<embed/it>

Testování

PA165
11. 12. 2012
Petr Adámek

Testování aplikací

- Ověřuje soulad implementace se specifikací a s očekáváním zákazníka.
- Je důležitou součástí procesu řízení kvality vývoje software
- Na rozdíl od formální verifikace neumožní odhalit všechny potenciální chyby

Základní pravidla

- Testy by měly být reprodukovatelné.
- Testy by měly být deterministické, tj. měly by mít na začátku vždy stejné vstupní podmínky.
- Testy by měly být nezávislé, tj. nebýt ovlivněny ostatními testy.
- Testy by měly být levně opakovatelné.

Druhy testování podle metody

- **Ruční testování:**
  - nízké vstupní náklady;
  - drahé opakování;
  - obtížné zajištění reprodukovatelnosti, determinismu a nezávislosti

- **Automatizované testování:**
  - vysoké vstupní náklady;
  - levné opakování;
  - snadné zajištění reprodukovatelnosti, determinismu a nezávislosti.

Druhy testování podle cíle

- Jednotkové testování
- Integrační testování
- Funkční testování
- Akceptační testování
- Testování výkonu a škálovatelnosti
- Testování uživatelské přívětivosti
- Testování bezpečnosti

Jednotkové testování

- U jednotkového testování se snažíme otestovat jednotlivé komponenty vyvíjeného systému na té nejnižší úrovni.
- Jednotlivé testované komponenty by měly být izolovány od svého okolí, aby se zamezilo vlivu tohoto okolí na testovanou komponentu.
- Interakce s okolím je simulována pomocí falešných objektů, které simulují chování okolí v konkrétním testovacím scénáři (viz Mock Objekty).
- Čím je lépe provedená dekompozice, tím je snadnější jednotkové testování.

Nástroje pro jednotkové testování

- JUnit
- TestNG

Příklad

```
public class CalculatorTest {

    private Calculator c;

    @Before
    public void setUp() {
        c = new Calculator();
    }


    @Test
    public void testDivide() {
        assertEquals( 9, c.divide( 99, 10));
        assertEquals(10, c.divide(100, 10));
    }


    @Test(expected = IllegalArgumentException.class)
    public void testDivideByZero() {
        c.divide(100, 0);
    }
}
```

Základní pravidla

- Výstupem testu je ANO/NE (boolean)
- Nejdříve testy, potom kód (viz XP a TDD)
- Při opravě chyby nejdříve testy, potom oprava (ochrana proti regresím)
- Triviální get/set metody se netestují
- Testujeme všechny nestandardní situace a hraniční hodnoty
- Chybové hlášky a komentáře nejsou vždy potřeba
- Testy se spouští po každé změně

Interakce s okolím

- **Komponenty by se měly testovat izolovaně. Je ale nutné nějak simulovat interakci s okolím**
- **K tomu slouží tzv. *Mock objekty*.**
- **Tyto objekty musí být typově kompatibilní se simulovanou komponentou.**
  - Dědění
  - Implementace rozhraní (vhodnější)
- **Mock objekty můžeme vytvářet ručně (pracné) nebo pomocí nástrojů**
  - Mockito, EasyMock, jMock

Příklad (ručně vytvářené Mock objekty)

```
public class CurrencyConvertorTest {
  @Test
  public void testConvert() {
    ExchangeRateTable exchangeRateTable = new ExchangeRateTable() {

      public void setExchangeRate(Currency currency, BigDecimal exchangeRate) {
        throw new UnsupportedOperationException("Not supported yet.");
      }
      public BigDecimal getExchangeRate(Currency currency) {
        return BigDecimal.valueOf(28.2);
      }
    };

    CurrencyConvertor convertor = new CurrencyConvertor(exchangeRateTable);
    Currency czk = Currency.getInstance("CZK");
    BigDecimal actualResult = convertor.convert(czk, BigDecimal.valueOf(10));
    BigDecimal expectedResult = BigDecimal.valueOf(282.0);
    assertEquals(expectedResult, actualResult)
  }
}
```

Příklad (Mockito)

```
@RunWith(MockitoJUnitRunner.class)
public class CurrencyConvertorTest {

  @Mock
  ExchangeRateTable exchangeRateTable;

  @Test
  public void testConvert() {

    when(exchangeRateTable.getExchangeRate(czk)).
      thenReturn(BigDecimal.valueOf(28.2));

    CurrencyConvertor convertor = new CurrencyConvertor(exchangeRateTable);
    Currency czk = Currency.getInstance("CZK");
    BigDecimal actualResult = convertor.convert(czk, BigDecimal.valueOf(10));
    BigDecimal expectedResult = BigDecimal.valueOf(282.0);
    assertEquals(expectedResult, actualResult)
  }
}
```

Jednotkové testování v Java EE

- **U Java EE aplikací je nutné vzít v úvahu existenci kontejneru.**
  - Testy mimo kontejner – otestuje se pouze business logika, nikoliv chování závisející na kontejneru (např. řízení transakcí, autorizace apod.)
  - Testy v kontejneru – otestuje se vše, ale tento druh testování se pro jednotkové testy moc nehodí.
- **Při testování mimo kontejner se používá koncept mock objektů, které simulují chování kontejneru.**

Jednotkové testování

- **Jak testovat vrstvu perzistence dat:**
  - Mock Objekty (snadné u JPA nebo jiných knihoven a rámců, komplikované u low-level JDBC).
  - Databáze uložená v paměti (snadné u JPA, u low-level JDBC může být problém s SQL dialektem).
- **Nezapomenout na zajištění stejných počátečních podmínek (uvést databázi vždy do stejného počátečního stavu).**
- **Co může pomoci**
  - DBUnit
  - Abstraktní DAO

Co nám ještě může pomoci

- **Nástroje pro měření míry pokrytí testy**
  - Line Coverage
  - Branch Coverage
- **Nástroje pro generování testovacích dat**
- **Rozšířené sady assert metod**
- **Atd.**

Integrační testování

- Integrační testování slouží k ověření správné interakce jednotlivých komponent a zda se sestavený systém chová tak, jak očekává specifikace.
- Viz též continuous integration

Funkční testování

- Funkční testování slouží k ověření funkcionality z pohledu koncového uživatele.
- Většinou se jedná o testování na úrovni uživatelského rozhraní.
- Rational Functional Tester – gui + web
  http://www-01.ibm.com/software/awdtools/tester/functional/index.html
- Selenium IDE – web
  http://selenium.openqa.org/
- Marathon – gui
  http://marathonman.sourceforge.net/
- Rational Robot – gui (pro staré aplikace), Rational Quality Manager, JWebUnit

Akceptační testování

- Akceptačním testováním zákazník ověřuje, zda aplikace odpovídá jeho požadavkům a představám.
- Absence akceptačního testování (případně jeho podcenění a nedostatečné provedení) téměř spolehlivě vede k pozdějším sporům a problémům.
- Zákazníci mají bohužel tendenci tuto věc velmi podceňovat a na nesoulad implementace s požadavky zákazníka se tak často přijde až v okamžiku produkčního nasazení :-(.

Testování výkonu a škálovatelnosti

- Testováním výkonu se ověřuje propustnost a odezvy systému při vysokém zatížení.
- Součástí specifikace by měla být i definice požadované propustnosti a reakčních dob systému při předepsaném zatížení.
- Rational Performance Tester (+ extensions)
  http://www-01.ibm.com/software/awdtools/tester/performance/index.htm
- Rational Service Tester for SOA Quality (funkční testy + testování výkonu)
- Jmeter
  http://jakarta.apache.org/jmeter/

Testy uživatelské přívětivosti

- Testy uživatelské přívětivosti
- V USA běžná věc, v Evropě to zatím firmy moc nedělají. Např. firma SUN Microsystems má v Praze svůj jediný Usability lab mimo území USA.
- Definice prototypu cílového uživatele.
- Výběr skupiny testovacích uživatelů (testovací vzorek).
- Testovací uživatel dostane seznam úkolů, které se snaží vyřešit bez pomoci někoho jiného.
- Jeho chování je sledováno a vyhodnocováno.
- Viz Štefkovič, M.: Použitelnost webových aplikací. (https://is.muni.cz/auth/th/166042/fi_b/)

Testování bezpečnosti

- Testování bezpečnosti ověřuje odolnost proti různým typům útoků.
- Nástroje:
- Rational AppScan – bezpečnost webových aplikací
  http://www-01.ibm.com/software/awdtools/appscan/

Otázky

?