Softwarové inženýrství

7. přednáška

Radek Janoštík

Univerzita Palackého v Olomouci

6. 11. 2025

Outline

- Validace a verifikace – úvod
- Kontrola a revize kódu
- Testování SW
- Doporučená četba

Validace a verifikace – úvod

- Během každé fáze vývoje bychom měli ověřovat, zda „dělám to, co máme dělat“
  - Ověřování správnosti požadavků (oprávněnost, konzistentnost, úplnost, realizovatelnost, ověřitelnost)
  - Ověřování návrhu aplikace (má se provádět další iterace návrhu?)
  - Inspekce a revize zdrojových kódů, testování finálního produktu
- *Validace == vytváříme* `ten správný` *produkt?*
- *Verifikace == vytváříme ten produkt* `správně`*?*
- Cíl obojího: splnit *očekávání* a dodat produkt splňující *požadavky*
  - $\Rightarrow$ produkt je *dostatečný* pro své fungování
  - *Dostatečný* závisí na kontextu – funkce, očekávání, stav trhu
  - Pro jiné aplikace budeme vyžadovat jiné nároky

Validace a verifikace – přístupy

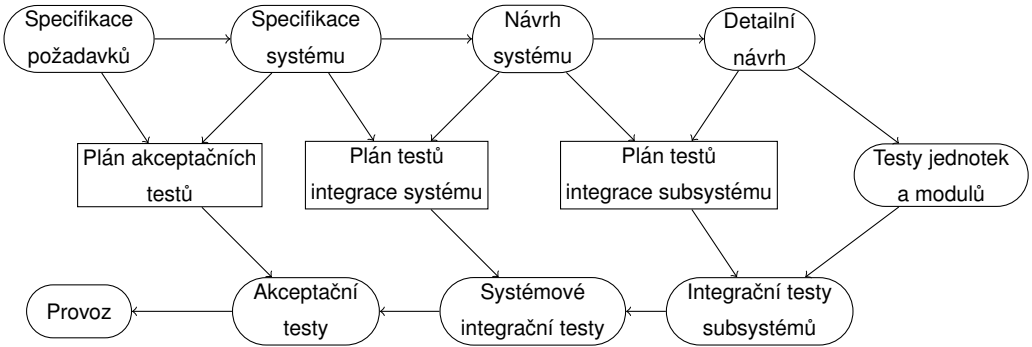
- Dva základní (doplňující se) přístupy:
- *Inspekce a revize softwaru* – zkontrolujeme, znovu projdeme výstupy z dané fáze
  - Kontrola znění požadavků, správnost diagramů
  - Ruční kontrola kódu, více očí
  - Korespondence DSP $\Leftrightarrow$ Návrh $\Leftrightarrow$ Kód
  - Statický proces
- *Testování softwaru* – spouštění SW s testovacími daty
  - Jaké vstupy dává na jaké výstupy
  - Jak se systém chová (výkon, robustnost)
  - Dynamický proces
- Měli bychom provádět oba způsoby
  - Zlepšení kvality výstupů
  - Ověření funkcionality

Debugování

- Nejedná se o testování – ale následující fázi po odhalení chyby testem(nebo praxí)
- Jak debugujete?
- Neexistuje jednoduchý návod („kuchařka“) jak zaručeně odhalit původ chyb
- Velkou roli hraje programátorova zkušenost
  - Hledá nějaké typické zdroje chyb (chybějící inkrementace, špatný pointer, . . .)
  - Programátor, který nezná daný systém, bude chybu hledat řádově déle než zkušený
- Velkou roli hrají použité nástroje a architektura software
  - Jak byste fungovali bez grafického, interaktivního debuggeru?
  - Zkoušel někdo např. klasické GDB?
- Původ chyby může být „hodně daleko“ od místa projevení (jak časově, tak místně)
- Důležité i rozumné logování za běhu (i na produkci)
- Poučit se z chyb (aktualizovat testy)

Plánování validace a verifikace

- Plánovat testy by se mělo v každé fázi SW procesu, tzv. *V-Model*:



- Původ jména – otočme tento diagram o 90° doprava

Plánování validace a verifikace

- Je potřeba rozhodnout, kolik času a energie budeme věnovat statickým metodám a kolik dynamickým
  - Neměli bychom žádnou vynechat úplně
- Ustanovit jednotný postup, rámec a formalizovat procesy (co dělat když)
  - Lépe se potom odhaduje časová náročnost, plánují „lidské zdroje“
- Doporučená struktura testovacího plánu:
  1. *Testovací proces* – popis hlavních fází testovacího procesu
  2. *Pokrytí požadavků* – které uživatelské požadavky budou kdy testovány
  3. *Testované elementy* – kterých částí procesu se test týká
  4. *Časový plán* – Kdy se která část bude testovat (korespondence s plánem vývoje)
  5. *Záznam výsledků* – Kde budou uloženy výsledky testů
  6. *HW a SW požadavky* – Co vše potřebujeme pro testování?
  7. *Podmínky* – které ovlivňují start a průběh testů

Kontrola(inspekce) a revize kódu

- Statický proces (bez spuštění), kde hledáme chyby, špatnosti a anomálie
  - Nemusí se týkat pouze zdrojového kódu, ale i specifikace, architektury
- Proč dělat kontrolu a revizi?
  - Dynamickým testováním se dvě chyby mohou „vyrušit“
  - Můžeme kontrolovat ještě nekompletní systém (nemusíme čekat, testování nekompletních drahé)
  - Nemusíme odhalit jen funkční chyby, ale nestandardní postupy a zvýšit udržovatelnost
- Revize kódu je až překvapivě mocná
  - Studie tvrdí, že lze odhalit až 60 % chyb
  - Dále bylo vyzkoumáno, že revize je levnější a odhalí více chyb než dynamické testování
- I přesto se špatně prosazuje – bývá nákladná a výsledek nemusí být ihned vidět
  - Nikde nevidíme zelené „fajfčičky a progress bary“

Doporučovaný proces revize

- Proces revize formalizován a „standardizován“ (IBM 70. léta, Fagan 1976-86, Graham 1993)
- Metody založené na: *pozorné a postupné čtení kódu řádek po řádku týmem lidí z rozličných oblastí*
- Zaměřují se pouze na hledání chyb nikoliv „vyšších návrhových prohřešků“
- Doporučuje se účast alespoň 4 osob v několika rolích
  - Autor, čtenář, tester a moderátor
  - Čtenář čte nahlas, každý může mít připomínky
- Je potřeba mít přesnou specifikaci
- Celý tým by měl znát standardy dané organizace
- Musí revidovat aktuální verzi

Doporučovaný proces revize

- *Moderátor* má na starost veškerou organizaci – vybírá tým
  - Zajistí úplnost specifikace, aktuálnost kódu
  - Určí místnost, svolá poradu, vede záznamy
- Nejprve by měla být *přehledová fáze*, kdy autor představí širší úsek kódu, uvede do kontextu
- Poté by měla být individuální příprava
- Samotná revize by neměla trvat dlouho (2 hodiny)
- Nemělo by se navrhovat jak chyby/prohřešky opravit, pouze najít
- *Autor* by pak měl zapracovat navrhované změny
- *Moderátor* zkontroluje a určí, zda je potřeba další iterace

Doporučovaný proces revize

- *Moderátor* navrhuje a udržuje *checklist* – co vše se má kontrolovat
  - Chyba dat (inicializace, magické konstanty, oddělovače, může nastat buffer overflow?)
  - Chyby toku (správnost podmínek, konečnost cyklů, break ve `switch`
  - Chyby vstupu a výstupu (využití všech vstupů, přiřazení výstupů, ošetření vstupů)
  - Chyby rozhraní (správný počet argumentů, správnost typů, pořadí)
  - Správa výjimečných stavů
- Částečně určeno programovacím jazykem
- Doporučení (empirické zjištění) rozsahu:
  - *přehledová fáze* – 500 řádků příkazů za hodinu
  - *individuální příprava* – 125 příkazů za hodinu
  - *samotná revize* – 90 – 125 příkazů za hodinu
- Časově nákladné – na kolik „člověkohodin“ vychází revize 100 řádků kódu?
  - I tak je to (údajně) polovina toho, co by stálo dynamické testování

Automatická (statická) analýza kódu

- Velké množství chyb (nebo podezření na ně) lze odhalit strojově
- $\Rightarrow$ nástroje na statickou analýzu kódu
- Nástroje, které prochází zdrojový kód a hledají podezřelé části
  - Detekce a správnost typů (u dynamicky typovaných jazyků)
  - Neinicializovaní proměnné, možnosti `null` pointeru
  - Výpočet možných hodnot v proměnných
  - Detekce nedosažitelného kódu
- Dnes nedílnou součástí IDE (analýza on-the-fly)
- Navrhují řádky, na které se máme v revizi zaměřit

Testování SW – úvod

- Dynamický proces – potřebujeme mít spustitelnou část SW
- Ověření, že SW splňuje požadavky
  - Důležité jak pro vývojáře i zákazníka
  - Pro každý uživatelský i systémový požadavek alespoň jeden test
  - Pro obecný produkt test veškeré funkcionality
  - $\Rightarrow$ Validace – úspěšný test = systém funguje správně
- Objevení chyb v SW, nežádoucího chování
  - Pády SW, chybné výpočty, narušení dat, ...
  - $\Rightarrow$ *defects testing* – úspěšný test = našli jsme chybu
- Zjistíme testováním zda „software neobsahuje chyby“?
  - Ne – vždy můžeme něco přehlédnout
  - Pouze přesvědčení, že je systém dostatečně dobrý
  - Jak moc dostatečně záleží na kvalitě, komplexitě testů

Testování SW – obecný model

- *Testovací případ (test case)* – specifikace vstupů a očekávaných výstupů
  - Specifikace části, které se testovací případ týká
  - Kombinace funkcionalit (samostatně může fungovat, v kombinace ne)
- *Testovací data* – pro každý případ musíme vytvořit vhodná data
  - Nemůžeme testovat úplně všechny možnosti (velké množství)
  - Musíme vhodně volit hodnoty (mezní případy, zástupce specifické množiny, . . .)
  - Často pevná pravidla, jaké hodnoty volit (organizačně dané)
  - Validní i nevalidní vstupy
- *Výsledky testů* – pro každý vstup výsledek
- *Zprávy o testování* – záznam, co prošlo a co ne
  - Vyvodit závěry z testů
  - Co se má přepracovat
  - Návrh nových případů, dat

Testování systémů – Integrační testy

- Testeři mají k dispozici zdrojové kódy – *white box, clear box, glass box*
- Snaha najít původce (komponentu) chyby $\Rightarrow$ bude se debugovat
  - Lokalizace náročná – složitá spolupráce komponent, chyba „může být kdekoliv“
- $\Rightarrow$ inkrementální přístup
  - Testujeme komponenty postupně a pomalu přidáváme další
  - Př. Dvě komponenty otestujeme čtyřmi testovacími případy
  - Přidáme třetí komponentu a další dva testovací případy (celkem 6)
  - Přidáme čtvrtou, dalších x testovacích případů (celkem $6 + x$)
  - ...
  - Pořadí součástí testovacího plánu (od důležitých, nejčastějších, komplikovaných?)
- Při implementaci nové funkcionality testovat i předešlé testovací případy (regrese)
  - Ověření, že jsme novou funkcionalitou „nerozbili“ předchozí
  - Drahé, snaha co nejvíce automatizovat

Testování systémů – testy před vydáním – funkční testování

- Testování *kandidáta na vydání (release candidate)* – ověření chování celého systému
- Nemáme k dispozici zdrojové kódy – *black box*
  - Zkoumáme pouze vstupy a výstupy
  - Původ chyby nehledáme, pouze ji odhalíme a předáme dál
- Cílená snaha rozbít systém, často zábavná až škodolibá činnost
  - Neměli by provádět autoři (author bias)
- Testování pomocí scénářů ze specifikace (ale změna dat, chování)
- Testovací případy netestují jednu funkcionalitu, ale posloupnost
  - Kontrola korektních přechodů mezi stavy
  - Doprovázeno *stavovými diagramy*

Funkční testování – obecné rady

- Volba scénářů a chování velmi ovlivněna testerovou zkušeností
  - Znalost programovacího jazyka, použitého frameworku *blackboxu* může být výhodou
- Několik doporučených bodů:
  - Volba vstupů, které vyvolají nějakou chybovou hlášku – kontrola hlášek, snaha vyvolat všechny (jak je známe?)
  - Pokusit se navrhnout vstupy, při kterých by mohlo dojít k pře/podtečení zásobníků
  - Několikrát opakovat tytéž vstupy (chaotické i záměrné)
  - Nebezpečné znaky na vstup i výstup
  - Pokusit se donutit systém vytvořit špatný výstup
  - Pokusit se dostat k neoprávněným datům
  - Příliš dlouhé vstupy/výstupy
  - Příliš krátké vstupy/výstupy

Testy výkonu

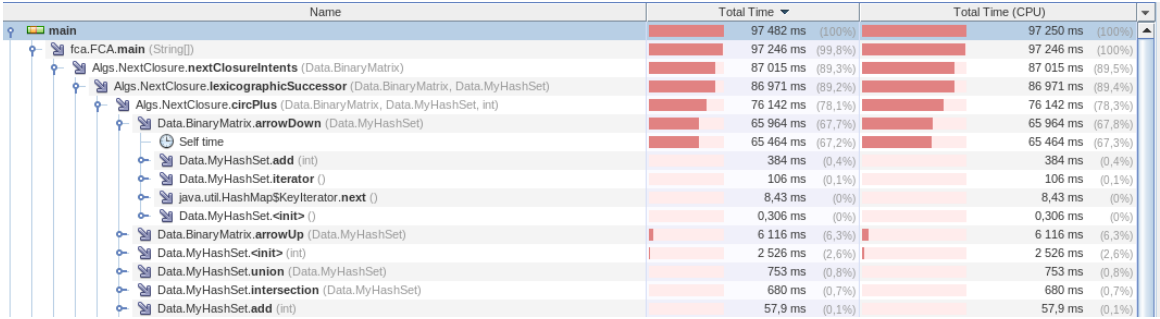
- Hotový systém bychom měli testovat pro ověření potřebných emergentních vlastností
- Testy výkonu – ověření, že systém zvládne očekávanou zátěž
- Testujeme nad daty reálné velikosti a „tvaru“
- Snaha odhalit *úzká hrdla (bottleneck)* systému
  - $\Rightarrow$ návrh na efektivnější implementaci
- *Zátěžové testy (stress tests)* – snažíme se systém plně vytížit
  - Nároky na něj postupně zvyšujeme
  - Sledujeme, jak se systém chová, jak reaguje na zahlcení
  - Při zahlcení může např. dojít k poškození dat, nechtěným stavům, souběhu
- Lepší představa o „trvanlivosti“ systému do budoucna

Intermezzo: Profiler

- Silný nástroj pro ladění výkonu aplikace
- Idea: *Spustíme program s testovacími daty a budeme sledovat v kterých částech strávíme nejvíce času*
  - Případně zaznamenání a kontrola využité paměti
- Dva základní režimy:
  - *Vzorkování* – V pevných intervalech kontrolujeme „kde zrovna jsme“
    - Méně přesná, rychlejší
  - *Instrumentace* – aplikace se krokuje po jednotlivých instrukcích
    - Přesná analýza
    - Velmi pomalá (řádově)
- Zaznamenávání telemetrie – využití CPU, paměti, práce GarbageCollectoru

Intermezzo: Profiler vzorkování

- K dispozici máme pouze časové údaje a poměry

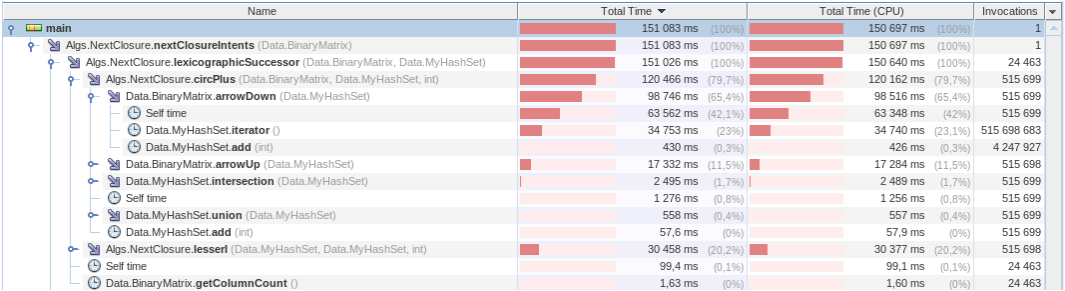


Name	Total Time ▼		Total Time (CPU)	
main	97 482 ms	(100%)	97 250 ms	(100%)
fca.FCA.**main** (String[])	97 246 ms	(99,8%)	97 246 ms	(100%)
Algs.NextClosure.**nextClosureIntents** (Data.BinaryMatrix)	87 015 ms	(89,3%)	87 015 ms	(89,5%)
Algs.NextClosure.**lexicographicSuccessor** (Data.BinaryMatrix, Data.MyHashSet)	86 971 ms	(89,2%)	86 971 ms	(89,4%)
Algs.NextClosure.**circPlus** (Data.BinaryMatrix, Data.MyHashSet, int)	76 142 ms	(78,1%)	76 142 ms	(78,3%)
Data.BinaryMatrix.**arrowDown** (Data.MyHashSet)	65 964 ms	(67,7%)	65 964 ms	(67,8%)
Self time	65 464 ms	(67,2%)	65 464 ms	(67,3%)
Data.MyHashSet.**add** (int)	384 ms	(0,4%)	384 ms	(0,4%)
Data.MyHashSet.**iterator** ()	106 ms	(0,1%)	106 ms	(0,1%)
java.util.HashMap$KeyIterator.**next** ()	8,43 ms	(0%)	8,43 ms	(0%)
Data.MyHashSet.**<init>** ()	0,306 ms	(0%)	0,306 ms	(0%)
Data.BinaryMatrix.**arrowUp** (Data.MyHashSet)	6 116 ms	(6,3%)	6 116 ms	(6,3%)
Data.MyHashSet.**<init>** (int)	2 526 ms	(2,6%)	2 526 ms	(2,6%)
Data.MyHashSet.**union** (Data.MyHashSet)	753 ms	(0,8%)	753 ms	(0,8%)
Data.MyHashSet.**intersection** (Data.MyHashSet)	680 ms	(0,7%)	680 ms	(0,7%)
Data.MyHashSet.**add** (int)	57,9 ms	(0,1%)	57,9 ms	(0,1%)

Obrázek: Profiler se vzorkováním v Netbeans

Intermezzo: Profiler instrumentace

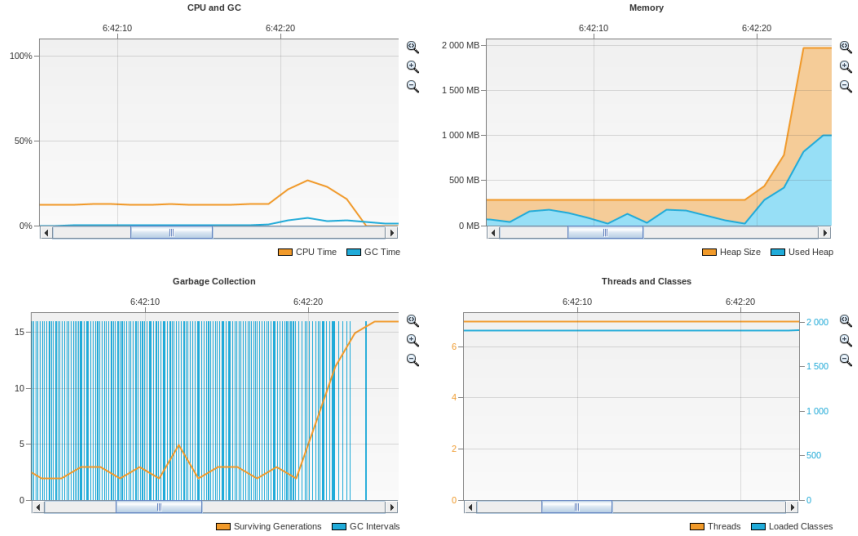
- Zvolíme podmnožinu kódu
- K dispozici máme absolutní čísla, trvá déle (vs. zobrazený čas)



Name	Total Time ▼		Total Time (CPU)		Invocations
main	151 083 ms	(100%)	150 697 ms	(100%)	1
Algs.NextClosure.**nextClosureIntents** (Data.BinaryMatrix)	151 083 ms	(100%)	150 697 ms	(100%)	1
Algs.NextClosure.**lexicographicSuccessor** (Data.BinaryMatrix, Data.MyHashSet)	151 026 ms	(100%)	150 640 ms	(100%)	24 463
Algs.NextClosure.**circPlus** (Data.BinaryMatrix, Data.MyHashSet, int)	120 466 ms	(79,7%)	120 162 ms	(79,7%)	515 699
Data.BinaryMatrix.**arrowDown** (Data.MyHashSet)	98 746 ms	(65,4%)	98 516 ms	(65,4%)	515 699
Self time	63 562 ms	(42,1%)	63 348 ms	(42%)	515 699
Data.MyHashSet.**iterator** ()	34 753 ms	(23%)	34 740 ms	(23,1%)	515 698 683
Data.MyHashSet.**add** (int)	430 ms	(0,3%)	426 ms	(0,3%)	4 247 927
Data.BinaryMatrix.**arrowUp** (Data.MyHashSet)	17 332 ms	(11,5%)	17 284 ms	(11,5%)	515 698
Data.MyHashSet.**intersection** (Data.MyHashSet)	2 495 ms	(1,7%)	2 489 ms	(1,7%)	515 699
Self time	1 276 ms	(0,8%)	1 256 ms	(0,8%)	515 699
Data.MyHashSet.**union** (Data.MyHashSet)	558 ms	(0,4%)	557 ms	(0,4%)	515 699
Data.MyHashSet.**add** (int)	57,6 ms	(0%)	57,9 ms	(0%)	515 699
Algs.NextClosure.**lesserI** (Data.MyHashSet, Data.MyHashSet, int)	30 458 ms	(20,2%)	30 377 ms	(20,2%)	515 698
Self time	99,4 ms	(0,1%)	99,1 ms	(0,1%)	24 463
Data.BinaryMatrix.**getColumnCount** ()	1,63 ms	(0%)	1,60 ms	(0%)	24 463

Obrázek: Profiler s instrumentací v Netbeans

Intermezzo: Profiler telemetrie



Obrázek: Profiler s telemetrií v Netbeans

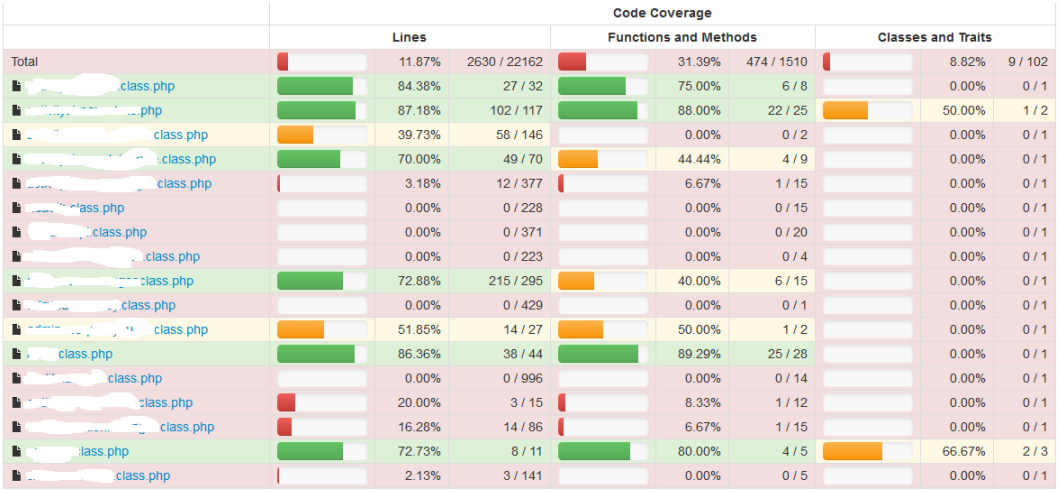
Testy komponent – jednotkové testy

- Snaha o odhalení chyb v jednotlivých komponentách
  - **Bez** (přímé) závislosti na jiných
  - Fake data, falešná implementace rozhraní, API
- Většinou provádí sami programátoři (autoři) komponent
- Testy jednotlivých funkcí/metod objektů (izolovaně)
- Testy celých tříd (průchod více stavy objektu)
- Testy složených (nikoliv složitých) komponent
- Pro většinu jazyků/frameworků máme k dispozici nástroje pro
  - Specifikace testovacích dat
  - Řízení průběhu testů
  - Definici očekávaných výstupů
- Nevýhoda: Píšeme kód pro testování kódu. Mohou být chyby v testech

Code Coverage

- Nepřímá metrika kvality jednotkových testů
- Kolik procent řádků kódu v daném úseku je pokryto jednotkovými testy?
- Pokryto == při testování se daný řádek reálně vykoná (instrumentace)
- Můžeme snížit „přesnosť“ code coverage z řádků na pokryté funkce
- Pokrytí podmínek (a jejich větví)
- Většina vývojových prostředí doprovází grafickou reprezentací
  - Prudce návykové, až hravé
- I ve 100% pokrytém kódu může být spousta chyba (špatné testy)

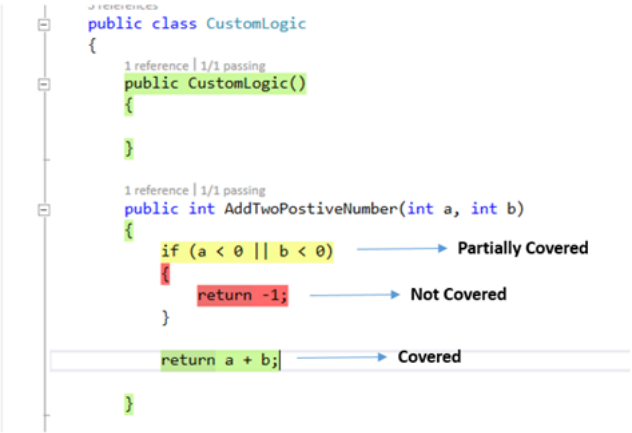
Code Coverage – ukázka



	Code Coverage							
	Lines			Functions and Methods			Classes and Traits	
Total		11.87%	2630 / 22162		31.39%	474 / 1510		8.82%	9 / 102
[redacted]class.php		84.38%	27 / 32		75.00%	6 / 8		0.00%	0 / 1
[redacted]php		87.18%	102 / 117		88.00%	22 / 25		50.00%	1 / 2
[redacted]class.php		39.73%	58 / 146		0.00%	0 / 2		0.00%	0 / 1
[redacted].class.php		70.00%	49 / 70		44.44%	4 / 9		0.00%	0 / 1
[redacted]class.php		3.18%	12 / 377		6.67%	1 / 15		0.00%	0 / 1
[redacted]lass.php		0.00%	0 / 228		0.00%	0 / 15		0.00%	0 / 1
[redacted].class.php		0.00%	0 / 371		0.00%	0 / 20		0.00%	0 / 1
[redacted].class.php		0.00%	0 / 223		0.00%	0 / 4		0.00%	0 / 1
[redacted]class.php		72.88%	215 / 295		40.00%	6 / 15		0.00%	0 / 1
[redacted].class.php		0.00%	0 / 429		0.00%	0 / 1		0.00%	0 / 1
[redacted]class.php		51.85%	14 / 27		50.00%	1 / 2		0.00%	0 / 1
[redacted]class.php		86.36%	38 / 44		89.29%	25 / 28		0.00%	0 / 1
[redacted]class.php		0.00%	0 / 996		0.00%	0 / 14		0.00%	0 / 1
[redacted]class.php		20.00%	3 / 15		8.33%	1 / 12		0.00%	0 / 1
[redacted]class.php		16.28%	14 / 86		6.67%	1 / 15		0.00%	0 / 1
[redacted]lass.php		72.73%	8 / 11		80.00%	4 / 5		66.67%	2 / 3
[redacted]class.php		2.13%	3 / 141		0.00%	0 / 5		0.00%	0 / 1

Obrázek: zdroj – `https://www.filecloud.com/blog/2015/01/how-to-perform-code-coverage-in-netbeans-ide-for-php-projects/`

Code Coverage – ukázka



Obrázek: zdroj – `https://dailydotnettips.com/how-to-customize-color-settings-for-code-coverage-result-in-visual-studio`

Návrh testovacích případů

- Testy na základě požadavků – každý (správný) požadavek by měl být testovatelný
  - Z každého požadavku vytvoříme testovací(více) případy
  - Ověříme, zda výsledky testů odpovídají očekávání
- Rozložení vstupních dat na třídy rozkladu
  - Pro každý prvek z dané třídy by se software měl chovat stejně
  - Testování *typických prvků*, hraničních
- Strukturální testování – na základě znalosti kódu navrhneme testy
  - Mezní případy podmínek
- Ověření všech cest (*path testing*) – vytvoření testu pro všechny možné větve programu
  - Nejlépe všechny kombinace
  - Pokrytí všech větví diagramu aktivit

Test Driven Development

- Hlavní idea: *Testy píšeme dříve než implementaci samotnou*
- Testy tedy neslouží pouze k odhalení chyb v programu
  - $\Rightarrow$ na testech lépe porozumíme požadavkům
  - Výrazná změna ve stylu vývoje
  - Oblíbené v agilních technikách
- Životní cyklus:
  - Napíšeme testy na zamýšlenou funkcionalitu
  - Napíšeme kód, který splní všechny testy
  - Refactoring – co největší zjednodušení
  - `GOTO start`
- Ze psaní testů se poučíme/zdokonalíme:
  - Vyjasníme si akceptační kritéria
  - Komponenty budeme mít méně závislé (samostatně testovatelné)
- Spouštění testů detekuje chyby v implementaci, po splnění víme, že máme hotovo

Testy bezpečnosti

- Jakým způsobem testovat bezpečnost aplikace?
- Co nás vlastně zajímá? Bezpečnost uložení dat? Kdo má nárok která data zobrazovat?
- Zabezpečení přenosu? Co vše může z aplikace uniknout?
- Statické metody mohou odhalit nebezpečná místa v kódu
  - Nebezpečné vzory v kódu (práce s ukazateli)
  - Fuzz testování
- Inspekce/revize kódu zaměřené na bezpečnost
- Penetrační testy (zkoušení různých útoků, externí tým specialistů, interně, . . .)
- Bug-bounty programy

Doporučená literatura

- Ian Sommerville. Software Engineering (10th Edition). Pearson, 2015. ISBN: 978-93-325-8269-9.
  - Chapter 8 – Software Testing
- Steve McConnel. Dokonalý kód. Computer Press, 2005. ISBN: 80-251-0849-X
  - Kapitola 22 – Vývojářské testování
- Steve Freeman, Nat Pryce. Growing Object-Oriented Software, Guided by Tests. Addison-Wesley 2010. ISBN: 0-321-50362-7, 978-0-321-50362-6