Verifikácia a validácia

alebo

Ako vytvoriť produkt **bez chýb** ?

Definícia

Verifikácia a validácia (V&V) je pomenovanie pre kontrolné procesy, ktorých cieľom je zabezpečiť, aby softvérový produkt zodpovedal potrebám zadávateľa.

Definícia (2)

- **verifikácia** – overenie správnosti produktu vzhľadom k formulovaným požiadavkám
  - **Vývojár: „Aplikácia sa nespráva podľa schváleného zadania.“**
- **validácia** – overenie správnosti produktu vzhľadom k reálnym požiadavkám
  - **Používateľ: „Aplikácia robí niečo iné, než chcem.“**

(validácia je v konečnom dôsledku to, čo je podstatné)

Zdroje chýb

- programátorské chyby
- chyby v špecifikácii požiadaviek
- technologické nedostatky
- kombinácia / iné

V & V

- je normálne, že chyby vznikajú
- **podstatné je, aby boli** počas vývoja (až na výnimky) **odstránené** – s čo najmenšími nákladmi

Koľko „stojí“ chyba ?

- potenciálne veľmi veľa ...
  - Therac-25
  - Ariane 5
  - Mars Climate Orbiter
  - Code Red Worm
  - Airbus 320
  - ...

Koľko stojí oprava chyby ?

(čím neskôr, tým viac)

Kedy V & V ?

V & V sa týka celého procesu vývoja softvéru – všetkých činností:

- špecifikácie požiadaviek
- plánovania
- návrhu
- implementácie
- integrácie
- údržby
- ...

čím skôr sa chyba objaví, tým lepšie

Prehľad techník

- **detekcia chýb počas písania programu**
  - overovacie podmienky, Design by Contract
  - logovanie, trasovanie
- **inšpekcie (revízie)**
  - „ručne“ sa kontroluje niektorý z výstupov (špecifikácia požiadaviek, návrh, zdrojový kód, testovacie údaje, konfigurácia, dokumentácia, ...)
  - ako pomôcku je možné použiť aj nástroje pre analýzu
  - ide o statickú techniku (bez spustenia vytvoreného kódu)
- **testovanie (funkčné a iné)**
  - funkčné testovanie: softvérový produkt (resp. jeho časť) sa spustí na vybraných testovacích údajoch a jeho výstup sa porovná s očakávaným
  - ide o dynamickú techniku (vytvorený kód sa spustí)
  - ďalšie formy testovania: používateľské, výkonnostné, bezpečnostné, ...
  - rôzne úrovne: testovanie komponentov, integračné, systémové, akceptačné

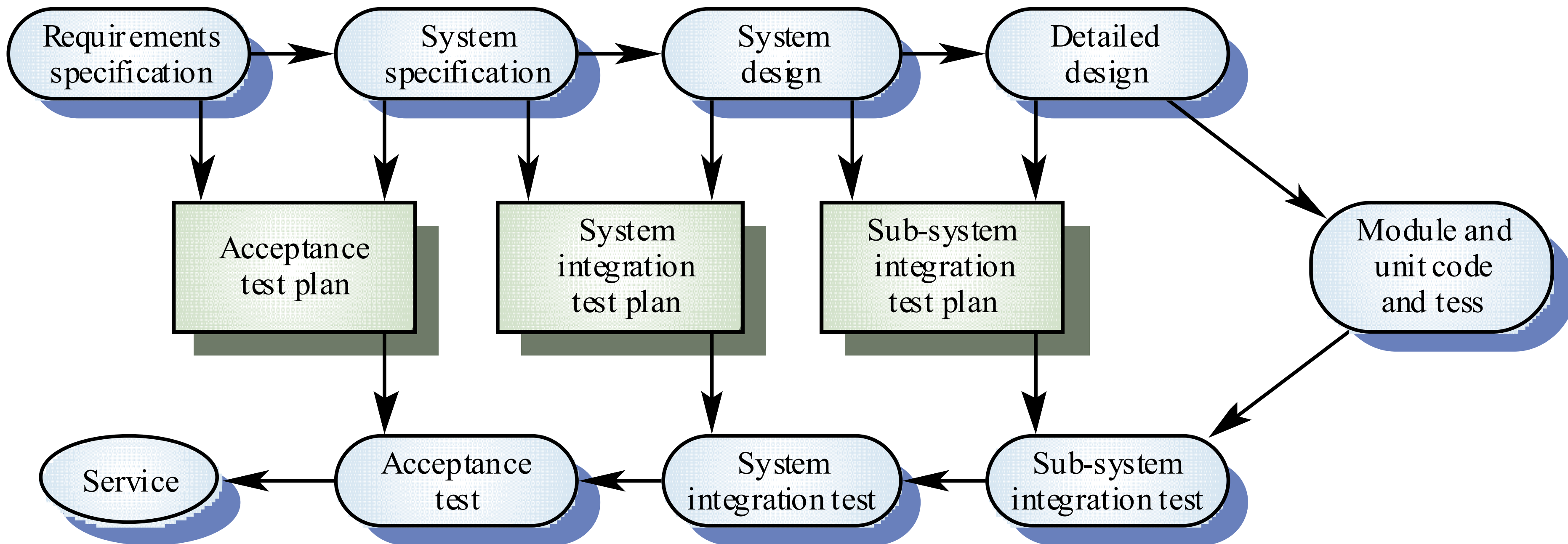
testovanie ≠ debugovanie

- testovanie: cieľom je odhaliť existenciu chyby
- debugovanie: cieľom je chybu nájsť a opraviť

Plánovanie V&V

- V&V predstavuje významnú časť vývoja (pri istých systémoch aj 50% nákladov)
- musí existovať plán pre V&V obsahujúci:
  - všeobecný popis procesu pre V&V
  - overované produkty (špecifikácia požiadaviek, návrh, kód, ...)
  - procedúry pri overovaní (napr. zaznamenávanie výsledkov)
  - plán (čas a zdroje)
  - hardvérové a softvérové požiadavky
- cieľom **<u>nie je</u>** 100% spoľahlivosť
  - plán závisí od **požadovanej spoľahlivosti**, ktorá zase závisí od:
    - účelu, pre ktorý bude softvér slúžiť
    - očakávaní používateľov
    - prostredia trhu
- jednotlivé úrovne plánu vzniknú v príslušnej etape vývoja

V-model



1.
Detekcia chýb počas písania programu

Design by Contract
(Bertrand Meyer)

Typ je viac než množina signatúr: podstatné sú vlastnosti (sémantika) jednotlivých operácií.

Abstraktný dátový typ ako prostriedok pre popis rozhrania

- Def: ADT je **množina** (matematických) **objektov** definovaná
  - zoznamom funkcií pracujúcich s týmito objektmi (funkcie môžu byť aj nulárne),
  - vlastnosťami týchto funkcií.

- Špecifikácia ADT obsahuje:
  - signatúry funkcií
  - určenia definičných oborov funkcií
  - axiómy

Príklad ADT: STACK

<u>FUNCTIONS</u>

- **put: STACK x INT → STACK**
- **remove: STACK ⇸ STACK**
- **item: STACK ⇸ INT**
- **empty: STACK → BOOLEAN**
- **new: STACK**

PRECONDITIONS

- **remove (s: STACK) requires not empty (s)**
- **item (s: STACK) requires not empty (s)**

AXIOMS

x: INT, s: STACK

- **item (put (s, x)) = x**
- **remove (put (s, x)) = s**
- **empty (new) = true**
- **not empty (put (s, x))**

Pre FIFO by prvé dve axiómy boli:

- ***item (put (q, x)) = if empty (q) then x else item (q)***
- ***remove (put (q, x)) = if empty (q) then new else put (remove (q), x)***

Súvis so softvérom ...

Triedu/modul môžeme vnímať ako **implementáciu** ADT.

(potenciálne viacerých ADT)

- ADT je matematický pojem, modul softvérový – z tohto hľadiska je vzťah *ADT* : *modul* podobný vzťahu *matematická funkcia : funkcia v programovacom jazyku* ($\sqrt{x}$: sqrt (x))

Design by Contract: princíp č.1

Súčasťou špecifikácie rozhrania modulu sú **vstupné a výstupné podmienky** pre jednotlivé operácie.

Zapisujú sa v programovacom alebo v prirodzenom jazyku.

(Ak používame ADT, tieto vlastnosti sú odvodené z AXIOMS a PRECONDITIONS príslušného ADT.)

Príklad (zápis v programovacom jazyku)

```
interface Stack
{
   Stack ();                    // POST: empty ()
   int item ();                 // PRE: !empty ()
   boolean empty ();
   void put (int x);            // POST: !empty () && item () == x
   void remove ();              // PRE: !empty ()
};
```

Výrazový jazyk podmienok zapísateľných v programovacom jazyku je **slabší** než algebraický jazyk ADT. V tomto príklade chýba podmienka „remove (put (s, x)) = s“

```
double sqrt (double x);  PRE: x >= 0
                         POST: abs (result*result-x) <= EPSILON
```

Design by Contract: Princíp č.2

Vstupné a výstupné podmienky predstavujú **zmluvu** medzi modulom a jeho klientami.

- Klient:
  - musí zabezpečiť platnosť PRE
  - môže očakávať platnosť POST
- Modul:
  - **môže očakávať platnosť PRE –** ak PRE nie je splnená, môže spraviť „čokoľvek“
  - musí zabezpečiť platnosť POST

Design by Contract: Princíp č.3

Modul sa nemá snažiť „riešiť“ prípady, v ktorých nie je splnená vstupná podmienka.

V prípade nesplnenej vstupnej podmienky by sa mala signalizovať výnimka.

Prečo neriešiť prípady, keď nie je splnená vstupná podmienka:

- volaný modul nemá možnosť tieto prípady **zmysluplne ošetriť** (príklad: výber z prázdneho zásobníka) – najrozumnejšie, čo môže spraviť, je signalizovať výnimku
- snaha „zachraňovať“ prípady mimo špecifikácie vedie k zväčšeniu množstva kódu a teda k **vyššej zložitosti** systému ako celku

Nesplnenie podmienky ≈ bug

- vstupnými podmienkami <u>nemá</u> byť ošetrený vstup prichádzajúci „zvonku“ (od používateľa, od iných aplikácií)
  - tieto vstupy, naopak, musia byť dôkladne ošetrené!
- nesplnenie vstupnej podmienky vždy signalizuje chybu v klientovi
- nesplnenie výstupnej podmienky vždy signalizuje chybu vo volanom module

Načo Design by Contract ?

- nástroj na dosiahnutie správnosti
  - jasne sa špecifikujú požiadavky na implementovanú operáciu (aj na klienta)
- komunikácia medzi vývojármi – popis rozhrania
- podpora pre testovanie a ladenie
  - zapnutá kontrola podmienok chybu rýchlejšie odhalí a lokalizuje
- podpora pre *fault-tolerance* na úrovni softvéru
  - výnimky generované pri nesplnenej podmienke môžu byť zachytené a ošetrené

Striktné alebo voľné vstupné podmienky ?

- závisí od návrhára
- vo všeobecnosti je lepšie ich stanoviť striktné – umožniť volanej operácii „sústrediť sa“ na svoju úlohu
- volaná operácia často nemá šancu chybový stav (nesplnenie vstupnej podmienky) riešiť
  - vypísať oznam na obrazovku ?
  - ignorovať chybu ?
  - ukončiť činnosť programu ?
  - vedieť, čo robiť pri nesplnenej vstupnej podmienke je úlohou klienta

Invariant

Doplnenie: invariant platný pre celý objekt

- príklad: trieda Ucet

  stav = vklady.sucet () – vybery.sucet ()

- invariant musí byť zachovaný pri vstupe i výstupe z verejnej metódy objektu
  - dá sa pochopiť ako doplnenie PRE i POST

Pre každú verejnú metódu musí platiť:

$$\{ \text{PRE} \wedge \text{INV} \} \text{ metódu } \{ \text{POST} \wedge \text{INV} \}$$

Assert

- overovanie predpokladov môže byť zaradené aj na iných miestach v kóde
- bežný nástroj: assert

```
if (i % 3 == 0) {
   ...
} else if (i % 3 == 1) {
   ...
} else {
   assert i % 3 == 2;
   ...
}
```

Napr. java: vyhodnocovanie podmienok nastaviteľné pri behu (na úrovni jednotlivých balíkov)

Monitorovanie predpokladov počas behu programu

- počas vývoja a skúšobnej prevádzky určite zapnuté
- počas ostrej prevádzky - čiastočne vec názoru
  - je lepšie, keď program ohlási chybu, ako keď dá zlý výsledok, ktorému bude používateľ dôverovať
  - monitorovanie splnenia predpokladov viac či menej negatívne ovplyvní výkon
  - rozhodnutie závisí od
    - stupňa dôvery v bezchybnosť programu
    - závažnosti prípadného (nezdetekovaného) nesprávneho výstupu
    - požiadaviek na výkon a od negatívneho vplyvu monitorovania
  - je možné testovanie obmedziť napr. len na preconditions

Logovanie

- zaznamenávanie všetkých dôležitých, neobvyklých resp. chybových udalostí
- ukladanie do súboru, databázy, resp. inde
- spravidla konfigurovateľná úroveň podrobnosti
- zapnuté pri vývoji, skúšobnej aj ostrej prevádzke
  - potenciálne s rôznymi úrovňami podrobnosti

```
Logger logger = Logger.getLogger(...);

if (...)
{
   logger.log(Level.SEVERE, "An I/O error occured: " + e);
}
```

Trasovanie (Tracing)

- selektívne zapínateľné uchovávanie podrobnejších informácií o behu istej časti programu

Všeobecné zásady

- chyby detegovať
  - testovať návratové kódy
  - zachytávať výnimky – na vhodnej úrovni
- modul pred „odovzdaním ďalej“ otestovať
  - napr. tak, aby boli prejdené všetky príkazy v module

2.
Inšpekcie

Inšpekcie programov

- prvýkrát formalizované: 70-te roky, IBM (M.Fagan)
- primárny cieľ inšpekcií programov: odhaliť nedostatky v programe
  - nie navrhovať ich opravu alebo iné vylepšenia kódu
  - nie skúmať vhodnosť návrhu, postup prác a pod.
- sekundárne prínosy: oboznamovanie sa s kódom iných
- za nedostatok v programe sa považuje:
  - chyba
  - podozrivé miesto (nedosiahnuteľný kód, nepoužitá vstupná hodnota, ...)
  - nedodržanie firemných programátorských štandardov

Podmienky

Pred začatím inšpekcie je nutné zaistiť, že:

- existuje hotová, aktuálna, syntakticky správna verzia kódu, ktorý sa má podrobiť inšpekcii
- existuje presná špecifikácia, čo má tento kód robiť
- členovia inšpekčného tímu sú oboznámení s relevantnými organizačnými štandardami
- existuje zoznam najčastejších programátorských chýb
  - závisí od programovacieho jazyka
  - čím slabšia kontrola na úrovni kompilátora, tým viac treba kontrolovať pri inšpekcii

Fault class	Inspection check
Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the lower bound of arrays be 0, 1, or something else? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a delimiter explicitly assigned?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output?
Interface faults	Do all function and procedure calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception management faults	Have all possible error conditions been taken into account?

Zoznam chýb - príklad

Spôsob vykonania inšpekcie

- skupinový (brainstorming)
- dokumentový (každý študuje kód/dokument sám)
- možnosť kombinácie

Postup – príklad kombinovaného prístupu

- plánovanie
  - výber inšpekčného tímu, zaistenie ostatných potrebných náležitostí (kód, špecifikácia, miestnosť, ...)
- prehľad
  - prezentácia, kde autor popíše, čo má program robiť
- individuálna príprava
  - každý člen tímu individuálne študuje špecifikáciu a kód
- inšpekčné stretnutie (trvanie nie viac ako 2 hodiny)
  - identifikovanie chýb
- opravenie nájdených chýb
- pokračovanie (follow-up)
  - rozhodnutie predsedajúceho, či je potrebné opakovať inšpekciu alebo či je možné program považovať za schválený

Role v procese inšpekcie

- **autor:** programátor zodpovedný za vytvorený kód a za opravu nájdených chýb
- **predkladateľ (reader):** vysvetľuje kód na inšpekčnom stretnutí
- **inšpektor:** hľadá chyby v kóde
- **zapisovateľ:** zapisuje výsledky inšpekčného stretnutia
- **predsedajúci:** riadi a zodpovedá za priebeh inšpekcie
- **„vrchný“ predsedajúci:** zodpovedá za celý proces inšpekcií, aktualizáciu spoločných dokumentov (napr. checklisty), procedúr a štandardov

Efektívnosť inšpekcií

- v praxi zistené hodnoty (Fagan):
  - vo fáze prehľadu: 500 príkazov/hod
  - vo fáze individuálnej prípravy: 125 príkazov/hod
  - na stretnutí: 90-125 príkazov/hod
  - (závisí od prog. jazyka, zložitosti a kvality kódu)
- príklad:
  - 500 riadkov, 4 ľudia – cca. 36 človeko-hodín (54.000 Sk?)
- ukazuje sa, že inšpekcie sú efektívnejšie ako testovanie so spustením
  - používajú sa obe techniky

Inšpekcie – záver

- podobne môžu byť kontrolované všetky výstupy – plán, špecifikácia, návrh, dokumentácia, ...
- dôležité je úspešné uvedenie inšpekcií do praxe
  - zaškolenie inšpektorov a moderátorov
  - pozitívne prijatie členmi tímu
  - nespájať výsledky inšpekcií s ohodnotením pracovníkov
- pomôcka pre inšpekcie programov: nástroje pre statickú analýzu kódu
- **vybrať vhodnú formu inšpekcií**

3.
Testovanie

Typy testovania

- funkčné
- používateľské
- výkonnostné
- záťažové
- bezpečnostné
- ...

Funkčné testovanie

- testuje sa, či program robí to, čo robiť má (a robí to správne) + či nerobí to, čo robiť nemá
- tento typ testovania vykonávajú programátori/testeri
- aplikáciu rozdelíme na jednotky (moduly, funkčné celky, ...), ktoré testujeme zvlášť
- ku každej takejto jednotke má vzniknúť testovací scenár, podľa ktorého testy prebiehajú
- vhodné je čo najviac testov vykonávať automaticky

Používateľské testovanie

- zisťuje sa, či je používateľské rozhranie vhodné (štýlom, štruktúrou, ...)
  - realizované používateľmi resp. ľuďmi, ktorí sa im znalosťami a skúsenosťami podobajú
  - vedľa testujúceho sedí člen tímu, ktorý jeho činnosť sleduje a prípadné problémy zaznamenáva
- často sa vykonáva „vzdialene“, čo nie je optimálne
  - navrhované sú často protichodné riešenia
  - návrh riešenia vyžaduje komunikáciu s vývojárom
  - nemusia sa zachytiť problémy s neefektívnosťou rozhrania

Testovanie výkonnosti a záťaže

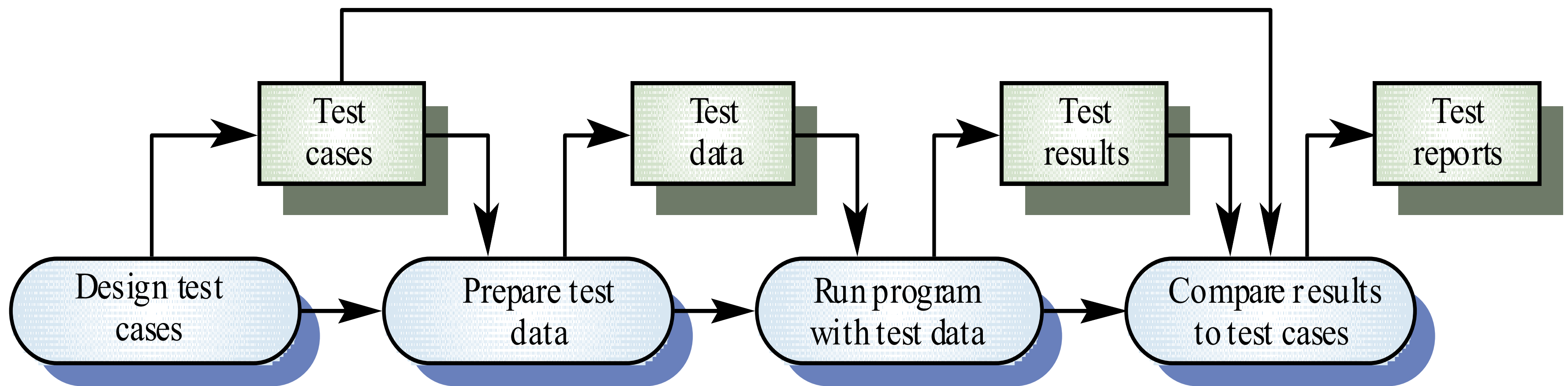
- cieľom je vytipovanie a odstránenie výkonnostne problematických miest (nie nutne 100% optimalizácia výkonu aplikácie)
- systém sa naplní údajmi a podrobí sa predpokladanej záťaži (danej napr. počtom naraz pracujúcich používateľov, počtom požiadaviek/správ za sekundu, ...)
- možné slabé miesta:
  - časová náročnosť (algoritmy, použité technológie)
  - pamäťová náročnosť
  - vzájomné blokovanie používateľov
  - ...

Testovanie na rôznych úrovniach

- testovanie komponentov
- integračné testovanie
- testovanie systému
- akceptačné testovanie

4.
Funkčné testovanie

Proces funkčného testovania



Ako vyberať testovacie údaje ?

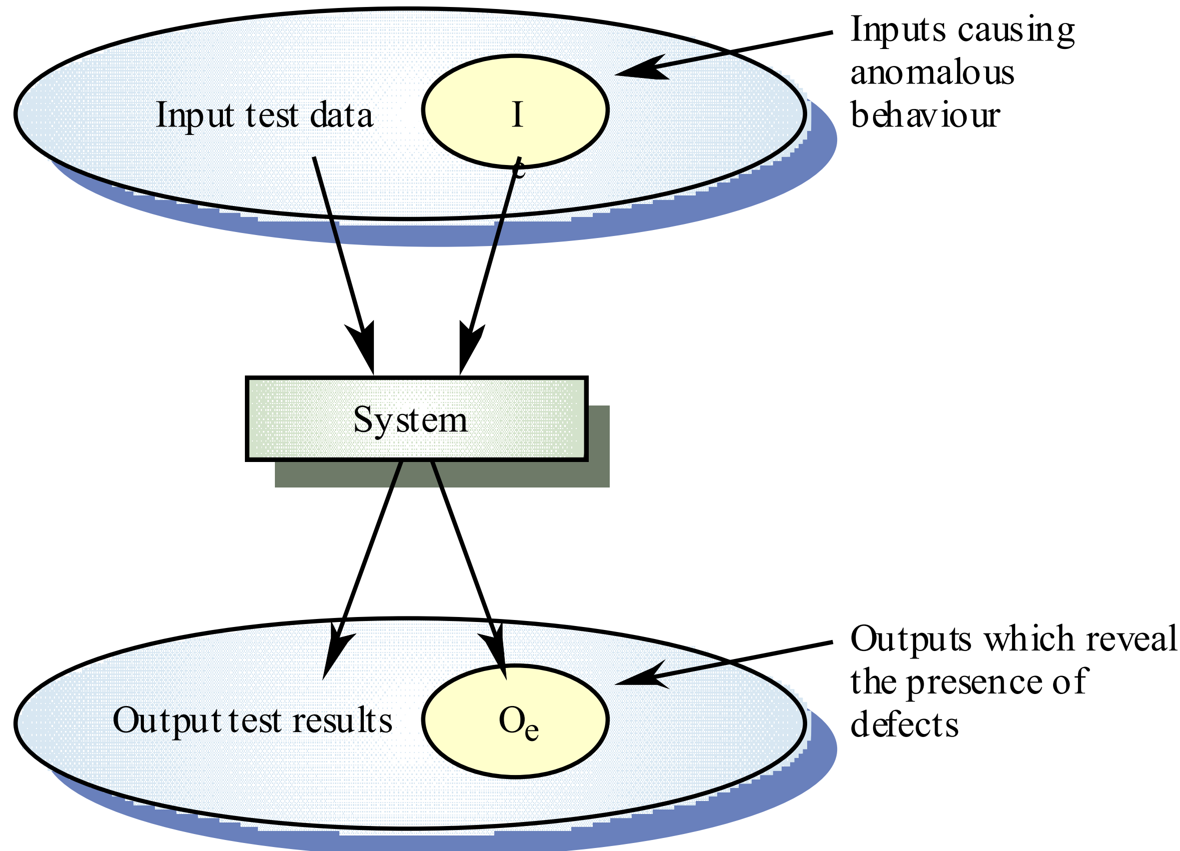
- preukázať bezchybnosť programu je možné len vyčerpávajúcim testovaním (exhaustive testing)
  - to je vo väčšine prípadov nemožné
- pre výber testovacích údajov treba mať definované kritériá (najhoršie je údaje vyberať náhodne)
  - napr: prejsť každým riadkom programu aspoň raz
  - napr: prejsť každou funkciou systému aspoň raz
  - napr: pre každú funkciu otestovať správne aj nesprávne používateľské vstupy
  - treba sa rozhodnúť, ako rozdeliť pozornosť medzi typické a hraničné situácie a medzi existujúce a pridané črty systému (pri vývoji novej verzie)

2 prístupy k výberu testovacích údajov

- black-box (podľa špecifikácie)
  - na kód nehľadím, snažím sa vytvoriť vstupy, ktoré budú mať vyššiu pravdepodobnosť odhalenia prípadnej chyby
- white-box (podľa kódu)
  - snažím sa vytvoriť vstupy, ktoré prejdú všetkými časťami kódu
- možné sú hybridné prístupy

Testovanie „black-box“

- podstatné je vybrať podmnožinu údajov, ktorá má najvyššiu pravdepodobnosť odhalenia chyby
- závisí od domény a konkrétnej špecifikácie



Výber hraničných hodnôt

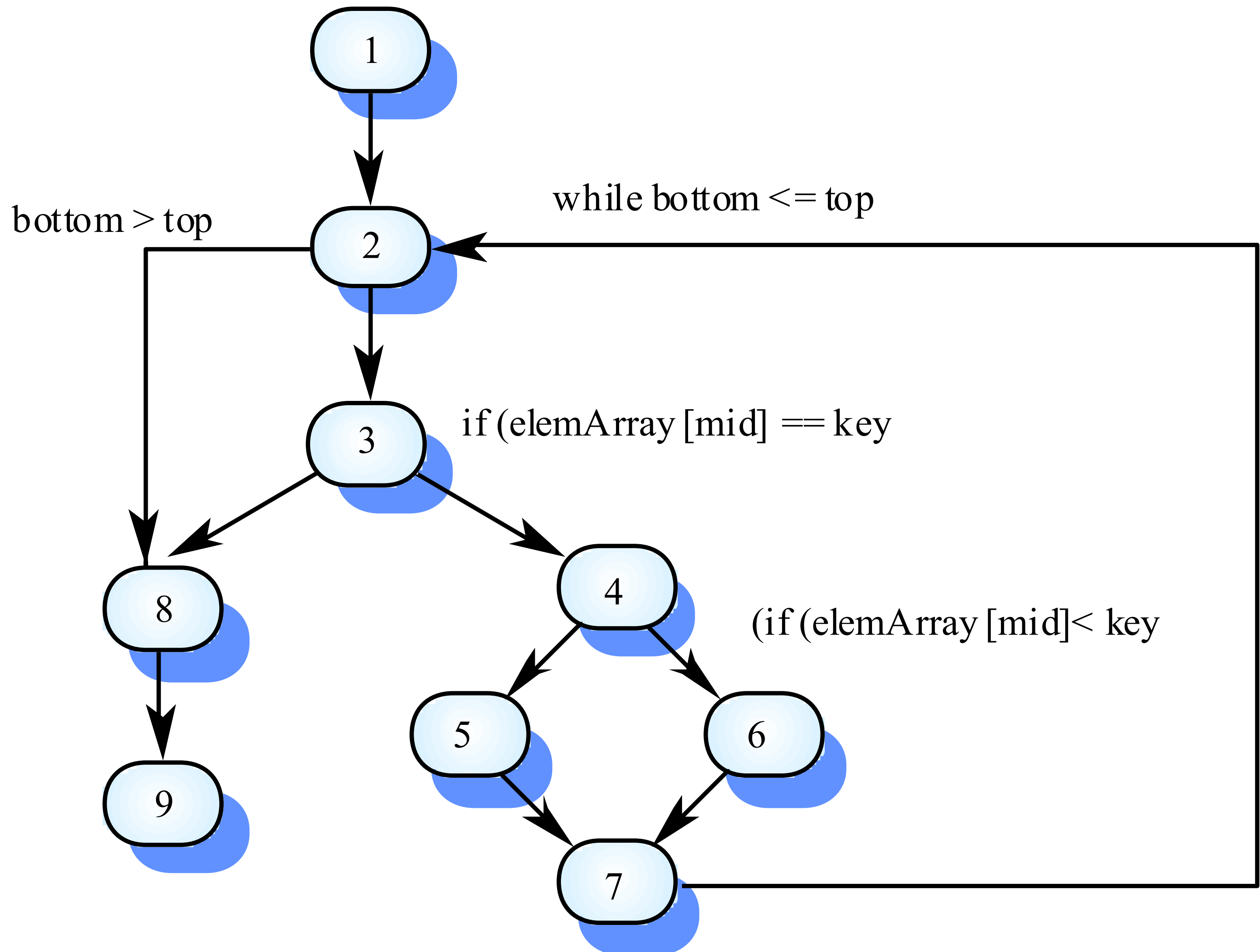
- Rozdelíme množinu vstupných hodnôt na *triedy ekvivalencie* (na ktorých očakávame, že sa bude program správať rovnakým spôsobom)
  - kladné čísla / záporné čísla / nula
  - prázdne polia / neprázdne polia
  - reťazce s medzerami / reťazce bez medzier
  - hľadaný prvok je v poli / nie je v poli

- Z každej triedy ekvivalencie vyberáme niekoľko prvkov.
- Sústreďujeme sa tiež na *hraničné prvky* (ležiace medzi triedami ekvivalencie)
  - príklad: vyhľadávanie v poli
    - nulová vs. nenulová veľkosť (test: pre 0, 1, 2, n)
    - prvok nájdený vs. nenájdený (test: nenájdený, nájdený na začiatku, v strede, na konci)
  - príklad: výpočet dane
    - triedy ekvivalencie – pásma v príslušnej tabuľke

Testovanie „white-box“

- vyberáme podmnožinu všetkých ciest programom
  - pre každdú cestu určíme jeden alebo viac *test cases*
- ktorú podmnožinu ciest vybrať ?
  - napr: pokrytie všetkých príkazov
  - napr: pokrytie všetkých nezávislých ciest v grafe programu
    - t.j. takých, kde každá cesta pokrýva aspoň 1 hranu nepokrytú inou cestou

Graf programu – príklad



Nezávislé cesty - príklad

- 1, 2, 3, 8, 9
- 1, 2, 3, 4, 6, 7, 2, 3, 8, 9
- 1, 2, 3, 4, 5, 7, 2, 3, 8, 9
- 1, 2, 3, 4, 6, 7, 2, 8, 9
- pre každú cestu musí existovať aspoň 1 *test case*
- pre overenie pokrytia existujú softvérové nástroje

Automatické testovanie

- dá sa spúšťať opakovane, napr. pri nových interných verziách, po vykonaní opráv, ...
- je možné použiť rôzne nástroje
- niektoré prístupy: písať test cases spolu s kódom

Ručné testovanie

- vykonáva niektorý z testerov podľa písomného testovacieho scenára
- vhodné ponechať istú voľnosť – väčšia šanca na odhalenie chyby

Integračné testovanie

- starší prístup: vytvoriť komponenty a (niekedy neskôr) ich začať spájať
- modernejšie riešenie: pravidelné vytváranie interných verzií systému (napr. týždenne alebo denne)
  - automatické funkčné testy vykonávané často
  - ostatné testy (napr. výkonnostné) podľa potreby

Správy o chybách

- popis, ako chybu vyvolať (úloha testera!)
  - treba mať dobre definované podmienky (napr. referenčnú databázu)
- ďalšie informácie (dátum, verzia, tester, komponent/funkcia, závažnosť, referenčné číslo)

- databáza chýb

6.
Skúšobná, pilotná a ostrá prevádzka

Skúšobná prevádzka

- cieľ: odstrániť posledné chyby systému
- vykonáva sa v prostredí používateľa
  - servery
  - pracovné stanice
  - údaje
  - iné aplikácie
- správna funkčnosť zatiaľ nie je zaručená
  - prevádzka paralelne so „starým“ systémom
  - aplikácia by však mala byť funkčná
- zavŕšenie používateľských testov

Skúšobná prevádzka (2)

- postup:
  - inštalácia systému
  - (konverzia údajov)
  - zaškolenie účastníkov
  - oživenie komunikácie s externými systémami
- testovanie
  - používatelia
  - testovací tím

Akceptačné testovanie

- cieľ: zákazník sa má uistiť, že produkt je vyhovujúci
- vykonáva sa v prostredí zákazníka
- vykonávajú ho pracovníci zákazníka (testeri alebo používatelia)

Pilotná prevádzka

- cieľ: vyskúšať aplikáciu v prevádzke blízkej realite
- aplikácia by už mala byť vo finálnej podobe, dolaďuje sa spôsob používania, vnútrofiremné pravidlá, ...
- niekedy sa systém používa paralelne s pôvodným
- často pilotná prevádzka prebieha v obmedzenom rozsahu (napr. jedna pobočka)
- výstupom sú niekedy požiadavky na zmenu (realizované v samostatnom projekte)