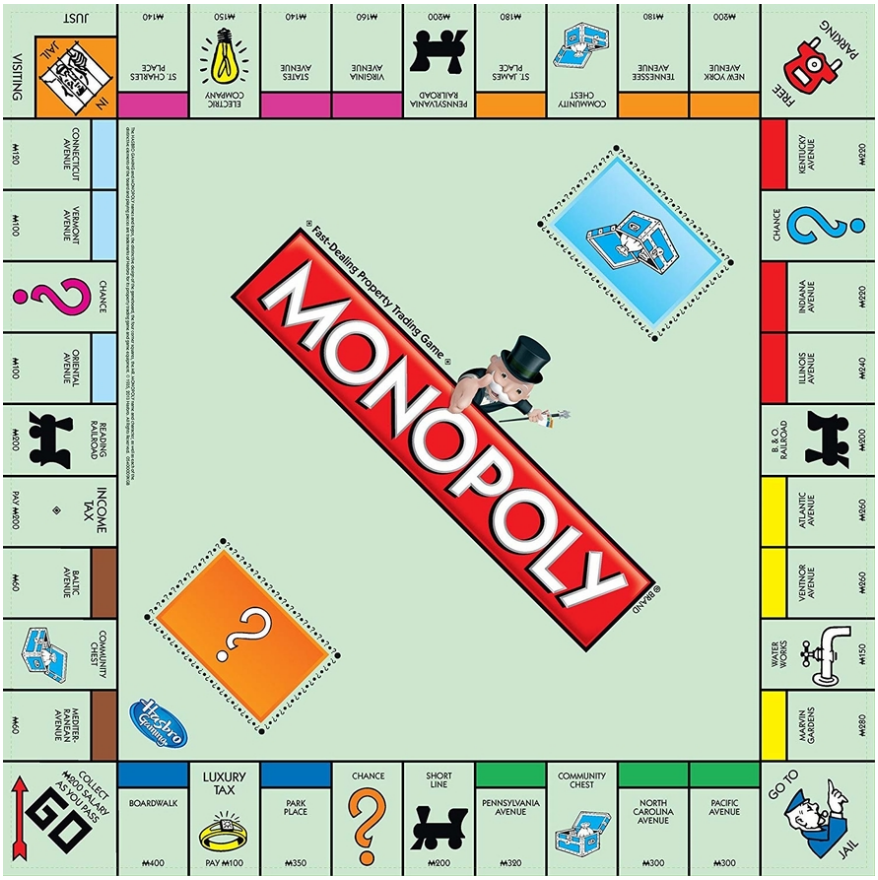
Analiza projekta "Monopol"

Milena Stojić

August 2023



Sažetak

U ovom dokumentu će biti prikazana analiza kompjuterske igre "Monopol". "Monopol"je studentski projekat rađen na kursu iz Razvoja softvera u školskoj 2018/2019 godini. Projekat je rađen u programskom jeziku C++ u QT radnom okviru.
Cilj analize projekta je bio da se ispita ispravnost ovog programa, utvrde performanse, kao i da se vidi kakav je kvalitet izvornog koda i da li ima mesta njegovom unapređenju.

Alati koji su korišćeni pri analizi projekta:

- QtTest (za jedinične testove)
- Valgrind alati:
  - memcheck
  - massif
  - callgrind
- Clang-Tidy analizator

U prvom poglavlju će biti uvodna reč o samom projektu koji se analizira. Drugo poglavlje će biti posvećeno jediničnom testiranju. U trećem poglavlju će biti izloženi rezultati profajliranja dobijeni *Valgrind*-ovim alatima. U četvrtom poglavlju ćemo razmatrati izlaze dobijene *Clang-Tidy* analizatorom i moguće pravce unapređivanja koda na koje nam sugerišu. I u petom poglavlju dajemo zaključak cele analize .

Sadržaj

1 Uvod u projekat **3**
1.1 Implementacija monopola 3

2 Jedinično (eng. *Unit*) testiranje **4**

3 Profajliranje **5**
3.1 Memcheck 5
3.1.1 Dobijeni rezultati 6
3.2 Massif 10
3.2.1 Dobijeni rezultati 12
3.3 Callgrind 12
3.3.1 Dobijeni rezultati 13

4 Statička analiza softvera **14**

5 Zaključak **15**

1 Uvod u projekat

Monopol je stara popularna društvena igra. U njoj mogu igrati od 2 do 4 igrača. Svaki igrač na početku dobija početni kapital, tj. početnu sumu novca. Celu partiju bacaju 2 kockice i pomeraju se po tabli, imitirajući tako putovanje oko sveta. Polja na tabli najčešće predstavljaju placeve u gradovima širom sveta (koje može da poseduje jedan od igrača ili koji mogu biti kupljeni od igrača), ali takođe mogu biti i druge vrste poseda ili tzv. specijalna polja koja mogu predstavljati razne akcije (mogućnost uzimanje karte šansi, naplate poreza, ...). Igrači tokom partije donose odluke o kupovini poseda preko kojih prolaze, potencijalnoj izgradnji kuća i hotela na njima. Kako bi što duže ostao u igri igrač bi trebalo da ima dobru strategiju ulaganja novca u svoje nekretnine tako da od svojih nekretnina imao dovoljno velike prihode. Kada jedan od igrača bankrotira (ostane bez novca i dovoljne vrednosti nekretnina) on ispada iz igre. Pobednik je poslednji igrač koji ostane u igri.

Danas u prodavnicama društvenih igara postoje različite verzije ove igre i neke od njih mogu imati i dodatna pravila koju igru čine zanimljivijom ili možda kraćom. (pošto partije klasičnog monopola traju i do nekoliko sati)

1.1 Implementacija monopola

Projekat koji ćemo analizirati predstavlja implementaciju ove čuvene igre. U ovoj implementaciji igrač može da igra i protiv računara, i to protiv jednog, dva ili tri fiktivna protivnika. Svi elementi igre su predstavljeni objektima, a svi potezi i koraci u potezima su predstavljeni odgovarajućim metodama. U tabeli ispod su predstavljeni objekti u igri i klase koje ih implementiraju.

Objekat u igri	**Klasa koja ga implementira**
Partija igre	*Game*
Igrač	*Player*
Tabla za igru	*Board*
Polje na tabli	*Space* (apstraktna klasa za sva polja)
Plac	*Property* (nasleđuje *Space*)
Železnička stanica	*Railroad* (nasleđuje *Space*)
Dobro (vodovod ili elektrodistribucija)	*Utility* (nasleđuje *Space*)
Polje na tabli koje predstavlja neku akciju	*ActionSpace* (nasleđuje *Space*)
Karta (šanse ili iznenađenja)	*Card*

2 Jedinično (eng. *Unit*) testiranje

Prvi korak kome smo pristupili u analizi ovog softvera je njegovo testiranje. Cilj ovog koraka je bio da se utvrdi ispravnost aplikacije, otpornost na greške i pokušaj da se pronađu bagovi u programu. Testiranje je inače jedna od tehnika dinamičke verifikacije softvera, tj. verifikacije programa u toku njegovog izvršavanja.

Nad programom smo izvšrili jedinično (eng. *Unit*) testiranje, tj. testirali smo ispravnost metoda klasa *Bank*, *Utility*, *Player* i *Game*. Proveravali smo da li metode daju korektan rezultat ili/i imaju ponašanje u skladu sa zadatim ulazima. Testiranje smo izvršili metodom bele kutije, tj. poznavajući izvorni kod.

Radni okvir koju smo koristili za pisanje testova je *QtTest* u okviru razvojnog okruženja *QT Creator*.

Testovi su imenovani tako da bude jasno koju funkciju testiramo i koji je željeni efekat posle izvršavanja te funkcije. Na slici ispod se nalaze svi testovi koje smo napisali za ovaj projekat:

```
void playersBalanceAfterBuyCase();  // Bank functionalities
void incrementedPlayersBalanceAfterGive();
void decrementedPlayersBalanceAfterTake();
void bankNotTakesIsufficientOnAccount();
void numberOfHousesIsReallySet();
void numberOfHotelsIsReallySet();

void addPlayerToUtility();  // Utility functionalities
void erasedPlayerFromUtility();

void numberOfPropertiesIncreasedAfterBuy(); // Player functionalities
void numberOfUtilitiesIncreasedAfterBuy();
void numberOfRailroadsIncreasedAfterBuy();
void propertyOwnedByPlayerAfterAddition();

void throwDiceIsCorrect(); // Game functionalities
```

A ovde se nalazi izlaz posle izvršavanja svih testova:

```
********* Start testing of monopolFunctionalities *********
Config: Using QtTest library 6.4.3, Qt 6.4.3 (x86_64-little_endian-lp64 shared (dynamic) release build; by Apple LLVM 14.0.0 (clang-1
13.5
PASS   : monopolFunctionalities::initTestCase()
PASS   : monopolFunctionalities::playersBalanceAfterBuyCase()
PASS   : monopolFunctionalities::incrementedPlayersBalanceAfterGive()
PASS   : monopolFunctionalities::decrementedPlayersBalanceAfterTake()
PASS   : monopolFunctionalities::bankNotTakesIsufficientOnAccount()
PASS   : monopolFunctionalities::numberOfHousesIsReallySet()
PASS   : monopolFunctionalities::numberOfHotelsIsReallySet()
PASS   : monopolFunctionalities::addPlayerToUtility()
PASS   : monopolFunctionalities::erasedPlayerFromUtility()
PASS   : monopolFunctionalities::numberOfPropertiesIncreasedAfterBuy()
PASS   : monopolFunctionalities::numberOfUtilitiesIncreasedAfterBuy()
PASS   : monopolFunctionalities::numberOfRailroadsIncreasedAfterBuy()
PASS   : monopolFunctionalities::propertyOwnedByPlayerAfterAddition()
PASS   : monopolFunctionalities::throwDiceIsCorrect()
PASS   : monopolFunctionalities::cleanupTestCase()
Totals: 15 passed, 0 failed, 0 skipped, 0 blacklisted, 4ms
********* Finished testing of monopolFunctionalities *********

Process exited with code: 0
```

Možemo da vidimo da je naš kod uspešno prošao sve testove. Iako pri ovom testiranju nismo naišli ni na jednu neispravnost, ne mora da znači da je ceo program potpuno korektan. Čak i kada bismo sve metode pokrili testovima i kada bismo testirali za više ulaza, to nije garancija da je naš program potpuno ispravan. Ovom metodom verifikacije softvera bismo jedino imali garantovanu ispravnost ako bismo svaki metod testirali za svaku moguću kombinaciju argumenata što je praktično neizvodljivo.

3 Profajliranje

Ovo poglavlje će biti posvećeno izvršenim profajliranjima. Za profajliranje su korišćeni *Valgrind*-ovi alati. *Valgrind* nam obezbeđuje mnoge korisne alate za profajliranje pomoću kojih možemo detektovati curenje i/ili preveliku potrošnju memorije, funkcije koje izvršavaju veliki broj instrukcija i sve one koje ih pozivaju, da li se dobro iskorišćava keš memorija, takođe alat za profajliranje višenitnih aplikacija... Za alate *Cachegrind* i *Callgrind* je obezbeđen i alat *KCachegrind* pomoću kog profil možemo gledati u okviru interaktivnog korisničkog interfejsa. (i on je korišćen u ovoj analizi)

Profajliranje je još jedna vrsta dinamičke verifikacije softvera. Ideja je da alat tokom izvršavanja beleži podatke o programu. Beleženje podataka predstavlja instrumentaciju, a svi zabeleženi podaci predstavljaju profil programa. Svaki alat radi drugačiju instrumentaciju i beleži drugačije informacije. Cilj profajliranja je da se na osnovu dobijenog profila pronađu tzv. uska grla programa i samim tim vidi koji je deo programa čija bi refaktorizacija najviše doprinela poboljšanju performansi, a time i celokupnoj upotrebljivosti softvera.

3.1 Memcheck

Prvo smo primenili alat *Memcheck*. *Memcheck* uočava različite tipove memorijskih grešaka, od upotrebe neinicijalizovanih promenljivih, dvostrukog oslobađanja

memorije, do grešaka kao što su curenje memorije.

Napomenuli bismo da smo pri prvobitnom pokretanju alata *Memcheck* nad našim projektom iz terminala dobili neočekivani *segmentation-fault* (iako uobičajeno program radi bez pucanja). Iz dobijenog izveštaja koji je sačuvan u datoteci *__invalid_memcheck_output.txt* uz tu grešku smo videli poruku *General Protection Fault* i pošto iz narednih linija nije bilo jasno odakle je ta greška proizašla, istraživanjem *QT*-ovih foruma smo videli da je problem u stvari bio u konfiguraciji *bash*-a i biblioteke za rad sa grafikom. Zato mislimo da bi i ovaj deo bio koristan da se naglasi, pošto se na kursu često analiziraju *QT*-ovi grafički projekti.

Konfiguracija lako može da se izmeni dodavanjem sledeće komande u odgovarajuću datoteku. (ovde je to ~/.bashrc konfiguraciona datoteka)

```
export LIBGL_ALWAYS_SOFTWARE=1
```

Posle ovog koraka smo mogli regularno da izvršimo profajliranje.
Pre ovoga smo preveli program u *debug* modu u terminalu sledećim komandama:

```
$ qmake CONFIG+=debug
$ make
```

I, zatim smo pokrenuli *Memcheck*:

```
$ valgrind --track-origins=yes --leak-check=full --show-leak-kinds=all  \
  --log-file=memcheck_output.txt ./RS019-monopol
```

Memcheck je podrazumevani *Valgrind*-ov alat, pa zato ne moramo da eksplicitno naznačavamo opciju da koristimo taj alat. Zahvaljujući opciji –track-origins možemo da lociramo gde smo koristili neinicijalizovanu vrednost, kao i gde je ona u kodu bila deklarisana. Uključivanjem opcija –leak-check=full i -show-leak-kinds=all dobijamo kompletan izveštaj i statistiku o curenju memorije. Opcijom –log-file naznačavamo gde želimo da sačuvamo izlaz ovog alata. (podrazumevano se ispisuje u terminal)

U okviru ovog dokumenta nećemo prikazati kompletan izveštaj jer je veoma obiman, nego samo neke od interesantnih delova. Kompletan izveštaj se može naći u datoteci *memcheck_output.txt* u okviru direktorijuma *Memcheck*.

3.1.1 Dobijeni rezultati

U prvom delu izveštaja možemo videti da je detektovana upotreba neinicijalizovanih vrednosti.

```
==1628==
==1628== Conditional jump or move depends on uninitialised value(s)
==1628==    at 0x132428: Player::send_to_jail() (in /home/user/milena/Verifikacija/RS019-monopol/RS019-monopol)
==1628==    by 0x119B08: Game::send_to_jail(Player*) (in /home/user/milena/Verifikacija/RS019-monopol/RS019-monopol)
==1628==    by 0x12C449: MainWindow::reactToField() (in /home/user/milena/Verifikacija/RS019-monopol/RS019-monopol)
==1628==    by 0x12EA4E: MainWindow::roll_dice() (in /home/user/milena/Verifikacija/RS019-monopol/RS019-monopol)
==1628==    by 0x12EEB0: MainWindow::roll_dice() (in /home/user/milena/Verifikacija/RS019-monopol/RS019-monopol)
==1628==    by 0x12EE0E: MainWindow::roll_dice() (in /home/user/milena/Verifikacija/RS019-monopol/RS019-monopol)
==1628==    by 0x57DD2FF: QMetaObject::activate(QObject*, int, int, void**) (in /usr/lib/x86_64-linux-gnu/libQt5Core.so.5.12.8)
==1628==    by 0x4AC2805: QAbstractButton::clicked(bool) (in /usr/lib/x86_64-linux-gnu/libQt5Widgets.so.5.12.8)
==1628==    by 0x4AC2A2D: ??? (in /usr/lib/x86_64-linux-gnu/libQt5Widgets.so.5.12.8)
==1628==    by 0x4AC3E72: ??? (in /usr/lib/x86_64-linux-gnu/libQt5Widgets.so.5.12.8)
==1628==    by 0x4AC4034: QAbstractButton::mouseReleaseEvent(QMouseEvent*) (in /usr/lib/x86_64-linux-gnu/libQt5Widgets.so.5.12.8)
==1628==    by 0x4A102B5: QWidget::event(QEvent*) (in /usr/lib/x86_64-linux-gnu/libQt5Widgets.so.5.12.8)
==1628==  Uninitialised value was created by a heap allocation
==1628==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==1628==    by 0x132EFA: Player::initializePlayers(std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
↪ std::allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >) (in /home/user/milena/Verifikacija/RS019-
↪ RS019-monopol)
==1628==    by 0x11AFC2: Game::Game(std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >,
↪ std::allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >) (in /home/user/milena/Verifikacija/RS019-
↪ RS019-monopol)
==1628==    by 0x130A38: MainWindow::MainWindow() (in /home/user/milena/Verifikacija/RS019-monopol/RS019-monopol)
==1628==    by 0x114FBF: main (in /home/user/milena/Verifikacija/RS019-monopol/RS019-monopol)
==1628==
```

Prvi deo *Memcheck* izveštaja

Imamo 2 greške upotrebe neinicijalizovanih vrednosti. (linije 7 i 20) One su u stvari međusobno povezane.
Vidimo da je prva greška iz izveštaja (linija 7) nastala u metodu *send_to_jail()* klase *Player*. Hajde da pogledamo kod tog metoda:

```
}

void Player::send_to_jail()
{
    if (m_in_jail == true) return;
    if (has_jail_card)
    {
        release_from_jail();
        has_jail_card = false;
        //TODO: staviti kartu nazad u spil
    }

    m_in_jail = true;
    num_turns_in_jail = 1;
}
```

Metod *send_to_jail()* klase *Player*

Prijavljena greška se odnosi na skok zasnovan na neinicijalizovanoj vrednosti, što znači da je atribut *m_in_jail* neinicijalizovan. Pogledajmo konstruktore ove klase:

```
std::vector<Player*> Player::initializePlayers(std::vector<std::string> player_names){
    std::vector<Player*> players;
    Player* p;
    std::string name;

    for(int i = 0; i < player_names.size(); i++){    ⚠ Comparison of integers of different signs: 'int' and 'size_type' (aka 'unsigne
        p = new Player();                                                                                    rubidijum 2018-12-3
        p->set_name(player_names.at(i));
        p->set_pos(0);
        p->init_wallet();
        players.push_back(p);
    }

    return players;
}

int Player::m_obj_count{0};

Player::Player()
{
    m_obj_count++;
    m_id = m_obj_count;
}

Player::Player(unsigned int id, std::string name, int wallet, bool jail, int pos, int turns_in_jail, bool jail_card)
    :m_id(id), m_name(name), m_wallet(wallet), m_in_jail(jail), m_pos(pos), num_turns_in_jail(turns_in_jail), has_jail_card(jail_card)
{
```

Konstruktori klase *Player* i metod *initializePlayers*

Konstruktor koji ne prihvata nijedan argument ne inicijalizuje ovu vrednost. Metodom *initializePlayers* se inicjalizuje lista igrača u partiji. (u to se možemo uveriti uvidom u konstruktor klase *Game*) U okviru te metode se upravo za svakog igrača poziva konstruktor bez argumenata, tako da je jasno odakle potiče ova greška i šta bi trebalo u kodu ispraviti.

Druga detektovana greška je kreiranje neinicijalizovanih vrednosti na hipu. To u stvari upravo potiče od dodavanja neinicijalizovanih objekata klase *Player* u listu igrača.

Dalje, u izveštaju, u delu *HEAP SUMMARY* možemo videti podatke o ukupnoj alociranoj memoriji, memoriji koja nije bila oslobođena pri završetku programa, kao i ukupnom broju alokacija i oslobađanja memorije.

```
==1628==
==1628== HEAP SUMMARY:
==1628==     in use at exit: 6,640,284 bytes in 14,051 blocks
==1628==   total heap usage: 997,991 allocs, 983,940 frees, 729,179,966 bytes allocated
==1628==
```

Sekcija *HEAP SUMMARY* izveštaja

Posle ove sekcije se najvećim delom izveštaja nalaze podaci o svim izgubljenim, indirektno izgubljenim, potencijalno izgubljenim i još uvek pristupačnim blokovima memorije. Isto za svaki blok u formi steka, počev od funkcije kojom se alocira taj blok memorije i zatim funkcije u kojoj je alociran taj blok memorije. Iz kratkog uvida se može videti da veliki deo tih potiču od poziva *QT* funkcija i drugih bibliotečkih funkcija koje su direktno ili indirektno (često preko funkcije *strdup()*) alociraju memoriju na hipu. Pretražićemo datoteku i analizirati blokove alocirane od strane metoda implementiraih klasa.

Možemo videti da postoje delovi koda koji bi se trebalo popraviti. Pogledajmo podatke o jednom od definitivno izgubljenih blokova:

```
==1628== 557 (288 direct, 269 indirect) bytes in 4 blocks are definitely lost in loss record 2,326 of 2,476
==1628==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==1628==    by 0x1193EB: Card::initialize_cards() (in /home/user/milena/Verifikacija/RS019-monopol/RS019-monopol)
==1628==    by 0x116CFF: Board::Board() (in /home/user/milena/Verifikacija/RS019-monopol/RS019-monopol)
==1628==    by 0x11B13C: Game::Game(std::vector<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >,
std::allocator<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >) (in /home/user/milena/Verifikacija/RS0
RS019-monopol)
==1628==    by 0x130A38: MainWindow::MainWindow() (in /home/user/milena/Verifikacija/RS019-monopol/RS019-monopol)
==1628==    by 0x114FBF: main (in /home/user/milena/Verifikacija/RS019-monopol/RS019-monopol)
==1628==
```

Podaci o jednom od definitivno izgubljenih blokova memorije

U pitanju su objekti klase *Card* koji predstavljaju karte šansi ili iznenađenja. Pokazivači na te objekte se čuvaju u nizu u okviru objekta klase *Board* i predstavljaju karte koje igrači tokom igre izvlače sa gomile. Šta se dešava sa kartom kada je korisnik izvuče sa gomile? Metodom *drawChanceCard()* igrač izvlači kartu šanse sa gomile. (potpuno je identična implementacija za izvlačenje karte iznenađenja)

```
Card Board::drawChanceCard() {
    Card result = *m_chanceDeck.at(0);
    m_chanceDeck.erase(m_chanceDeck.begin());
    //TODO: if not jail card put back, else add the card to player
    //TODO: remove magic nums
    if(result.getAction() != 7){
        m_chanceDeck.push_back(&result);
    }
    return result;
}
```

Implementacija metoda *drawChanceCard()*

Kada korisnik izvuče kartu sa gomile, odgovarajući član niza se briše. Član niza je u stvari pokazivač na dinamički alociran objekat klase *Card* (metodom *new()*) i blok memorije na koju je pokazivao nije bio prethodno oslobođen zbog čega je veza ka tom delu izgubljena. Tako da je to izvor curenja memorije i rešenje bi bilo samo da se u odgovarajući metod pre brisanja člana niza oslobodi memorija na koju je pokazivao.

Poslednja sekcija izveštaja je *LEAK SUMMARY* u kome možemo videti ukupnu količinu definitivno izgubljene memorije, indirektno izgubljene memorije, potencijalno izgubljene memorije i memorije kojoj još uvek možemo pristupiti.

I na samom kraju izveštaja je dat ukupan broj pronađenih grešaka.

```
==1628==
==1628== LEAK SUMMARY:
==1628==    definitely lost: 1,120 bytes in 19 blocks
==1628==    indirectly lost: 5,955,430 bytes in 3,386 blocks
==1628==      possibly lost: 64,580 bytes in 2 blocks
==1628==    still reachable: 619,154 bytes in 10,644 blocks
==1628==         suppressed: 0 bytes in 0 blocks
==1628==
==1628== For lists of detected and suppressed errors, rerun with: -s
==1628== ERROR SUMMARY: 11 errors from 10 contexts (suppressed: 2 from 2)
```

Poslednja sekcija izveštaja i zaključni deo

3.2 Massif

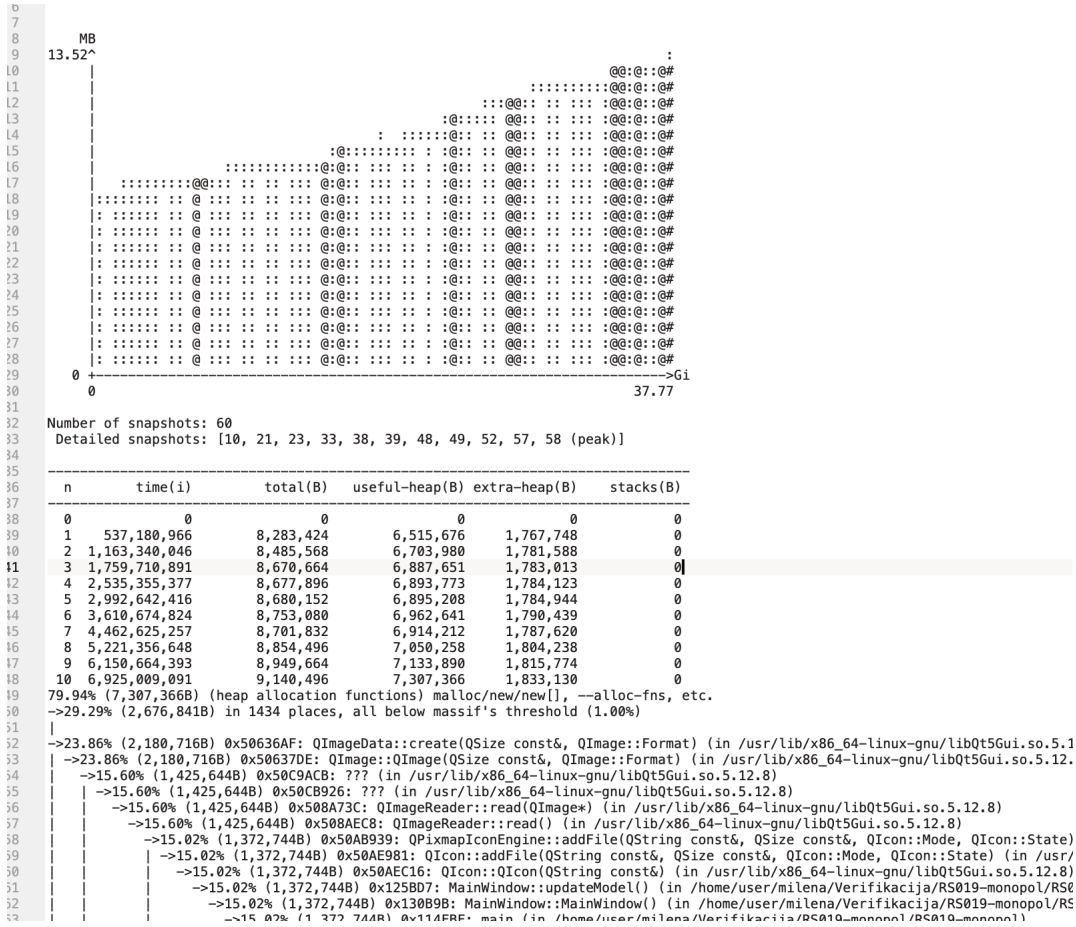
Sledeći alat za profajliranje koji smo upotrebili je *Massif*. *Massif* je alat koji prati stanje upotrebe *heap* segmenta. (i *stack* segmenta ukoliko zadamo odgovarajuću opciju) Prednost ovog alata je u tome što on detektuje memoriju koja faktički nije iscurela, ali se uopšte ne upotrebljava i samo zauzima prostor koji bi mogao da bude bolje iskorišćen. Takođe, u izveštaju dobijenim korišćenjem ovog alata se dobija lepa grafička reprezentacija koja može da nam da lep uvid u distribuciju korišćenja memorije na hipu tokom vremena.
Alat *Massif* pokrećemo istom komandom kao *Memcheck* uz opciju –tool=massif čime naglašavamo da koristimo taj alat:

```
$ valgrind --tool=massif --main-stacksize=4000000000 --massif-out-file=massif-output1.tx
```

Opcijom –main-stacksize smo regulisali veličinu steka, a opcijom –massif-out-file smo zadavali datoteku u koju smo želeli da se upiše izlaz iz alata.
Pošto smo dobili izlaz iz alata u datoteci *output1.txt*, potrebno je generisati izveštaj komandom *ms_print*:

```
$ ms_print massif-output1.txt > ms_massif-output1.txt
```

Dobijeni izveštaj izgleda ovako:



Izveštaj rada *Massif* alata

Prvi deo izveštaja je graf na koji možemo gledati kao na histogram ili stubasti dijagram. x - osa predstavlja vreme (podrazumevano izraženo u broju instrukcija), a y - osa ukupnu količinu memorije na hipu koju program zauzima u datom vremenskom trenutku. Naravno, nije moguće zabeležiti svaki trenutak izvršavanja, pa ako se program duže izvršava onda se deo zabeleženih podataka odbacuje. (pošto se program izvršava duže, dovoljno ima zabeleženih trenutaka izvršavanja do tada)

Uz svaki zabeleženi trenutak je stubićem na dijagramu obeleženo koliko je memorije na hipu bilo zauzeto u tom trenutku. Izgled stubića se razlikuje - samo u tome što je za trenutke popunjene znakom @ na taj način naznačeno da su o njima date detaljnije informacije. Specijalno, stubić za jedan trenutak je popunjen znakom - to je "*peak*"trenutak u kome je bilo alocirano najviše memorije. (ali koja je bila posle dealocirana) I za taj trenutak izvršavanja je prikupljeno više informacija.

Ispod grafika se nalazi informacija o ukupnom broju zabaleženih trenutaka izvršavanja, kao i zabeleženi trenuci izvršavanja za koje su zabeležene detaljne informacije i "peak"trenutak.
U nastavku do kraja izveštaja su podaci o svim zabeleženim trenucima izvršavanja. Za svaki trenutak izvršavanja je zabeležena ukupna veličina zauzete memorije na hipu (u bajtovima), količina te memorije koja se zaista u stvari koristi i dodatna količina memorije na hipu koja je neophodna zbog meta-informacija o alociranim blokovima kao i samog poravnanja memorije. Za trenutke za koje su zabeležene detaljnije informacije su obeležene funkcije koje su alocirale najveći procenat te zauzete memorije.

3.2.1 Dobijeni rezultati

Za dobijenog grafika možemo da primetimo da se ukupna količina memorije na hipu postepeno povećava tokom izvršavanja. Na slici smo prikazali izveštaj posle najdužeg izvršavanja programa. (a izvršavali smo program više puta, a izabrali smo 3 izveštaja koje smo ostavili u direktorijumu *Massif*)

Pretragom kroz izveštaj smo primetili da od metoda iz našeg projekta najviše alociranja imaju metodi klase *MainWindow*, tako da bi ti metodi mogli da se razmotre za potencijalne izmene.

3.3 Callgrind

Još jedan *Valgrind*-ov alat koji smo koristili u analizi je *Callgrind*. *Callgrind* je alat koji generiše graf poziva funkcija i na osnovu tog grafa računa koliko je puta neka funkcija pozvala neku drugu funkciju. Na osnovu podataka iz grafa računa koje funkcije imaju najveću cenu poziva i na osnovu tih podataka možemo da razmotrimo na koji način bismo mogli da refaktorišemo naš kod. (npr. da iz funkcije koja se često poziva izbacimo veliki broj poziva neke skupe funkcije)

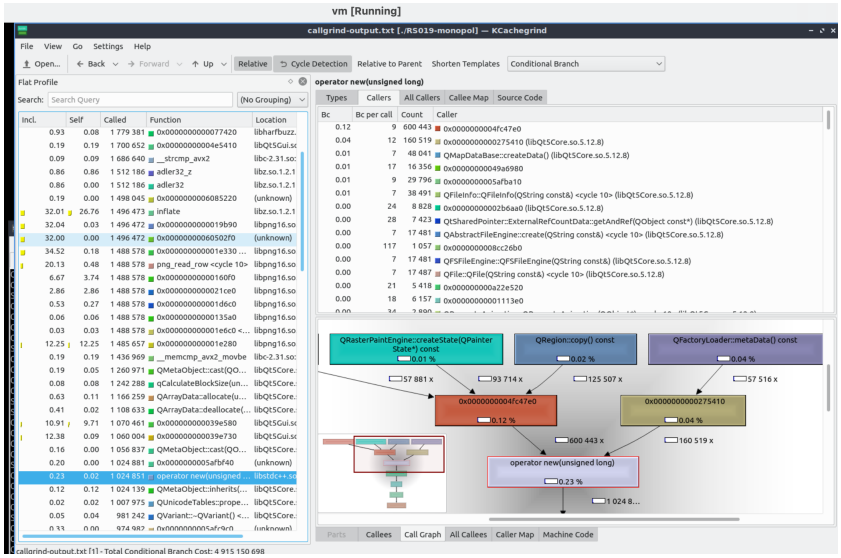
Valgrind se poziva sledećim komandom. Opcijom –callgrind-out-file zadajemo željenu datoteku u kojoj želimo da bude sačuvan izlaz programa:

```
$ valgrind --tool=callgrind --callgrind-out-file=callgrind-output.txt ./RS019-Monopol
```

Dobijeni izlaz bi trebalo da pregledamo kao izveštaj u čitljivijem formatu. Mi smo za prilike analize iskoristili alat sa GUI-jem *Kcachegrind*.

```
$ kcachegrind callgrind-output.txt
```

Na slici možemo videti prikaz profila u *Kcachegrind*-u:



Prikaz profila u *Kcachegrind* alatu

3.3.1 Dobijeni rezultati

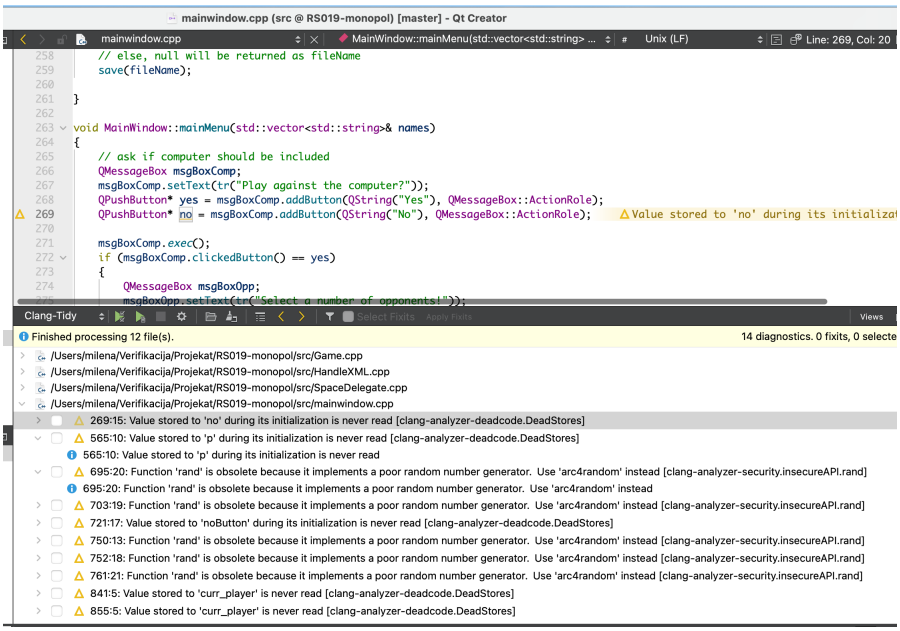
U slučaju projekta koji smo analizirali informacije koje je dao ovaj alat nisu od pomoći pri pronalaženju slabih tačaka u aplikaciji i/ili unapređivanju performansi. Nijedna od metoda klasa koje su implementirane nije direktno pozivala neku od najčešće pozivanih funkcija.

4 Statička analiza softvera

Za kraj ćemo iskoristiti statički analizator *Clang-Tidy* koji je dostupan u okviru *QT-Creator*-a.
Clang-Tidy je jedan od dodatnih *Clang* alata. Kao *Clang* alat je deo projekta *LLVM*.

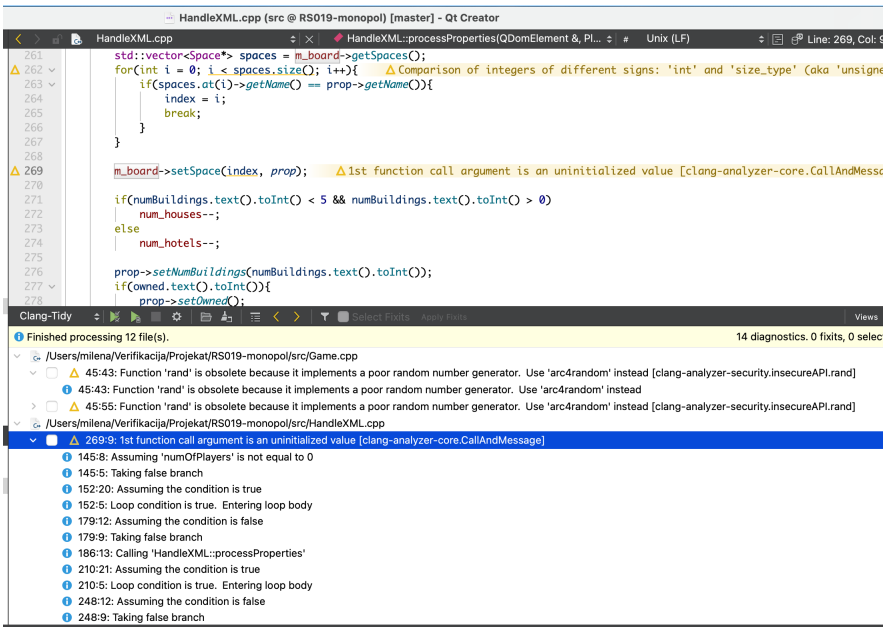
Statički analizator vrši analizu izvornog koda bez prevođenja i pokušava da pronađe nedostatke u kodu, kao što su na primer, upotreba neinicijalizovane promenljive ili greške koje se mogu detektovati bez procesa prevođenja.

Pokrenućemo analizator za sve izvorne datoteke u okviru projekta. Prvo ćemo prikazati izveštaj dobijen za izvornu datoteku *mainwindow.cpp* pošto smo tu dobili najviše sugestija za popravku koda. Slike ostalih izveštaja su dostupne u direktorijumu *StaticAnalysis*.



Rezultati statičke analize alata *Clang-Tidy* za datoteku *mainwindow.cpp*

Kao što možemo videti, rezultat rada statičkog analizatora je niz poruka kojim je ukazano na nedostatke u kodu i u nekim predlozi za poboljšanje koda. Vidimo da u ovoj datoteci postoji više promenljivih čije vrednosti nikada nisu iskorišćene. Takođe, dat je i predlog za upotrebu boljeg generatora pseudoslučajnih vrednosti. Ova poruka se javlja i u izveštajima za druge datoteke u kojima se koristi ovaj pseudoslučajni generator. Pored ovih poruka u izveštajima se javlja i upozorenje o upotrebi neinicijalizovane vrednosti.



Rezultati statičke analize alata *Clang-Tidy* za sve preostale datoteke kod kojih su nađeni nedostaci u kodu

5 Zaključak

Tokom analize program je pokazao očekivano ponašanje u skladu sa specifikacijom. Takođe, prošao je sve jedinične testove, ali kao što smo već napomenuli, oni nisu garancija ispravnosti.
Čak i ako je program zaista potpuno korektan, postoje nedostaci na kojima bi se trebalo raditi. To se prvenstveno odnosi na curenje memorije koje može drastično da naruši performanse posle dužeg izvršavanja programa. Takođe, *Memcheck*-om i statičkim analizatorom smo videli da su i u izvornom kodu potrebne ispravke. (prvenstveno kod korišćenja neinicijalizovanih vrednosti)

Dodali bismo još jedan aspekt u kome ima prostora za unapređivanje, a koji je drugačije prirode i koji nije detektovan alatima za verifikaciju. Projektu koji smo analizirali nedostaje dokumentovanost, većina izvornih datoteka nije komentarisana, ne postoji nikakav oblik tekstualnog dokumenta ni UML-a. Iako su promenljive, klase i metode u kodu na jasan i konzistentan način imenovane, trebalo je više vremena za sam pregled koda. Sa dovoljnom dokumentovanošću kod je dovoljno jednostavan za pregled i analizu i od strane programera koji uopšte nije ni radio na njemu.

Svakako je bilo zadovoljstvo rad analiza ovog lepog projekta i kroz ceo proces analize je mnogo naučeno i stečeno vredno praktično iskustvo.