Verifikace a validace

- verifikace = ověření, zda produkt dané fáze vývoje SW odpovídá konceptuálnímu modelu (např. zda kód odpovídá návrhu apod.) - tj. odpověď na otázku: vytvářím produkt správně?
- validace = vyhodnocení SW na konci procesu vývoje SW, abychom zajistili splnění požadavků na SW - tj. odpověď na otázku: vytvářím správný produkt?

- verifikace a validace - široké téma, dále pouze následující tři oblasti:

- automatická statická analýza - používá se nejčastěji pro kontrolu zdrojových textů SW systému, případně kontrola modelů apod.
- inspekce - ruční kontrola artefaktů SW procesu, typicky prováděná skupinou 3 až 5 lidí
- testování - spouštění programu s takovými daty, abychom v programu odhalili defekty

Základní termíny:

- omyl (error) - chybná úvaha nebo překlep vývojáře, vede k jednomu nebo více defektům
- defekt (fault, bug, defect) - rozdíl mezi chybným programem a jeho správnou verzí
- symptom (symptom, failure, run-time fault) - pozorovatelné chybné chování programu; defekt se při konkrétním běhu může projevit žádným, jedním nebo více symptomy

Automatická statická analýza

- používají se programy pro automatickou kontrolu modelů nebo zdrojových textů
  - překladač jazyka Java – silná typová kontrola,
  - slabě typované jazyky (např. C) - statické analyzátory (code checkers), např. lclint(1) - detekuje neinicializované proměnné, odchylky od standardů apod.

- další kontroly - mnoho programů používá pro detekci podezřelých míst heuristiky
  - nevýhoda: pokud zdrojový text neodpovídá heuristikám zabudovaným v programu, mohou tyto programy produkovat falešná chybová hlášení
  - příklad program: Jlint pro jazyk Java

Inspekce a procházení programů

- používají se při přezkoumávání (review) DSP, detailního návrhu, kódu (provádějí se před testováním programu, může jim předcházet statická analýza)
- zahrnují čtení dokumentu nebo programu týmem např. 3 nebo 4 lidí (jeden z nich je autor) s cílem nalézt defekty (nikoli jejich řešení)

- pro inspekce kódu:
- výhody:
  - bývají poměrně efektivní (najdou 30% až 70% defektů detailního návrhu a kódování)
  - úsilí bývá přibližně poloviční oproti ekvivalentnímu otestování na počítači (za předpokladu , že nemáme testy již připravené, jinak mohou testy běžet automaticky)

- cena opravy defektu bývá nižší než při testování na počítači (je známá přesná příčina defektu, zatímco testování na počítači najde pouze symptomy)
- nalézá jiné typy defektů než klasické testování, tj. je s ním komplementární (je vhodné provádět obojí)

- nevýhody:
  - je třeba získat s inspekcemi zkušenost
  - experimenty ukazují, že individuální čtení kódu je naopak méně efektivní než individuální testování

Faganovské inspekce kódu

- zahrnují procedury a techniky pro skupinové čtení kódu, poprvé využity ve firmě IBM

- tým provádějící inspekci - obvykle 4 osoby
  - moderátor
    - distribuuje materiály pro schůzku, plánuje a vede schůzku, , zaznamenává defekty, zajišťuje jejich opravy
    - schopný programátor, nikoli však autor programu;
    - nemusí mít ani detailní znalosti programu, jehož inspekce se provádí
  - programátor - autor programu
  - návrhář (pokud je odlišný od programátora)
  - specialista na testování

Před schůzkou:

- moderátor (v dostatečném čase předem) rozdistribuuje program a specifikaci návrhu programu, účastníci se mají s materiálem seznámit

Na schůzce:

- optimální doba inspekce cca 90 -120 min, bez přerušení

1. programátor požádán o vysvětlení logiky programu příkaz po příkazu
   - během programátorova proslovu mají ostatní účastníci otázky, jejichž cílem je zjistit, zda se v kódu nacházejí defekty
   - velkou část defektů najde programátor během výkladu (samotné čtení kódu před posluchači je efektivní - technika pro hledání defektů)
   - moderátor zodpovědný za to, že se účastníci zaměří na vyhledávání defektů, nikoli na jejich řešení

2. program analyzován vzhledem k seznamu obvyklých programátorských chyb (seznam byl vytvořen např. v průběhu předchozích inspekcí)

Po skončení schůzky:

- programátor dostane seznam defektů
    - pokud je nalezeno více defektů nebo pokud některý defekt vyžaduje podstatný zásah do programu, může se domluvit nová inspekce po opravě programu
    - defekty jsou analyzovány a kategorizovány, použijeme je pro zpřesnění seznamu obvyklých programátorských chyb použitých v bodě (2)

Další poznámky:

- při většině inspekcí se projde cca 150 příkazů za hodinu
- vedlejším efektem zpětná vazba týkající se programátorského stylu, výběru algoritmů a programovacích technik

- identifikace částí, které obsahují více defektů
  - defekty se vyskytují ve shlucích, pravděpodobnost existence dalších defektů v dané sekci programu (např. podprogramu) je přímo úměrná počtu defektů v příslušné sekci již nalezených
  - pokud jsou v některé sekci nalezeny defekty, měli bychom se na ní více zaměřit, např. při testování

- inspekce fungují jen v případě, že k nim všichni účastníci mají patřičný přístup (programátor může chápat inspekci jako útok na svou osobu nebo jako pomoc ke zlepšení kvality svého kódu)

Příklad obvyklých programátorských chyb (pro jazyk C):

- data
  - je proměnná inicializována?
  - jsou odkazy do pole v rámci definovaných mezí pole?
  - nenastává při indexování pole chyba off-by-one?
  - ukazuje ukazatel na alokovanou paměť?
  - pokud čteme záznam ze souboru, má proměnná správný typ?

- chyby výpočtu
  - jsou v kódu výpočty se smíšenými typy (např. sčítání float a int)?
  - je do kratší hodnoty (např. int) přiřazována hodnota s delší reprezentací?
  - je možné přetečení nebo podtečení během výpočtu?
  - může nastat případ, že dělitel je 0?
  - jaké jsou důsledky nepřesností reálné aritmetiky?
  - atd.

- řízení toku
- rozhraní
- vstup a výstup
- ostatní

Procházení kódu (walkthroughs)

- technika detekce defektů pomocí skupinového čtení kódu (podobně jako inspekce)- v podrobnostech se liší

- schůzka 3 až 5 lidí, 1 až 2 hodiny, nemá být přerušena
  - moderátor - podobně jako v případě inspekcí
  - sekretář - zaznamenává všechny nalezené defekty
  - tester
  - programátor - autor kódu
  - role ostatních členů týmu není ustálená, doporučuje se např. zkušený programátor, začínající programátor (má zatím nezkalený pohled), osoba, která bude provádět údržbu apod.

Před schůzkou: materiál předem

Schůzka: - hra na počítač:

- tester přijde na schůzku s malým počtem papírových testovacích případů - vstupy a očekávané výstupy programu (podprogramu)
- tým provádí testovací případ, stav programu (hodnota proměnných) zaznamenává na tabuli nebo na papír - v případě nejasností se ptá programátora na logiku programu a na předpoklady (většina defektů je nalezena otázkami, nikoli testovacími případy)

Po schůzce: obdobně jako u inspekcí - tj. programátor dostane seznam defektů, opraví

Další poznámky:

- opět podstatný přístup - tým by měl hodnotit program, nikoli toho, kdo program napsal (defekty nejsou způsobeny neschopností programátora, ale nutný důsledek nedokonalých metod programování a složitosti problému)

Testování

- testování = spouštění programu se záměrem najít v něm defekty (tj. snažíme se, aby se projevily symptomy případných defektů)

Black box a white box testování

- dva základní přístupy k testování - "black box" a "white box" testování

Black box testování (také názvy: functional, data-driven, input/output driven testing)

- tester na program pohlíží jako na černou skříňku s danou specifikací, vnitřní struktura a vnitřní funkce programu ho nezajímají
- hledá případy, ve kterých se program nechová podle specifikace

- pro nalezení všech defektů by bylo nutné otestovat program se všemi možnými vstupy (platnými i neplatnými), což je prakticky nemožné (např. překladač jazyka C bychom museli otestovat se všemi platnými i neplatnými programy)

- Otázka: jak maximalizovat počet defektů nalezený konečným počtem testovacích případů? -> k programu nemůžeme přistupovat čistě jako k černé skříňce, ale musíme učinit rozumné předpoklady o jeho vnitřním chování

White box testování (také: glass-box, clear-box, logic-driven testing)

- testovací data se odvozují z programové logiky
- pro úplné otestování programu bychom potřebovali pomocí testovacích případů otestovat všechny možné logické cesty v programu (analogie otestování programu se všemi možnými vstupy, viz výše)

- ale - dva zásadní problémy:
  1. počet logických cest je i v malých programech příliš velký – například

```
for (i=0; i<100; i++) {
    if (podmínka)
        příkaz1;
    else
        příkaz2;
}
```

- za předpokladu nezávislosti podmínek 2^100 logických cest, kterými může být vykonán (ve skutečnosti podmínky nebudou nezávislé, takže cest bude méně)

2. i po otestování všech logických cest mohou v programu zůstat nenalezené defekty - některé logické cesty mohou chybět a nemusejí být nalezeny defekty citlivé na data, např.

```
if ((a - b) < epsilon) ...  // místo: if (abs(a - b) < epsilon)
```

- při black-box i white-box testování se budou testovací případy skládat z popisu vstupních dat a z popisu správného výstupu pro daná vstupní data - program nebo jeho část spustíme se vstupními daty, porovnáme předpoklad se skutečným výstupem (nejlépe automaticky)

- testovací případy mají obsahovat platné i neplatné vstupy

- testovací případy je třeba uchovávat, protože je můžete znovu potřebovat (např. pro otestování programu po změně)

- je nutné také zkontrolovat, zda program neprovádí nechtěné vedlejší efekty (zápisy do databáze apod.)

Návrh testovacích případů

- úplné otestování programu není možné ->pro testování je velmi podstatný návrh efektivních testovacích případů, tj.hledáme podmnožinu všech testovacích případů, která má největší pravděpodobnost nalézt většinu defektů

- první nápad - náhodně vybraná podmnožina všech možných vstupů - pravděpodobně jedna z nejhorších možností - malá pravděpodobnost být optimální podmnožinou nebo alespoň se jí blížit

- použitelné metody - kombinace myšlenek black box a white box testování

- existuje několik metodik, každá má své silné a slabé stránky - tj. každá detekuje/přehlédne jiné typy defektů -> vhodné připravovat testovací případy pomocí více metod

Rozdělení vstupů do ekvivalentních tříd

- dobrý testovací případ bude mít dvě vlastnosti:
  - bude vyvolávat co nejvíc vstupních podmínek, tím omezí celkový počet potřebných testovacích případů
  - bude pokrývat určitou množinu vstupních hodnot

Množinu vstupů pak rozdělíme do tříd ekvivalence tak, abychom mohli rozumně předpokládat, že test nějaké reprezentativní hodnoty v dané třídě je ekvivalentní testu kterékoli další hodnoty

- z těchto úvah je odvozena metodologie pro black-box testování známá jako equivalence partitioning = rozdělení do tříd ekvivalence - nejprve identifikujeme třídy ekvivalence a pak definujeme testovací případy

identifikace tříd ekvivalence

- vezmeme každou vstupní podmínku a podle ní rozdělíme množinu všech vstupních hodnot do dvou nebo více podmnožin

- existují dva typy tříd ekvivalence - platné (reprezentující platné vstupy) a neplatné (reprezentující chybné vstupní hodnoty)

- rozdělení do tříd ekvivalence je heuristický proces, můžeme využít následujících doporučení:
    - pokud vstupní podmínka specifikuje interval hodnot (např. rok může být mezi 1900 a 2050), pak máme jednu platnou třídu ekvivalence

(hodnoty 1900 až 2050) a dvě neplatné třídy ekvivalence (hodnoty < 1900, hodnoty > 2050)

- pokud vstupní podmínka specifikuje množinu vstupních hodnot a pokud lze předpokládat, že každá z nich bude obsluhována jinak (např. "vlak", "autobus"), bude jedna platná třída ekvivalence pro každý prvek množiny; přidáme jednu neplatnou třídu ekvivalence pro další prvek množiny (např."letadlo")

- pokud vstupní podmínka specifikuje situaci, která "musí nastat", např. první znak identifikátoru musí být písmeno, bude jedna platná třída ekvivalence (první znak je písmeno) a jedna neplatná třída ekvivalence (není písmeno)

- pokud je důvod předpokládat, že prvky nějaké třídy ekvivalence nejsou obsluhovány stejně, rozdělte třídu do menších tříd ekvivalence

definice testovacích případů

- pro každou neplatnou třídu ekvivalence vytvoříme samostatný testovací případ (to je nutné, abychom otestovali každou podmínku kontrolující neplatný vstup); testovací případy budou typicky obsahovat:
  - příliš málo dat nebo žádná data
  - příliš mnoho dat
  - neplatná data (např. negativní počet zaměstnanců)

- pro platné třídy ekvivalence vytváříme testovací případy, pokrývající co nejvíce platných tříd ekvivalence; testovací případy budou typicky obsahovat:
  - nominální případy = běžné nebo očekávané hodnoty
  - minimální normální konfiguraci (např. jediný zaměstnanec)
  - maximální normální konfiguraci (pokud ji umíme určit)

- je velmi vhodné testovat hraniční hodnoty tříd ekvivalence vstupních hodnot – např. pokud je platný vstup -1.0 až +1.0, pak vytvoříme testovací případy pro vstupy -1.0, +1.0, -1.0001, +1.0001 (stejně dobré-nutné otestovat hranice výstupních hodnot)

- analýza hraničních hodnot je jedna z nejužitečnějších technik pro návrh testovacích případů, ve skutečnosti ale vyžaduje více přemýšlení, než to vypadá na první pohled.

Příklad (bankomat)

Např. SW bankomatu bychom mohli otestovat následujícími testovacími případy:

1. Vadná karta, konec.
2. Platná karta, chybné PIN, konec.
3. Platná karta, platné PIN, konec.
4. Výběr platné částky, dotaz na zůstatek.
5. Výběr neplatné částky.
6. Neplatný dotaz na zůstatek.

White-box testování

- využíváme znalost implementace
- obvykle pro testování relativně malých částí programu (podprogramy v modulu, metody třídy) - tj. testování jednotek
- pokud jsou moduly integrovány do systému, složitost narůstá tak, že jsou strukturální techniky prakticky neproveditelné
- se zvyšováním rozsahu projektu testy jednotek zabírají menší podíl na celkovém času vývoje (cca od 35% pro malé systémy až po cca 8% pro velké systémy)

- pokud známe strukturu implementace, můžeme pro testování použít třídy ekvivalence a testovat běžné a hraniční podmínky se znalostí kódu

- vytváření testů si vynucuje dobrý návrh - má-li být kód testovatelný, je nutné, aby moduly/třídy byly volně vázané, nevyžadovaly složitou inicializaci apod.

- z kódu můžeme určit také další typ hraniční podmínky, který nastává, pokud dochází ke kombinaci vstupních hodnot - například pokud podprogram hodnoty násobí, testovací případy mohou zahrnovat dvě velká kladná čísla, dvě velká záporná čísla apod.

Pokrytí kódu testovacími případy

- při white-box testování nás zajímá, do jaké míry testovací případy pokrývají zdrojový text programu - praktické metody testují pouze rozumnou podmnožinu cest v programu

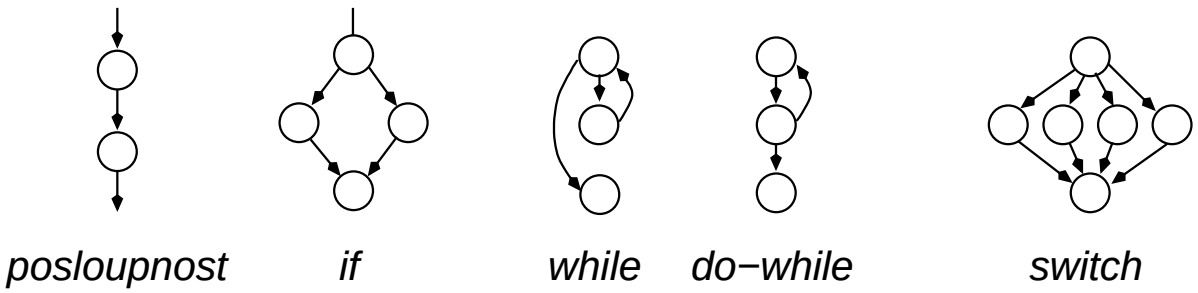
- je dobrým kritériem alespoň jedno vykonání každého příkazu? (statement coverage, pokrytí všech příkazů)

např.if (a>1) and (b=0) then x = x/a;

- oba příkazy (if a přiřazovací příkaz) by bylo možné pokrýt jediným testovacím případem (a=2, b=0) -> pak neodhalíme defekty
  - místo "and" má být "or",
  - místo "a>1" má být "a>=1" nebo "a>0"

- závěr: podmínka pokrytí všech příkazů je nutná, ale nikoli postačující

- kritérium pokrytí zesílíme - je dobrým kritériem pokrytí všech rozhodovacích příkazů tak, aby byly vykonány všechny jejich větve?
  - musíme vytvořit tolik testovacích případů, aby se v každém příkazu "if" vykonala alespoň jednou větev při podmínce "false" a alespoň jednou větev při podmínce "true" atd.
  - pro složitější podprogramy se vyplatí modelovat cestu podprogramem pomocí orientovaného grafu popisujícího možný tok řízení v podprogramu



- uzly reprezentují příkaz nebo část příkazu (přiřazovací příkazy, volání podprogramů, případně podmínky rozhodovacích příkazů)
- každý pár uzlů, pro který je možný přenos řízení, je spojen hranou

- nezávislá cesta = musí procházet alespoň jednu novou hranu v grafu, tj. provést jednu nebo více nových podmínek

- existují extra případy (podprogram neobsahuje rozhodování, má více vstupních bodů apod.) - připojujeme ještě podmínku pokrytí všech příkazů
- kritérium "pokrytí všech rozhodovacích příkazů" by postačovalo, pokud bychom měli vždy jedinou podmínku v rozhodovacím příkazu, ale např.

pro kód "if (a>1) and (b=0) then x = x/a" je to stále ještě slabé kritérium

- pokud otestujeme pomocí (a=2, b=0) a (a=2, b=1), neodhaleny by zůstaly defekty jako "místo a>1 má být a>=1"

->rozumným kritériem je **pokrytí všech kombinací podmínek v rozhodovacím příkazu** (multiple condition coverage)

- oproti branch coverage navíc vyžaduje, aby byly otestovány všechny kombinace hodnot logických operandů v podmínce

- pak kód "if (a>1) and (b=0) then x = x/a"

  můžeme otestovat čtyřmi testovacími případy:

  - . (a=2, b=0) => obě podmínky jsou true, větev "then" se provede
  - . (a=2, b=1) => true, false, větev "then" se neprovede
  - . (a=1, b=0) => false, true, větev "then" se neprovede
  - . (a=1, b=1) => false, false, větev "then" se neprovede

  - testovací případy není vhodné generovat strojově z kódu, ale můžeme si pomoci nástroji generujícím "nápady na testovací případy" z výrazů (např.pro Javu nástroj "multi")

- další obvyklé kritérium - **pokrytí smyček** – vyžaduje 3 testovací případy:

    - tělo smyčky se nevykoná, tj. při prvním vyhodnocení bude test "false"
    - tělo smyčky se vykoná právě jednou, tj. při prvním vyhodnocení bude test "true", poté "false"
    - tělo smyčky se vykoná více než jednou, tj. test bude "true" nejméně dvakrát

- další možné kritérium - all-du-path
    - jedna cesta pro každou definici-použití: pokud je proměnná definována v jednom příkazu a použita v jiném, cesta by měla procházet oběma příkazy

- pro určení pokrytí velkých sad testů spouštěných na celé systémy jsou užitečné tyto podmínky:

    - pokrytí podprogramů - zda byl podprogram vyvolán alespoň jednou
    - pokrytí volání - zda každý podprogram volal všechny podprogramy, které může volat

- testy prováděné bez měřené pokrytí kódu typicky pokrývají pouze 55% kódu (Grady 1993)
- pro zjištění, které cesty byly vykonány, se používají dynamické analyzátory programu - při překladu je ke každému příkazu připojen kód, který počítá, kolikrát byl daný příkaz vykonán (tzv. instrumentace)
- po běhu můžeme zjistit, které části programu nebyly pokryty příslušným testovacím případem
- např. jcoverage (http://www.jcoverage.com) - volně šířený nástroj pro Javu, GCT (Generic Coverage Tool) - volně šířený nástroj pro jazyk C.

Strategie testování

- testování by mělo být předem naplánováno spolu s celým SW procesem

- pro zvýšení efektivity testování je vhodné provádět také inspekce kódu

- testování by mělo začínat na úrovni jednotek (procedur, tříd - funkce každé jednotky se ověří samostatně) a postupovat směrem k větším celkům (podsystémům, celému systému)

- testování jednotek provádí obvykle ten, kdo danou část napsal; testování větších celků provádí nebo alespoň řídí specialista - tester (rozsáhlejší software testuje nezávislá testovací skupina)

Poznámka (psychologický problém testování)

- pro řadu programátorů je obtížné testovat vlastní programy, protože
  - jejich zájem je spíše ukázat, že jejich program neobsahuje defekty a pracuje podle požadavků zákazníka.
  - bývá obtížné přepnout se z kódování programu (proces konstrukce) na testování (destrukci).
  - program obsahuje chyby způsobené nepochopením specifikace, které programátor nemůže sám odhalit.

- proto je vhodné přenechat testování sestaveného programu někomu jinému, např. nezávislému testovacímu týmu. Programátor pak je zodpovědný za otestování jednotek, které vytvořil (procedur, funkcí, tříd).

Testování by mělo probíhat postupně současně s implementací systému v následujících krocích:

- testování jednotek (unit testing) - testujeme nejmenší jednotky návrhu, např. procedury nebo funkce; pro testování můžeme používat white-box techniky

- integrační testování (integration testing) - sestavujeme software, spolu s tím testujeme defekty týkající se rozhraní mezi jednotlivými částmi

- validační testování (validation testing) - po integraci se zaměřujeme na funkce viditelné uživatelem

- testování systému (system testing) - pokud SW je pouze jednou součástí většího celku, účelem je otestovat celek; např. zátěžové testování, zotavení po závadě apod.

Testování jednotek

- pojmem "jednotka" se v případě konvenčně napsaného SW obvykle myslí procedura, funkce, nebo nejmenší samostatně přeložitelná jednotka zdrojového textu (čili neexistuje všeobecně přijímaná definice)

- jednotka se testuje samostatně, okolní jednotky jsou nahrazeny ovladači testů (řídí testovanou jednotku) nebo testovacími maketami (nahrazují jednotky volané z testované jednotky)

- používají se již probrané white-box techniky

- pro objektově-orientovaný software se za jednotku považuje třída

- při testování třídy bychom měli provést:
  - samostatné otestování každé metody
    - některé metody lze testovat až po předchozím vyvolání jiných metod, např.po inicializaci objektu
    - pokud používáme dědičnost, musíme testovat i všechny zděděné operace (mohou obsahovat předpoklady o dalších operacích a atributech, které ale mohly být potomkem změněny; obdobně musíme znovu otestovat potomky při změně rodiče)
  - testovat průchod všemi stavy objektu, případně simulace všech událostí, které způsobují změnu stavu objektu
    - pokud jsme vytvořili stavový diagram objektu, můžeme z něj určit posloupnost přechodů, které chceme testovat, a najít posloupnost událostí, které ji způsobí
  - testovat nastavení všech atributů objektu

Nástroje JUnit a JUnitDoclet

- pomůcka pro testování jednotek v jazyce Java - knihovna JUnit, která poskytuje nástroje pro spouštění testovacích případů a umí graficky i textově zobrazit výsledky testů. Podobné nástroje existují i pro mnoho dalších programovacích jazyků.

- pro jednotkové testy je důležité mít také zavedené konvence, například:

  - pro každou třídu "Třída" vytvoříme třídu "TřídaTest"

  - pro každou veřejnou metodu vytvoříme testovací metodu "testJménoMetody()"

  - testovací metody můžeme vytvořit i pro ty soukromé metody, které obsahují nějaký složitější algoritmus

- testujeme vždy správné chování i chování při chybách, testovací metody můžeme někdy rozdělit do dvou skupin:
  - testJménoMetody - testuje obvyklé chování
  - testJménoMetodyFailures - testuje chování při chybách

- někdy můžeme navíc vytvořit testovací třídu pro testování spolupráce tříd

- s knihovnou JUnit souvisí pomocný nástroj JUnitDoclet (je implementován jako plug-in do nástroje JavaDoc), umí vygenerovat kostru testovacích případů pro JUnit.

Poznámka (jednotkové testy a makety)

- na to, aby kód (vytvářené třídy) byl testovatelný, je třeba pamatovat už při návrhu; nejlepší je navrhovat testy současně s kódem (či dokonce nejdříve testy, potom kód – tzv. programování řízené testy)

- pokud metoda "foo()" třídy A používá metodu "bar()" třídy B, máme dvě možnosti:
  - pokud je třída B dostatečně samostatná a dobře testovatelná, využijeme jí přímo - v testu se jí dotážeme na výsledek operace "bar()"
  - pokud je třída B špatně testovatelná (např. ve skutečnosti tiskne na tiskárnu, chová se nedeterministicky apod.), použijeme místo ní v testu jednoduchou maketu, které se můžeme dotázat na výsledek operace, aplikačnímu kódu obvykle předáváme maketu jako argument

- jednotkové testy většinou netestují spolupráci mezi třídami; proto jsou zapotřebí také ostatní typy testů

Integrační testování

- po otestování individuálních komponent musíme komponenty integrovat - sestavit do částečného nebo úplného systému

- výsledek musíme otestovat na problémy, které vznikají interakcí komponent

- (velký třesk) - po otestování jednotlivých modulů je z nich v jediném kroku sestavena aplikace

- použitelné pouze pro malé programy, ale i u nich mohou nastat problémy, pro větší systémy nejméně efektivní způsob integrace = vysoká pravděpodobnost neúspěchu

- hlavním problémem je lokalizace defektů, protože vztahy mezi komponentami mohou být značně složité

    - proto často pro integraci a testování doporučuje inkrementální přístup
    - nejprve integrujeme minimální konfiguraci systému, otestujeme
    - k systému přidáváme inkrementy, po každém přidání systém otestujeme
    - pokud nastaly problémy, budou pravděpodobně (ale ne nutně) způsobeny přidáním posledního inkrementu

- ve skutečnosti to není tak jednoduché, protože některé vlastnosti budou rozptýleny do několika komponent, vlastnost můžeme otestovat až po integraci těchto komponent => při plánování testů je třeba počítat s časovým plánem na dokončení modulů

- pokud má důležitý modul neočekávané problémy, může se tím zdržet celá integrace (programátor řeší problém, zatímco všichni ostatní na něj čekají)

Testování rozhraní

- cílem testování rozhraní je detekovat defekty, které mohou vzniknout chybnou interakcí mezi moduly nebo podsystému nebo chybným předpokladem o rozhraní

- testování rozhraní je obtížné, protože některé typy defektů se projeví pouze za neobvyklých podmínek

- obecná pravidla (volně podle Sommerville 2001):
  - v testovaném kódu najděte všechna volání externích komponent
  - navrhněte množinu testů tak, aby externí komponenty byly volány s parametry, které jsou extrémy jejich rozsahu (např. prázdný řetězec, dlouhý řetězec, který by mohl způsobit přetečení apod.)
  - navrhněte testy, které by měly způsobit neúspěch externí komponenty (zde často chybné předpoklady)
  - v systémech s předáváním zpráv použijte zátěžové testování (viz dále)

- pokud spolu komponenty komunikují prostřednictvím databáze, sdílené paměti apod., navrhněte testy, ve kterých se bude lišit pořadí aktivace komponent; testy mohou odhalit implicitní předpoklady programátora o pořadí, v jakém pořadí budou sdílená data produkována a konzumována

- mnoho defektů rozhraní odhalí statické testy
  - např. silná typová kontrola v překladačích jazyka Java
  - pro slabě typované jazyky (např. C) by se před integračním testováním měly použít statické analyzátory (code checkers)

- některé inspekce se mohou zaměřit také na rozhraní komponent a jejich předpokládané chování

Validační testování

- začíná tam, kde končí integrační testování
- testujeme, zda SW splňuje požadavky uživatele

- akceptační testování - test zda produkt splňuje požadavky zadavatele, testuje zadání na reálných datech, testuje pouze externí chování systému, vnitřní strukturu přitom ignorujeme

  - obsah testu má specifikovat zadavatel
  - obsahem by měly být instance případů použití (tj. obsah funkčního testu je výhodné specifikovat v souvislosti se sběrem požadavků)
  - snaha o zautomatizování, abychom mohli spouštět po změnách aplikace

Alfa a beta testování

- pro generické produkty není většinou možné vykonat přejímací testování u každého zákazníka, proto alfa a beta testování

- alfa testy: na pracovišti, kde se SW vyvíjí (známé prostředí) - testuje uživatel, vývojoví pracovníci ho sledují a zaznamenávají problémy

- beta testy: testují vybraní uživatelé ve svém prostředí (vývojářům neznámém) - defekty ohlášené uživateli jsou opraveny => finální produkt

Testování systému

- účelem otestovat celý systém, jehož je SW součástí
- mívají různý účel, např. otestovat vlastnosti jako je výkonnost, kompatibilita, bezpečnost, instalovatelnost, spolehlivost apod.

Testování výkonnosti

- v mnoha typech systémů je nepřípustné, aby SW nesplňoval požadavky na výkonnost (zejména v řídících systémech)
- v takovém případě by se výkonnost měla testovat ve všech krocích včetně jednotkových testů
- pomocné procedury monitorují dobu vykonání apod.

Zátěžové testování

- obvykle se používají testy, kde se zátěž postupně zvyšuje, dokud není výkonnost systému neakceptovatelná nebo dokud systém nehavaruje

- zátěž = množství dat, frekvence požadavků, data, která jsou extrémně náročná na zpracování

- je vhodné určit části kódu, které mohou být problematické při velké zátěži, zátěžové testy navrhnout tak, aby pokrývaly především tyto části kódu

- ověření, zda havárie systému nepoškodí data apod.
- může odhalit některé defekty, které se normálně neprojeví

- důležité zejména u internetových aplikací, v distribuovaných systémech apod., kde se vysoce zatížené systémy mohou zahltit, protože si vyměňují mnoho koordinačních dat, čímž se opět zvyšuje zátěž systému atd.

Testování zotavení systému po havárii

- mnoho systémů se musí být schopno zotavit po havárii v předepsaném čase, jinak hrozí značné finanční ztráty

- při testování zotavení systému způsobujeme různé havárie systému a ověřujeme, zda se systém zotavil správně a v časovém limitu