



profinit.
P R O F E S S I O N A L S I N I T



Testování a QA



Agenda

- Opakování
- Manuální testování
- Automatické testování
- Antipatterns
- Testování testů
- Pokud zbyde čas
 - Užitečné nástroje
 - Cobertura
 - DbUnit
 - Testovací frameworky pro (nejen) Javu
 - JUnit
 - TestNG



Opakování - pojmy

- Verifikace a validace
- Kvalifikační a akceptační testování
- Testování vs. QA
- Test Case (testovací případ)
- Test Scenario (testovací scénář)
- Testament
 - To sem nepatří :-)

Opakování - taxonomie

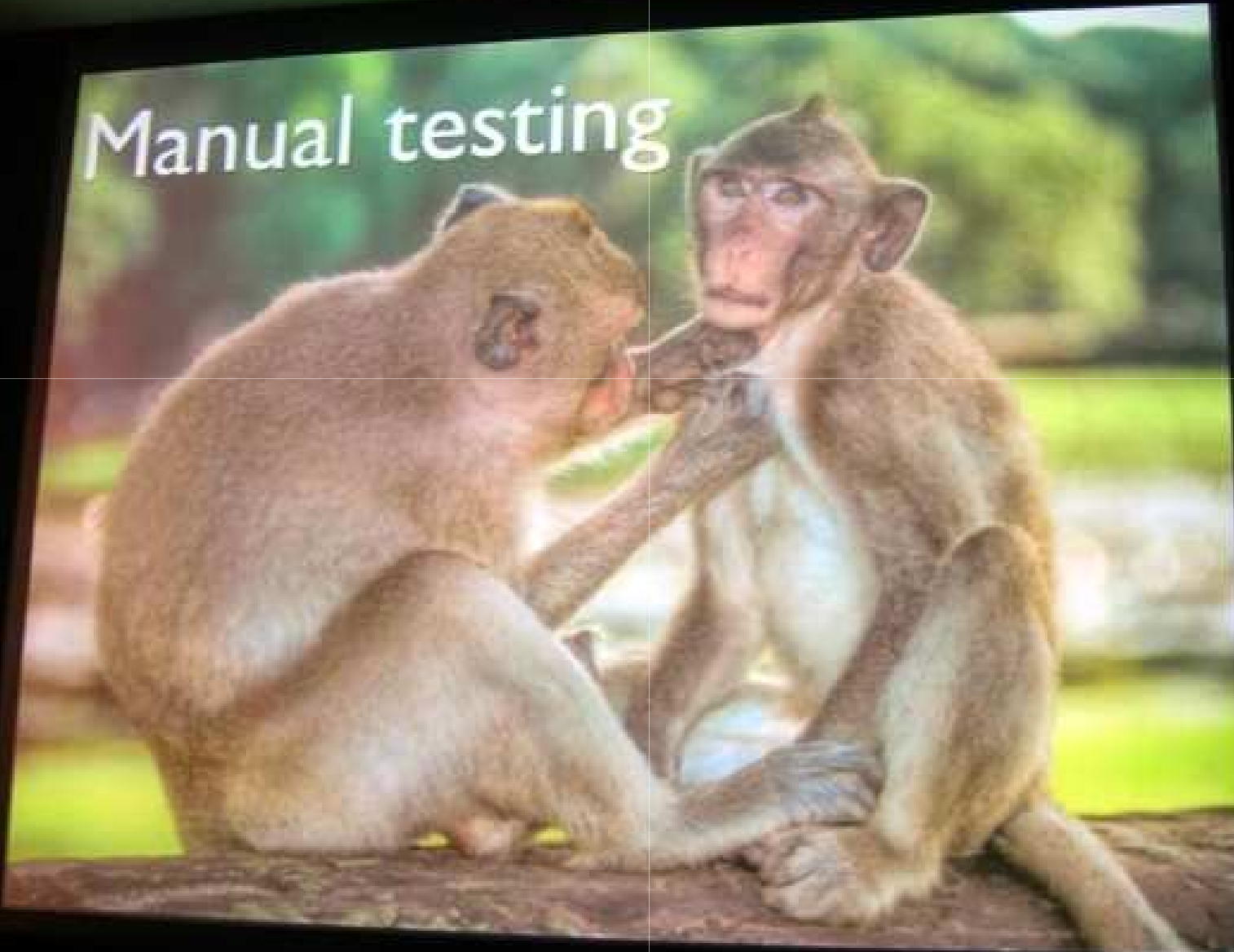
- Black box vs. white box
- Manuální vs. automatické
- Integrační vs. unit
- Kvalifikační, akceptační
- Regresní
- Výkonnostní, zátěžové
- Smoke
- Usability
- Lokalizační testování



- Fuzzy testing



Manual testing



Manuální vs. automatické

- Vzájemně se doplňují
- Většinou není možné (ekonomické) plně pokrýt aplikaci automatickými testy.
- Nám se osvědčilo:
 - Před prvním nasazením se aplikace důkladně prokliká.
 - Vytvoří se regresní testy (testují aplikaci z GUI)
 - Selenium, HttpUnit, Autolt
 - Vytvoří se (během vývoje) unit testy pokrývající „core“ funkcionalitu (výpočty, práce s daty ...)

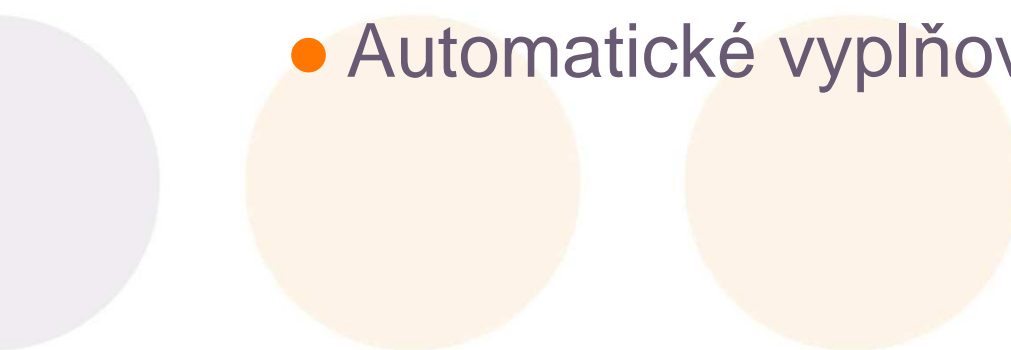


Manuální testování v praxi

- **Není dobrý nápad provádět testy nad produkční databází**
 - => Nutnost testovacího prostředí
- **Testovací prostředí by mělo být co nejpodobnější reálnému (HW, SW)**
 - Drahé – často se sahá ke kompromisům
- **Testovací prostředí a vývojové prostředí by měly být odděleny.**
 - Jinak se testy mohou chovat nedeterministicky.
 - Vývojáři a testéři se mohou vzájemně rušit.
 - Oddělení může být časové (od teď už se nevyvíjí, ale testuje)



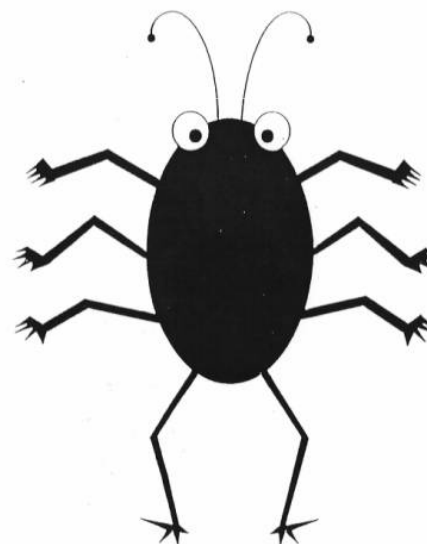
Co se nám osvědčilo

- Neoddělovat vývojáře a testery
 - Vývojáři již při vývoji myslí na testovatelnost
 - Nevzniká bariéra mezi vývojáři a testery
 - Efektivnější alokace lidí.
 - U velmi velkých systému toto neplatí
 - Využívat automatické testy i pro manuální testování
 - Automatické vyplňování formulářových polí
- 

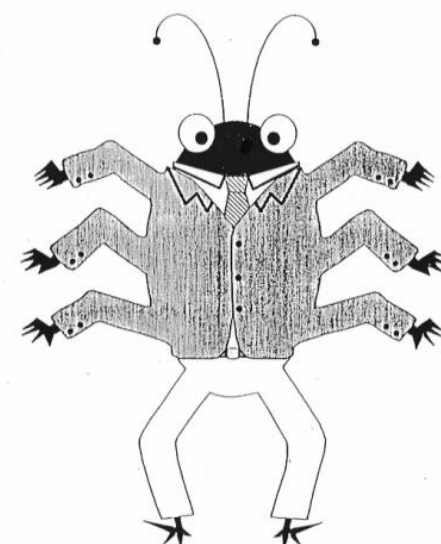


Bugzilla

- Bug tracking systém
 - Zřejmě nepoužívanější (v Profinitu tvoří páteř IS :-)
- Uchovává historii bugů – i po letech lze zjistit, proč se udělalo to či ono.
- Životní cyklus chyby
 - New (někdo nahlásí chybu)
 - Assigned (ujme se jí vývojář)
 - Fixed (chyba je opravena)
 - Verified (interní přetestování)
 - Closed (ověření zákazníkem).
- Umožňuje připojovat soubory
 - logy, screenshoty ...



BUG



FEATURE



Co se nám osvědčilo (2)

● Workflow

- Před dodávkou si vývojáři rozdělí všechny bugy ve stavu FIXED s targetem rovným dodávané verzi.
 - Nikdo netestuje chyby, které sám opravoval.
- Po otestování -> stav VERIFIED (nebo REOPEN)
- Zákazník po dodávce přetestovává -> CLOSED





Klasické problémy

- Neznalost kontextu zákazníka.
 - Například máte přetestovat opravu následující chyby: „V pojištění STR se u Doložky 22 špatně aplikuje pro-rata.“
 - Může pomoci Bugzilla (nebo kolegové).
- Chyba objevená při testování (nebo hůř – v produkci) se nedá v testovacím prostředí reprodukovat.
 - Syndrom WORKSFORME
 - Tady je každá rada drahá :-)



Unit testy

- Testují „jednotku“ (unit) nezávisle na ostatních
 - unit většinou = třída.
- Mockování ostatních (asociovaných) tříd.
 - Simulujeme jejich chování => testování třídy nezávisle na ostatních
 - Typicky – připojení k databázi
 - Vede k výraznému zrychlení
 - Nástroje: jMock, EasyMock





Unit testy (2)

- Specifická forma dokumentace kódu
 - Z testu lze vyčíst, jak daný kód použít
- Dá se na ně dívat jako na rozšíření kompilátoru
 - Kontrola sémantiky
- Frameworky
 - JUnit, NUnit, ...
 - TestNG (i integrační testy)



Doporučení pro psaní testů

- Testy psát co nejkratší – testující jen jednu funkčnost.
- Minimálně dva testy na úspěch (různé parametry)
- Testovat mezní hodnoty a nepřípustné hodnoty parametrů.
 - Hlavně pozor na null / 0
- Testy jsou také kód => dodržovat štabní kulturu.
 - Dokumentace, komentáře, pojmenování proměnných, ...

Odbočka – Co musí umět framework pro unit testy?

- Spouštět testy nezávisle na sobě
 - Pokud vývojář závislost explicitně nenadefinuje
 - Zpravidla realizováno pomocí `setUp` a `tearDown` metody
- Podporovat ověřování předpokladů
 - Například – testovaná metoda nevrátila null.
 - Zpravidla se realizuje pomocí metod `assertXXX`
- Přehledně vývojáři zobrazit výsledky testů
 - Musí být možné rychle určit příčinu selhání

Odbočka – Co by měl umět framework pro unit testy?

- Spustit jen některé testy
 - Například jen ty, co selhaly
 - Spuštění testů musí být snadné a rychlé
 - Jinak to vývojáři nebudou dělat
- Spouštět testy ve více vláknech
 - Testování bezbečnosti vůči přístupu z více vláken
- Timeouty testů – kvůli zacyklení či deadlockům
- Měl by být snadno integrovatelný s nástroji pro automatický build (CruiseControl, Hudson ...)



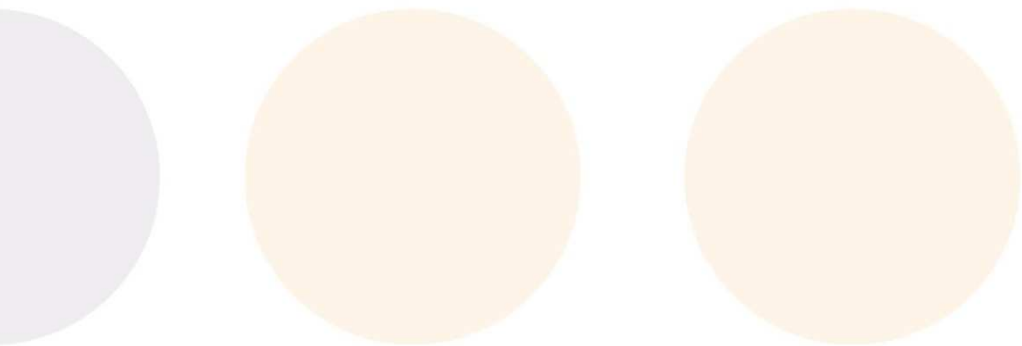
Automatické testování

- Čím jsou testy podrobnější, tím nákladnější je jejich údržba.
 - Tím větší tlak na „dočasné“ vyřazení rozbitých testů
- Čím déle testy trvají, tím menší je ochota vývojářů je pouštět.
 - Dá se řešit nočními buildy
 - Vysoká granularita spouštění testů
- Někdy je integrace testů do projektu netriviální.
 - Testování J2EE aplikací v kontejneru.



Automatické testy

- Automatické testy většinou zlepšují design aplikace.
 - Vývojáři jsou nuceni programovat s ohledem na snadnou testovatelnost – loose coupling.
- U složitějších aplikací usnadňují ladění
 - Nemusí se debuggovat přes UI.





Klasické chyby

- Odkládání tvorby testů „až bude čas“
- Vývoj testů ad-hoc – bez jakéhokoliv plánu, rozmyšlení, architektury ...
 - Někdy se vyplatí pojmout tvorbu testů jako samostatný projekt.
- Duplikace kódu, málo komentářů či absence dokumentace.
- Testy jsou, ale nikdo je nepouští. Testy se pouští, ale nikdo nekontroluje výsledky.

Cvičení – Jak otestovat

```
public static Integer fib(Integer n) {  
    if (n < 1 ) {  
        throw new NejakaVyjimka( )  
    } else if (n < 3) {  
        return 1;  
    } else {  
        return fib(n-1) + fib(n-2);  
    }  
}
```



Testování testů?

- Jak poznat, že testy k něčemu jsou?
 - Teoretici – i testy se musí testovat.
 - Do kódu záměrně zaneseme chyby a zkoumáme, zda je testy odhalí
 - Používá se jen u mission-critical systémů
 - Praxe – měření pokrytí kódu
 - Pokrytí metod
 - Pokrytí řádků kódu
 - Pokrytí větví.





Pokrytí kódu - Cobertura

- Poskytuje vývojáři informace o:
 - Pokrytí balíčků
 - Pokrytí tříd
 - Pokrytí metod
 - Pokrytí větví
- Reporty lze procházet až na úroveň zdrojových kódů
 - Přehledně znázorněno, kolikrát byl daný řádek navštíven.



Cobertura - Nevýhody

- Svádí k „uctívání“ pokrytí
 - 100 % pokrytí kódu neznamená, že je program bez chyb
 - Někteří projektoví vedoucí mají tendence stanovovat minimální procento pokrytí kódu.
=> Easy testy (testují se gettery, settery ...)
- Někdy obtížně integrovatelná s projektem
 - Například pokud testy běží v kontejneru
- Pouze pro automatické testy

Cobertura - Ukázkový report

Screenshots pochází z ukázkového reportu na adrese <http://cobertura.sourceforge.net/sample/>



Antipatterns



Jen jeden „happy path“ test

```
public class Factorial {  
    public int eval(int cislo) {  
        return (cislo != 1) ?  
            cislo * eval(cislo - 1) : 1;  
    }  
}
```

```
public void testEval() {  
    Factorial fact = new Factorial();  
    int vysledek = fact.eval(3);  
    assertEquals(6, vysledek);  
}
```



Kde je problém?

- Test projde i pro následující implementaci:

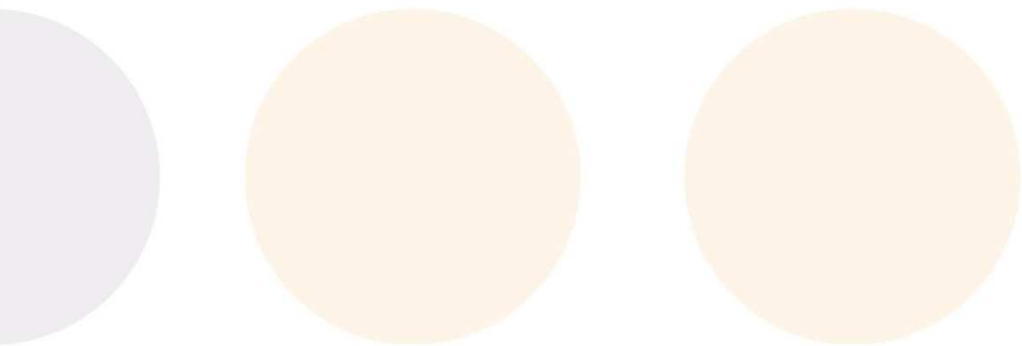
```
public class Factorial {  
    public int eval(int cislo) {  
        return (cislo != 1) ?  
            cislo + eval(cislo - 1) : 1;  
    }  
}
```

- Ale i předchozí implementace je chybná
 - eval(0);
 - Nutno testovat mezní hodnoty



Easy tests

- Testují se snadno testovatelné podpůrné metody, ale ne hlavní logika.
 - Gettery, settery, toString
- Problém hlavně u nezkušených programátorů
 - Časté rovněž při direktivním stanovení určitého pokrytí kódu testy.



Spoléhání na konkrétní implementaci

- Někdy je to vhodné, ale zpravidla je lepší se tomu vyhnout.

```
public class Data {  
    private Collection prvky = new ArrayList();  
  
    public void pridej(Object prvek) { prvky.add(prvek); }  
    public Collection getPrvky() { return prvky;}  
}  
  
public void testPridej () {  
    Data data = new Data();  
    data.pridej(0); // spolehame na autoboxing  
    assertEquals(0, ((ArrayList) data.getPrvky()).get(0));  
}
```



Komplexní testy

- Testy by měly být co nejkratší a vždy testovat jen jednu věc
- Unit testy slouží i jako dokumentace – jak daný kód použít
- Dlouhé a komplexní testy jsou nepřehledné a hůře se udržují
 - Pokud je rozdělíme do více testů => můžeme tyto testy spouštět samostatně.
- Obtížnější analýza výsledků, pokud některé testy testují více funkcí najednou.



Polykání výjimek

```
...  
try {  
    zavolejTestovanouMetodu(parametry);  
    fail(„Očekávána výjimka“);  
} catch (Throwable e) {  
    // OK  
}  
...
```

- Problém – projde, i když metoda vyhodí například `OutOfMemoryError`.
 - Pokrývá více výjimek než jen tu testovanou



Kde je chyba?

```
/** Testuje, zda metoda dokaze pracovat  
    s null parametry. Pokud ne, test  
    zkonci s chybou. */  
public void testScitaniSmoke {  
    // null na vstupu nesmi zpusobit NPE  
    secti(null,null);  
    secti(null,0);  
    secti(0, null);  
}
```


Cvičení – napište TCs:

- Specifikace říká:

2.1 Autentizace

Systém bude podporovat dva druhy autentizace – doménovou a autentizaci pomocí přihlašovacího jména a hesla.

2.1.1 Doménová autentizace

Pokud je uživatel v doméně určené konfigurací systému, je autentizován automaticky bez nutnosti zadávat přihlašovací jméno nebo heslo.

2.1.2 Formulářová autentizace

Anonymní uživatel na kterého se nevztahuje bod 2.1.1 je při přístupu na libovolnou stránku vyžadující neanonymní úroveň oprávnění přesměrován na Login stránku. Zde musí zadat svoje přihlašovací jméno a heslo.

Přihlašovací údaje se poté ověří proti databázi a pokud je autentizace úspěšná, je uživatel přesměrován na původně požadovanou stránku. Při neúspěšné autentizaci se zobrazí chybová hláška.

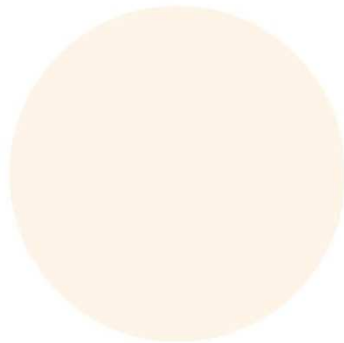
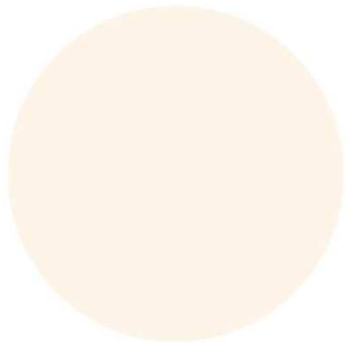
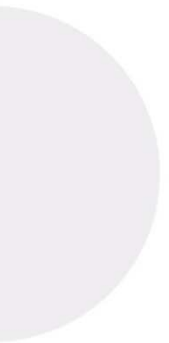


Cvičení – plán testů

- Napište plán testů pro následující projekt:
 - E-shop pro firmu zabývající se prodejem relaxačních pomůcek pro akvarijní ryby.
 - Rozsah projektu – 100 MD, 4 měsíce
 - Staví na firemním frameworku.
- Nejprve vytvořte osnovu plánu
 - Pak si ukážeme šablonu
- Poté vyplňte konkrétní údaje (ty nejzásadnější)
 - Není třeba zabývat se TCs

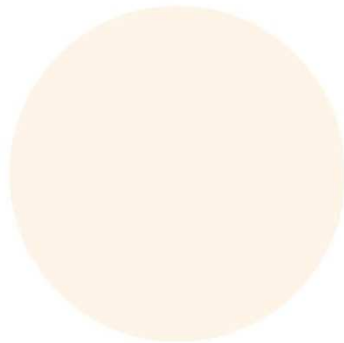
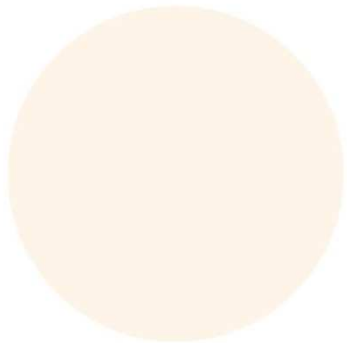


Optional slidy





DbUnit





DbUnit - motivace

- www.dbunit.org
- Chceme testovat aplikaci pracující s databází
 - Potřebujeme databázi uvést do předem známého stavu.
 - Nahrát testovací data
 - Potřebujeme ověřit, že databáze byla modifikována požadovaným způsobem.
 - Například porovnání dumpů
- Typické použití – testování DAO vrstvy
- Netradiční použití – nahrávání konfigurace z Excelu.



DbUnit - princip

- V XML (nebo XLS) jsou testovací data.
- Na začátku testování se data nahrají do DB
 - Je možné zvolit, zda se mají dotčené tabulky jen updatovat nebo vyprázdnit a znovu nahrát.
- V průběhu testování se stav databáze ověřuje proti „snapshots“.
- Pokud testy běží v transakci, je možné DB na konci uvést do původního stavu (rollback).

DbUnit – příklad dat

```
<dataset>
  <kategorie_zbozi id="1" nazev="Kategorie1" nadkategorie="null"
    poradi="0" />

  <osoba id="1" dtype="PravnickaOsoba" ulice="Tychonova 2"
    mesto="Praha" psc="16000" nazev="Profinit" ico="25650203" />

  <objednavka id="1" stav="2" cena="10000" zakaznik="1" />

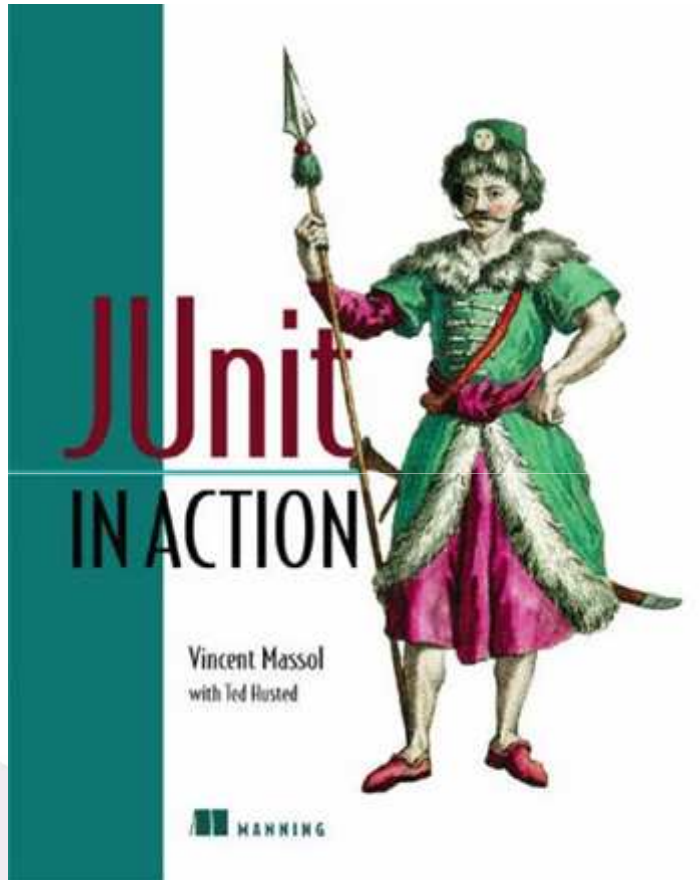
  <zbozi id="1" nazev="Pracka" cena="10000"/>
  <zbozi id="2" nazev="Pracka se susickou" cena="15000"/>

  <polozka id="1" obj_id="1" zbozi="4" kusu="1" cena="10000" />
  <polozka id="2" obj_id="1" zbozi="5" kusu="1" cena="10000" />

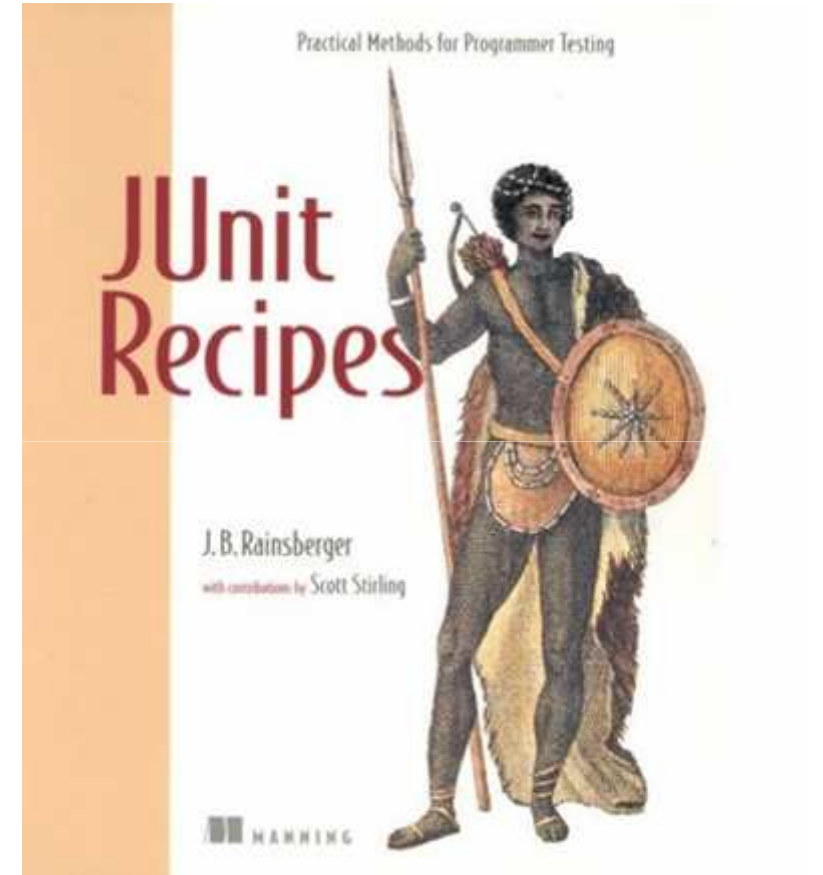
  <zbozi_v_kategoriich kategorie="1" zbozi="1" />
  <zbozi_v_kategoriich kategorie="1" zbozi="2" />
</dataset>
```

DbUnit – příklad použití

```
public void testMe() throws Exception {  
    // Execute the tested code that modify the database here  
    // Fetch database data after executing your code  
    IDataset databaseDataSet =  
        getConnection().createDataSet();  
    ITable actualTable =  
        databaseDataSet.getTable("TABLE_NAME");  
    // Load expected data from an XML dataset  
    IDataset expectedDataSet =  
        new FlatXmlDataSet(new File("expectedDataSet.xml"));  
    ITable expectedTable =  
        expectedDataSet.getTable("TABLE_NAME");  
    // Assert actual database table match expected table  
    Assertion.assertEquals(expectedTable, actualTable);  
}
```

JUnit





JUnit - úvod

- www.junit.org
- Nejstarší a nejpoužívanější unit testing framework
- Podporován prakticky všemi Java IDE
- JUnit3
 - Všechny testy musí dědit od třídy TestCase
 - Metody musí začínat na „test“
 - Metody nesmí mít parametry
 - Obchází se pomocí atributů testovacích tříd a Antu.



JUnit4

- Zpětně kompatibilní s JUnit3
- Podpora anotací
 - Silně inspirovány TestNG
 - Metody již nemusí začínat na „test“
 - Stačí je označit anotací @Test
- Testy již nemusí dědit od TestCase
 - TestRunners – jak se mají testy spouštět
 - Integrace s IDE, Springem, ...



JUnit4

- setUp a tearDown metody nahrazeny anotacemi @Before a @After
- Možnost specifikovat očekávanou výjimku
 - Usnadňuje tvorbu testů selhání
 - @Test(expected = NullPointerException.class)
- Spuštění testů z příkazové řádky (podobá se JUnit3):

```
java org.junit.runner.JUnitCore.runClasses(TestSuite1.class, ...);
```

- Třída `TestSuite1` musí definovat statickou metodu `suite`:

```
public class TestSuite1 {  
    public static junit.framework.Test suite() {  
        return new JUnit4TestAdapter(Test1.class);  
        ...  
    }  
}
```



JUnit – ověřování předpokladů

- Slouží k tomu metoda `assert` a její varianty
- Pokud předpoklad neplatí vyhodí se speciální výjimka.
- Lze specifikovat text, který se má při selhání zobrazit.
 - Doporučuji používat vždy
- `assertEquals` / `assertNotEquals`
- `assertTrue` / `assertFalse`
- `assertNull` / `assertNotNull`





Příklad – Junit3

```
public class TestCalculator
    extends TestCase {

    public void testAdd() {
        Calculator calc = new Calculator();
        double result = calc.add(10, 50);
        assertEquals(60, result);
    }
}
```



Příklad – Junit4

```
public class TestCalculator {  
    private Calculator calc;  
  
    @Before  
    public void init () {  
        calc = new Calculator();  
    }  
  
    @Test  
    public void add() {  
        double result = calc.add(10, 50);  
        assertEquals(60, result);  
    }  
}
```




Diskuse

- Komentáře
- Otázky
- Připomínky
- Upřesnění
- Poznámky
- ...

