Debugging, bugtracking a profiling

Michal Wiglasz

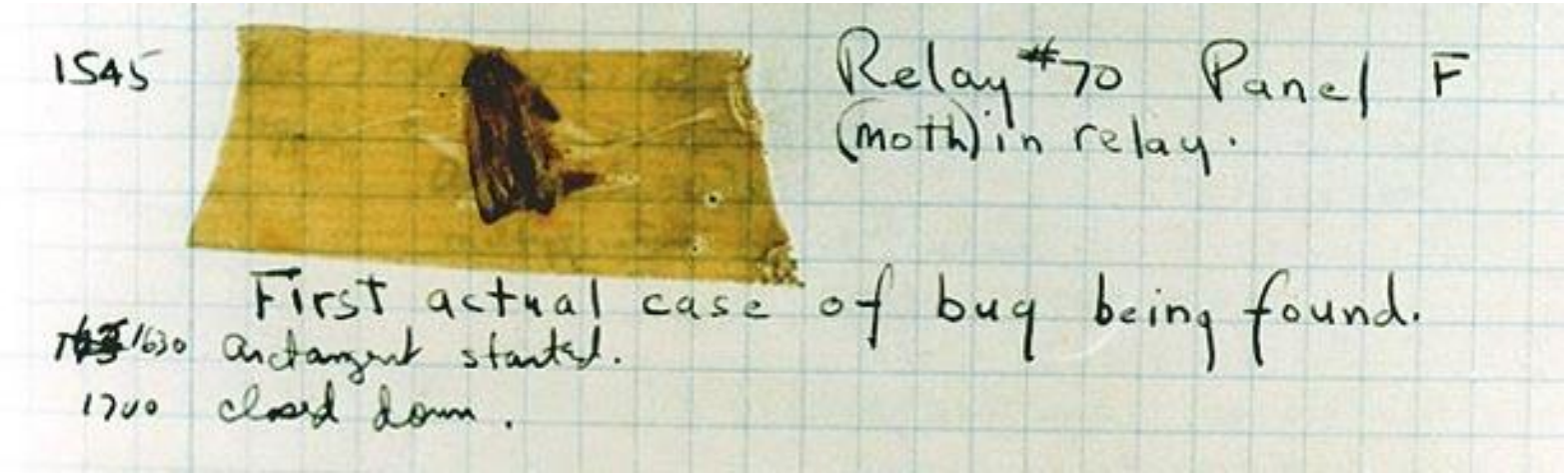
Brno University of Technology, Faculty of Information Technology
Božetěchova 1/2, 612 66 Brno - Královo Pole
iwiglasz@fit.vutbr.cz

27. 3. 2018
Praktické aspekty vývoje software (IVS)

Původ slova **bug?**

Thomas Edison v dopise z roku 1878.

Historka o molu v relé počítače Mark II v roce 1947.



Mariner 1 (1962)

- Chyba při přepisu vzorce
- Z původního $\overline{\dot{R}_n}$ se během přepisu stalo $\dot{R}_n$
- Čára znamená „vyhlazená data“, tzn. nevýznamné odchylky by neměly být brány v potaz

Mars Climate Orbiter (1999)

- Část software pracovala v imperiálních jednotkách, jiná část v metrických → chyba při výpočtu trajektorie

Protiraketové střely Patriot (25. 2. 1991)

- 28 mrtvých, 98 zraněných
- nepřesnost systémových hodin
- po 100 hod. běhu 0,34 s
- chyba půl kilometru při lokaci nepřátelských raket

USS Yorktown CG-48 (1997)

- Smart Ship, síť 27 počítačů
- Po zadání nuly na nesprávné místo došlo k dělení nulou, přetečení bufferu a téměř tříhodinový výpadek

Therac-25 (80. léta)

- Minimálně 5 mrtvých (údajně až 22)
- Obsluha omylem zvolila použití vysokoenergetického paprsku, po opravě na nízkoenergetický se ale přesto spustil vysokoenergetický paprsek
- Chybějící code review
- První testy proběhly až v nemocnici („na produkci“)
- Systém poznal, že něco nesedí a zobrazil hlášku MALFUNCTION 1-64, ale v manuálu nebylo popsáno, co se stalo a stiskem tlačítka šlo chybu přeskočit
- Chybějící hardwarové pojistky
- Race condition, pokud operátor byl moc rychlý

Y2K

- a čeká nás problém roku 2038

Boeing 787 Dreamliner

- Po 248 dnech přejdou elektrické generátory do nouzového režimu (2015)
- Po 22 dnech se bez varování restartují všechny tři řídicí moduly (2016)

Chyby jsou nevyhnutelné.

- vytvořit bezchybný kód je téměř nemožné
- pro většinu aplikací není nutná absolutní bezchybnost
- ale jsou výjimky: zdravotnictví, vojenství, aerospace

Péče věnovaná testování by měla být úměrná náročnosti, důležitosti a nebezpečnosti zakázky.

Metodologie: extrémní programování (XP), programování řízené testy (TDD), lean development, Crystal, Adaptive Software Development, ...

Možné chování chybného programu

Pád programu

- i laik pozná, že je něco špatně

Nekonečný cyklus či rekurze

- nemusí být jasné, jestli program něco dělá

Chyby ve výsledných hodnotách

- program zdánlivě funguje správně

Druhy chyb

Syntaktické chyby

- prohřešky proti gramatice jazyka
- program nelze přeložit
- u interpretovaných jazyků se odhalí až za běhu
- chytrý editor je označí už během psaní

Sémantické chyby

- program nedělá, co má
- obtížná autodetekce

Syntaktické:

- chybějící středník či závorka (v Pythonu odsazení)
- překlep v názvu proměnné, funkce, …

Sémantické:

- chybný přístup do paměti
- dělení nulou
- chyba o jedničku
- přetečení
- nekonečný cyklus
- chyby v synchronizaci vláken

Jiný pohled na typy chyb

- Textové = překlepy
- Pády aplikace → debugger / valgrind
  - segmentation fault, buffer overflow…
- Úniky paměti (memory leak) → valgrind
- Problémy s výkonem → profiler
- Neočekávané chování
  - program dělá něco jiného, než by měl
  - chyba v programu nebo dokumentaci?
- Bezpečnostní problémy
  - např. neošetřené vstupy (SQL injection, CSRF, XSS, …)

DEBUGGING

Postup:

1. Nalezení chyby (vývojář, tester, uživatel)
2. Reprodukce chyby
3. Zjednodušení problému na to důležité
4. Vytipování možných příčin
5. Zaměření se na pravděpodobné příčiny
6. Identifikace příčin
7. Oprava chyby
8. Testování (nejen opravy chyby)
9. Změna dokumentace

Umožňuje:

- krokování programu
- sledování hodnot proměnných (registrů CPU, RAM…)
- změna hodnot proměnných
- zastavení programu na určeném místě (breakpoint)
  - nepodmíněně
  - podmíněně
  - při změně hodnoty proměnné (watchpoint)

Při překladu je nutno přidat ladící informace

- jména proměnných, vazba instrukce ↔ řádek kódu…
- u gcc parametr **-g**

GDB – The GNU Project Debugger

- Konzolové rozhraní, standard pro UNIX-like systémy
- Ada, C, C++, Objective-C, Pascal…

DDD – Data Display Debugger

- grafická nadstavba nad konzolovými debuggery (GDB, DBX, WDB, Ladebug…)
- umí zobrazit grafy na základě dat programu

Prakticky každé IDE obsahuje debugger

- třeba i jako nadstavba nad GDB

- Textové rozhraní, velké množství funkcí
- HW i SW debugování
- Podporuje řadu procesorů a jazyků
- Vzdálené ladění (po síti, rs232) - embedded systémy
- Simulátor různých procesorů (bez periferií)
- Breakpointy, watchpointy, krokování (i zpět)
- Podporuje programy s vlákny
- Existují nadstavby (DDD, Eclipse)

http://sourceware.org/gdb/current/onlinedocs/gdb/

https://www.root.cz/clanky/trasovani-a-ladeni-nativnich-aplikaci-v-linuxu-pouziti-gdb-a-jeho-nadstaveb/

```
$ gcc ./segfault.c -o segfault -g
$ gdb ./segfault

GNU gdb (GDB) 7.12
...
Reading symbols from ./segfault...done.
(gdb)
```

```
$ gcc ./segfault.c -o segfault -g
$ gdb ./segfault

GNU gdb (GDB) 7.12
...
Reading symbols from ./segfault...done.
(gdb) run
Starting program: ~/segfault
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7aca301 in __strlen_sse2 () from /lib64/libc.so.6
(gdb)
```

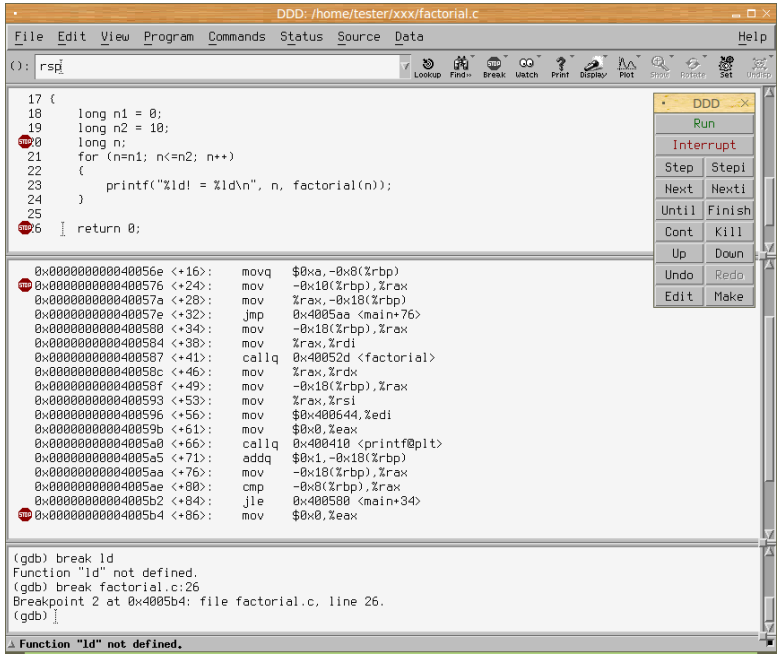
```
$ gcc ./segfault.c -o segfault -g
$ gdb ./segfault

GNU gdb (GDB) 7.12
...
Reading symbols from ./segfault...done.
(gdb) run
Starting program: ~/segfault
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7aca301 in __strlen_sse2 () from /lib64/libc.so.6
(gdb) backtrace
#0  0x00007ffff7aca301 in __strlen_sse2 () from
/lib64/libc.so.6
#1  0x00007ffff7ab199b in puts () from /lib64/libc.so.6
#2  0x0000000000400544 in main () at ./segfault.c:6
```

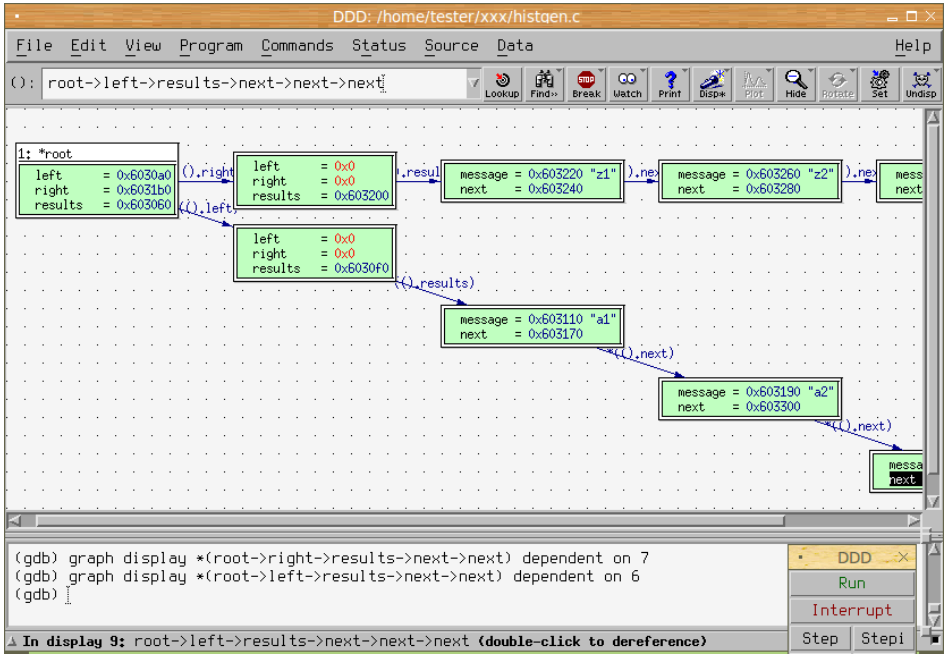
Základní příkazy GDB:

- **help [command]**
- **run** – spuštění programu
- **backtrace** – výpis zásobníku (call stack)
- **step** – postup o jeden krok (step into)
- **next** – krok, ale nevstoupí do cyklů a funkcí (step over)
- **break funkce** – breakpoint při zavolání funkce
- **break main.c:6** – breakpoint na konkrétním řádku
- **break main.c:6 if i>=10** – podmíněný breakpoint
- **watch myvar** – zastavení při změně hodnoty **myvar**
- **print myvar** – výpis hodnoty **myvar**
- **quit**

Funguje doplňování tabulátorem, šipky a zkratky (bt = backtrace)

- grafická nadstavba nad různými debuggery
- vývoj zamrzl (poslední verze v roce 2009)
- zobrazení datových struktur





https://www.gnu.org/software/ddd/manual/

https://mojefedora.cz/debuggery-a-jejich-nadstavby-v-linuxu-2-cast/

Ladění bez debuggeru

- Někdy to jinak nejde (např. embedded systémy)
- Výpisy přes printf
  - výpis „jsem zde“
  - výpis hodnoty proměnných
  - lze použít makra __FILE__, __LINE__ aj.
- Logování
  - chybu lze najít i zpětně („když se nikdo nedíval“)
  - log lze odeslat technické podpoře
  - ale logování může ovlivnit chování programu
    - např. pravděpodobnost vzniku race condition
- Assert
  - ověření platnosti podmínky

Statická analýza kódu

- Hledá problémy v kódu bez jeho spuštění
- Může detekovat více chyb než překladač
- Také jako plugin do editoru či IDE

- Typová kontrola, neinicializovaná data
- Kontrola indexování polí
- Přenositelnost konstrukcí
- Pravidla pro časté chyby
  - `if (confirmation = "yes") format_hdd();`
- Nedodržení stylu formátování
- Vlastní pravidla

Pro C: Lint, cppcheck, mygcc, codan…

Je třeba hledání chyb co nejvíce usnadnit:

- jasné názvy proměnných
- dodržování konvence pojmenování
- přehledné a konzistentní formátování
- dostatečně komentovaný kód
- seskupování kódu do skupin
- neduplikovat kód (Don‘t repeat yourself = DRY)
- nemít příliš hluboké zanoření (if, for, while)
- dodržovat best practices pro daný jazyk
  - Google Style Guide, PEP-8 pro Python, PSR-2 pro PHP…
- mít zapnuté varování překladače

Volby GCC pro ladění

-g

- vytváří ladící informace pro debugger

-ggdb

- rozšíření pro GDB

-Wall

- zapnutí všech varování

-Wextra

- zapnutí dalších varování

-pedantic

- striktně vyžaduje doržování normy (-std=xxx, -ansi)

https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html
https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

- Chyby mohou mít i exotické příčiny
  - chyba v překladači
  - dokonce i chyba v procesoru
- Chyba se nemusí projevit během ladění
  - race condition nevznikne, protože kód běží pomaleji
- Pozor na optimalizace
- Důležité je opravovat příčinu a ne následek

Nejlepší je chybám předcházet.

VALGRIND

Valgrind

- Valgrind = Posvátná brána do Valhally
  - Palác boha Ódina, který sem svolává padlé bojovníky, aby zde trénovali na poslední bitvu Ragnarök
- Původní název byl Heimdall
  - Strážce nordických bohů, který vidí stovky mil daleko ve dne i v noci, slyší růst trávu i vlnu na hřbetech ovcí
  - … ale již existoval balíček s tímto názvem
- Sada nástrojů pro ladění a profilování programů

Valgrind

- Pouze pro unixové systémy
- V podstatě virtuální stroj
  - analyzovaný program je oddělený od procesoru
  - podstatně pomalejší běh programu (4-5krát)
- Umožňuje připojení GDB k běžícímu programu

- Memcheck – správa paměti, nejpoužívanější
- Callgrind – profilování
- Helgrind – detekce race conditions
- Cachegrind – profilování cache CPU

Existují i externě vyvíjené nástroje.

Memcheck detekuje:

- použití neinicializované paměti
- čtení/zápis do uvolněné paměti
- čtení/zápis mimo alokovaný blok
- čtení/zápis nad vrchol zásobníku
- úniky paměti (memory leaks)
- neshody v použití malloc / new vs. free /delete
- špatné použití POSIX knihovny pthreads
- překrytí ukazatelů v memcpy

Čtení z neplatné adresy:

```
Invalid read of size 4
   at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40F6B804: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40B07FF4: read_png_image(QImageIO *)
                                    (kernel/qpngio.cpp:326)
   by 0x40AC751B: QImageIO::read() (kernel/qimage.cpp:3621)
 Address 0xBFFFF0E0 is is 0 bytes after a block of size 40 alloc'd
```

Zároveň se snaží zjistit, kde se neplatná adresa vzala:

- již uvolněná paměť – hlásí, kde se volalo free
- adresa těsně za alokovaným blokem (chyba o jedničku?)

Použití neinicializované paměti:

```
Conditional jump or move depends on uninitialised value(s)
   at 0x402DFA94: _IO_vfprintf (_itoa.h:49)
   by 0x402E8476: _IO_printf (printf.c:36)
   by 0x8048472: main (tests/manuel1.c:8)
```

Nehlásí kopírování neinicializovaných dat, ale až jejich použití, které může mít vliv na funkci programu

Úniky paměti:

```
LEAK SUMMARY:
   definitely lost: 48 bytes in 3 blocks.
   indirectly lost: 32 bytes in 2 blocks.
     possibly lost: 96 bytes in 6 blocks.
   still reachable: 64 bytes in 4 blocks.
        suppressed: 0 bytes in 0 blocks.
```

S parametrem **--leak-check=full** je výpis podrobnější:

```
8 bytes in 1 blocks are definitely lost in loss record 1 of 14
   at 0x........: malloc (vg_replace_malloc.c:...)
   by 0x........: mk (leak-tree.c:11)
   by 0x........: main (leak-tree.c:39)

88 (8 direct, 80 indirect) bytes in 1 blocks are definitely
lost in loss record 13 of 14
   at 0x........: malloc (vg_replace_malloc.c:...)
   by 0x........: mk (leak-tree.c:11)
   by 0x........: main (leak-tree.c:25)
```

- Je dobré chyby opravovat shora dolů
  - Chyby níže ve výpisu mohou být jen důsledkem předchozích
- Některé chyby jsou v systémových knihovnách
  - Není třeba se jimi trápit
  - Valgrind je umí skrýt
- Memcheck není stoprocentní!
  - Například chyba o jedničku v poli alokovaném na zásobníku může přepsat jinou lokální proměnou
  - Ale z pohledu Valgrindu jde o korektní přístup do paměti – neví, že program pracuje s polem

BUG TRACKING

Také *Issue Tracking*, *Ticket System*…

- Aby mohl vývojář vyřešit problém, musí o něm vědět
- Více kódu → více chyb → potřeba lepší evidence

- Hlášení chyb a požadavků
- Přiřazování vývojářům či testerům
- Sledování životního cyklu chyb
- Sledování závislosti chyb
- Lze provázat s verzovacím systémem

Bugzilla, Trac, Redmine, Mantis, JIRA, Team Foundation Server, GitHub, GitLab…

Podle závažnosti:

- blokující
- kritický
- významný
- normální
- méně důležitý
- triviální (drobnosti, chyby v překladu…)
- rozšíření (požadavek na vylepšení)

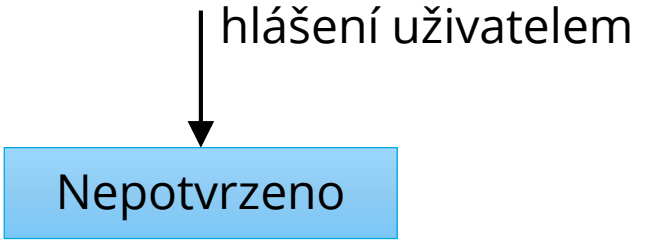
Požadavky jsou zpracovávány dle priority

- dle závažnosti, množství výskytů, …

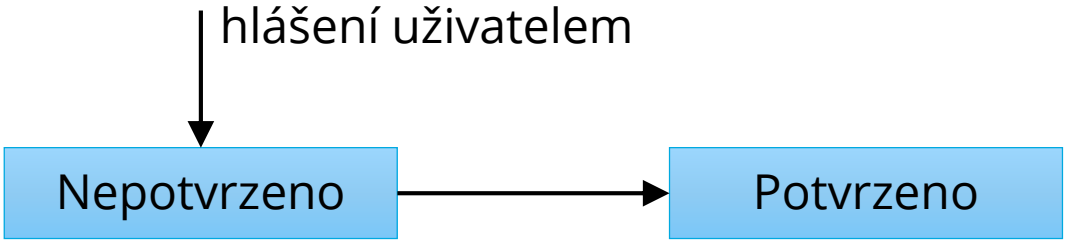
Hlášení chyby (požadavku) by mělo obsahovat maximum relevantních informací:

- jméno požadavku
- stručný popis
- verze SW
- verze operačního systému, knihoven, platforma…
- postup, jak chybu reprodukovat
- popis očekávaného chování
- popis skutečného chování
- kontaktní údaje

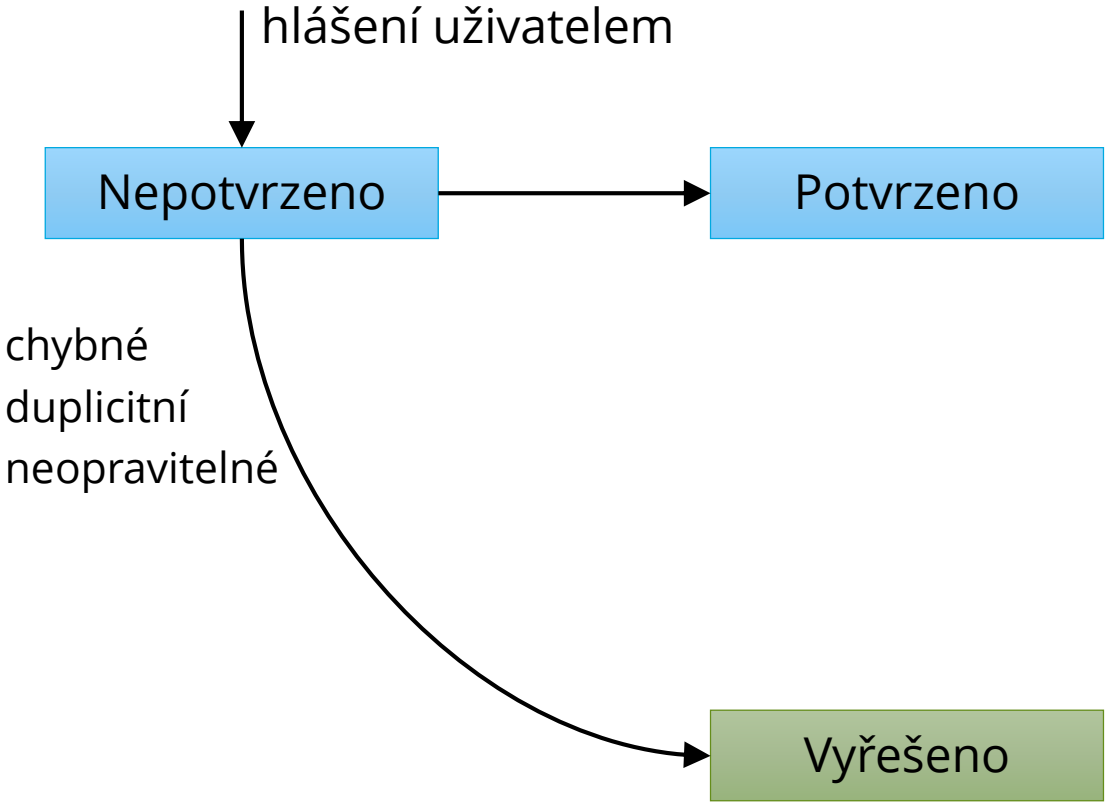
Životní cyklus chyby



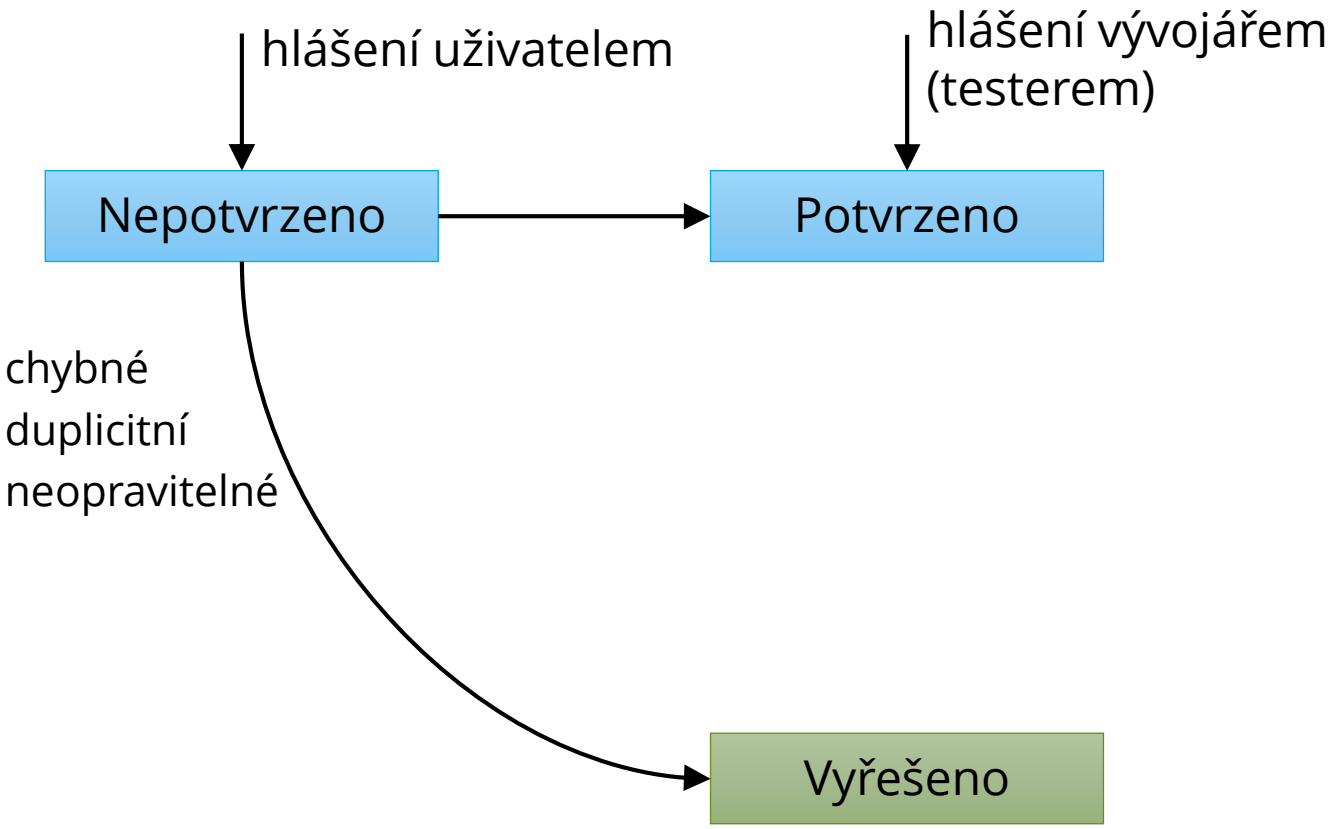
Životní cyklus chyby



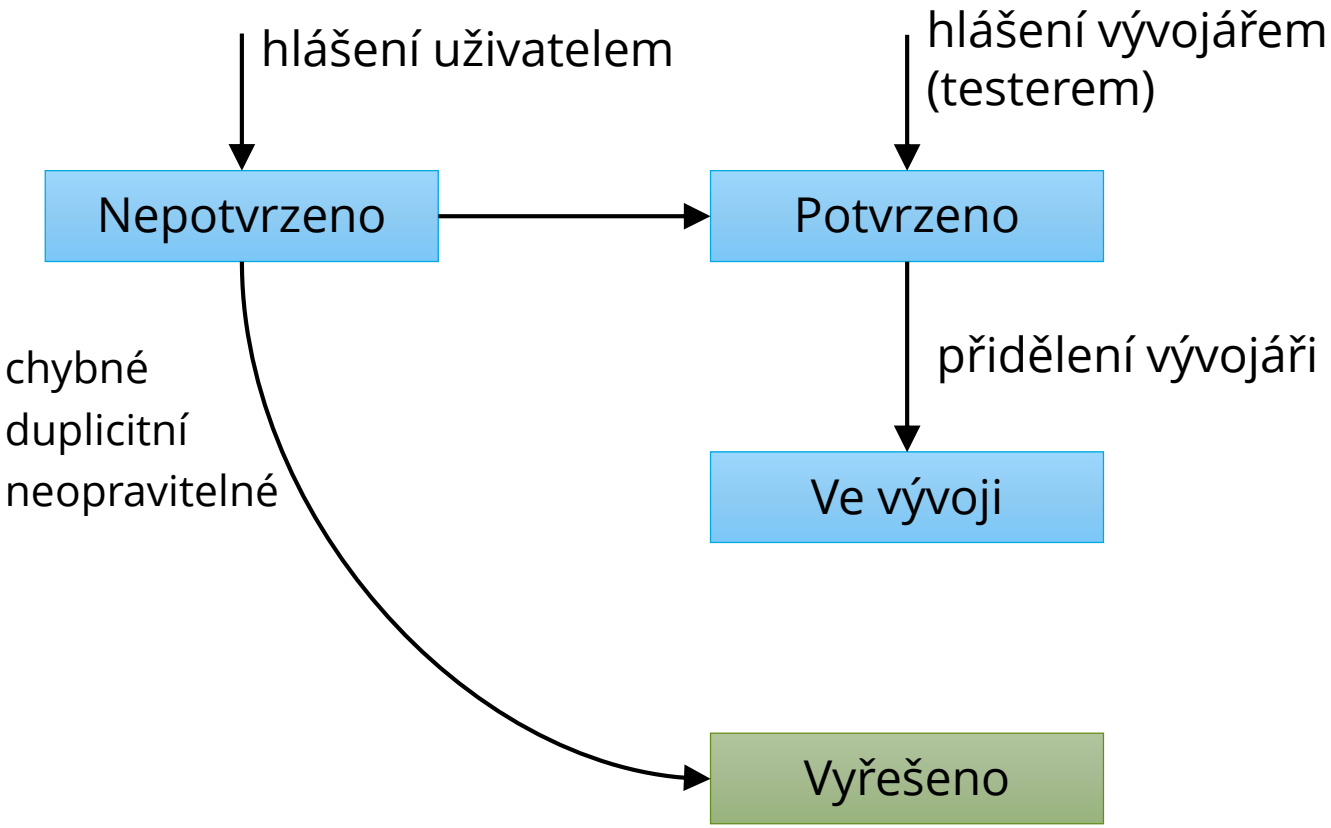
Životní cyklus chyby



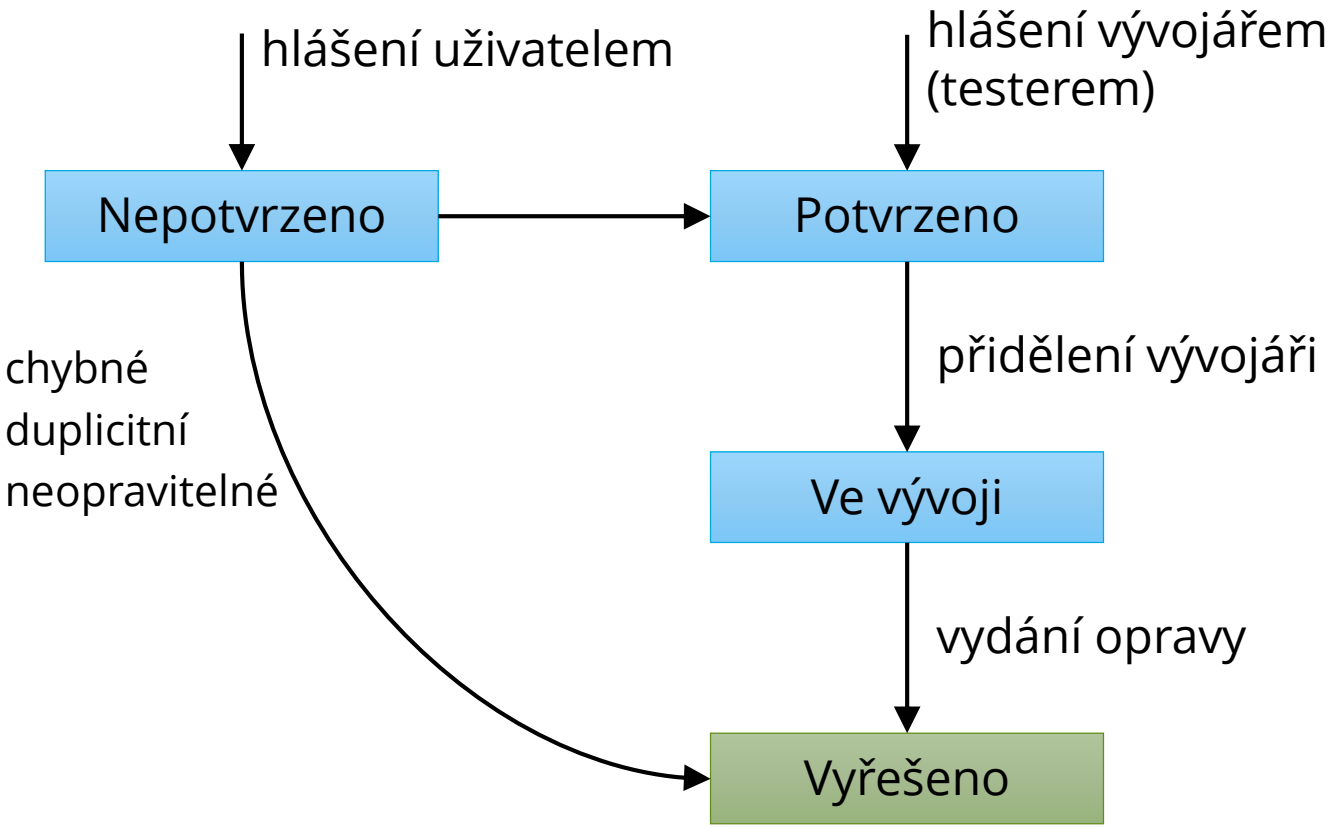
Životní cyklus chyby



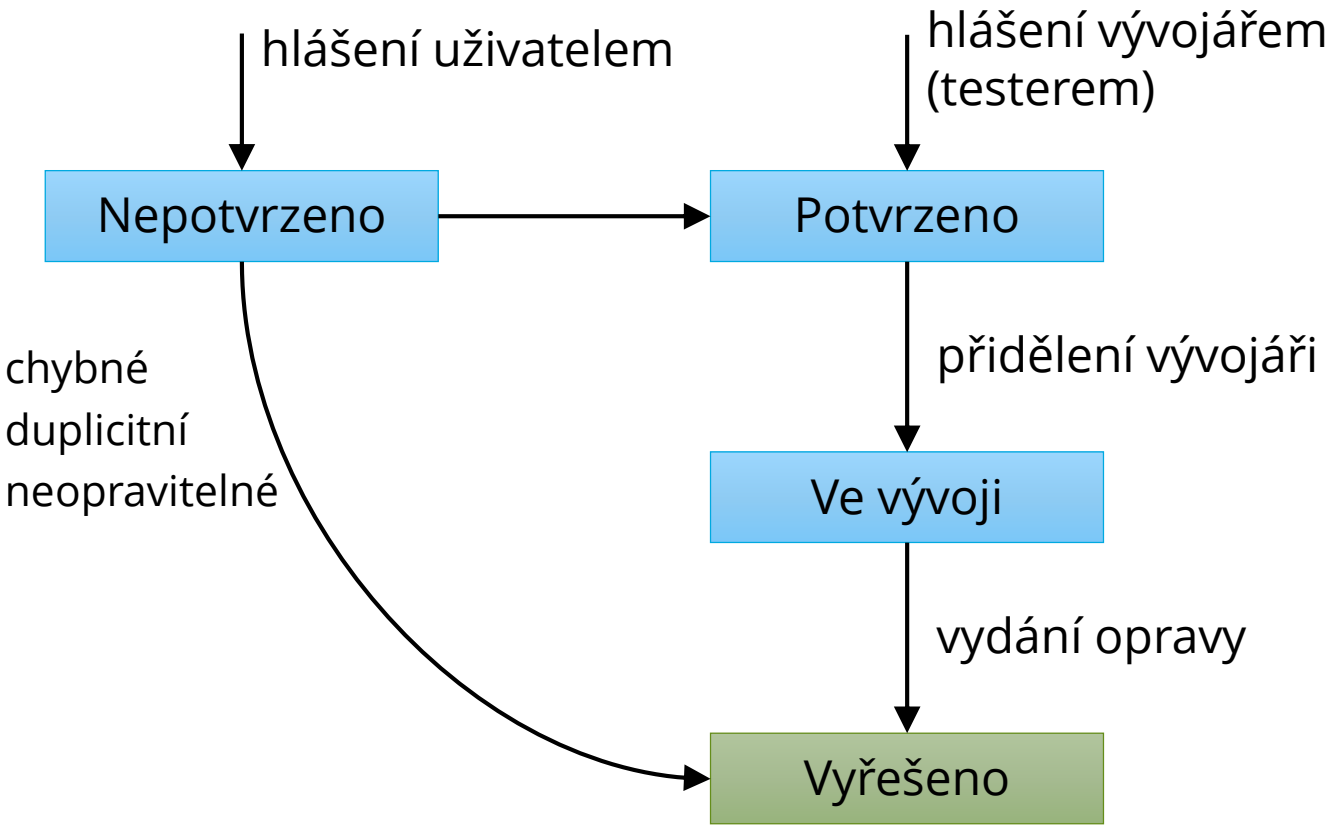
Životní cyklus chyby



Životní cyklus chyby



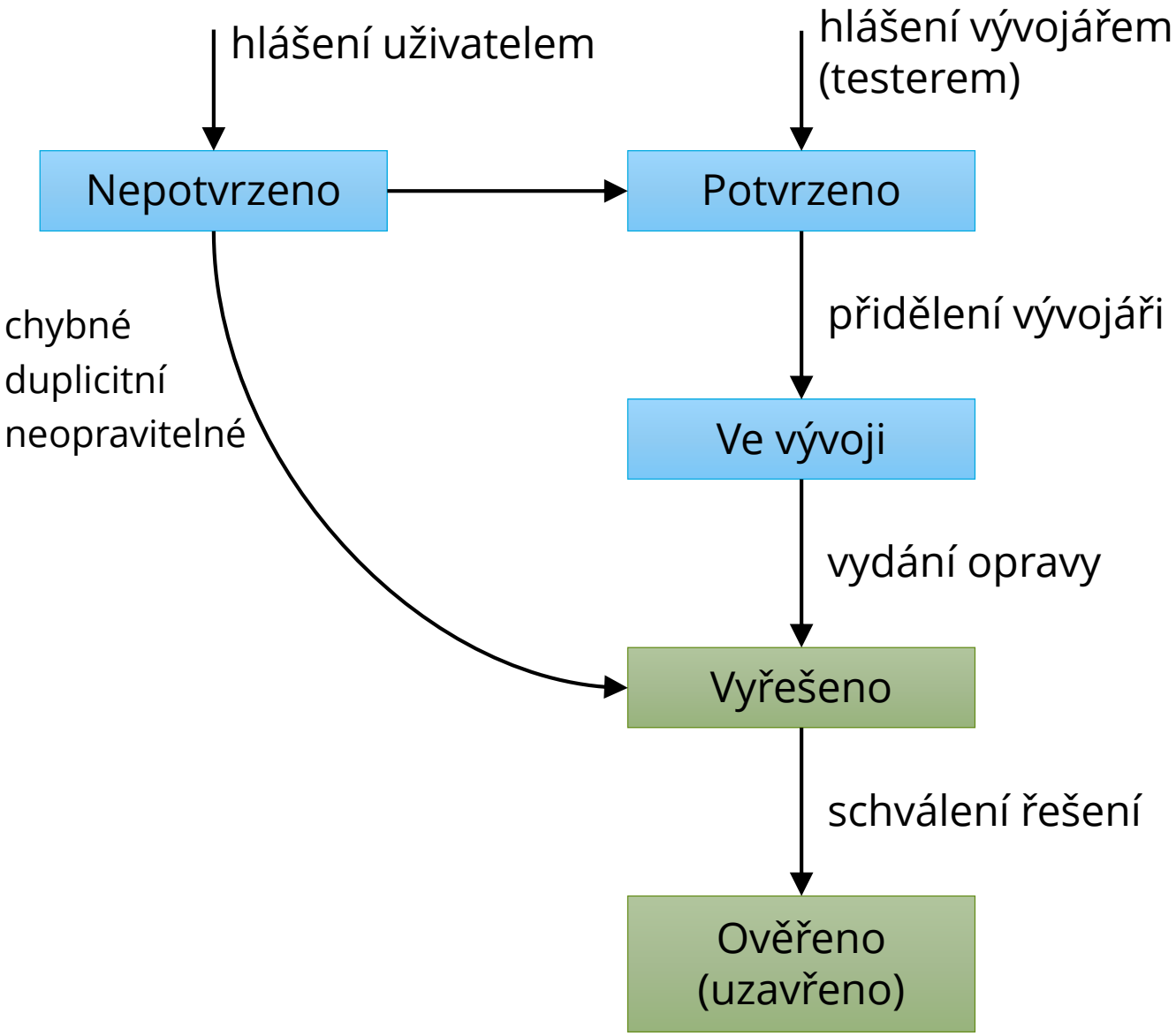
Životní cyklus chyby



Možná řešení:

- chybné
- opraveno
- duplicitní
- neopravitelné (wontfix)
- rozpracováno
- později

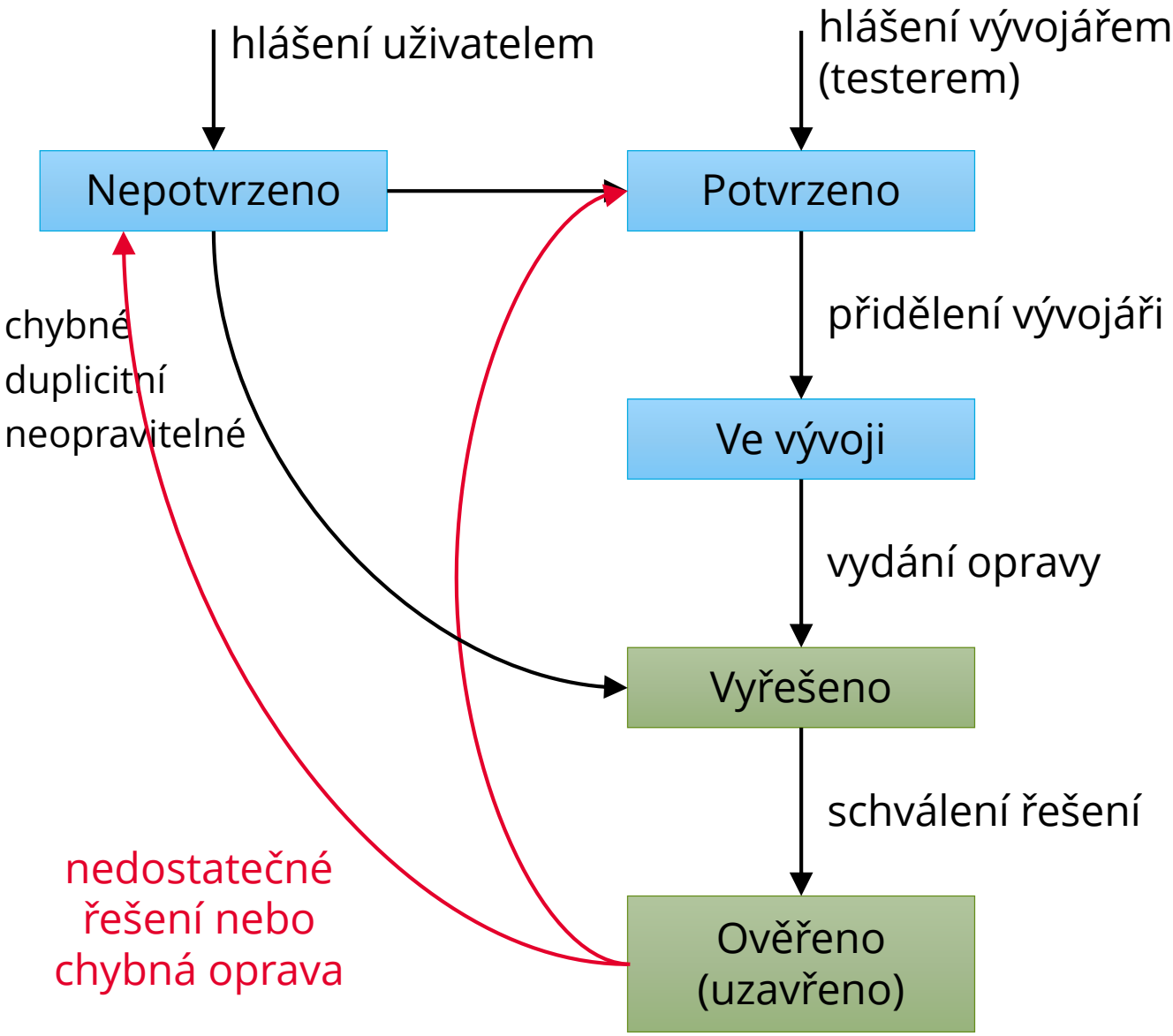
Životní cyklus chyby



Možná řešení:

- chybné
- opraveno
- duplicitní
- neopravitelné (wontfix)
- rozpracováno
- později

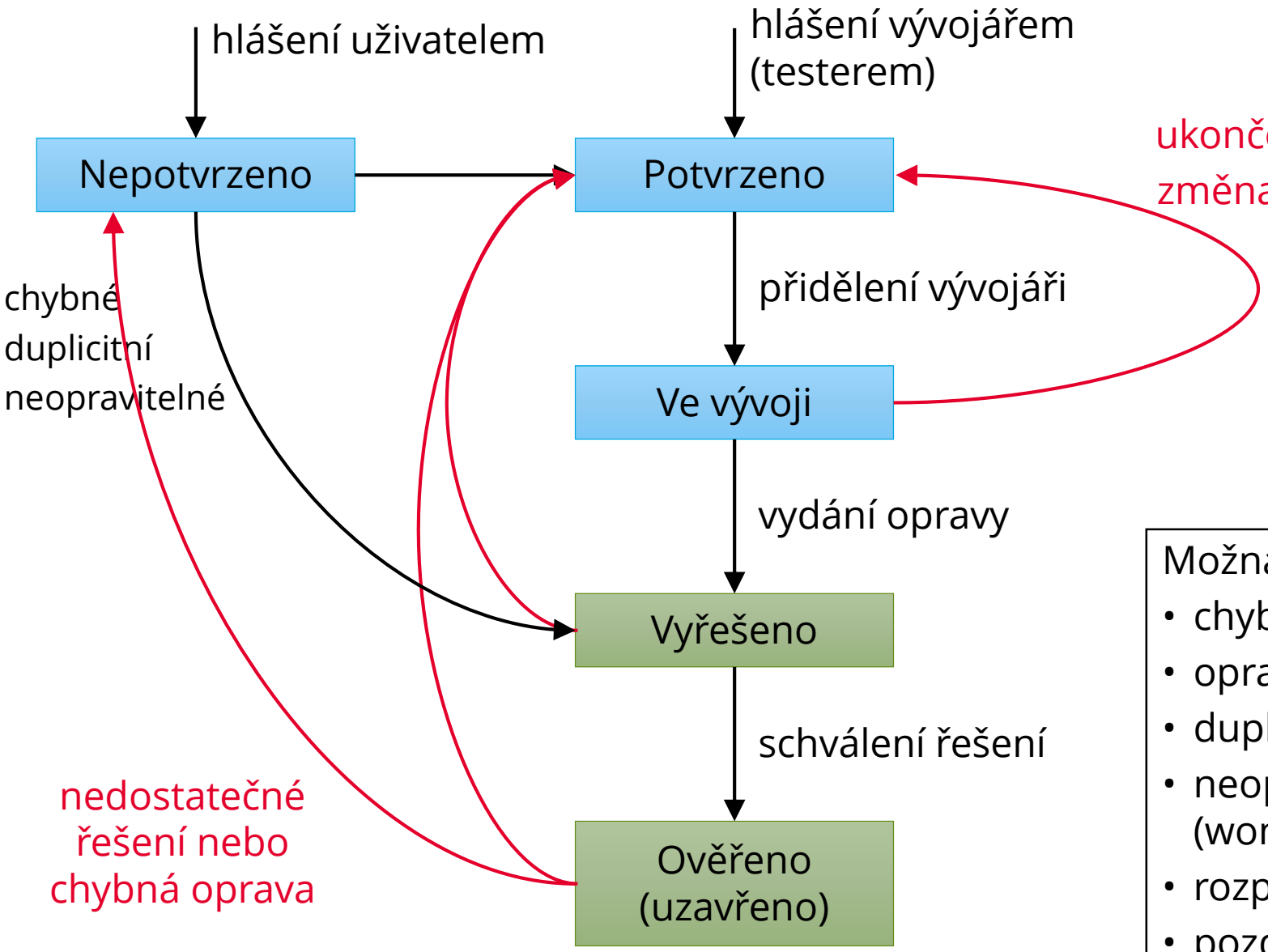
Životní cyklus chyby



Možná řešení:

- chybné
- opraveno
- duplicitní
- neopravitelné (wontfix)
- rozpracováno
- později

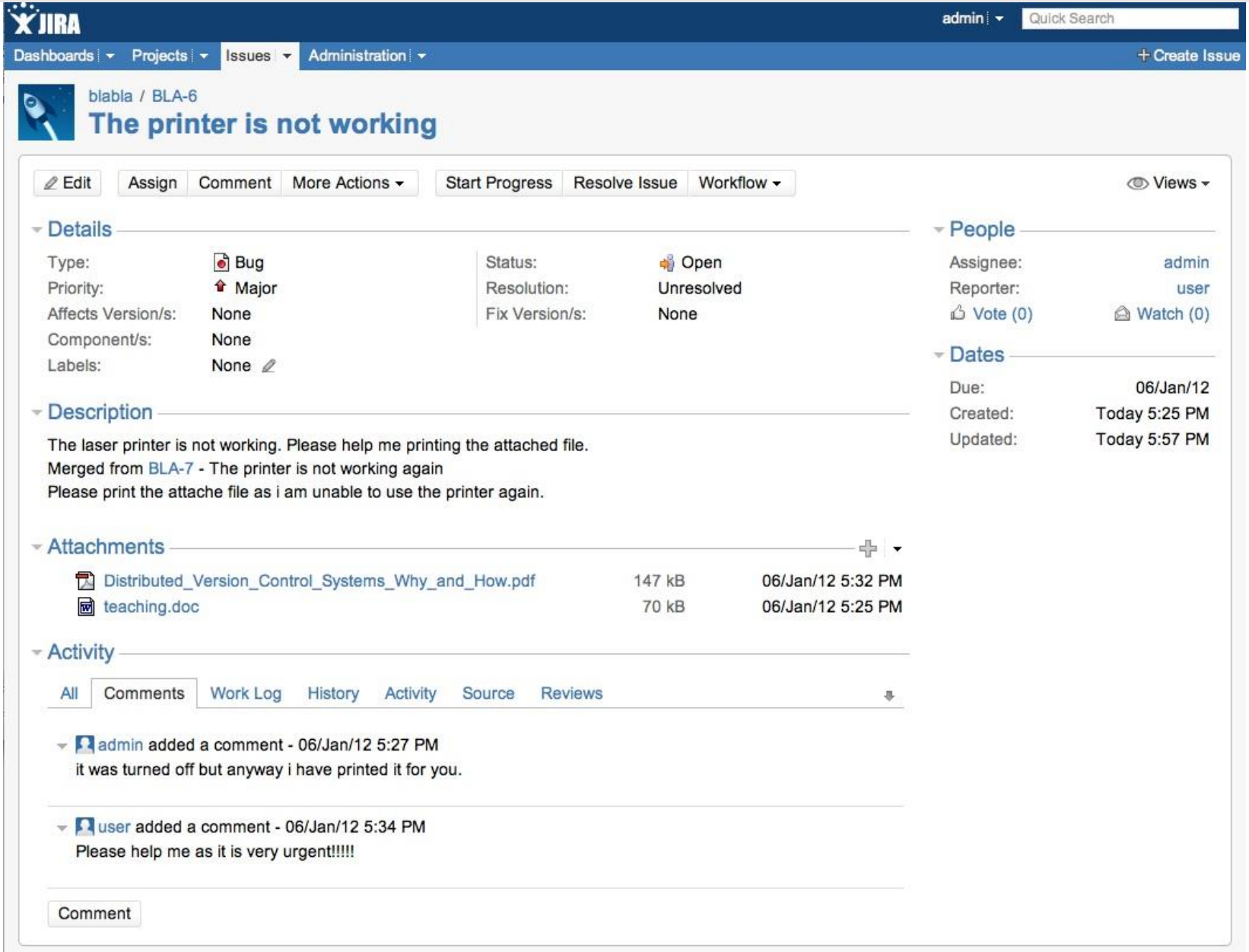
Životní cyklus chyby



Možná řešení:

- chybné
- opraveno
- duplicitní
- neopravitelné (wontfix)
- rozpracováno
- později

Ukázka chyby v bug trackingu (JIRA)



Ukázka chyby v bug trackingu (GitHub)

Redundant Folders of Pictures #3117

New issue

Open **ScottFreeCode** opened this issue on 25 Nov 2017 · 0 comments

ScottFreeCode commented on 25 Nov 2017 — Owner

Mocha's repo has no less than three different folders at the top level for pictures: `assets`, `images` and `media`. Some of these are used in Growl notifications and should live or die with the Growl integration. The rest... actually, I'm not sure they're used *anywhere*. Unless the (*currently* separate) mocha website is somehow pulling in images from this repo, or something.

While this isn't technically harmful, it's needlessly confusing for new contributors (one of whom brought it to my attention by asking when to look in which folder, more or less). I'd like to propose that:

- Any of these that need to be in this repo should get moved into a single folder and organized in subfolders that indicate how they're used or what they're for.
- Any of them that are needed but do not need to be in this repo get moved either to where they're needed (e.g. if they're part of the site put them with the site) or into some dedicated repo or host.
- Any of them that are no longer needed should be removed (if we really want to get them back for some reason we can always dig them out of the commit history).

(NOTE: If some of them are needed for running Mocha, e.g. the Growl ones, and others are only needed e.g. for the readme or the site, this plan *may* require us to be more specific with specifying which files to publish instead of publishing the whole images folder.)

1

ScottFreeCode added chore developer-experience labels on 25 Nov 2017

boneskull added the good-first-issue label on 9 Dec 2017

markowsiak self-assigned this 5 days ago

Write Preview

Leave a comment

Assignees: markowsiak

Labels: chore, developer-experience, good-first-issue

Projects: None yet

Milestone: No milestone

Notifications: Subscribe

You're not receiving notifications from this thread.

3 participants

PROFILING

- Program funguje správně, ale pomalu
- Malé urychlení jedné malé smyčky může způsobit velké urychlení celého programu
- Pravidlo 80/20:

  80 % strojového času se stráví nad 20 % kódu
- Profiler pomáhá identifikovat místa v programu, kde se spotřebuje nejvíce strojového času

Postup:

1. Měření
   - získání údajů o běžícím programu
   - počet vyvolání funkcí, počet iterací cyklů, čas strávený v jednotlivých částech kódu, čekání na I/O…

2. Analýza
   - statistické vyhodnocení naměřených dat
   - vizualizace pomocí tabulek a grafů

3. Optimalizace

Sampling (statistický přístup)

- periodické přerušení, ve kterém se zaznamená aktuální poloha v programu
- nepřesné, ale nezpomaluje tolik běh programu
- AMD CodeAnalyst, Apple Shark, Intel Vtune

Instrumentace programu

- do programu se vloží volání speciální funkce
- více ovlivňuje rychlost programu
- některé chyby se nemusí projevit, jiné se naopak začnou projevovat

Výstup profileru

Flat profile

- čas strávený v jednotlivých funkcích
- počet volání

Graf volání (Call graph)

- pro každou funkci: odkud byla volána, jaké funkce volala
- jak dlouho každé volání funkce trvalo

Anotovaný kód (Annotated source)

- Ke každému řádku kódu je přidán počet vykonání

- Kombinuje statistický přístup s instrumentací
- Čas strávený v jednotlivých funkcích pomocí periodického sledování
  - čas běhu programu by měl být výrazně vyšší než perioda vzorkování (0,01 s)
- Počet volání funkcí pomocí instrumentace
- Výstup: flat profile, graf volání

```
$ gcc -Wall -Wextra -O2 -g -pg ./program.c -o program
$ ./program
$ ls
program gmon.out
$ gprof program gmon.out > gprof-report.txt
```

- Soubor gmon.out vznikne v pracovním adresáři
- Každý modul je třeba přeložit pro profilování (-pg)
  - v opačném případě nebude na výstupu gprof

Pro zvýšení přesnosti lze zkombinovat více běhů:

```
gcc –Wall –Wextra –O2 –g -pg ./program.c –o program

# prvni beh
$ ./program
$ mv gmon.out gmon.sum

# opakuj n-krat
$ ./program
$ gprof -s program gmon.out gmon.sum

# souhrnne vysledky
$ gprof program gmon.sum > gprof-report.txt
```

gprof - flat profile

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 31.75      9.90     9.90        1     9.90     9.90  new_func1
 31.62     19.76     9.86        1     9.86     9.86  func2
 22.17     26.68     6.91        1     6.91    16.82  func1
  0.23     26.75     0.07                             main
```

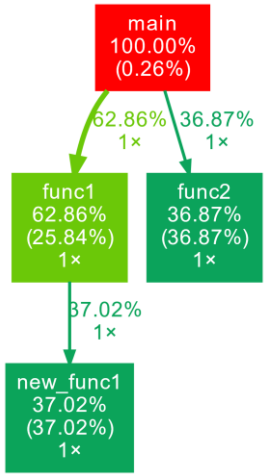
Časy které nejsou mnohem větší než vzorkovací perioda nejsou příliš věrohodné

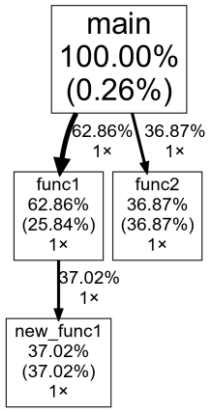
gprof - call graph

```
granularity: each sample hit covers 2 byte(s) for 0.04% of 26.75 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.07   26.68                 main [1]
                6.91    9.90       1/1           func1 [2]
                9.86    0.00       1/1           func2 [4]
-----------------------------------------------
                6.91    9.90       1/1           main [1]
[2]     62.9    6.91    9.90       1         func1 [2]
                9.90    0.00       1/1           new_func1 [3]
-----------------------------------------------
                9.90    0.00       1/1           func1 [2]
[3]     37.0    9.90    0.00       1         new_func1 [3]
-----------------------------------------------
                9.86    0.00       1/1           main [1]
[4]     36.9    9.86    0.00       1         func2 [4]
-----------------------------------------------
```

- https://github.com/jrfonseca/gprof2dot
- Převodník do formátu pro GraphViz
- Umí i jiné vstupní formáty než gprof

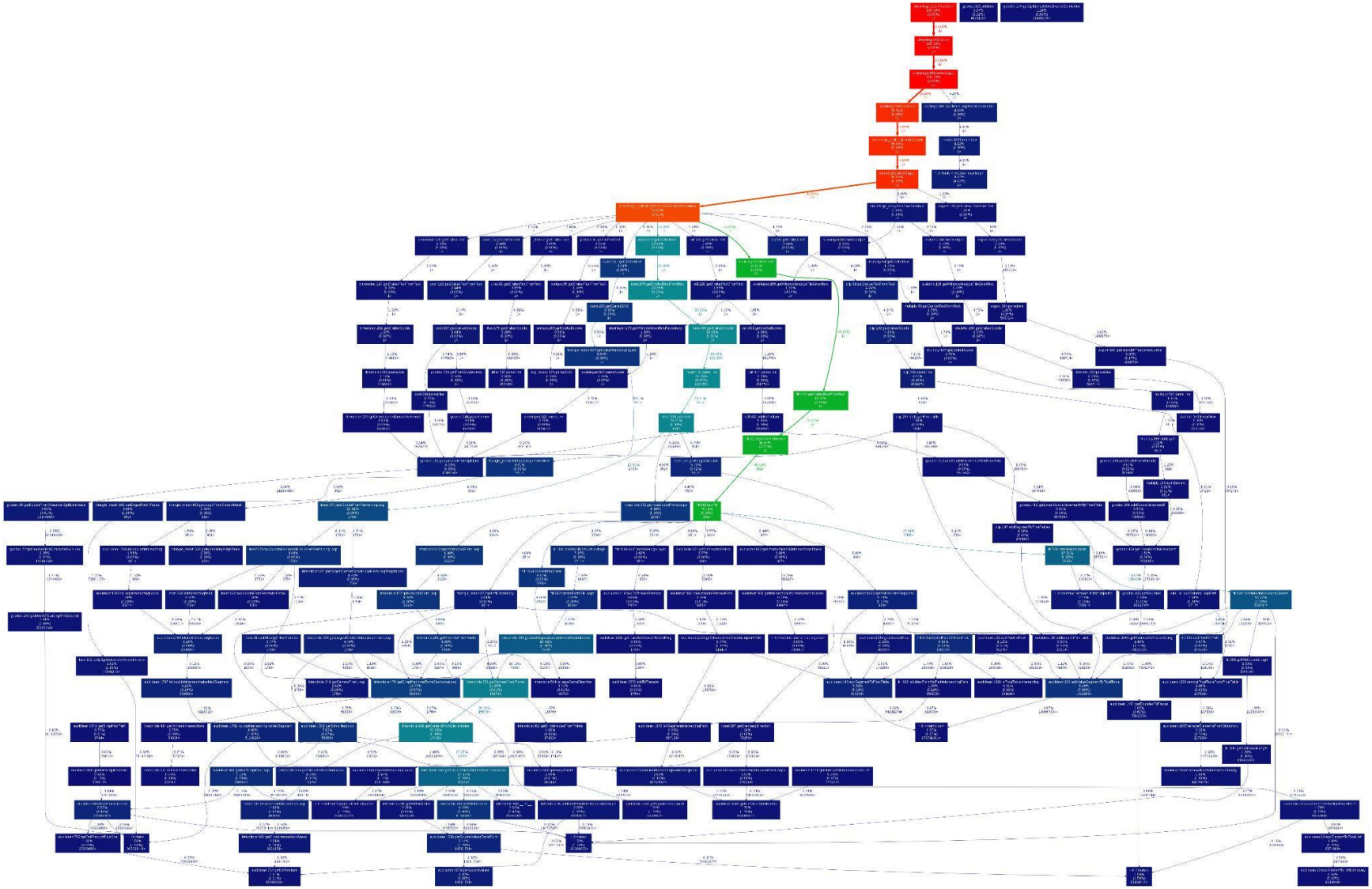
```
$ gprof2dot gprof-report.txt > gprof-report.dot
$ dot -Tpng –ogprof-report.png gprof-report.dot
$ dot -Tsvg –ogprof-report.svg gprof-report.dot
```





`gprof2dot –c print`

Vizualizace – gprof2dot

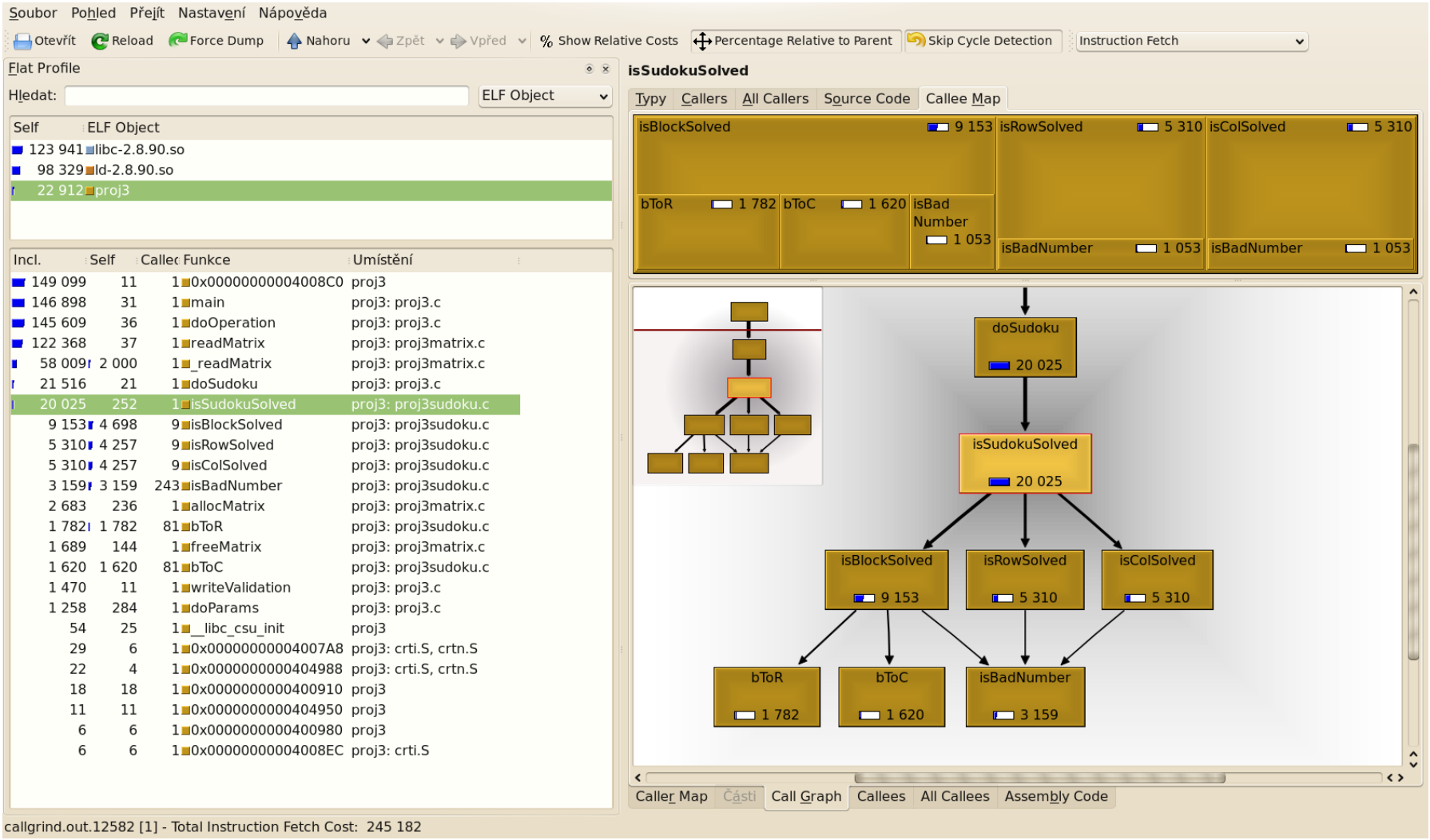


Callgrind

- Součást valgrindu
- Výrazně pomalejší běh programu, ale není třeba speciální překlad
- Analýza např. pomocí KCachegrind

```
$ valgrind --tool=callgrind ./program
$ ls
program program.out.pid

$ ./gprof2dot.py -f callgrind program.out.pid > vis.dot
```

Callgrind - KCachegrind



Profiling

- Profiler napoví, kde se vyplatí optimalizovat
- Je zbytečné optimalizovat funkci, která se zavolá jednou a neběží dlouho
- Výstup profileru může být zatížen statistickou chybou
- Vhodné pro rozsáhlejší programy
- Může pomoci odhalit chyby
  - více/méně volání funkce, než se očekává
- Spuštění s profilerem ovlivňuje běh programu
  - rychlost, heisenbugs
- Optimalizace kódu může zhoršit čitelnost
  - Mnohdy je lepší optimalizovat algoritmus než implementaci

QUALITY ASSURANCE

- Proces zajišťování kvality
- Týká se všech fází vývoje, pro procesy i pro výstup

Kontrola kvality procesů

- ISO 9001, CMM, SPICE
- certifikovat lze i proces výroby nekvalitních produktů

Kontrola kvality výstupu (software)

- code review
- testování
- QA inženýr kontroluje práci vývojářů

Capability Maturity Model (CMM)

Hodnocení vyspělosti procesů v organizaci

1 - Počáteční (Initial)
- Procesy jsou realizovány adhoc

2 - Opakované (Repeatable)
- Dodržuje se určitá kázeň nezbytná pro provádění základních opakovaných procesů

3 - Definovaná (Defined)
- Procesy organizace jsou zdokumentovány

4 - Řízená (Managed)
- Procesy jsou řízeny a provádí se měření jejich výkonnosti pomocí KPI

5 - Optimalizovaná (Optimized)
- Procesy jsou trvale zlepšovány, existuje inovační cyklus na procesech a řízení

iwiglasz@fit.vutbr.cz