B4B33PSY: Počítačové systémy

Lekce 07. Testování

Petr Štěpán

stepan@fel.cvut.cz

1. listopadu 2025

Testování softwaru

- Tato přednáška je pouze **úvodem** do testování softwaru
  - Cílem je pomoci Vám se studiem v programovacích předmětech *Procedurální programování* - **PRP**, *Programování v JAVA* - **PJV**, *Algoritmizace* - **ALG**
  - Testování Vám pomůže i v navazujících předmětech *Architektury počítačů* - **APO** a *Operačních systémech* - **OSY**
- Hlubší znalosti testování lze získat v předmětu *Testování softwaru* - **TS1**
  - 4. semestr
  - povinný pouze pro specializaci Software
- Testování pro jazyk Python se objeví i v předmětu *Řešení problémů a hry* - **RPH**

Testování softwaru - Motivace

Co si ukážeme v této přednášce:

- Jak vylepšit vývoj programů pro PRP a pro PSY
- Při zadání programovacích úloh v PRP a PSY dostanete sadu vstupních dat a sadu očekávaných dat
  - Ukážeme si jak zautomatizovat testování
  - Doporučujeme použít všechna testovací data
  - Úspěšné splnění všech testů není zárukou správnosti programu, ale výrazně zvyšují pravděpodobnost správné funkčnosti
  - Chyba v testu znamená prakticky jistotu, že Brute problém odhalí

Testování softwaru - Motivace

Co ještě ukážeme v této přednášce:

- Zveřejněná testovací data nejsou úplná
  - Brute má další testovací data na kterých testuje Váš program
- Jak zjistit, jestli jsou zadaná data dostatečná pro Váš program?
  - Zjistíte, zda zveřejněná testovací data, podrobně otestují Váš program.
  - Je potřeba otestovat všechny cesty algoritmu skrz Váš program.
  - I tak je možné, že systém Brute odhalí nějaký problém - tento problém bude vést ke změně algoritmu, budete muset předělat program

Testování software

- Cílem testování je zajistit správnou funkčnost
  - Z minulé přednášky víte, co je chyba - nesoulad s návrhem, případně s požadavky
- Testování je povinné pro tzv. bezpečnostně kritický software
  - Jedná se o programy řídící auta, letadla, železnice
  - Testování je povinnou součástí certifikace
  - Další povinnou součástí certifikace je například využití pouze některých částí programovacího jazyka
    - MISRA C, MISRA C++ - specifikace co lze použít pro software řídicí auta

Životní cyklus programu

- Správně by každý program měl vznikat takto:
  1. Analýza požadavků
  2. Návrh programu
  3. Specifikace programu
  4. Implementace – programování
  5. Testování
  6. Zavedení do provozu
  7. Případně znovu od 1.úprava požadavků

Návrh a specifikace programu

Návrh a specifikace programu:

- většinou se návrh a specifikace zadává prostým textovým popisem
- někdy lze pro specifikaci použit formální metody - složitější

Výše uvedený ostup je ideální stav

- reálně často návrh či specifikace chybí nebo se dodělávají později podle programu
- to vede k problémům při testování, ke špatnému návrhu testů.

Životní cyklus testování

- I testování má svůj vlastní cyklus:
  1. Analýza požadavků
  2. Plánování testů
  3. Návrh testovacích případů
  4. Příprava testovacího prostředí
  5. Spuštění testů
  6. Vyhodnocení a reporting
  7. Uzavření testů

Funkční vs. Ne-funkční testování

Základní dělení testů:

- **Funkční:**
  - Ověřuje, že systém dělá to, co má.
  - Tedy, že dané vstupy dávají správné výstupy
- **Ne-funkční:**
  - Ověřuje vlastnosti systému – výkon, rychlost, škálovatelnost
    - např. kolik dotazů je možné paralelně v systému zpracovat.

Typy testování podle úrovně

Co lze testovat:

- Celé programy - Akceptační uživatelské testy
  - často programy mají interakci s uživatelem a je potřeba používat nástroje pro simulaci uživatelských vstupů
    - pohyb myši, stisk tlačítka
    - zadání textu do vstupního okna
  - jednodušší je pokud program zpracovává vstup ze souboru nebo ze stadarního vstupu a vytvoří data
    - toto již znáte z odevzdávání domácích úkolů
    - stejný princip je ve vyhodnocování Vašich programů v Brute

Typy testování podle úrovně

Co lze testovat:

- Celé programy - Akceptační uživatelské testy
  - nejsložitější je testování řídicích systémů
    - závislé na časování vstupů
    - velmi špatně se reproduje chybné chování
    - vytváření logů s přesným časem událostí a jejich přehrávání s přesným časováním
      - například ROS - Robot Operting System - umí ukládat data do takzvaných rosbagů a pak je přehrávat
      - vytváření logů zatěžuje řídicí počítač, obzvláště, pokud je dat hodně (kamera, LiDary)

Typy testování podle úrovně

Co lze testovat:

- Části programů - jednotkové testování (Unit testy)
  - izolovaná funkce
    - definujeme výstupní hodnoty pro různé vstupní hodnoty
    - kontrola správné činnosti jednotlivých funkcí
    - pro provedení testů je nutné zabalit funkci do programu, který ji připraví vstupní data a zkontroluje vygenerovaná data
  - celé moduly
    - kontrola návaznosti jednotlivých funkcí
    - stále ještě to není celý program ale jedná se již o složitější systém než jednotlivé funkce
    - i zde je nutné vytvořit obalující program pro komunikaci s modulem
- Integrační testy
  - testy kompatibility dat mezi moduly
  - testování správné činnosti koordinace modulů

Testovací případy

Testování je založeno na kontrole testovacích případů:

- Definice: Testovací případ (test case) popisuje konkrétní situaci, kterou tester ověřuje, aby zjistil, zda systém funguje podle požadavků.
  - Je základní jednotkou testovacího procesu
- Účel:
  - Zajistit přesnou reprodukovatelnost testu.
  - Pokrýt různé vstupy, podmínky a očekávané výsledky.
  - Poskytnout jasné důkazy o chování systému.
- Co musí testovací případ obsahovat:
  - Prerekvizity - jaký má být stav systému, případně co se musí provést, aby se systém převedl z počátečního stavu
  - Kroky testu - přesné vstupy, jejich pořadí a časování
  - Očekávaný výsledek - jaké hodnoty by měl obsahovat výsledek a jakým způsobem tyto hodnoty ověřit, kde je možné výstupní hodnoty získat

Typy testování podle přístupu

Základní typy testů

- **Black box** – testování bez znalosti vnitřní struktury
- **White box** – testování s úplnou znalostí kódu
- **Gray box** – kombinace obou přístupů

Black Box Testing

Testování „zvenčí“ – bez znalosti vnitřního fungování systému.

- Testování je založeno na analýze požadavků a zohledňuje i návrh a specifikaci programu
- Zaměřuje se pouze na **vstupy** a **výstupy** testovaného programu, nebo jeho části.
  - Pro každý požadavek je nutné vygenerovat alespoň jeden testovací případ

Uvedeme si praktický příklad specifikace, který později otestujeme:

- Napište funkci, která v poli celých čísel najde hodnotu, která je třetí největší v zadaném vstupu. Návratová hodnota funkce bude true, pokud funkce našla třetí největší hodnotu a false pokud taková hodnota neexistuje.
  - jde nám o třetí největší hodnotu, tedy pokud se opakují prvky s maximální hodnotou, tak se nepočítají jako další prvek
  - například v poli `10 10 8 10 8 10 6 10 8` je třetí největší hodnota 6, největší hodnota je 10 a druhá největší hodnota je 8.

Příklady Black Box testů

- Funkční testování
- Testování podle požadavků
- Testování uživatelského rozhraní (UI)
- Testování hranic vstupů (Boundary Value Analysis)
- Testování ekvivalenčních tříd
  - neboli testujeme, zda výstup jednoho programu je shodný jako výstup jiného programu
  - nejčastěji se jedná o porovnání nové a staré verze, kdy chceme zachovat funkčnost a případně ověřit rozdílnou funkčnost podle nových požadavků

Praktický příklad Black Box testu

- Na základě výše uvedené specifikace, jsme vytvořili funkci `findThirdLargest`:

```
bool findThirdLargest(int array[], int len, int *third_largest) {
  int first_max, second_max, third_max;
  bool second_max_set = false;
  bool third_max_set = false;
  first_max = array[0];
  for (int i=1; i<len; i++) {
    if (array[i]>first_max) {
      third_max = second_max;
      third_max_set = second_max_set;
      second_max = first_max;
      second_max_set = true;
      first_max = array[i];
    } else if ((array[i]<first_max) && ((!second_max_set) ||
array[i]>second_max)) {
      third_max = second_max;
      third_max_set = second_max_set;
      second_max = array[i];
      second_max_set = true;
    } else if ((array[i]<first_max) && (array[i]<second_max) && ((!
third_max_set) || array[i]<third_max)) {
```

Praktický příklad Black Box testu

```
        third_max = array[i];
        second_max_set = true;
      }
    }
    if (third_max_set) {
      *third_largest = third_max;
    }
    return third_max_set;
}
```

Praktický příklad Black Box testu

- Nyní jsem požádal AI - ChatGPT o navržení testovacích případů, podle naší specifikace. Dostal jsem toto:

Test Case 1 – Základní funkčnost

ID: TC01 Název: Tři různé prvky

Vstup: [5, 2, 9]

Očekávaný výsledek: 2

Poznámka: 3. největší = nejmenší z těch tří.

Test Case 2 – Více prvků, různé hodnoty

ID: TC02 Název: Více než tři prvky

Vstup: [10, 5, 8, 20, 12]

Očekávaný výsledek: 10

Poznámka: Seřazeno [20, 12, 10, 8, 5].

Test Case 3 – Duplicitní hodnoty

ID: TC03 Název: Pole s duplicitami

Vstup: [4, 4, 6, 6, 8, 8]

Očekávaný výsledek: 4

Poznámka: Třetí unikátní hodnota je 4.

Test Case 4 – Třetí největší je záporné číslo

ID: TC04 Název: Obsahuje záporné hodnoty

Vstup: [−1, −10, −5, −3, −2]

Očekávaný výsledek: −3

Poznámka: Seřazeno [−1, −2, −3, −5, −10].

Test Case 5 – Méně než tři prvky

ID: TC05 Název: Nedostatek prvků

Vstup: [7, 1]

Očekávaný výsledek: NEEXISTUJE

Poznámka: Funkce musí správně ošetřit neplatný vstup.

Praktický příklad Black Box testu

- Protože budeme testovat pouze funkci, tak potřebujeme program, který volání funkce zabalí
  - vytvoříme program, který načte testovací data a vytiskne na výstup návratovou hodnotu
  - **POZOR** můžeme nechtěně udělat chybu i v tomto obalovacím programu.

Program volající testovanou funkci

```
int main() {
  int len, third, *array;
  bool ret;

  if (scanf("%d", &len)==1) {
    array = (int *)malloc(sizeof(int)*len);
    if (array!=NULL) {
      for (int i=0; i<len; i++) {
        if (scanf("%d",&(array[i]))!=1) {
          fprintf(stderr, "Chyba pri zadani prvku pole index %i\n", i); exit(1);
        }
      }
      if (findThirdLargest(array, len, &third)) {
        printf("%d\n", third);
      } else {
        printf("NEEXISTUJE\n");
      }
    } else {
      fprintf(stderr, "Nelze allokovat pole o velikosti %i\n", len); exit(1);
    }
  } else {
    fprintf(stderr, "Chybne zadana velikost pole\n"); exit(1);
  }
  return 0;
}
```

Testování praktického příkladu

Testy navržené umělou inteligencí jsem přepsal do souborů:

- vytvořil jsem skupinu vstupních souborů in_000.txt až in_005.txt obsahující vstupní data, délku pole a hodnotu jeho prvků
- vytvořil jsem skupinu výstupních souborů out_000.txt až out_005.txt obsahující očekávané hodnoty, pro návratovou hodnotu false, obalovací program vytiskne NEEXISTUJE a pro návratovou hodnotu true, vytiskne hodnotu třetího prvku

Nyní můžeme testovat buď ručně:

- spustit program pro každý vstup a otestovat správnost návratové hodnoty

nebo automaticky:

- vytvoříme program/skript, který otestuje zda program produkuje správná data pro všechny testovací případy

Automatické testování

Využijeme znalosti BASHe

- skript spustí program na všech vstupních datech a porovná očekávaná výstupní data se získanými výstupními daty
  - základem je spustit program pro všechny vstupní soubory vytvoříme soubory, které budou obsahovat výstup testované verze programu

```
for inp in in_*.txt
do
  out=prgout${inp#in}
  ./third_max <$inp >$out
```

Skript dále pokračuje

- porovná očekávaný výstup se skutečným výstupem

```
  out_orig=out${inp\#in}
  if diff $out $out_orig
```

Automatické testování

- výsledkem programu `diff` je
  - True, pokud jsou soubory shodné
  - False, pokud jsou soubory různé

```
if diff $out $out_orig
then
  echo Test $inp Ok
else
  echo Test $inp Failed
  let "fail++"
fi
```

Celý testovacího skriptu

```
fail=0s
for inp in in_*.txt
do
  out=prgout${inp#in}
  out_orig=out${inp#in}
  ./prg <$inp >$out
  if diff $out $out_orig >/dev/null 2>/dev/null
  then
    echo Test $inp Ok
  else
    echo Test $inp Failed; let "fail++"
  fi
done

if [ "$fail" == 0 ]
then
  echo Tests passed
else
  echo Tests fail; exit 1
fi
```

Zapojení makefile

Protože již umíme nástroj make, tak si můžeme testování přidat i do Makefile:

```
all: third_max

.PHONY: clean all test

third_max: third_max_final.o
  $(CC) $(LDFLAGS) -o $@ $^

clean:
  rm -f *.o
  rm -f third_max

test: third_max
  bash ./make_test.sh
```

Anketa

Už jste někdy použili podobně zautomatizované testování:

A Ano, používám ho běžně
B Ano, ale pouze občas
C Ano, jednou už jsem podobně testoval
D Ne, zatím jsem všechny testy spouštěl ručně

Výsledek testování

Po spuštění testů `make test` dostáváme:

```
bash ./make_test.sh
Test in_000.txt Ok
Test in_001.txt Ok
Test in_002.txt Ok
Test in_003.txt Ok
Test in_004.txt Ok
Tests passed
```

Všechny testy vyšly, je tedy vše v pořádku?

White Box Testing

White box testování, někdy také open box testování

- Testování s plnou znalostí **zdrojového kódu**.
- Zaměřuje se na **logiku, větvení a vnitřní strukturu programu.**
- Podobnost s Black box testováním:
  - Požadavky, specifikace a návrh systému slouží pro definici správných výstupních hodnot
- Rozdílnost s Black box testováním
  - Počet testovacích případů a jejich volba je veden vnitřní strukturou testů

Techniky White Box testování

White box testing se zaměřuje na:

- Pokrytí příkazů (statement coverage)
- Pokrytí větví (branch coverage)
- Pokrytí podmínek (condition coverage)
- Testování smyček a cest (path testing)

Co znamená pokrytí příkazů, větví, podmínek, případně cest?

- Zjistit co program vykonával při svém běhu, kterými cestami procházel a zda se všechen kód použil alespoň jednou.

Je to dostatečné?

- Není, i když zjistíme, že program běžel všemi svými částmi, tak ja možné, že některé chyby zůstanou skryté, protože se jejich vliv na konečný stav ztratí.
- Je to ale minimum, co můžeme udělat pro co nejdůkladnější otestování programu.

Techniky White Box testování

Jak zjistit kudy program procházel?

- Je nutné program přeložit s přepínači `-fprofile-arcs -ftest-coverage`
  - tyto přepínače zajistí vložení funkcí, které budou hlásit, zda tudy program prošel
  - funkce se vkládají prakticky na každý řádek programu, případně i do každé cesty programu (probereme za chvíli)
  - druhý přepínač zajistí vygenerování souboru o rozmístění kontrolních bodů v programu

Po překladu vznikne soubor `jméno_modulu.gcno`

- tento soubor obsahuje informace o poloze značek ve zdrojvém kódu

Po běhu vznikne soubor `jméno_modulu.gcda`

- tento soubor obsahuje informace o počtu proběhnutí značek

Pokrytí kódu

Soubor `.gcda`, který shromažďuje data o průchodu programu se rozšiřuje každým dalším během programu

- je nutné před začátkem testů případně smazat starý soubor `.gcda`, který obsahuje informace o předchozích testech
- spustit program znovu pro každý test a všechny nové průchody se přidají do souboru `.gcda`

Soubory `.gcno` a `.gcda` jsou binární a jejich analýza je pro člověka složitá.

- Pro přehlednou analýzu slouží nástroj **lcov**
  - tento program projde oba soubory a vytvoří textový statistický soubor `.info`, který obsahuje informace o vykonaných řádkách programu

Pro ještě přehlednější analýzu, je možné použít nástroj **genhtml**

- tenot nástroj generuje z .info souboru html stránky a názorně uvedenou statistikou testů

Zapojení makefile

Vše si opět můžeme usnadnit vložením do Makefile:

```
CC=gcc
LDFLAGS+=-fprofile-arcs -ftest-coverage
CFLAGS+=-fprofile-arcs -ftest-coverage

all: third_max

.PHONY: clean all test cover

third_max: third_max_final.o
  $(CC) $(LDFLAGS) -o $@ $^

clean:
  rm -f *.o third_max *gcno

test: third_max
  rm -f *gcda *info
  bash ./make_test.sh

cover: third_max test
  lcov --capture --directory . --output-file coverage-ful.info
  genhtml coverage-ful.info --output-directory out-ful
```

Analýza black box testů

Po zadání `make clean cover` se provede:

- vymazání starého programu – cíl `clean`
- protože cíl `cover` závisí na existenci programu, provede se překlad s nastavenými přepínači
- cíl `cover` závisí také na provedení testů, proto se pustí testy
- nakonec se vytvoří soubor .info a html stránky s výsledky pokrytí testy připravenými na základě black box specifikací

Analýza black box testů

LCOV - code coverage report

		Hit	Total	Coverage
Current view:	top level - example - third_max_final.c (source / functions)			
Test:	coverage-ful.info	**Lines:**	30	38	78.9 %
Date:	2025-11-02 14:08:32	**Functions:**	2	2	100.0 %

```
      Line data    Source code
    1             : #include <stdio.h>
    2             : #include <stdlib.h>
    3             : #include <malloc.h>
    4             : #include <stdbool.h>
    5             :
    6           5 : bool findThirdLargest(int array[], int len, int *third_largest) {
    7             :   int first_max, second_max, third_max;
    8           5 :   bool second_max_set = false;
    9           5 :   bool third_max_set = false;
   10             :
   11           5 :   first_max = array[0];
   12             :
   13          21 :   for (int i=1; i<len; i++) {
   14          16 :     if (array[i]>first_max) {
   15           4 :       third_max = second_max;
   16           4 :       third_max_set = second_max_set;
   17           4 :       second_max = first_max;
   18           4 :       second_max_set = true;
   19           4 :       first_max = array[i];
   20          12 :     } else if ((array[i]<first_max) && ((!second_max_set) || array[i]>second_max)) {
   21           9 :       third_max = second_max;
   22           9 :       third_max_set = second_max_set;
   23           9 :       second_max = array[i];
   24           9 :       second_max_set = true;
   25           3 :     } else if ((array[i]<first_max) && (array[i]<second_max) && ((!third_max_set) || array[i]<third_max)) {
   26           0 :       third_max = array[i];
   27           0 :       second_max_set = true;
   28             :     }
   29             :   }
   30           5 :   if (third_max_set) {
   31           4 :     *third_largest = third_max;
   32             :   }
   33           5 :   return third_max_set;
   34             : }
```

Analýza black box testů

Analýze podléhá i obalovací program, protože se nachází ve stejném modulu:

```
        :
5       : int main() {
        :   int len;
        :   int *array;
        :   bool ret;
        :   int third;
        :
5       :   if (scanf("%d", &len)==1) {
5       :     array = (int *)malloc(sizeof(int)*len);
5       :     if (array!=NULL) {
26      :       for (int i=0; i<len; i++) {
21      :         if (scanf("%d",&(array[i]))!=1) {
0       :           fprintf(stderr, "Chyba pri zadani prvku pole index %i\n", i);
0       :           exit(1);
        :         }
        :       }
5       :       if (findThirdLargest(array, len, &third)) {
4       :         printf("%d\n", third);
        :       } else {
1       :         printf("NEEXISTUJE\n");
        :       }
        :     } else {
0       :       fprintf(stderr, "Nelze allokovat pole o velikosti %i\n", len);
0       :       exit(1);
        :     }
        :   } else {
0       :     fprintf(stderr, "Chybne zadana velikost pole\n");
0       :     exit(1);
        :   }
5       :   return 0;
        : }
```

Analýza black box testů

Z analýzy je na první pohled zřejmé:

- ve funkci findThirdLargest se nikdy neprovedli řádky 26 a 27
  - tedy tyto řádky nebyly otestovány
- ve funkci main beyly provedeny řádky 47, 48, 57, 58, 61 a 62
  - to je správné, protože se jedná o hlášení vnitřních problémů testů
  - pokud by se tyto řádky provedli, tak by to byla naopak chyba

Doplníme tedy testy o nový testovací případ:

Test Case 6 – Změna třetí hodnoty

ID: TC06 Název: Změna 3. hodnoty

Vstup: [9 8 7]

Očekávaný výsledek: 7

Poznámka: Pokrytí kódu

Analýza black box testů

Výsledkem je, že poslední test neuspěje, protože jsme v programu měli chybu

```
    } else if ((array[i]<first_max) && (array[i]<second_max) &&
((!third_max_set) || array[i]<third_max)) {
      third_max = array[i];
      second_max_set = true;
    }
```

Sprrávně má být:

```
    } else if ((array[i]<first_max) && (array[i]<second_max) &&
((!third_max_set) || array[i]<third_max)) {
      third_max = array[i];
      third_max_set = true;
    }
```

Analýza white box testů

LCOV - code coverage report

		Hit	Total	Coverage
Current view:	top level - example - third_max_final.c (source / functions)			
Test:	coverage-ful.info	**Lines:**	32	38	84.2 %
Date:	2025-11-02 14:11:10	**Functions:**	2	2	100.0 %

```
      Line data    Source code
    1             : #include <stdio.h>
    2             : #include <stdlib.h>
    3             : #include <malloc.h>
    4             : #include <stdbool.h>
    5             :
    6           6 : bool findThirdLargest(int array[], int len, int *third_largest) {
    7             :   int first_max, second_max, third_max;
    8           6 :   bool second_max_set = false;
    9           6 :   bool third_max_set = false;
   10             :
   11           6 :   first_max = array[0];
   12             :
   13          24 :   for (int i=1; i<len; i++) {
   14          18 :     if (array[i]>first_max) {
   15           4 :       third_max = second_max;
   16           4 :       third_max_set = second_max_set;
   17           4 :       second_max = first_max;
   18           4 :       second_max_set = true;
   19           4 :       first_max = array[i];
   20          14 :     } else if ((array[i]<first_max) && ((!second_max_set) || array[i]>second_max)) {
   21          10 :       third_max = second_max;
   22          10 :       third_max_set = second_max_set;
   23          10 :       second_max = array[i];
   24          10 :       second_max_set = true;
   25           4 :     } else if ((array[i]<first_max) && (array[i]<second_max) && ((!third_max_set) || array[i]<third_max)) {
   26           1 :       third_max = array[i];
   27           1 :       third_max_set = true;
   28             :     }
   29             :   }
   30           6 :   if (third_max_set) {
   31           5 :     *third_largest = third_max;
   32             :   }
   33           6 :   return third_max_set;
   34             : }
```

Výsledek testování

Po spuštění testů `make test` dostáváme:

```
bash ./make_test.sh
Test in_000.txt Ok
Test in_001.txt Ok
Test in_002.txt Ok
Test in_003.txt Ok
Test in_004.txt Ok
Test in_005.txt Ok
Tests passed
```

Testy prošly, kód je pokryt

Můžeme testovat něco víc?

Pokrytí větví - cest

Pokrytí větví zkoumá, zda jsme skutečně prošli všemi možnými cestami uvnitř programu.

- Pokud navštívíme všechny řádky programu, to by mělo stačit?
- Nestačí, protože podmínky se vyhodnocují zkráceně, je možné, že se některé části nepoužily a tím pádem neotestovaly.

Příklad podmínky v programu:

```
    if ((array[i]<first_max) && (array[i]<second_max) && ((!
third_max_set) || array[i]<third_max)) {
```

Jedná se o podmínku typu:

```
    if (a && b && (c || d)) {
```

Pokrytí větví - cest

Jak se tato podmínka vyhodnocuje?

```
if (a && b && (c || d)) {
```

- Pokud `a==false` pak se podmínky b,c,d vůbec nekontrolují, protože výsledek je `false`
- Pokud `a==true , b==false` pak se podmínky c,d vůbec nekontrolují, protože výsledek je `false`
- Pokud `a==true , b==true, c==true` pak se podmínka d vůbec nekontroluje, protože výsledek je `true`
- Pokud `a==true , b==true, c==false` pak se konečně i podmínka d spočte a výsledek je hodnota `d`

Analýza cest

Analýzu cest zapneme v nástroji lcov přidáme přepínačem

`--rc lcov_branch_coverage=1` a v nástroji genhtml přepínačem

`--branch-coverage`

```
cover: third_max test
  lcov --capture --rc lcov_branch_coverage=1 --directory . --output-file coverage-ful.info
  genhtml coverage-ful.info --branch-coverage --output-directory out-ful
```

Analýza cest

LCOV - code coverage report

		Hit	Total	Coverage
Current view:	top level - example - third_max_final.c (source / functions)			
Test:	coverage-ful.info			
Date:	2025-11-02 14:24:00			
	Lines:	32	38	84.2 %
	Functions:	2	2	100.0 %
	Branches:	23	30	76.7 %

```
                Branch data     Line data    Source code
       1                  :              :  #include <stdio.h>
       2                  :              :  #include <stdlib.h>
       3                  :              :  #include <malloc.h>
       4                  :              :  #include <stdbool.h>
       5                  :              :
       6                  :            6 :  bool findThirdLargest(int array[], int len, int *third_largest) {
       7                  :              :    int first_max, second_max, third_max;
       8                  :            6 :    bool second_max_set = false;
       9                  :            6 :    bool third_max_set = false;
      10                  :              :
      11                  :            6 :    first_max = array[0];
      12                  :              :
      13        [ +  + ]  :           24 :    for (int i=1; i<len; i++) {
      14        [ +  + ]  :           18 :      if (array[i]>first_max) {
      15                  :            4 :        third_max = second_max;
      16                  :            4 :        third_max_set = second_max_set;
      17                  :            4 :        second_max = first_max;
      18                  :            4 :        second_max_set = true;
      19                  :            4 :        first_max = array[i];
      20  [ +  +  +  +    :           14 :      } else if ((array[i]<first_max) && ((!second_max_set) || array[i]>second_max)) {
                +  + ]
      21                  :           10 :        third_max = second_max;
      22                  :           10 :        third_max_set = second_max_set;
      23                  :           10 :        second_max = array[i];
      24                  :           10 :        second_max_set = true;
      25  [ +  +  +  -    :            4 :      } else if ((array[i]<first_max) && (array[i]<second_max) && ((!third_max_set) || array[i]<third_max)) {
             -  +  -  - ]
      26                  :            1 :        third_max = array[i];
      27                  :            1 :        third_max_set = true;
      28                  :              :      }
      29                  :              :    }
      30        [ +  + ]  :            6 :    if (third_max_set) {
      31                  :            5 :      *third_largest = third_max;
      32                  :              :    }
      33                  :            6 :    return third_max_set;
      34                  :              :  }
```

Analýza cest

Z analýzy je vidět:

- ve funkci findThirdLargest na řádku 25 nedojde ke průchodu některými větvemi

Doplníme tedy testy o nový testovací případ:

Test Case 7 – Změna třetí hodnoty

ID: TC07 Název: Změna 3. hodnoty

Vstup: [9, 8, 5, 6, 7]

Očekávaný výsledek: 7

Poznámka: Pokrytí větví

Test Case 8 – Opakování maxima

ID: TC08 Název: Opakování maxima a druhého maxima

Vstup: [9, 8, 5, 9, 6, 3, 8, 7]

Očekávaný výsledek: 7

Poznámka: Pokrytí větví

Analýza cest

Výsledkem je, že nové testy neuspějí, protože jsme v programu měli chybu

```
    } else if ((array[i]<first_max) && (array[i]<second_max) &&
((!third_max_set) || array[i]<third_max)) {
      third_max = array[i];
      third_max_set = true;
    }
```

Sprrávně má být:

```
    } else if ((array[i]<first_max) && (array[i]<second_max) &&
((!third_max_set) || array[i]>third_max)) {
      third_max = array[i];
      third_max_set = true;
    }
```

Analýza cest

LCOV - code coverage report

		Hit	Total	Coverage
Current view: top level - example - third_max_final.c (source / functions)				
Test: coverage-ful.info	Lines:	32	38	84.2 %
Date: 2025-11-02 14:51:27	Functions:	2	2	100.0 %
	Branches:	27	30	90.0 %

```
                Branch data     Line data    Source code
       1                  :              : #include <stdio.h>
       2                  :              : #include <stdlib.h>
       3                  :              : #include <malloc.h>
       4                  :              : #include <stdbool.h>
       5                  :              :
       6                  :            8 : bool findThirdLargest(int array[], int len, int *third_largest) {
       7                  :              :   int first_max, second_max, third_max;
       8                  :            8 :   bool second_max_set = false;
       9                  :            8 :   bool third_max_set = false;
      10                  :              :
      11                  :            8 :   first_max = array[0];
      12                  :              :
      13        [ +  + ]  :           37 :   for (int i=1; i<len; i++) {
      14        [ +  + ]  :           29 :     if (array[i]>first_max) {
      15                  :            4 :       third_max = second_max;
      16                  :            4 :       third_max_set = second_max_set;
      17                  :            4 :       second_max = first_max;
      18                  :            4 :       second_max_set = true;
      19                  :            4 :       first_max = array[i];
      20  [ +  +  +  +     :           25 :     } else if ((array[i]<first_max) && ((!second_max_set) || array[i]>second_max)) {
                +  + ]
      21                  :           12 :       third_max = second_max;
      22                  :           12 :       third_max_set = second_max_set;
      23                  :           12 :       second_max = array[i];
      24                  :           12 :       second_max_set = true;
      25  [ +  +  +  +     :           13 :     } else if ((array[i]<first_max) && (array[i]<second_max) && ((!third_max_set) || array[i]>third_max)) {
          +  +  +  + ]
      26                  :            7 :       third_max = array[i];
      27                  :            7 :       third_max_set = true;
      28                  :              :     }
      29                  :              :   }
      30        [ +  + ]  :            8 :   if (third_max_set) {
      31                  :            7 :     *third_largest = third_max;
      32                  :              :   }
      33                  :            8 :   return third_max_set;
      34                  :              : }
```

Analýza cest

Díky analýze cest se nám podařilo odstranit všechny chyby v programu.

Gray Box Testing

Kombinace obou přístupů předchozích postupů:

- Tester má **částečnou znalost** interní struktury systému.
- Často používané v **integračním testování**.
- Příklad:
  - Tester zná strukturu databáze, ale ne kód aplikace.
  - Může testovat:
    - správné dotazy a integritu dat,
    - chování systému při chybných vstupech,
    - výkon mezi moduly.

Závěr

- Testování je klíčové pro kvalitu softwaru
- Existují různé úrovně a přístupy
- **Black box**, **White box**, **Gray box** se liší mírou znalosti systému
- Kombinace přístupů je často nejefektivnější
  - Čím víc testů, tím lépe – pokud je zvládnete spočítat za rozumnou dobu (1 hodina - 1 den)