

# CppUnit - Biblioteka za testiranje C++ koda

Seminarski rad u okviru kursa  
Verifikacija softvera  
Matematički fakultet

Stefan Pantić, 1089/2019  
stefanpantic13@gmail.com

11. januar 2020

## Sažetak

U ovom radu obrađena je biblioteka za testiranje jedinica C++ koda *CppUnit* [4], koja je zasnovana na *xUnit* [7] strukturi biblioteka za testiranje jedinica koda. Obrađeni osnovni koncepti i apstrakcije biblioteke, njena struktura, kao i različiti načini korišćenja. Na kraju je pokazana praktična primena biblioteke za testiranje implementacije šablonske klase **vector** po ugledu na implemetaciju koja se nalazi u STL-u (eng. *Standard Template Library*) [1].

## Sadržaj

<b>1</b>	<b>Uvod</b>	<b>2</b>
<b>2</b>	<b>Arhitektura</b>	<b>2</b>
2.1	Reprezentacija testova	2
2.2	Pokretanje testova	2
2.3	Prikupljanje rezultata testova	3
2.4	Prikazivanje rezultata testova	3
2.5	Omotači i makroi	3
2.5.1	Omotači	3
2.5.2	Makroi	4
<b>3</b>	<b>Testiranje jedinica koda šablonske klase <i>vector</i></b>	<b>4</b>
3.1	Klasa <i>vector</i>	4
3.2	Implementacija testne klase	5
3.3	Pokretanje testova i tumačenje rezultata	6
<b>4</b>	<b>Zaključak</b>	<b>7</b>
	<b>Literatura</b>	<b>7</b>

# 1 Uvod

*CppUnit* je C++ reimplementacija (eng. *port*) biblioteke *JUnit* [6] za testiranje jedinica Java koda. Arhitektura biblioteke i način korišćenja zasnovani su na *xUnit* modelu koji odlikuje raslojena (eng. *decoupled*) višekomponentna arhitektura. Glavna razlika između C++ i Java implementacije jeste što se svaka u velikoj meri oslanja na koncepte i prakse jezika u kojem je napisana, pa samim tim *CppUnit* u velikoj meri koristi šablone (eng. *templates*) radi postizanja što većeg stepena apstrakcije koda i smanjivanja potrebne količine šablonskog koda (eng. *boilerplate*). *CppUnit* je brz i lak za korišćenje. Biblioteka podržava rad u višenitnom okruženju.

## 2 Arhitektura

Pojedinačne komponente biblioteke sastoje se od klasa i interfejsa, od kojih sve imaju strogo definisane funkcionalnosti i uloge. Prema funkcionalnosti, mogu se izdvojiti četiri grupe klasa i interfejsa i peta grupa korisnih makroa i omotača koji olakšavaju korišćenje biblioteke [5]:

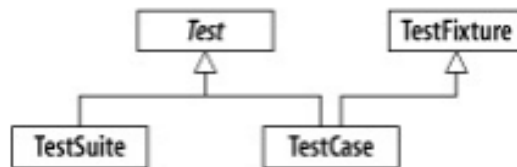
1. Reprezentacija testova
2. Pokretanje testova
3. Prikupljanje rezultata testova
4. Prikazivanje rezultata testova
5. Omotači i makroi

### 2.1 Reprezentacija testova

Osnovni gradivni element *CppUnit* biblioteke je apstraktni interfejs **Test**. On se sastoji iz nekoliko funkcija, od kojih je najbitnija `run(TestResult &result)` koja pokreće sam test i rezultate izvršavanja upisuje u **TestResult** objekat. Njega implementiraju klase **TestCase** - klasa koja sadrži atomičan test i **TestSuite** - kompozicija više **Test** objekata. Još jedna bitna klasa je **TestFixture**, koja implementira takozvani *test fixture* interfejs - atomičan test sa metodama:

- `setUp()` - poziva se pre testa i inicijalizuje okruženje za njegovo izvršavanje.
- `tearDown()` - poziva se nakon svakog testa i deinicijalizuje okruženje u kom je test izvršen.

Na Slici 1 mogu se videti odnosi između pomenutih klasa.



Slika 1: Odnosi osnovnih test klasa biblioteke *CppUnit*

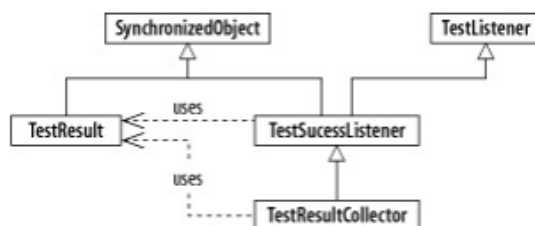
### 2.2 Pokretanje testova

Pokretanje svih testova u *CppUnit* vrši se preko **TestRunner** interfejsa. Ovaj interfejs se sastoji iz dve funkcije:

- `addTest(Test &test)` - dodaje prosledeni test u skup testova koji je potrebno izvršiti.
- `run(TestResult &controller, const std::string &testPath="")` - pokreće sve dodate testove. **TestResult** predstavlja klasu za upis rezultata testova.

## 2.3 Prikupljanje rezultata testova

Za prikupljanje rezultata izvršenih testova zadužena je klasa `TestResult`. Ova klasa ne upisuje rezultate testova direktno u sebe već koristi `TestListener` interfejs, tj. klasu `TestResultCollector` koja ga implementira, da obavesti posmatrača (eng. *observer*) o dobijenim rezultatima. Sve operacije u okviru ove petlje izvršavanja su sinhronizovane *mutex*-ima, te je moguće pokretanje testova u zasebnim nitima. Dobijeni rezultati se prosleđuju `Outputter` objektima koji ih prikazuju korisnicima. Na Slici 3 može se videti petlja izvršavanja (eng. *event loop*) navedenog procesa.

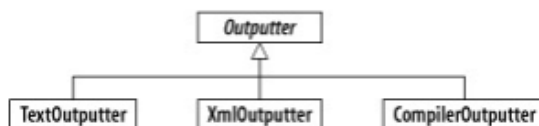


Slika 2: Prikupljanje rezultata testova u okviru biblioteke *CppUnit*

## 2.4 Prikazivanje rezultata testova

Za prikazivanje rezultata testova koristi se `Outputter` apstraktni interfejs koji se sastoji od funkcije `write()` kojom se definiše način ispisa rezultata izvršenih testova. *CppUnit* sadrži implementacije tri konkretne klase za ispis rezultata:

- `TextOutputter` - ispisuje rezultate izvršavanja u čitljivom tekstualnom formatu. Ovo je podrazumevani način ispisa.
- `XMLOutputter` - ispisuje rezultate izvršavanja u XML formatu.
- `CompilerOutputter` - ispisuje rezultate u formatu koji koriste razvojna okruženja koja imaju podršku za grafičko prikazivanje rezultata testova.



Slika 3: Klase za prikaz rezultata testova u okviru biblioteke *CppUnit*

## 2.5 Omotači i makroi

Pored navedenih klasa, biblioteka sadrži i određen broj omotača i makroa koji apstrahuju detalje niskog nivoa i time smanjuju količinu šablonskog (eng. *boilerplate*) koda, olakšavaju upotrebu i smanjuju mogućnosti za pogrešne upotrebe funkcionalnosti.

### 2.5.1 Omotači

Korišćenjem navedenih klasa i interfejsa dobija se velika fleksibilnost prilikom pokretanja testova jedinica C++ koda. Ipak, većina slučajeva upotrebe prati isti tok akcija, pa je moguće

grupisati određene često korišćene klase i interfejsu u jedan omotač koji će apstrahovati detalje niskog nivoa i olakšati korišćenje. Glavni omotači koji se mogu naći u biblioteci *CppUnit* su: `TextTestRunner`, `XMLTestRunner`, `CompileTestRunner`, `QtTestRunner`, i drugi. Oni apstrahuju proces pokretanja, prikupljanja i ispisivanja rezultata testova korišćenjem `TestRunner` interfejsa. Time je proces sakriven od korisnika, pa je dovoljno implementirati testove, registrovati ih i pokrenuti, bez brige o prikupljanju i ispisu rezultata.

### 2.5.2 Makroi

Opšte gledano, *CppUnit* sadrži dva tipa makroa:

- Makroi za deklarisanje testova
- Makroi za pisanje testova

**Makroi za deklarisanje testova** služe da sakriju implementacione detalje pojedinih testova od korisnika, tj. umesto da programer implementira čitavu klasu za svaki test, dovoljno je implementirati funkciju koja predstavlja test, a zatim je registrovati kao test korišćenjem makroa `CPPUNIT_TEST(func)`. Analogno navedenom makrou, `TestSuite` se implementira navođenjem jednog ili više `CPPUNIT_TEST(func)`-a između makroa `CPPUNIT_TEST_SUITE(cls)` i `CPPUNIT_TEST_SUITE_END()`. Navedeni su samo neki primeri makroa za deklarisanje testova, dok se pun spisak može naći u dokumentaciji [4].

**Makroi za pisanje testova** su skup naredbi pretpostavki (eng. *assert statement*) koje se koriste za ispitivanje pretpostavki koje moraju biti ispunjene da bi naš kod prošao test. Neki od makroa pretpostavki su:

- `CPPUNIT_ASSERT(expr)` - test prolazi ukoliko se `expr` evaluira na `true`.
- `CPPUNIT_ASSERT_MESSAGE(msg, expr)` - isto kao prethodni, s' tim što se poruka `msg` ispisuje ukoliko se `expr` evaluira na `false`.

## 3 Testiranje jedinica koda šablonske klase *vector*

U ovoj sekciji, biblioteka *CppUnit* primenjena na testiranje implementacije šablonske klase `vector` koja je implementirana prema ugledu na onu koja se nalazi u standardnoj biblioteci jezika C++. Klasa `std::vector` biće korišćena kao referentna tačka prilikom testiranja, tj. test će se smatrati uspešnim ukoliko članska funkcije naše klase proizvede isti rezultat kao ekvivalentna članska funkcija referentne klase. Za testiranje biće korišćen pomenuti interfejs `TestFixture` koji vrši kompoziciju više testova, a uz to ima par propratnih funkcija za (de)inicijalizaciju okruženja za izvršavanje testa (Sekcija 2.1). Pokretanje testova vršiće se preko klase omotača `TextTestRunner` pomenutog u Sekciji 2.5.1. Izvorni kod ovog primera može se naći na linku <https://github.com/stefanpantic/cpp-playground/tree/master/container-classes/Vector>.

### 3.1 Klasa *vector*

`std::vector` je šablonska klasa iz standardne biblioteke C++-a, koja predstavlja dinamički niz vrednosti. Spada u grupu *container*-a [2], tj. klasa čiji su sadržaj objekti drugih klasa. Svaki *container* iz standardne biblioteke ima uniforman interfejs koji omogućava polimorfno ponašanje svih algoritama standardne biblioteke (`std::transform`, `std::copy` i drugi), nezavisno od tipa samog *container*-a. Detaljan opis klase `std::vector` može se naći u njegovoj dokumentaciji [1].

Naša implementacija prati ovu specifikaciju i implementira sve potrebne funkcije javnog interfejsa radi postizanja uniformnosti sa standardnom bibliotekom. Deklaracija se može naći u Listingu 1. Šablon `T` predstavlja tip vrednosti koja se nalazi u `vector`-u, dok šablon `A` predstavlja alokator kojim su vrednosti vektora alocirane [3].

```

1 template <typename T, typename A = std::allocator<T>>
2 class vector;

```

Listing 1: Deklaracija klase

### 3.2 Implementacija testne klase

Pre implementacije testne klase, potrebno je definisati koje funkcionalnosti našeg koda želimo da testiramo. Za potrebe ovog primera fokusiraćemo se samo na proveravanje validnosti konstruktora, operacija brisanja/dodavanja elementata u `vector` i proveru kompatibilnosti sa standardnom bibliotekom. Potpuno testiranje obuhvatilo bi detaljno testiranje iteratora i performansi.

Nakon što smo odredili koje funkcionalnosti `vector`-a želimo da testiramo potrebno je napisati potrebne testove. U ovu svrhu biće korišćen interfejs `TestFixture` i nekoliko korisnih makroa koje *CppUnit* nudi korisnicima.

Klasu `VectorTestFixture` ćemo implementirati kao podklasom `TestFixture`-a, s' tim što ćemo je dodatno implementirati kao šablonsku da bismo podržali testiranje `vector`-a vrednosti proizvoljnih tipova alociranih proizvoljnim alokatorom. U Listingu 2 može se videti deklaracija našeg testnog postrojenja.

```

1 template<
2     typename T,
3     typename A = std::allocator<T>,
4     size_t _size = 1000000,
5     typename _element_generator = detail::_rand_gen,
6     typename _BinaryPredicate = detail::_comp<T>
7 >
8 class vector_test_fixture : public CppUnit::TestFixture;

```

Listing 2: Deklaracija testnog postrojenja

Primitimo da pored šablona `T` i `A` pomenutih u Sekciji 3.1, uvodimo tri dodatna šablona za veličinu vektora koji se testira, nasumični generator vrednosti i komparator generisanih vrednosti, redom, radi podizanja nivoa apstrakcije i ponovne upotrebljivosti koda. Zatim, potrebno je deklarirati funkcije za (de)inicijalizaciju okruženja, kao i deklarirati sve testne funkcije koje ćemo implementirati. Deklaracije pomenutih funkcija nalaze se u Listingu 3. Potrebno je napomenuti da se, pored navedenih deklaracija, testne funkcije moraju deklarirati u okviru klase i na klasičan C++ način.

```

1 public:
2     // <implement setUp() and tearDown() for CppUnit::TestFixture to
3     // use to initialize and delete variables before and after each test>
4     void setUp();
5     void tearDown();
6 private:
7     // <add vector_test_fixture<T, A, _size, _element_generator, _BinaryPredicate> to
8     // CppUnit::TestSuite>
9     CPPUNIT_TEST_SUITE(vector_test_fixture);
10
11     // <test methods>
12     CPPUNIT_TEST(default_constructor_test);
13     CPPUNIT_TEST(copy_constructor_test);
14     CPPUNIT_TEST(move_constructor_test);
15     CPPUNIT_TEST(copy_assignment_test);
16     CPPUNIT_TEST(move_assignment_test);
17     CPPUNIT_TEST(push_back_test);
18     CPPUNIT_TEST(pop_back_test);
19     CPPUNIT_TEST(swap_test);
20     CPPUNIT_TEST(clear_test);

```

```

20 CPPUNIT_TEST(access_operator_test);
21 CPPUNIT_TEST(insert_test);
22 CPPUNIT_TEST(erase_test);
23 CPPUNIT_TEST(range_erase_test);
24
25 CPPUNIT_TEST_SUITE_END();
26 // </add>

```

Listing 3: Deklaracija testnih funkcija

Testno postrojenje je potrebno registrovati pre korišćenja (Listing 4).

```

1 using def_vect = vector_test_fixture<double, std::allocator<double>, 30000000>;
2 CPPUNIT_TEST_SUITE_NAMED_REGISTRATION(def_vect, "..."); // named registration

```

Listing 4: Registrovanje testnog postrojenja

### 3.3 Pokretanje testova i tumačenje rezultata

Pokretanje testova vršiće se preko klase `TextTestRunner`. Ova klasa obuhvata implementaciju `TestRunner` interfejsa (Sekcija 2.2), prikupljača rezultata testova (Sekcija 2.3) i `TextOutputter` klasu za ispis rezultata u čitljivom formatu (Sekcija 2.4).

Proces pokretanja testova je jednostavan, potrebno je napraviti instancu klase `TextTestRunner`, zatim dohvatiti registrovane testne klase i pokrenuti testove korišćenjem funkcije `run()`. Testovi će biti pokrenuti redosledom kojim su dodati `runner`-u. Primer koda za dodavanje i pokretanje testova nalazi se u Listingu 5.

```

1 CppUnit::TextTestRunner runner;
2
3 runner.addTest(CppUnit::TestFactoryRegistry::getRegistry("...").makeTest());
4 runner.addTest(CppUnit::TestFactoryRegistry::getRegistry("...").makeTest());
5 ...
6 runner.addTest(CppUnit::TestFactoryRegistry::getRegistry("...").makeTest());
7
8 runner.run();

```

Listing 5: Kreiranje i pokretanje klase sa testovima

Prilikom izvršavanja testova na standardni izlaz biće ispisana tačka ukoliko se test uspešno izvrši, odnosno slovo F ukoliko test nije uspešan. Nakon izvršavanja svih testova na standardnom izlazu biće ispisano OK ukoliko su svi testovi bili uspešni, a u suprotnom ispisaće se detalji o neuspelim testovima. Primer ispisa neuspelog testa nalazi se u Listingu 6.

```

.....F

!!!FAILURES!!!
Test Results:
Run: 13  Failures: 1  Errors: 0

1) test: vector_test_fixture<double, std::allocator<double>, 30000000ul, detail::
   _rand_gen, detail::_comp<double> >::range_erase_test
(F) line: 536 cppunit/vector.test.hpp
assertion failed
- Expression: _v2->size() != std_vect.size()
- range_erase - size[during erase]

```

Listing 6: Ispis detalja o neuspelim testovima

## 4 Zaključak

*CppUnit* je biblioteka za testiranje jedinica C++ koda zasnovana na *xUnit* modelu. Ova činjenica je čini izuzetno fleksibilnom i lakom za korišćenje, pogotovo za programere koji su imali dodira sa bilo kojom bibliotekom zasnovanom na ovom modelu. Činjenica da je implementirana u C++-u uz korišćenje šablona znači da je izuzetno fleksibilna i efikasna. Visok stepen raslojenosti izmedju komponenti dovodi do velike modularnosti. Ipak, pored toga, postoje implementacije korisnih klasa koje apstrahuju ove granularnosti i omogućavaju izuzetno lako korišćenje bez lične implementacije sitnih detalja.

## Literatura

- [1] ISO C++ Standards Comity. cppreference, 2019. on-line at: <https://en.cppreference.com/w/>.
- [2] ISO C++ Standards Comity. cppreference, 2019. on-line at: <https://en.cppreference.com/w/cpp/container>.
- [3] ISO C++ Standards Comity. cppreference, 2019. on-line at: [https://en.cppreference.com/w/cpp/named\\_req/Allocator](https://en.cppreference.com/w/cpp/named_req/Allocator).
- [4] Jerome Lacoste (lacostej@altern.org) J.E. Hoffmann (e-h@gmx.net) Baptiste Lepilleur (gaiacrtn@free.fr) Bastiaan Bakker (bastiaan.bakker@lifeline.nl) Steve Robbins (smr99@sourceforge.net) Eric Sommerlade (sommerlade@gmx.net), Michael Feathers (mfeathers@objectmentor.com). CppUnit - Unit testing library, 2019. on-line at: <http://cppunit.sourceforge.net/doc/1.8.0/index.html>.
- [5] Paul Hamill. *Unit Test Frameworks*. O'Reilly, first edition, 2004.
- [6] The JUnit Team. JUnit 5, 2019. on-line at: <https://junit.org/junit5/>.
- [7] Wikipedia.org. xUnit, 2019. on-line at: <https://en.wikipedia.org/wiki/XUnit>.