БЕОГРАДСКА АКАДЕМИЈА ПОСЛОВНИХ И УМЕТНИЧКИХ СТРУКОВНИХ СТУДИЈА

ПРОЈЕКТНИ РАД

ОСНОВНЕ СТРУКОВНЕ СТУДИЈЕ

Ментор: Сузана Марковић

Кандидат: Миљана Милуновић

Београд

2024

БАПУСС

Београдска академија пословних и уметничких струковних студија

ОДСЕК ПОСЛОВНИХ И ИНФОРМАТИЧКИХ СТУДИЈА

Студијски програм: Информациони системи и технологије

ПРОЈЕКТНИ РАД

ОСНОВНЕ СТРУКОВНЕ СТУДИЈЕ

Јединично тестирање софтвера у C#

Комисија:

1. _____ _____, председник
   (потпис)
2. Сузана Марковић _____, ментор
   (потпис)
3. _____ _____, члан
   (потпис)

Датум одбране рада: _____

Оцена: _____

Студент: Миљана Милуновић

Број индекса: 1и1/0039/19

ПРЕДМЕТ: И З Ј А В А

Овај рад сам оригинално написао/ла, те тврдим на основу њега раније нико није имао корист у некој другој институцији.

Захваљујем се на свим корисним сугестијама коментору _____ (име и презиме коментора) који/а ми је обезбедио/ла подршку у процесу стручне праксе и ментору праксе Сузани Марковић (име и презиме ментора праксе).

Потпис студента:

_____

Садржај

1. Увод

Софтверски развој је комплексан процес који захтева пажљиво планирање, имплементацију и тестирање како би се осигурао квалитет коначног производа. У дигиталном добу, софтвер је постао неизоставан део свакодневног живота, од мобилних апликација и интернет платформи до пословних система. Због тога је кључно осигурати да софтверски производи испуњавају високе стандарде перформанси, сигурности и функционалности.

Један од најважнијих аспеката осигурања квалитета софтвера је процес тестирања. Тестирање софтвера обухвата различите типове тестирања, сваки са својом специфичном улогом у развоју софтвера. У овом пројектном раду фокусираћемо се на јединично тестирање као суштински алат за осигурање квалитета. Јединично тестирање је пракса која омогућава тестирање појединачних компоненти софтвера, познатих као "јединице" (енг. units), како би се проверило да ли раде исправно и у складу са очекивањима. Овај приступ омогућава рано откривање и исправљање грешака у коду, олакшава одржавање и побољшава стабилност и поузданост софтверског система.

Циљ овог пројектног рада је истражити важност, принципе, праксе и изазове јединичног тестирања у савременом софтверском развоју. Рад ће такође пружити детаљан преглед различитих типова тестирања, као што су интеграционо, функционално, системско и прихватно тестирање, и објаснити како се они надопуњују у процесу развоја софтвера.

У наредним поглављима, истражићемо принципе и праксе јединичног тестирања, укључујући кључна упутства за писање и одржавање тестова. Затим ћемо се фокусирати на имплементацију јединичних тестова у C# програмском језику, где ће бити представљен NUnit радни оквир за тестирање, са посебним осврtом на његове предности и различите типове Assert-ова. Практична примена јединичног тестирања биће демонстрирана на два примера: једноставнијој апликацији калкулатора и комплекснијој банкарској апликацији, што ће пружити увид у различите аспекте и изазове јединичног тестирања.

Кроз овај рад, надам се да ћу допринети бољем разумевању улоге и значаја јединичног тестирања у софтверском развоју и подстаћи ширу примену ове праксе у индустрији. Одабиром теме јединичног тестирања, желим да нагласим колико је важно тестирање за побољшање квалитета софтвера, посебно у савременом развоју софтвера где је поузданост и стабилност система од кључне важности.

2. Основе тестирања

Прво и најважније начело доброг софтвера је висок квалитет. Висок квалитет софтвера се постиже следећим карактеристикама [1]:

- Мора бити употребљив
- Робусан
- Компатибилан
- Ефикасан
- Поуздан
- Једноставан
- Проширив
- Портабилан и
- Поново употребљив

Тестирање софтвера је важан процес у развоју који обезбеђује да се софтвер понаша у складу са очекивањима и захтевима. Тестирање има за циљ да открије грешке, недостатке или пропусте у систему пре него што софтвер буде испоручен корисницима. Постоје различити типови тестирања, од којих сваки има специфичну сврху и примењује се у различитим фазама развојног процеса. У овом поглављу, фокусираћемо се на јединично тестирање, али ћемо такође укратко обрадити и друге врсте тестирања као што су интеграционо, функционално, системско и тестирање прихватљивости.

2.1. Типови тестирања

Разликујемо следеће типове тестирања:

- Јединично тестирање (енг. Unit Testing): Тестирање појединачних компоненти или модула софтвера. Ово тестирање се изводи како би се осигурало да свака компонента функционише исправно и понаша се у складу са очекивањима.
- Интеграционо тестирање (енг. Integration Testing): Тестирање комбинације компоненти како би се проверило да ли раде заједно у складу са очекивањима. Фокусира се на интеракцију између компоненти и модула.
- Функционално тестирање (енг. Functional Testing): Проверава да ли софтвер испуњава све функционалне захтеве. Фокус је на тестирању функција и особина софтвера према спецификацијама.
- Системско тестирање (енг. System Testing): Комплетно тестирање целокупног система како би се осигурало да ради у складу са свим захтевима и да се очекивано понаша у реалним сценаријима.
- Тестирање прихватљивости (енг. Acceptance Testing): Тестирање које потврђује да софтвер испуњава захтеве и потребе корисника. Ово тестирање се обично спроводи на крају развојног процеса.

Јединично тестирање би могли да дефинишемо као један целокупни процес тестирања појединачних компоненти софтвера, те појединачне компоненте су познатије као “Јунити” (енг. Units). Ја ћу се у даљем раду служити искључиво термином јединично

[1] Др Сузана Марковић, *Основе програмирања*, Академија пословних струковних студија, Београд, 2019.

тестирање, али и Јунит има потпуно исто значење. Сврха јединичног тестирања је осигурати да свака компонента софтвера ради исправно и да се очекивано понаша, као и идентификовати и исправити грешке у коду што је раније могуће.
Пошто се тестови јединица обично дешавају током фазе развоја, они омогућавају тимовима да идентификују и исправе проблеме пре објављивања софтвера. Јединични тестови упозоравају програмере на потенцијалне грешке или недостатке који би могли изазвати проблеме у будућности и побољшати укупан квалитет и перформансе [2].
Важно је издвојити неке од најважнијих предности коришћења овог принципа тестирања:

- Рано откривање грешака: Тестирање компоненти одвојено омогућава рано откривање грешака, што олакшава исправке и смањује трошкове.
- Лакше одржавање кода: Јединични тестови служе као документација понашања компоненти, олакшавајући одржавање и рефакторисање кода.
- Побољшана стабилност: Редовно извођење јединичних тестова осигурава стабилност софтвера и смањује ризик од неочекиваних проблема у продукцији.

Интеграционо тестирање је тестирање где се индивидуалне целине програмског кода комбинују и тестирају као група. Док јединично тестирање проверава исправност појединачних компоненти, интеграционо тестирање има сврху да открије потенцијалне проблеме и грешке у интеграцији између интегрисаних целина. [3]
Циљ интеграционог тестирања је да идентификује проблеме у комуникацији, синхронизацији и интеракцији између компоненти. На пример, у апликацији која има базу података, интеграционо тестирање би проверило да ли се подаци правилно чувају и враћају из базе података када се позове одговарајућа функција. Још неки примери су: тестирање интерфејса између модула или функција и тестирање комуникације између клијента и сервера у веб апликацији.

Функционално тестирање је врста тестирања која проверава да ли софтверски систем испуњава све функционалне захтеве и спецификације. Овај тип тестирања се фокусира на провери да ли су све функције софтвера имплементиране и да раде у складу са дефинисаним захтевима. Циљ функционалног тестирања је да осигура да софтвер испуњава очекивања корисника у погледу функционалности. Функционално тестирање обухвата разне тестове који проверавају понашање софтвера у различитим условима и сценаријима. Примери: Провера да ли корисник може успешно да се пријави на систем са исправним креденцијалима, тестирање процеса наручивања у е-трговинској апликацији, или провера исправног израчунавања резултата у калкулатору.

Системско тестирање је свеобухватно тестирање целокупног софтверског система како би се осигурало да све компоненте, модуле и интеграције функционишу заједно као један кохезиван систем. За разлику од јединичног и интеграционог тестирања, које су фокусиране на мање делове апликације, системско тестирање има за циљ да провери да ли је понашање система у складу са спецификацијом задатом од стране клијента. Овде се захтева и потпун приступ бази и хардверским деловима система. Системско тестирање

[2] Заптест вебсајт, *Шта је јединично тестирање*: https://www.zaptest.com/sr/%D1%88%D1%82%D0%B0-%D1%98%D0%B5-%D1%98%D0%B5%D0%B4%D0%B8%D0%BD%D0%B8%D1%87%D0%BD%D0%BE-%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%B0%D1%9A%D0%B5-%D0%B4%D1%83%D0%B1%D0%BE%D0%BA%D0%BE-%D0%B7%D0%B0%D1%80
[3] Дражен Драшковић, *Тестирање Софтвера - Интеграционо Тестирање*, Електротехнички факултет, Универзитет у Београду, Београд

може да укључује и функционалне и нефункционалне аспекте система [4]. Циљ системског тестирања је да се открију потенцијални проблеми који могу настати када се сви делови система комбинују и раде заједно. Ово укључује тестирање перформанси, сигурности, компатибилности и других аспеката софтвера који се могу појавити у реалном окружењу. Неки примери су тестирање одговора система на различите врсте оптерећења, провера да ли су сви безбедносни захтеви испуњени, тестирање интеракције са различитим оперативним системима и платформама.

Тестирање прихватљивости, познато и као User Acceptance Testing (UAT), је финална фаза тестирања у којој се софтвер тестира у реалним сценаријима да би се потврдило да испуњава све захтеве и очекивања корисника. Ово тестирање се спроводи након што је систем прошао све остале фазе тестирања, укључујући јединично, интеграционо, функционално и системско тестирање. Ову врсту тестирања изводе и процењују корисници, а развојни тим им пружа помоћ око техничких питања, уколико за тим има потребе. Тестирање прихватљивости обично спада у технике валидације софтвера [5].
Примери тестирања прихватљивости су: провера да ли су све функције софтвера интуитивне и доступне корисницима, тестирање процеса куповине у е-трговинској апликацији у реалном сценарију, или провера да ли сви излазни подаци испуњавају захтеване формате и стандарде.

3. Принципи и праксе јединичног тестирања

Принципи и праксе јединичног тестирања обухватају основне концепте и технике које су битне за разумевање и примену тестова у развоју софтвера. Јединично тестирање је основа за обезбеђивање квалитета софтвера, и његова исправна примена може значајно побољшати стабилност и поузданост система.

Један од кључних принципа јединичног тестирања је покривеност кода. Циљ је да јединични тестови покрију што више делова кода, укључујући различите услове и граничне случајеве. Висока покривеност кода осигурава да су сви критични делови софтвера адекватно тестирани и да немају скривених грешака.

Тестови треба да буду изоловани и независни један од другог, што значи да измена у једном тесту не сме да утиче на извршавање осталих тестова. Овај принцип осигурава да сваки тест даје јасну и недвосмислену информацију о исправности или неисправности дела кода који тестира. Брзина и ефикасност су такође кључни аспекти јединичног тестирања; тестови треба да се извршавају брзо како би обезбедили брзу повратну информацију програмеру.

Тест случајеви се обично односе на тестирање основних јединица кода. Јединица кода може бити функција, класа, модул или било која друга логичка јединица која представља понашање система. Ове јединице треба тестирати засебно, позивањем преко јавног

[4] Милена Вујошевић Јаничић, *Верификација софтвера*, Математички Факултет, Универзитет у Београду, Београд 2023
[5] Милена Вујошевић Јаничић, *Верификација софтвера*, Математички Факултет, Универзитет у Београду, Београд 2023

интерфејса, што омогућава да се свака јединица тестира у изолацији од осталих делова система [6].

Тест јединице кода може да изврши следеће радње:

1. Тестирање једне логичке функције.
2. Извршавање без одређеног редоследа извршења.
3. Пружање истих резултата за исти унос при сваком покретању.

Ови принципи чине основу за спровођење јединичног тестирања у различитим процесима развоја софтвера. Јединично тестирање је укључено у следеће процесе развоја:

1. Test Driven Development или развој вођен тестовима је техника развоја софтвера која подразумева често тестирање написаног кода. Тест се пише пре кода, а након тога се пише изворни код који треба да задовољи тест. Рефакторисањем се тај код даље пречишћава и поједностављује, али основно је да тест који се једном верификовао мора да се изнова верификује, при свим следећим изменама. [7].

2. Аутоматизовано тестирање је метода у тестирању софтвера која користи посебне софтверске алате за контролу извршења тестова и затим упоређује стварне резултате са очекиваним резултатима. Све се то ради аутоматски уз мало или никакву интервенцију од стране тест инжењера [8].

3. Интеграција у CI/CD. Важно је да тестови буду интегрисани у развојни процес и да се извршавају редовно, пожељно после сваке промене у коду. Аутоматизација извршавања унит тестова помоћу алата за континуирану интеграцију (CI) олакшава овај процес.

Ови принципи и праксе јединичног тестирања пружају јасне смернице за правилну примену и коришћење тестова у развоју софтвера, што резултује поузданим и квалитетним производима. Са правилно постављеним јединичним тестовима, тимови за развој софтвера могу брзо реаговати на промене, побољшати стабилност кода и смањити време потребно за испоруку софтвера.

4. Предности и мане јединичног тестирања

Јединично тестирање игра кључну улогу у развоју софтвера, омогућавајући програмерима да на време идентификују и реше проблеме, што води ка поузданијем и квалитетнијем софтверу. Иако доноси бројне предности, овај вид тестирања није без изазова и ограничења. У овом поглављу ћемо детаљније размотрити и предности и мане јединичног тестирања.

4.1. Предности јединичног тестирања

1. **Побољшање квалитета кода** - Јединично тестирање значајно доприноси побољшању квалитета кода јер омогућава рано откривање и исправљање грешака.

[6] Школа кода вебсајт, Јединични тестови: https://skolakoda.github.io/jedinicni-testovi
[7] Јединични тестови, Математички Факултет, Универзитет у Београду: https://poincare.matf.bg.ac.rs/~jelenagr/rs2/Testiranje.pdf
[8] QA посао вебсајт, аутоматизовано тестирање: https://qaposao.wordpress.com/2020/12/07/automatizovano-testiranje/

Програмери су приморани да размишљају о сврси и понашању компоненти пре него што имплементирају тестове, што доводи до бољег и структуиранијег кода. Кроз редовно извршавање тестова, могуће је одржати висок ниво квалитета током целокупног развојног процеса [9].

2. **Рано откривање и исправљање грешака** - Једна од највећих предности јединичног тестирања је могућност раног откривања грешака. Како се тестови извршавају на појединачним јединицама кода, грешке се могу идентификовати и исправити пре него што постану део већег система, где би њихово откривање и отклањање било много сложеније и скупље [5].
3. **Документација кода** - Јединични тестови служе као вид документације, јер показују како компонента треба да се понаша у различитим сценаријима. Ови тестови пружају вредан увид новим члановима тима или онима који треба да одржавају и рефакторишу код након почетне имплементације.
4. **Лакоћа одржавања кода** - Јединично тестирање олакшава одржавање кода јер омогућава брзу идентификацију проблема након промена у коду. Када се код рефакторише, једноставним покретањем постојећих тестова могу се проверити све критичне функционалности и осигурати да нове промене нису утицале на постојећи код.
5. **Тестна аутоматизација** - Јединични тестови су обично једноставни за покретање и могу се лако аутоматизовати. Ово омогућава програмерима да брзо добију повратне информације о стању кода након сваке промене, што је од суштинске важности за континуирани развој (CI/CD) и интеграцију.

4.2. Мане и ограничења јединичног тестирања

Иако јединично тестирање пружа бројне предности, оно није без својих изазова и ограничења:

1. **Трошкови имплементације** - Имплементација јединичних тестова захтева време и ресурсе, што може повећати трошкове развоја софтвера. Писање квалитетних тестова захтева дубље разумевање система, а додатно време које је потребно за писање и одржавање тестова може значајно повећати трајање пројекта.
2. **Изазови у одржавању** - Како пројекат расте, број јединичних тестова се повећава, што може довести до изазова у њиховом одржавању. Ако се захтеви или структура кода промене, сви релевантни тестови морају бити ажурирани како би одражавали те промене. Ово може довести до великих напора у одржавању тестног пакета.
3. **Покривеност кода** - Обезбеђивање адекватне покривености кода јединичним тестовима може бити изазов. Иако је могуће постићи високу покривеност, није увек лако осигурати да су сви делови кода адекватно тестирани. Недовољна покривеност може оставити критичне делове кода неиспитаним, што повећава ризик од грешака у продукцији [10].

---

[9] Никола Ковачевић, *Јединично тестирање помоћу xUnit алата (у C#.NET апликацијама),* Математички Факултет, Универзитет у Београду: http://www.verifikacijasoftvera.matf.bg.ac.rs//vs/predavanja/02_testiranje/16_NikolaKovacevic_xUnitTestiranje.pdf

[10] Никола Ковачевић, *Јединично тестирање помоћу xUnit алата (у C#.NET апликацијама),* Математички Факултет, Универзитет у Београду: http://www.verifikacijasoftvera.matf.bg.ac.rs//vs/predavanja/02_testiranje/16_NikolaKovacevic_xUnitTestiranje.pdf

4. **Ограничена обухватност** - Јединични тестови су усмерени на тестирање појединачних компоненти, што значи да не могу увек ухватити сложене сценарије који се јављају када различите компоненте раде заједно. За тестирање комплексних система потребно је користити друге врсте тестирања, као што су интеграционо и системско тестирање [11].

5. Технике и алати за Јединично тестирање

Јединично тестирање је кључни део развојног процеса, а употреба одговарајућих техника и алата може значајно унапредити његову ефикасност и квалитет. Радни оквири (енг. frameworks) за тестирање пружају структуру и алате за писање, организовање и извршавање јединичних тестова, омогућавајући програмерима да дефинишу тестне случајеве, провере очекиване резултате и изврше тестове на ефикасан и организован начин.

5.1. Радни оквири за тестирање

Радни оквири за тестирање су важни за било који развојни процес који укључује јединично тестирање. Они не само да пружају структуру за организовање и извршавање тестова, већ и омогућавају једноставну интеграцију тестова у свакодневне развојне активности. Ови оквири омогућавају програмерима да лако креирају тестне случајеве, провере очекиване резултате, аутоматизују извршавање тестова и добију брзу повратну информацију о стању кода.

Примери популарних радних оквира укључују:

- **JUnit** за Јаву: JUnit је широко коришћен оквир за јединично тестирање у Јава заједници, који омогућава једноставно креирање и извршавање тестова.
- **NUnit** за C#: NUnit је најпопуларнији оквир за тестирање у C# екосистему, нуди моћне алате за тестирање и лако се интегрише у Visual Studio.
- **pytest** за Python: pytest је флексибилан и моћан алат за тестирање у Python-у, који омогућава једноставно креирање комплексних тестова.
- **Mocha** за JavaScript: Mocha је популаран оквир за тестирање у JavaScript-у, који омогућава лако писање тестова за Node.js и веб апликације.

Ови радни оквири пружају различите функционалности за асертирање, организовање тестова, извршавање тестова у групама, управљање тестним подацима и многе друге корисне алате који омогућавају да процес тестирања буде ефикасан и скалабилан.

5.2. Радни оквири за имитацију (Mocking Frameworks)

Радни оквири за имитацију, познати и као Mocking Frameworks, омогућавају програмерима да креирају лажне објекте (mock објекте) који замењују стварне објекте у тестовима. Ова техника је посебно корисна када је потребно изоловати компоненту коју тестирате од њених спољних зависности, као што су базе података, веб сервиси или спољни апликативни програмски интерфејси.

[11] Никола Ковачевић, *Јединично тестирање помоћу xUnit алата (у C#.NET апликацијама),* Математички Факултет, Универзитет у Београду:

Изолација компоненти омогућава фокусирано тестирање специфичних аспеката кода без утицаја спољних фактора, чиме се постиже већа прецизност и стабилност тестова.

Примери популарних радних оквира за имитацију укључују:

- **Mockito** за Јаву: Mockito је широко коришћен оквир за имитацију у Јави, који омогућава креирање mock објеката и дефинисање њиховог понашања у различитим сценаријима.
- **unittest.mock** за Python: Овај модул у оквиру Python-овог unittest пакета омогућава једноставно креирање mock објеката и тестирање њихових интеракција.
- **Moq** за C#: Moq је популаран mocking framework за C#, који нуди једноставну и интуитивну синтаксу за креирање mock објеката и дефинисање њиховог понашања.
- **sinon** за JavaScript: sinon је свеобухватан алат за креирање mock објеката и шпијуна (енг. spies) у JavaScript-у, омогућавајући тестирање комплексних интеракција у веб апликацијама.

Ови радни оквири омогућавају креирање лажних објеката, постављање очекивања и понашања, проверу интеракција са тим објектима и многе друге функционалности које олакшавају тестирање у изолованим условима. Ово је посебно корисно у великим системима где је потребно тестирати одређене делове кода без потребе за укључивањем свих зависности.

5.3. Алати за CI/CD

У претходном поглављу смо поменули важност интеграције јединичних тестова у процес континуиране интеграције и континуиране испоруке (CI/CD). Алати за CI/CD омогућавају аутоматизацију свих аспеката развоја софтвера, укључујући компилацију, тестирање, и дистрибуцију. Ови алати осигуравају да се промене у коду аутоматски интегришу, тестирају и испоручују у производно окружење, што смањује ризик од грешака и побољшава ефикасност тима.

Примери популарних алата за CI/CD укључују:

- **Jenkins**: Један од најпопуларнијих CI/CD алата, омогућава подешавање аутоматизованих радних токова и извршавање тестова при свакој промени у коду.
- **Travis CI**: Веома популаран у open-source заједници, Travis CI омогућава једноставну интеграцију са GitHub-ом и аутоматско тестирање и изградњу пројеката.
- **GitLab CI/CD**: Интегрисани алат у оквиру GitLab-а, који омогућава континуирану интеграцију и испоруку, са снажном подршком за верзионисање и управљање пројектима.
- **CircleCI**: Модеран CI/CD алат који омогућава брзо и флексибилно подешавање радних токова за аутоматизоване процесе.

Ови алати омогућавају постављање аутоматизованих радних токова, праћење промена у репозиторијуму, извршавање тестова при свакој промени, аутоматско издавање верзија софтвера и многе друге функционалности које олакшавају процес развоја и испоруке софтвера.

6. Имплементација јединичних тестова

Имплементација јединичних тестова представља кључни корак у развоју софтвера који омогућава поуздано и ефикасно тестирање кода и компоненти. Овај процес игра суштинску улогу у откривању и исправљању грешака, обезбеђујући конзистентност кода и унапређени квалитет софтвера. Јединично тестирање омогућава програмерима да осигурају да свака компонента функционише на предвиђен начин и да се рано идентификују потенцијални проблеми.

Пре писања јединичних тестова, важно је припремити софтверски пројекат за тестирање. Ово укључује постављање окружења за тестирање, укључујући конфигурацију библиотека и алата за тестирање, као и подешавање зависности пројекта. Тестно окружење треба да буде изоловано и одвојено од продукционог окружења како би се избегли нежељени ефекти током тестирања. Такође, важно је обезбедити да је код пројекта добро организован и да су компоненте кода раздвојене на начин који омогућава лако тестирање.

Након што је пројекат припремљен, програмери могу започети писање јединичних тестова. Ови тестови треба да покривају функционалности и компоненте кода, проверавајући њихово исправно понашање у различитим сценаријима. Циљ јединичног тестирања је да се тестирају мали, изоловани делови кода, обично функције или методе, како би се осигурало да раде исправно у свим предвиђеним ситуацијама.

При писању тестова, потребно је обезбедити да они покривају различите сценарије и граничне случајеве (енг. edge cases), укључујући и оне који могу бити неочекивани или ретки. Ово осигурава да је код отпоран на различите врсте улазних података и да се понаша стабилно чак и у екстремним условима.

Јединични тестови се обично пишу у програмском језику који користи циљани софтвер, и уз помоћ специфичних библиотека или оквира за тестирање. На пример, ако се ради о пројекту у C#, тестови ће се најчешће писати користећи NUnit или сличан оквир. За сваку компоненту софтвера треба написати неколико тестова који покривају различите сценарије и случајеве употребе, како би се осигурала потпуна покривеност кода.

Када су јединични тестови написани, следећи корак је њихово извршавање и анализа резултата. Тестови се покрећу у аутоматизованом окружењу, и резултати тестирања пружају директне информације о исправности и стабилности кода. У идеалном случају, сви тестови треба да прођу успешно, што указује да се систем понаша онако како је предвиђено.

Јединични тестови треба да буду лако читљиви и разумљиви, не само за програмере који су их написали, већ и за остатак развојног тима. Ово је посебно важно за одржавање и прилагођавање тестова у складу са променама у коду. Како се пројекат развија и код се мења, тестови морају бити редовно ажурирани како би одражавали тренутно понашање софтвера.

Редовно ажурирање тестова је кључно за одржавање њихове релевантности и тачности. Ово обезбеђује да тестови и даље покривају све релевантне делове кода и да пружају поуздану заштиту од грешака које могу бити уведене током будућих измена и проширења система.

7. NUnit радни оквир за Тестирање у C#

NUnit је један од најпопуларнијих оквира за јединично тестирање у .NET екосистему. Развијен је као пројекат отвореног кода и представља стандард за тестирање C# апликација. У овом поглављу ћемо објаснити шта је NUnit, које су његове предности и зашто је толико популаран међу C# програмерима. Такође ћемо показати како се користи NUnit у контексту јединичног тестирања на примерима које ћемо приказати у наредним поглављима.

NUnit је тестни оквир који омогућава програмерима да лако креирају и извршавају јединичне тестове за своје C# апликације. Он обезбеђује све потребне алате и анотације за дефинисање, организовање и извршавање тестова. NUnit је инспирисан JUnit оквиром за Java, али је прилагођен за .NET радни оквир и C# језик.[12]

Предности NUnit-а су следеће:

1. Једноставност коришћења: NUnit је једноставан за коришћење, чак и за програмере који су нови у јединичном тестирању. Интуитивни атрибути и једноставна синтакса омогућавају брзо креирање тестова.
2. Флексибилност и проширивост: NUnit подржава различите типове тестирања, укључујући синхроне и асинхроне тестове. Поред тога, могуће га је проширити додатним библиотекама.
3. Широка подршка за Assert-ове: NUnit обезбеђује разноврсне Assert методе које омогућавају провере различитих услова у тестовима. Ово укључује проверу једнакости, неједнакости, бацања изузетака и још много тога.
4. Интеграција са CI/CD алатима: NUnit се лако интегрише са континуираним интеграционим и континуираним испорукама (CI/CD) алатима као што су Jenkins, Azure DevOps и други, што омогућава аутоматско извршавање тестова током процеса изградње и испоруке софтвера.
5. Подршка за различите платформе: NUnit подржава тестирање на различитим платформама, укључујући .NET Framework, .NET Core и .NET 5/6.
6. Лака интеграција са Visual Studio-ом: NUnit долази уз Visual Studio, што значи да програмери не морају додатно да га инсталирају. Ово значајно олакшава намештање и коришћење NUnit-а, јер се тестови могу креирати и извршавати директно из Visual Studio окружења.[13]

NUnit је веома популаран међу C# програмерима из више разлога:

- Зрела и стабилна заједница: NUnit има дугу историју и велику заједницу корисника који доприносе његовом развоју и подршци. То значи да је оквир добро тестиран и подржан.
- Добра документација и примери: NUnit има обимну документацију и бројне примере који омогућавају лако учење и употребу.
- Подршка за савремене развојне праксе: NUnit је у складу са савременим развојним праксама и алатима, што га чини природним избором за програмере који желе да примене најбоље праксе у својим пројектима.

[12] НУнит званични вебсајт: https://nunit.org/
[13] Мајкрософт вебсајт, покретање јединичних тестова: https://learn.microsoft.com/en-us/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code?view=vs-2022

8. Различити Assert-ови у NUnit радном оквиру

Један од кључних делова NUnit радног оквира су Assert методе које омогућавају проверу различитих услова у тестовима. [14]Assert-ови су суштински за потврђивање исправности кода и за осигурање да тестови адекватно верификују очекивано понашање апликације. У овом поглављу, објаснићемо различите типове Assert-ова који су доступни у NUnit-у и за шта се користе.[15]

8.1. Основни Assert-ови

Assert.AreEqual

Овај Assert се користи за проверу да ли су две вредности једнаке. Ако нису, тестирана функционалност неће проћи тестирање.

```
[Test]
public void AreEqualExample()
{
    int expected = 5;
    int actual = 2 + 3;
    Assert.AreEqual(expected, actual, "Values are not equal!");
}
```

У овом примеру, проверавамо да ли је резултат сабирања 2 и 3 једнак очекиваној вредности 5. Ако није, тестирана функционалност неће проћи тестирање са поруком "Values are not equal!".

Assert.AreNotEqual

Овај Assert се користи за проверу да ли две вредности нису једнаке. Ако су вредности једнаке, тестирана функционалност неће проћи тестирање.

```
[Test]
public void AreNotEqualExample()
{
    int unexpected = 5;
    int actual = 2 + 2;
    Assert.AreNotEqual(unexpected, actual, "Values are equal, but they
shouldn't be!");
}
```

У овом примеру, проверавамо да ли резултат сабирања 2 и 2 није једнак неочекиваној вредности 5. Ако су вредности једнаке, тестирана функционалност неће проћи тестирање са поруком "Values are equal, but they shouldn't be!".

Assert.IsTrue

1. [14] Јунит тестирање, Факултет Техничких Наука, Универзитет у Новом Саду http://www.esi.ftn.uns.ac.rs/images/predmeti/res/Vezbe_2021/V6%20-%20Unit%20testing.pdf

1. [15] НУнит званична документација за Асертове: https://docs.nunit.org/articles/nunit/writing-tests/assertions/assertions.html

Овај Assert се користи за проверу да ли је услов тачан (true). Ако услов није тачан, тестирана функционалност неће проћи тестирање.

```
[Test]
public void IsTrueExample()
{
    bool condition = 5 > 3;
    Assert.IsTrue(condition, "Condition is not true!");
}
```

У овом примеру, проверавамо да ли је услов "5 је веће од 3" тачан. Ако услов није тачан, тестирана функционалност неће проћи тестирање са поруком "Condition is not true!".

Assert.IsFalse

Овај Assert се користи за проверу да ли је услов нетачан (false). Ако услов није нетачан, тестирана функционалност неће проћи тестирање.

```
[Test]
 public void IsFalseExample()
 {
     bool condition = 5 < 3;
     Assert.IsFalse(condition, "Condition is not false!");
 }
```

У овом примеру, проверавамо да ли је услов "5 је мање од 3" нетачан. Ако услов није нетачан, тестирана функционалност неће проћи тестирање са поруком "Condition is not false!".

Assert.IsNull

Овај Assert се користи за проверу да ли је објекат null. Ако објекат није null, тестирана функционалност неће проћи тестирање.

```
[Test]
public void IsNullExample()
{
    object obj = null;
    Assert.IsNull(obj, "Object is not null!");
}
```

У овом примеру, проверавамо да ли је објекат obj null. Ако објекат није null, тестирана функционалност неће проћи тестирање са поруком "Object is not null!".

Assert.IsNotNull

Овај Assert се користи за проверу да ли објекат није null. Ако објекат јесте null, тестирана функционалност неће проћи тестирање.

```
[Test]
public void IsNotNullExample()
{
    object obj = new object();
    Assert.IsNotNull(obj, "Object is null!");
}
```

У овом примеру, проверавамо да ли објекат obj није null. Ако објекат јесте null, тестирана функционалност неће проћи тестирање са поруком "Object is null!".

Assert.Throws

Овај Assert се користи за проверу да ли одређени код баца изузетак. Ако изузетак није бачен, тестирана функционалност неће проћи тестирање.

```
[Test]
public void ThrowsExample()
{
    Assert.Throws<ArgumentException>(() => { throw new ArgumentException(); },
"Exception was not thrown!");
}
```

У овом примеру, проверавамо да ли код унутар витичастих заграда баца ArgumentException. Ако изузетак није бачен, тестирана функционалност неће проћи тестирање са поруком "Exception was not thrown!".

Assert.DoesNotThrow

Овај Assert се користи за проверу да одређени код не баца изузетак. Ако изузетак буде бачен, тестирана функционалност неће проћи тестирање.

```
[Test]
public void DoesNotThrowExample()
{
    Assert.DoesNotThrow(() => { int a = 5; }, "Exception was thrown!");
}
```

У овом примеру, проверавамо да ли код унутар витичастих заграда не баца изузетак. Ако изузетак буде бачен, тестирана функционалност неће проћи тестирање са поруком "Exception was thrown!".

Assert.AreSame

Овај Assert се користи за проверу да ли две референце упућују на исти објекат. Ако референце не упућују на исти објекат, тестирана функционалност неће проћи тестирање.

```
[Test]
public void AreSameExample()
{
    object obj1 = new object();
    object obj2 = obj1;
    Assert.AreSame(obj1, obj2, "References do not point to the same object!");
}
```

У овом примеру, проверавамо да ли референце obj1 и obj2 упућују на исти објекат. Ако референце не упућују на исти објекат, тестирана функционалност неће проћи тестирање са поруком "References do not point to the same object!".

Assert.AreNotSame

Овај Assert се користи за проверу да ли две референце не упућују на исти објекат. Ако референце упућују на исти објекат, тестирана функционалност неће проћи тестирање.

```
[Test]
public void AreNotSameExample()
{
    object obj1 = new object();
    object obj2 = new object();
    Assert.AreNotSame(obj1, obj2, "References point to the same object!");
}
```

У овом примеру, проверавамо да ли референце obj1 и obj2 не упућују на исти објекат. Ако референце упућују на исти објекат, тестирана функционалност неће проћи тестирање са поруком "References point to the same object!".

Assert-ови у NUnit-у пружају широк спектар могућности за проверу различитих услова у јединичним тестовима. Коришћењем ових Assert-ова, програмери могу осигурати да њихов код функционише исправно у свим предвиђеним сценаријима и да се одговарајући изузеци бацају у случају грешака. Правилно коришћење Assert-ова је кључно за писање ефикасних и поузданих јединичних тестова.

9. Јединично тестирање на примеру калкулатора

Као пример апликације, користимо једноставну класу калкулатора под називом Calculator. Ова класа садржи четири основне аритметичке операције: сабирање, одузимање, множење и дељење. Код за ову класу је приказан у наставку:

```
namespace CalculatorApp
{
    public class Calculator
    {
        public int add (int x, int y)
        {
            return x + y;
        }
        public int sub(int x, int y)
        {
            return x - y;
        }
        public int mul(int x, int y)
        {
            return x * y;
        }
        public int div(int x, int y)
        {
            return x / y - 1;
        }
    }
}
```

Са намером смо убацили грешку у функцији за дељење како би нам јединични тестови помогли да је идентификујемо и потом отклонимо.

За тестирање ове апликације, користићемо NUnit, популаран оквир за јединично тестирање који смо представили у претходном поглављу. NUnit омогућава лако креирање и извршавање јединичних тестова. Сви тестови се организују у тест класе, а појединачни тестови су означени атрибутом [Test].

У NUnit оквиру постоји и атрибут [SetUp] који се користи за означавање методе која се извршава пре сваког теста. Ова метода омогућава припрему објеката и постављање почетних услова потребних за тестирање.

9.1. Постављање и иницијализација

Методом Setup означеном са [SetUp], креирамо нову инстанцу класе Calculator пре сваког теста. Ова метода се извршава пре сваког теста како би се обезбедило да сваки тест има чисто окружење:

```
private Calculator calculator;

[SetUp]
public void Setup()
{
    calculator = new Calculator();

}
```

9.2. Тестирање сабирања

Тестирање сабирања се врши у две методе. Прва метода, Test_addition, проверава да ли је резултат сабирања две вредности тачан:

```
[Test]
public void Test_addition()
{
    int a = 123;
    int b = 325;

    int result = calculator.add(a, b);

    Assert.AreEqual(a + b, result);

}
```

Друга метода, Test_addition_not_equal, проверава да резултат сабирања није једнак некој погрешној вредности:

```
[Test]
public void Test_addition_not_equal()
{
    int a = 2142;
    int b = 22;

    int result = calculator.add(a, b);

    Assert.AreNotEqual(a + b + 1, result);
```

```
}
```

9.3. Тестирање одузимања

Методе за тестирање одузимања су сличне онима за сабирање. Прва метода, Test_subtraction, проверава тачност резултата одузимања:

```
[Test]
public void Test_subtraction()
{
    int a = 9;
    int b = 10;

    int result = calculator.sub(a, b);

    Assert.AreEqual(a - b, result);

}
```

Друга метода, Test_subtraction_not_equal, проверава да резултат одузимања није једнак некој погрешној вредности:

```
[Test]
public void Test_subtraction_not_equal()
{
    int a = 42;
    int b = 10;

    int result = calculator.sub(a, b);

    Assert.AreNotEqual(a - b + 1, result);

}
```

9.4. Тестирање множења

Методе за тестирање множења су такође сличне. Прва метода, Test_multiplication проверава тачност резултата множења:

```
[Test]
public void Test_multiplication()
{
    int a = 91;
    int b = 510;

    int result = calculator.mul(a, b);

    Assert.AreEqual(a * b, result);

}
```

Друга метода, Test_multiplication_not_equal , проверава да резултат множења није једнак некој погрешној вредности:

```
[Test]
public void Test_multiplication_not_equal()
{
    int a = 412;
    int b = 13;

    int result = calculator.mul(a, b);

    Assert.AreNotEqual(a * b + 1, result);

}
```

9.5. Тестирање дељења

Методе за тестирање дељења су приказане у наставку. Прва метода, Test_division, проверава тачност резултата дељења:

```
[Test]
public void Test_division()
{
    int a = 10;
    int b = 2;

    int result = calculator.div(a, b);

    Assert.AreEqual(a / b, result);

}
```

Друга метода, Test_division_not_equal , проверава да резултат дељења није једнак некој погрешној вредности:

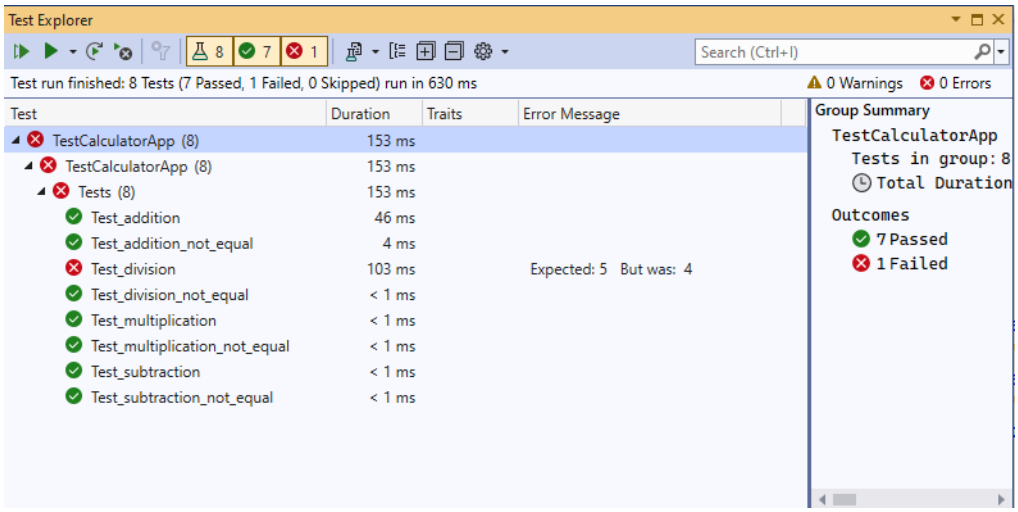
```
[Test]
public void Test_division_not_equal()
{
    int a = 220;
    int b = 5;

    int result = calculator.div(a, b);

    Assert.AreNotEqual(a / b + 1, result);
}
```

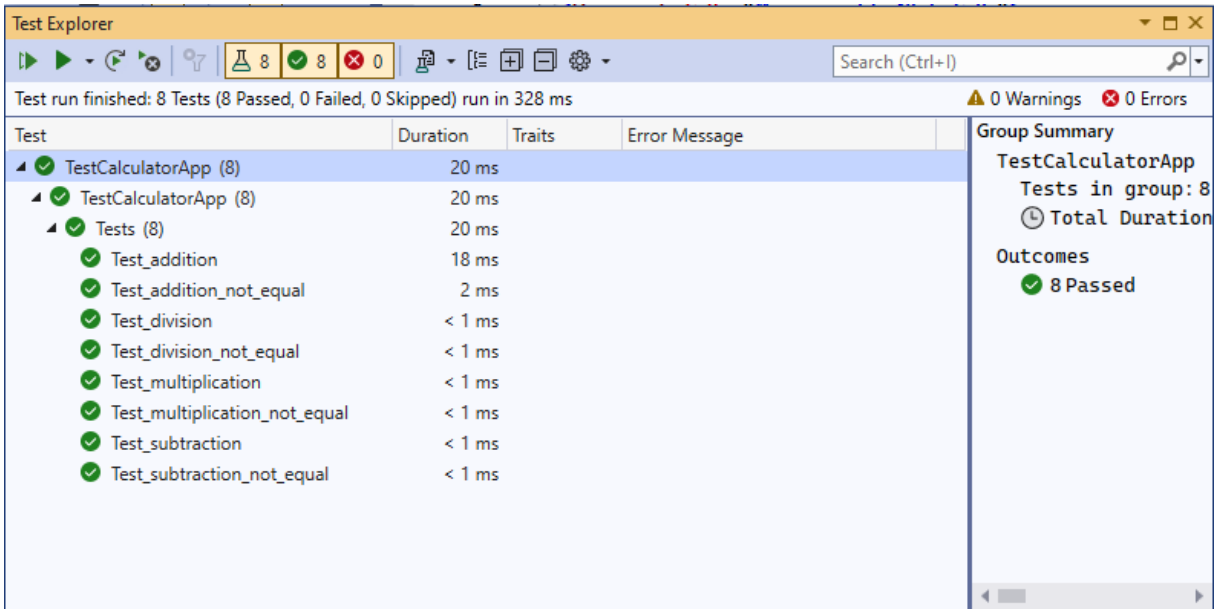
9.6. Резултати тестирања

Након покретања тестова, можемо да видимо да су сви прошли осим јединичног теста за дељење. Ово је било очекивано како смо свесно убацили грешку у тој функцији.



Слика 1. Неуспешан тест дељења

Након исправљања грешке у функцији за дељење и поновног покретања тестова, можемо да видимо да сви тестови пролазе и да се наша апликација понаша на предвиђен начин.



Слика 2. Успешни тестови

Коришћењем једноставног примера калкулатора, приказали смо како се имплементирају и извршавају јединични тестови за различите функционалности апликације. Ови тестови помажу у осигуравању исправности и поузданости кода, чиме се повећава квалитет софтвера.

10. Јединично тестирање на примеру банкарске апликације

Након примера са калкулатором, узимамо мало комплекснији пример са банкарском апликацијом. Банкарска апликација пружа могућност депоновања новца, подизања новца и трансфера новца на други налог. У овом поглављу, приказаћемо како се јединично тестирају ове функционалности користећи NUnit.

Банкарска апликација коју ћемо користити у овом примеру садржи класу BankAccount која има методе за депоновање, подизање и трансфер новца. Код за ову класу је приказан у наставку:

```
namespace BankingApp
{
    public class BankAccount
{
    public int Balance { get; private set; }

    public BankAccount(int initialBalance)
    {
        Balance = initialBalance;
    }

    public void Deposit(int amount)
    {
        if (amount <= 0)
        {
            throw new ArgumentException("Deposit amount must be
positive");
        }
        Balance += amount;
    }

    public void Withdraw(int amount)
    {
        if (amount <= 0)
        {
            throw new ArgumentException("Withdrawal amount must be
positive");
        }
        if (amount > Balance)
        {
            throw new InvalidOperationException("Insufficient funds");
        }
        Balance -= amount;
    }

    public void Transfer(BankAccount targetAccount, int amount)
    {
        if (targetAccount == null)
        {
            throw new ArgumentNullException(nameof(targetAccount));
        }
        Withdraw(amount);
        targetAccount.Deposit(amount);
    }
}
}
```

Банкарска апликација коју ћемо тестирати има следеће функционалности:

1. Депоновање новца:
    - Функционалност: Метод Deposit омогућава депоновање новца на налог.
    - Очековано функционисање: Корисник може депоновати позитивну суму новца, која ће бити додата на тренутно стање налога. Депоновање негативне или нулте суме новца треба да изазове изузетак ArgumentException.
2. Подизање новца:
    - Функционалност: Метод Withdraw омогућава подизање новца са налога.
    - Очековано функционисање: Корисник може подићи позитивну суму новца мању или једнаку тренутном стању налога. Подизање веће суме од стања на налогу треба да изазове изузетак InvalidOperationException. Подизање негативне или нулте суме новца треба да изазове изузетак ArgumentException.
3. Трансфер новца:
    - Функционалност: Метод Transfer омогућава трансфер новца са једног налога на други.
    - Очековано функционисање: Корисник може пренети новац са свог налога на други налог ако има довољно средстава. Ако су средства недовољна или ако је циљни налог неважећи (null), метода треба да баци одговарајући изузетак (InvalidOperationException за недовољна средства и ArgumentNullException за неважећи налог).

10.1. Тестирање депоновања

У овом делу ћемо тестирати функционалност депоновања новца тако што ћемо позивати методу Deposit. Тестираћемо позитивне и негативне сценарије како бисмо осигурали исправно понашање ове функционалности. Иницијално стање рачуна је 100.

Deposit_PositiveAmount_IncreasesBalance: Овај тест проверава да ли депоновање позитивне суме новца исправно повећава стање на рачуну. Циљ је да се осигура да метода Deposit исправно ажурира баланс.

```
[Test]
public void Deposit_PositiveAmount_IncreasesBalance()
{
    account.Deposit(50);
    Assert.AreEqual(150, account.Balance);
}
```

Deposit_NegativeAmount_ThrowsArgumentException: Овај тест проверава да ли депоновање негативне суме новца баца изузетак ArgumentException. Циљ је да се спречи депоновање неважећих вредности.

```
[Test]
public void Deposit_NegativeAmount_ThrowsArgumentException()
{
    Assert.Throws<ArgumentException>(() => account.Deposit(-50));
```

```
}
```

10.2. Тестирање подизања новца

У овом делу ћемо тестирати функционалност подизања новца тако што ћемо позивати методу Withdraw. Тестираћемо различите сценарије како бисмо осигурали да је подизање новца исправно и без грешака.

Withdraw_PositiveAmount_DecreasesBalance: Овај тест проверава да ли подизање позитивне суме новца смањује стање на рачуну. Циљ је да се осигура да метода Withdraw исправно ажурира баланс.

```
[Test]
public void Withdraw_PositiveAmount_DecreasesBalance()
{
    account.Withdraw(50);
    Assert.AreEqual(50, account.Balance);
}
```

Withdraw_AmountGreaterThanBalance_ThrowsInvalidOperationException: Овај тест проверава да ли подизање суме новца веће од тренутног стања баца изузетак InvalidOperationException. Циљ је да се спречи прекорачење рачуна.

```
[Test]
public void Withdraw_AmountGreaterThanBalance_ThrowsInvalidOperationException()
{
    Assert.Throws<InvalidOperationException>(() => account.Withdraw(150));
}
```

Withdraw_NegativeAmount_ThrowsArgumentException: Овај тест проверава да ли подизање негативне суме новца баца изузетак ArgumentException. Циљ је да се спречи подизање неважећих вредности.

```
[Test]
public void Withdraw_NegativeAmount_ThrowsArgumentException()
{
    Assert.Throws<ArgumentException>(() => account.Withdraw(-50));
}
```

10.3. Тестирање трансфера новца

У овом делу ћемо тестирати функционалност трансфера новца тако што ћемо позивати методу Transfer. Тестираћемо различите сценарије како бисмо осигурали да је трансфер новца исправан и без грешака.

Transfer_ValidAmount_UpdatesBothAccounts: Овај тест проверава да ли трансфер валидне суме новца исправно ажурира стања на оба рачуна. Циљ је да се осигура да метода Transfer исправно обрађује трансфере.

```
[Test]
public void Transfer_ValidAmount_UpdatesBothAccounts()
```

```
{
    var targetAccount = new BankAccount(50);
    account.Transfer(targetAccount, 50);

    Assert.AreEqual(50, account.Balance);
    Assert.AreEqual(100, targetAccount.Balance);
}
```

Transfer_AmountGreaterThanBalance_ThrowsInvalidOperationException: Овај тест проверава да ли трансфер суме новца веће од тренутног стања баца изузетак InvalidOperationException. Циљ је да се спречи прекорачење рачуна при трансферу.

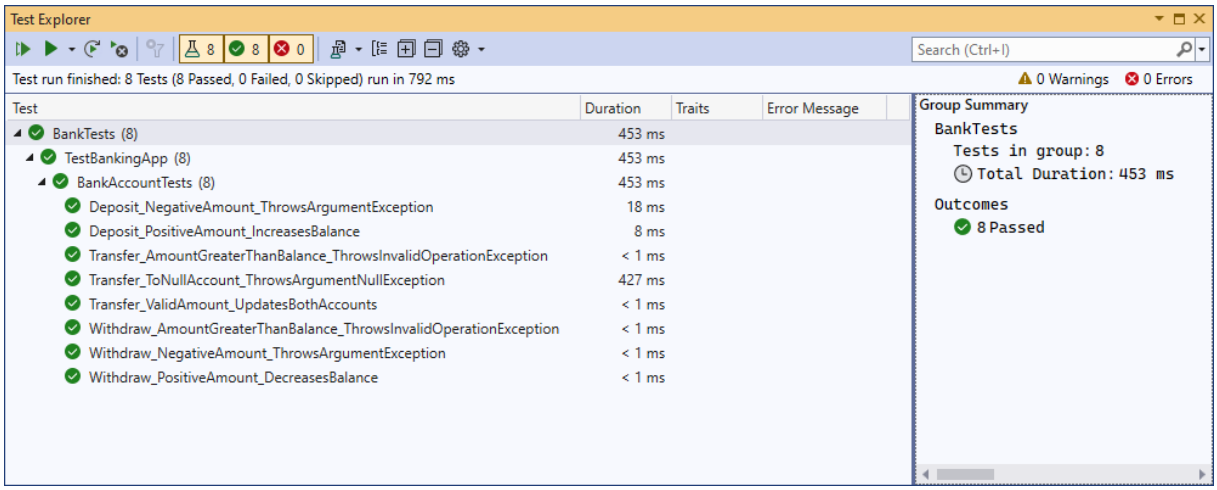
```
[Test]
public void Transfer_AmountGreaterThanBalance_ThrowsInvalidOperationException()
{
    var targetAccount = new BankAccount(50);
    Assert.Throws<InvalidOperationException>(() =>
account.Transfer(targetAccount, 150));
}
```

Transfer_ToNullAccount_ThrowsArgumentNullException: Овај тест проверава да ли трансфер новца на неважећи (null) рачун баца изузетак ArgumentNullException. Циљ је да се спречи трансфер на неважеће налоге.

```
[Test]
public void Transfer_ToNullAccount_ThrowsArgumentNullException()
{
    Assert.Throws<ArgumentNullException>(() => account.Transfer(null, 50));
}
```

10.4. Резултати тестирања

Након покренутих тестова видимо да су све функционалности апликације исправне и да апликацију можемо безбедно дати даље кориснику на коришћење.



Слика 3. Успешни тестови банковне апликације

У овом поглављу смо показали како се јединично тестирају функционалности банкарске апликације користећи NUnit. Кроз тестирање депоновања, подизања и трансфера новца, осигурали смо да све операције функционишу исправно и да се изузеци бацају у случају

невалидних операција. Овај приступ повећава квалитет и поузданост софтвера, што је кључно за банкарске апликације.

11. Закључак

Јединично тестирање представља један од најважнијих аспеката осигурања квалитета у процесу софтверског развоја. Кроз овај пројектни рад, истражили смо основне принципе, праксе, технике и алате који омогућавају ефикасно и поуздано јединично тестирање. Почевши од прегледа различитих типова тестирања, као што су интеграционо, функционално, системско и прихватно тестирање, показали смо како се јединично тестирање уклапа у шири контекст развојног циклуса.

Разматрали смо како јединично тестирање доприноси побољшању квалитета софтвера кроз рано откривање и исправљање грешака, као и кроз обезбеђивање конзистентности и стабилности софтвера. Такође, дискутовали смо о предностима и манама овог процеса, наглашавајући важност правилног планирања и примене како би се превазишли потенцијални изазови и ограничења.

Кроз практичне примере јединичног тестирања на апликацији калкулатора и банкарској апликацији, демонстрирали смо како се јединично тестирање може применити у различитим контекстима и како може допринети побољшању квалитета и поузданости софтвера. Ови примери илуструју важност тестирања како једноставних, тако и комплексних компоненти софтвера, чиме се осигурава да целокупан систем функционише у складу са очекивањима.

На крају, можемо закључити да јединично тестирање није само алат за откривање грешака, већ и средство за побољшање структуре и одрживости софтвера на дуге стазе. Увођење и примена јединичног тестирања у свакодневни радни процес доноси значајне бенефите, не само за квалитет кода, већ и за целокупан развојни тим и крајњи производ. Са правилним разумевањем и применом, јединично тестирање постаје суштински део модерног развоја софтвера, обезбеђујући стабилне, поуздане и висококвалитетне софтверске системе.

Референце

1. Др Сузана Марковић, *Основе програмирања*, Академија пословних струковних студија, Београд, 2019.
2. Заптест вебсајт, *Шта је јединично тестирање*: https://www.zaptest.com/sr/%D1%88%D1%82%D0%B0-%D1%98%D0%B5-%D1%98%D0%B5%D0%B4%D0%B8%D0%BD%D0%B8%D1%87%D0%BD%D0%BE-%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%B0%D1%9A%D0%B5-%D0%B4%D1%83%D0%B1%D0%BE%D0%BA%D0%BE-%D0%B7%D0%B0%D1%80
3. Дражен Драшковић, *Тестирање Софтвера - Интеграционо Тестирање,* Електротехнички факултет, Универзитет у Београду, Београд
4. Милена Вујошевић Јаничић, *Верификација софтвера*, Математички Факултет, Универзитет у Београду, Београд 2023
5. Школа кода вебсајт, Јединични тестови: https://skolakoda.github.io/jedinicni-testovi
6. Јединични тестови, Математички Факултет, Универзитет у Београду: https://poincare.matf.bg.ac.rs/~jelenagr/rs2/Testiranje.pdf
7. QA посао вебсајт, аутоматизовано тестирање: https://qaposao.wordpress.com/2020/12/07/automatizovano-testiranje/
8. Никола Ковачевић, *Јединично тестирање помоћу xUnit алата (у C#.NET апликацијама),* Математички Факултет, Универзитет у Београду: http://www.verifikacijasoftvera.matf.bg.ac.rs//vs/predavanja/02_testiranje/16_NikolaKovacevic_xUnitTestiranje.pdf
9. НУнит званични вебсајт: https://nunit.org/
10. Мајкрософт вебсајт, покретање јединичних тестова: https://learn.microsoft.com/en-us/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code?view=vs-2022