Aplikační vrstva javových aplikací - řízení toku, správa komponent. Pokročilé zásady a metodiky návrhu aplikační logiky - Design by Contract, Inversion of Control, Aspect-oriented Programming. Kontejnery, aplikační servery.

Obsah

Aplikační vrstva javových aplikací

Aplikační vrstva javových aplikací

Aplikační vrstva javových aplikací

Design by Contract

Design by Contract - návrh podle kontraktu

Nejde o nic jiného, než o zajištění, aby výsledný navržený program *splňoval specifikaci*, tj.:

- aby pro každý atomický, zvenčí viditelný/volatelný kus kódu (typicky metoda) byly specifikovány vstupní a výstupní podmínky
- a aby jejich platnost byla za běhu zaručena
- mezi zadavatelem (tj. analytikem, příp. zákazníkem) a návrhářem tak vzniká
- dohoda (contract), že specifikace bude dodržena

DBC - jak dosáhnout

K dosažení tohoto ideálního stavu vede budto čistě formální cesta:

- specifikace zmíněných podmínek matematickými prostředky a
- formální dokazování korektnosti

Dále zmiňované nástroje však toto nedokážou; omezují se na běhovou kontrolu platnosti předpsaných podmínek

DBC - nástroj jass

jass je preprocesor javového zdrojového textu. Umožňuje ve zdrojovém textu programu vyznačit podmínky, jejichž splnění je za běhu kontrolováno.

Podmínkami se rozumí:

- pre- a postconditions u metod (vstupní a výstupní podmínky metod)
- invarianty objektů - podmínky, které zůstávají pro objekt v platnosti mezi jednotlivými operacemi nad objektem

Postup při práci s `jass` [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=jass]

Postup práce s jass:

- stažení a instalace balíku z http://csd.informatik.uni-oldenburg.de/~jass/
- vytvoření zdrojového textu s příponou `.jass` [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=.jass], javovou syntaxí s použitím speciálních komentářových značek
- takový zdrojový text je přeložitelný i normálním překladačem **javac** [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=javac], ale v takovém případě ztrácíme možnosti jass
- proto nejprve `.jass` [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=.jass] souboru převedeme preprocesorem jass na javový (`.java` [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=.java]) soubor
- ten již přeložíme **javac** [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=javac] a spustíme **java** [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=java], tedy jako každý jiný zdrojový soubor v Javě
- z `.jass` [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=.jass] zdrojů je možné vytvořit také dokumentaci API obsahující jass značky, tj. informace, co kde musí platit za podmínky atd. - vynikající možnost!

Odkazy

- JUnit homepage [http://junit.org]
- Java 1.4 logging API guide [http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/]
- Log4j homepage [http://jakarta.apache.org/log4j/docs/index.html]
- Jakarta Commons Logging homepage [http://jakarta.apache.org/commons/logging/]
- jass homepage [http://csd.informatik.uni-oldenburg.de/~jass/]

- jass demo by Tomáš Pitner [http://www.fi.muni.cz/~tomp/dbc]
- úvodní materiálek [http://www.inf.fu-berlin.de/lehre/SS01/VIS/Dokumente/Vortraege/junit.pdf] k použití junit (v němčině, jako PDF)

Inversion of Control (IoC)

Nezbytné pojmy z komponentních systémů

Uvádíme pragmaticky jen to, co je potřeba zde (pro potřeby IoC), nechápat jako komplexní terminologii.

komponenta (component)	objekt poskytující navenek ucelenou funkcionalitu (část aplikační nebo pomocné logiky) komponenta je obvykle chápána jako "velký objekt" nebo graf více objektů s vnějším rozhraním ("fasádou") komponenta je sice do jisté míry samostatná, ale většinou nežije nezávisle; za běhu potřebuje návaznosti na další komponenty nebo hostující rámec (kontejner)
kontejner (container)	objekt, v němž jsou za běhu aplikace uloženy a spravovány komponenty (objekty) kontejner dokáže většinou komponenty i vytvářet a poskytovat odkazy na ně (vyhledávat je)

IoC - Motivace

V komponentních systémech bývá tradičním problémem zajistit správnou inicializaci a provoz komponent závislých na ostatních.

- jak závislosti popsat
- jak získat objekty (komponenty), na nichž vytvářená komponenta závisí
- jak tuto komponentu vytvořit
- jak závislosti předat ("injektovat") do ní

Tradiční řízení životního cyklu komponent

Co je třeba udělat při nasazení jedné nové komponenty

1. Připravit komponenty, na nichž "ta moje" závisí

2. Vytvořit "noji komponentu"
3. Nastavit závislosti

Postup vypadá přímočaře, ale je bohužel rekurentní... v bodě 1 (připravit komponenty...) se opakuje rekurentně celý postup

IoC - Hlavní princip

V Inversion of Control obracíme tento (pro komponentního programátora) nepraktický, obtížný postup.

O řešení závislostí se postará rámec (kontejner), komponenta pouze deklaruje na čem závisí.

IoC - Možné podoby

Historicky se postupně vyvinuly tři přístupy k "injektáži" potřebných závislostí; tedy k IoC:

- Interface Injection
- Setter Injection
- Constructor Injection

Blíže viz popis k IoC v systému (rámci) vraptor [http://vraptor.arca.ime.usp.br/beginner/ioc.html] a následující slidy.

Interface Injection

Komponenta MUSÍ IMPLEMENTOVAT určité, rámcem/kontejnerem dané rozhraní (příklad z článku Intro. to AOP [http://today.java.net/pub/a/today/2004/02/10/ioc.html]):

```
import org.apache.avalon.framework.*;

  public class JDBCDataManger implements Serviceable {
    DataSource dataSource;
    public void service (ServiceManager sm)
          throws ServiceException {
      dataSource = (DataSource)sm.lookup("dataSource");
    }

    public void getData() {
      //use dataSource for something
    }
}
```

Nevýhoda: musí implementovat dané rozhraní, nelze vyvíjet zcela nezávisle.

Setter Injection - komponenta

Komponenta je jako objekt JavaBean, má set*XXX* metody:

```
public class JDBCDataManger {
  private DataSource dataSource;

  public void setDataManager(DataSource dataSource {
    this.dataSource = dataSource;
  }

  public void getData() {
      //use dataSource for something
  }
}
```

Setter Injection - popis komponenty

Rámec (kontejner), např. Spring musí vědět, jak komponentu vytvořit a na čem závisí:

```
<bean id="myDataSource"
  class="org.apache.commons.dbcp.BasicDataSource" >
  <property name="driverClassName">
    <value>com.mydb.jdbc.Driver</value>
  </property>
  <property name="url">
    <value>jdbc:mydb://server:port/mydb</value>
  </property>
  <property name="username">
    <value>root</value>
  </property>
</bean>
```

a druhá komponenta:

```
<bean id="dataManager"
  class="example.JDBCDataManger">
  <property name="dataSource">
    <ref bean="myDataSource"/>
  </property>
</bean>
```

Setter Injection - výhody/nevýhody

Oproti Interface Injection: netřeba implementovat rozhraní

Je ale nezřetelné, které setXXX metody jsou pro účely nastavení závislostí přes Setter Injection a které ne.

Constructor Injection

Odpovídá přístupu "Good Citizen" (označení zavedl J. Bloch ze Sun):

- objekt je po vytvoření (a aplikaci konstruktoru) plnohodnotný, plně inicializovaný, platí pro něj všechny invarianty, lze jej použít

Constructor Injection - příklad komponenty

```
public class JDBCDataManger {
  private DataSource dataSource;

  public JDBCDataManger(DataSource dataSource) {
    this.dataSource = dataSource;
  }

  public void getData() {
      //use dataSource for something
  }
}
```

Použití IoC - kontejnery

Existují jednoduché (lightweight) kontejnery pro nasazení a provoz komponent s využitím IoC.

Tyto kontejnery mnohdy neumí nic navíc, jde jen o základní správu komponent.

Příkladem je PicoContainer [http://picocontainer.org/] a Spring [http://www.springframework.org/], který je však komplexnější (a složitější).

Aspect Oriented Programming (AOP)

AOP - Motivace

- Programový kód rozsáhlejších soudobých systémů je složitý, nepřehledný, nesnadno udržovatelný.
- U systémů jsou často implementovány mimofunkční požadavky: protokolování, zabezpečení, optimalizace.
- Pokrytí těchto požadavků jde napříč s požadavky funkčními - současné splnění vede nezřídka ke kombinatorické explozi a (téměř) exponenciálnímu nárůstu velikosti kódu.

- Kód je nečitelný a ještě obtížněji udržovatelný.
- I rozšiřování nelze většinou provést lokálně, nezřídka je jím zasaženo více částí kódu.

AOP - Motivační příklad

Příklad převzatý z weblogu Jablok [???] Pavla Kolesnikova (typický kód aplikační logiky):

```
public class KusAplikacniLogiky extends ObecnejsiKus {
  // data tridy;
  // jina pomocna data;

  // pretizeni rodicovskych metod

  public void provedNecoPodstatneho () {
    // autentizace
    // autorizace

    // dalsi nezajimavy kod
    // logovani zacatku operace

    // vlastni aplikacni logika — konecne!

    // logovani ukonceni operace
    // treba jeste neco
  }
}
```

- stále se opakující úkony na začátku před vlastní realizací "užitečné práce" a po ní
- aplikační logika tvoří jen zlomek rozsahu kódu

AOP - Principy

Překlad článku Graham O'Regana Introduction to Aspect-Oriented Programming [http://www.onjava.com/lpt/a/4448] na onjava.com nastiňuje hlavní principy:

- AOP umožňuje přidat do statického OO modelu programu (třídy) dynamické aspekty - např. ovlivňovat (vstupovat do) volání metod.
- Např. servlet očekává vstup z webového formuláře, naváže data z formuláře do vytvořeného datového objektu, ten zpracuje aplikační logikou a výsledek prezentuje.
- Kromě toho ale musí řešit:

- ošetření výjimek
- zabezpečení přístupu
- protokolování

Kontejnery a aplikační servery

Kontejnery a aplikační servery

Kontejnery a aplikační servery

Spring framework - podpora aplikační logiky

Klíčové pojmy

- Inversion of Control (IoC)
  - návrhový vzor, o instanciaci a správu objektů aplikace se stará třetí strana - Spring
- Dependency Injection (DI)
  - návrhový vzor, zajišťující provázání závislých objektů
- Plain Old Java Object (POJO)
  - označení objektů, které nejsou žádným způsobem závislé na aplikačním rámci, který je spravuje, takové objekty jsou snadno znovupoužitelné v různých prostředích - ve webové aplikaci, v EJB aplikaci, v testovacím prostředí, atd.

Koncepce

- Spring poskytuje specifické služby objektům aplikační (business) vrstvy
- podobný podpůrný rámec dosud chyběl (mimo EJB)
- Spring se zaměřuje na správu jednotlivých objektů, nikoliv celých komponentpod
- pora zajištěna primárně prostřednictvím IoC a AOP
- Spring umožňuje správu POJO objektů (EJB nikoliv)
- spravované objekty musí mít formát JavaBeans - bezparametrický konstruktor a set/get metody

Poskytované služby

- správa životního cyklu objektů
- jednotný procedurální i deklarativní transakční management
- jednotný způsob konfigurace aplikace v době nasazení
- DI - Spring podporuje setter a custructor DI
- pokročilá procedurální i deklarativní správa zabezpečení aplikace (Acegi Security)
- pooling objektů

Pod pokličkou

- Kontejner rámce Spring (nazývaný také aplikační kontext) při svém startu načte definiční soubor. Na základě definic v něm uvedených vytvoří pomocí reflexe specifikované objekty, vzájemně je provaže a tuto síť objektů spravuje po dobu běhu aplikace.
- Definiční soubor:
  - v drtivé většině případů je používán XML soubor
  - lze použít i soubor ve formátu properties

Příklad obsahu definičního souboru

```
<beans>
    <bean id="nejakaFirma" class="cz.neco.Zamestnavatel" />

    <bean id="franta" class="cz.neco.Zamestnanec" >
        <property name="jmeno"><value>František Novák</value></property>
        <property name="zamestnavatel" ><ref bean="nejakaFirma" /></property>
    </bean>
</beans>
```

Co provede Spring

V tomto případě by kontejner prostřednictvím reflexe provedl kód ekvivalentní tomuto:

```
Zamestnavatel nejakaFirma = new Zamestnavatel();
Zamestnanec franta = new Zamestnanec();
Franta.setJmeno = "František Novák";
Franta.setZamestnavatel(nejakaFirma);
```

Příklad konfigurace

Pojmenujeme-li tento definiční XML soubor "kontext.xml" a umístíme-li jej do classpath, pak kontejner nastartujeme například takto:

```
ApplicationContext context = new ClassPathXmlApplicationContext("/kontext.xml");
```

Vytvořeného zaměstnance s id "franta" získáme z kontejneru voláním:

```
Zamestnanec z = (Zamestnanec)context.getBean("franta");
```

Pomocí dalších deklarací v tomtéž XML souboru lze definovat, jakým způsobem budou dříve vyjmenované služby poskytnuty našim objektům. Naše třídy se tedy nemusí například transakčním či bezpečnostním managementem zabývat, vše zajistí Spring.

Užitečné linky

- stránky projektu [http://springframework.org/]
- obsáhlý úvod do Springu napsaný Rodem Johnsonem, autorem Springu [http://www.theserverside.com/articles/article.tss?l=SpringFramework]
- jednoduchá aplikace krok za krokem [http://www.springframework.org/docs/MVC-step-by-step/Spring-MVC-step-by-step.html]

Java Management extension (JMX)

Co je JMX

- JMX je rozhraním definujícím strukturu a chování jednotlivých prvků systému schopného snadné a standardizované správy, monitoringu....
- JMX je v současnosti řadou výrobců (i open-source vývojářů) považováno za nejlepší rozhraní pro správu takových systémů.
- JMX je formálně výsledkem práce JSR003.

Co řídí JMX

Přes JMX se řídí:

- moduly

- kontejnery, v nichž jsou moduly hostovány/provozovány
- zásuvné moduly (plug-ins)

Principy JMX

- Komponenty jsou v JMX deklarovány jako _lužby MBean_(MBean services).
- JMX následně umožní zavedení a správu těchto komponent.

Který objekt (komponentu) jako JMX?

Vhodnými kandidáty na JMX (MBeans) jsou takové komponenty, které:

- Interagují s (jsou ovládány) objekty z jiných vrstev aplikace.
- Objekt má stav, který má být sledován/řízen.
- Objekt je konfigurován nezávisle při startu/za běhu aplikace.

Úrovně JMX modelu

Model systému řízeného JMX zahrnuje:

Instrumentation	zdroje, jež jsou spravovány
Agents	kontrolery (ovladače) těchto zdrojů
Distributed Services	služby, jimiž (koncové) aplikace pro správu komunikují s výše uvedenými agenty

Ovládané zdroje

Přes JMX se může řídit prakticky cokoli v javovém systému:

- (celou) aplikaci
- komponentu služby
- zařízení

Jak se zdroje ovládají

JMX ovládá zdroje prostřednictvím tzv. _rapperu_ který:

- vystavuje (zpřístupňuje) vlastnosti spravovaných objektů
- díky tomu může externí, standardizovaný nástroj, přistupovat k spravovanému objektu

Jaké podoby může wrapper nabývat?

MBean

MBean javový objekt implementující jedno či více standardních MBean rozhraní a je konstruován podle odpovídajících návrhových vzorů

Co obsahují/zpřístupňují rozhraní MBean

- hodnoty atributů přístupných prostřednictvím jména atributů
- operace, které lze volat
- notifikace událostí, které zde mohou vzniknout
- konstruktory MBean třídy

Typy MBean

Standard MBean	pouze odpovídají JavaBean konvencím a staticky definovaným (JMX) rozhraním pro správu
Dynamic MBean	...
Open MBean	...
Model MBean	...

Aplikační rámec Tammi - případová studie

Charakteristika Tammi

- Komplexní (webový) aplikační rámec
- dostupný na http://tammi.sf.net

- Jedna z nejucelenějších aplikací JMX - používá se zde "na všechno"

Příklad jednoduché komponenty typu MBean

Komponenta koncipovaná jako MBean je často vytvářena takto:

1. Je vytvořen (nebo existuje) "běžný" bean s požadovanou funkcionalitou.
2. Je sestaveno rozhraní typu MBean, které bude navenek reprezentovat tuto funkcionalitu.
3. Je sestavena třída (wrapper, adaptér), který rozšiřuje původní bean a implementuje toto rozhraní.

FibonacciCounter.java - původní komponenta

CounterMBean.java - rozhraní

Counter.java - třída implementující rozhraní

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"
```

Řízení a sledování aplikací - protokolování

Protokolování (logging)

Protokolování je základní činností sledující běh software v

- testovacím i
- ostrém nasazení.

Protokolování - výhody

Protokolování má oproti klasickým přístupům

- System.out.println nebo o něco lepším
- System.err.println

jasné výhody:

- snadná konfigurovatelnost
- nezaměnitelnost s jinými (neladicími) výstupy programu
- možnost vazby na zasílání zpráv mailem, ukládání do souborů, databáze

Protokolování - možnosti v Javě

V zásadě dva možné přístupy:

- použít standardní API
- použít API daného aplikačního prostředí (ap. serveru)

Standardní API je však často ap. serverem také podporováno a má jasné výhody:

- je známé, existuje široká komunita se zkušenostmi
- obsluhu obvykle již sami známe

Protokolování - API

Existuje několik "standardních" protokolovacích API:

- od Java 1.4: balík java.util.logging
- již dříve nezávislé open-source řešení log4j WIKIPEDIA The Free Encyclopedia [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=log4j]

Obě řešení jsou srovnatelná, log4j je o něco elegantnější a propracovanější. Nejlépe (pokud to stačí) je použít zastřešujícího API:

- balík Apache Commons Logging

Protokolování - příklad

Existuje několik "standardních" protokolovacích API:

- od Java 1.4: balík java.util.logging
- již dříve nezávislé open-source řešení log4j WIKIPEDIA The Free Encyclopedia [http://cs.wikipedia.org/wiki/Speci%C3%A1ln%C3%AD:Search?search=log4j]

Obě řešení jsou srovnatelná, log4j je o něco elegantnější a propracovanější. Nejlépe (pokud to stačí) je použít zastřešujícího API:

- balík Apache Commons Logging