Testování

Účel testování

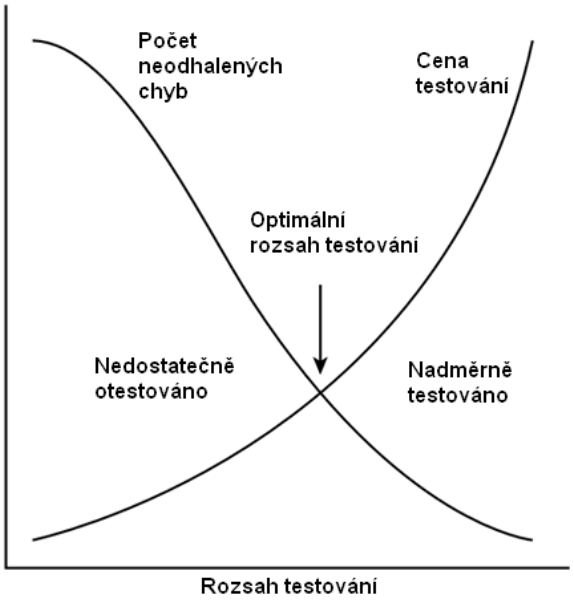
Testování souvisí s výskytem chyb v programu. Statistiky uvádí, že programy, i když dobře sestavené, obsahují v průměrném kódu s průměrně složitými příkazy 1 až 3 chyby na 100 příkazů.

Není možné kompletně otestovat program

Ani u jednoduchých programů nelze zajistit zpravidla jejich kompletní otestování. Stačí, aby nastala aspoň jedna z následujících situací:

- Počet možných vstupních hodnot je příliš veliký.
- Počet možných výstupních hodnot je příliš veliký.
- Počet možných větvení v programu je velmi vysoký (podmíněné příkazy, cykly, apod.).
- Specifikace požadavků je subjektivní. Co programátor považuje za normální, klient může považovat za chybu.

Proto ani rozsáhlým testováním nedokážeme zajistit, že v programu nejsou již žádné chyby. Cílem testování je snížit riziko výskytu chyb na přijatelnou úroveň.



Testováním lze snížit výskyt chyb na únosnou úroveň, ale testování nezaručí, že už žádná chyba v programu nezůstane.

Realita testování

Někdy nalezené chyby nejsou odstraněny, neboť:

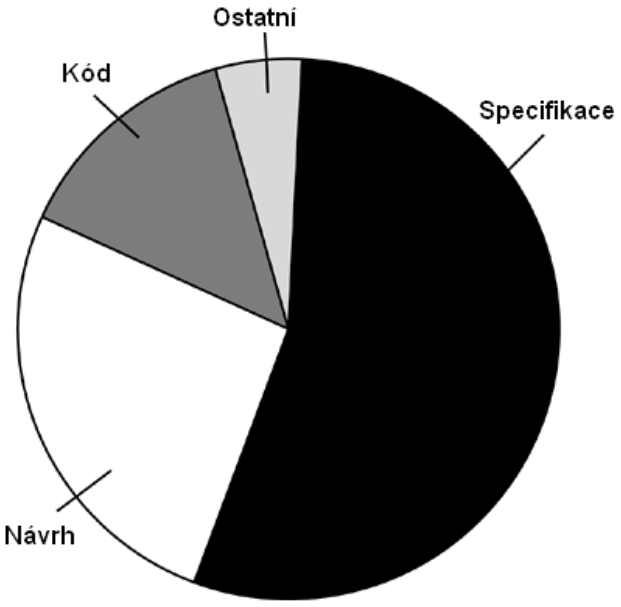
- Není dostatek času - blíží se konec termínu pro dokončení projektu.
- Je příliš riskantní chybu opravovat – programy jsou mnohdy tak komplikované, že oprava jedné chyby může vyústit ve vznik nových chyb a není už čas na rozsáhlejší opětný test systému.
- To nestojí za to tuto chybu opravovat – tato funkce systému se stejně bude používat jen zřídka nebo k chybě dojde jen v určité situace, které je možné se při používání systému vyhnout.

Co je chyba

V programu je chyba, jestliže nastane některá z následujících situací:

1. Program nedělá něco, co specifikace uvádí, že by měl dělat.
2. Program dělá něco, o čem specifikace uvádí, že by neměl dělat.
3. Program dělá něco, co ve specifikaci vůbec není uvedeno, že by měl dělat.
4. Program nedělá něco, co sice ve specifikaci ani není uvedeno, nicméně by to mělo v ní být.
5. Je těžké naučit se s programem pracovat, obtížně se ovládá, je pomalý.

Příčiny chyb



Je několik skutečností, proč specifikace je největším zdrojem chyb. V mnoha případech není ani vytvořen dokument specifikace požadavků. Dále specifikace mnohdy není úplná, neustále se mění, vývojový tým s ní není dostatečně seznámen. To všechno vede k chybám v realizaci projektu.

Vedle přímého testování lze kvalitu ovlivnit nebo zvýšit i dalšími postupy, zejména:

- **Prohlédnutí (inspekce) kódu**
  Tato metoda může odhalit některé chyby, které testování neodhalí.
- **Styl programování**
  Kód, který je psán s cílem, aby byl přehledný, snadno se testoval a bylo možné se v něm snadno orientovat, může snížit výskyt chyb. I když takový kód naopak může být na úkor jeho efektivnosti nebo jeho stručnosti.
- **Statická analýza**
  Tato metoda zahrnuje všechno, co lze zkontrolovat ve zdrojovém kódu během překladu nebo v souvislosti s překladem. Patří sem zejména promyšlené používání datových typů a jejich kontrola, která může vyloučit řadu chyb (překladače běžně poskytují zprávy o nesouladu datových typů v programu).

Význam termínů

- **Testování – ladění**
  Cílem testování je zjistit, že v programu jsou chyby. Cílem ladění je chyby nebo nedostatky v programu, které způsobují jeho neuspokojivou funkci, najít a navrhnout a implementovat změny, jež chyby odstraní.
  Mezi testováním a laděním jsou významné rozdíly. Testování lze dělat, aniž známe strukturu programu. Naproti tomu odstranit chybu může jen ten, kdo program zná. Navíc testy lze sestavit tak, že mohou probíhat automaticky, ale na opravu chyb žádné automatické postupy nemáme.
- **Funkční testování – strukturální testování**
  Funkční testování bere program jako „černou skříňku“. Jsou vkládány vstupní hodnoty a kontroluje se, zda výstup odpovídá požadovanému chování programu. Funkční testování se na program dívá z pohledu uživatele.
  Strukturální testování se zabývá implementačními aspekty programu – stylem programování, kontrolami začleněnými do programu a dalším.
- **Programátor – testující pracovník**
  Funkční testy může dělat kdokoliv – není k nim zapotřebí znát strukturu programu. Znalost programu je potřebná jen pro strukturální testy. Nicméně znalost programu v případě funkčních testů přispívá k tomu, že neděláme zbytečně testy, kdy funkce rozdílné z pohledu uživatele jsou v programu zajišťovány stejnými částmi programu, které byly již otestovány. Na druhé straně ten, kdo program nezná, testuje i takové funkce, o kterých je tvůrce programy přesvědčen, že v nich chyba nemůže být, a proto je ani netestuje.
- **Prostředí programu**
  Je technické vybavení (počítač, síťové komponenty, komunikační linky apod.) a programové vybavení (operační systém, další programy apod.) potřebné k tomu, abychom program mohli spustit. I když technické a standardní programové vybavení mohou způsobit chybné chování našeho programu, stává se to velmi zřídka, neboť obojí je zpravidla velmi stabilní a jejich spolehlivá funkce je ověřena jejich častým používáním.

Úrovně testování

Jsou tři typické úrovně testování programového vybavení:

1. **Testování komponenty nebo jednotky programu**. Komponenta je nejmenší jednotka systému, kterou lze přeložit a sestavit do spustitelné formy a následně spustit v testovacím prostředí. Cílem je zjistit, že jednotka nevyhovuje některým funkčním specifikacím nebo její implementace nesplňuje strukturu danou návrhem. Takto odhalené nedostatky označujeme jako chyby jednotky.
   Komponenta může být jak jedna jednotka, tak může vzniknou integrací více jednotek. Podle této definice může být velikost komponenty různá, může to být až celý systém. Testováním odhalené funkční nebo strukturální nedostatky komponenty označujeme jako chyby komponenty.
2. **Testování procesu integrace**. Proces integrace je spojování komponent ve větší komponentu. Dělá se v situaci, kdy samotné komponenty, z kterých bude sestavena větší komponenta, jsou otestovány a pracují uspokojivě. Testování je zaměřeno na zjištění, zda jejich spojení nezpůsobuje chyby nebo je nekonzistentní. Nejde zde

o testování komponenty, nýbrž o zjištění, zda jednotlivé komponenty nemají vzájemně nekompatibilní rozhraní, nepracují jinak s daty apod.

3. **Testování systému**. Systém je komponenta, která obsahuje všechny dílčí komponenty. Cílem tohoto testování je odhalit chyby, které nespadají do kategorie chyb způsobených nekonzistencí mezi jednotlivými komponentami nebo chybnou interakcí mezi komponentami. Jde o nalezení nedostatků, které lze zjistit jen právě testováním celého systému nebo jeho podstatné části. Může to zahrnovat testování výkonnosti, bezpečnosti, nastavení konfigurace systému, vlastního spouštění systému a zotavení systému po výpadku.

Lze udělat kompletní testování?

K testování můžeme použít 3 přístupy. Ale ani jeden z nich běžně nezajistí kompletní otestování programu.

1. **Funkční testování**. I když možných vstupních hodnot je zpravidla konečný počet, nelze je všechny v rozumném čase otestovat. Například je-li vstupem 32-bitové číslo, znamená to přes 4 miliardy možných vstupních hodnot.
2. **Strukturální testování**. Znamená navrhnout testy, které projdou všechna možné cesty (větvení) v programu a nejen to, u cyklů otestovat všechny možné počty jejich průchodů. Testování všech možných cest a možných počtů průchodů cyklů je značně nepraktické, i když v jednodušších případech by to bylo proveditelné.
3. **Formální důkaz korektnosti**. Znamená to formálním aparátem popsat každý příkaz programu a induktivním způsobem dokázat, že poskytuje korektní výsledek pro všechny možné vstupní hodnoty. To je typicky tak pracné a náročné, že testování by bylo neúnosně drahé.

Klasifikace chyb

Významnost chyby

Významnost chyby závisí na následujících okolnostech:

- **Četnost výskytu**. Uvádí, jak často se jednotlivé druhy chyby v praxi vyskytují. Chybám, které se vyskytují častěji, je účelné věnovat větší pozornost.
- **Cena opravy**. Cena opravy chyby je součtem nákladů na nalezení chyby a nákladů na vlastní opravu chyby. Výrazně roste s velikostí testovaných komponent nebo s velikostí testovaného systému.
- **Cena instalace**. Cena instalace udává závislost nákladů na opravu na počtu instalací. Nejmenší je v případě jedné instalace programu. Je-li ovšem instalací systému velký počet, mohou být náklady na opravu chyby ve všech instalacích značné a cena distribuce oprav může výrazně převýšit cenu vlastní opravy.
- **Důsledky.** Zde bereme v úvahu, jaké velké škody chyba způsobila.

Významnost chyby můžeme ohodnotit metrikou:

významnost = četnost_výskytu * (*cena_opravy* + *cena_instalace* + *cena_důsledků*)

Jak stanovit významnost důsledků chyb

Důsledky chyb můžeme například ohodnotit stupněm 1 až 10 podle kritérií:

1. **Velmi mírná**. Je chyba, která má jen estetickou povahu – například překlep v nabídce programu, nezarovnané hodnoty na výstupu.
2. **Mírná**. Výstup programu je zavádějící nebo nadbytečný. Taková chyba mnohdy i zbytečně snižuje výkonnost systému.
3. **Obtěžující**. Chování systému působí cize. Například jména jsou zkrácena nebo zkomolena. Vypisují se hodnoty, i když jsou třeba nulové a nemají žádný význam.
4. **Rušivá**. Systém odmítá provést transakce, i když vstup je v pořádku. Například systém při přihlášení odmítne uživatelské jméno, ačkoliv je správné. Nebo bankomat nepřijme kartu, třebaže je v pořádku.
5. **Závažná**. Ztratí se informace o transakci (transakce je přitom uskutečněna). Například výběr z účtu proběhne, nicméně vybíraná částka není odečtena z účtu.
6. **Velmi závažná**. Systém učiní chybnou transakci. Například při vkladu na účet vloženou částku z účtu odečte místo toho, aby o ni stav účtu zvýšil.
7. **Mimořádně závažná**. Je problém, který není omezen jen na několik uživatelů nebo některé typy transakcí. Je chyba, která se projevuje často a nahodile místo toho, aby se objevovala jen sporadicky nebo v neobvyklých případech.
8. **Nepřijatelná**. Vyskytuje se již nějakou dobu trvající nevratná ztráta dat, přičemž tento problém není navenek tak zřejmý, abychom ho hned na začátku odhalili. Zde už většinou lze tuto situaci řešit jen ukončením činnosti systému.
9. **Katastrofická**. Selhání systému, kdy systém sám ukončil svoji činnost.
10. **Infekční**. Horší než katastrofická chyba může být jen situace, kdy systém způsobí chyby nebo selhání jiných připojených systémů, i když systém sám ani neselže.

Závažnost chyb je relativní

Při hodnocení závažnosti chyby je nutno vzít v úvahu, že tato závisí rovněž na řadě vnějších faktorů, jako je prostředí, způsob použití systému a další. Vezměme například faktory:

- **Závislost na ceně opravy**. Cena opravy nijak nesouvisí se závažností chyby. Můžeme mít katastrofickou chybu, jejíž nalezení a odstranění je velmi jednoduché, a na druhé straně mít jen mírně obtěžující chybu, která ale vyžaduje přepsání větší části kódu k tomu, abychom ji odstranili.
- **Závislost na oblasti použití systému**. Stejný druh chyby v jedné oblasti nemusí způsobit výraznější komplikaci, zatímco v jiné oblasti může mít závažné důsledky. Například stejná numerická chyba se ve video hře téměř neprojeví, zatímco v nějakém řídícím systému může mít závažné následky.
- **Závislost na mentalitě uživatele**. Různí uživatele mohou mít různý náhled na stejnou chybu. Běžnému obchodníkovi chyba v zaokrouhlení částky nemusí výrazněji vadit, zatímco bankéř ji může považovat za nepřijatelnou.
- **Závislost na fázi vývoje systému**. Vesměs čím později je chyba zjištěna, tím je její výskyt závažnější, neboť její odstranění je dražší. Například jednoduchou chybu v době vývoje systému programátor snadno opraví, zatímco objeví-li se až v době provozu systému, pro pracovníka, který se jen stará o údržbu systému a nepodílel se na jeho vývoji, může být obtížné ji odstranit.

Kategorizace chyb

Rozdělení chyb do různých kategorií je poměrně obtížné. Stejná chyba může být zařazena do různých kategorií v závislosti na tom, jaká je historie její výskytu a jak ji vnímá programátor. Například chyba, kdy v identifikátoru je chybně zapsán jeden znak, přičemž změna znaku je tak nešťastná, že ji neodhalí syntaktická kontrola v době překladu programu, může mít za následek změnu nějaké hodnoty ve zcela jiné místě, než ve kterém se chybně zapsaný identifikátor nachází. Chybu tak můžeme považovat za překlep při psaní programu, za chybu zdrojového kódu, za chybu ve zpracování dat nebo za funkční chybu.

Chyby v požadavcích a specifikacích

Chyby v požadavcích patří mezi nejnákladnější chyby. Vznikají většinou při začátku vývoje systému a odhaleny jsou mnohdy až po dokončení systému, kdy je systém už nasazen do používání. Problémy ve specifikaci vedou k problémům v určité funkci systému. Může to být:

- Funkce pracuje v systému chybně. Tento druh chyby může být způsoben špatným pochopením specifikace, nejednoznačností specifikace apod.
- Funkce v systému chybí. Tento typ chyby se snadno zjistí a náprava je přímočará.
- Funkce je v systému nadbytečná. Tento případ lze považovat za neškodný, pokud se tím citelně nezvyšuje složitost systému.

Strukturální chyby

Do této kategorie řadíme chyby:

- **Chyby větvení a sekvenční**. Sem patří nedosažitelný kód, chybné vnoření cyklů, chyby v ukončení cyklů, špatně navržené přepínače, časté používání příkazu skoku (go to), chybějící příkazy, nadbytečné příkazy.
- **Chyby v použití logických operací**. Sem patří chyby plynoucí ze špatně navržených logických výrazů (například jako negace operátoru < je použit operátor >), mnohdy způsobené špatným pochopením, jak se logické operace vyhodnocují.
- **Výpočetní chyby**. Zahrnují chyby v aritmetických výrazech, ve výpočtu matematických funkcí, chybném sestavení algoritmů. Dále se patří chyby způsobené nevhodnou konverzí jednoho datového typu v jiný, přetečením při výpočtu, chyby související s operacemi srovnání mezi hodnotami v různých datových typech (ASCII, celočíselné datové typy, čísla v pohyblivé řádové čárce).
- **Chyby v inicializaci**. Jsou poměrně často vyskytující se chyby. Patří sem použití proměnných, u kterých se zapomnělo na inicializaci, i když je inicializace zapotřebí, nebo proměnné byly inicializovány chybnou hodnotou nebo hodnotou v chybném datovém typu nebo naopak proměnné byly inicializovány, aniž to bylo zapotřebí. Ten poslední případ, kdy inicializace je zbytečná, sice nezpůsobuje chybu ve funkcích systému, nicméně může snižovat jeho výkon, dochází-li k takovým inicializacím příliš často.
- **Neobvyklé toky dat**. Sem mimo jiné patří již uvedené chyby v inicializaci. Obecně jde o jakékoliv nelogické použití dat. Typicky jsou to případy, kdy vypočítaná nebo modifikovaná hodnota není vůbec použita a ani není nikam uložena.

Datové chyby

Jde o chyby související s datovými strukturami, formáty dat, počátečními hodnotami objektů.

Chyby kódování

Jsou všechny ostatní chyby, které nelze zařadit do již uvedených kategorií. Jde například o syntaktické chyby v kódu. Ty nicméně v současnosti odhalí překladače. Dále jde o nedeklarované proměnné, chybějící funkce v kódu. Tyto problémy jsou zjištěny v průběhu překladu nebo sestavení programu. Spíše rozsáhlejší výskyt těchto chyb ukazuje, že kód byl psán nedbale a dá se očekávat i větší výskyt jiných druhů chyb.

Statické testování - prohlédnutí (inspekce) kódu

Statické testování znamená, že projdeme kód, aniž bychom program spouštěli. Jde o *strukturální analýzu* programu. Cílem je najít chyby v programu co nejdříve a také najít chyby, které může být obtížné najít dynamickým testováním.

Chyby referencí na data

Tyto chyby jsou způsobeny chybným použitím proměnné (referencí na proměnnou).

- Je v kódu reference na proměnnou, které nebyla přiřazena počáteční hodnota?
- Je hodnota proměnné, která je použita jako index prvku pole v rozmezí hodnot, které odpovídají rozsahu pole? Obdobně je proměnná, která indexuje znaky řetězce, v rozsahu délky řetězce?
- Nezapomnělo se někde, že pole je indexováno od nuly a tudíž maximální hodnota indexu je o 1 menší, než je rozsah pole?
- Není někde použita proměnná, kde by bylo vhodnější použít konstantu? Například k ověření, zda hodnota indexu nepřekračuje rozsah pole.
- Není někde proměnné přiřazena hodnota, která má jiný datový typ, než je typ této proměnné? Například celočíselné proměnné je přiřazena hodnota v pohyblivé řádové čárce.
- Je dříve, než je ukazatel použit, alokována příslušná paměť a její adresa přiřazena ukazateli?
- Jestliže datová struktura je použita ve více funkcích, je tato datová struktura ve všech funkcích identická?

Chyby v deklaracích

- Mají všechny deklarace správný datový typ, rozsah (například u polí) a správnou specifikaci (*static*, *volatile*)? Neměli bychom například místo pole znaků použít datový typ řetězec?
- Je-li proměnná v deklaraci inicializována, je inicializační hodnota konzistentní s datovým typem této proměnné?
- Nemáme někde proměnné s velmi podobnými jmény? Sice to není chyba, ale vzniká zde nebezpečí, že při psaní kódu některou proměnnou zaměníme za proměnnou s podobným jménem.
- Nemáme někde deklarované proměnné, které jsme vůbec nepoužili?

Chyby výpočtu

- Nemáme výrazy, ve kterých jsou operace nad operandy s různými datovými typy? Například sčítáme celé číslo a číslo v pohyblivé řádové čárce?
- Nemáme operace nad proměnnými, které mají stejný typ dat, ale různou délku dat? Například sčítáme jednobytový datový typ (*char*, *byte*) s vícebytovým datovým typem (*int*).
- Pokud už kombinujeme proměnné s různými datovými typy a délkou, bereme v úvahu, jakým způsobem jsou v daném jazyce prováděny implicitní datové konverze?
- Nemá proměnná na levé straně operace přiřazení datový typ, který má menší rozsah, než je rozsah datového typu výrazu na pravé straně přiřazení?
- Nemůže někde při výpočtu výrazu nastat přetečení nebo podtečení?
- Nemůže při výpočtu podílu nebo zbytku dělení (operace modulo) nastat, že dělitel bude mít nulovou hodnotu?
- Je-li podíl celých čísel uložen do celočíselné proměnné, bereme v úvahu, že zde nastává ztráta desetinné části podílu?
- Nemůže někde nastat, že hodnota proměnné se dostane mimo rozsah odpovídající významu proměnné. Například máme proměnnou, ve které je výsledek výpočtu pravděpodobnosti, a její hodnota se dostane do záporných čísel nebo čísel větších než 1.
- Máme-li složitější výraz obsahující více různých operátorů, máme ho správně sestavený vzhledem k vzájemné precedenci operátorů. Neměli bychom precedenci zvýraznit použitím závorek?

Chyby srovnání

- Nepoužili jsme někde místo operace < (>) operaci <= (>=) nebo naopak? Tato záměna se objevuje poměrně často.
- Máme srovnání mezi vypočítanými hodnotami v pohyblivé řádové čárce? Bereme zde v úvahu chyby přesnosti, které mohou při výpočtu vzniknout? Zejména, pokud tyto hodnoty srovnáváme operátorem rovnosti (==) nebo zda jsou různé (!=).
- Máme logické (booleovské) výrazy dobře sestavené vzhledem k významu logických spojek a pořadí jejich vyhodnocování?
- Máme-li v logickém výrazu použity celočíselné proměnné, bereme v úvahu logickou interpretaci jejich celočíselné hodnoty (0..*false*, ostatní..*true*)?

Chyby větvení

- Obsahuje-li programovací jazyk blokové závorky ({ }), jsou zejména uzavírací závorky na správných místech?
- Nemůže u cyklu nastat, že neskončí? Nemůže u funkce nastat, že neskončí?
- Může u cyklu nastat, že je předčasně ukončen?
- Může u cyklu nastat, že ani jednou neproběhne. Pokud ano, je to v pořádku?

- Jestliže program obsahuje přepínač (*switch . . . case*), nemůže nastat, že žádný případ se nevybere. Pokud, ano je to v programu náležitě ošetřeno (*default*)?
- Není v podmínce cyklu chyba, která způsobí o 1 průchod více nebo méně?

Chyby v parametrech funkcí

- Odpovídají datové typy a rozsahy skutečných parametrů ve volání funkce datovým typům a rozsahům formálních parametrů v definici funkce. Jsou parametry napsány ve správném pořadí?
- Jsou-li při volání funkce jako argumenty předány konstanty, nemůže ve funkci nastat, že budou změněny?
- Nemění funkce parametr, který je zamýšlen jen jako vstupní?

Chyby vstupů a výstupů

- Je program přizpůsoben specifickému formátu dat, který používá vstupní nebo výstupní zařízení?
- Je v programu ošetřen případ, kdy soubor neexistuje nebo ho nelze otevřít?
- Je v programu, který pracuje s externím zařízením, ošetřena situace, kdy zařízení je odpojeno nebo vypnuto během operace zápisu na zařízení nebo čtení ze zařízení?
- Jsou v programu náležitě ošetřeny možné chyby, které mohou nastat během vstupních a výstupních operací?
- Jsou zprávy vypisované při výskytu chyb dostatečně srozumitelné a vyjadřující podstatu chyby?

Ostatní kontroly

- Je-li požadováno, aby program byl přenositelný na jiné platformy, odpovídá tomu i způsob, jak byl program sestaven (napsán)?
- Je zajištěno, že program bude pracovat na plánovaných počítačích s danou velikostí operační paměti a dalším technickým vybavením – grafickým adaptérem, tiskárnami atd.
- Vypisuje překladač při překladu varující zprávy? Jestliže ano, ověřili jsme, že tyto neukazují na nějaký problém v programu. Celkové je účelné udělat úpravy ve zdrojovém kódu, aby se žádné varující zprávy při překladu už neobjevovaly.

Funkční testování

Dynamické testování stylem „černé skříňky“

Program testujeme, jako bychom byli uživatelé (*dynamické testování*). Přitom vycházíme z toho, že neznáme vnitřní strukturu programu (*černá skříňka*). Na vstup vkládáme hodnoty a u výstupních hodnot ověřujeme, zda jsou správné. Tento způsob testování se také označuje jako testování chování programu.

U tohoto způsobu testování vycházíme ze specifikace požadavků. Z nich sestavíme testovací případy, což jsou dvojice vstupní údaje + požadovaný výsledek. Efektivita tohoto způsobu testování je závislá na tom, jak účelně dokážeme testovací případy naplánovat.

Test na základní použitelnost a test na selhání

Při testování na základní použitelnost se snažíme ověřit, že hlavní části programu fungují. Při tomto testování použijeme jen jednoduché a přímočaré testovací případy. Netestujeme zde, jaké jsou celkové schopnosti programu.

Test na základní použitelnost lze považovat jako úvodní fázi testování, kdy se odhalí základní chyby. Po jejich odstranění můžeme přejít na druhou fázi testování – test na selhání. Při něm se už snažíme odhalit všechny slabé stránky programu.

Výběr testovacích případů

Rozdělení testovacích případů na ekvivalentní případy

Metodicky se snažíme zredukovat značné množství možných testovacích případů na přijatelný počet jejich rozdělením do tříd testovacích případů, kdy ve stejné třídě jsou vždy testovací případy, které mají stejný charakter a je předpoklad, že pokud jeden z nich v testovaném programu projde, projdou i ostatní. Třídu můžeme také chápat jako množinu testovacích případů, které odhalují stejný druh chyby.

Například testujeme funkci výpočtu úroku pro účet. Jestliže pro náhodně zvolenou běžnou hodnotu stavu na účtu je úrok vypočítán dobře, můžeme očekávat, že i pro jiné hodnoty stavu na účtu bude výpočet v pořádku.

Při dělení testovacích případů na ekvivalentní třídy je účelné uvažovat případy.

- Vstupní hodnoty, které nejsou hraniční
- Hraniční vstupní hodnoty
- Nulové hodnoty
- Chybné vstupní hodnoty

U vstupních hodnot, které nejsou hraniční, testujeme funkčnost programu pro běžný případ vstupních údajů.

Někdy v programu bývá chyba, která se projeví jen u hraničních hodnot vstupních údajů. Například máme funkci pro uložení 0..100 údajů na zásobník. V důsledku chyby funkce pracuje správně až do 99 údajů uložených na zásobník. Až při uložení 100-ho údaje nastane chyba.

```
int zasobnik[100], vrchol=0;

void push(int c)
{
  zasobnik[++vrchol] = c;
}
```

Nulové hodnoty (číselné) nebo prázdné (žádná hodnota neuvedena nebo prázdný řetězec) vyžadují mnohdy specifické otestování. Například testujeme, co se stane, když uživatel zapomene ve vstupním formuláři vložit povinný údaj. Jiný případ může být informační systém banky, který každý měsíc posílá výpis pohybu na účtu. Otestujeme, zda tento výpis neposílá, i když v daný měsíc nebyl žádný vklad nebo výběr.

Kvalitní program počítá s tím, že uživatel může udělat při vkládání údajů chybu. Například máme-li jako vstupní údaj datum, testujeme, co se stane, když například číslo měsíce není v rozsahu 1..12 nebo datum je chybné, například 29.2.2011 .

Stavové testování

Prozatím jsme se omezili na testování, jak se program zachová při vložení různých vstupních hodnot. V průběhu zpracování nebo výpočtu program prochází různými stavy. Ověření, jakými stavy program postupně prochází, zda logicky navazují na sebe, je další možnost testování.

Stav programu je určitá fáze ve zpracování nebo výpočtu, do které se program dostane za určitých podmínek, nebo je to nějaké nastavení režimu, ve kterém program pracuje.

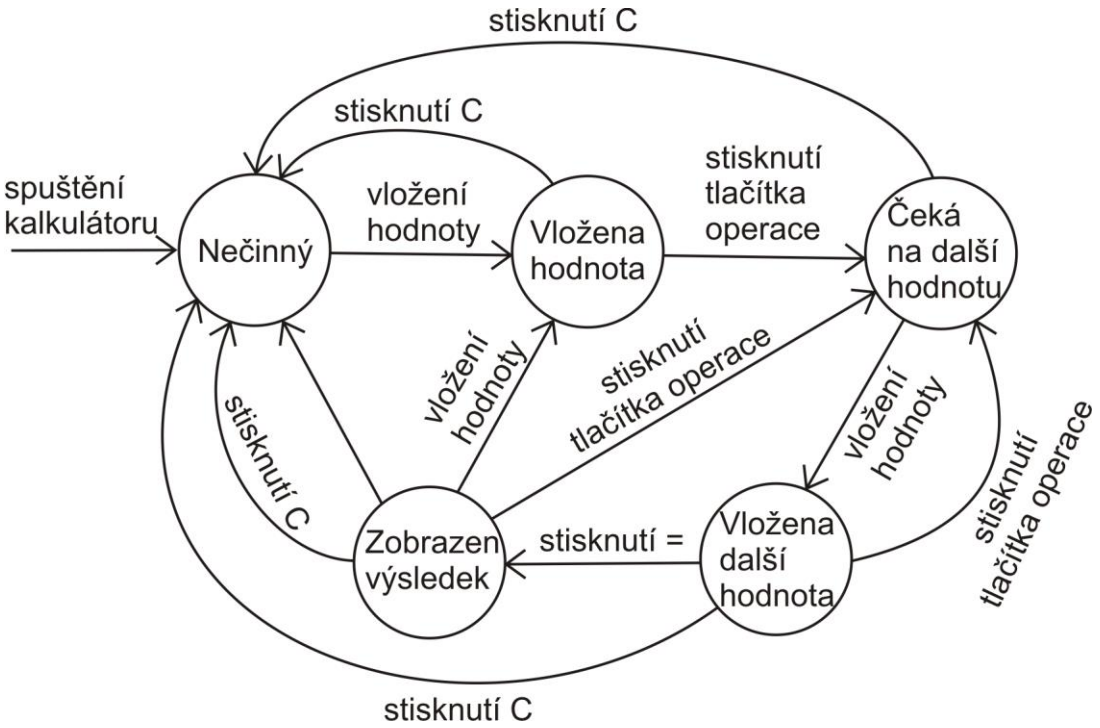
Například budeme-li sestavovat program pro jednoduchý kalkulátor, jednotlivé stavy mohou být vyvolány událostmi:

- není vložena žádná hodnota
- byla vložena hodnota
- bylo stisknuto tlačítko operace +, -, * nebo /
- byla vložena druhá hodnota
- bylo stisknuto tlačítko = (a je zobrazen výsledek)

a nastavení režimů může být

- vstup a zobrazení čísel v dekadické soustavě
- vstup a zobrazení čísel v binární soustavě
- vstup a zobrazení čísel v hexadecimální soustavě

Ze stavů a událostí, které způsobují vzájemný přechod mezi stavy, si můžeme sestavit diagram, který je východiskem pro testování.



Každý stav v diagramu reprezentuje určitý stav programu, ve kterém program může být. Pokud si při vytváření diagramu nejsme jisti, zda určitý stav je nový stav nebo je shodný s některým již vytvořeným stavem, raději ho stanovíme jako nový stav a případně, když se následně ukáže, že je identický s jiným stavem, můžeme oba dva stavy sloučit.

Přechod z jednoho stavu do druhého nemůže nastat samovolně, musí být vyvolán určitou událostí nebo podnětem – vstup z klávesnice, stisknutí tlačítka, vybrání nějaké volby, signál z vnějšího čidla atd.

Stanovíme podmínky, jaké mají být nebo mohou nastat při vstupu do stavu a výstupu ze stavu, a výstup který má být v daném stavu. Například je proveden určitý výpočet, zobrazena nějaká nabídka, je výstup na tiskárnu. Je to cokoliv, co proběhne při přechodu z jednoho stavu do druhého.

U složitějších diagramů není možné projít všechny cesty v grafu. Snažíme se jejich počet zredukovat při dodržení zásad:

- Do každého stavu bychom se během testování měli aspoň jednou dostat, libovolným způsobem.
- Otestujeme ty nejméně obvyklé přechody mezi stavy. Zde je největší pravděpodobnost, že se u nich v programu na něco zapomnělo nebo se něco přehlédlo.
- Nezapomeneme otestovat všechny chybové stavy v programu, jaké je ošetření chyby v nich, zda vypisovaná zpráv o chybě je korektní a zda je zajištěn žádoucím způsobem návrat z chybového stavu k normálnímu zpracování.
- Při výběru dalšího přechodu od jednoho stavu k druhému se snažíme vybírat přechod náhodným způsobem.

Při testování také vyzkoušíme, jak program reaguje při výskytu více událostí v době přechodu z jednoho stavu do druhého nebo když události probíhají současně. Například:

- Neočekávané stisknutí klávesy nebo tlačítka myši při přechodu z jednoho stavu do druhého.
- Spuštění dvou instancí stejného programu ve stejnou dobu.
- Otevření a případně ukládání stejného dokumentu dvěma programy ve stejnou dobu.
- Používání databáze více programy ve stejnou dobu.

I když náš testovaný program není pro takové situace určen, může k nim omylem dojít a je zapotřebí, aby byl dostatečně robustní a uspokojivě je zvládl.

Opakování operací, ztížené podmínky, zatížení programu

Další tři testy stavů jsou, zda program vydrží značné opakování operací, zda bude pracovat ve ztížených podmínkách a jak se zachová při značném zatížení. Tyto testy směřují ke zjištění, jaké problémy mohou nastat, když program bude pracovat v obtížných podmínkách, a zda bylo na tyto situace pamatováno při sestavování programu.

Test na opakování může být:

- opakované spuštění a ukončení programu
- opakované uložení a opětné načtení údajů
- opakované použití stejné funkce programu

Některé chyby se projeví až po mnoha opakování. Typickým příkladem je alokování paměti, kdy přitom paměť neuvolníme, když už není potřeba. Opakováním operací

u takové chyby množství alokované paměti nestále narůstá, zpomaluje se činnost programu nebo může v určité fázi dojít k selhání alokace paměti. Tento druh chyby se může třeba projevit teprve až po tisících opakování operace. Takový test už je neschůdné dělat ručně, je potřeba ho zautomatizovat.

Test na činnost programu ve ztížených podmínkách může zahrnovat test, jak bude program pracovat v těchto minimálních podmínkách:

- minimální rozsah operační paměti
- pomalý procesor
- malý rozsah volného místa na disku

Test na zatížení naopak testuje, jak bude program pracovat:

- Při největším možném objemu dat.
- Je-li program síťová aplikace, jak bude pracovat při velkém počtu souběžných připojení (například internetových obchod).
- Pokud program je pro obsluhu více periferních zařízení nebo pro vstup z více periferních zařízení (čidel), jak bude pracovat při připojení maximálního možného počtu těchto zařízení.

Hledejte chyby, které už jste našli

Nalezené chyby mohou být projevem určité špatného návyku nebo chybného uvažování programátora a mohou se objevit v programu vícekrát. Například nalezli jsme chybu v použití horní meze u pole. Je možné, že programátor tento druh chyby udělal v programu na více místech a tudíž stejná chyba se v programu stále vyskytuje.

Dynamické testování spojené s analýzou kódu

Opět testujeme běžící program. Na rozdíl od předchozího popsaného způsobu testování analyzujeme přitom kód programu a sledujeme, jak běží. Toto sledování nám umožňuje rozhodnout se, co dále je zapotřebí testovat nebo co už není zapotřebí testovat a jakým způsobem v testování pokračovat. Jde o strukturální testování, protože na základě struktury kódu navrhujeme a provádíme testy.

Tento způsob testování nám umožňuje:

- Přímo testovat funkce, podprogramy, knihovny.
- Testovat program jako celek, přičemž lze navrhnout testovací případy na základě znalosti, jak probíhá výpočet v programu.
- Zjišťovat hodnoty proměnných a stavové informace v programu k ověření, zda program dělá, co by měl.
- Stanovit, které části programu už byly v průběhu testování již prověřeny, a přizpůsobit tomu další testování tak, abychom znovu netestovali již otestované části nebo naopak některé části programu při testování nevynechali.

Testování versus ladění

Dynamické testování s použitím znalosti kódu a ladění kódu mají společné rysy. Obojí se zabývá chybami v kódu a obojí přitom analyzuje kód, nicméně jejich cíle jsou rozdílné. Cílem testování tohoto způsobu testování je odhalit chyby. Cílem ladění je odstranit chyby. Jejich společným rysem je, že hledají, na kterém místě v kódu je chyba. Ale

zatímco při testování dostáváme případně informaci, které místo v programu vypadá podezřele, při ladění musíme přesně stanovit příčinu chyby a pokusit se ji opravit.

Testování jednotek a testování po integraci

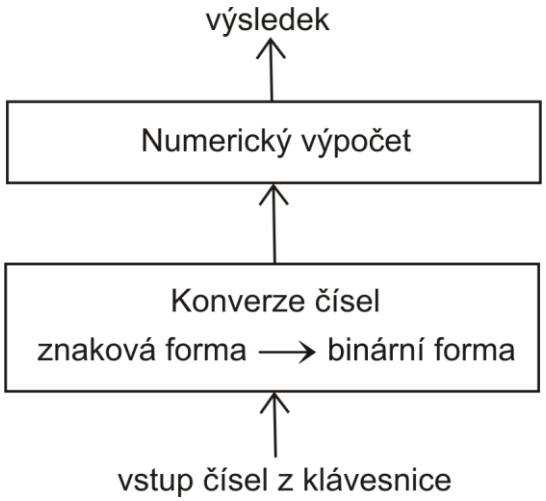
Testování začínáme od nejnižší úrovně testováním těch nejmenších částí programu – *testování jednotek nebo modulů* programu. Jakmile jsou jednotky nebo moduly otestovány a chyby v nich nalezené opraveny, postupně integrujeme již otestované části ve větší části a ty testujeme. Takto inkrementálním způsobem postupně spojujeme otestované části ve stále větší celky, až se dostaneme do úrovně *testování systému*, kdy už testujeme celý projekt nebo aspoň jeho podstatnou část. Tento inkrementální způsob nám usnadňuje lokalizaci chyby. Byla-li chyba nalezena při testování jednotky, jde zřejmě o chybu této jednotky. Jestliže chyba se objevila až po určité integraci více jednotek, patrně to souvisí se vzájemnou interakcí mezi jednotkami.

Jsou dva základní přístupy inkrementálního testování: *zdola-nahoru* a *shora-dolů*.

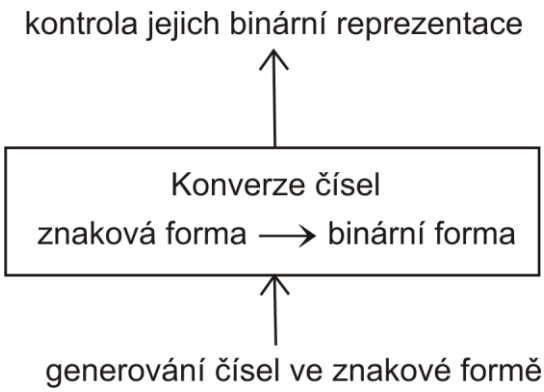
Při testování zdola-nahoru si pro své jednotky a moduly sestavujeme testovací systém, který simuluje jejich budoucí použití. Posílá jim testovací data, čte výsledky a kontroluje, zda jsou v pořádku. Vhodným výběrem testovacích dat lze tímto způsobem jednotky a moduly velmi důkladně otestovat. Lze zde běžně snadno otestovat chování jednotky nebo modulu i při takových případech vstupních dat, které na vyšší úrovni testování je obtížné navodit.

Při testování shora-dolů testujeme od vyšších úrovní integrace programu. Přitom jednotky nebo moduly na nižších úrovních nahradíme jednoduchým kódem, který přes rozhraní, kterým je jednotka nižší úrovně propojena s vyšší úrovní, posílá testovací data směrem k vyšší úrovni.

Příklad.



Zdola-nahoru:



Shora-dolů: