Kapitola 1. Přednáška 2 - SW architektury, nástroje správy SW projektů, skriptování

Obsah

Architektury rozsáhlých aplikací v Javě

Charakteristika a požadavky

Modely

Vrstvy

Komponenty

Orientace na služby

Správa

Zabezpečení

Kontejnery a rámce

Architektury orientované na služby (Service-oriented Architectures - SOA)

Motivace k SOA

- Obecný příklon k chápání IT jako poskytovatele služby - servisu - nikoli jen technologie.
- Stejně se začíná přistupovat i k vazbě mezi SW komponentami.
- Orientace na služby se ve vývoji SW stává vůdčím směrem.

Principy SOA

T. Hnilica, teze disertační práce, 2004:

- SOA lze chápat jako virtuální peer-to-peer sít softwarových komponent (služeb), které jsou vzájemně propojeny.

  Služba bývá obvykle aplikace, k níž je známé rozhraní.

  Vlastní implementace služby je pro celý SOA systém skrytá a služba v něm figuruje jako „černá skříňka“.

Pojmy SOA

Charakteristiky SW architektur

- Granularita
- Síla vazeb

Charakteristiky SOA

1. Nahraditelnost služeb
2. Interoperabilita
3. Ladění
4. Integrace systémů třetích stran

Správa sestavování - Ant

Charakteristika

- Ant je platformově přenositelnou alternativou nástroje typu Make
- Ant je rozšiřitelný - lze psát nejen nové "cíle", ale i definovat dílčí kroky - "úlohy"
- Popisovače sestavení používají XML syntaxi, psaní není tak náročné jako makefile

Motivace

- Proč Ant, když je tu dlouho a dobře fungující Make?
- Make byl koncipován jako doplněk shellu, psaly se skripty a nativní (platformové) nástroje
- Syntaxe byla složitá a neflexibilní
- Make jako takový nebyl rozšiřitelný
- Make byl zaměřen převážně na potřeby původního použití - jazyk C, unixový shell

Struktura projektu

Řízení sestavování Antem postupuje podle popisu v souboru build.xml.

Ten obsahuje následující prvky:

project	celý projekt, obsahuje sadu cílů, jeden z nich je hlavní/implicitní
target	cíl, nejmenší zvenčí spustitelná jednotka algoritmu sestavování
task	úloha, atomický krok sestavení, lze použít task vestavěný nebo uživatelský

Příklad 1

Závislosti

- Mezi cíly mohou být definovány závislosti.
- Závisí-li volaný cíl na jiném, musí být nejprve splněn cíl výchozí a pak teprve cíl závisející.
- Cíl může záviset i na více jiných.

```
<target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="B"/>
<target name="D" depends="C,B,A"/>
```

Varování

Pozor na cyklické závislosti!

Příklad 2

Systémy správy verzí

Motivace

Systémy pro správu verzí jsou nezbytností pro

- údržbu větších SW projektů
- projektů s více účastníky
- projektů s vývojem z více míst

Pouhý sdílený, vzdáleně přístupný souborový systém nestačí.

Principy

- efektivně ukládat více verzí souborů i adresářů (často s malými změnami)
- rychle získávat aktuální verzi, ale

- mít možnost vrátit se ke starší verzi
- zjistit rozdíly mezi verzemi
- zajišťovat proti souběžné editaci z více míst
- umožnit i lokální práci s následným potvrzením do systému (commit)

Klasická řešení - RCS a CVS

RCS Revision Control System

CVS Concurrent Version System

RCS je klasický, původně na UNIXech existující systém navržený pro sledování více verzí souborů.

Prvotním cílem bylo zefektivnit ukládání více verzí

- s novou verzí se nemusí ukládát celý soubor
- rychle se dají zjistit rozdíly (změny) mezi verzemi
- historie změn (changelog, history) se lehce udržuje
- změny lze identifikovat i názvy - pojmenovat symbolickými klíčovými slovy ($Author$, $Date$...)

Typické příkazy správy verzí

(podobné jsou v CVS, SVN)

commit	publikuje (odešle, potvrdí) změny z lokálního pracovního prostoru do skladu (repository)
remove	maže soubory z lokálního pracovního adresáře, ze skladu se smažou až po "commit"
add	přidá nový soubor do pracovního adresáře, do skladu se přidají až po "commit"
update	aktualizuje lokální kopii pomocí změn zaregistrovaných ostatními ve skladu
ckeckout	vytvoří soukromou lokální kopii požadovaných souborů ze skladu, tyto můžeme lokálně editovat a posléze potvrdit (commit) zpět

Klienti

Syst. správy verzí nabízejí

- nativní klienty integrované do hostujícího prostředí
- webové rozhraní spíše pro prohlížení
- API pro přímý/programový přístup

Pro co se systémy nehodí?

- pro ukládání (stabilních) artefaktů
- jako úložiště (read-only) souborů pro stahování
- ... kdyby se to hodilo na všechno, nabízel by to každý filesystém...

Subversion

- implementace systému řízení verzí
- moderní alternativa CVS
- jako server dostupný na všechny běžné platf. (zejm. Linux, Win)
- klienti taktéž, např. na Win

Subversion - klient Tortoise pro Windows

Klient pro Win 2k, XP i 98... vč. integrace do GUI

- kontextové nabídky
- nativní nástroje

Tortoise -- vzdálený přístup

- Tortoise umožňuje bezpečný přístup k vzdálenému úložišti i prostřednictvím šifrovaných protokolů
- používá se upravený klient PuTTY

Maven

Motivace

Pro sestavování, správu a údržbu SW projektů menšího a středního rozsahu se delší dobu úspěšně využíval systém Ant [http://ant.apache.org].

Oproti klasickým nástrojům typu unixového make poskytoval Ant platformově nezávislou možnost popsat i složité postupy sestavení, testování a nasazení výsledků SW projektu.

Ant měl však i nevýhody:

- pro každý projekt (i když už jsme podobný řešili) musíme znovu sestavit - poměrně velmi technický - popisovač (`build.xml`)
- popisovač je vždy téměř stejný a tudíž
- neříká nic o *obsahu* vlastního projektu, je jen o procesu sestavení, nasazení...
- neumožňoval zachytit *metadata* nezbytná pro zařazení projektu do širšího kontextu, mezi související projekty, atd.

Maven - charakteristika

- nástroj řízení SW projektů
- open-source, součást skupiny nástrojů kolem Apache
- dostupný a popsaný na http://maven.apache.org
- momentálně (září 2004) již jako použitelná, ostrá, verze 1.0

Project Object Model (POM)

- projekt řízený Mavenem je popsán tzv. *POM* (Project Object Model), obvykle `project.xml`
- POM nepopisuje postup sestavení, ale *obsah* projektu, jeho název, autora, umístění, licenci...
- postup sestavení je "zadrátován" v Mavenu, protože je pro většinu projektů stejný
- programátor není frustrován opakováním psaní popisovačů `build.xml`, návrhem adresářové struktury...
- nicméně, Maven je založen na Ant, jeho `build.xml` popisovače lze znovupoužít

Projekt v Mavenu

Základní filozofie projektu v Mavenu:

- jeden projekt => jeden tzv. *artefakt*

Artefaktem může být typicky:

- .jar - obyčejná aplikace nebo knihovna (javové třídy, soubory .properties, obrázky...)
- .war - webová aplikace (servlety, JSP, HTML, další zdroje, popisovače)
- .ear - enterprise (EJB) aplikace (vše výše uvedené pro EJB, popisovače)

Maven repository

- základním organizačním nástrojem pro správu vytvořených (nebo používaných) artefaktů je *repository*
- artefakt, tj. výstup projektu, se může v repository vyskytovat ve více verzích
- repository je:

vzdálená (remote)	slouží k centralizovanému umisťování jak vytvořených, tak používaných artefaktů
	dosažitelná pro čtení pomocí HTTP: je to de-facto běžné webové místo
lokální (local)	slouží k ozrcadlení používaných artefaktů ze vzdálené repository
	typicky zvlášť každému uživateli - v jeho domovském adresáři
	slouží též k vystavení vytvořených artefaktů "pro vlastní potřebu"

- Maven má nástroje (pluginy) pro vystavování artefaktů do repository

Instalace a nastavení

Maven lze stáhnout z http://maven.apache.org v binární i zdrojové distribuci.

Binární distribuce je buďto čistě "java-based" nebo ve formě windowsového `.exe`.

Pak se nainstaluje do `Program Files`

Po instalace je třeba nastavit proměnnou prostředí `MAVEN_HOME` na adresář, kam se nainstaloval.

Kromě toho ještě přidat adresář `%MAVEN_HOME%\bin` do `PATH`.

Příklad POM (project.xml)

Příklad minimálního popisovače `project.xml`:

```
<project>
```

```
   <pomVersion>3</pomVersion><!-- verze POM - zatím vždy 3 -->
   <id>RunningCalculator</id><!-- jednoznačné id projektu -->
   <name>RunningCalculator</name><!-- (krátké) jméno/nemusí být jednoznačné -->
   <currentVersion>0.1</currentVersion><!-- momentální verze -->
   <organization><!-- organizace vytvářející projekt -->
      <name>Object Computing, Inc.</name>
   </organization>
   <inceptionYear>2004</inceptionYear><!-- rok zahájení projektu -->
   <shortDescription>calculates running pace</shortDescription><!-- stručný popis
   <developers/>
```

Příklad POM - pokračování

Příklad minimálního popisovače `project.xml`:

```
   <dependencies><!-- závislosti -->
      <dependency><!-- závislost -->
         <groupId>junit</groupId><!-- skupina artefaktu -->
         <artifactId>junit</artifactId><!-- označení artefaktu -->
         <version>3.8.1</version><!-- verze artefaktu -->
      </dependency>
   </dependencies>
   <build><!-- odkud se co a jak sestavuje... -->
      <sourceDirectory>src/java</sourceDirectory><!-- adresář zdrojů -->
      <unitTestSourceDirectory>src/test</unitTestSourceDirectory><!-- adresář zdro
      <unitTest><!-- které soubory jsou třídy testů -->
         <includes>
            <include>**/*Test.java</include>
         </includes>
      </unitTest>
   </build>
</project>
```

Struktura POM obecně

```
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="MAVEN_HOME/maven-project.xsd">
   <pomVersion>3</pomVersion>
   <id>unique-project-id</id>
   <name>project-name</name>
   <groupId>repository-directory-name</groupId>
   <currentVersion>version</currentVersion>
   <!-- Management Section -->
   <!-- Dependency Section -->
   <!-- Build Section -->
   <!-- Reports Section -->
```

```
</project>
```

Proměnné (properties) v POM

- Jsou podobně jako u Antu definovatelné a využitelné (odkazovatelné) v popisovači, zde `project.xml`.
- Vyskytne-li se zavedení určité vlastnosti (property) vícekrát, uplatní se *poslední*.
- Vlastnosti jsou vyhledávány v pořadí:
  1. `project.properties`
  2. `build.properties`
  3. `${user.home}/build.properties`
  4. vlastnosti specifikované na příkazové řádce -D*`key`*=*`value`*
- Na vlastnost se lze odvolat pomocí *`${property-name}`*

Struktura repository

Týká se jak vzdálené, tak lokální repository.

Obecně je relativní cesta v rámci repository k hledanému artefaktu: repository/*`resource-directory`*/jars/*`jar-file`*

konkrétní např.: repository/junit/jars/junit-3.8.1.jar

Cíle v Mavenu

Cíle (goals) v Mavenu odpovídají zhruba antovým cílům (target).

Spouštění Mavenu vlastně odpovídá příkazu k dosažení cíle ("attaining the goal"):

maven *`plugin-name`*[:*`goal-name`*]

Maven Plugins

Zásuvné moduly (plugins) obsahují předdefinované cíle (goals) a jsou taktéž uloženy v repository.

Většinou jsou jednoúčelové, slouží/směřují k jednomu typu artefaktu, např.:

- checkstyle, clean, clover, cruisecontrol, dist, ear, eclipse, ejb, fo, genapp, jalopy, jar, java, javadoc, jboss, jcoverage, maven-junit-report-plugin, pom, site, test, war, xdoc

Často používané cíle

clean	smaže vygenerované soubory (podstrom `target`)
java:compile	přeloží všechny javové zdroje
test	spustí všechny testy
site	vygeneruje webové sídlo projektu
dist	vygeneruje kompletní distribuci

Vytvoření projektu

1. vytvořit prázdný adresář pro vytvářený projekt
2. spustit **maven genapp** (Zeptá se na id projektu, jeho jméno a hlavní balík. V něm předgeneruje jednu třídu.)
3. tím se vytvoří následující soubory:
   - project.xml, project.properties
   - src/conf/app.properties
   - src/java/*package-dirs*/App.java
   - src/test/*package-dirs*/AbstractTestCase.java
   - src/test/*package-dirs*/AppTest.java
   - src/test/*package-dirs*/NaughtyTest.java

Reporting

Generování reportů (zpráv) je jednou se základních funkcí Mavenu.

Které reporty se generují, je regulováno v project.xml v sekci *reports*:

```
<reports>
   <report>maven-checkstyle-plugin</report>
   <report>maven-javadoc-plugin</report>
   <report>maven-junit-report-plugin</report>
</reports>
```

Rozšíření možností Mavenu

Cílů (goals) je sice v Mavenu řada, ale přesto nemusejí stačit anebo je třeba měnit jejich implicitní chování.

Potom lze před nebo po určitý cíl připojit další cíl pomocí *preGoal* a *postGoal*.

Ty se specifikují buďto v nebo.

1. `maven.xml` ve stejném adresáři jako `project.xml` nebo
2. v zásuvném modulu (pluginu)

Zcela nové cíle je možné napsat ve skriptovacím jazyku *jelly* (s XML syntaxí).

Skriptování v javovém prostředí - BSF

Co je skriptování?

Co odlišuje skriptování od "ostatních" pg. jazyků?

- Rychlý vývoj, přímočarý životní cyklus SW: napiš - spusť (- potom odlaď)
- Obvyklé dynamicky (až za běhu) typovaný jazyk, nevyžaduje deklarace proměnných, definice tříd...
- Jazyk je často kombinovatelný s běžnými pg. jazyky - lze volat jejich metody, používat knihovny...
- Prostá, obvykle intuitivní, syntaxe
- Mnohdy jde de-facto o syntaktický klon "plného" jazyka - mj. aby se snáze učilo
- Obvykle malé nároky na udržovatelnost, dokumentovatelnost, rozšiřitelnost vzniklých SW výtvorů
- Jednoduché věci jdou napsat jednoduše, složité složitě, nepěkně nebo pořádně vůbec...

Proč skriptovat?

Kdy obvykle (nejen v Javě) cítíme potřebu skriptovat?

- Když zkoušíme, "hrajeme si", testujeme narychlo vytvořené věci
- Hledáme vhodné hodnoty parametrů, hezký vzhled něčeho
- Potřebujeme ovládat konfiguraci složitější aplikace - které objekty se mají vytvořit, jak je propojit...

Proč skriptovat právě teď?

Proč je potřeba skriptovat silná právě dnes, když je tolik dokonalých programovacích jazyků s rychlými

překladači?

- SW architektury jsou složité, je třeba je - mnohdy dynamicky - (re)konfigurovat.
- Často integrujeme - a při integraci je nutné zkoušet, ladit, ale i konfigurovat.
- Máme málo času přemýšlet nad složitou architekturou "úplné" aplikace, chceme rychle něco navrhnout a vyzkoušet nebo i používat.

Poznámka

Takové rychlovýtvory nezřídka mívají delší životnost než složitě a dlouho budované "pořádné" aplikace - přicházejí rychle a v pravý čas!

Bean Scripting Framework

Bean Scripting Framework (BSF) je projektem jakarta.apache.org, původně však vytvořený v IBM T.J.Watson Laboratories (jako produkt "alphaWorks").

Jedná se o rámec umožňující:

- přístup z javových aplikací ke skriptování v mnoha běžných skript. jazycích (Javascript, Python, NetRexx ...)
- naopak ze skriptů je možno používat javové objekty

Poznámka

To mj. dovoluje "save of investment" do stávajících skriptů - i z plnohodnotného prostředí (Java) je lze volat!

BSF - co nabízí

BSF obsahuje dvě hlavní komponenty:

BSFManager	spravuje javové objekty, k nimž má být ze skriptů přístup. Řídí provádění těchto skriptů.
BSFEngine	rozhraní, API, které musí hostující skriptovací jazyk nabídnout, aby jej bylo možné v rámci BSF pužívat.

BSF - typické použití

- je možné pouze instanciovat jeden `BSFManager`a

- z něj přes `BSFEngine`spouštět skripty s možností přístupu k objektům v kontextu manažeru.

BSF - download a další info

- BSF (software i další info) lze získat na http://jakarta.apache.org/bsf.
- Vynikající články o BSF jsou k dispozici přímo na http://jakarta.apache.org/bsf/resources.html.
- úvodní prezentace V. Orlikowského [http://www.dulug.duke.edu/~vjo/papers/ApacheCon_US_2002/intro_to_bsf.pdf].
- http://www.javaworld.com/javaworld/jw-03-2000/jw-03-beans.html

Skriptování v javovém prostředí - Groovy

Groovy - motivace

- Již delší dobu pro Javu existuje rámec BSF podporovaný řadou skriptovacích jazyků.
- Autorům Groovy se však většina z nich zdála syntakticky "těžko stravitelná" pro javového programátora, který "málo, ale občas přece" potřebuje skriptovat.
- Groovy je tedy vytvořet v Javě a na míru pro javové programátory.
- Nabízí velmi příjemnou a intuitivní syntaxi, hezkou konzolu pro spouštění atd.
- Skript se překládá do javového bajtkódu, je tedy na běhové úrovni dobře interoperabilní s Javou.

Stažení

- Groovy najdeme na http://groovy.codehaus.org.
- Plná, tj. zdrojová i binární distribuce má vč. dokumentace a příkladů přes 56 MB!!!
- Je zároveň hezkou ukázkou netriviální projektu řízeného Mavenem.

Instalace

- rozbalit distribuci do zvoleného adresáře
- nastavit systémovou proměnnou GROOVY_HOME na tento adresář
- přidat $GROOVY_HOME/bin do PATH

Spuštění

Tři základní způsoby:

groovysh	řádkový Groovy-shell
groovyConsole	grafická (Swing) konzola Groovy
groovy SomeScript.groovy	přímé (neinteraktivní) spuštění skriptu pod Groovy

Příklad - iterace

Iterace přes všechna celá čísla 1 až 10 s jejich výpisem:

```
for (i in 1..10) {
  println "Hello ${i}"
}
```

Příklad - mapa

Definice mapy (asociativního pole) a přístup k prvku:

```
map = ["name":"Gromit", "likes":"cheese", "id":1234]
assert map['name'] == "Gromit"
```

Příklad - switch

Řízení toku pomocí switch s velmi bohatými možnostmi:

```
x = 1.23
result = ""
switch (x) {
    case "foo":
        result = "found foo"
        // lets fall through
    case "bar":
        result += "bar"
    case [4, 5, 6, 'inList']:
        result = "list"
        break
    case 12..30:
        result = "range"
        break
    case Integer:
        result = "integer"
        break
```

```
    case Number:
        result = "number"
    break    default:
        result = "default"
}
assert result == "number"
```