Kapitola 1. Přednáška 7 - Design by Contract, Aspect Oriented Programming, Inversion of Control.

Obsah

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`

Postup práce s jass:

- stažení a instalace balíku z http://csd.informatik.uni-oldenburg.de/~jass/
- vytvoření zdrojového textu s příponou `.jass`, javovou syntaxí s použitím speciálních komentářových značek
- takový zdrojový text je přeložitelný i normálním překladačem **javac**, ale v takovém případě ztrácíme možnosti jass
- proto nejprve `.jass` souboru převedeme preprocesorem jass na javový (`.java`) soubor
- ten již přeložíme **javac** a spustíme **java**, tedy jako každý jiný zdrojový soubor v Javě
- z `.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]
- jass homepage [http://csd.informatik.uni-oldenburg.de/~jass/]
- úvodní materiálek [http://www.inf.fu-berlin.de/lehre/SS01/VIS/Dokumente/Vortraege/junit.pdf] k použití junit (v němčině, jako PDF)

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

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 ne-

	závisle; 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).