Základní přehled testování - a Crash Course

KIV/ASWI 2015

P.Brada, P.Herout

Zajištění kvality sw (QA)

- Prevence vzniku chyb
- Detekce již zanesených chyb
  - (prevence propagace)
- Způsoby nápravy

- Verifikace – cílem je potvrdit, že SW vyhovuje zadané specifikaci
- Validace – kontrola, jestli SW vyhovuje požadavkům uživatele / zadavatele

Proč testovat

- chyby jsou běžnou součástí našeho života
  - NIST 2002: softwarové chyby stojí USA ročně USD 59,5 mld
- testování je proces odhalování těchto chyb před okamžikem, kdy mohou uškodit

Cena za opravu chyby (čísla znamenají násobky)		Fáze, kdy byla chyba detekována				
		Specifikace	Návrh	Kódování	Systémové testy	Používání
Fáze, kdy chyba vznikla	Specifikace	1	3	5 – 10	10	10 – 100
	Návrh	–	1	10	15	25 – 100
	Kódování	–	–	1	10	10 – 25

Co je to chyba

- Termíny porucha, chyba, selhání
- V širším smyslu: defekt =
  - SW nedělá, co má dle specifikace
  - SW dělá, co nemá dle specifikace
  - SW dělá něco, co není specifikováno
  - SW nedělá něco, co není specifikováno, ale asi by to dělat měl
  - SW je nesrozumitelný, špatně se s ním pracuje apod.

- Primární cíl testování = ujistit se, zda sw je takový, jaký byl zamýšlen a zda vyhovuje specifikaci
  - ten, kdo stanoví očekávaný výsledek, je tzv. orákulum (oracle) – člověk nebo počítač
  - to, co se testuje, je tzv. system under test (SUT)

Úrovně testování

- z časového hlediska (úsek v životním cyklu, kdy se testuje) lze testy rozdělit do fází

- jednotkové (Unit) – správná funkčnost nejmenších částí systému – metod a tříd
- integrační (Integration) – správné chování nových podsystémů zapojených do celku
- systémové (System) – obsáhlé scénáře procházející napříč celým systémem
- akceptační (Acceptance) – převzetí produktu zákazníkem, často podle scénářů předem domluvených oběma stranami
  - též UAT – uživatelské akceptační testy

Struktura testů / pojmy

- Testovací případ (Test Case) - popisuje jeden konkrétní případ testování
  - detailně popsané kroky k provedení
  - výčet množiny vstupů, podmínek a očekávaných reakcí
  - ověřovací kroky, zda byl test splněn, které se dají jednoznačně vyhodnotit
- Testovací skript (Test Script)
  - přepis testovacího případu nebo scénáře do programovacího jazyka
- Testovací scénář (Test Suite)
  - skupina logicky souvisejících testovacích skriptů
  - většinou prováděných v definovaném pořadí
- Záznam/zpráva o testování (Test Log/Report)
  - záznam konkrétního průběhu testů / souhrn množiny testování
  - kdo kdy provedl, jaké testy proběhly, jaké byly výsledky

Podpůrné artefakty

- Testovací data (Test Data)
  - konkrétní sada dat použitá v průběhu provádění testu
  - skutečná data nebo data vygenerovaná podle předem určených pravidel
  - součást dokumentace testování
  - ne vždy je připravuje tester sám

- Hlášení chyby (Bug Report / Anomaly Report)
  - oznámení popisující možnou chybu (viz další ASWI)
  - množina bug reportů je ve skutečnosti nejdůležitějším výsledkem testů

- Metriky testování
  - bug fixing rate = počet uzavřených chyb / počet nalezených chyb * 100
  - pokrytí kódu testy (code coverage) = LOC prošlých testy / LOC celkem

Praktické ukázky: JUnit

TECHNICKÉ PROVEDENÍ TESTŮ: JEDNOTKOVÉ TESTY

Jednotkové testy

- též testování komponent

- První fáze testů
  - dobře navržené a provedené unit testy odhalí velké množství chyb již v rané fázi => nejlevnější oprava chyb
  - kvalitu vlastního kódu si (pomocí unit testů) hlídá vývojář

- vše dělá vývojář, který svému kódu rozumí nejvíce ze všech
- nikam se nemusí reportovat
- chyba se nalezne, detekuje příčina a opraví

- automatizace => jednou se připraví, další běhy „zdarma“

xUnit

- JUnit je velmi rozšířený způsob psaní a spouštění testů
  - tvůrce Kent Beck nejprve pro Smalltalk
  - de facto průmyslový standard (NUnit, PHPunit, …)
  - *junit.jar*

- podpora v RAD a build systémech
  - generování kostry testů, grafická nadstavba, hromadné spouštění, atd.

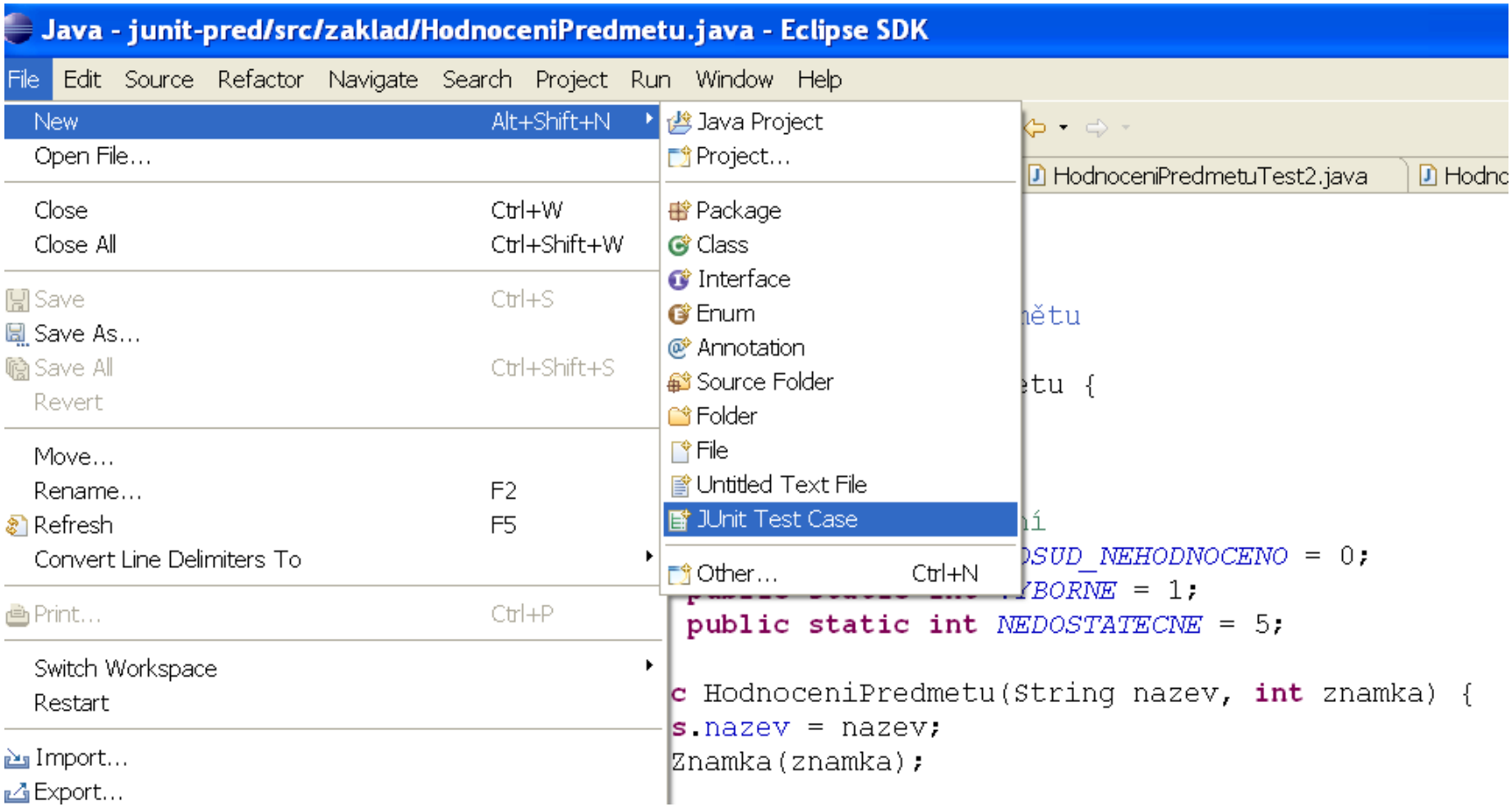
Základní konstrukce

- testovaná třída XY – beze změn (POJO)
- testovací kód
  - třída *XYTest* = test suite
  - metody *@Test testFooBarUcel*() = test cases
  - *assertMN*() = ověřovací podmínka
- testuje pouze non-private metody
- více tříd v jedné suite

Typy a parametry viz manuál JUnit

Příklad

Vygenerování kostry testu



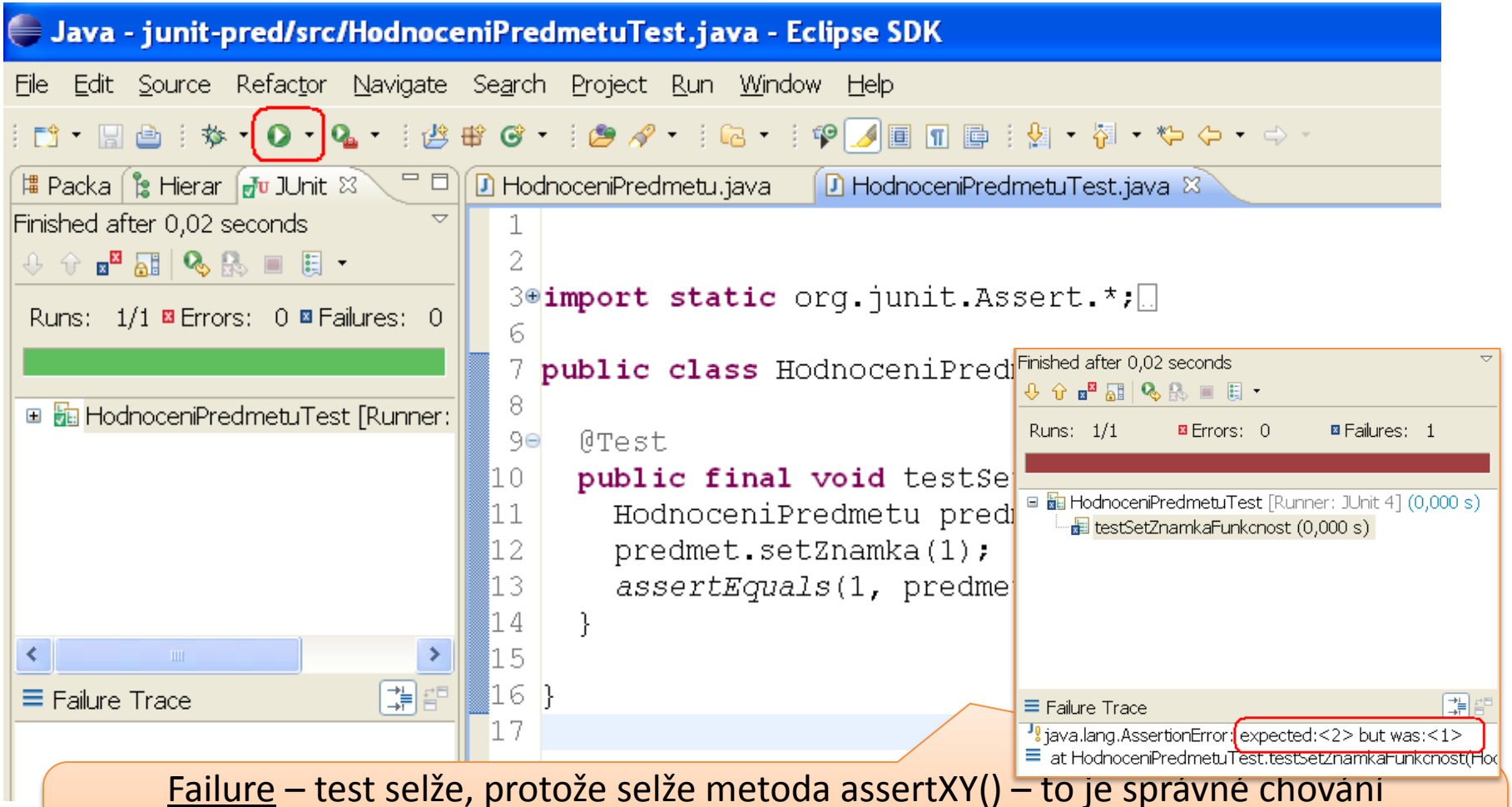
Testovaná třída

```java
public class HodnoceniPredmetu {
  private String nazev;
  private int znamka;

  // konstanty pro známkování
  final public static int DOSUD_NEHODNOCENO = 0;
  final public static int VYBORNE = 1;
  final public static int NEDOSTATECNE = 5;

  final public static String DOSUD_NEHODNOCENO_SLOVY = "N/A";

  public HodnoceniPredmetu(String nazev, int znamka) {..

  public HodnoceniPredmetu(String nazev) {..

  public String getNazev() {..

  public int getZnamka() {..

   * Nastavuje známku v rozsahu od VYBORNE do NEDOSTATECNE..
  public void setZnamka(int znamka) throws IllegalArgumentException {..

  /**
   * Zjistí, zda je předmět dosud nehodnocen
   * @return true, pokud je předmět nehodnocen
   */
  public boolean isNehodnoceno() {
    return (znamka == DOSUD_NEHODNOCENO);
  }

  @Override
  public String toString() {
    if (isNehodnoceno() == true) {
      return nazev + ": " + DOSUD_NEHODNOCENO_SLOVY + "\n";
    }
    else {
      return nazev + ": " + znamka + "\n";
    }
  }
```

Test suite

```
import static org.junit.Assert.*;

public class HodnoceniPredmetuTestMetody {
  final static String NAZEV_PREDMETU = "Matika";
  HodnoceniPredmetu predmet;

  @Before
  public void setUp() throws Exception {
    predmet = new HodnoceniPredmetu(NAZEV_PREDMETU);
  }

  @Test
  public final void testIsNehodnoceno() {
    assertTrue(predmet.isNehodnoceno());
    predmet.setZnamka(HodnoceniPredmetu.VYBORNE);
    assertFalse(predmet.isNehodnoceno());
  }

  @Test
  public final void testToStringNehodnoceno() {
    String vysledek = NAZEV_PREDMETU + ": "
            + HodnoceniPredmetu.DOSUD_NEHODNOCENO_SLOVY + "\n";
    assertEquals(vysledek, predmet.toString());
  }

  @Test
  public final void testToStringHodnoceno() {
    predmet.setZnamka(HodnoceniPredmetu.NEDOSTATECNE);
    String vysledek = NAZEV_PREDMETU + ": "
            + HodnoceniPredmetu.NEDOSTATECNE + "\n";
    assertEquals(vysledek, predmet.toString());
  }
}
```

Provedení testu



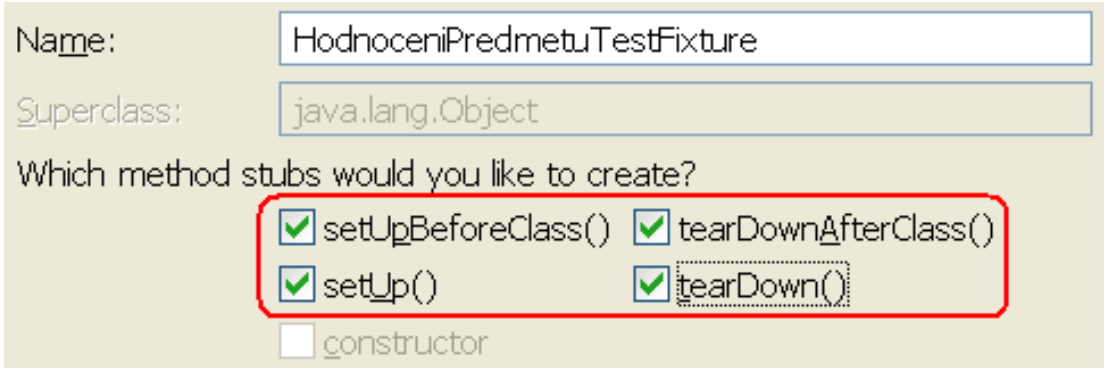
Failure – test selže, protože selže metoda assertXY() – to je správné chování

Error – test selže, protože je v něm nějaká programová chyba – je třeba opravit test

XOR pokud testujeme vyhození výjimky, pak je chybná funkčnost ohlášena právě takto

Různé rady

- Konfigurace testu
  - @BeforeClass, @AfterClass, @Before, @After



- Je vhodné převést na JUnit test:
  - cokoliv, co si kdykoliv zkusíte otestovat ručně – místo *main*()
  - chybu nalezenou zákazníkem apod.
- Typicky neděláme
  - testy pro getry a setry (v případě, že jsou jednoduché)
  - neodchytáváme run-time výjimky – nechávají se být, aby shodily test
  - jen jednu testovou třídu pro komplexní testy (dekompozice)
  - pořadí: u dobře napsaných testů nezáleží na pořadí jejich spouštění

Praktické ukázky: JUnit

TECHNICKÉ PROVEDENÍ TESTŮ: INTEGRAČNÍ A FUNKČNÍ TESTY

Integrační testy

- Metodika a nástroje na testování tříd/modulů se závislostmi
- Konflikt
  - díky vzájemným závislostem musíme mít stále komplikovanější testy zahrnující více doménových tříd najednou
  - požadavek, aby testovací případ prověřil pokud možno jen jednu funkcionalitu kódu, aby se snadno odhalilo místo chyby
- Řešení
  - stub pro chybějící/nežádoucí částí + unit test frameworky
  - díky programování do rozhraní
- jMock, EasyMock: frameworky pro tvorbu mock objektů – example based code definition

Použití mock objektu

- Deklarace
  createStrictMock
  + název, class/iface
- Definice
  expect
  + metoda, param
  andReturn
- Finalizace definice
  replay

- Injektování a
  volání v testované třídě

- Ověření využití
  verify

```
 8 import org.easymock.EasyMock;
 9 import org.junit.Test;
10
11 public class StatistikaHodnoceniTest {
12
13   @Test
14   public void testPrumerZPredmetu() {
15     // příprava dat pro mock objekt
16     List<HodnoceniPredmetu> testovaciData = n
17     testovaciData.add(new HodnoceniPredmetu('
18     testovaciData.add(new HodnoceniPredmetu('
19     testovaciData.add(new HodnoceniPredmetu('
20 //     testovaciData.add(new HodnoceniPredmetu
21
22     // příprava mock objektu
23     ISpravaDat spravaDatAtrapa =
24           EasyMock.createStrictMock("atrapaSD
25     EasyMock.expect(spravaDatAtrapa.getJedenP
26             .andReturn(testovaciData);
27     EasyMock.replay(spravaDatAtrapa);
28
```

Příklad

Funkční testy

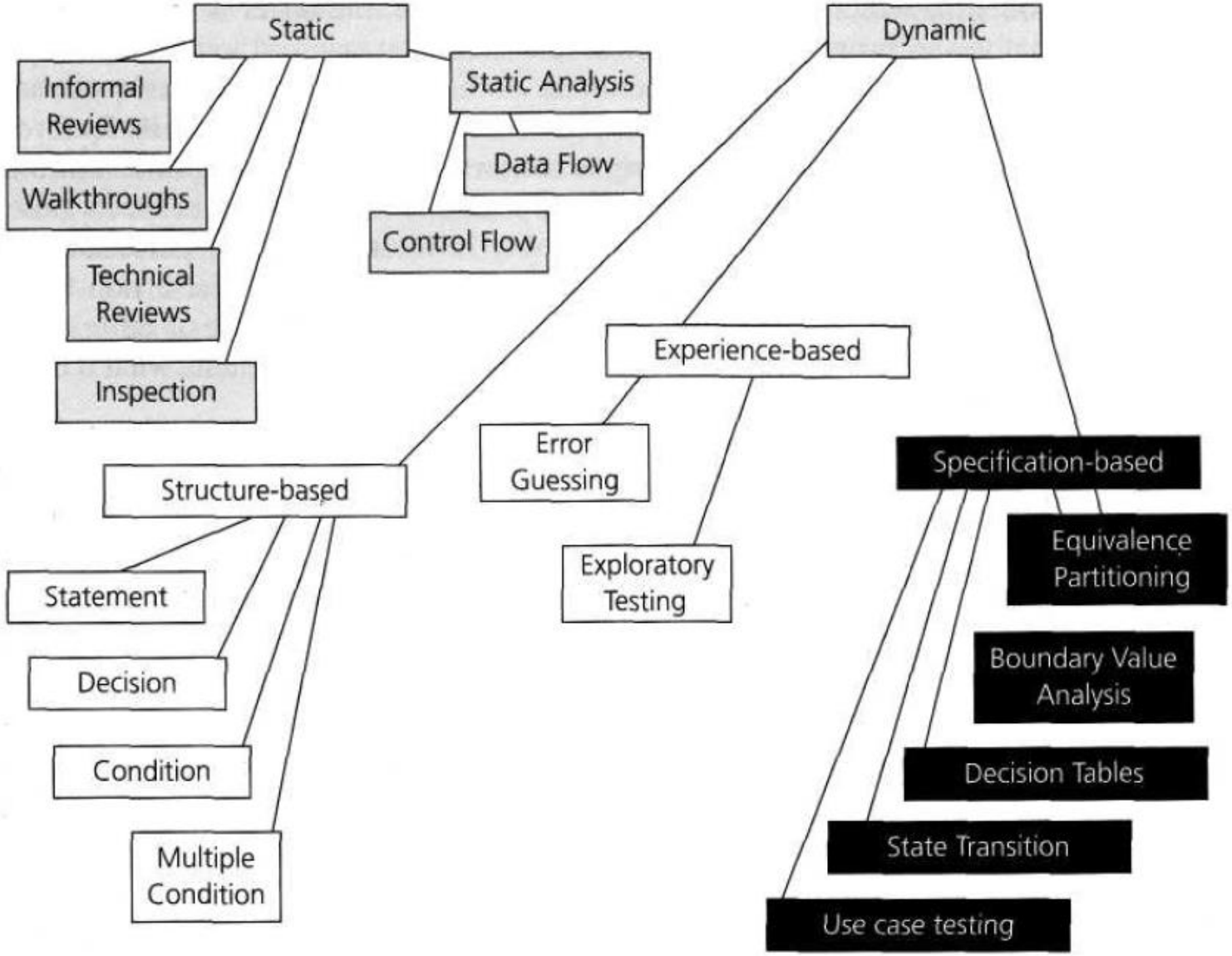
- Black-box testování celého produktu
  - uživatelské rozhraní
  - scénáře použití (use cases, user story tests)

- Konflikt
  - potřeba automatizace
  - obtížnost testování GUI
- Řešení
  - frameworky či nástroje pro definici a spouštění scénářů
  - ruční testování (důkladná definice test cases)
  - neřeší vše (UX, security, load, …)

- Selenium – nástroj pro nahrávání, definování a spouštění browser-based scriptů

POSTUPY PŘI TESTOVÁNÍ

Jak lze ověřovat sw



Důležité aspekty testování

- Co testuji?
  - black / white / gray box metody
- Jakým způsobem testuji?
  - viz předchozí obrázek
- Čím testuji? Co ověřuji?
  - metody výběru testovacích dat
  - tests-to-pass (testy splněním) = dělá co má
    tests-to-fail (testy selháním) = nedělá co nemá

- Testovací mix

Jak vybrat testovací data

- Rozdělení ekvivalentních případů
  - třída ekvivalence je množina TC, která testuje stejnou věc nebo odhaluje stejnou chybu
  - zahrnout: unikátní / prázdné / nesprávné hodnoty, zvláštní podmínky
- Hraniční podmínky (boundary values analysis)
  - jakékoliv situace na/za hranou plánovalých limitů
  - často jinde, než by si neprogramátor myslel (subhraniční)

- Neplatné, chybné, nesprávné a nesmyslné údaje
  - „Meze všech polí i řetězců budeš kontrolovati, neboť tam, kde ty pouziješ slovo ``test'', jiný zajisté napíše ``naprostoneuveritelnedlouhoublbost''.“
  - monkey testing

Jak vybrat testovací data (2)

- Rozhodovací tabulky (decision tables)
  - obsahují všechny kombinace podmínek, proměnných (hodnot vstupů) a výsledných výstupů
- Testování stavů
  - není možné vyzkoušet všechny cesty mezi stavy => třídy ekvivalence
  - každý stav musíme navštívit alespoň jednou
  - testujeme ty přechody a stavy, které jsou na první pohled nejpravděpodobnější; dále nejméně obvyklou cestu mezi stavy a chybové stavy a návraty z nich

- Výkonově orientované testy
  - stálé opakování téže operace -> hledání memory leaks
  - stres testy = suboptimální podmínky (dostupná RAM apod)
  - zátěžové testy = maximální zátěž za dobrých podmínek

Na co se při testování zaměřit

- Testování nemůže (z principu) dokázat správnost programu

- Kritické / nejzávažnější části programu
  - dle specifikace
- Části, u kterých existuje nejistota nebo podezření, že nejsou v pořádku
  - kód napsaný ve spěchu (často jako záplata)
  - kód napsaný nezkušeným programátorem
  - kód třetí strany neznámé kvality

„Hodina pátečního přesčasu programátora stojí tým celé pondělí strávené nad opravami.“

Vlastnosti dobrého testera

- má kritické myšlení – není pro něj nic obecně platné
- je zvídavý
- rád se snaží přijít věcem na kloub
- je neúnavný
- je tvořivý
- je perfekcionista
- má dobrý úsudek
- je taktní – není posměváček
- je přesvědčivý a nedá se snadno odbýt
- nestydí se klást otázky a skrze ně poznává produkt, který testuje

- Kladení otázek je jeden z nejlepších způsobů, jak testeři dělají svoji práci

NAMÍSTO ZÁVĚRU

Pamatovat

- Test musí být opakovatelný
- Testování musí být efektivní
- Samotné testování nestačí