Testowanie kodu pod Linuksem na sterydach

Jakub Juszczakiewicz

26 kwietnia 2026

O mnie...

Jakub Juszczakiewicz

członek alumni ASI
absolwent PWr
pracownik etatowy w Enamor Int.

https://juszczakiewicz.net
https://github.com/jakubjuszczakiewicz

Spis rzeczy

Jak można testować kod?

Sprawdzać ręcznie - na bieżąco

- Wymaga żmudnego sprawdzania na wyrywki
- Nie pokrywa większości przypadków
- Ogranicza powtarzalność
- Świadczy o złej organizacji pracy

Pisać testy automatyczne

- Pozwala sprawnie weryfikować pracę - również w przyszłości
- Umożliwia ciągłe zarządzanie pulą przypadków testowych
- Dostarcza powtarzalne metody sprawdzania
- Świadczy o dojrzałości organizacji pracy

Testy jednostkowe

Definicja z wikipedii

Test jednostkowy (ang. *unit test*) - metoda testowania tworzonego oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów (jednostek) programu – np. metod lub obiektów w programowaniu obiektowym lub procedur w programowaniu proceduralnym. (...)

Definicja intuicyjna

Kawałek kodu programu mający na celu uruchomić sprawdzany wycinek kodu (np. pojedynczą funkcję lub metodę) i zweryfikować wynik jej działania.

Jak pisać testy jednostkowe?

Co powinny sprawdzać testy jednostkowe?

- Prawidłowe dane wejściowe (dowolne)
- Warunki brzegowe danych wejściowych
- Błędne dane wejściowe

Do czego nie nadają się testy jednostkowe?

- Testowania wydajności
- Testowania stabilności systemu
- Testowania dużych części systemu

Po co pisać testy jednostkowe?

Kiedy testy jednostkowe się przydają?

- Podczas pisania pierwszej implementacji
- Podczas naprawiania błędów
- Podczas pisania poprawek
- Podczas optymalizacji
- Przy utrzymaniu kodu

Test-driven development

Definicja z wikipedii

Test-driven development (TDD) - technika tworzenia oprogramowania, zaliczana do metodyk zwinnych. Pierwotnie była częścią programowania ekstremalnego (ang. extreme programming), lecz obecnie stanowi samodzielną technikę. Polega na wielokrotnym powtarzaniu kilku kroków:

1. Najpierw programista pisze automatyczny test sprawdzający dodawaną funkcjonalność. Test w tym momencie nie powinien się udać.
2. Później następuje implementacja funkcjonalności. W tym momencie wcześniej napisany test powinien się udać.
3. W ostatnim kroku programista dokonuje refaktoryzacji napisanego kodu, żeby spełniał on oczekiwane standardy.

TDD - utopia?

Dlaczego TDD w oryginalnej formie nie sprawdzi się?

- Wymaga pełnego, spójnego i finalnego projektu całego systemu przed rozpoczęciem pisania kodu
- W początkowej fazie wprowadza zamieszanie - niedziałające testy
- Brak możliwości skutecznej integracji z systemami CI/CD

Naprawmy TDD

Propozycja modyfikacji TDD:

1. Opisać testy i przygotować ich wydmuszki. Opis przyszłych testów można (od biedy) umieścić jako komentarz w kodzie i/lub w dokumentacji
2. Napisać pierwszą implementację kodu
3. Napisać pierwsze testy do istniejącej implementacji
4. Poprawiać implementację i dopisywać testy do ustabilizowania rozwiązania

Naprawiliśmy TDD?

Dlaczego proponowane podejście jest lepsze?

- Nie robi zamieszania w CI/CD
- Jeśli podczas pierwszej implementacji wyjdą problemy - nie trzeba przepisywać testów
- Dalej umożliwia pisanie testów i kodu przez różne osoby (zalecane)
- To podejście lepiej się integruje z życiową dokumentacją projektową

Ułatwianie (?) pisania testów

Przykładowe frameworki do pisania testów jednostkowych w C/C++:

- Check (`https://libcheck.github.io/check/`)
- CUnit (`https://cunit.sourceforge.net/`)
- µnit (`https://nemequ.github.io`)
- CppUnit (`https://cppunit.sourceforge.net`)
- Google Test (`https://github.com/google/googletest`)
- Boost.Test (`https://www.boost.org`)
- ...

Cechy frameworków testowych

Zalety:

- Dostarczają infrastrukturę do testów
- Dostarczają zestaw funkcji porównujących
- Pozwalają na łatwą integrację wyników testów z innymi narzędziami
- Dostarczają jednolity sposób pisania testów

Wady:

- Dodatnie kolejnego testu wymaga narzutu kodu
- Korzystanie z nich jest żmudne
- Utrzymanie dużej liczby testów dla jednej jednostki potrafi być wyzwaniem

Przykład pojedycznego testu

```
TEST(test_logA_zero)
{
  uint128_t input;

  memcpy(&input, &data[ZERO_IDX].val, sizeof(uint128_t));
  ASSERT_EQ(uint128_logA(&input), 1);
}

TEST(test_logA_one)
{
  uint128_t input;

  memcpy(&input, &data[ONE_IDX].val, sizeof(uint128_t));
  ASSERT_EQ(uint128_logA(&input), 1);
}
```

kit-pval

kit-pval:

- prosta aplikacja do „ładnego“ wyświetlania dużych liczb z przedrostkami
- kod dostępny na mojej stronie i githubie
- otwarta licencja (GPL2)
- testy napisane wg „mojego” schematu
- obsługuje 128bitowe liczby - również na 32bitowych platformach
- zgodna z architekturami big endian i little endian

Przykład:

```
$ ./kit-pval -n 1024000 B
0.97MiB
$ ./kit-pval -n 123456789012 B
114.97GiB
```

Inne podejście do testów - cz. 1

```
struct {
  union {
    struct {
#ifdef BIG_ENDIAN
      uint64_t a, b;
#else
      uint64_t b, a;
#endif
    };
    uint128_t val;
  };
} data[] = {
  { .a = 0x0000000000000000LLU, .b = 0x0000000000000000LLU },
  { .a = 0x0000000000000000LLU, .b = 0x00000000000003E7LLU },
  { .a = 0x0000000000000000LLU, .b = 0x000000000000F423FLLU },
  { .a = 0x0000000000000000LLU, .b = 0x0004000000000000LLU },
  { .a = 0xFFFFFFFFFFFFFFFFLLU, .b = 0xFFFFFFFFFFFFFFFFLLU },
};
```

Inne podejście do testów - cz. 2

```
struct {
  size_t idx1;
  const char * output;
  unsigned int prefix_avail;
  unsigned int acomma;
  uint32_t base;
} const tests_01[] =
{
  { 0, "0",                                                    0,  2, 1024 },
  { 1, "0.97ki",                                               2,  2, 1024 },
  { 2, "976.56ki",                                             0,  2, 1024 },
  { 2, "976.56ki",                                             3,  2, 1024 },
  { 2, "976.56ki",                                             7,  2, 1024 },
  { 2, "0.95Mi",                                               4,  2, 1024 },
  { 2, "0.00Gi",                                               8,  2, 1024 },
  { 3, "1.00Pi",                                               0,  2, 1024 },
  { 3, "1125899906842624",                                     1,  2, 1024 },
  { 3, "1099511627776.00ki",                                   2,  2, 1024 },
  { 3, "1125899906842.62k",                                    2,  2, 1000 },
  { 3, "1073741824.00Mi",                                      4,  2, 1024 },
  { 3, "1125899906.84M",                                       4,  2, 1000 },
  { 3, "1.00Pi",                                              32,  2, 1024 },
  { 3, "1.12P",                                               32,  2, 1000 },
  { 3, "1.125P",                                              32,  3, 1000 },
  { 3, "1073741824.000Mi",                                     4,  3, 1024 },
  { 3, "1125899906.842M",                                      4,  3, 1000 },
  { 3, "0.001E",                                              64,  3, 1000 },
  { 3, "0.001E",                                              64,  3, 1000 },
  { 4, "281474976710655.99Yi",                                 0,  2, 1024 },
  { 4, "340282366920938.46Y",                                  0,  2, 1000 },
  { 4, "340282366920938463463374607431768211455",              1,  2, 1024 },
};
const size_t tests_01_len = sizeof(tests_01) / sizeof(tests_01[0]);
```

Inne podejście do testów - cz. 3

```
int test_print_pval(void)
{
  for (size_t i = 0; i < tests_01_len; i++) {
    uint128_t input;
    memcpy(&input, &data[tests_01[i].idx1].val, sizeof(uint128_t));
    print_cfg config;
    config.prefix_avail = tests_01[i].prefix_avail;
    config.base = tests_01[i].base;
    config.acomma = tests_01[i].acomma;
    char output[48];
    memset(output, 0, sizeof(output));

    size_t r = print_pval(output, &input, &config);
    if (r != 0) {
      fprintf(stderr, "Test (01a) %zu failed\n", i + 1);
      fprintf(stderr, "%zu\n0\n", r);
      return 1;
    }
    if (strcmp(output, tests_01[i].output) != 0) {
      fprintf(stderr, "Test (01b) %zu failed\n", i + 1);
      fprintf(stderr, "%s\n%s\n", tests_01[i].output, output);
      return 1;
    }
  }

  return 0;
}
```

Zalety i wady proponowanego podejścia

Wady:

- Brak łatwej integracji z narzędziami zewnętrznymi (np. do raportów)
- W niektórych przypadkach wynajduje się koło na nowo - np. porównywanie typów zmiennoprzecinkowych

Zalety:

- Łatwość dodawania kolejnych przypadków testowych
- Przejrzystość rozwiązania - można ją jeszcze poprawić dodając komentarze
- Elastyczność - nie ogranicza nas framework
- Możliwość integracji z frameworkami
- Możliwość zabawy kodem

Zabawa kodem podczas pisania testów

- Autor kodu jest jego najgorszym testerem
- Pisanie testów zwykle jest żmudne i nudne
- Do pisania testów automatycznych się dojrzewa
- Programiści lubią bawić się swoimi wypocinami
- Przygotowując testy tak, żeby były zabawą - rozwiązujemy kilka problemów na raz

Emulacja: qemu-user

qemu-user

- Emulator różnych architektur procesorów
- Działa w przestrzeni użytkownika
- Tłumaczy rozkaz po rozkazie procesora i wykonuje
- Tłumaczy zawołania do jądra (syscalle)
- Pozwala uruchomić aplikację binarną skompilowaną na architekturę X na architekturze Y
- Wspiera kilkanaście architektur
- Nie jest to maszyna wirtualna
- Działa na Linuksach i BSD
- Umożliwia integrację z `binfmt`

Przykład: qemu-user + binfmt

```
$ uname -m
x86_64

$ file ./kit-pval
./kit-pval: ELF 32-bit MSB pie executable, PowerPC or cisco 4500, version 1 (SYS
V), dynamically linked, interpreter /lib/ld.so.1, BuildID[sha1]=cb58bc5388e89afd
ffc07b62999af93135f49dc5, for GNU/Linux 3.2.0, not stripped

$ ./kit-pval -n 123456789012 B
114.97GiB
```

Połączmy kropki

qemu-user wraz z dobrze napisanymi testami automatycznymi umożliwia testowanie kodu skompilowanego na GNU/Linuksa i BSD na różnych architekturach bez dostępu do dodatkowego sprzętu.

Automatyzacja

Po co integrować qemu-user z binfmt?

Taka integracja pozwala bardzo łatwo automatyzować testy wielo-architekturowe. Można np. napisać zestaw prostych skryptów które będą konfigurowały projekt do kompilacji skrośnej w podkatalogach, potem kompilowały go pod wybrane architektury i na koniec uruchamiały testy - bez dodatkowych narzędzi czy wyrafinowanych konfiguracji.

Przykładowy zestaw skryptów jest dostępny na moim githubie:
`https://github.com/jakubjuszczakiewicz/march-demo`
Umożliwia on konfigurację projektu CMake'a, budowanie i uruchamianie testów (ctest) zgodnie z powyższym modelem.

march konfiguracja

```
#!/usr/bin/env -S bash -e

apt install binfmt-support qemu-user-binfmt \
gcc-aarch64-linux-gnu libc6-dev-arm64-cross \
gcc-arm-linux-gnueabi libc6-dev-armel-cross \
gcc-mips-linux-gnu libc6-dev-mips-cross \
gcc-mips64-linux-gnuabi64 libc6-dev-mips64-cross \
gcc-powerpc-linux-gnu libc6-dev-powerpc-cross \
gcc-powerpc64-linux-gnu libc6-dev-powerpc-ppc64-cross \
gcc-sparc64-linux-gnu libc6-dev-sparc64-cross \
gcc-i686-linux-gnu libc6-dev-i386-cross

mkdir -p /usr/gnemul/
cd /usr/gnemul/
ln -s /usr/aarch64-linux-gnu qemu-aarch64
ln -s /usr/arm-linux-gnueabi qemu-arm
ln -s /usr/i686-linux-gnu qemu-i386
ln -s /usr/mips-linux-gnu qemu-mips
ln -s /usr/mips64-linux-gnuabi64 qemu-mips64
ln -s /usr/powerpc-linux-gnu qemu-ppc
ln -s /usr/powerpc64-linux-gnu qemu-ppc64
ln -s /usr/sparc64-linux-gnu qemu-sparc64
```

march-demo

Przykład konfiguracji, budowania i testowania projektu:

```
$ march.cmake.init.sh  ../../../sources/kit-pval
$ march.make.sh
$ march.ctest.sh
```

Podsumowanie

- Testowanie jest potrzebne i trudne
- ... ale nie musi być bardzo żmudne
- Dla otwartych systemów istnieje wiele narzędzi wspomagających pracę z nimi
- Przy odpowiedniej organizacji testowanie kodu może być w miarę ciekawą zabawą

Dziękuję za uwagę!

Pytania?