Analiza projekta Online Banking System

Ana Veličković, 1128/2025,
Matematički fakultet, Univerzitet u Beogradu

Januar 2026

Sadržaj

1 Uvod

Predmet ovog seminarskog rada je analiza softverskog rešenja Online Banking Systema, kao i primena različitih alata i tehnika kako bi se utvrdila ispravnost softvera. Aplikacija je pisana u C++ programskom jeziku i pokreće se iz terminala, pri pokretanju se otvara glavni meni gde se može izabrati jedan od sledećih modula: 1.login klijenta (trenutno omogućava klijentina uvid u stanje na računu i promenu lozinke), 2. login zaposlenog (pruža funkcionalnosti za rad sa klijentima, uključujući registraciju novih korisnika, pregled postojećih podataka i upravljanje njihovim profilima) i 3. login admina (omogućava adminu kreiranje, modifikaciju i brisanje naloga zaposlenih i klijenata, kao i uvid u kompletan bankovni sistem).

Kako bi se obezbedio visok stepen pouzdanosti, efikasnosti i sigurnosti sistema, primenjen je niz alata za statičku i dinamičku analizu koda:

- **Catch2**: Automatizovano unit testiranje modula radi provere logičke ispravnosti i validacije ulaza.
- **LCOV**: Alat za merenje i vizuelni prikaz pokrivenosti koda testovima (*code coverage*).
- **Clang-Tidy**: Statička analiza za proveru stila koda i usklađenosti sa modernim C++ standardima.
- **Cppcheck**: Detekcija dubljih logičkih grešaka i curenja resurasa koje standardni kompajler ne prepoznaje.
- **Valgrind (Memcheck)**: Dinamička analiza memorije fokusirana na pronalaženje curenja memorije (*memory leaks*) tokom izvršavanja programa.
- **Flawfinder**: Skeniranje izvornog koda u potrazi za potencijalnim sigurnosnim ranjivostima, poput *buffer overflow* rizika.
- **Iwyu (Include What You Use)**: alat za statičku analizu koji analizira **#include** direktive u C++ fajlovima.

2 Jedinični testovi

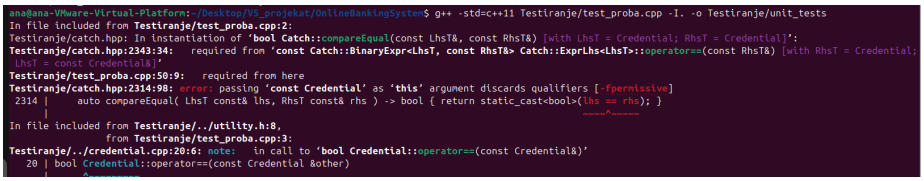
Jedinični testovi predstavljaju metodu testiranja softvera kojom se proveravaju najmanji delovi izvornog koda, poput pojedinačnih funkcija, metoda ili klasa, kako bi se potvrdila njihova ispravnost pre integracije u celinu. Cilj je izolacija svakog dela programa i potvrda da on ispunjava specifikacije i vraća očekivane rezultate za različite ulazne podatke.

Catch2 predstavlja open-source biblioteku namenjenu pisanju jediničnih testova u programskom jeziku C++. Njena glavna karakteristika je jednostavna i čitljiva sintaksa, koja omogućava programeru da kreira intuitivne testne scenarije bez potrebe za složenom konfiguracijom razvojnog okruženja. Catch2 automatski prikazuje rezultate testova, sa jasnim informacijama o uspehu, neuspehu, i razlozima neuspeha.

2.1 Analiza i primena LCOV alata

Automatsko testiranje realizovano je pomoću bash skripte run_tests.sh (nalazi se u unit_tests), koja služi za izvršavanje jedinčnih testova i pokretanje alata **lcov**. Alat lcov koristi se za merenje pokrivenosti izvornog koda testovima, omogućavajući identifikaciju linija koda koje su izvršene tokom testiranja, kao i onih koje nisu obuhvaćene testovima.

Tokom pokušaja pokretanja testova uočen je problem prilikom poređenja objekata klase Credential. Greška (Slika 1) je nastala zbog toga što operator poređenja (operator==) nije bio definisan kao const metoda. Catch2 prilikom evaluacije izraza za poređenje koristi konstantne objekte, tako da zahteva da operator ne menja stanje objekta nad kojim se poziva, zbog toga je neophodno izmeniti definiciju operatora == tako da bude const. Ovo omogućava poređenje konstantnih objekata kao i pokretanje testova. Takođe, postoji i drugi način, može se pri izvršavanju skripte dodati i `-fpermissive`, što omogućava kompilatoru da ignoriše neke vrste grešaka koje bi inače zaustavile kompilaciju. Korisno je kada je potrebno da kompilacija prođe bez obzira na neke manje greške, koje bi inače bile prijavljene kao fatalne, ali treba sa oprezom, jer može da sakrije stvarne probleme u kodu. Ovog puta je izabrana druga opcija, kako se ne bi dirao tuđi kod, ali je u nastavku predstavljena i moguće poboljšanje koda (na slici 1. je prikazana greška pri pokretanju testova, a ispod je moguće poboljšanje koda kako bi se pokrenuli testovi na prvi način).



Slika 1: Greška prilikom kompajliranja testova usled nekorektne upotrebe `const` kvalifikatora

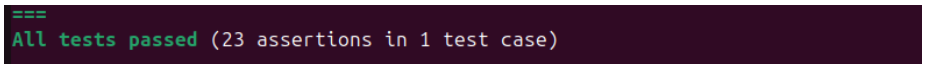
Kod pre izmene (prikazan je credential.h, ali je izmena i u credential.cpp):

```
bool Credential::operator==(const Credential &other);
```

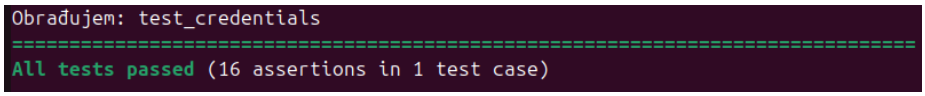
Kod nakon izmene koji se nalazi u folderu ispravljeni_fajlovi (prikazan je credential.h, ali je izmena i u credential.cpp):

```
bool Credential::operator==(const Credential &other) const;
```

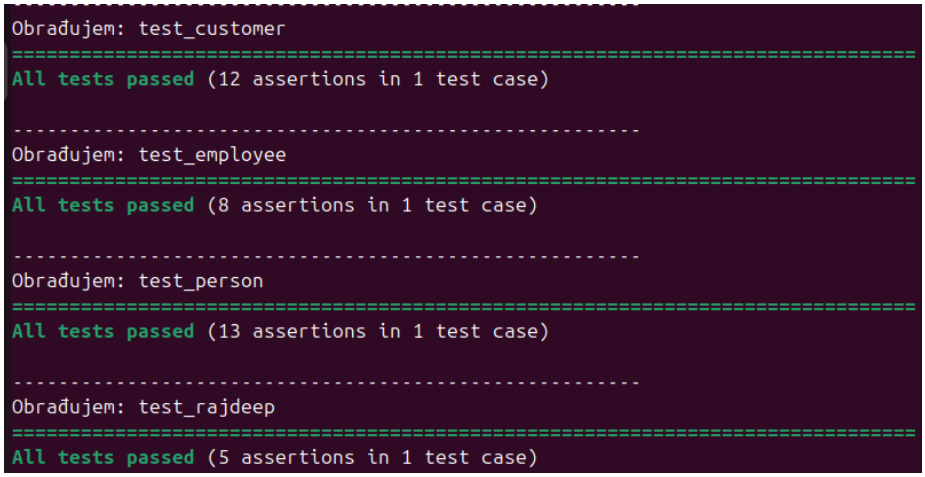
Testirane su klase **Controller**, **Credential**, **Customer**, **Employee**, **Person**, **Rajdeep**, **TimeStamp** i **Txn**. Implementacija nabrojanih klasa se nalazi u folderu **OnlineBankingSystem**. Pošto su klase podeljene po fajlovima to nam omogućava da i samostalno testiramo klase. Ukupan broj testova je 8 i uslova 93. Na sledećim slikama (2-5) je prikazan broj testova za svaku klasu posebnu i svi testovi su uspešni.



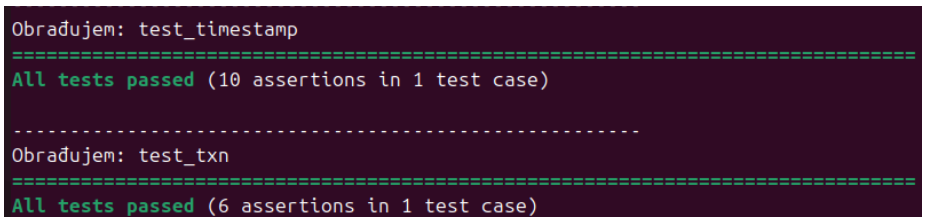
Slika 2: Uspešnost testova za controller.cpp



Slika 3: Uspešnost testova za credentials.cpp



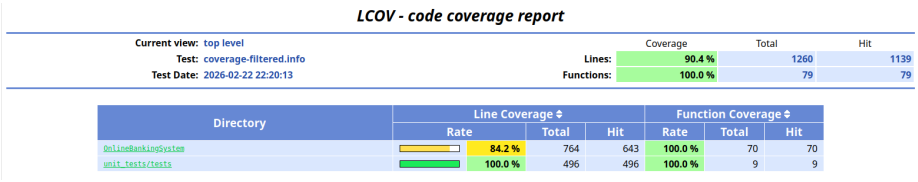
Slika 4: Uspešnost testova za customer.cpp, employee.cpp, person.cpp i rajdeep.cpp



Slika 5: Uspešnost testova za timestamp.cpp i txn.cpp

Pomoću lcov alata dobijen je izveštaj predstavljen na slici 6. Može se videti da je:

- pokrivenost linija koda: 1139 (90.4 %) od 1260 linija
- pokrivenost funkcija: 79 (100 %) od 79 funkcija



LCOV - code coverage report

		Coverage	Total	Hit
Current view:	top level			
Test:	coverage-filtered.info	Lines:	90.4 %	1260	1139
Test Date:	2026-02-22 22:20:13	Functions:	100.0 %	79	79

Directory	Line Coverage			Function Coverage		
	Rate	Total	Hit	Rate	Total	Hit
OnlineBankingSystem	84.2 %	764	643	100.0 %	70	70
unit_tests/tests	100.0 %	496	496	100.0 %	9	9

Slika 6: Izveštaj lcov alata

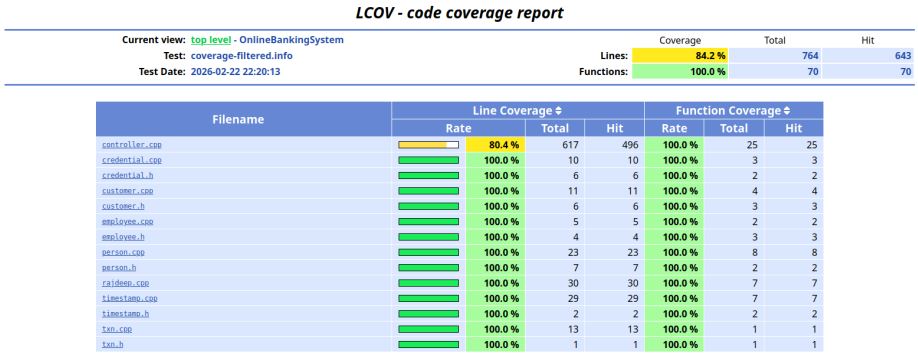
U OnlineBankingSystem se nalazi pokrivenost testova samog koda:

- pokrivenost linija koda: 643 (84.2 %) od 764 linija
- pokrivenost funkcija: 70 (100 %) od 70 funkcija

U unit_tests pokrivenost tesotva i pokrivenost je:

- pokrivenost linija koda: 496 (100 %) od 496 linija
- pokrivenost funkcija: 9 (100 %) od 9 funkcija

Izveštaj za OnlineBankingSystem folder:



LCOV - code coverage report

		Coverage	Total	Hit
Current view:	top level - OnlineBankingSystem			
Test:	coverage-filtered.info	Lines:	84.2 %	764	643
Test Date:	2026-02-22 22:20:13	Functions:	100.0 %	70	70

Filename	Line Coverage			Function Coverage		
	Rate	Total	Hit	Rate	Total	Hit
controller.cpp	80.4 %	617	496	100.0 %	25	25
credential.cpp	100.0 %	10	10	100.0 %	3	3
credential.h	100.0 %	6	6	100.0 %	2	2
customer.cpp	100.0 %	11	11	100.0 %	4	4
customer.h	100.0 %	6	6	100.0 %	3	3
employee.cpp	100.0 %	5	5	100.0 %	2	2
employee.h	100.0 %	4	4	100.0 %	3	3
person.cpp	100.0 %	23	23	100.0 %	8	8
person.h	100.0 %	7	7	100.0 %	2	2
rajdeep.cpp	100.0 %	30	30	100.0 %	7	7
timestamp.cpp	100.0 %	29	29	100.0 %	7	7
timestamp.h	100.0 %	2	2	100.0 %	2	2
txn.cpp	100.0 %	13	13	100.0 %	1	1
txn.h	100.0 %	1	1	100.0 %	1	1

Slika 7: Izveštaj lcov alata za OnlineBankingSystem

Može se videti da je dosta pokriveno testiranjem, ali preostaju delovi koda u controller.cpp koje se treba bolje razmotriti, jer je lcov u izveštaju pokazao žutom bojom rezultat pokrivenosti, što znači da je pokrivenost između 75 % i 90 %, što ovde i jeste slučaj 80.4 %. To znači da je delimična pokrivenost, tj. upozorenje da je pokrivenost dobra, ali ne i potpuna. Pokušano je dodavanje još nekih testova, kako bi se pokrio preostali deo koda, ali dolazi do nekih drugih grešaka poput Segmentation fault-a. U buduće treba razmotriti da se kod promeni, pa tek onda definisati nove testove.

Posebnu pažnju treba posvetiti funkcijama koje još uvek nisu implementirane (**TODO**), pažljivo ih treba implementirati tako da ne narušavaju već

postojeći sistem. Takođe, treba obratiti pažnju na neinicijalizovane default-ne konstruktore.

Umesto:

```
Employee():Person(),salary(0){}
```

Treba npr. za ID ubaciti:

```
Employee():Person():ID(-1),salary(0){}
```

Kao što je prethodno rečeno dolazi i do **Segemntation fault** greške kada se definišu testovi gde program pokušava da pristupi praznim vektorima, kao i neispravno rukovanje iteratorima.

2.2 Zaključak

Testiranje je od ključnog značaja za osiguranje ispravnosti sistema, a jedinični testovi su omogućili detaljnu proveru funkcionalnosti ključnih klasa u projektu. Korišćenjem biblioteke **Catch2** i alata **lcov** postignuta je visoka pokrivenost testovima, sa 80.4% pokrivenosti linija koda i 100% pokrivenosti funkcija. Iako je većina sistema temeljno testirana, uočeni su problemi sa delimičnom pokrivenošću u određenim delovima koda, poput `controller.cpp`, koji treba da bude predmet daljeg istraživanja. Potrebno je rešiti greške poput `Segmentation fault` koje se javljaju pri radu sa praznim vektorima i iteratorima.

3 Valgrind

Valgrind je alat otvorenog koda namenjen za dinamičku analizu koda, prvenstveno pisanih u jezicima C i C++, sa ciljem otkrivanja grešaka u radu sa memorijom i nitima. Najčešće se koristi njegov alat Memcheck koji omogućava detekciju curenja memorije, detekciju pristupa neinicijalizovanoj memoriji i detekciju neispravnog oslobađanja memorije. Alat funkcioniše izvršavanjem programa u kontrolisanom okruženju, poput virutuelnih mašina, čime se omogućava detaljno praćenje memorijskih operacija, bez potrebe za modifikacijom izvornog koda. Iako usporava izvršavanje programa, Valgrind pruža precizne informacije koje značajno doprinose poboljšanju kvaliteta i pouzdanosti softvera.

Glavni Valgrind alati:

- **Memcheck** – detektuje greške u memoriji, neinicijalizovane vrednosti, duplo oslobađanje memorije i curenje memorije.
- **Massif** – prati i optimizuje korišćenje dinamičke memorije.
- **Cachegrind** – analizira performanse keš memorije i skokova u programu.
- **Callgrind** – generiše graf poziva funkcija i predviđa skokove, koristi se za optimizaciju.
- **Helgrind / DRD** – otkriva greške u višenitnim programima, poput utrka podataka i deadlock-a.

3.1 Valgrind Memcheck

Memcheck je najčešće korišćen alat u okviru Valgrinda, koji analizira program na nivou mašinskog koda i može raditi sa bilo kojim programskim jezikom. Detektuje korišćenje nedefinisanih ili neinicijalizovanih vrednosti, neispravan pristup memoriji, duplo oslobađanje memorije, curenje memorije i greške pri kopiranju podataka. Takođe prati parametre sistemskih poziva i beleži sve alocirane blokove memorije. Opcije kao što su `-track-origins=yes` i `-leak-check=full` omogućavaju precizno lociranje uzroka grešaka i detaljan izveštaj o curenju memorije.

3.1.1 Analiza rezultata

Alat se pokreće bash skriptom run_valgrind_memcheck.sh, koja se nalazi u direktorijumu valgrind_memcheck, a izveštaj se nalazi u istom tom folderu valgrind_report.txt. Prilikom pokretanja valgrind alata detektovano je nekoliko grešaka vezanih za neinicijalizovanu memoriju i rad sa baferima u funkciji `Controller::addEmployee()`.

U nastavku je prikazan izveštaj i objašnjenje linija:

- `==23172== Memcheck, a memory error detector`
  Tumačenje: Pokrenut je Memcheck alat, koji detektuje greške u memoriji, kao što su neinicijalizovane vrednosti, curenje memorije i duplo oslobađanje memorije.

- `Syscall param write(buf) points to uninitialised byte(s)`
  Tumačenje: Sistemskom pozivu `write` prosleđen je bafer koji sadrži neinicijalizovane bajtove, što može dovesti do nepravilnog upisivanja podataka u fajl ili uređaj.

- `Address 0x4e29437 is 23 bytes inside a block of size 8,192 alloc'd`
  Tumačenje: Greška je nastala unutar dinamički alociranog bloka memorije veličine 8192 bajta. Pokazivač upućuje na lokaciju 23 bajta unutar tog bloka.

- `Uninitialised value was created by a stack allocation at Controller::addEmployee() (controller.cpp:545)`
  Tumačenje: Neinicijalizovana promenljiva je kreirana na steku unutar funkcije `addEmployee()` na liniji 545. To znači da pre upotrebe promenljiva nije dobila početnu vrednost.

- `HEAP SUMMARY: in use at exit: 0 bytes in 0 blocks`
  Tumačenje: Nakon završetka programa, svi dinamički alocirani blokovi memorije su oslobođeni. Nema curenja memorije.

- `total heap usage: 65 allocs, 65 frees, 244,553 bytes allocated`
  Tumačenje: Tokom izvršavanja programa alocirano je 65 blokova ukupne veličine 244,553 bajtova, koji su svi pravilno oslobođeni.

- `ERROR SUMMARY: 5 errors from 1 contexts`
  Tumačenje: Memcheck je prijavio 5 grešaka u memoriji, sve povezane sa neinicijalizovanim promenljivama ili baferima.

3.1.2 Zaključak

Glavni problem u programu je korišćenje neinicijalizovanih promenljivih u funkciji `Controller::addEmployee()`, što može dovesti do nepravilnog upisivanja podataka ili neočekivanog ponašanja programa. Dinamička memorija je pravilno oslobođena, tako da curenje memorije nije prisutno. Preporučuje se inicijalizacija svih promenljivih pre upotrebe i dodatna provera bafera koji se prosleđuju sistemskim pozivima.

4 Cppcheck

Cppcheck je alat za statčku analizu koda koji je specijalizovan za programske jezike C i C++. On analizira sam izvorni kod bez potrebe za njegovim pokretanjem, fokusirajući se na pronalaženje bagova koje kompajleri često propuštaju, kao što su nedefinisano ponašanje, curenje resursa i logičke greške, kao i greške u sintaksi. Može se koristiti direktno iz komandne linije. Glavna prednost je što je brz, kao i što ne zahteva posebno okruženje za izvršavanje. Posebno je koristan u ranoj fazi razvoja, kada testovi nisu kompletni.

4.1 Analiza rezultata

Alat `Cppcheck` pokrenut je korišćenjem bash skripte `run_cppcheck.sh` koja se nalazi u direktorijumu `cppcheck`. Detaljan izlaz iz alata nalazi se u istom direktorijumu pod nazivom cppcheck_output.xml. Alat je detektovao različite greške i upozorenja koja možemo klasifikovati u sledeće kategorije:

4.1.1 Greške vezane za nedostajuća standardna zaglavlja (missingIncludeSystem)

- **Opis:** Cppcheck ne može da pronađe standardna zaglavlja kao što su `<iostream>`, `<vector>`, `<fstream>`, `<cstring>` itd. Ova upozorenja su informativna i ne utiču na samu analizu koda.

4.1.2 Greške vezane za neinicializovane članove klase (uninitMemberVar)

- **Opis:** Članovi klase `Person`, uključujući `ID`, `name`, `address`, `phone` i `pass`, nisu inicijalizovani u konstruktoru. To može dovesti do nepredvidivog ponašanja programa.

4.1.3 Greške vezane za preklapanje funkcija bez `override` (missingOverride)

- **Opis:** Funkcije `showDetails` u izvedenim klasama preklapaju virtuelne funkcije baze, ali nisu označene sa `override`, što je preporučena praksa u C++ radi jasnoće i sigurnosti.

4.1.4 Greške vezane za C-style kastovanje (cstyleCast)

- **Opis:** Korišćeno je C-style kastovanje (`(Type*)pointer`) umesto bezbednijih C++ kastova (`static_cast`, `dynamic_cast` itd.).

4.1.5 Greške vezane za zasenjene promenljive (shadowVariable)

- **Opis:** Lokalne promenljive `cust` i `emp` zasenjuju [1] spoljne promenljive istog imena, što može otežati čitanje koda.

4.1.6 Greške vezane za nepročitane promenljive (unreadVariable)

- **Opis:** Promenljive `rajdeep` i `SESSION` su dodeljene, ali nikada iskorišćene u programu.

4.1.7 Greške vezane za neiskorišćene funkcije (unusedFunction)

- **Opis:** Funkcije `drawLineTrans`, `transText` i `printTxn` nisu pozvane nigde u programu, što može ukazivati na nepotreban kod.

4.1.8 Greške vezane za potencijalnu optimizaciju članova klase (functionStatic)

- **Opis:** Funkcije iz `Rajdeep`: `delay`, `drawLine` i `transText` ne koriste članove klase i mogu biti statičke radi bolje performanse.

4.1.9 Greške vezane za efikasnost prosleđivanja parametara (passedByValue)

- **Opis:** Parametar `str` je prosleđen po vrednosti, što je manje efikasno; preporučuje se prosleđivanje po `const reference`.

4.1.10 Greške vezane za efikasnost inkrementa/dekrementa (postfixOperator)

- **Opis:** Za ne-primitive tipove preporučuje se prefiks (`++i`) umesto postfiks (`i++`) radi bolje efikasnosti.

[1] Zasena promenljiva (engl. *shadow variable*) se javlja kada lokalna promenljiva unutar funkcije ili bloka ima isto ime kao promenljiva iz spoljnog opsega. U tom slučaju lokalna promenljiva "zasenjuje" spoljašnju, što može dovesti do logičkih grešaka ili zabune kod programera.

4.1.11 Greške vezane za korišćenje STL algoritama (useStlAlgorithm)

- **Opis:** Alat predlaže korišćenje `std::any_of` ili `std::find_if` umesto ručno pisanih petlji.

4.1.12 Informativna upozorenja – neusklađena supresija (unmatchedSuppression)

- **Opis:** Navedena supresija [2] su često prijavljeni problemi pri statičkoj analizi koda. za `missingInclude` nije pronađena ili nije aktivna.

4.2 Zaključak

Alat `Cppcheck` nije detektovao značajne greške koje bi mogle da utiču na stabilnost i ispravnost programa. Upozorenja i informacije koje alat pruža pomažu u unapređenju kvaliteta koda i mogu se koristiti za dalju optimizaciju i poboljšanje stila programiranja.

5 Clang-tidy

Clang-tidy je statički analizator koda za jezike C i C++ koji se koristi za automatsko pronalaženje grešaka, problema sa stilom i potencijalnih bagova pre nego što se kompajlira ili pokrene program.

Osnovne funkcionalnosti alata su:

1. **Statička analiza koda** – detektuje potencijalne greške, uključujući neinicijalizovane promenljive, logičke greške i neefikasne izraze.
2. **Provera stila koda** – upozorava na odstupanja od preporučenih standarda.
3. **Automatski refaktoring** – može predložiti i primeniti promene u kodu.
4. **Integracija sa build sistemom** – radi zajedno sa CMake-om ili drugim build alatima.

5.1 Analiza rezultata

Alat `clang-tidy` pokrenut je korišćenjem bash skripte `run_clang.sh` koja se nalazi u direktorijumu `clang`. Detaljan izlaz iz alata nalazi se u istom direktorijumu pod nazivom clang-tidy_output.txt.

Analiza je imala za cilj identifikaciju stilskih nedoslednosti, potencijalnih grešaka i odstupanja od preporučenih pravila C++ standarda. Ukupno je generisano više od 25000 upozorenja, od kojih većina ukazuje na unapređenje čitljivosti, bezbednosti i održivosti koda.

[2] Neusklađena supresija (engl. *unmatched suppression*) se javlja kada je u Cppcheck-u navedena supresija greške koja ne odgovara nijednom stvarnom upozorenju u kodu. To znači da je supresija prisutna, ali Cppcheck nije pronašao grešku koju bi trebalo ignorisati.

5.1.1 Upotreba tzv. *magic numbers* (readability-magic-numbers)

U fajlovima identifikovana je učestala upotreba numeričkih konstanti direktno u kodu (`50, 100, 10000, 20, 30, 10, 48`). Ovo smanjuje čitljivost i otežava održavanje. Preporučuje se zamena sa imenovanim konstantama ili `constexpr` promenljivama, npr.: U kodu:

```
rajdeep.drawLine(100);
```

Bolje:

```
constexpr int defaultLineLength = 100;
rajdeep.drawLine(defaultLineLength);
```

5.1.2 Performanse i efikasnost ispisa (performance-avoid-endl)

U fajlovima `employee.cpp`, `rajdeep.cpp` i `txn.cpp` uočena je neefikasna upotreba `std::endl`. Pošto `std::endl` pored novog reda vrši i pražnjenje bafera, preporučuje se korišćenje `\n` gde nije neophodno pražnjenje.

5.1.3 Problemi sa imenovanjem i stilom identifikatora i stavljanje više promenljivih u jednu liniju (readability-identifier-length, readability-isolate-declaration)

U više fajlova (posebno `rajdeep.cpp` i `timestamp.cpp`) uočena su prekratka imena promenljivih (`i, j, ss, ch, ms`), što otežava razumevanje i održavanje koda. Preporučuje se korišćenje opisnijih naziva.

Kada se nalazi više promenljivih u jednoj liniji (`std::string id,pass`), smanjuje čitljivost, bolje je svaku promenljivu staviti u posebnu liniju.

5.1.4 Nepoštovanje pravila o formatiranju i čitljivosti koda (readability-braces-around-statements)

U više fajlova uočen je nedostatak vitičastih zagrada kod `if/else` blokova (*readability-braces-around-statements*) i korišćenje `else` nakon `return` (*readability-else-after-return*). Na primer, u `employee.cpp`: Bez promene:

```
if(c1.getBalance() < c2.getBalance())
    return true;
else
    return false;
```

Preporučuje se zamena sa:

```
if(c1.getBalance() < c2.getBalance()){
        return true;
   } else {
        return false;
    }
```

5.1.5 Problemi sa modernim C++ stilom

Više funkcija u fajlovima `employee.cpp`, `timestamp.cpp` i `rajdeep.cpp` nije definisano u skladu sa modernim C++ stilom. Preporučuje se korišćenje *trailing return type* sintakse (**modernize-use-trailing-return-type** ili **modernize-use-auto**), npr.:

```
auto TimeStamp::toString() const -> std::string
```

Umesto klasičnih potpisa funkcija:

```
std::string TimeStamp::toString() const
```

Ovo poboljšava čitljivost i konzistentnost koda.

Još jedan od preporuke je da se umesto 0/1 koristi false/true **(modernize-use-bool-literals)**.

5.1.6 Problemi sa konverzijama i tipovima podataka

Alat je detektovao:

- sužavajuće konverzije tipova (**bugprone-narrowing-conversions**) u `timestamp.cpp` kod funkcije `timeWeight()`,
- C-stil kastovanja u nekim fajlovima, što smanjuje čitljivost i može dovesti do nepredvidivih grešaka **(bugprone-narrowing-conversions)**.

Rešenje je korišćenje specifičnih operatora, npr.:

```
double b = static_cast<double>(a);
```

5.1.7 Problemi sa grananjem i logikom poređenja

U fajlovima `timestamp.cpp` i `employee.cpp` uočena su:

- redundantna vraćanja `true/false` u poređenjima (**readability-simplify-boolean-expr**),
- nepotpuno pokrivanje svih grana u `switch` naredbama (**bugprone-switch-missing-default-case**),
- višestruki uzastopni parametri istog tipa (**bugprone-easily-swappable-parameters**), što može dovesti do grešaka pri pozivanju funkcija.

5.1.8 Problemi sa inicijalizacijom promenljivih

(clang-analyzer-optin.cplusplus.UninitializedObject)

U fajlovima `rajdeep.cpp` i `employee.cpp` postoje promenljive koje nisu inicijalizovane pri deklaraciji. Svaka promenljiva treba biti inicijalizovana radi izbegavanja nepredvidivog ponašanja.

5.1.9 Problem sa korišćenjem funkcije strcpy

(clang-analyzer-security.insecureAPI.strcpy)
U fajlovima person.cpp i credential.cpp se koristi strcpy, bez prethodne provere veličine, bolje je koristiti **strncpy** ili **strlcpy**.

5.1.10 Ostala upozorenja i preporuke

- Upotreba C-stil nizova (`char suits[4]`) preporučuje se zameniti sa `std::array` ili `std::vector` **(modernize-avoid-c-arrays)**.
- Klasične for-petlje sa brojačem (`for(int i=0;i<4;i++)`) mogu se zameniti *range-based for loops* radi bolje čitljivosti **(modernize-loop-convert)**.
- Provera praznog kontejner sa `size()==0` zamenjuje se metodom `empty()`, koja bolje izražava nameru programera.
- Više funkcija može biti deklarisano kao `const`, jer ne menjaju stanje objekta (`getSalary()`, `drawLine`, itd.).
- else posle return je nepotreban **(readability-else-after-return)**
- kada funkcija šalje veliki objekat bolje da šalje po referenci, a ne po vrednosti
  (**performance-unnecessary-value-param**)

5.2 Zaključak

clang-tidy analiza je identifikovala uglavnom stilske nedoslednosti, rizične konstrukcije i mogućnosti za unapređenje modernog C++ stila. Primena svih preporuka poboljšava čitljivost, održivost i performanse projekta, kao i usklađenost sa savremenim C++ standardima.

6 Flawfinder

Flawfinder je open-source alat za statičku analizu C i C++ koda, razvijen sa ciljem da pomogne programerima u otkrivanju potencijalnih sigurnosnih slabosti. Alat pretražuje izvorni kod i identifikuje upotrebu funkcija koje su poznate kao nesigurne, poput `strcpy`, `gets` ili `system`. Svaka pronađena funkcija se rangira po nivou rizika od 0 do 5, pri čemu viši broj označava ozbiljniju ranjivost. Flawfinder je kompatibilan sa standardom CWE (Common Weakness Enumeration), što omogućava da se rezultati povežu sa poznatim kategorijama sigurnosnih problema.

6.1 Analiza rezultata

Alat `flawfinder` pokrenut je korišćenjem bash skripte `run_flawfinder.sh` koja se nalazi u direktorijumu `flawfinder`. Detaljan izlaz iz alata nalazi se u istom direktorijumu pod nazivom `flawfinder_report.txt`. Analizirani kod sadrži ukupno 41 potencijalni problem, raspoređenih po različitim nivoima rizika. Najviši nivo rizika uočava se kod funkcija `system()` i `strcpy()`, koje su označene sa nivoom 4. Funkcija `system()` (CWE-78) predstavlja ozbiljan problem jer omogućava izvršavanje spoljnog programa kroz shell, što može dovesti do napada komandnom injekcijom. Funkcija `strcpy()` (CWE-120) ne proverava granice bafera, pa može izazvati buffer overflow.

Pored toga, u kodu se pojavljuju statički definisani nizovi tipa `char[]` (nivo rizika 2), koji mogu biti nedovoljno veliki i dovesti do prepisivanja memorije (CWE-119/120). Preporuka je da se koristi `std::string` u C++ ili da se uvede stroga kontrola granica prilikom upisa u niz. Takođe, funkcija `read()` (nivo rizika 1) se koristi bez adekvatne provere granica bafera, što može izazvati probleme u slučaju nepažljivog rukovanja ulazom, ukoliko se upiše više podataka nego što je predviđeno, pogotovo ako se koristi u petljama ili rekurzivnim pozivima. Najbolje je da se uvek prosledi veličina bafera i da se proveri povratna vrednost funkcije.

6.2 Zaključak

Rezultati pokazuju da kod sadrži značajan broj potencijalnih ranjivosti, od kojih su najkritičnije one vezane za `system()` i `strcpy()`. Iako se ranjivosti nivoa 1 i 2 ne smatraju kritičnim u poređenju sa prethodno navedenim funkcijama, učestalost u kodu ukazuje na nedostak pažnje pri radu sa memorijom i ulazno-izlaznim funkcijama. Kombinacija više slabosti nižeg rizika može povećati ukupnu površinu napada i dovesti do ozbiljnih posledica. Ova analiza je posebno važna za sigurnosne revizije i ukazuju na potrebu zamene nesigurnih funkcija bezbednijim alternativama.

7 Iwyu

`Include What You Use` (IWYU) je alat za statičku analizu C++ koda koji pomaže u optimizaciji direktiva `#include`. Cilj IWYU-a je da pomogne programerima da identifikuju i uklone nepotrebne zavisnosti u kodu, čime se smanjuje veličina i složenost projekta, a time može da poboljša brzinu kompilacije. Alat analizira izvorni kod i prepoznaje sve neiskorišćene ili redundantne biblioteke koje su uključene u zaglavlja ili izvornu datoteku. Takođe, IWYU preporučuje dodavanje nedostajućih zavisnosti koje se koriste u kodu, čime se obezbeđuje tačnost i funkcionalnost programa.

7.1 Analiza rezultata

IWYU analiza pokrenuta je za ceo projekat `OnlineBankingSystem` koji sadrži brojne C++ izvore i zaglavlja. Detaljan izlaz iz analize može se pronaći u direktorijumu `iwyu_report.txt`, a analiza se fokusira na ispravno upravljanje zavisnostima između različitih modula projekta.

Uočen je veliki broj predloga za optimizaciju, među kojima se ističu sledeće ključne tačke:

- Uklanjanje nepotrebnih `#include` direktiva: npr. u datoteci `controller.h` detektovano je da nije potrebna zavisnost ka `iostream` i `customer.h`, što može smanjiti veličinu zaglavlja i vreme kompilacije.
- Dodavanje nedostajućih zavisnosti: npr. u `controller.cpp` bila je potrebna dodatna zavisnost za `timestamp.h`, kao i biblioteke `string.h` i `string`, koje su potrebne za ispravno funkcionisanje koda.
- Preporuke za uvođenje forward deklaracija: U nekim slučajevima, kao što je `controller.h`, umesto direktnih `#include` direktiva, preporučuje se korišćenje forward deklaracija kao što je `class Customer;` kako bi se smanjila zavisnost između modula.
- Optimizacija uključivanja u `cpp` datotekama: Na primer u `customer.cpp` i `employee.cpp`, preporučeno je uklanjanje redundatnih zavisnosti koje nisu bile korišćene, kao što je `#include <iostream>`.

7.2 Zaključak

Rezultati IWYU analize ukazuju na brojne mogućnosti za poboljšanje upravljanja zavisnostima u projektu. Iako mnoge preporuke nisu kritične, optimizacija `#include` direktiva može značajno uticati na performanse kompilacije i održavanje koda. Najveća prednost IWYU-a leži u smanjenju kompleksnosti koda kroz eliminaciju nepotrebnih zavisnosti i unapređenje pregleda zavisnosti između modula.

8 Zaključak analize

Sprovedena je statička i dinamička analiza kojom je prikazano da Online Banking System ima dobar nivo funkcionalne ispravnosti, visoku pokrivenost testovima i pravilno upravljanje memorijom. Iako nisu uočeni kritični problemi, identifikovane su određene slabosti u pogledu bezbednosti, inicijalizacije promenljivih i stila koda koje je potrebno unaprediti. Kombinacija korišćenih alata omogućila je sveobuhvatnu procenu kvaliteta softvera i ukazala na pravce daljeg unapređenja, poput refaktorisanja rizičnih delova koda i dodatnog proširenja testiranja.

Literatura

[1] Milena Vujošević Janičić. *Verifikacija softvera*. Matematički fakultet, Beograd.

[2] *Materijali iz kursa Verifikacija softvera sa vežbi*

[3] `https://dwheeler.com/flawfinder/`

[4] `https://github.com/include-what-you-use/include-what-you-use`