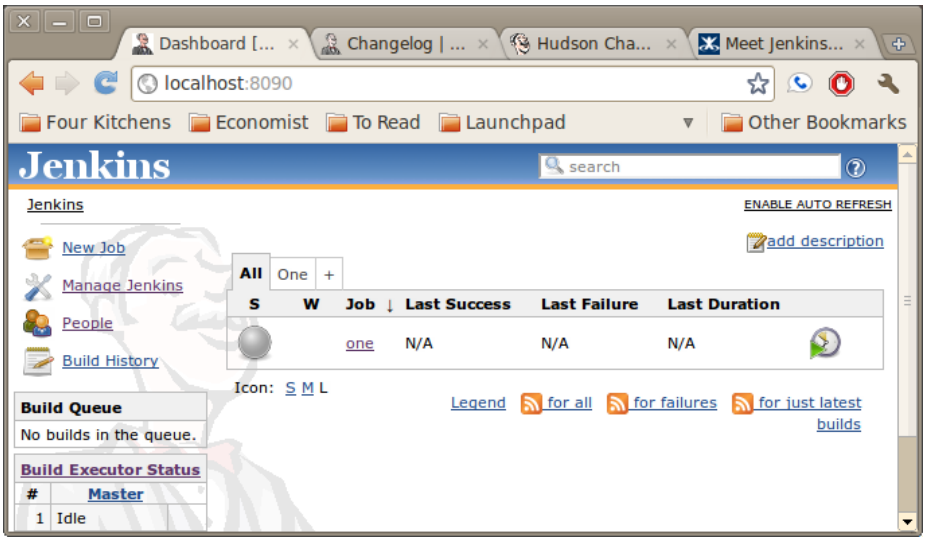
## 4.2 Charakteristika Jenkins

Projekt Jenkins CI je nástrojem průběžné integrace softwarových produktů. Je odnoží populárního projektu Hudson, nyní spravovaným firmou Oracle. Je open source, psán v jazyce Java, široce rozšířen a oblíben pro vývoj aplikací. Je volně dostupný a upravitelný. Existuje také odnož a placená verze, spravována firmou CloudBees. Je cílen na vývoj v jazyce Java, ale podporuje i další jazyky či technologie jako Ruby, PHP, CVS, SVN, Ant, Maven, Junit, TestNG a nejrozšířenější operační systémy, z nejznámějších Linux, Unix, MAC OS a MS Windows. Teoreticky je však možné nástroj díky obecnosti použít pro vývoj v jakémkoli programovacím jazyce. Je do značné míry modifikovatelný formou plugin modulů, kterých existuje široká řada. Jenkins je jednoduše spustitelný v systému s nainstalovaným a funkčním prostředím Java pomocí příkazu 4.1.

Úryvek kódu 4.1: Spuštění Jenkins serveru

```
java -jar jenkins.war
```

Po spuštění serveru je možné přistupovat k Jenkins server webovému rozhraní implicitně na portu 8080 (obrázek 4.1).




Obrázek 4.1: Pohled na webové rozhraní Jenkins [11]

Funkcionalita Jenkins na základní úrovni v podstatě spočívá ve vykonání uživatelsky nastavitelných příkazů ve správně nakonfigurovaném lokálním či vzdáleném operačním systému, s využitím jeho proměnného prostředí (Linux, Unix, MS Windows, MAC OS). Soubor takových příkazů se nazývá úloha („job“) a je elementární pracovní jednotkou. Uživatel může vytvořit úlohu pomocí grafického (webového) uživatelského rozhraní kliknutím na položku „New job“ (obrázek 4.1). Pro zjednodušení práce s určitými typy projektů je možno doinstalovat plugin pro různé typy úloh. Každé úloze je přidělena pracovní plocha prostřednictvím absolutní cesty na disku, ve které jsou poté příkazy vykonávány. Mohou to být instrukce pro stažení zdrojových souborů z repozitáře, příkazy pro kompilaci a testování, práci s různými frameworky či jak přímo standardní příkazy cílového OS, tak i skripty. Příklad úlohy pro načtení z repozitáře, spuštění a otestování standardního Maven modulu bez podpory jakýchkoli plugin modulů v Linux, může být po správné konfiguraci Jenkins projektu následující:

Úryvek kódu 4.2: Příkazy pro otestování Maven projektu na OS Linux

```
rm -rf module/
git clone https://git.source.com/module
cd module/
mvn compile
mvn test
```

Uvedené příkazy jsou spuštěny nad přidělenou pracovní plochou a produkují výsledek, který je prezentován na samostatné stránce ve webovém rozhraní. Nechybí ani informace o úspěšném dokončení nebo selhání procesu.

Do prostředí Jenkins je také vhodné zařadit množinu strojů (uzlů), které bude mít pod svou správou. Existují dva typy uzlů, uzel master a uzly slave. Uzel master musí vždy existovat jeden a myslí se jím stroj, na kterém běží aktuální instance Jenkins. Zde jsou prováděny veškeré činnosti plánování, režie, práce s úlohami i persistence dat. Uzly slave jsou stroje, které jsou přidány do konfigurace Jenkins za běhu programu a zpravidla fungují jako rozšiřování výpočetní kapacity systému. Jenkins do jisté míry poskytuje nástroje pro automatickou konfiguraci uzlu, ne vždy je však efektivní a je třeba konfigurovat manuálně. Automaticky je možno nainstalovat na cílový uzel prostředí pro jazyk Java, či nainstalovat framework Maven. V případě užití master/slave režimu (viz 4.2.1) je třeba také nakonfigurovat jednotlivé slave uzly na úrovni operačního systému a komunikaci mezi nimi. Je možné také nastavovat rozsáhlé možnosti zabezpečení, testování, sestavování, vizualizace, agregace dat, notifikací e-mailem, messengery, IRC, SMS apod.

### 4.2.1 Režimy práce

Jenkins může operovat ve dvou módech: master/slave módu a normálním módu. V normálním módu probíhají veškeré úkony na stroji, kde běží Jenkins CI server. Tento postup ale není vhodný pro paralelní běh částí integrace, kterého se práce snaží docílit.

V master/slave módu je možnost staticky přidat pod správu Jenkins serveru nakonfigurované uzly. Existuje řada druhů ustavení master-slave spojení, a to pomocí nakopírování slave klienta (soubor slave.jar) přes SSH na uzel, či pomocí Java Web Start (JNLP) technologie. Tento mód je tedy teoreticky vhodný pro spolupráci s gridovou infrastrukturou, má však značná omezení, která se snaží odstranit alespoň některé pluginy. Nativně nelze například provádět distribuované sestavování a paralelní testování na úrovni slave uzlů (testy

musí běžet vždy na slave uzlu, na kterém se provedlo úspěšné sestavení projektu). V kapitole 5 se autor bude snažit navrhnout funkcionalitu odstraňující tyto nedostatky. Je zde možnost i výběru speciálních uzlů pro speciální projekty, a to pomocí „label“ tagů, kdy je možno explicitně zadat, na kterém uzlu se má projekt sestavit a otestovat.

### 4.2.2 Struktura úlohy

Úloha („job“) v Jenkins se dá obecně rozdělit do několika částí či fází:

- Příprava („Pre-build“)
- Vykonání („Execution“)
- Dokončení a úklid („Post-build“)

Jednotlivé části lze dále dělit. Přípravna fáze se dále skládá v závislosti na typu úlohy z kroků, jako je stažení zdrojového kódu úlohy z lokálního nebo vzdáleného úložiště, určováním pořadí jednotlivých úloh (čekání na „upstream“ závislosti) nebo prací s konfiguračními soubory. Proces vykonání úlohy je v závislosti na typu úlohy konfigurovatelný, vetšinou se skládá z vykonání příkazu či dávky na cílovém stroji, nezávisle na použitém operačním systému (abstrakci zajišťuje Java, viz kap. 2). Závěrečná fáze procesu se skládá z úklidu použitých zdrojů, například dočasně vytvořených složek a souborů, dále ze spuštění úloh závislých („downstream“ úloh) na aktuální nebo vykonání pro každou úlohu specifických, nastavitelných akcí. Může se jednat o agregace testovacích výsledků napříč moduly nebo publikování výsledků testů. Po dokončení úlohy je také možné zaslat upozornění e-mailem nebo využít RSS kanál.

### 4.2.3 Vlastnosti úlohy

Kromě struktury úlohy je třeba definovat i její vlastnosti. Úloha je určena svým stavem, konfigurací, pozicí v globálním grafu závislostí, volitelně také počtem podúloh a výsledky testů. Vlastnosti jednotlivé úlohy jsou archivovány v historii, která je indexována číslem exekuce úlohy („build number“). Po spuštění úlohy je vždy načteno a inkrementováno číslo předchozí exekuce nezávisle na stavu jejího dokončení.

Konfigurace úlohy sestává z uživatelského nastavení jednotlivých parametrů úlohy, které vyplývají z její struktury. Je třeba uvést, že je možné uvést jakými fázemi má úloha projít, dále pak lze také konfigurovat jak zacházet výsledkem úlohy či jak reagovat na určité stavy.

### 4.2.4 Podúlohy

Koncept podúloh má smysl jen v souvislosti s úlohou typu „maven 2/3 project“, jejíž implementace je součástí Jenkins. Ostatní základní typy úloh zpravidla nepotřebují další vrstvu abstrakce, také z důvodů univerzálnosti použití. Struktura podúlohy je ve své podstatě podmnožina obecné struktury úlohy v Jenkins, uvedené v 4.2.2, její určení je tak specifičtější a vlastnosti omezené. Oproti úloze je určena jen cíli exekuce (goals), pozicí v grafu závislostí a rodičem, tedy úlohou, která ji vlastní. Konfigurovatelnost je omezena jen na určité parametry, není například možno vybrat si, na kterém uzlu bude podúloha spuštěna, protože kopíruje tuto vlastnost od rodiče. Je také třeba dodat, že pokud chceme dále v této práci podúlohy reprezentující moduly distribuovat skrze gridovou infrastrukturu, je takové chování potřeba doimplementovat.

### 4.2.5 Typy úloh

V Jenkins existují 4 základních typy úloh. Jmenovitě jde o tyto:

- Free-style software project
- Maven2/3 project
- External job
- Multi-configuration project

Každá z typu úloh je definována určitým počtem specifických úkonů a určitou mírou použitelnosti. „Free-style“ projekt slouží k zadání obecného úkolu formou dávkové úlohy pro nejběžnější operační systémy (MS Windows, Unix, MAC OS).

Funkce úlohy typu „maven 2/3 project“ vyplývá již z názvu. Jedná se v podstatě o předchozí typ úlohy, který je optimalizován pro použití s Maven verze 2 nebo 3 jako spodní vrstvou. Tento typ úlohy jako jediný není součástí přímo jádra systém Jenkins, nýbrž je s ním distribuován jako modul plug-in, který k jádru implicitně připojen. Modul plug-in, implementován jako součást této práce, je ve své podstatě jeho přepracovanou verzí. Některé z parametrů programu Maven jsou také prezentovány uživateli pomocí volitelných položek v uživatelském rozhraní. Tuto funkcionalitu lze také chápat jako mapování určitých funkcí mezi systémy Maven a Jenkins či specifikaci, čímž dochází k odlišení typu úlohy Maven od typu úlohy „Free-style“. Prostřednictvím proměnných prostředí lze v Jenkins také předávat úloze proměnnou MAVEN_OPTS, která modifikuje její chování.

Externí úloha slouží jako přístupový bod pro úlohy podobné unixovému programu *cron*, které nejsou přímo pod správou Jenkins, jsou spouštěny mimo toto prostředí a nevyžadují žádnou podporu.

Poslední typ úlohy, „Matrix project“, umožňuje parametrizovat některé vlastnosti úloh. Je možné vytvořit množinu proměnných formou konfigurací, se kterými je pak úloha spouštěna. Každá konfigurace je určena hodnotou pro konfiguraci společné proměnné, která je předána do sestavovacího příkazu.

### 4.2.6 Vzájemné závislosti úloh

Mezi úlohami, jako takovými, mohou v Jenkins existovat relace závislosti. Tyto závislosti pomáhají automatizovat určité činnosti, množiny úloh či práci systému obecně. Jsou tvořeny za běhu Jenkins uživatelem nebo načteny z perzistentního Jenkins úložiště a při dokončení konfigurace uloženy spolu s dalšími metadaty o úloze do interních XML souborů. Existují dva druhy závislostí mezi úlohami a podúlohami:

- Upstream
- Downstream

Prostřednictvím těchto závislostí mohou být spojeny jednotlivé úlohy do užších celků, přesněji lze vytvořit orientovaný graf pořadí vykonávání úloh. Kořenová úloha je pro svoje následníky tzv. upstream závislost, kdežto z jejího pohledu je následník závislost typu downstream. Orientace závislosti je reprezentována směrem pohledu z aktuální úlohy. Zde si lze všimnout určitého podobnosti mezi prací Jenkins a Maven, kdy Maven v podstatě syntaktickou analýzou celé struktury multimodulárního projektu vytváří graf závislostí, které

mohou být mapovány jako jednotlivé podúlohy úlohy typu „maven 2/3 project“. Je také potřeba si dát pozor na symetrické relace závislostí mezi úlohami už při jejich vytváření, Jenkins si snimi při spuštění úlohy nemusí poradit.

### 4.2.7 Mapování Maven modulů na Jenkins úlohy a podúlohy

V Jenkins existuje možnost uživatelské nebo automatizované tvorby podúloh a závislostí mezi nimi. Tyto závislosti pak tvoří tzv. orientovaný graf podúloh $\lambda$, který modeluje do jisté míry Maven graf asociovaného projektu. Mějme konečnou množinu hran grafu podúloh tvořených závislostmi downstream D či upstream U, značeny E, a konečnou množinu jednotlivých uzlů reprezentující podúlohy J. **Graf podúloh** $\lambda$ lze poté definovat jako dvojici

$$\lambda =< J, E >,\ kde\, E \subseteq J \times J, D \subseteq E \wedge U \subseteq E$$

Graf podúloh částečně kopíruje strukturu Maven grafu. **Kořenové podúlohy** (kořeny) takového grafu budou odpovídat vrcholům, které reprezentují kořenové moduly v Maven grafu. Vlastnosti Maven grafu, definované v kap. 2.3.4, budou definovány i pro graf podúloh s tím rozdílem, že místo relace závislosti mezi moduly zaujme relace downstream závislosti mezi podúlohami.

Již v kapitole 2.3.3 byla uvedena jako podmínka předpoklad, že moduly v projektu musí tvořit strukturu typu graf, tedy Maven grafu. Konkrétní existující globální Maven graf (definován v kap. 2.3.4) modelující strukturu některého existujícího projektu označme obecně $G =< U, H >$. Zde bude hranou opět dvouprvková množina H, obsahující dva Maven moduly, reprezentující orientovanou závislost mezi nimi jakéhokoli typu. Vrcholy grafu jsou moduly projektu, jejich množina je označena U. Pokud by se nejednalo o tento typ grafu, například obecný graf, neměla by paralelizace sestavovacího či testovacího procesu pomocí Maven smysl, protože by se mohli vyskytnout cyklické závislosti a nebylo by možné moduly přeložit. Jednoprvkovou množinu obsahující graf G označme

$$\omega = \{G\}$$

Ze znalosti uvedených informací vytvoříme mapovací funkci *map*

$$map(w) = \lambda$$

mapující existujících grafovou strukturu multimodulárního projektu na jedno z možných grafových vyjádření posloupnosti úloh v Jenkins. Uvedené je v podstatě proces tvorby takového orientovaného grafu z Maven grafu, kdy dojde k nahrazení orientované hrany závislostí upstream a přidání závislosti downstream, která má opačný směr než hrana upstream, mezi těmi samými uzly. Tímto dochází k přidání informace. Tento proces se zpravidla zajišťuje přímo Jenkins a je částečně redundantní, protože přidanou informaci přímo nepotřebujeme k průchodu grafem úloh, postačují závislosti jednoho druhu. Univerzalita systému však tento převod vyžaduje, protože veškeré úlohy v systému musejí dodržovat určitý model, jehož součástí je také informace o upstream a downstream závislostech úlohy.

Celý proces je prováděn pomocí Jenkins na straně (pod)úloze přiděleného (master či slave) uzlu. Nejprve je inicializováno a spuštěno rozhraní pro práci s Maven projekty, které analyzuje grafovou strukturu modulů v projektu a poté je přeposlána uzlu master informaci ve formě Maven grafu. Nejprve definujme relaci $<$ takovou, že $a < b \Leftrightarrow$ a má menší úroveň uzlu v Maven grafu než b (úroveň grafu viz kap. 2.3.4). Uzel master nyní vytvoří orientovaný graf úloh $\lambda$ pomocí algoritmu 1.

**Algoritmus 1** Deklarativní zápis mapovací funkce *map*

1: $J, D, U \ = \emptyset$ ▷ na začátku je graf podúloh prázdný
2: $E = D \cup U$
3: $\lambda =< J, E >$
4: $\omega = \{G\} = \{< U, H >\}$
5: **for all** $(a, b) \in H,\ a, b \in U$ **do** ▷ je třeba projít všechny existující hrany
6: $J = J \cup \{createSubJob(a)\}$ ▷ Maven grafu a pro každý vrchol vytvořit podúlohu
7: $J = J \cup \{createSubJob(b)\}$
8: $U = U \ \cup (b, a)\ if\ a < b$ ▷ dále vytvořit upstream
9: $D = D \cup (a, b)\ if\ a < b$ ▷ a downstream závislosti mezi podúlohami
10: **end for** ▷ výsledkem je graf podúloh $\lambda$

Uvedenou operací je převeden Maven projekt na systém podúloh v systému Jenkins. Celý projekt je také zapouzdřen globální úlohou zaobalující všechny podúlohy, reprezentující jednotlivé moduly. Tuto úlohu vytváří uživatel, a k načtení struktury projektu dojde až při jejím prvním spuštění. Pomocí nově vytvořených upstream a downstream závislostí je možno pro každou jednotlivou podúlohu okamžitě zjistit, které podúlohy je také potřeba spouštět a v jakém pořadí. Toto pořadí je určeno polohou podúlohy reprezentující modul projektu v orientovaném grafu podúloh. Pomocí průchodu grafem užitím upstream hran směrem ke kořenu lze zjistit, na kterých podúlohách (modulech) daná podúloha závisí, průchodem downstream závislostí lze naopak zjistit, které podúlohy závisí na aktuální.

### 4.2.8 Úrovně paralelismu

Dále je také třeba vzít již existující možnosti paralelismu systému a zhodnotit, které lze pro účel práce využít. V Jenkins nalezneme následující možnosti paralelismu:

- Úroveň úloh
- Úroveň Maven modulů
- Úroveň Maven procesů

Mluvíme-li o paralelismu v Jenkins, je potřeba také uvést, že na jediném uzlu může zároveň běžet ve stejný čas několik úloh v tom případě, že má uzel dostatek prostředků. Tyto prostředky se nazývají exekutory. Master i každý nově přidaný slave uzel má implicitně jeden exekutor, který může být přiřazen vykonání čekající úlohy. Úloha je tak spuštěna a provedena. První dvě úrovně paralelismu využívají tento systém, který si tedy lze, mimo jiné, představit také jako koncept kritické sekce. Každý uzel má přidělenou a zároveň pevně danou velikost množiny procesů, které může pustit do kritické sekce (procesu mohou být přiděleny prostředky) a postupně je obsluhuje na vyžádání. Tento proces řídí plánovací politika Jenkins (kap. 4.2.9).

Paralelismus na úrovni úloh je možno vidět například při spuštění úlohy, která pomocí závislosti downstream spouští úlohy další, například na jiných uzlech. Pokud je taková úloha dokončena, spustí při ukončení svého procesu tyto úlohy paralelně, tedy pokud nejsou vzájemně závislé a pokud je dostatek volných exekutorů. Tento druh paralelismu je vhodný pro spouštění operací nad vzájemně nezávislými daty, lze jej ale také použít pro operace nad moduly v multimodulárních projektech. Problémem je však nepraktičnost takového použití. Jednotlivé úlohy musejí být vytvořeny pro každý modul projektu separátně uživatelem -

čili probíhá mapování úlohy na modul v poměru 1:1. Toto je nepohodlné, každou úlohu je třeba konfigurovat zvlášť a vytvářet vzájemné závislosti, které už jsou v podstatě vytvořeny na úrovni Maven modulů a zaznamenány v kořenovém POM souboru. Další nevýhodou takového přístupu je samozřejmě značně narůstající celkový počet úloh v seznamu, díky kterému klesá přehlednost rozhraní, potažmo i produktivita systému.

Druhou možností je paralelismus na úrovni modulů. Tento přístup je v současné verzi Jenkins (1.480) uveden jako experimentální, a to pouze při použití úlohy typu Maven 2. Jedná se o automatické rozpoznání struktury modulů multimodulárního projektu a jeho současné naplánování jako úlohy a jejich podúloh běžících právě na stejném uzlu. K paralelizaci dochází tedy na úrovni kterou potřebujeme, ne však přes celou množinu různorodých výpočetních uzlů. Nevýhodou je také velmi omezené použití, například při užití agregací v kořenovém POM souboru, což je velmi častá metoda abstrakce multimodulárních projektů. V tomto případě nemusí být schopna úloha strukturu a závislosti Maven modulů načíst. U této metody lze však vidět snahu o automatizované sestavení, testování a integraci celého projektu na úrovni Maven paralelně. Implementace zásuvného modulu do Jenkins, tvořená prostřednictvím této práce, bude z uvedeného přístupu vycházet.

Poslední z možností sestává z užití experimentální funkcionality obsažené v Maven 3, která dovoluje spouštět jednotlivé procesy nad moduly paralelně na úrovni vláken. Tento přístup nechává na instanci Maven jak výpočet grafu závislostí modulů, tak i samotné spuštění uživatelského příkazu nad nimi pomocí plugin modulu Reactor. Tímto je možno dosáhnout zrychlení vykonání práce nad moduly o 20-50% [19]. Příkladem spuštění Maven procesu v několika vláknech může být následující příkaz, který vykoná cíle clean a install ve 4 vláknech:

```
mvn -T 4 clean install
```

Samozřejmě, z pohledu gridové architektury, nemůže být v tomto případě o distribuovaném výpočtu řeč. Jedná se opět o práci nad moduly v rámci jednoho uzlu. Nevýhodou je také nemožnost využít některé plugin moduly Maven, které nejsou programovány s vícevláknovou podporou, tzn. nejsou anotovány *@threadSafe* anotací. Z nejdůležitějších jsou nekompatibilní hlavně plugin moduly zabývající se tvorbou archivů ze zdrojových souborů, ale také některé součásti Surefire zásuvného modulu, který se používá při testování. Celkově tato možnost nepřináší nejvyšší možný nárůst výkonu a slouží jen jako simulace konkurentního chování, což může sloužit jako testovací prvek pro vícevláknové aplikace.

### 4.2.9 Interní plánovací politika Jenkins

Správu úloh v Jenkins zajišťuje plánovač, který je implementován jako fronta FIFO. Jeho úkolem je vhodně distribuovat úlohy přes množinu všech výpočetních prostředků. Globální politika plánování je zaměřena na minimalizaci přetěžování sítě při komunikaci mezi uzly master a slave. Celý proces plánování lze popsat následujícími pravidly[9]:

1. Je-li úloha přiřazena k určitému uzlu explicitně, je takový uzel použit
2. Pokud již byla úloha úspěšně dokončena na některém uzlu, je tamtéž spuštěna opět
3. Úlohy časově náročnější jsou distribuovány na slave uzly

Autor Jenkins udává, že závislost vytížení sítového připojení mezi master a slave uzly je logaritmická k délce trvání úlohy. Pokud máme, například, ve frontě dvě úlohy a náhle jsou dostupné prostředky pro jejich spuštění, je kratší úloha naplánována na uzel master, delší

na některý ze slave uzlů. Vzhledem k tomu, že mezi oběma typy uzlů je třeba komunikace za běhu úlohy (výpis aktuálního stavu, monitoring procesu), jejíž míra není lineárně závislá na její délce, je užitím bodu 3. optimalizováno vytížení vnitřní sítě.

## 4.3 Závěr

Je potřeba brát v potaz, že pokud chceme, aby integrační software spolupracoval s distribuovanou paralelizací testovací a sestavovací fáze na gridové architektuře, je třeba tuto podporu doimplementovat, protože Jenkins takovou funkcionalitu nativně nepodporuje. Pouze master/slave mód v kombinaci s podporou paralelizace a Maven frameworkem se k takové funkcionalitě blíží. Vše je však prováděno na vybraném uzlu a ne distribuovaně, podpora paralelizace je navíc stále experimentální. V této kapitole však byly uvedeny nejdůležitější body, které mohou sloužit jako odrazový můstek pro následující návrh architektury či implementaci.

# Kapitola 5

# Návrh systému

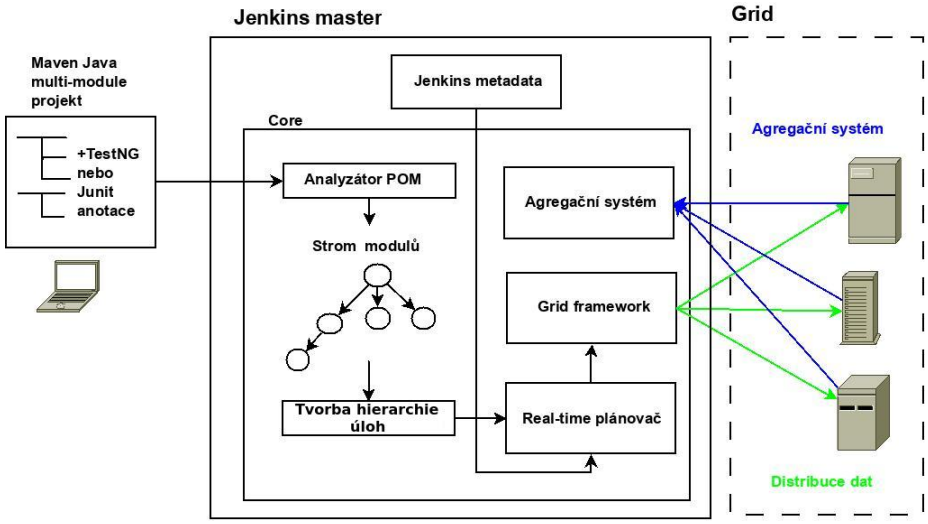
V předchozích kapitolách jsme se zaměřili na rozbor problematických součástí paralelní kompilace v jazyce Java, dále byly popsány nejznámější gridové implementace a nástroj pro postupnou integraci Jenkins. Byly rozebrány klíčové prvky jazyka Java pro paralelní sestavovaní či testování s využitím Jenkins. Nyní je třeba ze získaných znalostí vybrat vhodný přístup k řešení úkolu tvorby paralelního testovacího systému formou plugin modulu systému Jenkins. Je potřeba také připomenout, že paralelní proces testování v sobě bude zahrnovat také paralelní proces kompilace či sestavení, který je tedy také jádrem problému.

## 5.1 Architektura

Návrh architektury je na obrázku 5.1. Vstupem celého procesu je multimodulární projekt využívající framework Maven, sestávající ze zdrojových kódů, které je potřeba zkompilovat a otestovat. V projektu jsou obsaženy také testovací třídy pro frameworky Junit či TestNG. Projekt takového typu nebo odkaz na něj bude vstupem vytváření nové úlohy v uživatelském rozhraní Jenkins. Dále je třeba provést konfiguraci úlohy, kde musí být pro úlohu vyhrazena určitá skupina nebo skupiny uzlů, na kterých může běžet. Poté bude možno přidat specifická metadata a omezení (na obrázku „Jenkins metadata“).

Maven projekt definuje závislosti kořenového modulu a submodulů v souborech POM, které tvoří hierarchickou grafovou strukturu. Tuto strukturu bude zpracovávat jednotka „Analyzátor POM“, která vytvoří grafovou reprezentaci závislostí modulů (globální graf modulů), a tím zajistí, že nedojde k sestavování či testování modulu bez jeho potřebných prerekvizit. Toto je v podstatě také práce Maven Reactor Plugin, která je nyní vykonávána pomocí Jenkins. Projekty samozřejmě musí splňovat požadavky uvedené v kapitole 2.3.3, aby byl proces paralelizovatelný.

Nechť je základní jednotkou pro testování tedy jeden Maven modul (volba vysvětlena v následující kapitole 5.1.1) reprezentován podúlohou, jehož metadata mohou být uložena přímo v Jenkins (s takovými metadaty již umí pracovat). Součástí analýzy struktury modulů může být i rozšiřující jednotka, která provede zpracování přídavných metadat jednotlivých testů uvnitř modulu ve formě anotací nebo dodatečných souborů v případě, že bude zavedena větší granularita možností testování (5.1.1). Tehdy mohou být tato metadata přidána jako další restrikce nebo selekce pro testování (například informace o náročnosti testu, vyžadování běhu na uzlech speciálního typu). V tomto případě by musela jednotka implementovat nejen syntaktickou analýzu na úrovni POM, ale také procházet jednotlivé moduly, hledat přídavné anotace a doplňovat ke grafu modulů informace o testech obsa-




Obrázek 5.1: Návrh architektury systému

žených uvnitř, což by mohlo pro master uzel velmi výpočetně náročné a vést naopak ke zpomalení procesu. Taková funkcionalita je brána jako rozšíření a do prvotní implementace nebude zahrnuta.

Výstup předchozích jednotek se dostává na vstup plánovače. Tato jednotka může být spojena s gridovou vrstvou, resp. mohla by využívat plánovač gridového frameworku místo interního plánovače Jenkins, to je ale nad rámec potřeb vytvářeného plugin modulu.

Plánovač je důležitou částí celého systému. Přijímá Maven graf spolu s metadaty o jejich testování transformovaný na graf závislých podúloh (4.2.7) pomocí jednotky tvorby hierarchie úloh. Plánovač prochází tímto grafem od kořene k listům a jednotlivé (pod)úlohy odesílá ve spolupráci s gridovou vrstvou na vhodný uzel v závislosti na metadatech k podúloze asociovaného modulu. Plánovací jednotka tedy řídí distribuci dat gridem a zadává práci jednotlivým uzlům.

Nyní je řada na pracovních uzlech zapojených v gridu, aby vykonaly přidělenou práci ve formě spuštění Maven s uživatelsky zadanými vstupy. Tyto uzly musí mít přístup ke gridové vrstvě, s jejíž pomocí budou získávat zdrojové soubory určené pro jim přiřazenou část úkolu. Budou ale také pomocí této vrstvy ukládat data, která reprezentují výsledek jejich úsilí. Toto však není jejich jediný úkol. Jenkins vyžaduje také informace o běhu procesu a výsledcích úlohy a proces běžící na uzlu musí tyto informace poskytovat, aby jej bylo možné kontrolovat a informovat uživatele. Takové informace sdružuje agregační systém, který také prezentuje výsledky úlohy jako celku uživateli.

### 5.1.1 Teoretické úrovně granularity testování

V předchozí kapitole jsme udali jeden modul jako jednotku granularity testování. Teoreticky by však také šlo úroveň granularity snížit na jednotku menší než modul nebo naopak zvýšit. Testovací sady ve struktuře modulu mohou být dále rozděleny v závislosti na zvoleném testovacím frameworku (kap. 2.5.1) nebo může například existovat externí informace, která mapuje testy nebo testovací sady na podmnožinu uzlů, které jsou v Jenkins přiřazeny právě gridovým úlohám. Tímto by mohlo dojít ke zvýšení flexibility testování, ale také ke zvýšení zatížení jednotky „Analyzátor POM“, plánovače i gridové vrstvy. V navrženém systému může tedy existovat několik úrovní granularity:

- Celý projekt
- Množina modulů
- Právě 1 modul
- Množina (skupina) testů
- Právě 1 test

### Celý projekt

První, naivní, způsob nezavádí žádnou granularitu, distribuce celého projektu probíhá na všechny dostupné uzly infrastruktury. Projekt se může nejprve zkompilovat na některém (např. velmi výkonném) uzlu a poté se distribuovat na každý uzel zapojený do gridu. Nad takovým prostředím je pak možno testovat paralelně jednotlivé moduly bez problému s kompilačními závislostmi (závislosti třetích stran nejsou a nebudou dále uvažovány) a většina režie leží na komunikaci s gridovém frameworkem. Nevýhodou je tedy distribuce celého projektu na celý počet přidělených uzlů a redundance dat. Přístup lze kombinovat s následující úrovní granularity, čímž redundance dat klesá.

### Množina modulů

Volbou této úrovně granularity se vytvoří na základě uživatelské informace disjunktní množiny modulů, se kterými bude formou podúloh pracovat přiřazený uzel. Tohoto přístupu je však možno dosáhnout již zavedením granularity na úrovni právě jednoho modulu a asociovat manuálně podúlohy z jedné množiny s určitým uzlem. Tento proces je bez ztráty flexibility, naopak ji zvyšuje.

Druhou možností na této úrovni by bylo asociovat právě jednu podúlohu s několika Maven moduly. U tohoto způsobu ale lineárně klesá flexibilita práce s rostoucím počtem modulů v množině. Pokud bude třeba například otestovat jeden modul z množiny, ve kterém jsou dva další, musíme pracovat s takovou množinou jako celkem, což zavádí provádění redundantních úkonů a může být nepřijatelné. Ze strany Maven není s tímto přístupem problém, Maven Reactor Plugin si dokáže, pokud jsou na sobě moduly v množině závislé, s takovou strukturou poradit. Plánování by pak probíhalo tak, že na jednotlivé uzly by byl distribuován podgraf reprezentující takto závislé moduly. Toto je částečně výhodné, protože po sestavení je možno otestovat všechny moduly v podgrafu naráz, navíc závislosti pro sestavení podgrafu jsou načteny do repozitáře Maven na uzlu jen jedenkrát.

Pokud budou moduly ve skupině na sobě nezávislé nebo nebudou definovány ve společném POM, je nutné jejich obsluhu spouštět jako více nezávislých Maven procesů, což by vedlo k přidávání zbytečné funkcionality a zmatkům.

### Jeden a právě jeden modul

Tato úroven granularity je podporována v Jenkins implicitně pomocí Maven 2 Project Plugin. Jeden modul je mapován na jednu podúlohu (viz kap. 4.2.7). Tato úroveň granularity práce je výhodná, protože Jenkins disponuje jak přímou podporou, tak rozhraními spolupracujícími s Maven na úrovni modulů. Jenkins je schopný pomocí příkazu spustit na cílovém uzlu Maven proces, který načte zdrojová data, spustí požadované cíle, bude monitorovat jeho průběh a prezentovat výsledky.

Plánování by sestávalo z procházení grafu podúloh po jednom od kořene a distribuce podúloh na uzly po jednom dle metadat. Na uzlu je poté spuštěna kompilace a sestavení modulu, následně pak otestování. Je důležité poznamenat, že tento přístup musí zaručovat, že na uzlu, na kterém se provádí kompilace a testování asociovaného modulu, musejí být také přítomny předchozí závislosti ve směru ke kořenu Maven grafu, což může náročné na využití gridového repozitáře při stahování zdrojů. Pro multimodulární projekty s vysokou hloubkou Maven grafu to pro moduly na nejvyšší úrovni může znamenat opakovanou potřebu přítomnosti velkého množství závislostí.

#### Množina (skupina) testů nebo samostatné testy

Skupinou testů může být, v rámci frameworků Junit a TestNG, například kolekce testů „Suite“ v rámci modulu, kterou se zabývala kapitola 2.5.1. V případě že se jedná o balík testů Junit, je potřeba například doplnit do kódu nějakou informaci prostřednictvím další anotace k anotacím @RunWith a @Suite. Tento přístup by ale nutil k dalším zásahům do kódu a navíc k inkluzi knihovny dalších anotací, což může být nežádoucí. Druhou možností je použít externí soubor, například umístěný ve složce s testy modulu, který bude přídavná metadata obsahovat. U frameworku TestNG je možné použít oba přístupy, ale vzhledem k tomu, že TestNG již externí soubor XML podporuje, šlo by jej teoreticky využít. Vzhledem k tomu, že se jedná o XML soubor, mohlo by předefinováním jeho DTD souboru dojít k přidání libovolných metadat.

V případě volby granularity na úrovni skupin testů by jednotka analýzy POM musela doplňovat funkcionalitu pro analýzu zdrojových souborů, je otázkou, zda by ale takový zásah přinesl dostatečné výhody. Musel by se také obcházet framework Maven. Je potřeba při vývoji projektu zhodnotit, zda existují požadavky na tak vysokou úroveň granularity testování a zda jsou opravdu testy v rámci jedné jednotky tak heterogenní, že se vyplatí testování těchto jednotek paralelizovat i za cenu větších nároků na jednotku analýzy POM. Další otázkou je také, co kdyby se nejednalo o pouze o skupiny testů v rámci modulu, ale přes skupinu modulů nebo dokonce cely projekt.

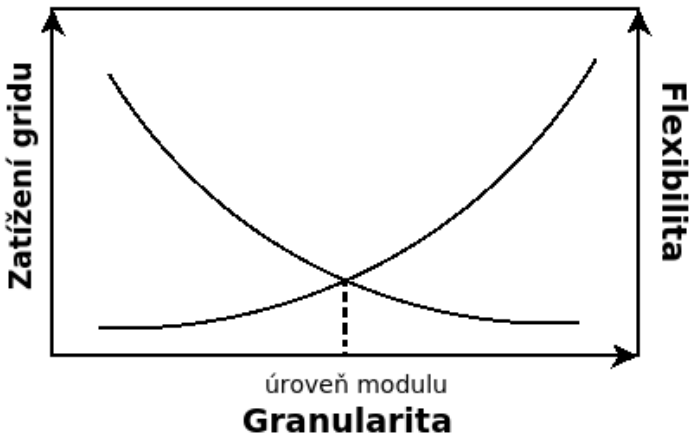
Je také potřeba dodat, že gridová vrstva by musela pracovat s velmi malými soubory, což není její produkční účel. Naopak cíl práce gridových frameworků je dnes zaměřen spíše na velké soubory, jako úložiště malých souborů by bylo potřeba využít služeb některého ze specializovaných distribuovaných nebo sdílených souborových systémů.

#### Závěr a volba granularity

Závěrem sekce lze říci, že nejvýhodnější je zvolit granularitu na úrovni jednoho Maven modulu jako testovacího prvku. Tento přístup má minimum nevýhod v případě, že není známa struktura projektu a snažíme se zachovat obecnost. Pokud by byla struktura dopředu známa, dávalo by smysl zvýšení granularity směrem od modulů k testům, tuto funkcionalitu je v tomto případě možno implementovat jako rozšíření. Naopak zavedení menší až žádné granularity by snížilo flexibilitu a zvýšilo náročnost na výkon gridové vrstvy (obrázek 5.2). Práce bude dále od tohoto bodu vycházet z takto zavedené granularity.

### 5.1.2 Plánování a analýza složitosti

Máme-li vytvořen graf podúloh (4.2.7), jsou přiděleny uzly pro gridové úlohy a jsou volné prostředky, je možné začít rozesílat podúlohy na uzly. Pro jednoduchost vezmeme případ, že nejsou zadány žádné restrikce pro jednotlivé moduly a že každý uzel má přidělen jeden




Obrázek 5.2: Volba granularity testování

exekutor, který může spouštět (pod)úlohy. Plánovač Jenkins (4.2.9) bude tedy pracovat dle algoritmu 2. Jednotlivé moduly reprezentují podúlohy v Jenkins.

Není třeba implementovat plánovač nový, protože interní plánovač Jenkins umí pracovat s paralelními úlohami, ne však podúlohami. Koncept tedy bude třeba rozšířit i na tento případ. Cílem je také se držet toho, že vyvíjený systém bude implementován jako plugin modul do Jenkins, tedy není možno plánovač reimplementovat, ale rozšířit, omezit či optimalizovat (pomocí 6.4.1).

Algoritmus je samozřejmě o něco složitější, protože nebere v úvahu uživatelská omezení, kdy uživatel přiřazuje některým uzlům některé moduly. Pro analýzu zavedeme pojem časové složitosti t(n) jako funkce počtu průchodů algoritmu bodem 13. Dále zavedeme pojem prostorové složitosti p(n) jako počet uzlů staticky vyhrazených pro řešení úlohy, vždy se předpokládá $n \geq 1$. Cena algoritmus je poté veličina

$$c(n) = t(n).p(n) \tag{5.1}$$

a udává zda je algoritmus optimální. Pokud ano, je $c(n)$ algoritmu stejná jako $t(n)$ řešení stejné úlohy na jednom uzlu.

### Nejvhodnější struktura vstupních dat

Mějme multimodulární projekt, jehož hierarchii závislostí modeluje takový Maven graf o jedné komponentě, kde úroveň všech vrcholů kromě kořenových je jedna a nekořenové moduly jsou na sobě nezávislé, tedy strom. Dále předpokládejme, že máme k dispozici takový počet volných uzlů $p$, jako modulů v grafu $n$, tedy platí podmínka

$$p = n \tag{5.2}$$

Pokud takový graf namapujeme na graf podúloh v Jenkins a zapouzdříme jej úlohou, platilo by pro dobu běhu úlohy $t_{goal}$ reprezentující takový projekt, spuštěný v Jenkins pomocí klasického Maven 2 Project Plugin, který nepracuje distribuovaně, následující

$$t_{goal} = t_{job} + t_r + \sum_{m=1}^{n-1} t_m \tag{5.3}$$

**Algoritmus 2** Algoritmus plánování, spuštění a běhu (pod)úloh v Jenkins

```
1: result = run(master, job)      ▷ vezmi rodičovskou úlohu a spusť ji na přiděleném uzlu
2: if result = SUCCESS then                                          ▷ pokud skončila úspěšně
3:     subjob = kořenová úloha                                       ▷ začni s kořenovým modulem
4:     schedule(subjob)              ▷ naplánuj podúlohu reprezentující kořenový modul
5: end if

6: function SCHEDULE(subJobT jobs)                                   ▷ funkce beží jako proces
7:     while not node = getFreeGridNode() do                         ▷ získej libovolný z volných uzlů
8:         waitInQueue()                                             ▷ pokud nelze, čekej ve frontě
9:     end while
10:    result = run(node, subjob)                                    ▷ spusť podúlohu
11:    if result = SUCCESS then                                      ▷ pokud skončila úspěšně
12:        for all subjob ∈ downstream(jobs) do
13:            schedule(subjob)                                      ▷ naplánuj všechny závislé podúlohy
14:        end for                          ▷ vložením do interní FIFO fronty čekajících úloh
15:    end if
16: end function
```

kde $t_m$ je čas běhu Maven nad modulem $m$ se zadaným cílem $goal$, $n-1$ je počet modulů v projektu, $t_{job}$ reprezentuje dobu spuštění a zpracování rodičovské úlohy jako celku na přiděleném uzlu (algoritmus 2, body 1-5) a veškeré další režie úlohy i Maven vrstvy. Proměnná $t_r$ reprezentuje dobu běhu kořenové podúlohy, na která bodem 13 algoritmu prochází jako první a pouští ostatní podúlohy.

Naopak v režimu distribuovaném, který bude vyvíjen v implementaci, bude doba běhu $tp_{goal}$ jednotlivé podúlohy v rámci vyvíjeného plugin modulu o něco složitější

$$t_{pmgoal} = t_{fetch} + t_m + t_{store} \tag{5.4}$$

$$t_{root} = t_{fetch} + t_{root} + t_{store} \tag{5.5}$$

kde $t_{root}$ je doba běhu kořenové podúlohy v neparalelním režimu a protože čas distribuce dat sítí je nenulový platí také

$$t_{pmgoal} > t_m \tag{5.6}$$

$t_m$ je doba běhu podúlohy v nedistribuovaném režimu. Doby $t_{fetch}$ a $t_{store}$ reprezentují režie spojené s načtením a uložením dat gridové vrstvy. V případě běhu celé úlohy pomocí navrhovaného plugin modulu do Jenkins, který pracuje distribuovaně, bude doba běhu úlohy ideálně:

$$t_{goal} = t_{job} + t_{root} + \max(t_{p1goal}, t_{p2goal}, t_{p3goal}, ..., t_{pngoal}) \tag{5.7}$$

Předpokládejme, že pro rozsáhlé projekty bude platit

$$t_{fetch} + t_{store} \ll t_m \tag{5.8}$$

pak lze psát

$$t_{goal} = t_{job} + t_{root} + \max(t_1, t_2, t_3, ..., t_n) \tag{5.9}$$

čímž se výrazně redukuje doba dosažení cíle (například testování) projektu. V tomto bodě dochází k paralelizaci běhu a zrychlení, je nutno však zopakovat předpoklad 5.8 a také to, že v tomto případě se jedná o strom, který se v praxi téměř nevyskytuje. Tyto závěry jsou vyvozeny tedy pro teoreticky nejvhodnější případ struktury vstupního projektu.

Pokud dále budeme navíc předpokládat, že běh všech $n$ podúloh trvá stejně, a to čas $t$, můžeme psát vztah pro zrychlení A úlohy, které se vypočte jako podíl sériového a paralelního běhu úlohy

$$A = \frac{t_{job} + t_{root} + \sum_{m=1}^{n} t_m}{t_{job} + t_{root} + \max(t_1, t_2, t_3, ..., t_n)} \tag{5.10}$$

$$= \frac{t_{job} + t_{root} + tn}{t_{job} + t_{root} + t} \tag{5.11}$$

$$= n \tag{5.12}$$

čímž by došlo například ke zrychlení otestování celého projektu n-krát, ovšem případě, že je před startem úlohy $n$ volných uzlů. Pokud tedy proběhne naplánování všech $n$ podúloh najednou na $n$ volných uzlů v jednom kroku, je časová složitost algoritmu konstantní a prostorová lineární. Cena takového paralelního zpracování je pak $c(n) = 1.n$ přitom časová složitost sekvenčního řešení je také lineární. V tomto případě je algoritmus optimální.

Pokud ovšem zavedene počet procesorů

$$p < n \tag{5.13}$$

bude časová složitost opět lineární s konstatní prostorou složitostí,

$$t(n) = 1 + \left\lceil \frac{n}{p} \right\rceil \tag{5.14}$$

ale počet kroků naroste, protože v bodě 8 algoritmu 2 se čeká na uvolnění uzlů.

### Nevhodná struktura vstupních dat

Mějme Maven graf o jedné komponentě úrovně $v$ s $n$ uzly, kde každý uzel je rozdílné úrovně z itervalu $< 1, v >$. Takový graf opět tvoří strom, který má v každé úrovni právě jeden uzel. Již z takového zadání je patrné, že projekt bude velmi nevhodný k paralelizaci.

V tomto případě nemá smysl volit počet uzlů $p$ vyšší než jedna, protože využit by byl v čase jen jeden. Tedy volíme $p = 1$. V tomto případě ale nedojde k paralelizaci vůbec, naopak dojde ke zpomalení celého procesu. Vzhledem k době, kterou trvá úloha sekvenčně (5.3), nyní bude proces delší o $n$ násobek proměnných $(t_{fetch} + t_{store})$ (5.4).

$$t_{goal} = t_{job} + t_{root} + \sum_{m=1}^{n-1} (t_{fetch} + t_m + t_{store}) \tag{5.15}$$

Vztah 5.15 tedy říká, že místo urychlení procesu dojde ke zpomalení vlivem neustálé interakce s gridovou vrstvou oproti běhu v neparalelním režimu. Naštěstí v praxi se tento typ projektové struktury opět téměř nevyskytuje.

Pokud bychom ideálně zavedli i tady předpoklad 5.8, dosáhneme maximálně stejného času, jako v sekvenčním provedení 5.3. Časová složitost algoritmu $t(n)$ bude opět lineární, cena $c(n) = t(n)$ tedy opět optimální.

### Obecná struktura vstupních dat

Vstupní obecný globální Maven graf může být libovolné úrovně, sestávat z několika komponent, může mít různě složitě organizované závislosti mezi uzly a zpracování uzlu může trvat rozdílně dlouhou dobu. Chceme-li zjistit dobu běhu úlohy, která vznikla převodem z obecného Maven grafu, vyskytne se řada problémů. Vzhledem k tomu, že plánovač Jenkins je dynamický, budeme předpokládat že neexistují žádné informace o délce budoucího běhu podúloh.

Vzhledem k řádku 12 algoritmu 2 není možné s jistotou určit, ve kterém čase bude která podúloha provedena za předpokladu, že v době tohoto úkonu nebude dostatek volných uzlů. Vložení několika podúloh jako downstream závislostí do fronty na úrovni řádku 13 je náhodné s důrazem na přístup FIFO. Nevíme však, která z podúloh plánovaných ve stejném čase bude ve frontě na které pozici, navíc neznáme ani čas běhu podúloh, takže průchod grafem nelze odhadovat. Fronta také už může obsahovat další podúlohy, což přináší další problémy.

Zjistit tedy přesný čas běhu celé úlohy i v případě předpokladu, že všechny podúlohy poběží stejnou dobu, nelze. Lze však v tomto případě určit horní a dolní mez dobu běhu úlohy. V praxi ale tento předpoklad u obecných struktur využít nelze.

Problém tedy vzniká při nedostatku zdrojů. Vzhledem k tomu, že neznáme budoucí trvání běhu jednotlivých podúloh, nelze tedy jednoznačně určit ani čas běhu projektu. Pokud ale zavedeme předpoklad, že bude na počátku k dispozici pro úlohu o $m$ podúlohách počet uzlů $p$ takový, že $p = m$. Pak nebude záležet na pořadí vložení podúloh do fronty, protože v kterémkoli bodu výpočtu bude vždy dostatek uzlů pro spuštění naplánované podúlohy. Každá struktura projektu bude poté vhodná pro výpočet, omezení na čas běhu úlohy bude plynout jen ze struktury samotné. Luxus takového předpokladu ale v praxi opět nemáme, proto je třeba pracovat hlavně s podmínkou $p < m$, se kterou čas výpočtu ani nejvhodnější strukturu obecně stanovit díky zmíněné neurčitosti s jistotou nelze.

### Optimalizace algoritmu plánování pomocí heuristiky

Vzhledem k tomu, že fronta (pod)úloh je v Jenkins teoreticky nekonečná, lze ale alespoň optimalizovat dynamickou povahu plánovače úloh zavedením heuristiky. Tato heuristika bude udržovat frontu v maximální možné velikosti v čase, protože můžeme předpokládat, že výhodnější je naplánovat podúlohu mající podgraf vyšší úrovně. Pokud přijdeme do vyšší úrovně grafu dříve, tj. budeme plánovat nejdříve nejdelší cesty, bude pak čas běhu dalších úloh záviset pouze na době jejich zpracování a závislostech mezi nimi, dále bude existovat minimální možné zpoždění způsobené vyhladověním čekající podúlohy. Tento postup je výhodný i z pohledu faktu, že je třeba zavést také restrikce pro některé podúlohy (například testování modulu na specifickém uzlu). Pokud bude ve frontě větší množství podúloh je větší šance, že se najde některá, která nebude vyžadovat aktuálně obsazené zdroje.

Mějme dvě podúlohy A a B vloženy do fronty (naplánovány) ve stejný čas. Je však dostupný jen jeden uzel pro vykonání některé z nich. Potom je možné, že budou prostředky přiděleny například podúloze B, která je časově náročná a nemá podgraf downstream závislých podúloh potřebných pro dokončení celé úlohy, kdežto podúloha A, která je krátká a takový podgraf vlastní bude čekat ve frontě. Tímto může dojít ke zpomalení celého procesu běhu úlohy, protože kdyby byla nejdříve naplánována a dokončena úloha A, mohou být v následujícím čase volné prostředky pro souběžný běh některé z dále naplánovaných downstream závislostí úlohy A i pro běh podúlohy B.

Je tedy nejvhodnější upravit algoritmus plánovače tak, aby plánoval co nejdříve podúlohy, jejichž podgraf obsahuje takové uzly, které mají nejvyšší úroveň. Pokud budeme disponovat informací o maximální úrovni v podrafu podúloh dané úlohy a tuto hodnotu přiřadíme podúloze, můžeme s její pomocí v bodě 13 algoritmu 2 rozhodnout, kterou podúlohu naplánovat dříve. Funkci vracející tento údaj budeme říkat heuristika. V případě, že bude heuristika pro několik podúloh udávat stejnou hodnotu, opět vybereme libovolnou z podúloh.

**Algoritmus 3** Úprava algoritmu 2 zavedením heuristiky

```
if result = SUCCESS then                    ▷ seřaď podúlohy dle
    sortedJobs = sort(downstream(jobs))     ▷ úrovně jejich podgrafu od největší
    for subjob ∈ downstream(jobs) do        ▷ naplánuj takto seřazené úlohy
        schedule(subjob)
    end for
end if
```

V algoritmu 3 je předřazena funkce sort(), která zapouzdřuje heuristiku a která vhodným algoritmem řazení s jejím využitím vytvoří seřazenou posloupnost downstream úloh, které jsou poté plánovány ke spuštění.

Plánovač by také jako rozšíření mohl pracovat „realtime“ s prostředky na úrovni uzlů, kdy od dalšího přídavného software může na vyžádání dostávat dynamicky se měnící množinu dostupných uzlů, které jsou například připravovány, instalovány a konfigurovány „on-the-fly“. Takto by se počet použitelných uzlů mohl adaptovat na vyžadovaný počet podúloh dynamicky.

### 5.1.3 Gridová vrstva

V kapitole 3 jsme rozebrali v současné době nejznámější implementace gridové architektury, nyní je třeba jednu z nich vybrat pro použití ve vyvíjené aplikaci. V předchozí kapitole jsme optimalizovali dynamický plánovač Jenkins, není tedy třeba vkládat do navrhovaného systému plánovač gridové vrstvy. Bude vhodné tedy volit takový framework, u kterého lze používat také jeho součásti samostatně. Frameworky GridGain a Infinispan pracují na úrovni datového gridu, který ukládá data paměti. Toto je nevhodné pro spolupráci s Maven, který vyžaduje přítomnost souborů v lokálním datovém úložišti. Při takové volbě bychom pak museli počítat s neustálým ukládáním dat z paměti na disk a naopak, pro což frameworky produkčně použít nelze. Frameworky nepracují totiž s daty na úrovni souborů, ale se s daty v surovém formátu. Infinispan verze 4 však disponuje implementací gridového souborového systému, ten je ale v současné době zatím v experimentálním stádiu. Co se týče Globus frameworku, toto řešení je pro potřeby vyvíjeného systému značně komplexní a nepraktické. Jako úložiště by musel být použít systém GridFTP, který však souborový systém nenahrazuje a byl by náročný jak na instalaci, tak na obsluhu.

#### HDFS

Zvolíme tedy jako zástupce gridové vrstvy součást framework Apache Hadoop, a to gridové úložiště HDFS. Toto úložiště je možno spustit samostatně a zajistí běh sdíleného gridového souborového systému napříč všemi uzly. Tímto způsobem vytváří v podstatě další lokální repozitář Maven. V tomto úložišti budou uložena veškerá dílčí data ve formě modulů či

archivů na souborové úrovni. Opět je potřeba kopírovat data do a z lokálního systému, to však bude rychlejší díky přímému přístupu k souborové struktuře HDFS.

Další výhodou je replikace souborů, která může zajišťovat také zotavení z výpadků uzlů či chyb výpočtu. Pro velké soustavy uzlů bude také zodpovědná za lineární škálovatelnost gridu, což by například datová vrstva realizovaná kopírovaním souborů mezi uzly zajistit nemohla. Problémem je však nutnost zadání počtu souborů, na které se má každý soubor replikovat, před startem HDFS. Vzhledem k tomu, že Jenkins musí mít ve své právě při startu alespoň jeden uzel (master), je nutno replikaci $r$ nastavit nejprve $r = 1$ a po přiřazení dalších uzlů do správy Jenkins HDFS restartovat. Replikaci lze měnit i dynamicky za běhu programu, tím je ale narušena lineární škálovatelnost.

Další nevýhodou je na první pohled i to, že souborový systém HDFS je určen produkčně pro ukládání velkých souborů. Implicitní velikost bloku je 64MB a neměnná velikost metadat o každém souboru je 120B. Tímto vzniká problém s ukládáním velkého množství malých zdrojových souborů, který lze ale vyřešit snížením hodnoty velikosti bloku. Vliv tohoto problému lze také snížit archivováním veškerých zdrojových souborů, například pomocí formátu TAR bez komprese. Dalšími soubory, se kterými bude úložiště pracovat, už jsou v podstatě jen Java archivy, které vzniknou spuštěním Maven nad soubory zdrojovými. Těchto souborů se ovšem uvedený problém netýká.

### Nároky na úložiště metadat

Po startu jádra Jenkins je také na master uzlu startován HDFS Namenode a Datanode. Namenode ukládá názvy souborů a ukládá informace o blocích, které je tvoří, Datanode se stará o uložení bloků samotných. Jelikož je možné, že budeme do úložiště ukládat velké množství dat a velikost metadat o souboru je neměnných 120B, je třeba vypočítat jak porostou nároky na velikost úložiště metadat se změnou velikostí bloku. Dále budeme předpokládat, že není zavedena žádná replikace.

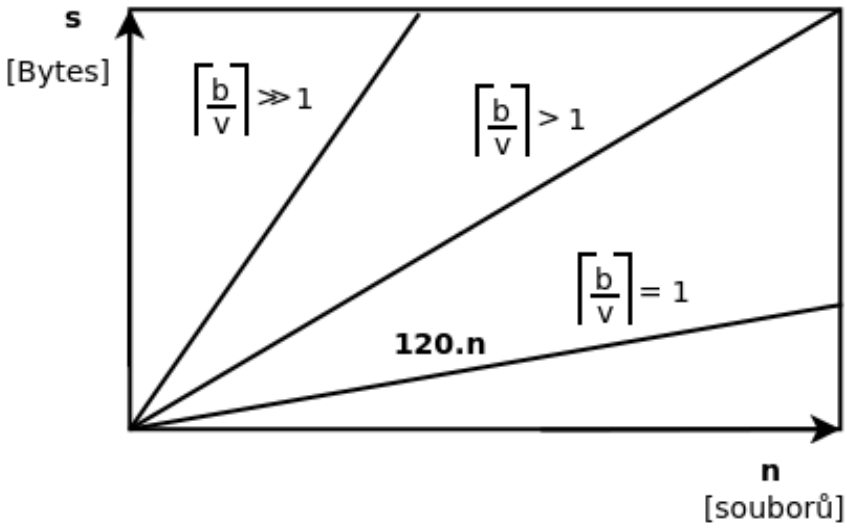
Nejprve je třeba zjistit průměrnou velikost vstupních dat ukládaných $v$. Zavedeme-li velikost bloku $b$, pak jeden uložený soubor $f$ bude zabírat průměrně

$$f = \left\lceil \frac{b}{v} \right\rceil \tag{5.16}$$

bloků. Bude-li v souborovém systému uloženo $n$ souborů, a každý blok vyžaduje na Namenode uzlu 120B dat, bude počet bytů pro všechna metadata souborového systému $s$

$$s = 120.n. \left\lceil \frac{b}{v} \right\rceil \tag{5.17}$$

S tímto problémem je třeba počítat při navrhování velikosti bloku souborového systému a navrhovat velikost tak, aby vzhledem k velikosti vstupních dat příliš nezatěžoval úložiště na master uzlu, to znamená snažil se držet poměru velikosti souboru ku velikosti bloku co nejblíže jedné (obrázek 5.3).




Obrázek 5.3: Velikost metadat v gridovém úložišti

# Kapitola 6

# Implementace plugin modulu pro Jenkins CI

Kapitola návrh architektury byla obsahem implementovaného plugin modulu, kapitola implementace se bude zabývat naopak jeho formou.

## 6.1 Úvod

Webová aplikace Jenkins je implementována v jazyce Java užitím technologií Java EE, webový server běží v servlet kontejneru Winstone, je možnost pustit jej i v Jetty. Autorem a stálým vývojářem je Kohsuke Kawaguchi. Pro návrh grafického webového uživatelského rozhraní užívá Apache Groovy a Apache Jelly. Sestavení, testování a generování kódu probíhá pomocí Apache Maven. Jenkins je distribuován v archivu WAR, je ale také možné stáhnou v některém z nativním balíčků pro různé nejznámější distribuce (Ubuntu, Fedora, FreeBSD) či MS Windows. Z pohledu implementace je Jenkins rozmanitě modifikovatelný software, v současné době existuje kolem 400 plugin modulů upravujících, rozšiřujících nebo doplňujících funkcionalitu. Plugin moduly jsou centralizovány na adrese https://wiki.jenkins-ci.org/display/JENKINS/Plugins, správa jejich zdrojových kódů je delegována na různé repozitáře, nejčastěji však na repozitáře užívající verzovací systém GIT hostovány na https://github.com.

Nejdůležitější části jádra systému jsou přehledně dokumentovány, slabší je situace u vývoje plugin modulů, u nichž často dokumentace neexistuje a je třeba se k sémantice kódu dopídit pomocí debuggingu nebo krokování spuštěného kódu. Existuje také dokumentace zabývající se tvorbou specificky použitelných plugin modulů. Studium pro tvorbu vlastního plugin modulu se tedy částečně opírá jak o dokumentaci formou webu a Javadoc, tak studium existujících plugin modulů, implementujících přibližnou funkcionalitu. Konkrétně se jedná o Maven 2 Project Plugin a Hadoop Plugin.

Existuje více možností pro tvorbu plugin modulu do Jenkins, starší a osvědčený způsob je využít nástroj Maven (výsledkem je soubor HPI či HPL), který se postará o jeho celý životní cyklus. Nově je také možné využít nástroj Gradle (soubory typu JPI), který se snaží zkombinovat některé vlastnosti Apache Ant a Maven. Plugin modul, vyvíjený spolu s touto prací, užívá první způsob, jehož aspekty budou dále také dopodrobna rozebrány.

Z pohledu implementace je Jenkins rozsáhlým projektem, vyvíjeným několik let autorem i komunitou. Obsahuje množství *@deprecated* metod, úprav pro zpětnou kompatibilitu, metod pro udržení rozšiřitelnosti systému či používá další neuvedené frameworky. Tyto

metody se snaží o maximální rozšiřitelnost, interoperabilitu napříč operačními systémy a důraz na tvorbu plugin modulů, tím ale klesá přehlednost kódu a náročnost na studijní a praktické schopnosti programátora. Jenkins implementuje v jazyce Java široké spektrum funkcionality od nejnižší úrovně, přes serializace objektů, tvorbu separátních JVM prostředí, classloading či správu a tvorbu speciálních procesů. Celkově je popis celé funkcionality nad rámec této práce, popsány budou pouze nejdůležitější součásti využity pro implementaci doprovodného plugin modulu s názvem „**Maven Grid Plugin**“.

Vzhledem k neexistenci téměř žádných knižních publikací zabývajících se vývojem v Jenkins, jsou veškeré zdroje v této kapitole čerpané z internetových stránek projektu Jenkins a doprovodných ([11], [9], [10]).

## 6.2 Adresářová struktura plugin modulu

Struktura plugin modulu kopíruje strukturu běžného Maven projektu obsahujícího jeden modul, tedy až na drobné odlišnosti. Nejčastěji obsahuje na nejvyšší úrovni soubor pom.xml a adresář src, popřípadě adresář s testy. Poté se s plugin modulem pracuje jako se standardním Maven projektem, je možné zkompilovat zdrojové soubory, vygenerovat dokumentaci nebo vyčistit jej od vygenerovaných souborů. Pro vývoj je potřeba minimálně Maven verze 3.

Úryvek kódu 6.1: Struktura Jenkins plugin modulu

```
pom.xml
|_ src
    |_ main
    |    |_ java
    |    |_ resources
    |    |_ webapp
    |_ test
...
```

Složka *java* obsahuje veškeré zdrojové soubory plugin modulu, ve složce *resources* jsou umístěny zpravidla soubory Apache Jelly nebo Groovy. Složka *webapp* obsahuje webové zdroje. Složka *test* obsahuje nejčastěji Unit testy a jejich zdroje. Strukturu lze vytvořit pomocí následujícího příkazu (je třeba upravit soubor settings.xml instance Maven podle návodu na https://wiki.jenkins-ci.org/display/JENKINS/Plugin+tutorial) nebo pomocí online generátoru na adrese http://plugin-generator.jenkins-ci.org.

Úryvek kódu 6.2: Tvorba adresářové struktury plugin modulu

```
mvn –U org.jenkins-ci.tools:maven-hpi-plugin:create
```

Minimální obsah souboru pom.xml je zobrazen v úryvku 6.3. Musí obsahovat identifikaci rodičovského modulu Jenkins a verzi, pro kterou je plugin vyvíjen. Tímto dojde ke spojení modulu plugin s implementací jádra Jenkins určité verze, takže nedojde k používání aplikačního rozhraní verzí jiných, čímž by došlo k porušení konzistence celého systému (použití odstraněné funkcionality nebo funkcionality, která ve spojované verzi ještě neexistuje). Protože velmi často vycházejí nové verze Jenkins a aplikační rozhraní se mění, autor si musí hlídat, zda se jeho plugin modul drží rozhraní svými vlastními prostředky nebo se může například spolehnout na služby některého vývojového prostředí v jazyce Java.

Úryvek kódu 6.3: Struktura pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
<modelVersion>4.0.0</modelVersion>
<parent>
        <groupId>org.jenkins-ci.plugins</groupId>
        <artifactId>plugin</artifactId>
        <version>1.400</version>
</parent>

<artifactId>MyPlugin</artifactId>
<packaging>hpi</packaging>
<name>My Jenkins Plugin</name>
<version>0.1-SNAPSHOT</version>
<url>http://wiki.jenkins-ci.org/display/JENKINS/MyPlugin</url>
</project>
```

Další zajímavost je vlastní druh archivace v pom.xml - HPI. Jedná se o formát používaný výhradně pro distribuci Jenkins plugin modulů, strukturou velmi podobný Java archivu WAR (bez souboru web.xml). Je tvořen pomocí plugin modulu pro Maven s názvem „Maven Jenkins Plugin“. Exekucí některých jeho cílů je možné jak archiv vytvářet, tak spouštět jako instanci Jenkins přímo s aktivovaným vyvíjeným plugin modulem. Dovoluje také vygenerovat ze souboru typu HPI soubor HPL („Hudson plugin link“), který pouze odkazuje na umístění zdrojových souborů místo toho, aby zde byla nutnost archivace modulu a jeho kopírování do pracovní složky Jenkins, což je vhodné pro vývojáře a zvláště pro debugging. HPL soubor je obdoba klasického MANIFEST.MF, jen s přidanými metadaty.

Struktura archivu HPI je tedy generována z lokální struktury plugin modulu a znázorněna je v úryvku 6.4. Ve složce *classes* je přeložená hierarchie zdrojových souborů, složka *lib* obsahuje klasické externí knihovny.

Úryvek kódu 6.4: Struktura HPI archivu

```
PluginName.hpi
— META-INF
|    +— MANIFEST.MF
— WEB-INF
|    +— classes
|    +— lib
|         ...
```

Soubor MANIFEST.MF obsahuje některé další důležité informace oproti běžné praxi. Jedná se o volitelný atribut „Plugin-Dependencies“, který určuje, které plugin moduly musejí být v Jenkins nainstalovány pro povolení načtení a správný chod vyvíjeného modulu. Dále je to povinný atribut „Plugin-Class“, která určuje, kterou třídu v hierarchii bude Jenkins instanciovat jako vstupní bod modulu. Soubor může obsahovat i další atributy, ukázka MANIFEST.MF u plugin modulu Maven 2 Project Plugin je v úryvku 6.5.

Úryvek kódu 6.5: Ukázka MANIFEST.MF v HPI archivu

```
...
Plugin-Class: hudson.maven.PluginImpl
Group-Id: org.jenkins-ci.main
Short-Name: maven-plugin
Long-Name: Maven Integration plugin
Plugin-Version: 1.480.4-SNAPSHOT
Plugin-Dependencies: config-file-provider:1.9.1;resolution:=optional,
javadoc:1.0,token-macro:1.1;resolution:=optional
...
```

## 6.3 Možnosti běhu

### Normální mód

Standardní postup tvorby plugin modulu v Jenkins je tvorba struktury, programování funkcionality, kompilace, tvorba HPI archivu pomocí příkazu 6.6.

Úryvek kódu 6.6: Tvorba HPI archivu

```
mvn hpi:hpi
```

Tímto příkazem se také vytvoří HPI archiv ve složce target. Tento archiv je poté možno manuálně vložit do již existující připravené běžící instance Jenkins a v záložce Manage Jenkins→Manage Plugins→Advanced→Upload Plugin plugin modul aktivovat. Po restartu prostředí je nový plugin modul připraven k použití. Celá procedura je však viditelně komplikovaná a nepoužitelná pro produkční vývojové prostředí, proto existuje vhodnější způsob.

### Vývojový mód

Je-li připraven plugin modul pomocí Maven, může být přímo spuštěn ve svém kořenovém adresáři příkazem 6.7.

Úryvek kódu 6.7: Spuštění plugin modulu

```
mvn hpi:run
```

Řízení je předáno Maven, je načten zdrojový POM a je stažena a rozbalena v něm specifikovaná verze Jenkins do aktuální složky. Dále je vytvořen HPL soubor uvnitř takto připraveného pracovního adresáře ve složce plugins, který odkazuje na úroveň, kde se nacházejí zdrojové soubory. Je spuštěna a připravena instance Jenkins s aktivovaným vyvíjeným plugin modulem.

Výhodou je také fakt, že mohou být upravovány zobrazovací komponenty Apache Jelly či Groovy bez znovuspouštění celé aplikace, pro jejich načtení stačí obnovit stránku s komponentou v prohlížeči.

### Debug mód

Maven verze 2 nebo 3 také disponuje skriptem *mvnDebug*, který funguje jako automatizovaný prostředek pro abstrakci spuštění klasického Maven, ale navíc přidá do prostředí

proměnné MAVEN_OPTS parametry, které spustí JVM v tzv. „debug módu“. V tomto módu je spuštěn spolu s programem také speciální JPDA (Java Platform Debugger Architecture) server, který může spolupracovat s různými IDE prostředími či nástroji podporujícími protokol JDPW (Java Debug Wire Protocol). V podstatě spouští prostředí JVM s parametry -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=8000 pomocí příkazu

Úryvek kódu 6.8: Spuštění JVM v debug módu pomocí Maven

```
mvnDebug hpi:run
```

První parametr aktivuje JVM debug mód, druhý parametr určuje vlastnosti doprovodného serveru. Parametr *Dt_socket* určuje, že připojení bude provedeno pomocí socketů[1]. Parametr *suspend* pozdrží start, dokud nebude připojen klient, parametr *server* aktivuje serverový mód. V případě debug módu prostředí Netbeans IDE lze kombinovat celý proces i s velmi užitečnou vlastností, a to Debug → Apply Code Changes (implementováno od Netbeans verze 6.7), která kompiluje a aktualizuje upravený kód za běhu. Tato vlastnost však slouží jen pro dolaďování kódu v rámci metody, nelze ji použít pro úpravu hierarchie, metod a jejich modifikátorů. Produktivnějšího vývojového řešení lze však dosáhnout pomocí frameworku JRebel.

### Vývoj pomocí JRebel framework

Spoluprácí některého z vývojových prostředí IDE podporujícího tento framework a frameworku samotného je možno z velké části vyhnout se ztrátám času s rekompilací a „redeployment“ procesem kódu. Existuje také způsob nevyužívající prostředí IDE, framework je distribuován formou JAR archivu a je možné s ním adekvátně pracovat.

JRebel je v podstatě o plugin modul do Java JVM vyvíjený firmou ZeroTurnaround, který se stará na pozadí o kompilaci v době běhu aplikace upravovaného kódu pomocí classloadingu. Má podporu pro spojení jak na lokální tak na vzdáleně spuštěné aplikace. Výrobce udává, že čas potřebný k projektování změn v kódu lokálně je menší než 1s. Disponuje řadou vlastností, které z něj dělají velice užitečný nástroj pro vývojáře, pracující s webovými, serverovými a vůbec na čas kompilace náročnými Java aplikacemi.

JRebel si umí poradit, kromě jednoduchých změn v kódu, také se změnami struktury, jako přidávání/odstraňování metod, anotací, konstruktorů, atributů tříd či změn rozhraní. Nedokáže však stále řešit problematické části jako přidávání/odebírání rozhraní nebo změnu struktury dědičnosti, čili neumožňuje plnou náhradu procesu rekompilace. Framework není freeware, pro nekomerční vývoj je ale možné zažádat o volné tzv. „social“ licence.

Pro spuštění aplikace s frameworkem je třeba nejprve vygenerovat deskriptor formou souboru rebel.xml, který je vložen do složky *target* kompilovaného plugin modulu. V deskriptoru je obsažena informace o uložení přeložených zdrojových souborů. Poté je možno spustit plugin modul s přidáním parametru

Úryvek kódu 6.9: spuštění JRebel

```
–javaagent:/cesta_k_archivu/jrebel.jar
```

do proměnné prostředí MAVEN_OPTS. Poté lze spustit plugin pomocí předcházejících metod, a to nejen v normálním módu, ale také v debug módu pomocí příkazu 6.8.

[1] Kompletní výpis vlastností parametru -Xrunjdwp na http://docs.oracle.com/javase/1.4.2/docs/guide/jpda/conninv.html#Xrunjdwp

### Vývoj v jádru Jenkins

Pro účely studia kódu nebo potřeby testovat nějakou funkcionalitu Jenkins je také potřeba zmínit nejen možnosti spouštění plugin modulů, ale také jádra Jenkins. Vývojový mód v upraveném kontejneru Jetty je možné spustit ve složce *war* pomocí příkazu

Úryvek kódu 6.10: Spuštění Jenkins ve vývojovém módu

```
mvnDebug hudson-dev:run
```

### Připojení plugin modulů

Jenkins si nese s sebou řadu základních plugin modulů, které implementují nebo doplňují funkcionalitu jádra. Jedná se například o zmiňovaný Maven 2 plugin, SVN plugin, LDAP plugin apod. Každá verze plugin modulu je zpravidla svázáná s verzí Jenkins (například pomocí zmiňovaného POM) a po jeho update dochází také k update plugin modulů, staré plugin moduly jsou přepsány novějšími verzemi. Tento přístup může být nevhodný pro vývoj, kdy je potřeba zachovávat aktuálně vyvíjený plugin modul v současné verzi a neaktualizovat jej. Pokud chceme vytrhnout plugin modul z toho procesu, je potřeba v domovském adresáři Jenkins, ve složce plugins, požadované chování specifikovat pomocí vytvoření souboru s názvem plugin modulu a přidáním koncovky „.pinned“. Tímto ale ztrácí uživatel či programátor možnost automatické aktualizace plugin modulu.

## 6.4 Integrace plugin modulu

Jenkins provádí za běhu automatickou kontrolu přítomnosti veškerých plugin modulů. Vstupním bodem nově importovaného plugin modulu je zpravidla třída rozšiřující třídu Plugin, která je mimo jiné specifikována v archivu HPI v souboru MANIFEST.MF (úryvek 6.5). Dědičnosti z třídy *Plugin* je ovšem volitelná. Tato třída je tedy instanciována a je dobrou praxí překrýt některé metody. Například metodu *start()*, která slouží pro inicializaci prostředků plugin modulu, analogicky také metodu stop(), která slouží pro jejich úklid. Dědit je možno i další nebo žádné metody z třídy *Plugin*. Tímto procesem je ale však jen plugin modul načten do běžícího systému, práceschopnost plugin modulu je implementována pomocí tzv. „Extension points“.

### 6.4.1 Extension point

Jenkins definuje tzv. „Extension point“ (EP) jako rozhraní nebo abstraktní třídu, kterou je modelován některý z aspektů systému. Tyto rozhraní říkají, co je potřeba implementovat a Jenkins jejich prostřednictvím tuto možnost plugin modulům zpřístupňuje[10].

Na adrese https://wiki.jenkins-ci.org/display/JENKINS/Extension+points je definována celá řada oficiálně podporovaných EP, je možno také přidávat nové. Chceme-li ve vyvíjeném plugin modulu některý z EP implementovat, označíme třídu anotací *@Extension*. Následovat musí také informace o třídě, ze které se bude dědit, podobně jako v úryvku 6.11. Anotováním je implementace odpovídajícího EP zaregistrována do Jenkins.

Úryvek kódu 6.11: Ukázka implementace EP

```
@Extension
public class GridQueueTaskDispatcher extends QueueTaskDispatcher {

        @Override
        public CauseOfBlockage canTake(Node node, BuildableItem item) {
                ...
        }

        ...
}
```

Zmíněný úryvek reprezentuje ukázku z části kódu vyvíjeného plugin modulu, který implementuje EP s názvem *hudson.model.queue.QueueTaskDispatcher* sloužící pro změnu chování rozhodovacího procesu plánovače. Třída je pojmenována *GridQueueTaskDispatcher* a anotována pomocí *@Extension*. Jádro Jenkins za běhu zjistí přítomnost této implementace a zajistí, aby metoda *canTake()*, která rozděluje úlohy na různé uzly, byla brána v potaz v rozhodovacím procesu.

Koncept Extension points je tedy bránou k rozšiřitelnosti celého systému. V plugin modulu je zpravidla potřeba implementovat těchto bodů více v závislosti na plánovaném užití modulu a požadované funkcionalitě. EP pomocí jasně daných rozhraní (již za kompilace) udržuje tedy uživatelské implementace plugin modulů od samostatně distribuovaného systému Jenkins, což je velkou výhodou a také důvodem existence jejich rozmanitého množství.

## 6.5 Deskriptory

Další zajímavostí v programovacím stylu Jenkins je tvorba tzv. deskriptorů. Deskriptory jsou v podstatě singleton objekty, které udržují centralizované informace o třídě, se kterou jsou spojeny. Většinou jsou implementovány také jako jejich vnitřní třídy a slouží jako generátory instancí vnější třídy. Tento mechanismus je podobný mechanismu vytvoření instance ze definice třídy v objektově orientovaných jazycích[13]. Je použit z důvodů širších možností konfigurovatelnosti, rozšiřitelnosti, zobrazovatelnosti (pomocí Jelly) a perzistence. Urývek 6.12 ukazuje model struktury třídy, která obsahuje i deskriptor.

Úryvek kódu 6.12: Ukázka deskriptoru jako vnořené třídy

```
public class BuildInfoRecorder extends MavenReporter {

    @Extension
    public static final class DescriptorImpl extends MavenReporterDescriptor {
            public String getDisplayName() {
                return Messages.BuildInfoRecorder_DisplayName();
            }
    }
}
```

V úryvku vidíme, že je vnořená třída pojmenována v Jenkins často používaným identifikátorem *DescriptorImpl* a obsahuje jednu metodu, která slouží ke zjištění jména třídy pro

rendering ve webovém rozhraní. Podstatným faktem je také anotace pomocí *@Extension*, kterou musí obsahovat, jinak nebude načtena za běhu Jenkins.

## 6.6 Apache Jelly

Apache Jelly je skriptovací jazyk sloužící pro tvorbu pohledů v Jenkins a využívá jej velká část webového rozhraní. Jedná se o jazyk syntaxí podobný JSP a XML. Vytváří z jazyka XML v podstatě spustitelný kód. Snaží se oddělit grafické uživatelské rozhraní od implementace v jazyce Java. Skládá se z párových i nepárových tagů, které pomocí předpřipravených knihoven mohou tvořit přehledné formuláře nebo pohledy. Jenkins může mít různé pohledy na existující objekt, proto je potřeba je přehledně odlišit. Pohledem se myslí například zobrazení si některého objektu v grafickém rozhraní z globálního nastavení programu nebo z vlastností úlohy - informace prezentovaná uživateli o tomtéž objektu může být různá. Ukázka kódu v Jelly je v úryvku 6.13.

Úryvek kódu 6.13: Ukázka Jelly kódu

```
<j:jelly xmlns:j="jelly:core" xmlns:f="/lib/form">
        <f:section title="${%Build}">
                <f:entry title="${%Goals}">
                <f:textbox name="goals" value="${it.userConfiguredGoals}"/>
                </f:entry>
        </f:section>
</j:jelly>
```

Na prvním řádku kódu jsou uvedeny použité Jelly knihovny, které budou v následujícím kódu zpřístupněny formou písmena, které předchází dvojtečce. Na druhém řádku je již vytvořen prvek sekce importován z knihovny */lib/form*, který vykreslí na stránku horizontální čáru s nadpisem. Nadpis je určen proměnnou *Build*, jejíž hodnota není přímo zadána, ale je načtena pomocí lokalizačního frameworku (znak %). Dále je vykreslen popisek a textové pole, do kterého uživatel zadává cíle pro spuštění úlohy v Jenkins. Databázi použitelných Jelly výrazů je možné najít na https://jenkins-ci.org/maven-site/jenkins-core/jelly-taglib-ref.html.

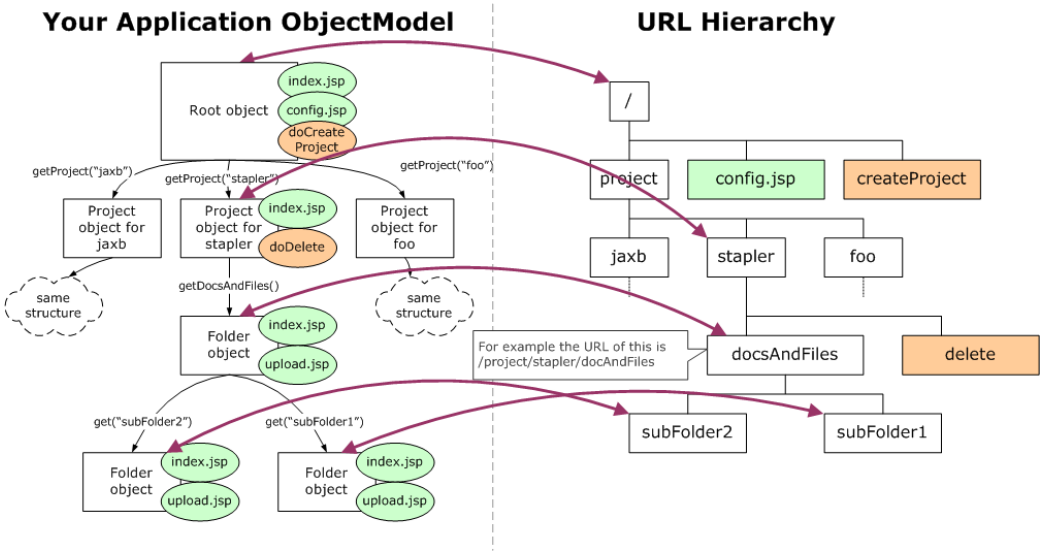
Jelly kód je spojen se svou třídou pomocí umístění v adresářové struktuře. Pro soubory s příponou „.jelly“ je vyhrazena složka uvnitř struktury plugin modulu *src/main/resources*. Tady je třeba vytvořit hierarchii složek přes celý název třídy včetně balíku, se kterou bude Jelly soubor asociován a umístit jej tam. Struktura plugin modulu (6.1) pro soubor index.jelly asociovaný s třídou *org.helloworld.Hello* pak může vypadat, jako v úryvku 6.14.

Úryvek kódu 6.14: Asociace jelly souborů s Java třídou

```
pom.xml
|_ src
    |_ main
        |_ java
            |_ org
                |_helloworld
                    |_Hello.java
        |_ resources
            |_ org
                |_helloworld
                    |_Hello
                        |_index.jelly
```

## 6.7 Stapler

Dalším nástrojem, také od autora Jenkins, je webový framework Stapler. Slouží po párování Java objektů s určitou URL adresou, čili mapuje existující hierarchii Java objektů na hierarchii URL. K tomuto účelu využívá také serializace webových objektů pomocí technologie JSON. Kořenový objekt, označen „/“, je spárován s jedinou existující instancí třídy *Hudson*. Další objekty v podstromu připojují ke kořenové URL řetězce označující cestu a cílový uzel. Tímto způsobem lze přes URL přistupovat ke všem objektům, které jsou pomocí Stapler párovány. Ukázka použití frameworku je na obrázku 6.1.




Obrázek 6.1: Stapler framework[8]

Stapler spolupracuje nejen s JSP, místo „.jsp“ souborů na obrázku si lze představit jako zdroje pohledů a formulářů soubory typu „.jelly“. Stapler umožňuje také asociovat některé atributy serializovaného Jelly souboru přímo s parametry konstruktoru odpovídajícího Java objektu. Objekt musí být anotován pomocí *@DataBoundConstructor*, který říká Jenkins, jako ho instanciovat. Serializovaná data ve formátu JSON jsou prohledána na shodu s něk-

terým z parametrů konstruktoru, pokud se najde shoda, je objekt instanciován. Pomocí této anotace je možno mapovat i zanořené prvky JSON struktury na hierarchii konstruktorů.

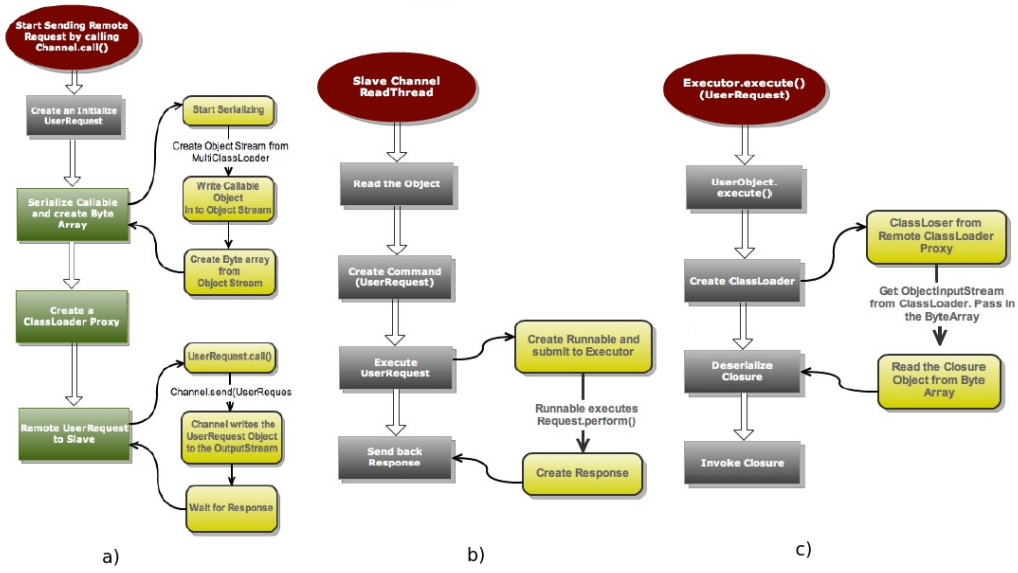
Stapler také řeší některé nedostatky JSP, jako jsou nemožnost tvorby hierarchie, problémy s rekurzivní strukturou či opakovanou alokací objektů.

## 6.8 Husdon Remote Interface

Jenkins je, jak již bylo dříve řečeno, odnoží projektu Hudson a používá některá společná rozhraní. Jedním z nich je i Hudson Remote Interface (HRI). Jedná se o rozhraní pro předávání řízení programu či zasílání požadavků („request“) a příkazů („command“) mezi uzly.

> Hudson is an distributed execution platform. The master can send closures to remote machines, then get the result back when that closure finishes computation. This mechanism is called Hudson remoting. [14]

Architektura HRI se skládá z několika částí. Základem je opět koncept uzlů master a slave. Mezi nimi je vytvořen virtuální kanál („channel“), což je mechanismus zasílání zpráv implementován jako proud I/O příkazů. Může být jak blokující tak neblokující („asyncCall“). Tímto kanálem jsou zasílány požadavky nebo příkazy, od požadavků se také očekává odezva („response“). Třída, která chce být prostředkem ke komunikaci mezi uzly, musí implementovat rozhraní *Callable* nebo jeho potomka. Třída implementující toto rozhraní je serializována, zaslána na cílový uzel a vykonána pomocí metody *Channel.call(Callable)*[14].




Obrázek 6.2: Struktura HRI a) Odeslání požadavku master uzlem b) Přijetí požadavku cílovým uzlem c) Exekuce požadavku [14]

Důležitou funkcí rozhraní je také inicializace a vytvoření spojení master uzlu se slave uzly. Přidání nového uzlu do Jenkins probíhá v režii uživatele, který si může vybrat některou ze čtyř metod. Je možné spustit slave agenta pomocí SSH, JNLP technologie, vykonáním příkazů na master uzlu nebo přidáním kontroly nad službou (MS Windows). Po přidání

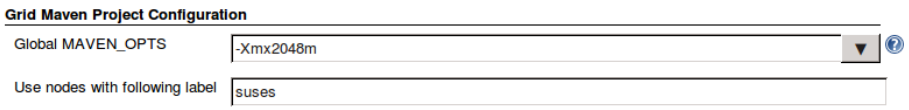
uzlu jsou nakopírovány také důležité archivy, které mohou být classloading subsystémem načteny do JVM straně uzlu. Tento proces probíhá zpravidla při první známce přítomnosti uzlu v systému. Obsluhu této události zajišťují potomci třídy *ComputerListener*.

## 6.9 Maven Grid Plugin

Tato kapitola se bude zabývat již konkrétní implementací plugin modulu s užitím prvků jazyka Java a Jenkins popsaných v předchozích kapitolách.

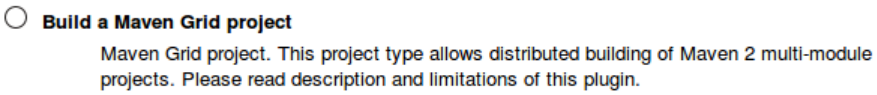
### 6.9.1 Konfigurace úlohy

Adresářovou strukturu plugin modulu není třeba popisovat, kopíruje obecnou strukturu popsanou v sekci 6.2. Po spuštění a integraci do Jenkins plugin modul nejprve očekává konfiguraci globálních nastavení prostřednictvím menu Manage Jenkins→ Configure System→Grid Maven Project Configuration (viz obrázek 6.3). Zde je možné zadat parametry MAVEN_OPTS, se kterými bude spouštěn Maven úloha a také je nutné zadat název uzlů, které může plugin modul využívat. Druhý parametr je povinný a není možné, aby bez něj Jenkins spustil nějakou úlohu tohoto typu.




Obrázek 6.3: Globální nastavení plugin modulu

Globální vlastnosti jsou nastaveny, dalším krokem je tvorba nové úlohy. Nová úloha se vytvoří kliknutím na New Job→Build a Maven Grid project (obrázek 6.4).

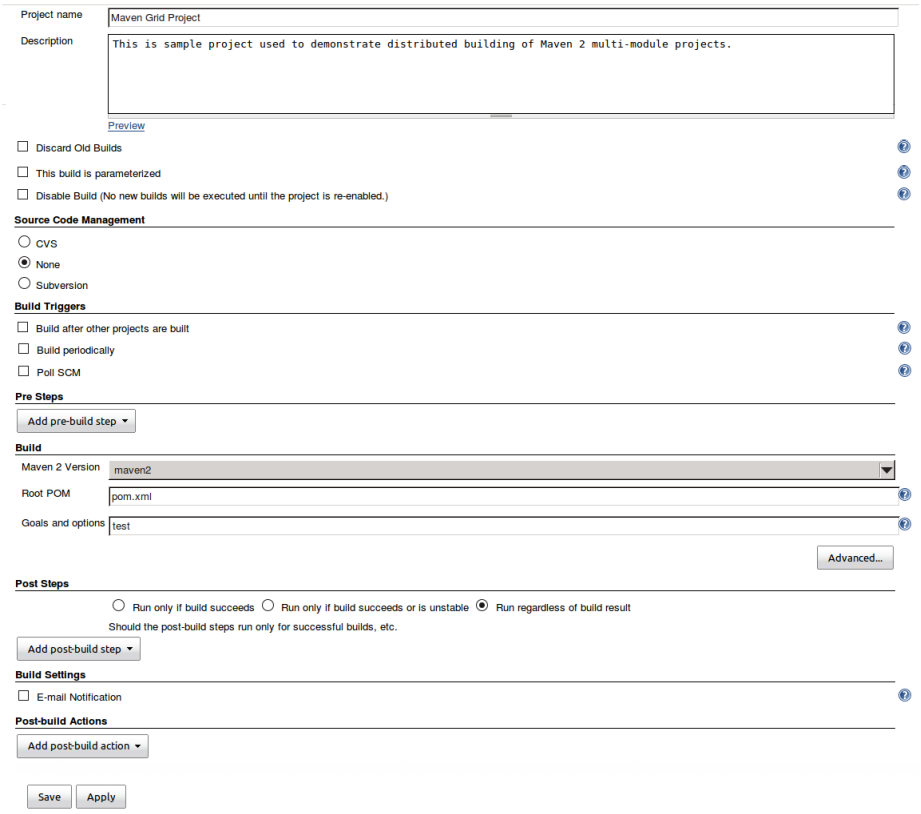



Obrázek 6.4: Tvorba nové úlohy v režii Maven Grid Plugin

Nyní se otevře konfigurační stránka úlohy. Stránka je vzhledem podobná konfigurační stránce Maven 2 Project Plugin, na kterém je tento plugin modul založen, jsou tu však některé odlišnosti.

V gridovém prostředí nelze z principu využívat funkcí Maven 2 Project Plugin, jako jsou inkrementální sestavování, přiřazení úlohy jen některým uzlům nebo velkou část rozšířených možností v sekci Build→Advanced. Odtud stránka vypadá a chová se obdobně jako konfigurační stránka standardní Maven úlohy. Nejdříve je potřeba nakonfigurovat sekci „Source Code Management“ nebo některý z „pre-build“ kroků, které jsou zejména určeny pro stahování zdrojových souborů do lokálních umístění pomocí některého z verzovacích systémů (SVN, CVS, Git - ten je ale třeba doinstalovat jako plugin modul a není součástí jádra Jenkins verze 1.480). Pomocí příznaku „Build after other projects are built“ v sekci „Build Triggers“ je také možné uvést výčtem upstream závislosti pro tuto úlohu.

Poté přichází část „Build“, ve které je třeba definovat s jakým cílem se bude Maven proces na cílovém uzlu spouštět a kde v adresářové struktuře zdrojových kódů má Jenkins najít




Obrázek 6.5: Konfigurace úlohy v Maven Grid Plugin

kořenový POM soubor (implicitně pom.xml). V tomto kroku se tedy vybírají nejdůležitější parametry úlohy. Je také důležité vybrat některou z instalací Maven 2, kterou lze nakonfigurovat v globálním nastavení Jenkins. Pokud bude detekován Maven řady nižší nebo vyšší, úloha skončí s chybou a upozorněním. Práce s jednotlivými moduly pomocí Maven 3 není v Jenkins v současné verzi (1.480) implementována a starší verze Maven nejsou podporovány. V sekci „Advanced“ je také možné redefinovat globální MAVEN_OPTS proměnnou nebo vybrat si explicitně konfigurační soubor pro Maven.

V sekci „Post Steps“ je možno nakonfigurovat kroky po spuštění úlohy, tzv. „post-build“ kroky. Tady je možnost například smazat vygenerovaná data nebo provést úklid. Pomocí „Post-build actions“ lze také přidat downstream úlohy nebo archivovat přeložené zdrojové soubory.

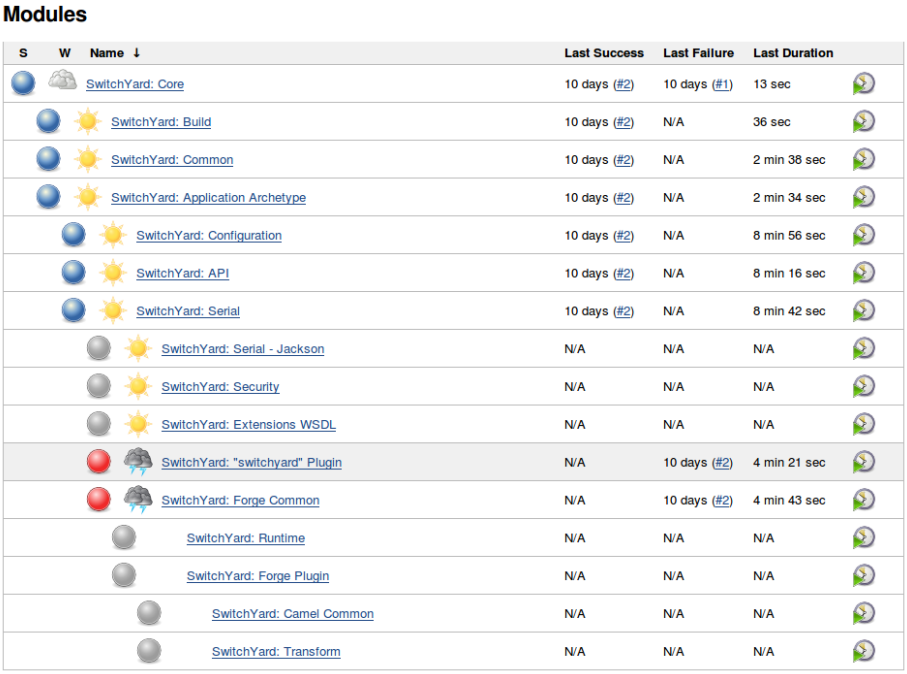
Nyní je třeba konfiguraci uložit tlačítkem „Apply“ nebo přímo tlačítkem „Save“, které stránku uloží a přesměruje prohlížeč na stránku s informacemi o úloze a jejím stavu.

### 6.9.2 Konfigurace podúloh

V levém menu Jenkins najdeme také záložku „Modules“ (obrázek 6.6). Konfigurační stránka konfiguruje nově vytvořenou úlohu pouze globálně, v této sekci je však možno konfigurovat jednotlivé podúlohy reprezentující moduly multimodulárního projektu individuálně. Pokud se jedná o první přidání úlohy do systému, je třeba alespoň jedenkrát spustit úlohu, aby byla struktura modulů načtena a namapována na podúlohy (kap. 4.2.7). Třída zapouzdřující úlohu se jmenuje *MavenModuleSet* a jednotlivé podúlohy *MavenModule*. Po prvním spuštění je načtená struktura zobrazena na stránce formou podúloh tak, že podúloha reprezentují modul, který je v určité úrovni grafu modulů hlouběji, je odsazena od levého okraje více. Podúlohy reprezentující modulu ve stejné hloubce grafu jsou odsazeny stejně. Tímto lze docílit v rámci rozhraní Jenkins optického znázornění úrovní modulů ve grafu, což může sloužit například jako odrazový bod pro monitorování běhu úlohy a podúloh nebo k identifikaci „úzkého hrdla“ v procesu výpočtu.

Obrazovka také slouží jako rozcestník pro konfiguraci jednotlivých podúloh. Kliknutím na některou z podúloh se otevře stránka s informacemi, obdobně jako tomu bylo u rodičovské úlohy. Tady je možné opět zvolit v levém menu záložku „Configure“ a přejít ke konfiguraci podúlohy. Rychlejší přístup je v sekci „Modules“ najet myší na název podúlohy a vyčkat než se objeví menu, tam poté zvolit „Configure“. Objeví se obrazovka konfigurace podúlohy, jejíž konfigurací lze přepsat konfiguraci globální úlohy, kterou podúlohy automaticky dědí. Důležitou položkou je „Restrict where this project can be run“. **Zde je také prostor pro výběr uzlu podle jeho atributu „Label“, na kterém chceme například modul selektivně testovat** (lze například asociovat s podúlohou uzel, který je stavěn na výkonnostní testování, protože víme, že tato podúloha reprezentuje modul, který v testech obsahuje například výkonnostně náročné optimalizace GWT frameworku pro různé prohlížeče, viz obr. 6.7). Označení uzlu je v plné režii Jenkins. Podmínkou je, aby množina označení uzlu (sekce „Labels“ v Manage Jenkins→Manage Nodes→<Node>→Configure) obsahovala také globální označení uzlů vyhrazených pro Maven Grid Plugin (kap. 6.9.1). Ukázka konfigurační obrazovky podúlohy, která je svými funkcemi podmnožinou (rodičovské) úlohy je na obrázku 6.7.

Stránka obsahuje, podobně jako stránka konfigurace úlohy, také položku „Goals“, která může přepsat globálně zadaný cíl pro Maven. Opět je možno uložit stránku pomocí tlačítek „Apply“ či „Save“.




Obrázek 6.6: Obrazovka „Modules“

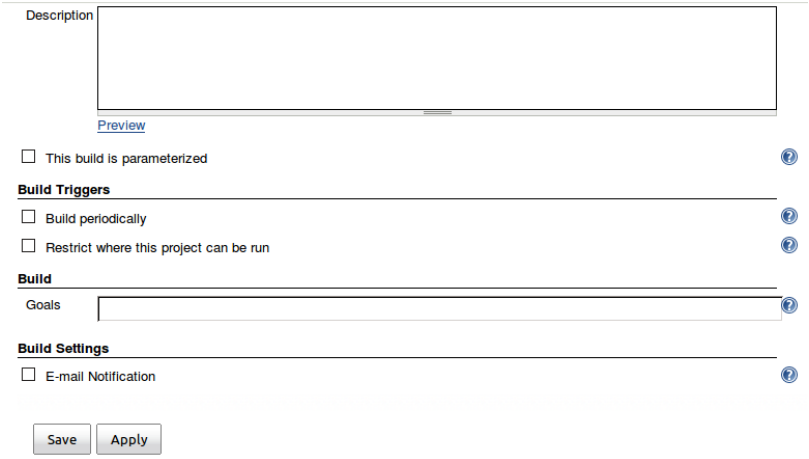
### 6.9.3 Gridová vrstva

Gridová vrstva sestává z instance Hadoop, ale nejsou spouštěny všechny jeho komponenty. Na master uzlu jsou spuštěny dvě další lokální prostředí JVM, ve kterých běží nezávisle na sobě HDFS Datanode a Namenode. Job Tracker není použit, protože není potřeba. Gridová vrstva pracuje jako úložiště či lokální repozitář. Plánovač běžet nemusí, protože o plánování úloh se stará Jenkins (kap. 4.2.9). Nové JVM přebírá některé parametry jako systémové proměnné, cesty k potřebných knihovnám či konfiguračním souborům. Ke každému JVM je vytvořen kanál, kterým komunikuje se systémem, zejména slouží také k posílání serializovaných dat nebo instrukcí mezi sebou. Uvedené uzly jsou spuštěny při startu Jenkins po načtení základní konfigurace. Tvorbu separátních JVM zajišťuje metoda *createHadoopVM()* v třídě *PluginImpl*, která slouží jako vstupní bod plugin modulu (6.4). Jádro metody zobrazuje úryvek 6.15.

Úryvek kódu 6.15: Tvorba nových JVM pro součásti Hadoop

```
public static Channel createHadoopVM (...) {
        ...
        return Channels.newJVM("Hadoop",listener,null,
        new ClasspathBuilder().addAll(distDir,"lib/**/*.jar"),
        Collections.singletonMap("hadoop.log.dir",logDir.getAbsolutePath()));
}
```

Pro správný chod plugin modulu však takto statické chování nestačí, je třeba monitorovat stav nebo přidání dalších slave uzlů do Jenkins. Pro tento účel slouží třída *ComputerListenerImpl*, která dědí z třídy *ComputerListener*. Pokud je v Jenkins zaregistrován nový uzel spravovaný pomocí Maven Grid Plugin nebo takový uzel přešel do stavu online, je vytvořeno další JVM a v jeho rámci je vzdáleně na uzlu spuštěno prostředí Hadoop




Obrázek 6.7: Konfigurační úloha podúlohy

DataNode. Tento DataNode se vzápětí připojuje do datového gridu.

### 6.9.4 Jednotlivé kroky úlohy a podúloh

Máme-li vytvořenou a nakonfigurovanou úlohu typu Maven Grid Plugin (nezaměňovat s úlohou typu „maven2/3 project“) a její podúlohy, můžeme přejít k jejímu spuštění. Jednotlivé kroky životního cyklu spuštěné úlohy jsou tedy

- Načtení struktury podúloh
- Vložení archivů ve formátu TAR reprezentující jednotlivé moduly do inicializovaného HDFS úložiště (gridového repozitáře) do složky */tar/<název_úlohy>/<název_maven_modulu.archiv>*, čímž budou zdrojové soubory dostupné pro jakýkoli uzel v gridu
- Naplánovaní podúlohy reprezentující kořenový modul (kořenové podúlohy) a její vložení do fronty
- Ukončení procesu úlohy, úklid a naplánování závislých podúloh

V tomto bodě již proběhla úloha a vložila do fronty závislé podúlohy. Nyní je prostor pro plánovač, který určí, na kterém uzlu bude podúloha v případě volných zdrojů spuštěna. V rámci (po spuštění) každé podúlohy probíhají následující kroky rekurzivně:

1. Sběr informací o úloze na straně master uzlu
2. Serializace těchto informací a dalších dat, poté odeslání procesu na plánovačem přidělený uzel
3. Deserializace dat - v rámci procesu běžícím na cílovém uzlu jsou staženy a rozbaleny z HDFS úložiště do nově vytvořené složky *deps* zdrojové soubory modulu asociovaného s právě vykonávanou podúlohou
4. Na cílovém uzlu jsou připraveny zdrojové soubory a adresářová struktura ke spuštění operací nad nimi

5. Je vytvořen a spuštěn na operačním systému nezávislý skript, obsahující příkazy pro instalaci Maven závislostí, které jsou potřebné jako prerekvizity pro spuštění Maven se zadaným cílem nad aktuálním modulem

6. Po instalaci prerekvizit je spuštěn hlavní cíl (Maven „goal“) úlohy, například testování

7. Bez ohledu na předchozí cíl je opět prostřednictvím skriptu spuštěn Maven s cílem *package*.

8. Zdaří-li se provedení zadaného cíle v bodě 6. i vytvoření archivu v bodě 7., je archiv vložen v původní formě do HDFS úložiště do složky */repository* a lokální složka *deps* je odstraněna

9. V případě úspěšného ukončení jsou naplánovány downstream závislé podúlohy, v případě chyby v kterémkoli z předchozích bodů dojde k ukončení podúlohy s chybou a závislé podúlohy nejsou spuštěny. Podúlohy reprezentující moduly mající tuto podúlohu jako upstream závislost nemohou být spuštěny (moduly v podgrafu s kořenem v právě zpracovávaném modulu nebudou zpracovány)

V následujících odstavcích budou podrobněji rozebrány některé důležité části běhu podúlohy. V kroce 1. probíhá sběr informací o Maven modulu, se kterým má být v podúloze pracováno. Tyto informace jsou sbírány v metodě *doRun()* třídy *MavenBuild*, která slouží jako vstupní bod pro programování úkonů podúlohy. Tato metoda se převážné zabývá tvorbou řetězce reprezentujícího příkaz, který se má vykonat na cílovém stroji. Část metody také naplňuje informacemi o modulu a jeho prerekvizitách instanci třídy *HadoopSlaveRequestInfo*, která slouží jako struktura pro data, která mají být vzápětí serializována.

Tady stojí za zmínku důležitá metoda invokace kódu na cílovém (vzdáleném či lokálním) uzlu, zobrazena v úryvku 6.16. Nad proměnnou *process* typu *MavenProcess* je zavolána funkce *call()* s parametrem implementujícím rozhraní *Callable* nebo některé z jeho následníků, což je ukázkový příklad invokace kódu pomocí HRI (viz kap. 6.8).

Úryvek kódu 6.16: Invokace (potenciálně) vzdálené metody

```
try {
        Result r = process.call(new Builder(
        listener, new ProxyImpl(),
        getProject(), margs.toList(), systemProps, workspace, serialInfo))
        ...
}
```

Třída implementující uvedené rozhraní je třída *Builder*. Tato důležitá třída slouží ke správě serializovaných dat, provádění většiny interakcí s gridovou vrstvou, spouštění skriptů a ke spuštění samotného sestaveného příkazu v prostředí JVM na cílovém uzlu, které zapouzdřuje přítomnost prostředí Maven i HRI. Při vytvoření třídy jsou data serializována a odeslána kanálem procesu ve vzdálené JVM. Vstupním bodem této třídy je metoda *call()*, která je zavolána vzdáleně při vykonání příkazu v úryvku 6.16.

V této metodě je provedena extrakce zaslaných serializovaných dat (bod 3.). Nyní jsou data zpracována na řetězce informací o modulu a jeho prerekvizitách. Z těchto řetězců je vytvořen příkaz pro instalaci prerekvizit užitím cíle *install* instance Maven, viz úryvek 6.17.

Úryvek kódu 6.17: Instalační příkaz užitý při instalaci prerekvizit

```
mvn install:install-file -Dfile=./deps/<nazev_prerekvizity> \
-DgroupId=<groupId=> -DartifactId=<artifactId> \
-Dversion=<version> -Dpackaging=<packaging>
```

Tento příkaz je vytvářen pro každou jednotlivou prerekvizitu (závislost) zpracovávaného modulu zvlášť a pokud je prerekvizit více, dochází k řetězení těchto příkazů za sebe. Tento přístup má svá omezení, ale další možnosti jak importovat vlastní archivy do lokálního repozitáře Maven jsou přes editaci souborů POM a přidání závislostí pomocí „system scope“ nebo vytváření speciálních lokálních repositářů, což může také narušovat konzistenci repozitáře nebo způsobovat větší závislost na operačním systému. Zde končí bod 5.

V bodě 6. je spuštěn zmiňovaný uživatelsky zadaný cíl pro Maven. Již při startu uzlu a jeho registraci do Jenkins je před samotným spuštěním cíle do prostředí JVM načten achiv „maven-agent“, který je distribuován na uzel spolu s ostatními archivy potřebnými k běhu JVM (viz kapitola 6.8). Tyto prostředky umožňují spolupráci s Jenkins master ze strany cílového uzlu. Slouží ke spuštění Maven s určitými parametry, poskytují jak prostředky pro monitoring průběhu spuštěného Maven procesu, tak pro sběr informací či výsledků testů. Třída *Builder* volá pro exekuci příkazu v rámci Maven procesu funkci *launch()* (úr. 6.18), která je vstupním bodem zmiňovaného archivu a spouští metodu *main()* ve třídě *Main*.

Úryvek kódu 6.18: Spuštění Maven procesu na cílovém systému

```
int r = Main.launch(goals.toArray(new String[goals.size()]));
```

Nyní (bod 7.) je třeba vytvořit archiv z úspěšně zpracovaného modulu, protože další moduly jej mohou vyžadovat za závislost. Závislé moduly mohou být zpracovávány na jiném uzlu, proto je třeba tento archiv uložit do HDFS úložiště, které zajistí jeho dostupnost celým gridem. Archivace se provede opět vykonáním nově vytvořeného dočasného skriptu, kterů obsahuje následující povel pro Maven

Úryvek kódu 6.19: Archivace výsledků exekuce Maven procesu

```
mvn -N -B package -Dmaven.test.skip=true
```

Parametr *-Dmaven.test.skip=true* je nutný, protože zajistí vytvoření archivu specifikovaného v POM bez implicitně zapnutého otestování zdrojových souborů, což může být časově náročné i nechtěné. Tady je také třeba dodat, že cíl *package* je součástí životního cyklu Maven a ten v sobě již obsahuje všechny předchozí cíle, jako například *compile*. Pokud tedy uživatel zadá na konfigurační stránce cíl *package*, provede se tento cíl dvakrát s tím, že při druhém průběhu budou již předchozí cíle v životním cyklu dokončeny a na druhé provedení cíle bude vyplýtváno minimum času i prostředků.

Na konci procesu (body 8. a 9.) dochází k uložení archivu do gridového repozitáře, vyhodnocení výsledků procesu, zachytávání či obsluze výjimek a poslední důležité části, která je spouštěna v případě nevýskytu žádné chyby, a to spouštění downstream závislých podúloh.

Tato část ale neprobíhá v rámci třídy *Builder*, řízení je po dokončení procesu vráceno uzlu master a třídě *Build*, která provede obsluhu výsledků a dokončí metodu *doRun()*. Teprve úspěšné dokončení metody spustí řetězec událostí, které vyústí ve vstup programu do metody *cleanUp()* v tetéž třídě. Tady je provedena obsluha příčiny spuštění podúlohy. Byla-li podúloha spuštěna uživatelem ze stránky *Modules* rodičovské úlohy, je možné předpokládat, že se uživatel snaží spustit právě tuto podúlohu samostatně a ne celý graf podúloh,

které následují, proto v tomto bodě nedojde k naplánovaní závislých podúloh. Naopak, pokud podúloha spuštěna uživatelem není, je možno předpokládat, že se jedná o downstream závislost některé dříve dokončené podúlohy, a závislé podúlohy naplánovány ke spuštění jsou, protože jde zřejmě o globálně spuštěnou úlohu, jejíž cílem je projít celým grafem podúloh.

### 6.9.5 Omezení implementace

- V současné době bohužel Jenkins nepodporuje práci na úrovni jednotlivých podúloh s projekty Maven verze 3, pouze Maven verze 2. I tak je ale plugin modul připraven na případné zavedení podpory
- Všchny moduly v projektu musí být archivovatelné pomocí standardních metod definovaných v Maven 2 (JAR, WAR, EAR, POM)
- Instance Jenkins nesmí běžet na adrese „localhost“ nebo jiném nefyzickém rozhraní, protože HDFS vyžaduje identifikaci všech uzlů vůči sobě a konfigurace musí být na každém uzlu stejná
- Může vznikat problém s konkuretním přístupem do lokálního Maven adresáře, pokud jsou Maven repozitáře umístěny na sdíleném souborovém systému (například problém s NFS zámky)
- Na všech uzlech musí být řádně nainstalováno prostředí Jenkins a s tím spojená práva (zejména ke složce workspace na uzlech)
- Každý modul musí mít nezávislou testovací sadu
- Modul je kompilovatelný pomocí Oracle Java verze 1.6.0 (verze 45) a Maven 3.0.5
- V původní implementaci jsou podporovány operační systémy Unix a Linux

# Kapitola 7

# Testování v praxi

Nyní je třeba zhodnotit výsledky implementace na reálných příkladech. Jako testovací data byly vybrány multimodulární projekty SwitchYard (0.6.0.Final), ModeShape (3.2.0.Final) a Infinispan (5.2.1.Final). Všechny jsou vyvíjeny v jazyce Java. Jejich životní cyklus je spravován pomocí Apache Maven a sestávájí řádově z desítek modulů obsahujících Junit nebo TestNG testovací data. Tato testovací data budou přivedena na vstup plugin modulu, který provede vlastní testování tak, že každý modul dle plánovacího rozhodnutí rozdistribuuje na cílové uzly zapojené v grid architektuře a spustí kompilaci a testování. Jednotlivé projekty jsou vyvíjeny pomocí Maven 3 (kap. 6.9.5), který implementovaný plugin modul nepodporuje, tudíž bylo třeba uvedené projekty upravit tak, aby je bylo možné zpracovávat pomocí Maven 2.

Vzhledem k variabilitě vstupních dat bude jako měrná veličina sloužit čas dokončení kompilace či testování. Měřená data budou vztažena k počtu uzlů zapojených do gridu, které jsou vyhrazeny připravené instanci Jenkins. Cílem bude ukázat, že nejvhodnější vstupní data jsou takové multimodulární projekty, které mají takový počet testů na modul, kdy čas testování modulu (předpokládejme tím i počet testů na modul) je mnohonásobně větší než čas kompilace.

### 7.0.6 Testovací prostředí

Testovací sada strojů sestává z pěti fyzických uzlů, vybavených takto

- Uzel master
  - Intel(R) Xeon(R) CPU X3460, 2.80GHz, 8 jader
  - Red Hat Enterprise Linux Server release 6.4 (Santiago) 64-bit
  - Souborový systém NFS3, verze 3.2.9
- 4x uzel slave
  - Quad-Core AMD Opteron(tm) Processor 2350, 8 jader
  - Red Hat Enterprise Linux Server release 5.9 (Tikanga) 64-bit
  - Souborový systém NFS3, verze 3.2.9

Všechny projekty byly testovány v prostředí Oracle Java 1.6.0 verze 33 s Apache Maven verze 2.2.1. Při testování hodnot pro všech 5 uzlů je použit i uzel master. Na každém uzlu je v Jenkins vyhrazen právě jeden exekutor úloh. Dále uvedené výsledky nelze brát jako

| Projekt | úroveň | šířka | počet zdroj. s. | počet test. s. | počet modulů | **testů/modul** |
|---|---|---|---|---|---|---|
| SwitchYard | 10 | 5 | 627 | 79 | 24 | **3.29** |
| Infinispan | 9 | 8 | 2553 | 834 | 40 | **20.90** |
| ModeShape | 8 | 17 | 1666 | 380 | 50 | **7.60** |

Tabulka 7.1: Atributy testovaných projektů

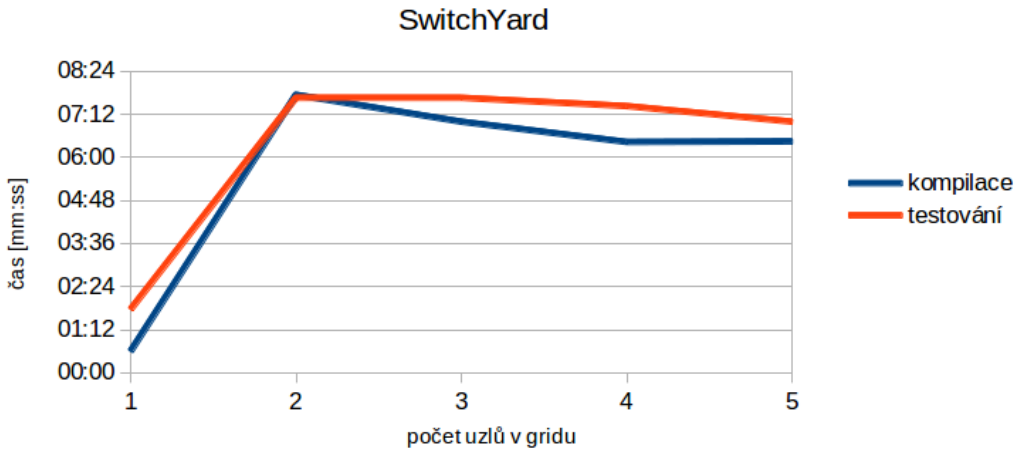
extensivní testování jednotlivých projektů, ale jako demonstraci funkčnosti a vlastností implementace.

V tabulce 7.1 jsou uvedeny nejzajímavější parametry testovaných projektů. Kromě metrik kódu je také uvedena úroveň grafu a šířka grafu, která definuje maximální počet modulů v jedné úrovni napříč všemi úrovněmi grafu. Nejdůležitější je však poslední sloupec, který udává, kolik testovacích souborů připadá na jeden modul v projektu. Tato informace ale není také spolehlivým měřítkem, protože nemusí u projektů obecně odrážet délku testů. Kvůli omezeným možnostem testování a vstupních dat tento předpoklad zanedbáme.

Jednotlivé projekty budou testovány paralelně v závislosti na počtu uzlů. V případě měření pro jeden uzel bude projektu vyhrazen některý z uzlů slave.

### 7.0.7 SwitchYard

Tento projekt slouží jako framework pro vývoj enterprise aplikací poskytující „Java services“. Projekt obsahuje velmi malé množství nenáročných testů na modul, což jej činí nevhodným vstupem.

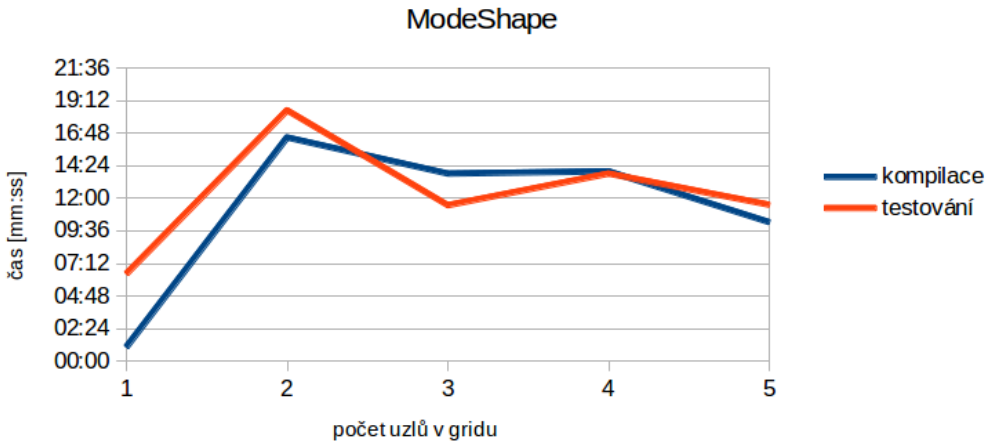



Obrázek 7.1: Ukázka běhu kompilace a testování projektu SwitchYard

Z grafu 7.1 vidíme, že čas testování se v paralelním provedení pohybuje okolo 7 minut a klesá, kdežto čas sekvenčního testování se pohybuje okolo dvou minut. Toto je způsobeno vlivem nízkého poměru testů na modul (3.29). Pro tento typ multimodulárních projektů je tedy paralelní testování nevhodné, protože doba běhu testů nepřekrývá velikost režií gridové vrstvy, Jenkins a Maven.

### 7.0.8 ModeShape

ModeShape je nástroj sloužící jako hierarchické úložiště dat či databáze. Disponuje lepším poměrem testů na modul než Switchyard, poměr však stále dosahuje nízkého čísla 7.6. Rozdíl mezi časem testování sekvenčně a paralelně na pěti uzlech je však stále značný.

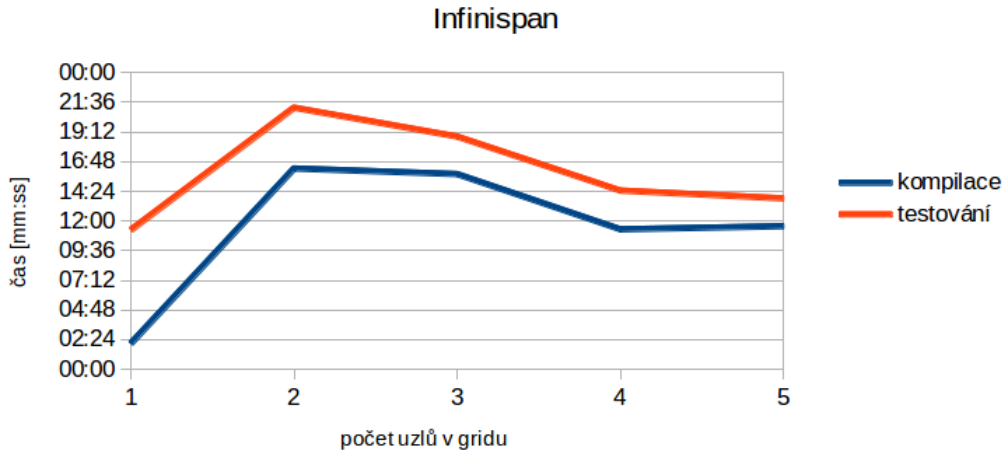



Obrázek 7.2: Ukázka běhu kompilace a testování projektu ModeShape

Zajímavým bodem jsou také délky časů testování a kompilace projektu s užitím 3 uzlů. Testování, přestože je součástí kompilace, bylo provedeno rychleji než další běh kompilace. Toto bylo způsobeno plánovačem, který náhodně vybral jinou, rychlejší, cestu možností naplánování podúloh.

### 7.0.9 Infinispan

Na závěr je měřen projekt Infinispan, popisován v kapitole 3.5. Projekt obsahuje relativně značné množství testovacích souborů, poměr testů na modul je 20.9. V grafu 7.3 lze vidět počáteční rozdíl mezi sekvenčním testováním a kompilací. Tento rozdíl se při zvyšování počtu uzlů snižuje. Jak bylo popsáno v kapitole zabývající se návrhem (kap. 5), při zapojení dalších uzlů do gridu by šlo docílit doby testování, která se rovná maximu z časů otestování jednotlivých modulů. Doba otestování celého projektu by klesala až pod úroveň doby sekvenčního testování. Tato tendence je patrná i z průběhu testování projektu Infinispan. Více uzlů, na kterých by došlo k viditelnému vyrovnání či překročení hranice času sekvenčního testování, nebylo ale v době testování k dispozici.




Obrázek 7.3: Ukázka běhu kompilace a testování projektu Infinispan

# Kapitola 8

# Závěr

Cílem práce bylo navrhnout architekturu software formou plugin modulu pro systém Jenkins CI, který slouží k testování a sestavování rozsáhlých projektů v jazyce Java, s využitím paralelizace těchto procesů a co nejširšího spektra dostupných open source prostředků.

V kapitolách 2, 3 a 4 byly podrobně rozepsány nejdůležitější koncepty možných způsobů paralelního sestavování a testování projektů v jazyce Java, který nativně s takovou funkcionalitou nepočítal, a to na gridové architektuře. V kapitole 5 autor navrhl systém, který s plným využitím popisovaných prostředků dosáhne požadované funkcionality, navíc se bude snažit užitím integračního software ulehčit práci vývojáře delegováním co nejvíce úkolů na automatizační systém. Popsány byly také koncepty zvyšování úrovně granularity testování či možností rozšíření. V kapitole 6 byla rozebrána konkrétní implementace a její detaily, v kapitole 7 jsou prezentovány výsledky měřením na reálných příkladech.

Software se podařilo navrhnout, implementovat a otestovat. Na příkladech reálných multimodulárních projektů byla demonstrována tendence ukazující, že s rostoucím poměrem počtu testovacích dat na modul v projektu lze dosáhnout zrychlení celého procesu paralelního testování na architektuře grid. V případě projektu Infinispan, jehož poměr testovacích souborů na modul je nejvyšší z testovaných (20.9), se čas paralelního testování blížil k času sekvenčního otestování již pro 5 uzlů zapojených v gridu. Systém jako celek je tedy určen pro rozsáhlé multimodulární projekty, jejichž doba testování je několikanásobně vetší než doba kompilace. V tomto případě se tedy projevuje hlavní výhoda plugin modulu, který pomocí distribuce kompilace, testování a lineární škálovatelnost gridové vrstvy, umožňuje zrychlené, efektivní a produktivní paralelní testování softwarového projektu.

Vyvinutý plugin modul je tedy určen pro návrháře softwaru, softwarové inženýry, inženýry kvality produktu či programátory, kteří chtějí zvýšit produktivitu týmu nebo rychlosti integrace jejich software, vyvíjeného v jazyce Java s užitím frameworku Maven. Vzhledem k tomu, že v praxi je s roustoucím multimodulárním projektem potřeba zvyšovat modularitu či dělit projekt na projektů více, čímž klesá přehlednost, může být plugin modul také prostředkem k jejímu udržení na aktuální úrovni. Výhody má i pro projekty, které jsou vyvíjeny bez explicitní snahy programátora o zefektivnění a paralelizaci integračních procesů, kterou zajistí až implementovaný plugin modul.

# Literatura


[1] *JUnit 4 Vs TestNG – Comparison*. 2009-05-23. Dostupné na http://www.mkyong.com/unittest/junit-4-vs-testng-comparison [online, cit. 2012-26-12].

[2] Beust, C.: *TestNG Documentation*. Dostupné na http://testng.org/doc/documentation-main.html [online, cit. 2013-11-04].

[3] Broekhuis A., Zelzer S.: *Native OSGi - Modular Software Development in a Native World*. Dostupné na http://www.osgi.org/wiki/uploads/CommunityEvent2012/Native%20OSGi-%20Alexander%20Broekhuis%20Sascha%20Zelzer.pdf [online, cit. 2013-10-04].

[4] GridGain Systems, Inc.: *GridGain 3.0 - High Performance Cloud Computing*. 2011. Dostupné na http://www.gridgain.com/media/gridgain_white_paper.pdf [online, cit. 2012-26-12].

[5] IBM corp.: *Java Runtime Optimizations*. Dostupné na http://www.research.ibm.com/compsci/project_spotlight/plansoft/index.html [online, cit. 2012-26-12].

[6] Intel® Corp.: *Intel® Cloud Builders Guide to Cloud Designand Deployment on Intel® Platforms: Apache Hadoop*. February 2012. Dostupné na http://www.intel.com/content/www/us/en/big-data/cloud-builders-xeon-apache-hadoop-guide.html [online, cit. 2012-26-12].

[7] Jaiswal, S. K.: *Ontology based Resource Discovery Approach in Grid Environment.* Master thesis. May 2007. Thapar University. Patiala.

[8] Kawaguchi, K.: *Stapler - What is*. Dostupné na http://stapler.kohsuke.org/what-is.html [online, cit. 2013-12-05].

[9] Kawaguchi, K.: *Distributed builds*. Dostupné na https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds [online, cit. 2013-03-02].

[10] Kawaguchi, K.: *Extension points*. Dostupné na https://wiki.jenkins-ci.org/display/JENKINS/Extension+points [online, cit. 2013-03-05].

[11] Kawaguchi K., Molter T.: *Meet Jenkins*. 2012-04-27. Dostupné na https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins [online, cit. 2012-26-12].

[12] Milan Straka a kol.: *Korespondeční seminář z programování XVII. ročník*. 2004-2005. Matfyzpress. Praha. Strana 50.

[13] Oracle: *Descriptor Javadoc documentation*. Dostupné na http://javadoc.jenkins-ci.org/hudson/model/Descriptor.html [online, cit. 2013-21-04].

[14] Prakash, W.: *Hudson Remoting Architecture*. Dostupné na http://hudson-ci.org/docs/HudsonArch-Remoting.pdf [online, cit. 2013-01-05].

[15] Rudolf, M.: *Java Class Loading in ComeBack, version 0.4*. 2009–04–13. Dostupné na http://comeback.sourceforge.net/publications/classloader.pdf.

[16] Smart, J. F.: *Jenkins: The Definitive Guide by John Ferguson Smart (O'Reilly)*. Creative Commons, 2011, ISBN 978-1-449-30535-2.

[17] Sotomayor, B.: *The Globus Toolkit 4 Programmer's Tutorial*. 2004. Dostupné na http://gdp.globus.org/gt4-tutorial/singlehtml/progtutorial_0.2.1.html [online, cit. 2012-26-12].

[18] Surtan, M.: *Clustering modes*. 2010-09-08. Dostupné na https://docs.jboss.org/author/display/ISPN51/Clustering+modes [online, cit. 2012-26-12].

[19] The Apache Software Foundation: *Parallel builds in Maven 3*. Dostupné na https://cwiki.apache.org/MAVEN/parallel-builds-in-maven-3.html [online, cit. 2013-02-02].

[20] The Globus Alliance: *Examples of the Globus Alliance's Impact*. Dostupné na http://www.globus.org/alliance/impact/ [online, cit. 2013-01-03].

[21] The Globus Alliance: *About the Globus Toolkit*. Dostupné na http://www.globus.org/toolkit/about.html [online, cit. 2013-05-04].

[22] Vincent Massol, Jason van Zyl: *Better Builds with Maven. The How-to Guide for Maven 2.0*. 2006. Mergere. Dostupné na http://www.topazproject.org/trac/attachment/wiki/MavenInfo/BetterBuildsWithMaven.pdf?format=raw [online, cit. 2012-26-12].

# Příloha A

# Metriky kódu

Počet zdrojových souborů v jazyce Java: 83
Počet řádků ve zdrojových souborech: 16732
Velikost zdrojových dat: 640294B (640KB)
Název plugin modulu HPI vložitelného do Jenkins: „maven-grid-plugin.hpi“
Velikost spustitelného souboru (Linux, 64-Bit, Maven 3.0.5): 17522261B (17MB)
Název kořenového balíku: *hudson.gridmaven*
Počet vnořených balíků: 6
Přibližná velikost potřebných závislostí v lokálním repozitáři Maven pro spuštění modulu: 700MB

Maven Grid Plugin je licencován pod licencí „The MIT License“.

# Příloha B

# Obsah DVD

- Složka inputs - zde jsou umístěny upravené zdrojové multimodulární projekty, které byly použity jako vstupy
- Složka prerequisites - obsahuje součásti prostředí potřebné pro spuštění implementace
- Složka source - zdrojové soubory implementace v jazyce Java
- Složka thesis - zdrojové soubory práce v prostředí LaTeX
- Soubor README.TXT - informační soubor a návod pro spuštění implementace