Verifikacija softvera - Vežbe

Ivan Ristović Ana Vulović

2022-10-10

Kolekcija materijala sa vežbi za kurs Verifikacija softvera na Matematičkom fakultetu, Univerziteta u Beogradu.

Teme i alati rađeni na vežbama:

- Debagovanje koristeći alate za debagovanje i razvojna okruženja
  - `gdb` - debagovanje koda na niskom nivou
  - `QtCreator` - debagovanje C/C++ kodova
- Testiranje jedinica koda
  - Pisanje testabilnog koda
  - `QtTest`
  - `JUnit` (Java)
  - `xUnit` , `NUnit` (C#)
- Praćenje pokrivenosti koda testovima
  - `lcov` (C, C++)
  - `JaCoCo` (Java)
- Testiranje pomoću objekata imitatora
  - Ručno pisanje imitator klasa (C++)
  - Imitatori baza podataka (C#)
  - `Moq` (C#)
- Profajliranje
  - `Valgrind` (memcheck, cachegrind, callgrind, hellgrind, drd)
  - `perf`
  - `Intel® VTune™`
- Statička analiza
  - `KLEE`
  - `CBMC`
  - `Clang` statički analizator
- Alati i jezici za formalnu verifikaciju softvera
  - `Dafny`

Sadržaj

1 Debagovanje

1.1 QtCreator Debugger

QtCreator dolazi sa debugger-om koji predstavlja interfejs između QtCreator-a i native debugger-a (gdb, CDB, LLDB, ...). Moguće je debagovati Qt aplikacije, ali i native C/C++ aplikacije, kačiti se na već pokrenute procese i formirati debug sesije, i mnogo toga.

Više informacija je moguće pronaći u Qt dokumentaciji.

1.1.1 Buffer Overflow primer (QtCreator)

Ukoliko debagujemo Qt aplikacije, dovoljno je pokrenuti aplikaciju u Debug modu. Pokrenuti QtCreator i otvoriti BufferOverflow.pro projekat. Prilikom otvaranja projekta, QtCreator će otvoriti konfiguracioni dijalog - konfigurisati projekat sa podrazumevanim podešavanjima.

Napomena: QtCreator podrazumevano uključuje `clang` statički analizator ukoliko je on dostupan. Iako nije neophodno, ukoliko želite da sami pronađete probleme u izvornom kodu a tek potom proverite svoje pronalaske koristeći `clang` , isključite `clang` analizator u podešavanjima (`Edit -> Preferences -> Analyzer`).

Unutar QtCreator-a vidimo nekoliko pogleda (`Edit` , `Design` , `Debug` itd.). Trebalo bi da je `Edit` pogled već selektovan. Otvoriti `main.c` fajl i prodiskutovati o logici programa i njegovim slabim tačkama.

Unutar `Project` prozora možemo dodavati ili menjati postojeće konfiguracije za kompilaciju i pokretanje programa. To uključuje i dodavanje argumenata komandne linije (`Run` odeljak).

Pokrenuti debug sesiju. Isprobati debug akcije za kontrolisanje izvršavanja programa (`Debug` meni u glavnom meniju):

- `Interrupt`
- `Continue`
- `Step over`
- `Step into`
- `Step out`
- `Set/Remove breakpoint`

Možemo testirati ponašanje programa tako što prvo unesemo ispravnu lozinku `MyPassword` a zatim i proizvoljnu neispravnu lozinku (npr. `SomePassword`). Program se naizgled ispravno ponaša ali to ne znači da je u potpunosti ispravan.

Posmatrajući definicije promenljivih `password` i `ok` , možemo zaključiti da će se one na steku naći jedna do druge. Proverimo da li je zaista tako: desnim

klikom na promenljive u prozoru za prikaz promenljivih na steku selektovati opciju `Open Memory Editor -> ... at object address ...` . Nastaviti izvršavanje i primetiti inicijalizaciju memorije za promenljivu `ok` . Pošto je promenljiva `password` neposredno pre promenljive `ok` u memoriji, i pošto se sadržaj promenljive `password` unosi sa standardnog ulaza, ukoliko uspemo da pređemo granice promenljive `password` onda možemo upisati proizvoljnu vrednost u promenljivu `ok` (pa i `"yes"`)!

Posmatrajmo isečak koda:

```
  scanf ("%s", password);
```

`scanf` nam ne garantuje da će učitati najviše `16` karaktera (koliko smo rezervisali za promenljivu `password`). Testirajmo ponašanje za niske dužine veće od 16 karaktera, npr. `WillThisPassNow?yes` . Vidimo da smo dobili privilegije iako lozinka nije ispravna.

Postavimo uslovni *breakpoint* nakon poziva `scanf` , da se program zaustavi ukoliko se promeni vrednost promenljive `ok` . Desnim klikom na `Breakpoint` i selekcijom opcije `Edit Breakpoint` možemo definisati uslov `ok != "no"` u `Condition` polju. Možemo takođe pratiti inicijalizaciju promenljive `ok` u `Memory Editor` -u. Nakon što se vrednost promeni, možemo je ručno vratiti na `"no"` .

Tok izvršavanja možemo pratiti i preko prikaza steka, odakle možemo videti redosled pozivanja funkcija. Konkretnu liniju u kodu koja je sledeća za izvršavanje, QT obeležava žutom strelicom levo od koda. Ukoliko se dogodi da smo mnogo napredovali i želimo da se pomerimo na neku ranije naredbu, dovoljno je da strelicu prevučemo na naredbu koja nam je potrebna, bez potrebe za ponovnim pokretanjem programa. Za ispekciju asemblerskog koda možemo otvoriti *Disassembler* prozor.

Ostaje pitanje - šta naš program radi ukoliko je lozinka toliko dugačka da prevazilazi veličinu steka (npr. `ThisPasswordSimplyExceedsMaximumLengthAvailableOnStack`)? Dobićemo poruku `*** stack smashing detected ***` . To je posledica opcije `gcc` -a `-fstack-protector` koja generiše zaštitnu promenljivu koja je se dodeljuje osetljivim funkcijama. Osetljivim se smatraju one funkcije koje koriste dinamičku alokaciju ili imaju bafere veće od `8B` . Zaštitna promenljiva se inicijalizuje pri ulasku u funkciju i proverava se na izlasku. Ukoliko provera ne bude tačna, štampa se prikazana poruka i program prekida sa izvršavanjem. Ukoliko želimo da učinom kod još ranjivijim na ovakve napade, možemo pomenutu zaštitu isključiti opcijom `-fno-stack-protector` . U slučaju Qt projekta potrebno je dodati ovu opciju u `QMAKE_CFLAGS` promenljivu. U definiciji projekta (datoteci ekstenzije `.pro`) uneti sledeći red a zatim pozvati `qmake` i izvršiti ponovno prevođenje projekta:

```
QMAKE CFLAGS += -fno-stack-protector
```

Da bismo sprečili napade prekoračenjem bafera savet je da se primenjuju dobre programerske prakse i da se:

- preoverava upravljanje memorijom tokom programa koristeci neki od alata poput `valgrind memcheck`
- upotrebljava `fgets()` funkcije umesto `gets()` ili `scanf()` koje ne vrše provere granica promenljivih.
- upotrebljava `strncmp()` umesto `strcmp()` , `strncpy()` umesto `strcpy()` , itd.

1.1.1.1 Debagovanje spoljašnje aplikacije kroz QtCreator Ukoliko pokrenemo prethodni primer izvan QtCretor-a, to ne znači da ne možemo debagovati taj program. Štaviše, moguće je QtCreator debugger zakačiti za bilo koji program koji se izvršava ili će se tek izvršiti. Iz menija `Debug` biramo `Start Debugging` . Na raspolaganju su nam opcije:

- *Attach debugger to started application* - da bismo debagovali aplikaciju koja je već pokrenuta.
- *Attach debugger to unstarted application* - da bismo debagovali aplikaciju koja je će biti pokrenuta. Zadaje se putanja do izvršne verzije programa. Debugger će se aktivirati kada aplikacija bude pokrenuta.
- *Start and Debug External Application* - slično kao prethodno samo što će se program pokrenuti odmah. Možemo čekirati opciju da se doda tačka prekida na početak `main` funkcije.

Napomena: Ako QtCreator ne uspe da se poveže na proces, moguće je da treba da se dozvoli povezivanje na procese tako što se upiše `0` u fajl `/proc/sys/kernel/yama/ptrace_scope` .

Da bismo imali informacije o linijama izvornog koda koji odgovara programu koji se debaguje, potrebno je da taj program bude preveden sa uključenim debug informacijama (za `gcc` to je opcija `-g`). Inače, debugger će nam prikazivati memoriju kojoj pristupa program.

1.2 gdb

GNU Debugger (gdb) je debugger koji se može koristiti za debagovanje (najčešće) C/C++ programa. Preko gdb je moguće pokrenuti program sa proizvoljnim argumentima komandne linije, posmatrati stanje promenljivih ili registara procesora, pratiti izvršavanje kroz naredbe originalnog ili asembliranog koda, postavljanje bezuslovnih ili uslovnih tačaka prekida i sl. Više o gdb se može pročitati na ovom linku.

Koristeći gdb možemo učitati program iz prethodnog primera. U slučaju da želimo da ručno prevedemo program, neophodno je da se postaramo da je prosleđen `-g` flag `gcc` kompilatoru:

```
$ gcc main.c -g -o BufferOverflow
```

Pošto je QtCreator već preveo ovaj program i u `build-*` direktorijum ostavio izvršivi fajl, možemo ga direktno učitati u gdb:

```
$ gdb BufferOverflow
...
Reading symbols from BufferOverflow...
(gdb)
```

Alternativno, možemo program učitati u već pokrenuti gdb proces:

```
$ gdb
...
(gdb) file BufferOverflow
Reading symbols from BufferOverflow...
(gdb)
```

U slučaju da debug simboli nisu prisutni, možemo u definiciji projekta dodati odgovarajući flag (pod `QMAKE_CFLAGS`).

Spisak komandi koje gdb pruža možemo dobiti komandom `help` . Za sve ove naredbe postoje i skraćene varijante - npr. za `running` možemo kucati `run` ili samo `r` .

```
(gdb) help
List of classes of commands:

aliases -- User-defined aliases of other commands.
breakpoints -- Making program stop at certain points.
data -- Examining data.
files -- Specifying and examining files.
internals -- Maintenance commands.
obscure -- Obscure features.
running -- Running the program.
stack -- Examining the stack.
status -- Status inquiries.
support -- Support facilities.
text-user-interface -- TUI is the GDB text based interface.
tracepoints -- Tracing of program execution without stopping the program.
user-defined -- User-defined commands.
```

Tačke prekida možemo postaviti komandom `breakpoints` , aktiviramo/deaktiviramo komandama `enable` / `disable` , a brišemo komandom `delete` . Komandom `continue` nastavljamo rad programa do naredne tačke prekida. Tačke prekida mogu biti linije, npr. `main.c:15` ali i funkcije, npr. `main` ili `main.c:grant_privilege` . Postavimo tačku prekida na funkciju `grant_privilege` , kontrolišimo izvršavanje programa naredbama `step` (nalik na *step into*) i `next` (nalik na *step over*):

```
(gdb) b main.c:grant_privilege
Breakpoint 1 at 0x1171: file ../01_buffer_overflow/main.c, line 5.
(gdb) r
Starting program: BufferOverflow
Breakpoint 1, grant_privilege () at ../01_buffer_overflow/main.c:5
5       {
(gdb) s
7               char ok[16] = "no";
(gdb) n
9               printf("\n Enter the password : \n");
(gdb) n

 Enter the password :
10              scanf("%s", password);
(gdb) n
MyPassword12        if (strcmp(password, "MyPassword")) {
(gdb) info locals
password = "MyPassword\000\367\377\177\000"
ok = "no", '\000' <repeats 13 times>
(gdb) n
15                      printf("\n Correct Password \n");
(gdb) n

 Correct Password
16                      strcpy(ok, "yes");
(gdb) n
19              if (strcmp(ok, "yes") == 0) {
(gdb) n
23                  printf("\n Root privileges given to the user \n\n");
(gdb) n

 Root privileges given to the user

25      }
(gdb) finish
Run till exit from
#0  grant_privilege () at ../01_buffer_overflow/main.c:25
main () at ../01_buffer_overflow/main.c:43
43              return 0;
```

Komandom `info` možemo dobiti informacije o lokalnim promenljivim, registrima itd. (npr. `info locals` tj. `info registers`). Komandom `info breakpoints` možemo videti spisak tačaka prekida.

Alternativno, moguće je iskoristiti drugačiji korisnički interfejs, komandom `tui` npr. `tui enable` . Tako dobijamo pogled na izvorni kod na jednoj polovini korisničkog interfejsa. Komandom `tui reg` možemo dobiti i prikaz registara sa `tui reg all` .

Koristeći gdb možemo dodati i uslovne tačke prekida, npr.:

```
(gdb) b grant_privilege:12 if ok != "no"
```

2 Pisanje testabilnog koda

Da bismo efikasno pisali testove jedinica koda, neophodno je da kod pišemo tako da bude pogodan za testiranje, ali i da softver razvijamo tako da je moguće paralelno pisati testove. Jedan način da se obezbedi takav način rada je razvoj vođen testovima (engl. *Test-Driven-Development*, skr. *TDD*). U situacijama kada već imamo dostupan izvorni kod i treba da napišemo testove jedinica koda, treba znati kako pristupiti pisanju testova i kako refaktorisati kod tako da bude pogodan za testiranje. Nekada funkcije koje treba da testiramo nemaju deterministički izlaz tako da ih je nemoguće testirati u izvornom obliku.

U ovom poglavlju će biti reči o tehnikama za pisanje testabilnog i skalabilnog koda. Primeri na kojima će ove tehnike biti prikazane su pisani u raznim programskim jezicima ali su koncepti koje prikazuju univerzalni, kao npr. inverzija zavisnosti, zamena implementacije itd.

2.1 C++ primer - kalkulator

Naš kod za kalkulator je nepodesan za jedinično testiranje jer se mnogo posla zapravo obavlja u main funkciji. Da bismo sve delove testirali moramo kod izdeliti na funkcije koje obavljaju po jednu celinu. Već postoje funkcije za svaku pojedinačnu operaciju koje imaju dva argumenta, njih nećemo dirati.

U `main` funkciji test podaci se učitavaju sa standardnog ulaza i sve se ispisuje na standardni izlaz. Da bismo mogli u jediničnim testovima da zadajemo svoje ulaze, promenićemo da funkcije ne učitavaju sa `std::cin` već sa proizvoljnog toka tipa `std::istream` , i da ispisuju ne na `std::cout` , već na proizvoljni tok tipa `std::ostream` . Na taj način ćemo izbeći učitavanje sa standardnog ulaza, a moći ćemo da unapred zadamo ulaz na kom se testira.

Iz tog razloga već postojećoj funkciji `showChoices` treba dodati argument `std::ostream & ostr` na koji će ispisivati ponudene opcije umesto na `std::cout` . Vidimo da se u `main` funkciji nakon prikaza opcije očekuje unos izbora operacije i proverava ispravosti unetog podatka. Taj deo objedinjen izdvajamo u posebnu funkciju `readChoice` koja će pozivati funkciju `showChoices` i učitavati opciju sa ulaznog toka sve dok se ne unese jedna od ispravnih cifara.

Nakon toga bi trebalo da se unesu dva operanda nad kojima će se primeniti izabrana operacija. To se može izdvojiti u posebnu funkciju `readOperands` koja će učitati dva realna broja sa ulaznog toka. Primenu odabrane operacije na unete operande na osnovu izabrane opcije izdvajamo u posebnu funkciju `calculate` . U njoj ćemo proveriti da li je opcija validna, primeniti operaciju i vratiti rezultat. Ukoliko se ne pošalje dobra vrednost argumenta `choice` funkcija bi, npr., mogla da baci izuzetak tipa `std::invalid_argument` ili vrati vrednost `false` . Proveru možemo uraditi i sa `assert(choice >= 1 && choice <= 4);` iz `cassert` zaglavlja. Ukoliko se ne pošalje

očekivana vrednost za argument `choice` program će biti prekinut. Ovo nam je način da nametnemo važenje preduslova za funkciju. Istu proveru bismo mogli da stavimo i u `printResults` funkciju. Dobili smo funkcije koje rade nezavisne poslove. Time su potencijalno upotrebljive i za neku dalju upotrebu ili pisanje kompleksnijeg kalkulatora. Sada i `main` funkcija može izgledati jednostavno kao niz poziva ovih funkcija.

2.2 Java OOP primer - Game of Life

Program predstavlja implementaciju poznate ćelijske automatizacije pod imenom Game of Life. Početna konfiguracija igre (početno stanje table kao i veličina mreže) su konfigurabilni. Stanje igre se ispisuje na standardni izlaz nakon svake iteracije.

Da bismo pisali testove jedinica koda, moramo razumeti odnose klasa u okviru aplikacije:

- paket `app` sadrži implementaciju `GameOfLife` igre
- paket `model` sadrži implementacije klasa `Cell` i `Grid` , koje se koriste za reprezentaciju stanja igre
  - paket `model.conf` sadrži implementacije početnih stanja igre (nasumičnu i jednu unapred kreiranu sa specifičnim osobinama, pod nazivom `Glider`)

Testovi koje bismo voleli da napišemo bi testirali:

- da li se mreža ispravno kreira na osnovu konfiguracije
- da li se pravila igre ispravno primenjuju iz generacije u generaciju
- da li se stanje igre ispravno ispisuje na standardni izlaz

Testove da li se mreža ispravno kreira na osnovu konfiguracije možemo lako napraviti. `Grid` klasa već implementira sve neophodno. Jedini problem predstavlja to što nemamo način da ručno postavimo konfiguraciju koja će se koristiti kao početna. Možemo testirati nasumičnu konfiguraciju međutim radije bismo da imamo deterministične testove. Takođe, naši testovi bi idealno testirali konfiguraciju u specijalnim slučajevima, npr. ivice mreže, što ne možemo kontrolisati nasumičnom konfiguracijom. Možemo ručno promeniti stanje klase `Grid` nakon kreiranja ali testovi **nikada** ne treba da zalaze u detalje implementacije klasa, tj. njihova stanja. Kreiranje ručnih determinističkih konfiguracija nam omogućava i testiranje pravila igre - možemo kreirati `Grid` objekat sa specifičnom konfiguracijom namenjenom da testira određeno pravilo igre ili kombinaciju više pravila, pozivajući metod `Grid.advance()` i testirati da li su se ćelije promenile onako kako bi trebalo. Zatim možemo testirati i specijalne slučajeve kao što su ivice mreže kao i situacije u kojima se na jednu ćeliju primenjuje više pravila.

Da bismo implementirali ručne konfiguracije, primetimo da je `model.conf.GliderConfiguration` jedna implementacija ručne konfiguracije. Možemo da implementiramo naše specifične konfiguracije na sličan način, međutim imali bismo ponavljanje koda pošto bi se jedino menjao konstruktor klase. Apstrahujmo `GliderConfiguration` implementaciju - dodajemo klasu `ManualConfiguration` i menjamo klasu `GliderConfiguration`

da nasleđuje klasu `ManualConfiguration`. Sada naše test konfiguracije mogu da instanciraju `ManualConfiguration` sa odgovarajućom konfiguracijom mreže, a takođe aplikacija može da se proširi dodavanjem novih predefinisanih konfiguracija.

Slično kao u kalkulator primeru, treba apstrahovati rad sa standardnim izlazom kako bismo mogli da testiramo prikaz stanja igre. Prikaz stanja se trenutno vrši u glavnoj klasi aplikacije, što takođe nije optimalno. Dodajmo `views` paket sa implementacijom `View` interfejsa koji predstavlja apstraktnu implementaciju prikaza aplikacije. Sada možemo kreirati implementaciju koja ispisuje stanje igre na proizvoljni izlazni tok (`PrintStreamView`) odnosno `System.out` ukoliko izlazni tok nije naveden. Logiku ispisa stanja igre pomeramo iz `app.GameOfLife` u `view.ConsoleView`.

2.3 C# - REST API klijent primer

Aplikacija je primer RESTful API klijenta koji prikazuje trenutnu temperaturu i dnevnu prognozu tako što kontaktira servis OpenWeather, tačnije njegov API server (*Napomena: pokretanje primera zahteva ključ koji aplikacija traži u fajlu* `key.txt`). Detalji funkcionalnosti ovog servisa nisu od značaja za razumevanje ovog primera. Pojednostavljeno, klijent će serveru poslati HTTP zahtev za odgovarajućim resursom (trenutna temperatura, prognoza, i sl.) i server će poslati objekat sa odgovarajućim informacijama serijalizovanim u JSON. Pogledajmo implementaciju:

- prostor imena `Common` sadrži klase koje se koriste za deserijalizaciju odgovora servera
- prostor imena `Services` sadrži klasu `WeatherService` koja će biti meta naših testova
- glavni prostor imena koristi `WeatherService` da prikaže trenutnu temperaturu i prognozu za odgovarajući upit

Da bismo pisali testove za klasu `WeatherService`, pogledajmo njen javni interfejs (to što su neke funkcije asinhrone nema uticaj na suštinu primera):

```
bool IsDisabled();
async Task<CompleteWeatherData?> GetCurrentDataAsync(string query);
async Task<Forecast?> GetForecastAsync(string query);
```

Metod `IsDisabled` može lako da se testira kreiranjem servisa bez ključa. Druge metode, međutim, nisu toliko jednostavne pošto u sebi rade više od jednog posla - kreiranje HTTP zahteva, slanje zahteva, primanje odgovora i deserijalizacija odgovora. Ukoliko bismo testirali ove metode bez ikakve izmene, onda bismo stalno slali HTTP zahteve servisu u našim testovima - što je sporo i nepovoljno. Štaviše, nemoguće je ovako testirati ponašanje naše implementacije u slučaju da server vrati nevalidan ili nekompletan odgovor. Čak i da možemo nekako rešiti sve te problema, ne možemo znati unapred koje odgovore servera da očekujemo - temperaturu i prognozu ne možemo znati unapred. Možemo pokrenuti lokalnu instancu OpenWeather servera modifikovanu za

naše potrebe ali to nije optimalno rešenje. Oba ova problema (nedeterminističnost rada servisa i testiranje višestrukih funkcionalnosti jednog metoda) možemo rešiti tako što ručno ubrizgamo odgovor servera. Trenutna implementacija nam to ne dozvoljava, tako da hajde da je modifikujemo, ali ujedno i proširimo.

Pre svega, preimenujmo `WeatherService` u `OpenWeatherService` i apstrahujmo interfejs ove klase `IWeatherService`. Kontaktiranje servera možemo apstrahovati u logiku klase `WeatherHttpService`. U našim testovima, možemo koristiti implementaciju nalik na onu u klasi `TestWeatherHttpService`. Ne želimo da `OpenWeatherService` direktno koristi `WeatherHttpService` pošto nam to ne omogućava zamenu implementacije klase `WeatherHttpService` klasom `TestWeatherHttpService` u našim testovima. Stoga kreirajmo interfejs `IWeatherHttpService` koji će implementirati klase `WeatherHttpService` i `TestWeatherHttpService`. Sada klasa `OpenWeatherService` može da ima zavisnost na interfejs u konstruktoru (što dodatno omogućava druge povoljnosti kao što je ubrizgavanje zavisnosti).

3 Testiranje jedinica koda metodom bele kutije

Jedinični testovi (engl. *Unit tests*) treba da budu kreirani za sve **javne** metode klasa, uključujući konstruktore i operatore. Trebalo bi da pokriju sve glavne putanje kroz funkcije, uključujuću različite grane uslova, petlji itd. Jedinični testovi bi trebali da pokriju i trivijalne i granične slučajeve, kao i situacije izvršavanja metoda nad pogrešnim podacima da bi se testiralo i reagovanje na greške.

3.1 C++ - QtTest, gcov, lcov

3.1.1 Pisanje testova jedinica koda pomoću QtTest radnog okvira

QtCreator kao razvojno okruženje pruža mogućnost pisanja jediničnih testova. To je moguće učiniti pisanjem projekta tipa `Qt Unit Test` na više načina, u zavisnosti od toga da kako želimo da organizujemo stvarni i test projekat.

3.1.1.1 Uključivanjem biblioteke Prvi način kako možemo kreirati test projekat koji testira već postojeći projekat je tako što kreiramo novi projekat tipa `Qt Unit Test` :

```
New Project -> Other Project -> Qt Unit Test
```

Pošto želimo da pišemo testove za već postojeći projekat, treba da dodamo lokaciju projekta opcijom `Add Existing Directory` .

Unutar test projekta će se kreirati klasa koja nasleduje `QObject` . Ispod modifikatora `private Q_SLOTS` pišemo bar jednu test funkciju. Trebalo bi da su nam već ponuđene funkcije npr. `testCase1()` .

Dodatno, postoje i četiri privatne metode koje se ne tretiraju kao test funkcije, ali će ih test radni okvir izvršavati bilo kada inicijalizuje ili čisti za celim testom ili trenutnom test funkcijom:

- `initTestCase()` će biti pozvana pre izvršavanja prve test funkcije
- `cleanupTestCase()` će biti pozvana nakon izvršavanja poslednje test funkcije
- `init()` će biti pozvana pre svakog poziva test funkcije
- `cleanup()` će biti pozvana nakon svakog izvrašavanja test funkcije

Ukoliko se `initTestCase()` ne izvrši uspešno, nijedna test funkcija neće biti izvrašavana. Ako `init()` funkcija ne prode, njena prateća test funkcija se neće izvršiti, ali će se nastaviti sa sledećom.

3.1.1.2 Qt Subdirs projekat Drugi način da testiramo projekat je da kreiramo nov `Qt Subdirs` projekat u koji ćemo uključiti postojeći i novi `Qt Unit Test` projekat. Ovaj način organizacije je koristan ukoliko pišemo projekat od nule i želimo da usput pišemo i testove.

3.1.2 Pisanje testova

QtTest radni okvir pruža makroe za testiranje, i neke od njih ćemo koristiti da testiramo naš kalkulator:

- `QCOMPARE`
- `QVERIFY`
- `QVERIFY_EXCEPTION_THROWN`

3.1.3 Analiza pokrivenosti

Uz `gcc` kompajler dolazi i `gcov` alat za odredivanje pokrivenosti koda prilikom izvršavanja programa (engl. *code coverage*). Koristi se zajedno sa `gcc` kompajlerom da bi se analizirao program i utvrdilo kako se može kreirati efikasniji, brži kod i da bi se testovima pokrili delovi programa.

Alat `gcov` se može koristiti kao alat za profajliranje u cilju otkrivanja dela koda čija bi optimizacija najviše doprinela efikasnosti programa. Korišćenjem `gcov` -a možemo saznati koje su naredbe, linije, grane, funkcije itd. izvršene i koliko puta. Zarad lepše reprezentacije rezultata detekcije pokrivenosti koda izvršavanjem test primera, koristimo alat `lcov` .

Prilikom kompilacije neophodno je koristiti dodatne opcije kompajlera koje omogućavaju snimanje koliko je puta koja linija, grana i funkcija izvršena. Ti podaci se čuvaju u datotekama ekstenzije `.gcno` za svaku datoteku sa izvornim kodom. One će kasnije biti korišćene za kreiranje izveštaja o pokrivenosti koda.

```
$ g++ -g -Wall -fprofile-arcs -ftest-coverage -O0 main.cpp -o test
```

Alternativno:

```
g++ -g -Wall --coverage -O0 main.cpp -o test
```

Nakon izvršavanja test programa, informacije o pokrivenosti prilikom izvršavanja će biti u sačuvane u datoteci tipa `.gcda` , ponovo za svaku datoteku sa izvornim kodom. Pokrenimo alat `lcov` da bismo dobili čitljiviju reprezentaciju rezultata:

```
$ lcov --rc lcov_branch_coverage=1 -c -d . -o coverage-test.info
```

Opcija:

- `--rc lcov_branch_coverage=1` uključuje odredivanje pokrivenosti grana, koje podrazumevano nije uključeno
- `-c` kreiranje pokrivenosti
- `-d .` koristi tekući direktorijum, jer u našem slučaju on sadrži potrebne `.gcda` i `.gcno` datoteke
- `-o coverage-test.info` zadaje naziv izlazne datoteke sa izveštajem koji treba da ima ekstenziju `.info`

Možemo neke datoteke isključiti iz analize pokrivenosti. Na primer, biblioteke jezika koje ne testiramo mogu nam samo zamagliti pokrivenost koja nas zanima - pokrivenost funkcionalnosti koje testiramo.

```
$ lcov --rc lcov_branch_coverage=1 \
    -r coverage.info '/usr/*' '/opt/*' '*.moc' \
    -o coverage-filtered.info
```

Opcija `-r coverage.info` uklanja iz prethodno dobijenog izveštaja `coverage.info` izvorne fajlove koji odgovaraju nekom od šablona koji su navedeni kao argumenti opcije.

Alat `lcov` ima podalat `genhtml` koji na osnovu prethodno generisanog izveštaja pravi `.html` datototeke za jednostavniji pregled. Potrebno je izvršiti naredbu:

```
$ genhtml --rc lcov_branch_coverage=1 -o Reports coverage-filtered.info
```

Opcija `-o Reports` određuje naziv direktorijuma koji će biti kreiran i popunjen generisanim `.html` dokumentima.

Izveštaj možemo otvoriti u Web pretraživaču, npr.

```
$ firefox Reports/index.html
```

Ukoliko ne obrišemo `.gcda` datoteke od prethodnih pokretanja programa, prikaz pokrivenosti će uključiti sve, zbirno.

3.1.4 Kreiranje izveštaja uz pomoć skripte

Postupak kreiranja izveštaja, nakon kompilacije programa se može automati zovati pokretanjem bash skripte `generateCodeCoverageReport.sh` :

```
$ ./generateCodeCoverageReport.sh . test data
```

Argumenti:

- `.` - Direktorijum u kom se nalaze potrebne `.gcda` i `.gcno` datoteke i izvršni program. U našem slučaju to je tekući direktorijum.Inače bismo navodili relativnu ili apsolutnu putanju do potrebnog direktorijuma
- `test` - Drugi argument treba da je naziv izvršne verzije programa koja će se pokretati.
- `data` - Naziv direktorijuma u koji će alati `lcov` i `genhtml` upisivati svoje rezultate. Ukoliko se ne navede, sve će se upisivati u tekući direktorijum skripta. Ukoliko navedeni direktorijum ne postoji, biće kreiran.

Skript briše `.gcda` datoteke od prethodnih pokretanja programa. Prikazana pokrivenost je samo za poslednje pokretanje programa.

3.1.5 Integracija pokrivenosti u QtCreator

Ukoliko želimo da kreiramo pokrivenost koda prilikom izvršavanja postojećeg Qt projekta, potrebno je da definiciji projekta (fajl tipa `.pro`) dopišemo:

```
QMAKE_CXXFLAGS += -g -Wall -fprofile-arcs -ftest-coverage -O0
QMAKE_LFLAGS += -g -Wall -fprofile-arcs -ftest-coverage -O0

LIBS += \
    -lgcov
```

Opcije `-fprofile-arcs -ftest-coverage` i linkovanje sa `-lgcov` menja opcija kompajlera `--coverage`. Nakon pokretanja projekta, `.gcda` i `.gcno` datoteke i izvršivi program biće u direktorijumu gde se nalaze ostali artifakti prevođenja (podrazumevano u direktorijumu sa prefiksom `build_`). Izveštaj možemo potom napraviti prema ranije prikazanom postupku.

3.2 Java - JUnit, JaCoCo

JUnit je jedan od najpopularnijih radnih okvira za testiranje jedinica koda u programskom jeziku Java. Neke od osobina JUnit radnog okvira su jednostavnost pisanja testova uz bogat skup anotacija koje opisuju testove, kao i veoma velika podrška za najčešće situacije u procesu pisanja testova kao što su uslovno uključivanje/isključivanje testova na osnovu promenljivih iz okruženja, operativnog sistema, proizvoljnih predikata itd.

JUnit svoje artifakte isporučuje na Maven Central i moguće ga je uključiti u Maven (i Maven-kompatibilne) alate za prevođenje. U ovom primeru ćemo koristiti jedan drugi popularni alat za prevođenje pod imenom Gradle. Gradle koristi `build.gradle` fajl (slično kao što Maven koristi `pom.xml`) za definiciju projekta i zavisnosti, ali i podešavanja dodataka.

Kreirajmo novi projekat:

```
$ gradle init
```

Možemo u `build.gradle` dodati zavisnost za JUnit:

```
  | dependencies {
  |     ...
+ |     testImplementation "junit:junit:4.13"
  | }
```

Za praćenje pokrivenosti koda, koristićemo JaCoCo. JaCoCo možemo lako uključiti u projekat dodavanjem niske `'jacoco'` u spisak dodataka za projekat i dodatno konfigurisati izveštaj koji JaCoCo pravi:

```
  | plugins {
  |     id 'java'
+ |     id 'jacoco'
  | }
  |
  | ...
  |
+ | jacocoTestReport {
+ |    reports {
+ |        xml.required = false
+ |        csv.required = false
+ |        html.outputLocation = layout.buildDirectory.dir('jacocoHtml')
+ |    }
+ | }
+ |
+ | check.dependsOn jacocoTestReport
```

Nakon ovoga možemo iskoristiti sledeće komande:

- `gradle build` - prevođenje projekta
- `gradle test` - pokretanje testova jedinica koda
- `gradle jacocoTestReport` - kreiranje izveštaja o pokrivenosti

Izveštaji o pokrenutim testovima i statusu izvršavanja se mogu naći u direktorijumu `build/reports/tests/test` u HTML formatu. JaCoCo izveštaj o pokrivenosti se može naći u direktorijumu koji smo naveli u `build.gradle` fajlu - `build/jacocoHtml` , takođe u HTML formatu, kao što smo naveli.

Pregled nekih od korisnih anotacija JUnit radnog okvira:

- `@Test` - Radni okvir će pokrenuti ovaj metod automatski prilikom pokretanja testova.
- `@TestFactory` - Metod koji generiše testove u vremenu izvršavanja. Najčešće se koristi da pokrene nasumične testove ili testove bazirane na spoljnim podacima.
- `@DisplayName` - Čini izveštaje čitljivijim tako što testovima daje navedeno ime.
- `@BeforeAll` / `@BeforeEach` - Izvršava metod pre svih odnosno svakog testa.
- `@AfterAll` / `@AfterEach` - Izvršava metod posle svih odnosno svakog testa.
- `@Tag` - Dodaje oznaku testu radi kategorisanja testova u svite, npr. `@Tag("fast")` dodaje test u svitu sa oznakom `"fast"` .
- `@Disabled` - Isključuje test metod iz radnog okvira.
- `@Nested` - Koristi se u unutrašnjim klasama najčešće radi definisanja redosleda kojim se pokreću testovi.

Primeri korišćenja JUnit anotacija, test metoda i metoda pretpostavki:

```
import org.junit.jupiter.api.*;
public class AppTest {
    @BeforeAll
    static void setup(){
        System.out.println("Executes a method Before all tests");
    }
    @BeforeEach
    void setupThis(){
        System.out.println("Executed Before each @Test method " +
                           "in the current test class");
    }
    @AfterEach
    void tearThis(){
        System.out.println("Executed After each @Test method " +
                           "in the current test class");
    }
    @AfterAll
    static void tear(){
        System.out.println("Executes a method After all tests");
    }
}
```

```
Assertions.assertAll("heading",
    () -> assertTrue(true),
    () -> assertEquals("expected", objectUnderTest.getSomething()
);
```

```
@TestFactory
Stream dynamicTests(MyContext ctx) {
    // Generates tests for every line in the file
    return Files.lines(ctx.testDataFilePath)
                .map(l -> dynamicTest("Test: " + l,
                                      () -> assertTrue(runTest(l))
                );
}
```

```
@Test
void exampleTest() {
    Assertions.assertTrue(trueBool);
    Assertions.assertFalse(falseBool);
    Assertions.assertNotNull(notNullString);
    Assertions.assertNull(notNullString);
    Assertions.assertNotSame(originalObject, otherObject);
    Assertions.assertEquals(4, 4);
```

```
    Assertions.assertNotEquals(3, 2);
    Assertions.assertArrayEquals(
        new int[] { 1, 2, 3 },
        new int[] { 1, 2, 3 },
        "Array Equal Test"
    );
    Iterable<Integer> listOne = new ArrayList<>(Arrays.asList(1,2,3,4));
    Iterable<Integer> listTwo = new ArrayList<>(Arrays.asList(1,2,3,4));
    Assertions.assertIterableEquals(listOne, listTwo);
    Assertions.assertTimeout(Duration.ofMillis(100), () -> {
        Thread.sleep(50);
        return "result";
    });
    Throwable exception = Assertions.assertThrows(
        IllegalArgumentException.class,
        () -> throw new IllegalArgumentException("error message");
    );
    Assertions.fail("not found good reason to pass");
}
```

```
@Test
void testAssumption() {
    System.setProperty("prop", "foo");
    Assumptions.assumeTrue("foo".equals(System.getProperty("prop")));
}
```

```
@Test
@EnabledForJreRange(min = JRE.JAVA_8, max = JRE.JAVA_11)
public void test1()
{
    System.out.println("Will run only on JRE between 8 and 11");
}

@Test
@EnabledOnJre({JRE.JAVA_8, JRE.JAVA_11})
public void test2()
{
    System.out.println("Will run only on JRE 8 and 11");
}

@Test
@DisabledForJreRange(min = JRE.JAVA_8, max = JRE.JAVA_11)
public void test3()
{
```

```
    System.out.println("Will NOT run on JRE between 8 and 11");
}

@Test
@DisabledOnJre({JRE.JAVA_8, JRE.JAVA_11})
public void test4()
{
    System.out.println("Will NOT run on JRE 8 and 11");
}
```

```
@Test
@EnabledOnOs({OS.LINUX, OS.WINDOWS})
void onLinuxOrWindows() {
    System.out.println("Will run on Linux or Windows.");
}

@Test
@DisabledOnOs({OS.WINDOWS, OS.SOLARIS, OS.MAC})
void notOnWindowsOrSolarisOrMac() {
    System.out.println("Won't run on Windows, Solaris or MAC!");
}
```

```
@Test
@EnabledIf("myCustomPredicate")
void enabled() {
    assertTrue(true);
}

@Test
@DisabledIf("myCustomPredicate")
void disabled() {
    assertTrue(true);
}

boolean myCustomPredicate() {
    return true;
}
```

```
@Test
@EnabledIfEnvironmentVariable(named = "ENV", matches = ".*oracle.*")
public void executeOnlyInDevEnvironment() {
    return true;
}
```

```java
@Test
@DisabledIfEnvironmentVariable(named = "ENV", matches = ".*mysql.*")
public void disabledOnProdEnvironment() {
    return true;
}

@Test
@EnabledIfSystemProperty(named = "my.property", matches = "prod*")
public void onlyIfMyPropertyStartsWithProd() {
    return true;
}
```

3.3 C# (.NET) - xUnit, NUnit

U okviru .NET ekosistema postoji bogat skup radnih okvira za testiranje jedinica koda. Neki od njih olakšavaju repetitivno pisanje testova ubrizgavanjem vrednosti u šablone testova (tzv. *teorije*), ili pružaju interfejs za pisanje testova navođenjem ograničenja. Primeri ovakvih radnih okvira koje ćemo razmatrati su xUnit i NUnit. Oba radna okvira podržavaju sve jezike u okviru .NET ekosistema (C#, F#, VB.NET, ...).

3.3.1 xUnit

Osim jednostavnog interfejsa za pisanje testova nalik na JUnit u programskom jeziku Java, gde se testovi markiraju odgovarajućim anotacijama (što je i slučaj u xUnit radnom okviru markiranjem metoda atributom `[Fact]`), jedna od najpopularnijih osobina xUnit radnog okvira je mogućnost pisanja *teorija* - šablona za testove. Umesto da pišemo isti skup pod-testova iznova i iznova za različite podatke (ili umesto da ih izdvajamo u funkcije), možemo zakačiti atribut [1] `[Theory]`, a unutar atributa `[InlineData]` definisati podatke koji će biti ulaz za test:

```
public class ParameterizedTests
{
    public bool SampleAssert1(int a, int b, int c, int d)
    {
        return (a + b) == (c + d);
    }

    public bool SampleAssert2(int a, int b, int c, int d)
    {
        return (a + c) == (b + d);
    }
```

[1] Atributi u programskom jeziku C# su donekle ekvivalentni anotacijama u programskom jeziku Java. Za razumevanje primera nije neophodno duboko poznavanje koncepta atributa.

```
// Regular xUnit test case
// Sub-optimal (repeated asserts)
[Fact]
public void SampleFact()
{
    Assert.True(SampleAssert1(4, 4, 4, 4));
    Assert.True(SampleAssert2(4, 4, 4, 4));

    Assert.True(SampleAssert1(3, 2, 2, 3));
    Assert.True(SampleAssert2(3, 2, 2, 3));

    Assert.True(SampleAssert1(7, 0, 0, 7));
    Assert.True(SampleAssert2(7, 0, 0, 7));

    Assert.True(SampleAssert1(0, 7, 7, 0));
    Assert.True(SampleAssert2(0, 7, 7, 0));
}

// Regular xUnit test case
// No repeated asserts but requires a local method
[Fact]
public void SampleFact()
{
    Assert.True(PerformAsserts(4, 4, 4, 4));
    Assert.True(PerformAsserts(3, 2, 2, 3));
    Assert.True(PerformAsserts(7, 0, 0, 7));
    Assert.True(PerformAsserts(0, 7, 7, 0));

    void PerformAsserts(int a, int b, int c, int d)
    {
        Assert.True(SampleAssert1(a, b, c, d));
        Assert.True(SampleAssert2(a, b, c, d));
    }
}

// Using Theory and InlineData
// Optimal solution, replaces above patterns
[Theory]
[InlineData(4, 4, 4, 4)]
[InlineData(3, 2, 2, 3)]
[InlineData(7, 0, 0, 7)]
[InlineData(0, 7, 7, 0)]
```

```
    public void SampleTheory(int a, int b, int c, int d)
    {
        Assert.True(SampleAssert1(a, b, c, d));
        Assert.True(SampleAssert2(a, b, c, d));
    }

    // There exist special "InlineData" variants, for example "SqlServerData"
    [Theory]
    [SqlServerData("(local)",
                   "TestDatabase",
                   "select FirstName, LastName from Users")]
    public void SqlServerTests(string FirstName, string LastName)
    {
        Assert.Equal("Peter Beardsley", $"{FirstName} {LastName}");
    }
}
```

Druga popularna odlika xUnit radnog okvira je jednostavna izolacija test metoda. To se postiže kreiranjem zasebne instance test klase za svaki test metod. Za razliku od drugih popularnih radnih okvira, xUnit ne daje interfejs za markiranje metoda sa ciljem pokretanja tog metoda pre ili posle jednog ili svih testova, već se na osnovu izolacije testova po instanci klase, piše čitljiviji kod koji u konstruktoru i destruktoru klase vrši odgovarajuću pripremu odnosno čišćenje pre odnosno posle pokretanja testova.

Primeri korišćenja xUnit radnog okvira su preuzeti iz zvaničnog repozitorijuma sa primerima i mogu se naći kao git podmodul u okviru repozitorijuma sa materijalima.

3.3.2 NUnit

Glavna odlika NUnit radnog okvira je model ograničenja (constraint model). Takav model u radnom okviru pruža samo jedan metod za implementaciju testova. Logika potrebna za testiranje se kodira u objektu ograničenja koji se prosleđuje toj metodi:

```
Assert.That(myString, Is.EqualTo("Hello"));
Assert.That(myString, Is.Not.EqualTo("Bello"));
```

Primeri korišćenja NUnit radnog okvira su preuzeti iz zvaničnog repozitorijuma sa primerima i mogu se naći kao git podmodul u okviru repozitorijuma sa materijalima.

4 Testiranje pomoću objekata imitatora (engl. Mock testing)

Prilikom pisanja jediničnih testova fokusiramo se samo na jednu funkciju i ispitujemo njeno ponašanje u kontrolisanom okruženju. Sadržaj testa je uvek skup inicijalizacija okruženja, pokretanje funkcije koju testiramo i zatim poredenje dobijenog i očekivanog rezultata. Nekada je kod takav da je nemoguće napraviti kontrolisano okruženje za testiranje, npr. korišćenje sistemskih poziva, podataka iz baze podataka, mrežna komunikacija i sl. U tim situacijama pribegava se pisanju klasa koje imitiraju realne objekte u svrhu testiranja.

Na primer, funkcija može da odbrađuje podatke iz datoteke i uzima naziv datoteke kao ulazni parametar. Superiornije rešenje je prepraviti funkciju tako da radi sa ulaznim tokom do fajla koji treba da obradi. Funkcija potom neće raditi sve stvari kao ranije - otvarati datoteku i obradivati podatke.

U jediničnom testu, objekti imitatori mogu da imitiraju ponašanje kompleksnog stvarnog objekta. Vrlo su korisni u situacijama kada stvarni objekat nije praktično ili je nemoguće uklopiti u jedinični test. Objekat imitator se obično koristi ukoliko stvarni objekat ima neki od narednih karakteristika:

- obezbeđuje nedeterministički rezultat (npr. trenutno vreme, trenutnu temperaturu, ...)
- ima stanja koja je teško kreirati ili reprodukovati, (npr. greška u mrežnoj komunikaciji)
- spor je (npr. baza podataka, koja bi pre svakog testa morala biti inicijalizovana)
- ne postoji još uvek, ili može promeniti ponašanje u budućnosti
- morao bi da dobije nove informacije i metode da bismo mogli da ga koristimo za testiranje, a inače mu nisu potrebne

Objekti imitatori treba da imaju isti interfejs kao stvarni objekti koje imitiraju. Tako omogućavaju da objekat koji ih koristi ne pravi razliku izmedu stvarnog ili imitator objekta. Mnogi radni okviri za objekte imitatore omogućavaju da se samo naglasi objekti koje klase se imitiraju i potom da programer zada koji metodi se pozivaju na objektu imitatoru, kojim redom i sa kojim parametrima, kao i koja vrednost se očekuje kao povratna. Na taj način se mogu imitirati ponašanja kompleksnih objekata (npr. socket) i omogućiti da programer testira da li se objekat ponaša korektno sa svim različitim stanjima. To je daleko jednostavniji postupak nego izazivanje svih situacija na stvarnom objektu.

Rad sa objektima imitatorima obično obuhvata sledeće korake:

- kreiranje interfejsa za klasu koju bi trebalo testirali
- kreiranje klase imitatora ručno ili pomoću nekog radnog okvira:
  - C++ - `FakeIt` , `CppUMock` (unutar `CppUnit`), `GoogleMock`

    - Java - `Mockito` , `JMock` , `EasyMock` , `PowerMock`
    - .NET - `Moq`
- priprema koda koji će se testirati na objektu imitatoru;
- pisanje testa koji će koristiti objekat imitator umesto stvarnog objekta

Unutar testa je potrebno: – kreirati instancu objekta klase imitatora – podesiti ponašanje i očekivanja od objekta imitatora – pokrenuti kod koji će koristiti objekat imitator – po izvršavanju, porediti dobijene i očekivane vrednosti (ovaj korak obično izvršava radni okvir prilikom uništavanja objekta imitatora)

4.1 C++ - ručno kreiranje objekata imitatora

Dobili smo zadatak da u igru koju razvijamo dodamo novu funkcionalnost koja meri koliko je igrač aktivno igrao igru. Vreme provedeno u glavnom meniju i pauze ne treba da se uključe u vreme igranja. U tu svrhu kreiramo jednostavnu klasu `play_time` koja ima metode za pokretanje i zaustavljanje sesije i vraća ukupno vreme igranja.

Testiranje ove jednostavne klase zahteva par koraka:

1. kreiranje instance klase `play_time`
2. započinjanje sesije
3. uspavljivanje programa na neko vreme
4. zaustavljanje sesije
5. pozivanje metoda za dobijanje ukupno vreme igranja
6. poređenje dobijene vrednosti sa vremenom uspavljivanja programa

Problem je u tome što program treba da spava neko vreme. Nije test sam po sebi problem, klasa `play_time` koja zavisi od sistemskog sata. Rešenje je da generalizujemo konstruktor klase i da eksplicitno naglasimo zavisnost klase od sistemskog sata. Sve dok `play_time` dobija trenutno vreme nekako, pravi izvor nam nije presudno bitan.

Kreirajmo interfejs `second_clock` . Menjamo konstruktor klase `play_time` tako da kao argument dobija instancu `second_clock` interfejsa. Time je svakom jasno da naša klasa zavisi od sata. Menjamo i metode za pokretanje i zaustavljanje sesije, jer sada treba da zavise od parametra klase, sata.

Kreirajmo klasu `system_clock` koja će implementirati već kreiran interfejs `second_clock` . Kada želimo da objekat klase `play_time` koristi sistemski sat, konstruktoru ćemo slati objekat klase `system_clock` .

Kreirajmo sada klasu imitatora `mock_clock` koja će da odgovara ponašanju sata bez baterija. Uvek će pokazivati podešeno vreme. Implementiramo konstruktor, metode `get` i `set` za postavljanje vremena.

Prilikom testiranja, koristimo instancu `mock_clock` prilikom konstrukcije instance klase `play_time` . Umesto da uspavamo program, pomerićemo vreme na satu za neki interval i očekujemo da isti interval vrati i metoda `played_time` .

4.2 C# - Moq

Moq (izgovara se *Mock-you* ili jednostavno *Mok*) je vodeći radni okvir za pisanje objekata imitatora u .NET ekosistemu. Dizajniran je da bude veoma praktičan i bezbedan. Neke od osobina:

- Jako tipiziran (ne koriste se niske za definisanje očekivanja, povratne vrednosti metoda su specifični tipovi a ne opšti `object` tip)
- Jednostavni idiomi - konstrukcija imitatora, podešavanje ponašanja imitatora, očekivanja
- Granularna kontrola ponašanja imitatora
- Imitira i interfejse i klase
- Presretanje događaja nad imitatorima

U fajlu `MockExamples.cs` imamo definicije nekoliko interfejsa i klasa:

- `IBookService` - predstavlja interfejs servisa koji koristimo da dovučemo informacije o knjigama na osnovu kategorije ili ISBN
- `IEmailSender` - predstvlja interfejs servisa koji koristimo da pošaljemo e-mail
- `AccountsService` - koristi navedene servise
- `SampleAccountsServiceTests` - testovi za `AccountsService`

Pošto `IBookService` može na proizvoljan način da dovlači informacije o knjigama (REST-ful API, baza podataka itd.), jasno je da ne želimo da je testiramo direktno. Štaviše, naši testovi se tiču `AccountsService` klase, a ne klasa koje ona koristi, dakle podrazumevamo da se implementacije `IBookService` i `IEmailSender` ispravno ponašaju. Tu pretpostavku implementiramo pomoću objekata imitatora.

Posmatrajmo klasu `SampleAccountsServiceTests` . Naredni primer prikazuje jednostavan idiom za korišćenje Moq radnog okvira:

```
public void GetAllBooksForCategory_returns_list_of_available_books()
{
    // 1
    var bookServiceStub = new Mock<IBookService>();

    // 2
    bookServiceStub
        .Setup(x => x.GetBooksForCategory("UnitTesting"))
        .Returns(new List<string> {
            "The Art of Unit Testing",
            "Test-Driven Development",
            "Working Effectively with Legacy Code"
        });

    // 3
```

```
    var accountService = new AccountService(bookServiceStub.Object, null);
    var result = accountService.GetAllBooksForCategory("UnitTesting");

    // 4
    Assert.Equal(3, result.Count());
}
```

Sličan imitator možemo napisati i za `IEmailService` , ukoliko želimo da testiramo metod `SendEmail` :

```
public void SendEmail_sends_email_with_correct_content()
{
    // 1
    var emailSenderMock = new Mock<IEmailSender>();

    // 2
    var accountService = new AccountService(null, emailSenderMock.Object);
    // 3

    accountService.SendEmail("test@gmail.com", "Test - Driven Development");
    // 4
    emailSenderMock.Verify(
        x => x.SendEmail(
            "test@gmail.com",
            "Awesome Book",
            $"Hi,\n\nThis book is awesome: Test - Driven Development"
        ),
        Times.Once
    );
}
```

Moguće je modelirati višestruke pozive. U primeru koji sledi zadajemo povratne vrednosti za prva četiri poziva metoda `GetISBNFor` , poziv metoda `GetBooksForCategory` podešavamo tako da baca izuzetak:

```
bookServiceStub
    .SetupSequence(x => x.GetISBNFor(It.IsAny<string>()))
    .Returns("0-9020-7656-6")   //returned on 1st call
    .Returns("0-9180-6396-5")   //returned on 2nd call
    .Returns("0-3860-1173-7")   //returned on 3rd call
    .Returns("0-5570-1450-6")   //returned on 4th call
    ;

bookServiceStub
    .Setup(x => x.GetBooksForCategory(It.IsAny<string>()))
```

```
    .Throws<InvalidOperationException>();
```

Za poznavaoce asinhronog šablona zasnovanog na zadacima (engl. Task-Based Asynchronous Pattern, skr. TAP) u programskom jeziku C#, mogu od značaja biti i asinhroni primeri korišćenja Moq radnog okvira:

```
httpClientMock
    .Setup(x => x.GetAsync(It.IsAny<string>()))
    .ReturnsAsync(true);
```

4.3 C# - ”imitatori” baza podataka

Objekti imitatori, iako korisni, ne mogu zameniti ponašanje pravih sistema za upravljanje bazama podataka. Iako je testove potrebno izvršiti nad produkcijskom bazom podataka pre isporučivanja aplikacije, neefikasno je te testove pokretati tokom razvoja aplikacije. Želeli bismo da testove pokrenemo u okruženju što sličnijem pravoj bazi podataka, ali ne bismo da gubimo na efikasnosti - potrebno je bazu pripremiti za svaki test posebno, što uključuje veliki broj upita. S druge strane, možda smo već napisali imitatore i zadovoljni smo ponašanjem naših servisa, ali bismo da testiramo logiku ostvarivanja odnosno raskidanja veze sa bazom. U ovakvim slučajevima, ali i mnogim drugim, izvršavanje testova nad ”pravom” bazom podataka je nešto što bismo voleli da imamo. Međutim, voleli bismo da to sve bude transparentno i da ne zahteva nikakve izmene u kodu, ali i takođe dovoljno efikasno tako da ne odlazi dosta vremena na izvršavanje testova.

Objektno-relacioni maperi (engl. Object Relational Mappers, skr. ORM) se često koriste radi apstrahovanja specifilnosti konkretnog sistema za upravljanje bazama podataka. Neki poznati ORM radni okviri su Hibernate (Java), Entity Framework (.NET). U ovom primeru ćemo iskoristiti pogodnosti koje Entity Framework i SUBP SQLite pružaju, kako bismo testove izvršili nad privremenom bazom podataka lociranoj u radnoj memoriji.

Postoje sistemi za upravljanje bazama podataka koji mogu da operišu sa privremenim bazama podataka skladištenim u radnoj memoriji. Jedan od takvih SUBP je SQLite, koji pruža tzv. *in-memory database provider* upravo za ove svrhe. SQLite čuva baze podataka u fajlovima na fajl sistemu. Da bismo kreirali bazu podataka u memoriji, potrebno je da u nisku za konekciju na bazu ubacimo `:memory:` na mesto gde bi inače išla putanja do fajla na fajl sistemu gde bi se baza čuvala. Privremena baza živi u radnoj memoriji sve dok postoji otvorena veza ka njoj, drugim rečima raskid veze povlači brisanje baze podataka - što je idealno za naše testove jer svakako bismo da testove pokrećemo u izolovanom okruženju.

Klasa `SampleDbContext` predstavlja kontekst veze s bazom podataka. Taj kontekst u sebi ima svojstva tipa `DbSet<T>` koja će se mapirati u odgovarajuće tabele. Pošto koristimo Entity Framework, možemo ga konfigurisati tako da, na osnovu konfiguracije

koju korisnik navodi, koristimo odgovarajući zadnji deo (engl. back-end) koji će komunicirati sa odgovarajućim SUBP. Recimo da želimo da podržimo naredne SUBP:

```
public enum Provider
{
    Sqlite = 0,
    PostgreSql = 1,
    SqlServer = 2,
    SqliteInMemory = 3
}
```

Prilikom konfigurisanja EF radnog okvira, možemo odabrati odgovarajući zadnji deo:

```
switch (this.Provider) {
    case Provider.PostgreSql:
        optionsBuilder.UseNpgsql(this.ConnectionString);
        break;
    case Provider.Sqlite:
    case Provider.SqliteInMemory:
        optionsBuilder.UseSqlite(this.ConnectionString);
        break;
    case Provider.SqlServer:
        optionsBuilder.UseSqlServer(this.ConnectionString);
        break;
    default:
        throw new NotSupportedException("Provider not supported!");
}
```

Možemo implementirati provajder baze za testove tako što prosledimo odgovarajuču nisku za konekciju ka bazi podataka:

```
public class TestDbProvider
{
    public readonly string ConnectionString { get; }
    public readonly SqliteConnection DatabaseConnection { get; }

    public TestDbProvider()
    {
        ConnectionString = "DataSource=:memory:;foreign keys=true;";
        DatabaseConnection = new SqliteConnection(ConnectionString);
    }

    private void CreateContext()
    {
        var options = new DbContextOptionsBuilder<SampleDbContext>()
            .UseSqlite(DatabaseConnection)
```

```
            .Options;

        return new SampleDbContext(
            SampleDbContext.Provider.SqliteInMemory,
            ConnectionString,
            options
        );
    }

    // ...
}
```

Za izvršavanje testova je neophodno da:

- ostvarimo konekciju ka bazi
- ubacimo podatke u bazu (pošto se baza uvek briše nakon raskidanja konekcije)
- odradimo logiku koju test treba da proveri
- proverimo rezultat
- raskinemo vezu sa bazom

Da bismo smanjili ponavljanje koda, dodaćemo metod `SetupAlterAndVerify` koji će da primi funkcije:

- `void Setup(SampleDbContext ctx)` - priprema bazu podataka za test
- `void Alter(SampleDbContext ctx)` - izvršava logiku koju test treba da proveri
- `void Verify(SampleDbContext ctx)` - testira rezultujuće stanje baze podataka

```
public void SetupAlterAndVerify(
    Action<SampleDbContext>? setup,
    Action<SampleDbContext>? alter,
    Action<SampleDbContext>? verify)
{
    DatabaseConnection.Open();
    try {
        this.CreateDatabase();
        this.SeedDatabase();

        if (setup is not null) {
            using SampleDbContext context = this.CreateContext();
            setup(context);
            context.SaveChanges();
        }

        if (alter is not null) {
```

```
            using SampleDbContext context = this.CreateContext();
            alter(context);
            context.SaveChanges();
        }

        if (verify is not null) {
            using SampleDbContext context = this.CreateContext();
            verify(context);
        }
    } finally {
        DatabaseConnection.Close();
    }
}
```

Primetimo da, iako kontekst kreiramo više puta, veza ka bazi i dalje ostaje aktivna dok se ne pozove `DatabaseConnection.Close()` metod. Kreiranje zasebnih konteksta je poželjno pošto bismo da sačuvamo stanje baze nakon svakog koraka (setup, alter, verify). Tip `Action<T1, T2, ..., Tn>` u programskom jeziku C# predstavlja funkciju: `void f(T1, T2, ... Tn)` . Oznaka `?` je skraćenica za tip `Nullable<T>` koji predstavlja opcioni tip. Drugim rečima, metodi `SetupAlterAndVerify` prosleđujemo opcione akcije, i možemo da odlučimo da naš test ne mora da ima neku od njih (tako što prosledimo `null` , stoga provere u telu funkcije pre poziva funkcija `setup` , `alter` i `verify`). Ključna reč `using` je deo upravljanja resursa nad objektima koji implementiraju `IDisposable` interfejs u programskom jeziku C#, sa ciljem da se automatski počisti objekat nakon što kontrola toka izađe iz opsega u kojem je vidljiv (nešto nalik na *try-with-resources* šablon u programskom jeziku Java). Drugim rečima, automatski će se pozvati metod `IDisposable.Dispose()` nad kontekstom koji je definisan naredbom koja je kvalifikovana ključnom rečju `using` (interfejs `IDisposable` je implementiran u natklasi klase `SampleDbContext` koja dolazi iz EF radnog okvira).

Testove onda možemo veoma jednostavno pisati:

```
[Test]
public void SampleTest()
{
    this.DbProvider.SetupAlterAndVerify(
        // setup
        ctx => ctx.Students.Clear(),

        // alter
        ctx => Service.PerformLogic(ctx.Students),

        // verify
        ctx => Assert.That(ctx.Students, Is.Not.Empty)
```

```
    );
}
```

Za poznavaoce asinhronog šablona zasnovanog na zadacima (engl. Task-Based Asynchronous Pattern, skr. TAP) u programskom jeziku C#, mogu od značaja biti i asinhrona varijanta metoda `SetupAlterAndVerify` :

```
public async Task SetupAlterAndVerifyAsync(
    Func<SampleDbContext, Task>? setup,
    Func<SampleDbContext, Task>? alter,
    Func<SampleDbContext, Task>? verify)
{
    DatabaseConnection.Open();
    try {
        this.CreateDatabase();
        this.SeedDatabase();

        if (setup is not null) {
            await using SampleDbContext context = this.CreateContext();
            await setup(context);
            await context.SaveChangesAsync();
        }

        if (alter is not null) {
            await using SampleDbContext context = this.CreateContext();
            await alter(context);
            await context.SaveChangesAsync();
        }

        if (verify is not null) {
            await using SampleDbContext context = this.CreateContext();
            await verify(context);
        }
    } finally {
        DatabaseConnection.Close();
    }
}
```

5 Profajliranje

Profajliranje je vrsta dinamičke analize programa (program se analizira tokom izvršavanja) koja se sprovodi kako bi se izmerila, npr. količina memorije koju program zauzima, vreme koje program provodi u određenim funkcijama, iskorišćenost keša itd. Programi koji vrše profajliranje se zovu *profajleri*. Na ovom kursu će biti reči o popularnim profajlerima, njihovim prednosima i manama, uz primere upotrebe.

Profajliranje bi trebalo da nam da jasnu informaciju o tome da li imamo značajna uska grla u kodu. Ako primetimo da neka funkcija uzima 60% vremena izvršavanja, onda je ona glavni kandidat za optimizaciju. Sa druge strane, ako nemamo nijednu funkciju koja troši više od par procenata vremena onda treba pažnju usmeriti na druge pristupe poboljšanja performansi aplikacije (brži hardver, bolja arhitektura/dizajn aplikacije, paralelizacija) ili´ce biti potrebno da se optimizuje mnogo koda da bi se napravila ve´ca razlika.

Profajleri mogu obezbediti informaciju o tome koliko je vremena potrošeno u svakoj funkciji i u pozivima drugih funkcija, pa i koliko je potrošeno u svakoj liniji koda. Te informacije se, za neke alate kao što su Cachegrind i Callgrind koji generišu izlaz koji ima dosta zajedničkog, mogu prikazati bilo kroz alate koje Valgrind pruža ili kroz specijalizovane alate kao što je *Kcachegrind*.

5.1 Valgrind

Valgrind je platforma za pravljenje alata za dinamičku analizu mašinskog koda, snimljenog ili kao objektni modul (nepovezan) ili kao izvršivi program (povezan). Postoje Valgrind alati koji mogu automatski da detektuju probleme sa memorijom i procesima.

Valgrind se može koristiti i kao alat za pravljenje novih alata. Valgrind distribucija, između ostalih, uključuje sledeće alate: detektor memorijskih grešaka (Memcheck), detektor grešaka u višenitnim programima (Hellgrind i DRD), optimizator keš memorije i skokova (Cachegrind), generator grafa skrivene memorije i predikcije skoka (Callgrind) i optimizator korišćenja dinamičke memorije (Massif).

5.1.1 Struktura i upotreba Valgrind alata

Alat Valgrind se sastoji od alata za dinamičku analizu koda koji se kreira kao dodatak pisan u C programskom jeziku na jezgro Valgrinda. Jezgro Valgrinda omogućuje izvršavanje klijentskog programa, kao i snimanje izveštaja koji su nastali prilikom analize samog programa. Alati Valgrinda koriste metodu bojenja vrednosti. Oni svaki registar i memorijsku vrednost boje (zamenjuju) sa vrednošću koja govori nešto dodatno o originalnoj vrednosti. Proces rada svakog alata Valgrinda je u osnovi isti.

Valgrind deli originalni kod u sekvence koje se nazivaju osnovni blokovi. Osnovni blok je pravolinijska sekvenca mašinskog koda, na čiji se početak skače, a koja se završava

skokom, pozivom funkcije ili povratkom u funkciju pozivaoca. Svaki kod programa koji se analizira ponovo se prevodi na zahtev, pojedinačno po osnovnim blokovima, neposredno pre izvršavanja samog bloka. Veličina osnovnog bloka je ograničena na maksimalno šezdeset mašinskih instrukcija.

Alat analizira dobijen kod i vrši translaciju - proces koji se sastoji od sledećih koraka:

1. Disasembliranje (razgradnja) - prevodenje mašinskog koda u ekvivalentni interni skup instrukcija koje se nazivaju međukod instrukcije. U ovoj fazi međukod je predstavljen stablom. Ova faza je zavisna od arhitekture na kojoj se program izvršava.
2. Optimizacija 1 - prva faza optimizacije linearizuje prethodno izgradeni međukod. Primenjuju se neke standardne optimizacije programskih prevodilaca kao što su uklanjanje redudantnog koda, eliminacija podizraza itd.
3. Instrumentacija - Blok međukoda se prosleduje alatu, koji može proizvoljno da ga transformiše. Prilikom instrumentacije alat u zadati blok dodaje dodatne međukod operacije, kojima proverava ispravnost rada programa. Treba napomenuti da ubačene instrukcije ne narušavaju konzistentno izvršavanje originalnog koda.
4. Optimizacija 2 - jednostavnija faza optimizacije od prve. Uključuje izračunavanje matematičkih izraza koji se mogu izvršiti pre faze izvršavanja i uklanjanje mrtvog koda.
5. Izgradnja stabla - linearizovani međukod se konvertuje natrag u stablo radi lakšeg izbora instrukcija.
6. Odabir instrukcija - Stablo međukoda se konvertuje u listu instrukcija koje koriste virtualne registre. Ova faza se takode razlikuje u zavisnosti od arhitetkure na kojoj se izvršava.
7. Alokacija registara - zamena virtualnih registara stvarnim. Po potrebi se uvode prebacivanja u memoriju. Ne zavisi od platforme. Koristi se poziv funkcija koje pronalaze iz kojih se registara vrši čitanje i u koje se vrši upis.
8. Asembliranje - kodiranje izabranih instrukcija na odgovarajući način i smeštaju u blok memorije. Ova faza se takode razlikuje u zavisnosti od arhitekture na koji se izršava.

Jezgro Valgrinda troši najviše vremena na sam proces pravljenja, pronalaženja i izvršavanja translacije (originalni kod se nikad ne izvršava). Treba napomenuti da sve ove korake osim instrumentacije izvršava jezgro Valgrinda dok samu instrumentaciju izvršava odredeni alat koji smo koristili za analizu izvornog koda.

Sve međukod instrukcije, originalne i dodate translacijom, prevode se u mašinske reči ciljne platforme i snimaju u prevedeni osnovni blok. Alat u originalni kod umeće operacije u svrhu instrumentalizacije, zatim se takav kod prevodi.

Prilikom analize programa alatom Valgrind izvršavanje programa traje 20-100 puta duže nego inače. Analiza prevedenog programa Valgrindom, vrši sledećom naredbom:

```
valgrind --tool=alat [argumenti alata] ./a.out [argumenti za a.out]
```

ili pokretanjem Valgrind memory analizer-a iz QtCreator-a za aktivan projekat.

Ukoliko se ne zada vrednost argumenta `--tool` podrazumeva se `memcheck` .

Prve tri linije izlazne poruke štampaju se prilikom pokretanja bilo kog alata koji je u sklopu Valgrinda. U nastavku se prikazuju poruke o greškama koje je alat pronašao u programu. Zatim sledi izlaz samog programa, praćen sumiranim izveštajem o greškama.

Nekada informacija koja se dobije o grešci nije dovoljno detaljna da se u hiljadama linija koda nade pravo mesto. Da bismo u okviru poruke o grešci imali i informaciju o liniji koda u kojoj je detektovana potrebno je da program prevedemo sa debug simbolima (opcija `-g` za `gcc`). Da se ne bi dogodilo da se ne prijavljuje tačna linija u kojoj je detektovana greška preporučuje se da se isključe optimizacije (opcija `-O0` za `gcc`).

5.1.2 Memcheck

Memcheck detektuje memorijske greške korisničkog programa. Kako ne vrši analizu izvornog koda već mašinskog, Memcheck ima mogućnost analize programa pisanom u bilo kom programskom jeziku. Za programe pisane u jezicima C i C++ detektuje sledeće probleme:

- Korišćenje nedefinisanih vrednosti, vrednosti koje nisu inicijalizovane ili koje su izvedene od drugih nedefinisanih. vrednosti. Problem se detektuje tek kada su upotrebljene.
- Čitanje ili pisanje u nedopuštenu memoriju na hipu, steku, bilo da je potkoračenje ili prekoračenje dozvoljene memorije ili pristupanje već oslobodenoj memoriji.
- Neispravno oslobadanje memorije na hipu, npr. duplo oslobadanje memorije na hipu ili neupareno korišćenje funkcija `malloc/new/new[]` i `free/delete/delete[]` .
- Poklapanje argumenata `src` i `dest` funkcije `memcpy` i njoj sličnim.
- Prosledivanje loših vrednosti za veličinu memorijskog prostora funkcijama za alokaciju memorije, npr. negativnih.
- Curenje memorije, npr. gubitak pokazivača na alociran prostor.

5.1.2.1 Korišćenje nedefinisanih vrednosti Program `01_uninitialized.c` koristi nedefinisanu promenljivu `x` . Prevedimo kod i pokrenimo `memcheck` :

```
$ gcc -g -O0 -Wall 01_uninitialized.c -o 1
$ valgrind ./1
```

Nedefinisana promenljiva može više puta da se kopira. Memcheck prati i beleži podatke o tome, ali ne prijavljuje grešku. U slučaju da se nedefinisane vrednosti koriste tako da od te vrednosti zavisi dalji tok programa ili ako je potrebno prikazati vrednosti nedefinisane promeljive, Memcheck prijavljuje grešku.

```
==11003== Memcheck, a memory error detector
==11003== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==11003== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==11003== Command: ./1
==11003==
==11003== Conditional jump or move depends on uninitialised value(s)
==11003== at 0x48DEE40: __vfprintf_internal (vfprintf-internal.c:1644)
==11003== by 0x48C98D7: printf (printf.c:33)
==11003== by 0x109162: main (01_uninitialized.c:7)
==11003==
==11003== Use of uninitialised value of size 8
==11003== at 0x48C332E: _itoa_word (_itoa.c:179)
==11003== by 0x48DE9EF: __vfprintf_internal (vfprintf-internal.c:1644)
==11003== by 0x48C98D7: printf (printf.c:33)
==11003== by 0x109162: main (01_uninitialized.c:7)
==11003==
==11003== Conditional jump or move depends on uninitialised value(s)
==11003== at 0x48C3339: _itoa_word (_itoa.c:179)
==11003== by 0x48DE9EF: __vfprintf_internal (vfprintf-internal.c:1644)
==11003== by 0x48C98D7: printf (printf.c:33)
==11003== by 0x109162: main (01_uninitialized.c:7)
==11003==
==11003== Conditional jump or move depends on uninitialised value(s)
==11003== at 0x48DF48B: __vfprintf_internal (vfprintf-internal.c:1644)
==11003== by 0x48C98D7: printf (printf.c:33)
==11003== by 0x109162: main (01_uninitialized.c:7)
==11003==
==11003== Conditional jump or move depends on uninitialised value(s)
==11003== at 0x48DEB5A: __vfprintf_internal (vfprintf-internal.c:1644)
==11003== by 0x48C98D7: printf (printf.c:33)
==11003== by 0x109162: main (01_uninitialized.c:7)
==11003==
x = -16778112
==11003== Conditional jump or move depends on uninitialised value(s)
==11003== at 0x48DEE40: __vfprintf_internal (vfprintf-internal.c:1644)
==11003== by 0x48C98D7: printf (printf.c:33)
==11003== by 0x109189: main (01_uninitialized.c:10)
==11003==
==11003== Use of uninitialised value of size 8
==11003== at 0x48C332E: _itoa_word (_itoa.c:179)
3
==11003== by 0x48DE9EF: __vfprintf_internal (vfprintf-internal.c:1644)
```

```
==11003== by 0x48C98D7: printf (printf.c:33)
==11003== by 0x109189: main (01_uninitialized.c:10)
==11003==
==11003== Conditional jump or move depends on uninitialised value(s)
==11003== at 0x48C3339: _itoa_word (_itoa.c:179)
==11003== by 0x48DE9EF: __vfprintf_internal (vfprintf-internal.c:1644)
==11003== by 0x48C98D7: printf (printf.c:33)
==11003== by 0x109189: main (01_uninitialized.c:10)
==11003==
==11003== Conditional jump or move depends on uninitialised value(s)
==11003== at 0x48DF48B: __vfprintf_internal (vfprintf-internal.c:1644)
==11003== by 0x48C98D7: printf (printf.c:33)
==11003== by 0x109189: main (01_uninitialized.c:10)
==11003==
==11003== Conditional jump or move depends on uninitialised value(s)
==11003== at 0x48DEB5A: __vfprintf_internal (vfprintf-internal.c:1644)
==11003== by 0x48C98D7: printf (printf.c:33)
==11003== by 0x109189: main (01_uninitialized.c:10)
==11003==
t = 0
==11003==
==11003== HEAP SUMMARY:
==11003== in use at exit: 4 bytes in 1 blocks
==11003== total heap usage: 2 allocs, 1 frees, 1,028 bytes allocated
==11003==
==11003== LEAK SUMMARY:
==11003== definitely lost: 4 bytes in 1 blocks
==11003== indirectly lost: 0 bytes in 0 blocks
==11003== possibly lost: 0 bytes in 0 blocks
==11003== still reachable: 0 bytes in 0 blocks
==11003== suppressed: 0 bytes in 0 blocks
==11003== Rerun with --leak-check=full to see details of leaked memory
==11003==
==11003== For counts of detected and suppressed errors, rerun with: -v
==11003== Use --track-origins=yes to see where uninitialised values come from
==11003== ERROR SUMMARY: 24 errors from 10 contexts (suppressed: 0 from 0)
```

Da bi nam bilo lakše da pronademo glavni izvor greške sa korišćenjem nedefinisanih promenljivih koristimo opciju `--track-origins=yes` .

```
$ valgrind --track-origins=yes ./1
```

Tada uz poruku o upotrebi neinicijalizovane promenljive dobijamo i informaciju o liniji u kojoj je deklarisana:

```
==18060== Conditional jump or move depends on uninitialised value(s)
==18060== at 0x48DEE40: __vfprintf_internal (vfprintf-internal.c:1644)
==18060== by 0x48C98D7: printf (printf.c:33)
==18060== by 0x109162: main (01_uninitialized.c:7)
==18060== Uninitialised value was created by a stack allocation
==18060== at 0x109145: main (01_uninitialized.c:5)
==18060==
==18060== Use of uninitialised value of size 8
==18060== at 0x48C332E: _itoa_word (_itoa.c:179)
==18060== by 0x48DE9EF: __vfprintf_internal (vfprintf-internal.c:1644)
==18060== by 0x48C98D7: printf (printf.c:33)
==18060== by 0x109162: main (01_uninitialized.c:7)
==18060== Uninitialised value was created by a stack allocation
==18060== at 0x109145: main (01_uninitialized.c:5)
```

Primetimo da nemamo grešku da je promenljiva `y` inicijalizovana neinicijalizovanom promenljivom `x`. Tada se samo obelezava da ni `y` nije inicijalizovana. Tek prilikom prve upotrebe promenljive `y` biće detektovana greška, čiji uzrok je neinicijalizovano `x`.

5.1.2.2 Prosleđivanje sistemskim pozivima neinicijalizovane ili neadresirane vrednosti Memcheck prati sve parametre sistemskih poziva. Proverava svaki pojedinačno, bez obzira da li je inicijalizovan ili ne. Ukoliko sistemski poziv treba da čita iz prosledenog bafera, Memcheck proverava da li je ceo bafer adresiran i inicijalizovan. Ako sistemski poziv treba da piše u memoriju, proverava se da li je adresirana. Posle sistemskog poziva Memcheck ažurira svoje informacije o praćenju stanja memorije tako da one precizno opisuju promene koje su nastale izvršavanjem sistemskog poziva.

Program `02_undefined.c` sadrži dva sistemska poziva sa neinicijalizovanim parametrima. Memcheck je detektovao prvu grešku u prosledivanju neinicijalizovanog parametra `arr` sistemskom pozivu `write()`. Druga je u tome što sistemski poziv `read()` dobija neadresiran prostor. Tre ća greška je u tome što se sistemskom pozivu `exit()` prosleduje nedefinisan argument. Prikazane su nam i linije u samom programu koje sadrže detektovane greške.

```
$ valgrind --track-origins=yes ./02_undefined.out
```

```
==3422== Memcheck, a memory error detector
==3422== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3422== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==3422== Command: ./a.out
==3422==
==3422== Syscall param write(buf) points to uninitialised byte(s)
```

```
==3422== at 0x4978024: write (write.c:26)
==3422== by 0x10919E: main (02_undefined.c:9)
==3422== Address 0x4a59040 is 0 bytes inside a block of size 10 alloc'd
==3422== at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64
==3422== by 0x109176: main (02_undefined.c:6)
==3422== Uninitialised value was created by a heap allocation
==3422== at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64
==3422== by 0x109176: main (02_undefined.c:6)
==3422==
==3422== Syscall param read(buf) contains uninitialised byte(s)
==3422== at 0x4977F81: read (read.c:26)
==3422== by 0x1091B4: main (02_undefined.c:10)
==3422== Uninitialised value was created by a stack allocation
==3422== at 0x109165: main (02_undefined.c:5)
==3422==
==3422== Syscall param read(buf) points to unaddressable byte(s)
==3422== at 0x4977F81: read (read.c:26)
==3422== by 0x1091B4: main (02_undefined.c:10)
==3422== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==3422==
==3422== Syscall param exit_group(status) contains uninitialised byte(s)
==3422== at 0x494C926: _Exit (_exit.c:31)
==3422== by 0x48B23A9: __run_exit_handlers (exit.c:132)
==3422== by 0x48B23D9: exit (exit.c:139)
==3422== by 0x1091C1: main (02_undefined.c:11)
==3422== Uninitialised value was created by a heap allocation
==3422== at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64
==3422== by 0x109184: main (02_undefined.c:8)
5
==3422==
==3422==
==3422== HEAP SUMMARY:
==3422== in use at exit: 14 bytes in 2 blocks
==3422== total heap usage: 2 allocs, 0 frees, 14 bytes allocated
==3422==
==3422== LEAK SUMMARY:
==3422== definitely lost: 0 bytes in 0 blocks
==3422== indirectly lost: 0 bytes in 0 blocks
==3422== possibly lost: 0 bytes in 0 blocks
==3422== still reachable: 14 bytes in 2 blocks
==3422== suppressed: 0 bytes in 0 blocks
==3422== Rerun with --leak-check=full to see details of leaked memory
```

```
==3422==
==3422== For counts of detected and suppressed errors, rerun with: -v
==3422== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 0 from 0)
```

Takođe, Memcheck prilikom izvršavanja beleži podatke o svim dinamički alociranim blokovima memorija. Po završetku programa, ima sve informacije o neoslobodenim memorijskim blokovima. Opcijom `--leak-check=yes` za svaki neosloboden blok se odreduje da li mu je moguće pristupiti preko pokazivača (still reachable) ili ne (definitely lost).

Opcijama `--leak-check=full --show-leak-kinds=all` tražimo da nam se prikaže detaljan izveštaj o svakom definitivno ili potencijalno izgubljenom bloku, kao i o tome gde je alociran u delu sa izveštajem sa hipa - `HEAP SUMMARY` .

```
==3439== HEAP SUMMARY:
==3439== in use at exit: 14 bytes in 2 blocks
==3439== total heap usage: 2 allocs, 0 frees, 14 bytes allocated
==3439==
==3439== 4 bytes in 1 blocks are still reachable in loss record 1 of 2
==3439== at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64
==3439== by 0x109184: main (02_undefined.c:8)
==3439==
==3439== 10 bytes in 1 blocks are still reachable in loss record 2 of 2
==3439== at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64
==3439== by 0x109176: main (02_undefined.c:6)
==3439==
==3439== LEAK SUMMARY:
==3439== definitely lost: 0 bytes in 0 blocks
==3439== indirectly lost: 0 bytes in 0 blocks
==3439== possibly lost: 0 bytes in 0 blocks
==3439== still reachable: 14 bytes in 2 blocks
==3439== suppressed: 0 bytes in 0 blocks
```

5.1.2.3 Nedopušteno oslobadanje memorije Memcheck prati svaku alokaciju memorije pozivom funkcija kao što su `malloc` i `calloc` , ali i alokacije uzrokovane konstrukcijom objekata (`new`). Iz tog razloga tačno zna da li je argument funkcije `free` , odnosno `delete` , ispravan ili ne. U sledećem programu isti blok dinamički alocirane memorije se oslobada dva puta. Memcheck prijavljuje tu grešku a potom detektuje i drugu grešku prilikom pokušaja oslobadanje bloka memorije sa adrese koja nije na hipu.

```
$ valgrind --show-leak-kinds=all --leak-check=full --track-origins=yes ./03_malloc.out
```

```
==3914== Memcheck, a memory error detector
==3914== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
```

```
==3914== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==3914== Command: ./3
==3914==
==3914== Invalid free() / delete / delete[] / realloc()
==3914== at 0x483997B: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-l
==3914== by 0x10919F: main (03_malloc.c:11)
==3914== Address 0x4a590a0 is 0 bytes inside a block of size 12 free'd
==3914== at 0x483997B: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-l
==3914== by 0x109193: main (03_malloc.c:10)
==3914== Block was alloc'd at
==3914== at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64
==3914== by 0x109183: main (03_malloc.c:9)
==3914==
==3914== Invalid free() / delete / delete[] / realloc()
==3914== at 0x483997B: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-l
==3914== by 0x1091B3: main (03_malloc.c:14)
==3914== Address 0x1ffefffd0f is on thread 1's stack
==3914== in frame #1, created by main (03_malloc.c:5)
```

I u ovom programu imamo i pored svega curenje memorije jer 19B bivaju alocirani i potom pokazivač `p` dobije vrednost adrese novog prostora. Pokretanjem sa opcijama `--leak-check=full` i `--show-leak-kinds=all` dobijamo informaciju da imamo 19B koji su sasvim izgubljeni jer smo izgubili adresu alociranog bloka na hipu.

```
==3914== HEAP SUMMARY:
==3914== in use at exit: 19 bytes in 1 blocks
==3914== total heap usage: 2 allocs, 3 frees, 31 bytes allocated
==3914==
==3914== 19 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3914== at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64
==3914== by 0x109175: main (03_malloc.c:8)
==3914==
==3914== LEAK SUMMARY:
==3914== definitely lost: 19 bytes in 1 blocks
==3914== indirectly lost: 0 bytes in 0 blocks
==3914== possibly lost: 0 bytes in 0 blocks
==3914== still reachable: 0 bytes in 0 blocks
==3914== suppressed: 0 bytes in 0 blocks
==3914==
==3914== For counts of detected and suppressed errors, rerun with: -v
==3914== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

5.1.2.4 Nekorektno oslobađanje memorije U primeru `04_new_delete.cpp` se ne upotrebljavaju odgovarajuće funkcije za oslobadanje dinamički alocirane memorije. Pokrenimo Memcheck:

```
$ valgrind --track-origins=yes ./04_new_delete.out
```

```
==4011== Memcheck, a memory error detector
==4011== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4011== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==4011== Command: ./4
==4011==
==4011== Invalid free() / delete / delete[] / realloc()
==4011== at 0x483997B: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-l
==4011== by 0x1091AD: main (04_new_delete.cpp:10)
==4011== Address 0x4db6c88 is 8 bytes inside a block of size 168 alloc'd
==4011== at 0x48394DF: operator new[](unsigned long) (in /usr/lib/x86_64-linux-gnu/valgrind/v
==4011== by 0x10916B: main (04_new_delete.cpp:8)
==4011==
==4011==
==4011== HEAP SUMMARY:
==4011== in use at exit: 168 bytes in 1 blocks
==4011== total heap usage: 2 allocs, 2 frees, 72,872 bytes allocated
==4011==
==4011== LEAK SUMMARY:
==4011== definitely lost: 168 bytes in 1 blocks
==4011== indirectly lost: 0 bytes in 0 blocks
==4011== possibly lost: 0 bytes in 0 blocks
==4011== still reachable: 0 bytes in 0 blocks
==4011== suppressed: 0 bytes in 0 blocks
==4011== Rerun with --leak-check=full to see details of leaked memory
==4011==
==4011== For counts of detected and suppressed errors, rerun with: -v
==4011== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

5.1.2.5 Delimično preklapanje izvorne i ciljne memorije Program `05_overlap.c` prepisuje nisku u lokaciju koja se preklapa sa onom odakle se prepisuje. Ako unesemo reč `Da` , neće se detektovati gredška, jer prilikom izvrdšavanja za unetu reč neće biti preklapanja. Ponovo pokrenimo program i unesimo reč `dobro` . Memcheck detektuje problem preklapanja memorijskih lokacija sa koje se kopira i one na koju se kopira.

```
==16178== Source and destination overlap in memcpy(0x5204041, 0x5204040, 5)
==16178== at 0x4C32513: memcpy@@GLIBC_2.14 (in /usr/lib/valgrind/vgpreload_memcheck-amd64-lin
==16178== by 0x400617: main (05_overlap.c:9)
```

Ukoliko se unese kraća reč od 3 slova ovo će biti jedina gredška koju imamo. Ukoliko unesemo dužu reč na ulazu, `printf` će nam prijaviti pristup neinicilizovanoj vrednosti.

5.1.2.6 Detekcija neispravnog argumenta pri alokaciji memorije Memcheck može da detekuje grešku slanja negativnog argumenta za veličinu alociranog prostora. Program `06_fishy_arguments.c` ilustruje tu grešku. Ukoliko se program pokrene za negativne brojeve neće se ništa desiti jer petlja u tom slučaju neće imati ni jednu iteraciju, pa neće biti ni poziva `malloc` funkcije. Pozivi sa `n < 27` neće biti problematični - sve će se lepo alocirati i osloboditi. Imaćemo samo prijavljeno upozorenje:

```
(Warning: set address range perms: large range)
```

Ovo znači da se velikom bloku memorije menjaju prava pristupa. To upozorenje je namenjeno zapravo najviše Valgrind developerima i možemo ga ignorisati jer će memorija biti alocirana. Ukoliko pokrenemo ponovo Valgrind i za `n` unesemo `28` , dobićemo grešku jer smo prekoračili opseg `int` domena.

```
sada je i = 26 allocated = 1073741824
==5105== Warning: set address range perms: large range [0x59c89028, 0x99c89058) (noaccess)
sada je i = 27 allocated = -2147483648B
==5105== Argument 'size' of function malloc has a fishy (possibly negative) value: -214748364
==5105== at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64
==5105== by 0x109219: main (06_fishy_arguments.c:20)
==5105==
Realloc failed!
```

5.1.3 Massif

Massif je profajer hip memorije - beleži iskorišćen prostor i dodatne bajtove koji se zauzimaju radi poravnanja i vodenja evidencije bajtova u upotrebi. Može da meri i zauzeće memorije na steku, ali to ne radi bez uključivanja dodatne opcije (`--stacks=yes`) jer dosta usporava rad alata.

Profajliranje hip memorije nam može pomoći da eventalno smanjimo korišćenu memoriju i da otkrijemo neka curenja memorije koja se ne mogu prepoznati Memcheck -om. Prevashodno kad memorija nije još uvek izgubljena, imamo pokazivač, ali nije u upotrebi. Kod takvih programa nepotrebno se povećava memorija koja se koristi tokom vremena. Massif nam može reći i koliko memorije na hipu program koristi i tačnu liniju koda koja je zaslužna za njegovu alokaciju.

Kao i pri upotrebi drugih Valgrind alata, program prevodimo sa informacijama debug sombolima (opcija `-g`). Optimizacija neće uticati na količinu upotrebljene memorije. Massif svoj izveštaj upisuje u datoteku `massif.out.<pid>` , gde je `<pid>` ID procesa. Ukoliko želimo da se upiše u drugu datoteku koristimo opciju `--massif-out-file` i navedemo naziv izlazne datoteke. Informacije iz izveštaja prikazujemo programom `ms_print` prosledujući mu datoteku sa izveštajem.

`ms_print` proizvodi graf koji prikazuje zauzeće memorije tokom izvršavanja programa, kao i detaljnije informacije o različitim tačkama programa koje su odgovorne za alokaciju/dealokaciju memorije. Vrednosti na y osi predstavljaju presek stanja iskorišćenosti memorije u odredenom vremenskom trenutku. Na x osi, Massif podrazumevano koristi broj izvršenih instrukcija kao jedinicu vremena. To se može promeneti opcijom `--time-unit=B` da nam jedinica vremena bude broj bajtova alociran/dealociran na hipu.

Massif pravi preseke stanja iskorišćenosti memorije za svaku alokaciju i dealokaciju hipa, ali ako se program duže izvršava Massif ređe pravi preseke. Kada dostigne maksimalni broj preseka on odbaci oko pola ranijih preseka. Podrazumevan broj preseka koje čuva je 100, ali se to može promeniti opcijom `--max-snapshots` . Detaljnije obradeni preseci su na grafu predstavljeni simbolom `@` . Podrazumevano detaljnije obraduje svaki deseti presek, ali se i to može promeniti opcijom `--detailed-freq` .

```
$ valgrind --tool=massif ./massif_example.out
```

Na grafu je simbolom `#` predstavljen još jedan detaljan presek koji je obraden i ujedno je i presek sa najvećim iskorišćenjem memorije. Odredivanje preseka sa najvećim iskorišćenjem memorije nije uvek tačno jer Massif uzima u obzir samo preseke kod kojih se desila dealokacija. Time se izbegava mnogo nepotrebnih pravljenja preseka za najveću iskorišćenost memorije, npr. u slučaju da program iz više navrata alocira dosta memorije, i svaki put to je nova najveća iskorišćenost memorije. Takode, ako program nikada ne dealocira memoriju, nećemo ni imati ovakav presek. Takode ako program i dealocira memoriju ali kasnije alocira još veći blok koji kasnije ne oslobodi, imaćemo presek sa najvećom iskorišćenosti memorije, ali će on biti dosta niži od stvarnog.

Alat Massif meri samo hip memoriju, tj. onu koju smo alocirali funkcijama `malloc` , `calloc` , `realloc` , `memalign` , `new` , `new[]` . Ne meri memoriju alociranu sistemskim pozivima kao što su `mmap` , `mremap` , `brk` . Ukoliko nam je od značaja merenje celokupne alocirane memorije, potrebno je uključiti opciju `--pages-as-heap=yes` . Sa ovom opcijom Massif neće profajlirati hip memoriju, već stranice u memoriji.

5.1.4 Cachegrind

Merenje performansi keša je postalo važno jer se eksponencijalno pove´ćava razlika u brzini RAM memorije i performansi procesora. Uloga keša je da premosti tu razliku u brzini. Da bismo utvrdili koliko je keš sposoban da to učini prate se pogodci (*hits*) i promašaji (*misses*) keša. Jedan od koraka za poboljšanje je da se smanji broj promašaja na najvišim nivoima keša.

Cachegrind je alat koji omogućava softversko profajliranje keš memorije tako što simulira i prati pristup keš memoriji mašine na kojoj se program, koji se analizira, izvršava. Može se koristiti i za profajliranje izvršavanja grana, korišćenjem opcije `--branch-sim=yes` . Cachegrind simulira memoriju mašine, koja ima prvi nivo keš

memorije podeljene u dve odvojene nezavisne sekcije: **I1** - sekcija keš memorije u koju se smeštaju instrukcije **D1** - sekcija keš memorije u koju se smeštaju podaci

Drugi nivo keš memorije koju Cachegrind simulira je objedinjen - **LL**, skraćeno od eng. *last level*. Ovaj način konfiguracije odgovara mnogim modernim mašinama. Postoje mašine koje imaju i tri ili četiri nivoa keš memorije. U tom slučaju, Cachegrind simulira pristup prvom i poslednjem nivou. Generalno gledano, Cachegrind simulira I1, D1 i LL (poslednji nivo keš memorije). Cachegrind prikuplja sledeće statističke podatke o programu koji analizira:

- O čitanjima instrukcija iz keš memorije:
  - **Ir** - ukupan broj izvršenih instrukcija
  - **I1mr** - broj promašaja čitanja instrukcija iz keš memorije nivoa I1
  - **ILmr** - broj promašaja čitanja instrukcija iz keš memorije nivoa LL
- O čitanjima brze memorije:
  - **Dr** - ukupan broj čitanja memorije
  - **D1mr** - broj promašaja čitanja nivoa keš memorije D1
  - **DLmr** - broj promašaja čitanja nivoa keš memorije LL
- O pisanjima u brzu memoriju:
  - **Dw** - ukupan broj pisanja u memoriji
  - **D1mw** - broj promašaja pisanja u nivo keš memorije D1
  - **DLmw** - broj promašaja pisanja u nivo keš memorije LL
- O izvršavanju grana:
  - **Bc** - broj uslovno izvršenih grana
  - **Bcm** - broj promašaja uslovno izvršenih grana
  - **Bi** - broj indirektno izvršenih grana
  - **Bim** - broj promašaja indirektno izvršenih grana

Broj pristupa D1 delu keš memorije je jednak zbiru statistika D1mr i D1mw. Ukupan broj pristupa LL nivou jednak je zbiru ILmr, DLmr i DLmw. Primeri u C/C++ koji proizvode indirektno grananje su pozivi funkcija preko pokazivača na funkcije ili pozivi virtuelnih funkcija i `switch` naredbe. Uslovne grananje se generše `if` naredbama i uslovnim ternarnim operatorom `?:` .

Statistika se prikuplja na nivou celog programa, kao i pojedinačno na nivou funkcija. Na modernim mašinama L1 promašaj košta oko 10 procesorskih ciklusa, LL promašaj košta oko 200 procesorskih ciklusa, a promašaji uslovno i indirektno izvršene grane od 10 do 30 procesorskih ciklusa. Detaljno profajliranje upotrebe keš memorije može pomoći u poboljšanju performansi programa. Izvršavanjem komande `lscpu` na Linux sistemima, dobićemo informacije o veličini keša na mašini na kojoj radimo.

Program koji želimo da analiziramo propuštamo kroz Cachegrind navodeći opciju `--tool=cachegrind` . Za razliku od ostalih Valgrind alata želimo uključenu optimizaciju, tako da ne koristimo opciju `-O0` prilikom prevođenja programa. Kompilacija treba da bude sa optimizacijom, jer nema smisla ovako profajlirati kod koji je drugačiji od onoga

koji će se normalno izvršavati. Na standardni izlaz se ispisuju sumarne informacije na nivou celog programa, dok se detaljne informacije upisuju u `cachegrind.out.<pid>` datoteku, gde `<pid>` predstavlja ID procesa. Alat grupiše sve troškove po datotekama i funkcijama kojima ti troškovi pripadaju.

Detaljniji izveštaj možemo videti iz iste datoteke korišćenjem alata `cg_annotate` . Ukoliko imamo izveštaje iz više pokretanja Cachgrind-a za isti program, možemo ih sumirati u jednu datoteku korišćenjem alata `cg_merge` i njegov izlaz kasnije pregledati alatom `cg_annotate` . Moguće je i praviti razliku izmedu više izveštaja Cachgrind-a pomoću alata `cg_diff` i njegov izlaz, slično, otvoriti pomoću `cg_annotate` . Ukoliko modifikujemo kod, to nam može biti od korisiti da pratimo kako modifikacija utiče na performanse programa. Programom `cg_annotate` podrazumevano se prikazuje izveštaj sumiran po funkcijama.

Ako kolona sadrži samo tačkicu, to označava da funkcija ne sadrži instrukcije koje bi prouzrokovale taj dogadaj. Ukoliko u koloni za ime funkcije stoji `???` , to znači da nije bilo moguće odrediti ime na osnovu simbola za debagovanje. Ukoliko većina redova sadrži ??? za ime funkcije program verovatno nije preveden sa opcijom kompajlera -g.

Ukoliko želimo da vidimo izveštaj o broju promašaja po linijama koda, potrebno je da prosledimo izvorne datoteke programu `cg_annotate` ili da uključimo opciju `--auto=yes` kada će se anotirati svaki izvorni kod koji se može naći.

Daleko preglednije je gledati izveštaj pomoću alata KCachgrind.

6 Instalacije

6.1 Alati za debagovanje i razvojna okruženja

6.1.1 QtCreator

Instalirati QtCreator sa zvanične stranice. Alternativno, moguće je i instalirati ceo Qt radni okvir koji uključuje i QtCreator.

Za neke Linux distribucije je dostupan paket `qt<VERZIJA>-creator` .

6.1.2 gdb

Za većinu Linux distribucija je dostupan paket `gdb` . `gdb` je za neke distribucije deo paketa za razvoj (npr. `build-essential` za Ubuntu).

6.2 Alati/Biblioteke za testiranje jedinica koda i pokrivenosti koda

6.2.1 gcov, lcov

`gcov` dolazi podrazumevano uz `gcc` kompajler. Alat `lcov` je obično dostupan u okviru paketa sa istim imenom. Instalacija na Ubuntu distribuciji bi, na primer, izgledala ovako: `sudo apt-get install lcov`

6.2.2 Gradle

Da bi se Gradle instalirao, neophodno je na sistemu imati verziju JDK-a veću od 8. Gradle se potom jednostavno instalira kroz `gradle` za većinu popularnih Linux distribucija. Alternativno, moguće je preuzeti unapred spremne Gradle artifakte i ručno instalirati Gradle. Primeri pretpostavljaju da je izvršivi fajl (ili alias) `gradle` dostupan na `PATH` -u i pokreće Gradle alat.

Neobavezno za ovaj primer, za laku organizaciju u okruženju sa više različitih JDK verzija, može se koristiti alat SDKMAN. Gradle se može instalirati korišćenjem SDKMAN-a:

```
$ sdk install gradle <verzija>
```

Na primer:

```
$ sdk install gradle 7.5.1
```

6.2.3 xUnit / NUnit

`xUnit` i `NUnit` se jednostavno instaliraju sa NuGet repozitorijuma (xUnit, NUnit) ili uz pomoć IDE-a, ili kroz komande:

```
$ dotnet add package xunit --version 2.4.2
$ dotnet add package NUnit --version 3.13.3
```

6.3 Alati/Biblioteke za Mock testiranje

6.3.1 Moq

`Moq` se jednostavno instalira sa NuGet repozitorijuma ili uz pomoć IDE-a, ili kroz komandu:

```
$ dotnet add package Moq --version 4.18.2
```

6.4 Profajleri

6.4.1 Valgrind

Valgrind se na većini Linux distribucija može instalirati kroz paket `valgrind` . Npr., za Ubuntu:

```
$ sudo apt-get install valgrind
```

Za grafički prikaz izveštaja nekih Valgrind-ovih alata može se koristiti program *KCachegrind*. Instalacija, npr., za Ubuntu:

```
$ sudo apt-get install kcachegrind
```