KIV/ZSWI 2003/2004
Přednáška 12

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 je široké téma, my se budeme zabývat pouze následujícími třemi oblastmi:

* 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, které budu dále používat:

* 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
  - například překladač jazyka Java obsahuje silnou typovou kontrolu
  - pro slabě typované jazyky (např. C) se používají statické analyzátory (code checkers)
    . nejstarší ze známých statických analyzátorů je program lint(1), který byl součástí historických systémů UNIX
    . v současnosti se často používá volně šířený lclint(1)
    . detekuje neinicializované proměnné, odchylky od standardů apod.
    . měl by se použít vždy před inspekcemi

* 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 (inspekce kódu se provádějí před testováním programu, inspekcím by měla předcházet statická analýza)
* zahrnují čtení dokumentu nebo programu týmem např. 3 nebo 4 lidí (jeden z nich autor) s cílem nalézt defekty (nikoli jejich řešení)
* budu popisovat pro testování kódu, testování ostatních artefaktů SW procesu je analogické

* výhody:
  - bývají poměrně efektivní (typicky najdou 30% až 70% defektů detailního návrhu a kódování)

- úsilí bývá přibližně poloviční oproti ekvivalentnímu otestování na počítači (na druhou stranu - pokud máme testy již připravené, mohou běžet automaticky)
- cena opravy defektu bývá nižší než při testování na počítači (protože 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:
- pro maximální efektivitu je třeba, aby s nimi tým získal zkušenost

Faganovské inspekce kódu
.....

* zahrnují procedury a techniky pro skupinové čtení kódu (Fagan 1976)
* poprvé využity ve firmě IBM

* tým provádějící inspekci se obvykle skládá ze 4 osob
- moderátor - jeho úkolem je rozdistribuovat materiály pro schůzku, naplánovat schůzku, vést jí, zaznamenávat defekty, zajistit aby byly opraveny
  . moderátor by měl být schopný programátor, ale ne autor programu; nemusí mít detailní znalosti programu jehož inspekce se provádí
- programátor - autor programu
- dalšími členy obvykle návrhář (pokud je odlišný od programátora) a specialista na testování

* před schůzkou moderátor (např. několik dní předem) rozdistribuuje program a specifikaci návrhu programu, účastníci se mají s materiálem seznámit

* inspekce probíhají podle následujícího scénáře:
- optimální doba inspekce je asi 90 až 120 min, měla by probíhat bez přerušení
1. na schůzce je programátor požádán o vysvětlení logiky programu příkaz po příkazu
   - moderátor je zodpovědný za to, že se účastníci zaměří na vyhledávání defektů, nikoli na jejich řešení
   - během proslovu vznikají otázky, usilující o určení, zda se v kódu nacházejí defekty
   - zkušenost ukazuje, že velkou část defektů najde programátor během výkladu (jinými slovy: samotné čtení kódu před posluchači je efektivní technika pro hledání defektů)
2. program je analyzován vzhledem k seznamu obvyklých programátorských chyb (seznam se vytváří v průběhu předchozích inspekcí)
3. po skončení schůzky dostane programátor 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)

* při většině inspekcí se projde cca 150 příkazů za hodinu
* kromě nalezení defektů je 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 na počítači

Poznámka pro zajímavost (Internet Explorer)

V souvislosti v výše uvedeným je zajímavé si přečíst následující (cit. z http://www.zive.cz, článek z 11.2.2004):

Podle britského šéfa bezpečnosti společnosti Microsoft je Internet Explorer nejbezpečnější dostupný internetový prohlížeč. Toto tvrzení je založeno na počtu chyb, které již byly odstraněny. K prohlášení došlo po vydání

```
 bezpečnostní záplaty z minulého pondělí, která však přinesla problémy. ...

[]

* aby inspekce fungovaly, musí k nim mít všichni účastníci správný přístup
 - pokud programátor chápe inspekci jako útok na svou osobu a brání se, bude
   inspekce nutně neefektivní
 - naopak funguje pokud bude chápat 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)?
  . často zdrojem chyb
 - 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)
..............................

* podobně jako inspekce je technika detekce defektů pomocí skupinového čtení,
  v podrobnostech se liší

* schůzka 3 až 5 lidí, trvá 1 až 2 hodiny, také nemá být přerušena
 - role moderátora - podobná 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.

* stejně jako v případě inspekcí by měli dostat materiál několik dní předem
 - členové týmu si hrají na počítač: tester přijde na schůzku s malým počtem
   papírových testovacích případů - vstupy a očekávané výstupy programu nebo
   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
 - následuje obdoba bodu 3. z popisu inspekcí - tj. programátor dostane
   seznam defektů, opraví...

* podobně jako v případě inspekcí je podstatný přístup
 - tým by měl hodnotit program, nikoli toho, kdo program napsal
 - na defekty nehledět jako na neschopnost programátora, ale chápat je jako
   nutný důsledek doposud nedokonalých metod programování

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

* existují dva základní způsoby testování - ”black box” a ”white box” testování

* black box testování (používají se 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
 - víme, že úplné otestování programu je nemožné - jak ale maximalizovat počet defektů nalezený konečným počtem testovacích případů?
  . k programu už nemůžeme přistupovat čistě jako k černé skříňce, ale musíme učinit nějaké 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)

 - má dva zásadní problémy:
  . počet logických cest je i v malých programech příliš velký - například fragment programu

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

    má za předpokladu nezávislosti podmínek 2^100 logických cest, kterými může být vykonán (ve skutečnosti podmínky nebudou nezávislé, takže cest bude méně)
  . i po otestování všech logických cest mohou v programu zůstat nenalezené defekty, protože některé logické cesty mohou chybět a protože 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

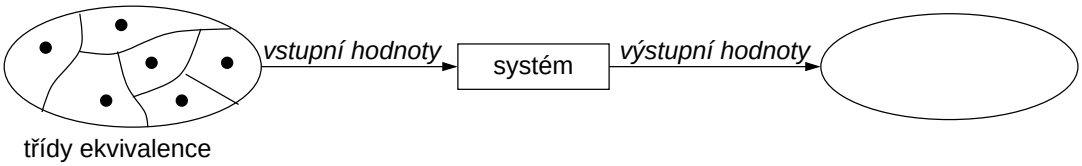
* už jsme viděli, že úplné otestování programu není možné
 - proto je pro testování velmi podstatný návrh efektivních testovacích případů
 - tj. klademe si otázku - jaká podmnožina všech testovacích případů má největší pravděpodobnost nalézt většinu defektů?

* první nápad - náhodně vybraná podmnožina všech možných vstupů
 - pravděpodobně jedna z nejhorších možností, protože má malou pravděpodobnost být optimální podmnožinou nebo být alespoň blízká optimální podmnožině

* použitelné metody budou kombinací myšlenek black box a white box testování
 - existuje několik metodik, každá má své silné a slabé stránky - tj. každá detekuje/přehlédne jiné typy defektů
 - proto je dobré připravovat testovací případy pomocí více metod

Rozdělení vstupů do ekvivalentních tříd
.....

* dobrý testovací případ bude mít dvě vlastnosti:
 - bude vyvolávat co nejvíc vstupních podmínek a tím omezí celkový počet potřebných testovacích případů
 - testovací případ by měl pokrývat určitou množinu vstupních hodnot
  . množinu vstupů bychom měli rozdělit 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
 - pak definujeme testovací případy

* identifikace tříd ekvivalence
 - vezmeme každou vstupní podmínku (často větu nebo frázi z DSP) 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 2000 a 2100), pak máme jednu platnou třídu ekvivalence (hodnoty 2000 až 2100) a dvě neplatné třídy ekvivalence (hodnoty < 2000, hodnoty > 2100)
  . 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 (je písmeno) a jedna neplatná třída ekvivalence (není písmeno)
  . pokud je důvod předpokládat, že prvky nějaké třídy 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)
 - dokud jsou platné třídy ekvivalence nepokryté testovacími případy, vytvoříme testovací případ 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 jí umíme určit)

* je výhodné testovat hraniční hodnoty tříd ekvivalence vstupních hodnot,
 - například pokud je platný vstup -1.0 až +1.0, pak vytvoříme testovací případy pro vstupy -1.0, +1.0, -1.0001, +1.0001
 - stejně otestovat hranice výstupních hodnot

Příklad (bankomat)

Například SW bankomatu bychom mohli otestovat pomocí následujících testovacích případů:

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
---

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

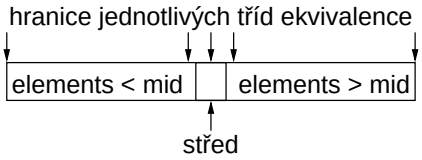
* pokud známe strukturu implementace, můžeme pro testování použít opět třídy ekvivalence a testovat běžné a hraniční podmínky se znalostí kódu
* ilustrovat budu na základě binárního vyhledávání v jazyce Java:
  - pro ty kdo znají pouze Pascal uvádím ekvivalentní konstrukce v Pascalu:

Java	Pascal	Java	Pascal
{	begin	int x;	var x: integer;
}	end	a = b;	a := b;
//	{ komentář }		

```
class BinSearch {

    // binární vyhledávání
    //  key       - klíč
    //  elemArray - seřazené pole prvků, ve kterém se vyhledává
    //  r         - výsledek, obsahuje r.index a boolovskou hodnotu
    //              r.found, která je true pokud byl klíč v poli

    public static void search(int key, int elemArray[], Result r) {
        int bottom = 0;
        int top = elemArray.length - 1;
        int mid;

        r.found = false;                      // n1
        r.index = -1;                         // n2
        while (bottom <= top) {               // n3
            mid = (top + bottom) / 2;         // n4
            if (elemArray[mid] == key) {      // n5
                r.index = mid;                // n6
                r.found = true;               // n7
                return;                       // n8
            } else {                          // n9
                if (elemArray[mid] < key)     // n10
                    bottom = mid + 1;         // n11
                else                          // n12
                    top = mid - 1;            // n13
            } // konec "if"                   // n14
        } // konec "while"                    // n15
    } // konec "search()"
```

```
} // konec "BinSearch"
```

- z kódu můžeme zjistit, že binární vyhledávání rozděluje vyhledávací prostor do třech částí (prostřední prvek pole elemArray[mid], začátek pole, konec pole), každá část tvoří třídu ekvivalence



- pro testování bychom mohli použít například následující testovací případy:

vstupní pole (elemArray) (elemArray)	klíč (key)	výstup (found, index)	třída ekvivalence (pole, prvek)
17	17	true, 0	jedna hodnota, výskyt v poli
17	0	false, -1	jedna hodnota, není v poli
17, 21, 23, 29	17	true, 0	více hodnot, první prvek
9, 16, 18, 30, 31, 41, 45	45	true, 6	více hodnot, poslední prvek
17, 18, 21, 23, 29, 38, 41	23	true, 3	více hodnot, prostřední prvek
17, 18, 21, 23, 29, 33, 38	21	true, 2	více hodnot, soused prostředního
12, 18, 21, 23, 32	23	true, 3	více hodnot, soused prostředního
21, 23, 29, 33, 38	25	false, -1	více hodnot, není v poli

- testování bychom provedli vytvořením ovladače (např. metody s názvem "test", případně "main"), který bude volat testovaný podprogram nebo metodu
- zadané vstupy (případně i výstupy) mohou být zakódovány v ovladači, převzaty z příkazového řádku, zadány uživatelem, přečteny ze souboru apod.
- například:

```
    public static void main(String[] args)
    {
        int[] values = { 21, 23, 29, 33, 38 };
        Result r = new Result();

        search(25, values, r);
        if (r.index != -1 || r.found != false)
            System.out.println("Test 8 FAILED");
    }
```

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

Poznámka (nástroje JUnit a JUnitDoclet)

Jako pomůcka pro testování se v jazyce Java často používá knihovna JUnit, která poskytuje nástroje pro spouštění testovacích případů a umí graficky i textově zobrazit výsledky testů.

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

[]

* 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.

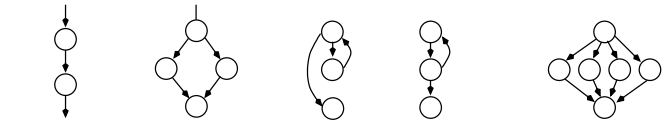
Pokrytí kódu testovacími případy
.....

* při white-box testování nás zajímá, do jaké míry testovací případy pokrývají zdrojový text programu

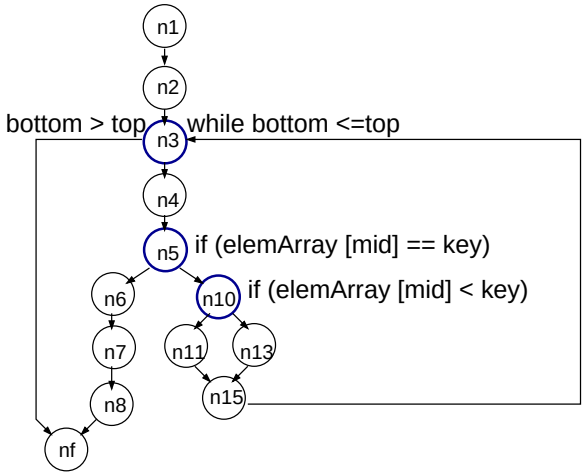
- jak už jsme si uvedli, otestovat všechny cesty v programu je obvykle neproveditelné, proto se o to nebudeme pokoušet
- praktické metody by měly testovat pouze rozumnou podmnožinu cest v programu

* je dobrým kritériem alespoň jedno vykonání každého příkazu? (statement coverage, pokrytí všech příkazů)
 - například mějme příkaz: 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)
 - příklady defektů, které by testem zůstaly neodhaleny: místo ”and” má být ”or”, místo ”a>1” má být ”a>=1” nebo ”a>0” apod.
 - 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? (branch coverage)
 - 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
  . příklad - graf toku pro dříve uvedený příklad na binární vyhledávání v Javě:



  . nezávislá cesta = musí procházet alespoň jednu novou hranu v grafu, tj. provést jednu nebo více nových podmínek
 - kvůli ”patologickým případům” (podprogram neobsahuje rozhodování, má více vstupních bodů apod.) připojujeme ještě podmínku pokrytí všech příkazů
 - jako kritérium by postačovalo, pokud bychom měli vždy jedinou podmínku v rozhodovacím příkazu
 - pro kód ”if (a>1) and (b=0) then x = x/a” je 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”

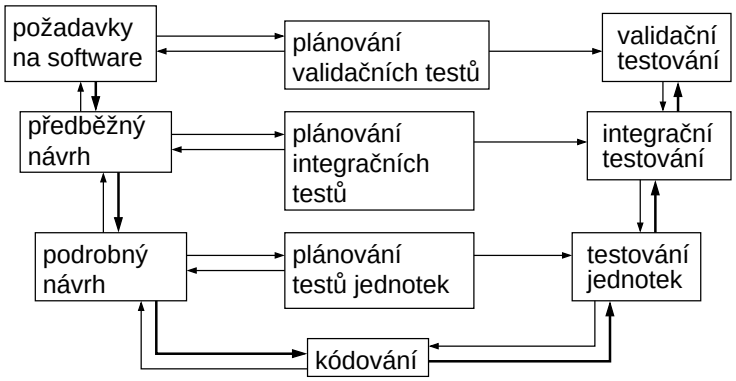
* rozumným kritériem je pokrytí všech kombinací podmínek v rozhodovacím příkazu (multiple condition coverage)
 - oproti branch coverage navíc vyžaduje, aby byly otestovány všechny kombinace hodnot logických operandů v podmínce
 - kód ”if (a>1) and (b=0) then x = x/a” můžeme otestovat čtyřmi testovacími

```
    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 nástrojem generujícím ”nápady na testovací případy” z výrazů (např.
    pro Javu nástroj ”multi”)

* dalším možným kritériem je 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 časté 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
    . příklad nástroje: GCT (Generic Coverage Tool) volně šířený nástroj pro
      jazyk C, viz ftp://ftp.cs.uiuc.edu/pub/testing/gct.files; příklad
      části výstupu: ”program.c”, line 9: loop one time: 10, many times: 2.


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 většinu programátorů je obtížné testovat vlastní programy, protože
 jejich zájem je spíše ukázat, že jejich program neobsahuje defekty a pracuje
 podle požadavků zákazníka.

 Jinými slovy, protože testování je destruktivní proces, pro programátora
 může být velmi obtížné přepnout se z kódování programu (proces konstrukce)
 na testování (destrukci). Program také může obsahovat chyby způsobené
 nepochopením specifikace, které programátor nemůže sám odhalit.

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

[]

Testování 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 by mělo být plánováno v souvislosti s aktivitami konstrukce SW:



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
  - třídy jako samostatné komponenty jsou obvykle rozsáhlejší než samostatné podprogramy
* 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 nastavení všech atributů objektu, dotaz na všechny atributy
  - 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 jí způsobí

Integrační testování

* po otestování individuálních komponent musíme komponenty integrovat = sestavit do částečného nebo úplného systému
* výsledek musíme otestovat na problémy, které vznikají interakcí komponent

* ”big bang” (velký třesk) - po otestování jednotlivých modulů je z nich v jediném kroku sestavena aplikace
  - použitelné pouze pro malé programy

- pro větší systémy nejméně efektivní způsob integrace = vysoká pravděpodobnost neúspěchu

* hlavním problémem je lokalizace defektů, protože vztahy mezi komponentami mohou být značně složité
 - proto se často pro integraci a testování používá 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 nebude 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

* uvedu pouze 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 jsou 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 = zadavatel určí, zda produkt splňuje zadání
 - testuje se na reálných datech

* pro generické produkty není většinou možné vykonat akceptační 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

❇