

# Analiza i Verifikacija softvera

## UNO

Marija Ristić  
mi241008@alas.matf.bg.ac.rs

9. oktobar 2025.

## Sadržaj

<b>1</b>	<b>Uvod</b>	<b>2</b>
<b>2</b>	<b>Formatiranje koda</b>	<b>2</b>
2.1	Clang-format . . . . .	2
2.2	Analiza dobijenih rezultata . . . . .	3
<b>3</b>	<b>Statička analiza koda</b>	<b>4</b>
3.1	CodeChecker . . . . .	4
3.2	Analiza dobijenih rezultata . . . . .	4
<b>4</b>	<b>Analiza pokrivenosti koda</b>	<b>6</b>
4.1	Lcov . . . . .	8
4.2	Analiza dobijenih rezultata . . . . .	8
<b>5</b>	<b>Dinamička analiza koda</b>	<b>9</b>
5.1	Valgrind Memcheck . . . . .	9
5.2	Analiza dobijenih rezultata . . . . .	9
<b>6</b>	<b>Zaključak</b>	<b>12</b>

# 1 Uvod

Za potrebe kursa *Verifikacija Softvera*, analiziran je projekat **UNO**. Projekat je razvijen u programskom jeziku C++ kao GUI aplikacija, sa klijent-server arhitekturom koja koristi POSIX socket API za mrežnu komunikaciju.

Aplikacija predstavlja multiplayer kartašku igru koja se može igrati protiv botova ili pravih igrača povezanih preko servera, sa ukupno 2, 4 ili 8 učesnika. Cilj igre je da igrač ostane bez svih karata u ruci, dok istovremeno zarađuje poene tako što postavlja karte na talon. Na kraju igre, ukupan broj osvojenih poena određuje konačni rezultat svakog igrača.

Cilj ove analize je da se proceni pouzdanost, efikasnost i kvalitet implementacije kroz upotrebu savremenih alata za statičku i dinamičku analizu softvera. U okviru rada korišćeni su sledeći alati:

- ***Clang-format*** - za formatiranje koda
- ***CodeChecker*** - za statičku analizu koda
- ***Lcov*** - za merenje pokrivenosti koda testovima
- ***Valgrind Memcheck*** - za detekciju problema u radu sa memorijom tokom izvršavanja.

Analiza i dobijeni rezultati omogućavaju uvid u potencijalne slabosti projekta i daju preporuke za dalje unapređenje koda, sa ciljem povećanja njegove stabilnosti, efikasnosti i dugoročne održivosti.

## 2 Formatiranje koda

Formatiranje koda je proces automatskog uređivanja izgleda izvornog koda prema definisanim stilskim pravilima, čime se postiže doslednost, bolja čitljivost i lakše održavanje softvera. Najčešće korišćeni alati za automatsko formatiranje koda su Clang-Format za C++, Black za Python, Prettier za JavaScript, HTML i CSS, dotnet-format za C# i google-java-format za Java — svaki od njih primenjuje standardizovane stilove pisanja koda i olakšava doslednost u timskom radu.

### 2.1 Clang-format

Clang-Format je alat za automatsko formatiranje izvornog koda koji podržava više programskih jezika, uključujući C, C++, Java, JavaScript i Objective-C. Razvijen kao deo LLVM projekta, Clang-Format omogućava dosledno stilizovanje koda prema unapred definisanim stilovima kao što su LLVM, Google, Mozilla, WebKit i Microsoft, ali takođe pruža mogućnost detaljnog prilagođavanja putem konfiguracionog fajla `.clang-format`. Korisnici mogu precizno kontrolisati izgled zagrada, razmaka, poravnanja, komentara i drugih sintaktičkih elemenata, čime se olakšava timski rad, smanjuju stilističke razlike i unapređuje čitljivost koda. Alat se lako integriše u razvojna okruženja i može se koristiti kao deo automatskih skripti ili CI/CD procesa.

- ***-i*** - formatira fajl *in-place*, tj. direktno menja sadržaj fajla.
- ***-style=llvm*** - koristi ugrađeni stil kao što su `llvm`, `google`, `webkit`, `mozilla`, `microsoft`.
- ***-style=file*** - učitava stil iz lokalnog `.clang-format` fajla.
- ***-fallback-style=none*** - ne koristi podrazumevani stil ako `.clang-format` nije pronađen.
- ***-assume-filename=ime.cpp*** - pretpostavlja ekstenziju fajla (korisno kada se formatira iz `stdin`).
- ***-lines=5:10*** - formatira samo linije od 5 do 10.
- ***-dry-run*** - prikazuje koje bi promene bile napravljene, ali ne menja fajl.
- ***-Werror*** - prijavljuje greške ako formatiranje nije u skladu sa zadatim stilom.
- ***-verbose*** - prikazuje dodatne informacije tokom izvršavanja.

```

49,51c45,47
<     if (inet_pton(AF_INET,
<         _serverAddress.toString().c_str(),
<         &server_addr.sin_addr) <= 0) {
< ---
>     if (inet_pton(AF_INET, _serverAddress.toString().c_str(),
>         &server_addr.sin_addr) <= 0)
> {

```

Slika 1: Primer upotrebe komande diff nad fajlom UNO/sources/client.cpp.

```

57,62c58,61
< GameGUI::GameGUI(QWidget* parent,
<     GameParameters* gameParams,
<     Server* server_,
<     Client* client_)
< : QWidget(parent)
< , ui(new Ui::TESTUI)
< ---
> GameGUI::GameGUI(QWidget *parent, GameParameters *gameParams, Server *server_,
>     Client *client_)
> : QWidget(parent)
> , ui(new Ui::TESTUI)

```

Slika 2: Primer upotrebe komande diff nad fajlom UNO/sources/gameGUI.cpp.

Fajl *.clang-format* je konfiguracioni fajl koji definiše pravila formatiranja koda za alat Clang-Format, a piše se u YAML sintaksi. Sva pravila formatiranja koja se mogu definisati nalaze se u clang-format [dokumentaciji](#). Pre verzije 14.0 opcija `-style=file` podrazumevala je da se fajl *.clang-format* nalazi u istom direktorijumu kao i fajl koji se formatira. U slučaju da ga ne pronade u istom direktorijumu, pretražuje rekurzivno roditeljske direktorijume sve dok ne dodje do roditeljskog direktorijuma. Nakon verzije 14.0 [4] dozvoljeno je da se uz style opciju definiše i putanja do *.clang-format* fajla na sledeći način `-style=file:<format_file_path>`.

## 2.2 Analiza dobijenih rezultata

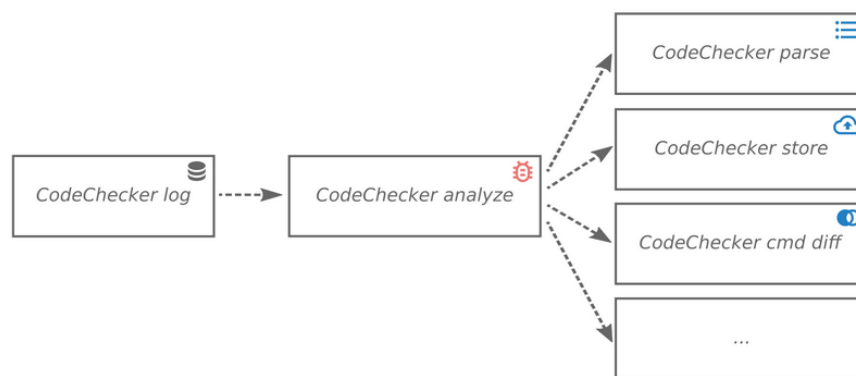
U okviru *clang-format* direktorijuma nalazi se *.clang-format* fajl u kome su definisana sledeća pravila formatiranja:

- **Language: Cpp** - definiše jezik za koji se primenjuju pravila (C++), korisno kada projekat koristi više jezika
- **BasedOnStyle: Mozilla** - definiše polazni način formatiranja čiji se parametri mogu pregaziti eksplicitnim definisanjem (na primer: indentacija koristi 2 razmaka, komentari se formatiraju sa razmakom nakon `//`) [5]
- **IndentWidth: 4** - definiše broj razmaka za indentaciju
- **AlignAfterOpenBracket: Align** - poravnava parametre funkcije ili liste nakon otvorene zagrade
- **DerivePointerAlignment: false** - ne koristi automatski stil poravnanja pokazivača, već koristi pravilo definisano u `PointerAlignment`
- **PointerAlignment: Left** - pokazivači se pišu sa zvezdicom uz tip, a ne uz ime promenljive.

Nakon pokrenutog clang-format alata nad svim .cpp i .h fajlovima projekta, formatirani fajlovi se smeštaju u poseban izlazni direktorijumu *clang-format/clang-format-output/*, pri čemu se zadržava ista relativna struktura putanja kao u originalnom projektu.

Pokretanjem komande:

`diff clang-format/clang-format-output/relativna_putanja_do_fajla 12-UNO/relativna_putanja_do_fajla` mogu se videti razlike koje su napravljene formatiranjem. Na slikama 1 i 2 mogu se videti rezultati za fajlove *UNO/sources/client.cpp* i *UNO/sources/gameGUI.cpp*.



Slika 3: CodeChecker proces analize.

## 3 Statička analiza koda

Statička analiza koda je proces automatskog ispitivanja izvornog koda bez njegovog izvršavanja, sa ciljem da se otkriju greške, sigurnosni propusti, neefikasnosti i kršenja stilskih pravila. Koristi se u ranim fazama razvoja kako bi se poboljšala pouzdanost i održivost softverskog sistema.

### 3.1 CodeChecker

CodeChecker je napredan alat za statičku analizu C i C++ koda, zasnovan na LLVM/Clang infrastrukturi. Omogućava automatsko otkrivanje grešaka, sigurnosnih propusta i stilskih odstupanja kroz više analizatora [2] kao što su Clang-Tidy, Cppcheck i GCC Static Analyzer. Rezultati se mogu pregledati u komandnoj liniji ili kroz statički HTML izveštaj, uz podršku za inkrementalnu analizu - analiziraju se samo izmenjeni fajl i njegove zavisnosti - i upoređivanje grešaka koje su uvedene od poslednjeg pokretanja analizatora.

Na slici 3 prikazan je proces rada CodeChecker alata. Proces analize koda se započinje komandnom log, koja pokreće izgradnju projekta (npr. pomoću make) i beleži sve korake kompilacije u JSON fajl poznat kao kompilaciona baza. Zatim se koristi komanda `analyze`, koja na osnovu te baze vrši statičku analizu koda i generiše rezultate u mašinski čitljivom formatu (plist) pogodnom za dalju obradu. U završnoj fazi, korisnik može parsirati i pregledati rezultate (parse), sačuvati ih na CodeChecker server (store), uporediti različite analize (cmd diff), kao i koristiti druge funkcionalnosti za detaljnu obradu i vizualizaciju nalaza.

Lista opcija sa kojima se može pokrenuti CodeChecker alat detaljnije se može dobiti pokretanjem komande `CodeChecker {command} --help` u terminalu. [1]

### 3.2 Analiza dobijenih rezultata

Pre pokretanja CodeChecker alata bilo je neophodno napraviti promene u `CMakeLists.txt` fajlu projekta koje su zabeležene u `patches/qt_fix.patch`.

U bash skripti, koja se nalazi u direktorijumu `codeChecker`, se najpre aktivira virtuelno okruženje za CodeChecker alat, a zatim se podešava `PATH` promenljiva kako bi se omogućio pristup njegovim izvršnim fajlovima. Sledeće, pokreće se statička analiza koda pomoću `CodeChecker check`, koja obuhvata logovanje kompilacije i analizu, a rezultati se smeštaju u `codeChecker/results/` direktorijum. Na kraju se rezultati analiziranog koda parsiraju i konvertuju u HTML format radi lakšeg pregleda, nakon čega se virtuelno okruženje deaktivira. Pokretanjem `index.html` datoteke, koja se nalazi u `codeChecker/report.html/` dobija se HTML stranica sa detaljnim izveštajem grešaka koje je pronašao CodeChecker alat.

Na slici 4 prikazana je statistika grešaka koje je CodeChecker alat uspeo da pronadje. Iz priložene statistike se može primetiti da CodeChecker klasifikuje greške po tipu ozbiljnosti (visok, srednji i nizak) i takođe daje statistiku koliko je koji analizator našao grešaka. Na početnoj HTML stranici za svaku grešku možemo pronaći: fajl i liniju u kojoj se greška nalazi, nivo rizičnosti te greške, checker alat koji

[Go To Bug List](#)

## Statistics

Number of processed analyzer result files 18

Number of analyzer reports 72

## Checker statistics

Checker name	Severity	Number of reports
bugprone-signed-char-misuse	M	2
bugprone-switch-missing-default-case	L	1
clang-diagnostic-double-promotion	M	1
clang-diagnostic-reserved-identifier	M	6
clang-diagnostic-unused-parameter	M	2
clang-diagnostic-unused-variable	M	1
core.CallAndMessage	H	2
core.NullDereference	H	1
cppcheck-uninitMemberVar	M	56

## Severity statistics

Severity	Number of reports
H	3
M	68
L	1

Slika 4: CodeChecker statistika grešaka.

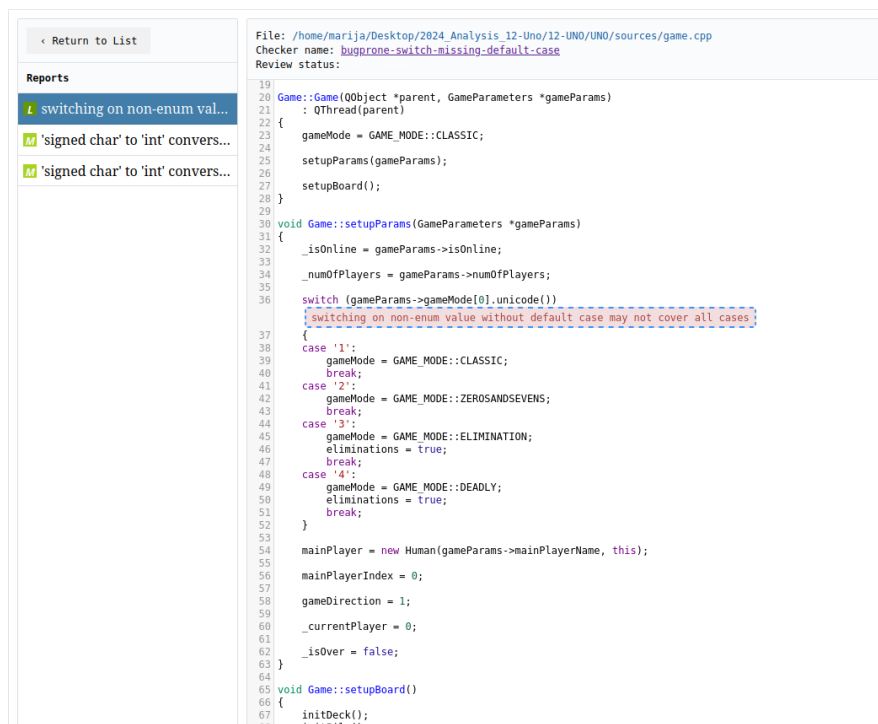
je uspeo da detektuje tu grešku, poruku o grešci, broj koraka od početne tačke do mesta gde se greška dešava, status greške.

Vikoko rizičnih grešaka ima tri i na slici 5 možemo videti više detalja o ovim greškama. Uočiti da su sve tri visoko rizične greške detektovane u spoljnim zavisnostima koje se koriste u projektu. Takve greške se često javljaju jer analizatori nemaju pun kontekst (na primer makroi, specifične konfiguracije, platformske optimizacije), pa mogu prijaviti lažne pozitivne nalaze u bibliotekom kodu.

Srednje rizičnih grešaka pronadjeno je 68. Najveći broj ovih grešaka pronadjeno je uz pomoć `cppcheck-uninitMemberVar` i ukazuje da nije urađena inicijalizacija promenljive u konstruktoru. Ostale srednje rizične greške ukazuju na nekorišćene promenljive, pronadjene uz pomoć `clang-diagnostic-unused-parameter` i `clang-diagnostic-unused-variable`. Greške pronadjene po-

Go To Statistics						
	File	Severity	Checker name	Message	Bug path length	Review status
1	/home/maria/Ou-5.3.3/src_64/include/OuCore/qanovstrineview.h @ Line 121	H	core.NullDereference	Array access (from variable 'str') results in a null pointer dereference	13	Unreviewed
2	/usr/lib/dvm-18/lib/clang/18/include/cetintrin.h @ Line 49	H	core.CallAndMessage	1st function call argument is an uninitialized value	2	Unreviewed
3	/usr/lib/dvm-18/lib/clang/18/include/cetintrin.h @ Line 62	H	core.CallAndMessage	1st function call argument is an uninitialized value	2	Unreviewed

Slika 5: Detalji o visoko rizičnim greškama koje je CodeChecker pronašao.



Slika 6: Detalji o nisko rizičnoj grešci koju je CodeChecker pronašao.

moću **clang-diagnostic-reserved-identifier** odnose se na pravilo iz C++ standarda koje zabranjuje korišćenje identifikatora koji počinju sa donjom crtom (`_`) na globalnom nivou (van funkcija i klasa). Ovo pravilo je propisano jer su identifikatori koji počinju sa `_` na globalnom nivou rezervisani za implementaciju kompajlera i standardne biblioteke. Takođe, prijavljena je i jedna greška uz pomoć **clang-diagnostic-double-promotion** prilikom implicitne konverzije iz float u qreal. Još dve greške nastale zbog implicitne konverzije iz char u int, detektovana je uz pomoć **bugprone-signed-char-misuse**. Jer `std::string` koristi char tip, koji je na većini sistema signed i kada ga kastuješ u int, negativne vrednosti (npr. -1) ostaju negativne i zato da bismo dobili numeričku vrednost bajta (0–255), potrebno je koristiti unsigned char.

Jedina greška niskog nivoa ozbiljnosti ukazuje da fali default klauza u switch-case naredbi. Na slici 6 nalaze se detalji o pronađenoj grešci. Greška je detektovana pomoću **bugprone-switch-missing-default-case** alata.

Prilikom otklanjanja grešaka koje su pronađene CodeChecker alatom, fokus je bio na nisko rizičnoj grešci i greškama srednjeg nivoa rizičnosti. Greška niskog nivoa rizika razrešena je dodavanjem default klauzule u switch-case naredbi koja implementira izlazak iz aplikacije ako se udje u default granu. Korekcija greške se može naći u `patches/codeChecker_fixed_switch_case.patch`. Fajlovi `patches/codeChecker_fixed_char_to_int_conversion.patch`, `patches/codeChecker_fixed_reserved_identifier.patch` i `patches/codeChecker_fixed_unused.patch` pokazuju kako razrešiti srednje rizične greške koje se tiču implicitnog kastovanja (char tipa u int), refaktorisanja korisnički definisanih globalnih promenljivih koje počinju sa `_` i uklanjanja mrtvog koda. Nakon otklonjenih nekoliko detektovanih grešaka, ponovo je pokrenut alat i generisan html izveštaj. Na slici 7 je prikazana statistika grešaka u novom izveštaju.

## 4 Analiza pokrivenosti koda

Analiza pokrivenosti koda meri koliko je deo izvornog koda zaista izvršen tokom testiranja, pomažući programerima da identifikuju neaktivne ili neproverene delove aplikacije. Ova analiza je ključna za otkrivanje skrivenih grešaka i proveru pokrivenosti koda testovima.

---





[◀ Go To Bug List](#)

## Statistics



Number of processed analyzer result files 14

Number of analyzer reports	60
----------------------------	----

## Checker statistics

Checker name	Severity	Number of reports
clang-diagnostic-double-promotion		1
core.CallAndMessage		2
core.NullDereference		1
cppcheck-uninitMemberVar		56

## Severity statistics

Severity	Number of reports
	3
	57

Slika 7: CodeChecker statistika grešaka iz drugog izveštaja.

LCOV - code coverage report						
Current view: top level			Coverage		Total	Hit
Test: coverage_filtered.info			Lines: 16.5 %		2133	353
Test Date: 2025-10-09 01:23:51			Functions: 27.6 %		312	86
Directory	Line Coverage %			Function Coverage %		
	Rate	Total	Hit	Rate	Total	Hit
headers	33.3 %	3	1	0.0 %	1	0
src	16.5 %	2130	352	27.7 %	311	86

Generated by: LCOV version 2.0-1

Slika 8: LCOV statistika pokrivenosti koda testovima.

## 4.1 Lcov

Lcov je alat za prikupljanje i vizualizaciju podataka o pokrivenosti koda u projektima koji koriste GCC i gcov. Omogućava generisanje HTML izveštaja koji jasno prikazuju koje linije koda su pokrivene testovima.

- **-capture** - prikuplja podatke o pokrivenosti iz .gcda fajlova
- **-directory <dir>** - definiše direktorijum sa objekt fajlovima
- **-output-file <file>** - određuje izlazni fajl za podatke o pokrivenosti
- **-remove <pattern>** - uklanja fajlove iz izveštaja
- **-extract <pattern>** - uključuje samo određene fajlove u izveštaj

## 4.2 Analiza dobijenih rezultata

Pre pokretanja lcov alata, u *external/catch2* direktorijumu dodato je *catch.hpp* zaglavlje biblioteke Catch2, koja se koristi za pisanje i pokretanje unit testova u C++ projektima. Takođe, napravljene su neophodne promene u *CMakeLists.txt* fajlu projekta koje su zabeležene u *patches/lcov\_cmake\_flags.patch*.

U bash skripti *cmake* komanda se pokreće u Coverage režimu. Komandom **ctest --output-on-failure** pokrenuti su svi testovi definisani u *CMake* projektu. Ako neki test padne, prikazuje se detaljan izlaz (log) tog testa. Takođe, uz pomoć **export GCOV=/usr/bin/gcov-10** postavljena je promenljiva okruženja GCOV da koristi verziju gcov-10, kako bi bilo kompatibilno sa projektom.

Pokrenut je lcov alat sa definisanim gcov-10 i sačuvani su rezultati u fajlu *coverage.info* unutar direktorijuma *coverage*. Kako ne bismo dobijali informacije o pokrivenosti fajlova koji se nalaze u tests ili *external* direktorijumima, filtriran je fajl *coverage.info* i rezultat je smešten u *coverage\_filtered.info*. Na osnovu *coverage\_filtered.info* fajla generisan je HTML izveštaj koji se nalazi na putanji *coverage/html/*. Na slici 8 prikazana je statistika pokrivenosti koda testovima koji se nalaze u projektu koju je izgenerisao LCOV alat.

U LCOV izveštaju u direktorijumu *headers* nisu prikazane sve .h datoteke jer alat za merenje pokrivenosti koda uzima u obzir samo one fajlove i linije koje sadrže stvarni izvršni kod, odnosno kod koji kompajler stvarno generiše prilikom izgradnje projekta. Većina .h datoteka sadrži samo deklaracije klase i funkcija, bez implementacije, pa samim tim ne sadrže linije koje LCOV može da registruje kao pokrivene ili nepokrivene. Headeri koji se ipak pojave u izveštaju obično imaju vrlo nisku ili nultu pokrivenost upravo zato što sadrže samo deklaracije ili apstraktne metode (npr. = 0), koje se ne izvršavaju. Konkretno, u *player.h* datoteci, LCOV je prepoznao jednu liniju koda kao merljivu jer funkcija *drawCard* ima podrazumevani argument (= 1). U C++ jeziku, podrazumevani argumenti moraju biti definisani u mestu deklaracije, što uzrokuje da kompajler generiše dodatne informacije u header fajlu — zbog toga je ta linija evidentirana i može biti pokrivena u toku testiranja, za razliku od ostalih funkcija koje nemaju slične karakteristike.

LCOV izveštaj prikazan na slici 9 pokazuje da je ukupna pokrivenost testovima u okviru fajlova iz *sources* direktorijuma relativno niska. Pokrivenost linija koda iznosi 16.5%, dok je pokrivenost funkcija nešto viša, na nivou od 27.7%. Detaljnija analiza po fajlovima pokazuje da su samo pojedini moduli, poput *deck.cpp* (97.1% linija, 91.7% funkcija) i *pile.cpp* (100% linija i funkcija), dobro pokrivene testovima. Ostali fajlovi, uključujući ključne komponente kao što su *game.cpp*, *client.cpp*, *server.cpp* i



LCOV - code coverage report

Current view: [top level - sources](#)

Test: coverage.filtered.info

Test Date: 2025-10-09 01:23:51

Lines: 16.3 %

Functions: 27.7 %

Total2130

Hit352

311

86

Filename	Line Coverage ↕			Function Coverage ↕		
	Rate	Total	Hit	Rate	Total	Hit
card.cpp	<div><div></div></div> 55.6 %	196	109	<div><div></div></div> 63.6 %	22	14
client.cpp	<div><div></div></div> 0.0 %	80		<div><div></div></div> 0.0 %	9	
deck.cpp	<div><div></div></div> 97.1 %	68	66	<div><div></div></div> 91.7 %	12	11
game.cpp	<div><div></div></div> 19.7 %	269	53	<div><div></div></div> 26.0 %	50	13
gameGUI.cpp	<div><div></div></div> 0.0 %	567		<div><div></div></div> 0.0 %	47	
gameclient.cpp	<div><div></div></div> 0.0 %	95		<div><div></div></div> 0.0 %	11	
gameserver.cpp	<div><div></div></div> 0.0 %	82		<div><div></div></div> 0.0 %	15	
human.cpp	<div><div></div></div> 42.9 %	28	12	<div><div></div></div> 58.3 %	12	7
instructionsWidget.cpp	<div><div></div></div> 0.0 %	14		<div><div></div></div> 0.0 %	4	
leaderboards.cpp	<div><div></div></div> 0.0 %	49		<div><div></div></div> 0.0 %	7	
lobby.cpp	<div><div></div></div> 0.0 %	120		<div><div></div></div> 0.0 %	11	
mainwindow.cpp	<div><div></div></div> 0.0 %	181		<div><div></div></div> 0.0 %	33	
rs.cpp	<div><div></div></div> 18.6 %	161	30	<div><div></div></div> 53.8 %	13	7
tile.cpp	<div><div></div></div> 100.0 %	30	30	<div><div></div></div> 100.0 %	11	11
player.cpp	<div><div></div></div> 60.9 %	69	42	<div><div></div></div> 52.9 %	34	18
server.cpp	<div><div></div></div> 0.0 %	98		<div><div></div></div> 0.0 %	10	
wildcard.cpp	<div><div></div></div> 43.5 %	23	10	<div><div></div></div> 50.0 %	10	5

Generated by: LCOV version 2.0-1

Generated by: LCOV version 2.0-1

Slika 9: LCOV statistika pokrivenosti sources fajlova testovima.

više GUI orijentisanih fajlova (gameGUI.cpp, mainWindow.cpp), imaju veoma nisku ili čak potpunu nultu pokrivenost. Ovakav rezultat ukazuje na potrebu za sistematičnijim pristupom pisanju testova, naročito za logiku igre i mrežnu komunikaciju, kako bi se unapredila stabilnost i održivost koda u celini.

## 5 Dinamička analiza koda

Dinamička analiza koda predstavlja tehniku ispitivanja programa tokom njegovog izvršavanja, kako bi se otkrile greške koje se ne mogu uočiti statičkom analizom — poput curenja memorije, neinicijalizovanih promenljivih i problema sa konkurentnošću. Ova analiza omogućava precizno praćenje ponašanja aplikacije u realnim uslovima.

### 5.1 Valgrind Memcheck

Valgrind Memcheck je alat za dinamičku analizu koji otkriva greške u upravljanju memorijom, kao što su curenja, pristupi neinicijalizovanoj memoriji i dvostruka oslobađanja. Koristi se najčešće za debugovanje C i C++ aplikacija i opcije sa kojima se može pokrenuti Valgrind alat navedene su u sledecoj listi:

- **-tool=memcheck** - koristi Memcheck alat podrazumevano, pored toga dostupne opcije opisane su u [dokumentaciji](#)
- **-leak-check=yes** - detaljno proverava curenje memorije.
- **-show-leak-kinds=all** - prikazuje sve vrste curenja (definitivna, indirektna, itd.).
- **-track-origins=yes** - prati poreklo neinicijalizovanih vrednosti (sporije, ali korisno)
- **-log-file=<ime\_fajla>** - zapisuje izlaz u fajl umesto na terminal
- **-verbose** - prikazuje dodatne informacije o izvršavanju
- **-gen-suppressions=all** - generiše pravila za potiskivanje grešaka
- **-suppressions=<ime\_supression\_fajla>** - potiskuje sve greske na osnovu pravila koje se nalaze unutar prosledjenog fajla.

### 5.2 Analiza dobijenih rezultata

Pre pokretanja alata izvršena su podešavanja CMakeLists.txt projekta kako bi cmake mogao da se pokrene u “Debug” režimu. Promene koje su izvršene nad CMakeLists.txt fajlu zabeležene su u

```

==22508==
==22508== 56 bytes in 1 blocks are still reachable in loss record 4,473 of 7,517
==22508== at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==22508== by 0x6B05B09: g_malloc (in /usr/lib/x86_64-linux-gnu/libglib-2.0.so.0.8000.0)
==22508== by 0xB046F96: ??? (in /usr/lib/x86_64-linux-gnu/libgtk-3.so.0.2409.32)
==22508== by 0xB050B02: gtk_entry_set_icon_from_pixbuf (in /usr/lib/x86_64-linux-gnu/libgtk-3.so.0.2409.32)
==22508== by 0xB0455B5: ??? (in /usr/lib/x86_64-linux-gnu/libgtk-3.so.0.2409.32)
==22508== by 0xB8BCE63: g_object_run_dispose (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.8000.0)
==22508== by 0xAEBCE7C: ??? (in /home/marija/Qt/6.5.3/gcc_64/plugins/platformthemes/libqgtk3.so)
==22508== by 0xAEB31C4: ??? (in /home/marija/Qt/6.5.3/gcc_64/plugins/platformthemes/libqgtk3.so)
==22508== by 0xAEBBF24: ??? (in /home/marija/Qt/6.5.3/gcc_64/plugins/platformthemes/libqgtk3.so)
==22508== by 0x52286C6: QGuiApplicationPrivate::~QGuiApplicationPrivate() (in /home/marija/Qt/6.5.3/gcc_64/lib/libQt6Gui.so.6.5.3)
==22508== by 0x5228518: QGuiApplicationPrivate::~QGuiApplicationPrivate() (in /home/marija/Qt/6.5.3/gcc_64/lib/libQt6Gui.so.6.5.3)
==22508== by 0x157380: main (main.cpp:7)
==22508==

```

Slika 10: Detalji o Qt grešci koju je Valgrind Memcheck pronašao.

patches/memckeck\_cmake\_flags.patch datoteci.

Svi rezultati pokretanja skripte *run\_memcheck.sh* nalaze se u direktorijumu *memcheck\_output*. Pri likom prvog pokretanja Valgrind alata *memcheck*, putem skripte *run\_memcheck.sh*, izlaz je smešten u *memcheck\_report\_01.txt*. Kako dobijena datoteka *memcheck\_report\_01.txt* ima preko sto hiljada linija, pri čemu se dosta pronađenih grešaka odnosi na Qt bibliotečke greške, napravljeni su koraci kako bi izveštaj bio čitljiviji. Prvi korak se odnosi na pravljenje skripte *run\_get\_all\_suppressions.sh* koja pokreće valgrindov *memcheck* alat sa opcijom *--gen-suppressions=all* i preusmerava izlaz u datoteku *memcheck\_all\_suppressions.txt*. U datoteci *memcheck\_all\_suppressions.txt* za svaku pronadje-nu grešku nalazi se njen opis, tačnije lanac poziva koji je proizveo grešku, a potom se ispod opisa izmedju i nalazi blok teksta koji opisuje pravilo za potiskivanje te greške. Napravljena je skripta *run\_extract\_qt\_suppression.sh* koja parsira datoteku *memcheck\_all\_suppressions.txt* tako sto izdvaja sa-mo relevantne blokove pravila za potiskivanje - one koji u svom opisu ne sadrže .cpp datoteke - i smešta ih u *qt\_supp* datoteku. Pokrećemo ponovo Valgrind-ov Memcheck alat, ali ovaj put sa uključenom opcijom *--suppressions=qt\_supp* čime dobijamo datoteku *memcheck\_report\_02.txt* koja sada sadrži oko 1200 linija izveštaja.

Na slici 10 može se primetiti da nisu uklonjene sve greške koje nastaju zbog korišćenja Qt razvojnog okruženja, ali je izveštaj *memcheck\_report\_02.txt* dosta čitljiviji. Greška 10 ukazuje da se radi o “still reachable” memoriji iz spoljnih biblioteka (kao što su GTK, Qt, GLib) i to je obično bezopasno — biblioteke često ne oslobađaju sve resurse jer se oslanjaju na OS da ih očisti pri izlasku.

Na slici 11 nalazi se izdvojena statistika o tipu greške koju je uspeo da pronadje Valgrindov Memc-check alat, broju bajtova i blokova koji je izgubljen tim tipom greške. U tabeli 1 pored tipa i opisa greške nalazi se i kolona “uzrok greške” koji ukazuje na delove projekta koji su prouzrokovali taj tip greške. Informacije u koloni “uzrok greške” dobijene su detaljnom analizom *memcheck\_report\_02.txt* datoteke.

Tip greške	Opis greške	Uzrok greške
Definitivno izgubljena memorija	stvarna curenja memorije koja nisu dostupna ni kroz jedan pokazivač	testovi
Indirektno izgubljena memorija	memorija izgubljena preko pokazivača koji su sami izgubljeni	testovi i kod projekta
Moguće izgubljena memorija	alokacija koje bi mogle biti curenje	/
Još uvek dostupna memorija	memorija koja nije oslobođena, ali je dostupna kroz pokazivače	Qt biblioteke
Potisnuta memorija	greške koje su ignorisane prema definisanim pravilima	Qt biblioteke

Tabela 1: Valgrind Memcheck uzroci grešaka.

Prilikom pokretanja alata Memcheck koriscenjem unit testova, neki od tetova nisu prošli. Na slici 12 je prikazan test koji nije prošao prilikom testiranja pile.cpp datoteke. U testu se porede dve in-

```

==22508==
==22508== LEAK SUMMARY:
==22508==    definitely lost: 13,816 bytes in 555 blocks
==22508==    indirectly lost: 2,730 bytes in 49 blocks
==22508==    possibly lost: 0 bytes in 0 blocks
==22508==    still reachable: 690 bytes in 8 blocks
==22508==                of which reachable via heuristic:
==22508==                  newarray      : 760 bytes in 5 blocks
==22508==    suppressed: 2,334,212 bytes in 22,869 blocks
==22508==
==22508== For lists of detected and suppressed errors, rerun with: -s
==22508== ERROR SUMMARY: 50 errors from 50 contexts (suppressed: 55 from 55)

```

Slika 11: Statistika pronađenih grešaka alatom Valgring Memcheck.

```

~~~~~
run_tests is a Catch v2.13.10 host application.
Run with -? for options

-----
Testing correctness of methods in class Pile
  If the pile is empty, method lastCard() should return a blank card
-----
/home/marija/Desktop/2024_Analysis_12-Uno/12-UNO/specification/tests/test_pile.cpp:115
.....

```

Slika 12: Greška pronađena alatom Valgrind Memcheck koja prouzrokuje pad testa.

stance klase `card.cpp`, pri čemu su obe instance napravljene pomoću podrazumevanog konstruktora. U dokumentaciji za C++ [3] nije definisano koju vrednosti uzimaju članice klase ako nisu prethodno inicijalizovane, stoga ovo poređenje nije bezbedno.

Prilikom otklanjanja grešaka koje su pronađene Valgrind-ovim Memcheck alatom, fokus je bio na indirektno izgubljenoj memoriji. Skripta `run_extract_indirectly_lost.sh` u fajl `indirectly_lost.txt` izdvaja sve blokove iz memcheck izveštaja koji ukazuju na to da je memorija indirektno izgubljena. Dve greške koje su inicirale indirektno izgubljenu memoriju koje se ne nalaze u testovima prikazane su na slikama 13 i 14. Daljom analizom utvrđeno je da je uzrok ovih grešaka to što se destruktorklase `Player` nikada ne poziva, dok klasa `Game` sadrži listu elemenata klase `Player`. Kako bi se otklonile ove greške eksplicitno se u konstruktoru klase `Game` poziva destruktorklase `Player` za svakog igrača koji se nalazi u listi koju čine instance klase `Player`. Nakon ove izmene, pokrenuta je ponovo skripta `run_memcheck.sh` i dobijeni izveštaj je smešten u `memcheck_report_03.txt`.

Dalji fokus je bio na otklanjanju indirektno izgubljene memorije u testovima koji se odnose samo na testiranje klase `Pile`. Stoga je skripta `run_extract_indirectly_lost.sh` unapređena da izdvaja blokove koji se odnose na indirektno izgubljenu memoriju samo u `test_pile.cpp` fajlu. Pronađena greška ukazuje da se u `test_pile.cpp` fajlu u 41. liniji nastali blokovi memorije koji nikad nisu oslobođeni. U pronađenom

```

==22508== 130 bytes in 1 blocks are indirectly lost in loss record 6,741 of 7,517
==22508==    at 0x484DB80: realloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==22508==    by 0x5CB8920: QArrayData::reallocUnaligned(QArrayData*, void*, long long, long long, QArrayData::AllocationOption) (in /home/marija/Qt/6.5.3/gcc_64/lib/Qt6Core.so.6.5.3)
==22508==    by 0x23358A: QTypedArrayData<Card*>::reallocUnaligned(QTypedArrayData<Card*>*, Card**, long long, QArrayData::AllocationOption) (qarraydata.h:117)
==22508==    by 0x232F8C: QtPrivate::QPodArrayOps<Card*>::realloc(long long, QArrayData::AllocationOption) (qarraydataops.h:259)
==22508==    by 0x2327F2: QArrayDataPointer<Card*>::reallocAndGrow(QArrayData::GrowthPosition, long long, QArrayDataPointer<Card*>*) (qarraydatapointer.h:208)
==22508==    by 0x232B1A: QArrayDataPointer<Card*>::detachAndGrow(QArrayData::GrowthPosition, long long, Card* const**, QArrayDataPointer<Card*>*) (qarraydatapointer.h:194)
==22508==    by 0x232208: void QtPrivate::QPodArrayOps<Card*>::emplace(Card*&)(long long, Card*&) (qarraydataops.h:174)
==22508==    by 0x231707: Card& QList<Card*>::emplaceBack<Card*&)(Card*&) (qlist.h:856)
==22508==    by 0x230E12: QList<Card*>::append(Card*) (qlist.h:433)
==22508==    by 0x230A08: QList<Card*>::push_back(Card*) (qlist.h:651)
==22508==    by 0x236746: Player::obtainCard(Card*) (player.cpp:52)
==22508==    by 0x235C08: Human::drawCard(int const&) (human.cpp:45)
==22508==

```

Slika 13: Greška pronađena alatom Valgrind Memcheck koja reprezentuje indirektno izgubljeni blok memorije.

```

==22508== 390 bytes in 3 blocks are indirectly lost in loss record 7,138 of 7,517
==22508== at 0x484DB80: realloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==22508== by 0x5CBB920: QArrayData::reallocateUnaligned(QArrayData*, void*, long long, long
long, QArrayData::AllocationOption) (in /home/marija/Qt/6.5.3/gcc_64/lib/libQt6Core.so.6.5.3)
==22508== by 0x23358A: QTypedArrayData<Card*>::reallocateUnaligned(QTypedArrayData<Card*>*,
Card**, long long, QArrayData::AllocationOption) (qarraydata.h:117)
==22508== by 0x232F8C: QtPrivate::QPodArrayOps<Card*>::reallocate(long long,
QArrayData::AllocationOption) (qarraydataops.h:259)
==22508== by 0x2327F2: QArrayDataPointer<Card*>::reallocateAndGrow(QArrayData::GrowthPosition,
long long, QArrayDataPointer<Card*>*) (qarraydatapointer.h:208)
==22508== by 0x232B1A: QArrayDataPointer<Card*>::detachAndGrow(QArrayData::GrowthPosition, long
long, Card* const**, QArrayDataPointer<Card*>*) (qarraydatapointer.h:194)
==22508== by 0x232208: void QtPrivate::QPodArrayOps<Card*>::emplace<Card*>(long long, Card*&)
(qarraydataops.h:174)
==22508== by 0x231707: Card*& QList<Card*>::emplaceBack<Card*&>(Card*&) (qlist.h:856)
==22508== by 0x230E12: QList<Card*>::append(Card*) (qlist.h:433)
==22508== by 0x230A08: QList<Card*>::push_back(Card*) (qlist.h:651)
==22508== by 0x236746: Player::obtainCard(Card*) (player.cpp:52)
==22508== by 0x23B716: PC::drawCard(int const&) (pc.cpp:27)
==22508==

```

Slika 14: Greška pronađena alatom Valgrind Memcheck koja reprezentuje indirektno izgubljeni blok memorije.

```

==100093== LEAK SUMMARY:
==100093== definitely lost: 13,312 bytes in 549 blocks
==100093== indirectly lost: 386 bytes in 12 blocks
==100093== possibly lost: 0 bytes in 0 blocks
==100093== still reachable: 690 bytes in 8 blocks
==100093== of which reachable via heuristic:
==100093== newarray : 760 bytes in 5 blocks
==100093== suppressed: 2,334,212 bytes in 22,869 blocks
==100093==
==100093== For lists of detected and suppressed errors, rerun with: -s
==100093== ERROR SUMMARY: 42 errors from 42 contexts (suppressed: 55 from 55)

```

Slika 15: Statistika pronađenih grešaka alatom Valgrind Memcheck, nakon nekoliko razrešenih grešaka.

testu pravi se novi talon karata, ali se taj talon nikada ne oslobađa. Kada je pozvan destruktore nad napravljenim talonom karata, u *memcheck\_report\_04.txt* izveštaju pronađena je nova greška koja govori o dvostrukom oslobađanju iste memorije u tom delu koda. Ustanovljeno je da se talon karata pravi dodavanjem jedne iste karte na kraj talona dok se ne dostigne odgovarajuća veličina talona. Prilikom poziva destruktora za talon karata, prolazi se kroz sve karte koje talon sadrži i za svaku kartu se eksplicitno poziva destruktore za nju - gde je i nastao problem jer talon čini ista karta dodavana više puta. Da bi se ovo razrešilo, napravljen je talon karata koji sadrži pokazivače na različite karte i potom je pozvan destruktore za taj talon.

U *memcheck\_report\_05.txt* izveštaju ostale su greške koje su pronađene Valgrind-ovim alatom Memcheck, ali nisu razrešene. Na slici 15 je prikazana statistika iz *memcheck\_report\_05.txt* izveštaja koja pokazuje drastično smanjeni broj izgubljenih bajtova registrovanih kao indirektno izgubljeni. Sve popravke koda koje su razrešile određeni broj grešaka nalaze se u *patches/memcheck\_fixed\_indirect\_lost.patch*.

## 6 Zaključak

Analiza projekta **12-UNO** kombinovanjem statičkog alata (*CodeChecker*), alata za merenje pokrivenosti koda testovima (*Lcov*), dimačke analize (*Valgrind Memcheck*) i formatiranja koda (*Clang-format*) obezbedila je sveobuhvatni uvid u kvaliteti pouzdanost implementacije.

Ključni rezultati analize ukazuju da je neophodno napisati dodatne testove koji bi povećali pokrivenost koda i time testirali veći broj komponenti sistema. Prilikom implementacije testova potrebno je više pažnje posvetiti curenju memorije koja nastaje tokom njihovog izvršavanja. Dodatno je potrebno izbegavati koriscenje default-nog konstruktora u testovima koji porede objekte, kako ne bi dolazilo do nedefinisog ponašanja koje prouzrokuje pad testova.

## Literatura

- [1] CodeChecker. CodeChecker command line features. on-line at: <https://codechecker.readthedocs.io/en/latest/#command-line-features>.
- [2] CodeChecker. CodeChecker supported analyzers. [https://codechecker.readthedocs.io/en/latest/supported\\_code\\_analyzers/](https://codechecker.readthedocs.io/en/latest/supported_code_analyzers/).
- [3] cppreference. Cppreference Default Initialization. [https://cppreference.net/cpp/language/default\\_initialization.html](https://cppreference.net/cpp/language/default_initialization.html).
- [4] LLVM. Clang 14.0.0 documentation, 2022. on-line at: <https://releases.llvm.org/14.0.0/tools/clang/docs/ReleaseNotes.html#clang-format>.
- [5] Mozilla. Mozilla- style options. <https://firefox-source-docs.mozilla.org/code-quality/coding-style/index.html>.