***Softvérové inžinierstvo**, zimný semester 1998/99*

Časť 4: Testovanie, integrácia, údržba

Verzia: 1.0, 16.10.1998

Pavol Mederly, pavol.mederly@fmph.uniba.sk

Časť 4: Testovanie, integrácia, údržba

4.1. Testovanie

Testovanie v užšom zmysle slova označuje *testovanie vytváraného programu*, resp. programov. V širšom zmysle tento pojem označuje *kontrolu vytváraných (medzi)produktov*, od plánu projektu, cez špecifikáciu požiadaviek, návrh, až po používateľskú dokumentáciu. V tejto časti sa budeme zaoberať testovaním vytváraného programu. Kontrole kvality vo všeobecnosti bude venovaná časť o riadení projektov.

Pod testovaním rozumieme jednak neformálne testovanie kódu programátorom a tiež systematické testovanie. Systematické testovanie (ktorým sa budeme v ďalšom zaoberať) je obvykle vykonávané skupinou pre zabezpečovanie kvality softvéru.

Testuje sa nielen správnosť produktu, ale aj jeho použiteľnosť, spoľahlivosť, robustnosť a výkonnosť.

Poznamenajme, že správnosť vzhľadom na špecifikáciu nie je postačujúcou ani nutnou podmienkou pre to, aby mohol byť produkt použitý. Postačujúcou podmienkou nie je preto, lebo aj samotná špecifikácia môže byť chybná. Nutnou podmienkou nie je preto, lebo v istých prípadoch môže byť produkt v praxi použiteľný aj za predpokladu, že v detailoch nezodpovedá špecifikácii (napríklad kompilátor, ktorý pri výskyte istej programátorskej konštrukcie vypíše nadbytočnú varovnú správu).

Existuje testovanie bez spustenia (nonexecution-based) a testovanie so spustením (execution-based).

Techniky pre testovanie bez spustenia:

- inšpekcia kódu,
- matematické metódy overovania správnosti, resp. iných vlastností programu,
- strojová analýza kódu.

Testovaniu so spustením sa venujeme v ďalšom texte.

4.1.1. Testovanie podľa špecifikácie vs. testovanie podľa kódu

Testovanie so spustením pozostáva z výberu testovacích vstupných údajov, určenia im zodpovedajúcich očakávaných výstupov, spustenia testovaného programu na vstupných údajoch a z porovnania skutočných výstupov s očakávanými výstupmi.

Ako vyberať vstupné údaje ?

Najhoršou možnou metódou pre výber testovacích údajov je nesystematické vkladanie ľubovoľných údajov.

Pri systematickom prístupe k výberu vstupných údajov sú tieto dve možnosti (možné sú aj ich kombinácie):

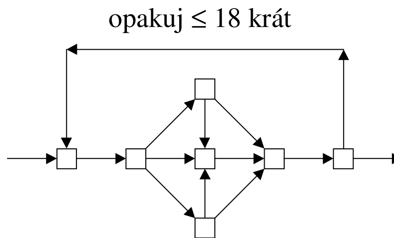
- testovanie podľa špecifikácie (alebo tiež black-box, data-driven, functional, input/output-driven testing), kde sa kód programu neberie do úvahy a testovacie dáta sa vyberajú len na základe zadania,

- testovanie podľa kódu (tiež glass-box, white-box, logic-driven, path-oriented), kde sa ignoruje zadanie a testovacie dáta sa vyberajú len na základe kódu – tak, aby bolo pri vykonávaní testov prejdených čo najviac rôznych ciest v programe.

Bohužiaľ v oboch prípadoch je na úplné otestovanie programu spravidla potrebných toľko testovacích vstupov, že testovanie vyšetrením všetkých možností je nerealizovateľné.

Príklad 1: Majme 20 vstupných parametrov, každý so 4 možnými hodnotami. Pri testovaní podľa špecifikácie toto predstavuje približne 1.1 bilióna možných vstupov.

Príklad 2: Pri testovaní podľa kódu v prípade nasledujúceho fragmentu programu



máme v ňom viac než 4 bilióny možných ciest.

Naviac ani pri prejdení všetkých ciest nemáme záruku, že program je korektný! Ako príklad si vezmime funkciu porovnávajúcu 3 čísla založenú na nesprávnom predpoklade, že tri čísla sú rovnaké, ak sa ich priemer rovná prvému z nich:

```
if ((x+y+z)/3 == x)
  printf ("sú rovnaké");
else
  printf ("nie sú rovnaké");
```

Chceme túto funkciu otestovať tak, že budeme testovacie údaje vyberať prístupom „podľa kódu“. Pre každú cestu (sú len dve) vyberieme vstupy tak, aby program na základe nich danou cestou prešiel. Pre vetvu „sú rovnaké“ vyberieme trojicu x = y = z = 2, pre vetvu „nie sú rovnaké“ vyberieme trojicu x = 1, y = 2, z = 3. Po vložení týchto údajov dostaneme v oboch prípadoch výsledok zhodný s očakávaným výsledkom.

Poznámka: chybu by sme boli bývali odhalili, ak by sme pre vetvu „sú rovnaké“ vybrali napríklad trojicu x = 2, y = 1, z = 3.

4.1.2. Testovanie podľa špecifikácie – triedy ekvivalencie, analýza hraničných hodnôt

Keďže v praxi nie je možné otestovať všetky možné kombinácie vstupných údajov, hľadajú sa techniky výberu testovacích dát, ktoré aspoň zvýšia pravdepodobnosť toho, že prípadné chyby v programe budú odhalené. Jednou z takýchto techník je testovanie na základe tried ekvivalencie spojené s analýzou hraničných hodnôt.

Myšlienka tejto techniky je taká, že množinu vstupných hodnôt rozdelíme na triedy, v rámci ktorých by sa produkt mal správať rovnakým spôsobom. Ako testovacie vstupy použijeme reprezentantov týchto tried a hodnoty ležiace na hraniciach, resp. blízko hraníc medzi nimi.

Triedy ekvivalencie môžeme hľadať aj medzi výstupnými hodnotami. K takto očakávaným výstupným hodnotám potom (ručne) nájdeme zodpovedajúce vstupné hodnoty, ktoré použijeme ako testovacie vstupy.

Príklady:

Predpokladajme, že špecifikácia požiadaviek pre produkt hovorí, že tento musí byť schopný

spracovať súbory s ľubovoľným počtom záznamov od 1 po 16383, v ostatných prípadoch má ohlásiť chybu. (Nech „spracovať“ znamená napríklad spočítať priemernú hodnotu vybraných položiek v rámci týchto záznamov.) Ak produkt dokáže spracovať 34 záznamov a 14870 záznamov, potom je dosť veľká pravdepodobnosť, že bude fungovať aj pre, povedzme, 8252 záznamov. Pravdepodobnosť nájdenia chyby (ak existuje), je pravdepodobne rovnako veľká pre ľubovoľný testovací vstup v rozsahu 1 až 16383 záznamov. Obrátene, ak produkt funguje pre jeden prípad v rozsahu 1 až 16383, potom bude pravdepodobne fungovať pre ľubovoľný iný prípad v tomto rozsahu. Rozsah 1 - 16383 predstavuje triedu ekvivalencie, množinu testovacích hodnôt, kde každý člen je rovnako dobrý ako ľubovoľný iný. V našom prípade máme teda 3 triedy ekvivalencie:

- Trieda 1: Menej ako 1 záznam
- Trieda 2: 1 - 16383 záznamov
- Trieda 3: Viac ako 16383 záznamov

Testovanie technikou tried ekvivalencie potom vyžaduje výber jedného reprezentanta z každej z týchto tried.

Analýza hraničných hodnôt: skúsenosť ukazuje, že pravdepodobnosť nájdenia chyby sa zvyšuje vtedy, keď sa použije testovací vstup na hranici alebo blízko hranice tried ekvivalencie.

Takže v našom prípade by sa použili tieto hodnoty (napríklad):

- Test 1: 0 záznamov (člen triedy 1 a zároveň sused hraničnej hodnoty)
- Test 2: 1 záznam (hraničná hodnota)
- Test 3: 2 záznamy (sused hraničnej hodnoty)
- Test 4: 723 záznamov (člen triedy 2)
- Test 5: 16382 záznamov (sused hraničnej hodnoty)
- Test 6: 16383 záznamov (hraničná hodnota)
- Test 7: 16384 záznamov (člen triedy 3 a zároveň sused hraničnej hodnoty)

Podobne sa dajú vytvárať testovacie údaje na základe očakávaných obmedzení pre *výstupy*. Nech je minimálna hodnota zrážok pre sociálne poistenie $0.00 a maximálna $5328.90. Pri testovaní programu pre spracovanie miezd by mali testovacie prípady zahŕňať údaje vedúce k hodnotám $0.00 a $5328.90. Bolo by tiež užitočné vyskúšať vstupné údaje, ktoré by teoreticky mohli spôsobiť výsledok menší ako $0.00, resp. väčší ako $5328.90.

4.1.3. Testovanie podľa kódu

Techniky pre výber vstupných údajov pri testovaní podľa kódu:

- pokrytie príkazov (statement coverage): vyberáme vstupy tak, aby každý príkaz bol vykonaný aspoň raz,
- pokrytie vetvenia (branch coverage): vyberáme vstupy tak, aby každá vetva v rozhodovacom strome bola prejdená aspoň raz,
- pokrytie ciest (path coverage): vyberáme vstupy tak, aby každá cesta bola prejdená aspoň raz. Bohužiaľ, ako sme videli, týchto môže existovať neúnosne veľa. Hľadajú sa spôsoby, ako rozumne zredukovať počet otestovaných ciest bez toho, aby sa výrazne znížila efektívnosť testovania.

Praktické skúsenosti ukazujú, že neexistujú výrazné rozdiely v účinnosti metód testovania black-box, glass-box a testovania inšpekciou kódu.

4.1.4. Testovanie distribuovaných systémov a systémov pracujúcich v reálnom čase

Náročnosť testovania prudko narastá v prípade distribuovaných systémov a systémov pracujúcich v reálnom čase. Nie sú tu totiž splnené základné predpoklady, z ktorých sa pri testovaní „bežných" programov vychádza:

- inštrukcie programu sú vykonávané sekvenčne,
- beh programu ako celku je deterministický,
- vloženie ladiacich inštrukcií do programu nezmení jeho správanie.

Pri produktoch pracujúcich v reálnom čase je tiež ťažké generovať testovacie vstupy – často je jedinou možnosťou použitie simulátora (softvér pre jadrovú elektráreň, lietadlo, jednotku intenzívnej starostlivosti, ...).

Pri týchto produktoch, najmä v prípade, že ide o systémy, ktoré musia byť extrémne spoľahlivé, sa obvykle okrem techník spomínaných v tejto časti používajú aj:

- matematické dôkazy správnosti,
- simulácia,
- štatistické metódy – výber takej podmnožiny všetkých možných vstupov, na základe ktorej bude možné určiť pravdepodobnosť zlyhania produktu.

4.1.5. Poznámky k testovaniu

- Ukazuje sa, že počet chýb modulu súvisí s jeho veľkosťou určenou počtom riadkov (Lines of Code, LOC), presnejšie s počtom vykonateľných príkazov zdrojového kódu (Delivered Source Instructions, DSI) a s mierou zložitosti modulu. Ako miera zložitosti sa často (napriek svojim nedostatkom) používa McCabeho *cyklomatická zložitosť* (cyclomatic complexity), ktorá sa určuje ako počet binárnych rozhodnutí (predikátov) + 1.

  Napríklad Walsh analyzoval 276 procedúr v systéme Aegis. Zistil, že v procedúrach s cyklomatickou zložitosťou M aspoň 10 (ktorých bolo 23% z celkového počtu procedúr) sa vyskytovalo 53% z celkového počtu nájdených chýb. Naviac, procedúry s $M \geq 10$ mali o 21% viac chýb na riadok kódu ako procedúry s $M < 10$.

- Kedy ukončiť testovanie ? Na základe štatistických metód sa dá povedať, kedy prestať s testovaním tak, aby bol dosiahnutý stanovený stupeň spoľahlivosti produktu. V princípe platí, že čím dlhšie je produkt testovaný bez nájdenia chyby, tým väčšia je pravdepodobnosť, že je bez chýb.

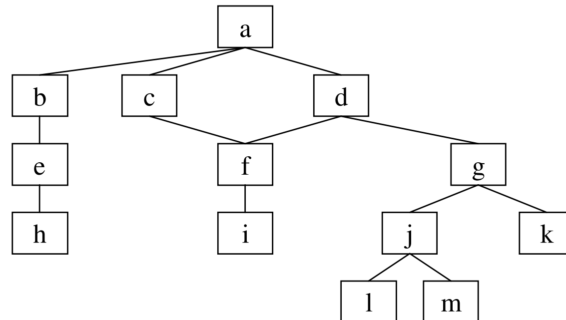
- Kedy „zlý" modul neopravovať, ale radšej napísať nanovo ? Motivácia: majme dva podobne veľké a podobne zložité moduly, A a B, ktoré boli testované približne rovnaký čas. Za tento čas sa v A našli 2 chyby, v B 47 chýb. Je pravdepodobné, že v B ešte dosť veľa chýb ostáva a že ani po ich opravení nebude B príliš kvalitný.

  Ukazuje sa, že distribúcia chýb nie je uniformná. Príklady: V systéme OS/370 bolo 47% chýb sústredených v 4% modulov. V systéme DOS/VS bolo pri testovaní nájdených 512 chýb v 202 moduloch, pričom 112 modulov bolo s jednou chybou a naopak, našli sa moduly s 14, 15, 19 a 28 chybami. Modul so 14 chybami nebol pritom príliš veľký ani zložitý (ostatné tri boli). Tento typ modulu je prirodzeným kandidátom na prepísanie.

  Možnosťou, ako vyššie položenú otázku riešiť, je pre každý z modulov určiť hranicu počtu chýb, pri prekročení ktorej sa modul má navrhnúť a napísať nanovo. Určenie tejto hranice môže vychádzať zo skúseností s údržbou podobných modulov a samozrejme zo zložitosti a veľkosti daného modulu.

4.2. Integrácia

Otázky integrácie budeme ilustrovať na produkte pozostávajúcom z nasledujúcich modulov.



Poznámka: Tento diagram sa nazýva diagram spojenia modulov (module interconnection diagram) a znázorňuje vzťahy typu „modul **a** volá moduly **b**, **c**, **d**".

Jedným z možných prístupov k integrácii je implementovať a testovať každý modul osobitne a potom celý systém naraz spojiť a otestovať ako celok. Tento prístup má 2 problémy:

- Na to, aby sme boli schopní otestovať napríklad modul g, potrebujeme vytvoriť náhradu za moduly j, k („stubs") a d („driver"). Ak testujeme každý modul zvlášť, je potrebné týchto náhrad vytvoriť relatívne veľa.
- Pri testovaní produktu ako celku sa budú pomerne ťažko hľadať chyby, potenciálne totiž môžu byť na ľubovoľnom mieste v produkte.

Inou možnosťou je postup **zhora nadol**: pred tým, ako implementujeme a testujeme modul X, musíme implementovať a otestovať moduly v diagrame spojenia modulov umiestnené nad ním.

Výhody:

- Pri odhaľovaní chýb súvisiacich s integráciou postupujeme po krokoch.
- Nemáme toľko práce s vytváraním náhrad za moduly pri testovaní – musíme síce robiť prázdne podriadené moduly, avšak tieto sa môžu využiť pri následnej implementácii týchto modulov.
- Ak je nejaká chyba v celkovom návrhu systému, nájde sa pomerne skoro – vďaka tomu, že moduly obsahujúce rozhodovanie sa implementujú najskôr, pretože tieto sa obvykle vyskytujú vyššie v diagrame spojenia modulov.

Problém: moduly na spodnej úrovni nebudú tak dôkladne otestované, čo znižuje možnosť ich efektívneho opätovného použitia. Situácia je ešte horšia, ak sú vyššie moduly naprogramované *defenzívne* (čo je samozrejme správny prístup), t.j. ak pred zavolaním spodného modulu sami otestujú parametre, ktoré mu odovzdávajú. Príklad:

```
if (x >= 0)
  y = compute_square_root (x, &error_flag);
```

Pri postupe **zdola nahor** postupujeme opačne a výhody, resp. nevýhody sú presne opačné ako v predchádzajúcom prípade.

Pri **„sendvičovom postupe"** sa moduly rozdelia na logické a výkonné, pričom logické moduly sa implementujú a integrujú zhora nadol a výkonné zdola nahor. Na záver sa obidve skupiny modulov spoja. Takto sa na jednej strane pomerne skoro nájdu chyby v „horných" moduloch a na druhej strane sa „spodné" (výkonné) moduly dôkladne otestujú.

Po ukončení integrácie sa produkt testuje ako celok. Testuje sa jeho funkčnosť podľa

špecifikácie požiadaviek a ďalšie vlastnosti, napríklad doba odozvy, spoľahlivosť, robustnosť. Skúma sa správanie produktu pri plnej záťaži, napríklad v situácii, keď všetci používatelia vyvolajú niektorú z funkcií naraz, resp. keď treba spracovať veľké vstupné súbory. Kontroluje sa úplnosť a správnosť dokumentácie.

Na záver sa vykonáva akceptačné testovanie. Toto vykonáva zadávateľ, a to v prostredí, kde bude produkt prevádzkovaný a na svojich vlastných údajoch.

4.3. Údržba

Keď je úspešne ukončené akceptačné testovanie, produkt je odovzdaný zadávateľovi. Všetky ďalšie zmeny predstavujú údržbu.

Prieskum v 69 organizáciách v roku 1978 ukázal, že

- 17,5% času venovaného údržbe tvorila **korektívna údržba**: opravovanie chýb, ktoré neboli odhalené počas vývoja,
- 18% tvorila **adaptívna údržba**: prispôsobovanie produktu zmeneným vonkajším podmienkam,
- 60,5% tvorila **perfektívna údržba**: rozširovanie produktu, zlepšovanie jeho vlastností a podobne.

Údržba je najnáročnejšou časťou vývoja softvéru, najmä preto, lebo zahŕňa všetky ostatné etapy vývoja. Napriek tomu je často prideľovaná začiatočníkom a považovaná za menej hodnotnú prácu. Opak je pravdou a údržbu musia robiť špičkoví profesionáli.

Príklad: po zahlásení chyby používateľom musí programátor zodpovedný za údržbu:

- zistiť, či nie je náhodou chyba v tom, akým spôsobom používateľ produkt používa, prípadne či nie je chyba v používateľskej dokumentácii,
- ak je skutočne chyba v programe, je potrebné ju nájsť (tu treba viac než bežné schopnosti ladenia),
- ďalej treba chybu opraviť bez toho, aby sa do programu zaviedla ďalšia chyba (tzv. regresná chyba) – toto vyžaduje podrobné pochopenie fungovania programu, často však je potrebná dokumentácia neaktuálna, prípadne úplne chýba,
- upravený program treba dôkladne otestovať, jednak na základe testovacích údajov zostrojených špeciálne pre tento prípad a tiež na základe pôvodných testovacích údajov,
- všetky vykonané zmeny treba zdokumentovať.

Rovnako široká škála činností je spojená s adaptívnou a perfektívnou údržbou.

4.4. Literatúra

Stephen R. Schach: **Software Engineering**, 2nd ed., Aksen Associates, 1993