Techniky testování softwaru

Mýtus praví, že ten, kdo je skutečně dobrý programátor nedělá chyby.

Když se koncentruje, když používá strukturované programování, návrh shora dolů atd., tak chyby neudělá.

Mýtus říká, že děláme chyby, protože nejsme dobří.

Testování je jedna z důležitých technik zabezpečení jakosti výsledného produktu.

Charakteristika testování

- Testování je proces spouštění programu za účelem nalezení chyb.
- Dobrá testovací data jsou taková, která s velkou pravděpodobností objeví dosud neobjevené chyby.
- Úspěšný test je takový, který objevil dosud neobjevené chyby.

Principy testování

- Testy by se měly vztahovat k požadavkům zákazníka
- Testy by měly být plánovány v předstihu
- *Princip Pareto (pravidlo 80:20)* : většina všech neobjevených chyb má původ v několika málo modulech
- Testování by mělo začít testováním “v malém” a pokračovat testování “ve velkém”.
- Úplné otestování není možné
- Testování by mělo být vedeno nezávislou třetí stranou

Testovatelnost

Výčet vlastností vedoucích k dobré testovatelnosti:

Spustitelnost **čím lépe SW pracuje, tím účinněji může být otestován**

chyby nebrání běhu programu, může být testován a vyvíjen současně

Přehlednost **testuješ, co vidíš**

různé výstupy pro různé vstupy, stavy systému a proměnné jsou viditelné během chodu, faktory ovlivňující výstup jsou viditelné, nesprávné výstupy jsou lehce identifikovatelné, vnitřní chyby jsou automaticky detekovány a zaznamenávány, je dostupný zdrojový kód

Kontrolovatelnost **čím lépe lze software kontrolovat/řídit, tím spíše lze testování automatizovat a optimalizovat**

všechny možné výstupy jsou generovány nějakou kombinací vstupů, veškerý kód je spustitelný nějakou kombinací vstupů, vstupní a výstupní formát je konsistentní a strukturovaný, testy mohou být specifikovány, automatizovány a reprodukovány.

Testovatelnost (pokr.)

Dekomponovatelnost **kontrolováním rozsahu testování můžeme rychleji oddělit problémy a provést následné testy**

Jednoduchost **čím méně máme testovat, tím rychleji můžeme testy provést**

jednoduchost funkce, struktury, kódu

Stabilita **čím méně změn softwaru, tím lépe**

Změny nejsou časté, jsou řízené, neovlivní provedené testy, jsou srozumitelné - srozumitelný návrh, srozumitelné závislosti mezi vnitřními, vnějšími a sdílenými komponentami, dostupnost a srozumitelnost technické dokumentace

Vlastnosti dobrého testu

- Dobrý test s velkou pravděpodobností objeví chybu.
- Dobrý test není redundantní
- Dobrý test by měl objevit celou třídu chyb
- Dobrý test by neměl být ani moc jednoduchý ani moc složitý

Testy by se měly spouštět separátně, aby jejich případné vedlejší efekty nepřekryly chyby.

Návrh testu

black-box testing (funkcionální testování) - testování správného provádění všech navržených funkcí produktu *(ale nemusí postihnout všechny podrobnosti implementace)*

white-box testing (strukturální testování) - testování toho, zda všechny vnitřní komponenty správně fungují *(ale nemusí postihnout chybějící alternativy)*

White-box testing

glass-box testing,

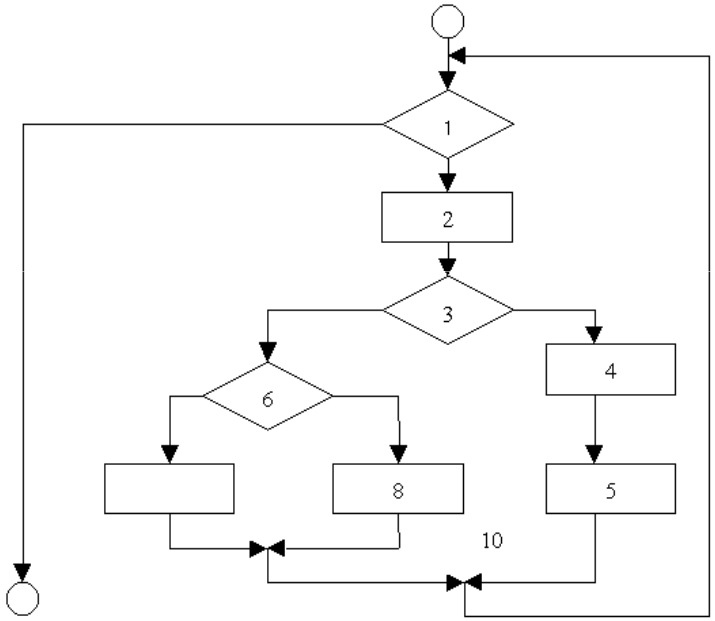
strukturální testování

White-box testing

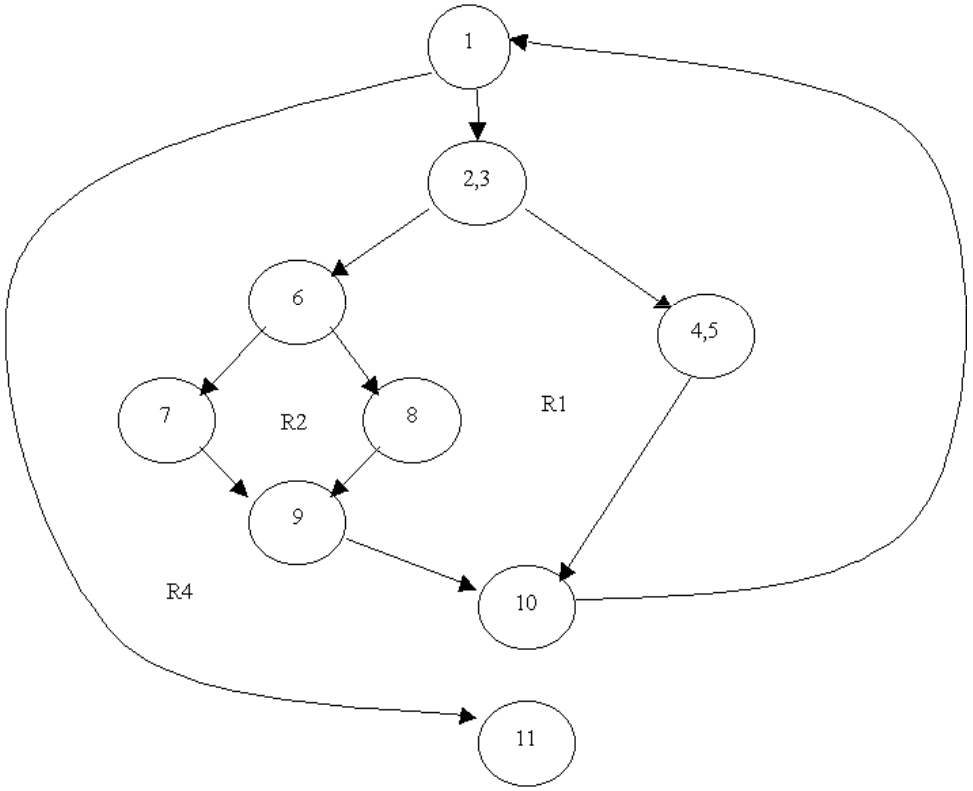
- všechny samostatné cesty uvnitř modulu budou provedeny alespoň jednou
- všechny podmínky se projdou podél větve s hodnotou ANO (true) i podél větve s hodnotou NE (false)
- projde všechny cykly
- prověří vnitřní datové struktury

Testování podle základních cest

vývojový diagram



graf toku řízení



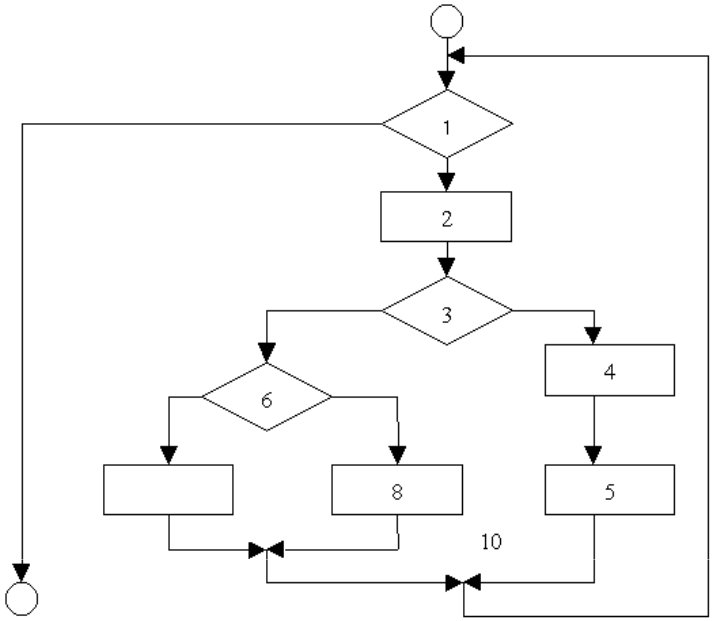
Cyklomatická složitost

Je metrika složitosti programu,

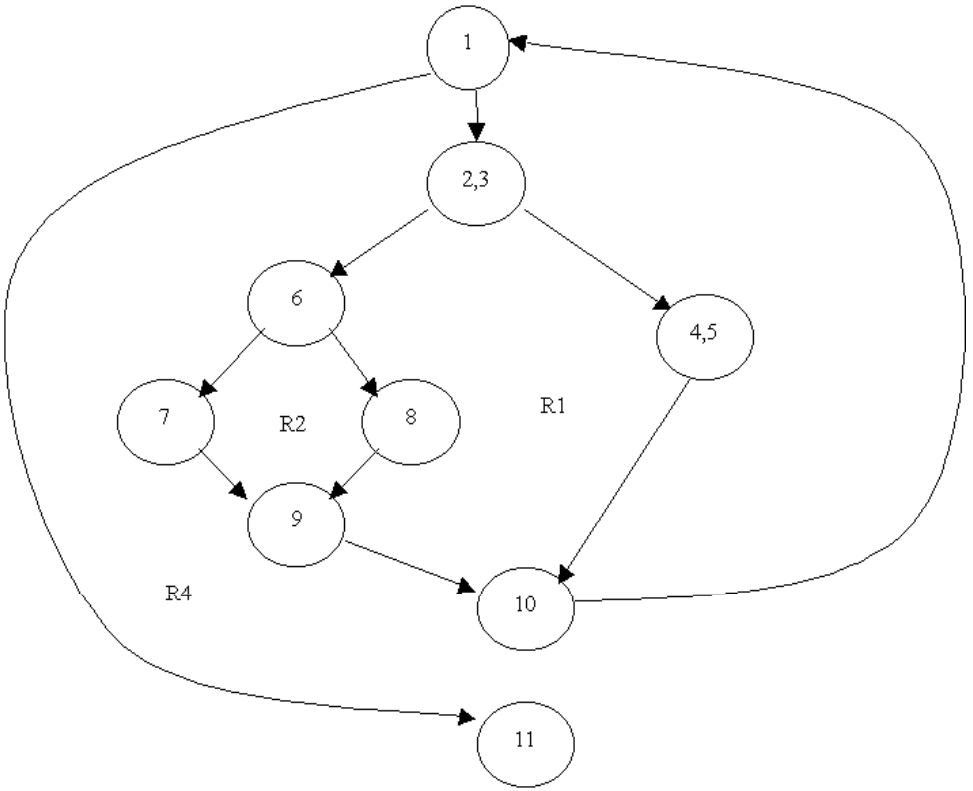
která udává počet **nezávislých cest** v programu.

Nezávislá cesta je taková, která prochází od začátku do konce programem a prochází alespoň jedním příkazem nebo podmínkou, kterým neprochází jiná taková cesta.

vývojový diagram



graf toku řízení



cesta1: 1-11

cesta2: 1-2-3-4-5-10-1-11

cesta3: 1-2-3-6-8-9-10-1-11

cesta4: 1-2-3-6-7-9-10-1-11

Cyklomatická složitost:

V(G) je počet oblastí v grafu toků

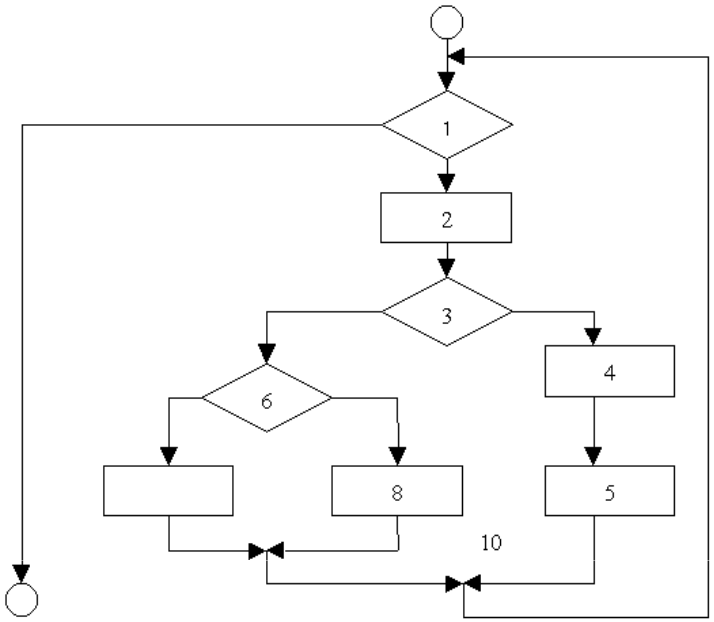
$$V(G) = E - N + 2$$

kde *E je počet hran grafu G*

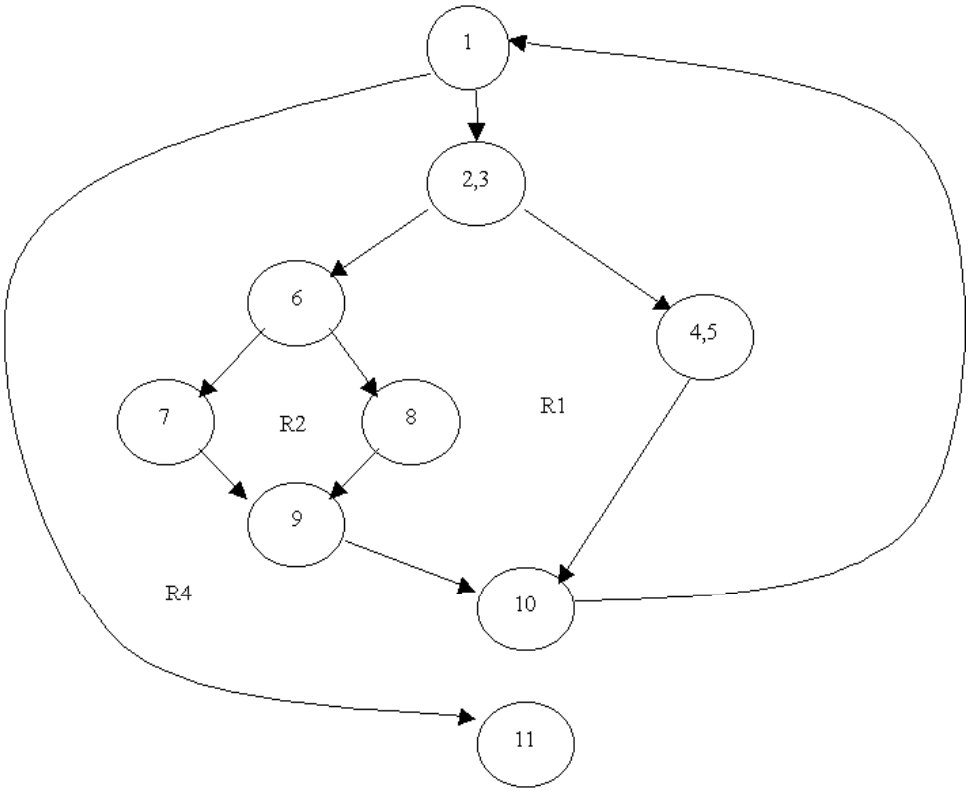
N je počet uzlů.

Počet nezávislých cest je menší nebo roven číslu V(G), současně je to počet testů, které musí být provedeny, aby byl každý příkaz proveden alespoň jednou.

vývojový diagram



graf toku řízení



cesta1: 1-11
cesta2: 1-2-3-4-5-10-1-11
cesta3: 1-2-3-6-8-9-10-1-11
cesta4: 1-2-3-6-7-9-10-1-11

V(G) = E-N+2

E=11 N=9

V(G) = 11-9+2=4

Návrh testovacích dat

- vytvoř graf toku řízení
- urči cyklomatickou složitost
- urči množinu nezávislých cest
- odvoď testovací data pro každou cestu

Testování podle řídicí struktury

Testování podmínek - testování všech podmínek v programu

- testování větví (branch testing) - otestování obou větví podmínky
- testování domén (domain testing) - otestování všech kombinací vztahů proměnných testu *(např. pro A relace B, budou tři testy, kdy A>B, A<B, A=B)*

Testování chyb v booleovském operátoru, booleovské proměnné, chyby v závorkách, relačním operátoru, aritmetickém výraze……

Testování datového toku

Metoda DU

Pro každý příkaz S definujme dvě množiny: DEF(S) a USE(S). Jsou to množiny proměnných, které jsou v něm definovány a které jsou v něm používány.

Proměnná X , která je definovaná v příkazu S, ***žije*** v příkazu S', jestliže existuje cesta z S do S', která neobsahuje jinou definici proměnné X.

Řetěz definice-použití (DU chain) proměnné X je

trojice [X, S, S'], kde X patří do DEF(S) a USE(S') a žije v S'.

Strategie testování spočívá v tom, že každý DU řetěz bude pokryt alespoň jednou.

Testování cyklů

Jednoduchý cyklus

- přeskoč cyklus
- projdi právě jednou
- projdi dvakrát
- projdi m krát, m< n, kde n je max počet průchodů
- n-1, n, n+1 průchodů

Vložené cykly

(redukovaná strategie jednoduchého cyklu)

1. začni nejvnitřnějším cyklem, ostatní cykly nastav na minimum průchodů
2. proveď všechny testy pro jednoduchý cyklus
3. postupuj směrem k vnějšímu cyklu, vnější cykly nastav na minimum průchodů, vnitřní na “typickou hodnotu”
4. pokračuj dokud neotestuješ všechny cykly

Zřetězené cykly

pokud jsou nezávislé, postupuj jako u samostatného jednoduchého cyklu, pokud jsou navzájem závislé (mají např. stejný čítač) postupuj jako u vložených cyklů

Nestrukturované cykly by měly být pokud možno předělány na strukturované

Black-box testing

behavior testing,

partition testing,

funkcionální testování

Black-box testing

Je založeno na funkčních požadavcích.

Vzhledem k white-box testing jde o doplňkovou nikoli alternativní metodu.

Hledá

- nesprávné nebo chybějící funkce
- chyby rozhraní
- chyby datových struktur nebo přístupu k externím databázím
- chybné chování (performance)
- chyby inicializace nebo skončení

Aplikuje se až po white-box testing.

Hledají se odpovědi na otázky:

- Jaká je validita funkcí?
- Jaké třídy vstupů tvoří dobrá testovací data?
- Je systém zvlášť citlivý na některá vstupní data?
- Jaká jsou hranice tříd vstupních dat?
- Jaké rozsahy a hodnoty dat systém toleruje?
- Jaký efekt na systém mají speciální kombinace dat?

Testy by měly redukovat počet dalších testů.

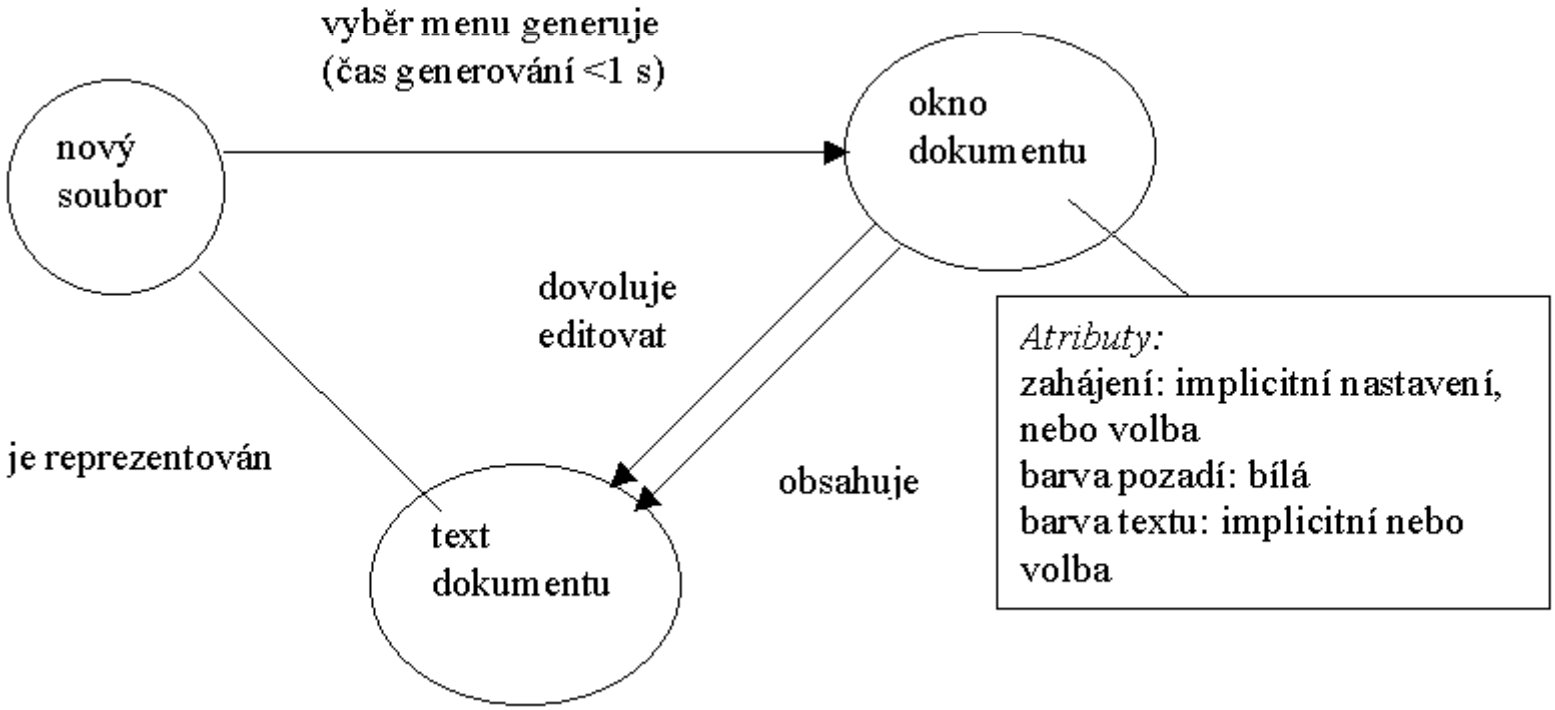
Testy by měly napovědět něco o přítomnosti či absenci tříd chyb.

Metody založené na grafu

Ověřit, že všechny objekty jsou v požadovaných vzájemných vztazích.

Znázornit vztahy graficky a navrhnout test, který pokrývá graf.

Objekty grafu mohou být jak datové objekty tak i programové objekty (moduly, nebo množina programových příkazů).



Modelování

- ***Transaction flow modeling.***

Uzly jsou kroky nějaké transakce (např. leteckého rezervačního systému), hrany jsou logické vztahy mezi nimi. (např. vstup *je následován* ověřením práv). Je možné vycházet z DFD.

- ***Finit state modeling.***

Uzly jsou různé uživatelsky sledovatelné stavy systému (např. sled obrazovek), hrany reprezentují přechody mezi stavy. Vychází se ze STD.

- ***Data flow modeling.***

Uzly jsou datové objekty, hrany reprezentují transformace jednoho datového objektu na druhý.

- ***Timing modeling.***

Uzly jsou programové objekty a hrany reprezentují sekvenční následnost mezi nimi. Mohou mít váhy - předpokládaný čas provedení.

Metoda rozdělení do tříd ekvivalencí *(Equivalence Partitioning)*

Rozdělení vstupní domény programu do tříd dat a odvození testovacích dat.

Třídy ekvivalence odvozené ze vstupní podmínky.

- Pokud vstupní podmínka specifikuje rozsah, jedna třída bude pro platná data a dvě pro neplatná.
- Pokud vstupní podmínka požaduje specifickou hodnotu, bude jedna platná a jedna neplatná.
- Pokud vstupní podmínka specifikuje množinu dat, bude jedna platná a jedna neplatná.
- Pokud vstupní podmínka je booleovská, bude jedna platná a jedna neplatná.

Příklad: automatický bankovní systém

Uživatel komunikuje s bankou použitím osobního počítače. Posílá data následujícího formátu:

kód oblasti, prefix, suffix, heslo, příkaz.

Vstupní podmínky:

Kód oblasti: booleovská podm. - kód oblasti je nebo není přítomen, rozsah - hodnoty mezi 200 a 999 se speciálními výjimkami.

Prefix: rozsah - hodnoty >200, s žádnými nulami

Suffix: hodnota - čtyři číslice

Heslo: booleovská podm. - heslo je nebo není přítomné, hodnota - šestimístný znakový řetězec

Příkaz: množina obsahující možné příkazy

Metoda analýzy okrajových hodnot
(Boundary Value Analysis)

Z málo známých důvodů se velký počet chyb vyskytuje kolem okrajových podmínek vstupních dat spíše než uprostřed.

Doplňuje předešlou metodu. Vybírají se data blízká hranicím tříd ekvivalence.

Uvažují se jak vstupní podmínky tak i výstupní podmínky.

Pokud vstupní podmínka specifikuje rozsah od A do B, testovací data mají být A, B, bezprostředně nad a pod A a B.

Pokud vstupní podmínka specifikuje počet hodnot, testovací data mají obsahovat minimální a maximální hodnotu.

Body 1 a 2 aplikuj pro výstupní specifikace. *(např. výstup je tabulka hodnot, navrhněme test, který by produkoval maximální (minimální) hodnoty tabulky.)*

Pokud vnitřní datová struktura má předepsané ohraničení (např. limit položek), navrhni test ohraničení.

Srovnávací testování (Comparison Testing, back-to-back testing)

Pro požadavky zvýšené spolehlivosti (navigace letadel, řízení nukleární elektrárny a pod.) mohou být některé časti (HW i SW) redundantní, zpracovávané nezávislými týmy ze stejných specifikací.

Každá aplikace je testována se stejnými daty a výsledky by měly být stejné. Pak jsou všechny verze prováděny paralelně s porovnáním reálného času výsledků a zajištění konsistence.

Někdy se pouze kritické části programují paralelně, ale předává se jen jedna verze.

Testy se provádějí podle některé z výše uvedených metod.

Testování pro speciální prostředí a aplikace

Testování GUI

série obecných testů

pro okna:

- Bude okno správně otevřeno klávesovými příkazy či příkazy menu?
- Může být okno zmenšeno/zvětšeno, přesunuto, schováno?
- Jsou všechna data uvnitř okna dostupná myší, funkční klávesou, šipkami a klávesnicí?
- Je okno správně obnoveno, když se překryje a znovu zavolá?
- Jsou všechny funkce v okně dostupné, když je potřeba?
- Jsou všechny funkce v okně spustitelné?

- Jsou všechny roletová menu, nástrojové lišty, dialogová okénka, knoflíky, ikony a ostatní řídicí prvky správně zobrazeny?
- Je aktivní okno správně vysvícené?
- Jsou všechny roletová menu, nástrojové lišty, dialogová okénka, knoflíky, ikony a ostatní řídicí prvky správně zobrazeny?
- Je aktivní okno správně vysvícené?
- Pokud se užívá multitasking (zpracování několika úloh současně), jsou všechna okna správně a ve správný čas aktualizovaná?
- Nezpůsobí vícenásobné kliknutí myši, nebo kliknutí mimo okno neočekávaný vedlejší efekt?
- Objevují se správně zvuková nebo barevná upozornění?
- Zavře se okno správně?

Pro roletová menu a operace myši

- Je zobrazena správná příkazová lišta v odpovídajícím kontextu?
- Jsou správně zobrazeny doplňující údaje (např. čas)?
- Pracuje správně stahování menu?
- Jsou všechny funkce v menu a případné podfunkce správně zobrazeny?
- Jsou všechny funkce v menu správně ovladatelné myší?
- Dají se příslušné funkce spustit správně odpovídajícím klávesovým příkazem?
- Jsou funkce správně vysvíceny nebo neosvíceny podle daného kontextu?
- Dělá každá funkce, co má?
- Jsou názvy funkcí názorné a srozumitelné (self-explanatory)?
- Je pro každou položku menu dostupný help a je kontextově závislý?
- Jsou operace myši správně rozpoznány?
- Pokud je požadované vícenásobné kliknutí, je správně rozpoznáno?
- Je správně zobrazen a změněn kurzor v daném kontextu?

Pro datové vstupy:

- Jsou alfanumerické datové vstupy správně zavedeny do systému?
- Jsou nesprávná data správně rozpoznána?

Navíc se může použít metoda konečných automatů pro odvození doplňujících testů.

Testování uživatelské dokumentace a helpu

(Je frustrující, když postupujete podle dokumentace a dostanete nesprávné výsledky)

Testování dokumentace musí být součástí plánu testování.

Dvě fáze testování:

- inspekce prověřuje obsah a formu samotné dokumentace
- živé (life) testy prověřují dokumentaci ve spojení s programem

Používají se analogické metody jako pro black-box testování.

- Popisuje dokumentace správně provedení změny módu?
- Je dobře popsaná posloupnost interaktivní komunikace?
- Jsou uvedené příklady správné?
- Je terminologie, popisy menu, a systémových odezev popsána konsistentně s programem?
- Je relativně snadné nalézt v dokumentaci případnou radu, návod?
- Je popsáno chování v případě problému? (trableshooting)
- Má dokumentace správný obsah a rejstřík?
- Je grafická úprava přehledná (a není matoucí)?
- Jsou všechna chybová hlášení podrobně popsána v dokumentaci?
- Jsou hypertextové odkazy správné?

Závěr:

Testování má zvýšit pravděpodobnost, že v programu nejsou chyby.

White-box testing vychází ze znalosti programové řídicí struktury. Testy by měly vyzkoušet alespoň jeden průchod všemi příkazy a vyzkoušet všechny logické podmínky.

Je to testování v malém, pro malé programy a moduly.

Black-box testing hledá chyby ve funkčním chování, bez ohledu na jejich implementaci. Je zaměřeno na informační doménu, odvozuje testovací data ze vstupních a výstupních podmínek.

Speciální metody testování jsou pro různé typy programů a aplikačních oblastí - GUI, klient/server, real time, dokumentace a help.

Zkušení programátoři říkají:

“Testování nikdy nekončí, jenom se přesouvá od programátora k zákazníkovi.

Pokaždé, když zákazník spouští program, tak ho vlastně testuje.”