Pregled python alata za testiranje s primjerima

Pera, Fabio

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:679213

Rights / Prava: Attribution-ShareAlike 3.0 Unported/Imenovanje-Dijeli pod istim uvjetima 3.0

Download date / Datum preuzimanja: **2024-04-20**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Fabio Pera

PREGLED PYTHON ALATA ZA TESTIRANJE S PRIMJERIMA

DIPLOMSKI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Fabio Pera

Matični broj: 44569/16–R

Studij: Informacijsko i programsko inženjerstvo

PREGLED PYTHON ALATA ZA TESTIRANJE S PRIMJERIMA

DIPLOMSKI RAD

Mentor :

Doc. dr. sc. Marcel Maretić

Varaždin, rujan 2021.

Fabio Pera

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Pregled teorijskih koncepata testiranja i metoda za testiranje. Razrada teorijskih pricncipa testiranja kroz praktične primjere u Python programskom jeziku s naglaskom na unit testing i automated testing. Prikaz rada Python biblioteka za testiranje te njihova primjena nad konkretnim primjerima. Obrada koncepata razvoja programa pogonjenih testiranjem.

Ključne riječi: unit testing, acceptance testing, mocks, testable documentation, test driven development, acceptance testing.

Sadržaj

1. Uvod 1

2. Alati i metodologija unutar rada 4
2.1. Python 3.8 4
2.2. Pregled korištenih biblioteka 5
2.3. Biblioteke korištene za izradu grafičkog sučelja 5
2.4. Biblioteke korištene za testiranje 6
2.4.1. Unittest biblioteka 7
2.4.2. PyTest biblioteka 7
2.4.3. Doctest biblioteka 8
2.4.4. Robot Framework okvir za testiranje 8
2.4.5. Gherkin i Behave biblioteke za ponašanjem pogonjeno testiranje 9
2.5. Ostale korištene biblioteke 10
2.6. Testiranjem pogonjen razvoj 10
2.7. Ponašanjem pogonjeno testiranje 11

3. Testiranje programskih proizvoda 13
3.1. Testiranja prema razini pristupa programskom kodu 13
3.1.1. Testiranje metodom bijele kutije 13
3.1.2. Testiranje metodom crne kutije 14
3.1.3. Testiranje metodom sive kutije 14
3.2. Vrste testiranja prema razinama 15
3.2.1. Jedinično testiranje 15
3.2.2. Integracijsko testiranje 16
3.2.3. Sustavsko tesiranje 17
3.2.4. Testiranje prihvatljivosti 17

4. Praktični prikaz testiranja u Pythonu 19
4.1. Podaci s kojima aplikacija radi 19
4.2. Razvoj i testiranje glavnog izbornika 20
4.3. Testiranje dohvata podataka iz baze 23
4.4. Funkcionalnost ažuriranja i testiranje točnosti podataka 29
4.5. Integracijsko testiranje komponenti čitanja i ažuriranja 34
4.6. Testiranje i razvoj komponente za dodavanje novih zapisa 35
4.7. Integriranje komponente za dodavanje zapisa u sustav 41
4.8. Testiranje i dodavanje komponente za brisanje zapisa 43

4.9. Integracija komponente za brisanje zapisa 45
4.10. Korištenje oponašanja u testiranju 47
4.11. Testiranje i dodavanje ekrana s grafičkim podacima 48
4.12. Integracija grafičkog prikaza mjera u sustav 54
4.13. Testiranje ponašanja pomoću Gherkin i Behave biblioteka 57
4.14. Testiranje unutar dokumentacije 61
4.15. Automatizirano testiranje pomoću Robot Frameworka 65

5. Zaključak 71

Popis literature 74

Popis slika 76

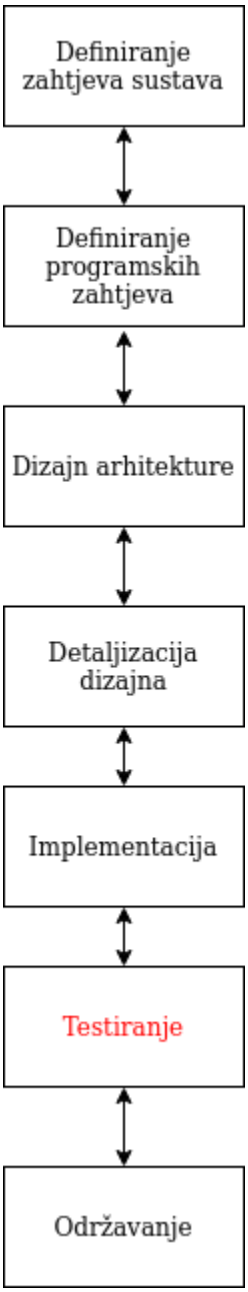
1. Uvod

Unutar ovog diplomskog rada bavit ću se temom testiranja programskog rješenja unutar programskog jezika Python. Testiranje je danas standardan dio razvoja programskog proizvoda, a svi veći kao i svi ozbiljniji projekti imaju timove ljudi koji se bave isključivo testiranjem programa. Iako se testiranje prilikom razvoja smatra nečim što se podrazumijeva prilikom isporuke programa često vidimo brojne primjere aplikacija koja odlazi u produkciju, a da nisu testirani svi aspekti te da zbog takvog pristupa prema testiranju devalviraju svoj rad i svoj konačni proizvod. Postoje brojni razlozi zbog kojih se to događa. Često su razlozi čisto ekonomski (manjak ljudi za testiranje, pritisak rokova unutar kojih se proizvod mora isporučiti), a ponekad su i razlozi da testiranje provode ljudi koji su sudjelovali u izradi aplikacije te zbog toga nemaju "vanjski pogled" na aplikaciju.

Već je rečeno da je testiranje sastavni dio ciklusa razvoja programskog proizvoda, ali nije točno objašnjeno što točno obuhvaća taj pojam. Iako se čini samoobjašnjiv često se pojam testiranja trivijalizira te se poistovjećuje s korištenjem same aplikacije te prijavljivanjem bugova na taj način. Iako je to jedan od načina i metoda za testiranje nikako ga ne smijemo smatrati jedinim. Testiranje najlakše možemo definirati kao proces unutar razvoja aplikacije prilikom kojega provjeravamo radi li aplikacija ono što smatramo da bi trebala. Također poanta testiranja nije u tome da se isprave greške koje su napravljene unutar aplikacije već da se spriječi implementacija pogrešaka, smanjuje se trošak razvoja aplikacije te se poboljšavaju same performanse aplikacije. Unutar samog testiranja postoje brojne podjele (s obzirom na domenu koja se testira, s obzirom na način na koji testiramo, na vrijeme u životnom ciklusu kada testiramo, dio programa koji testiramo) dok bi neka okvirna podjela bila na acceptance testing, integration testing, unit testing, stress testing, validation testing itd.[1] S ovim pojmovima ću se više baviti kroz praktične primjere u kasnijem dijelu rada, ali ih trenutno valja spomenuti kako bi se dobio dojam što se sve obuhvaća s testiranjem.

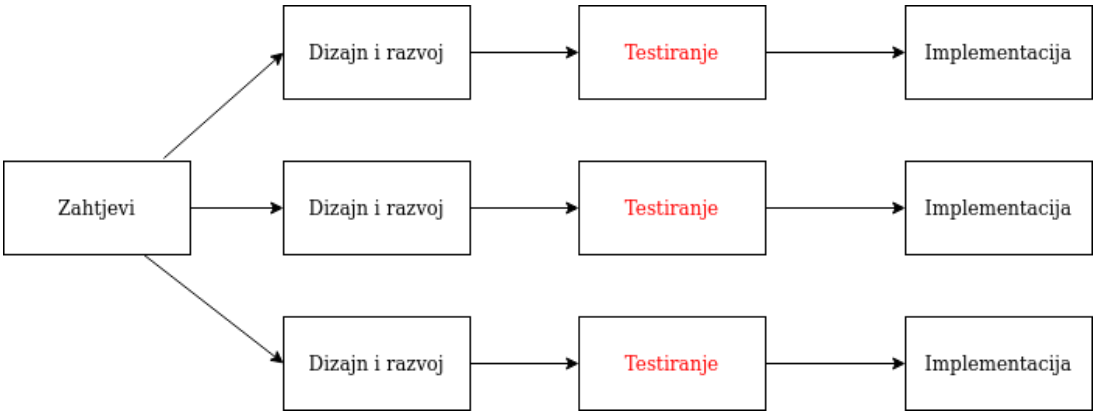
Nekoliko je puta spomenuto da je testiranje dio životnog ciklusa razvoja programskih proizvoda. Unutar samog ciklusa testiranje se obično nalazi neposredno prije uvođenja sustava. Iako se testiranje može obavljati kontinuirano (u nekim slučajevima je i poželjno) najčešće vrijeme za testiranje se nalazi nakon što je proizvod razvijen te se testira kako bi se filtrirale greške prema klijentima te se vratilo na razvoj ako se pronađu neke greške. Također pozicija testiranja unutar ciklusa razvoja ovisi i od pristupa kojeg koristimo u razvoju. Isto tako uloga testiranja i veličina programskog proizvoda koja se testira u fazi testiranja (dio ili cijeli program) ovisi od pristupa.

Prvi primjer koji ćemo uzeti je vodopadni pristup koji je jedan od tradicionalno najpopularnijih. Unutar ovog pristupa razvoj je podijeljen u faze i nakon što je odrađena cijela jedna faza (nad cijelim programom, nema podjela na manje dijelove i paralelnog rada) prelazi se na drugu fazu (ako se u nekoj fazi utvrdi greška vraćamo se jedan korak unatrag). Unutar ovog pristupa testira se cijela aplikacija te se testira između faze pisanja koda i faze održavanja rješenja (testira se prije davanja proizvoda klijentu). [2]



Slika 1: Testiranje u vodopadnom razvoju (Prema: Rastogi, 2015)

Drugi popularan pristup je iterativni pristup kod kojeg se projekt dijeli na manje dijelove te se nad svakim od njih provodi skup koraka. U ovom načinu testiramo samo dijelove programa, a ne cijeli program. Također kako ne testiramo cijeli sustav već dio po dio vrlo je važno obratiti pažnju na konačno testiranje integracije dijelova programa u cjelinu. [2]



Slika 2: Testiranje u iterativnom razvoju (Prema: Rastogi, 2015)

2. Alati i metodologija unutar rada

Cilj ovog rada je prikazati načine i prakse koje se koriste prilikom testiranja aplikacija unutar Python programskog jezika. Stoga je jasno kako će se među alatima koji se koriste naći Python programski jezik i brojne biblioteke koje se koriste za automatizaciju testiranja. Ideja izrade praktičnog dijela aplikacije je bila da se umjesto na nekim generičkim primjerima koji su često lišeni neke primjenjivosti u stvarnom svijetu testiranje provodi na nekoj aplikaciji (umjesto na skupu nepovezanih funkcija kojima je jedina poanta da ih se napravi za testiranje). Kao aplikacija bit će prikazana standardna CRUD (eng. create-read-update-delete) aplikacija koja će služiti kao korisnikovo sučelje za upravljanje podacima u bazi podataka (ovakve aplikacije su najraširenija oblik komercijalnih aplikacija). S obzirom na to da je bitno da takva aplikacija ima korisničko sučelje bit će korištene i biblioteke za izradu grafičkog sučelja.

Metode koje ćemo koristiti (osim onih koje se podrazumijevaju kao koncepti objektno-orijentiranog programiranja) će biti i metode nekih razvojnih principa kao što su *Test Driven Development* i *Behavior Driven Development*. Ove metode su nam zanimljive s aspekta teme zato što mogu pokazati kako testiranje može utjecati na sam proces pisanja koda, kakve tu tehnike postoje, koje su njihove prednosti i nedostaci, kada i gdje se koriste i sl. Također možemo vidjeti kako se ti koncepti uklapaju u sam životni ciklus razvoja programskog proizvoda te zašto je važno da testiranje obuhvaća testiranje zahtjeva, a ne samo izvedivost koda.

U nastavku ću detaljnije objasniti pojedine alate i biblioteke te njihove uloge za demonstraciju praktičnih primjera te metode koje se koriste i razloge za njihovo korištenje.

2.1. Python 3.8

Kao što je jasno iz naslova programski jezik u kojem ću izrađivati aplikaciju i testove je Python. Python je interpreterski objektno orijentirani jezik široke namjene, a osobito je popularan u znanstvenim krugovima kao i sferama AI, ML i Data Sciencea zbog širokog broja specijaliziranih biblioteka razvijanih od strane zajednice (open source projekti). Prednosti se ponajviše očituju u jednostavnoj sintaksi te širokom broju specijaliziranih biblioteka od kojih ćemo se posebno posvetiti onima koje se koriste za testiranje. Verzija koju sam koristio je Python 3.8 koja je jedna od novijih, ali ne posljednja (u trenutku pisanja Python 3.9), a razlog tome je što poneke biblioteke nisu ažurirane na najnoviju verziju jezika te su manje kompatibilne.

Kada radimo projekt koji uključuje više source fileova, a pogotovo kada su neki od tih source fileova testne datoteke korisno nam je koristiti neki napredniji IDE (eng. *integrated development environment*). Za izradu ovog rada sam koristio PyCharm Community Edition od JetBrainsa koji osim što je besplatan (u javnoj verziji gdje su neke značajke ograničene) nudi jako puno mogućnosti poput jednostavnog dodavanja vanjskih biblioteka, automatsko uređivanje koda, naprednog *debuggera* itd.

2.2. Pregled korištenih biblioteka

Python, kao i mnogi drugi programski jezici ima arhitekturu da se sastoji od jezgre programskog jezika koja sadrži osnovne klase i metode te proširenja koja se nazivaju modulima. U Pythonu ne postoji distinkcija u korištenju modula, paketa i biblioteka već se svodi sve na pojam modula. Kako pojmovi nisu striktno definirani, a uvriježeno je da se skup klasa i metoda koje obavljaju istu zadaću nazivaju bibliotekama (eng. *library*) taj izraz ću koristiti u nastavku umjesto izraza modul koji se ipak više odnosi na podskup biblioteke koji se bavi jednom zadaćom. Prilikom dodavanja novih biblioteka u programski kod potrebno je pripaziti postoji li ta biblioteka unutar skupa osnovnih Python biblioteka ili ju je potrebno uvesti s vanjskog resursa (korištenjem Python instalacijskog alata pip kojeg je moguće koristiti putem grafičkog sučelja u IDE).

Biblioteke koje sam koristio prilikom izrade je moguće ugrubo podijeliti unutar tri kategorije: biblioteke korištene za grafičko sučelje, biblioteke za testiranje i ostale pomoćne biblioteke (npr. za rad s bazom podataka ili operacijskim sustavom).

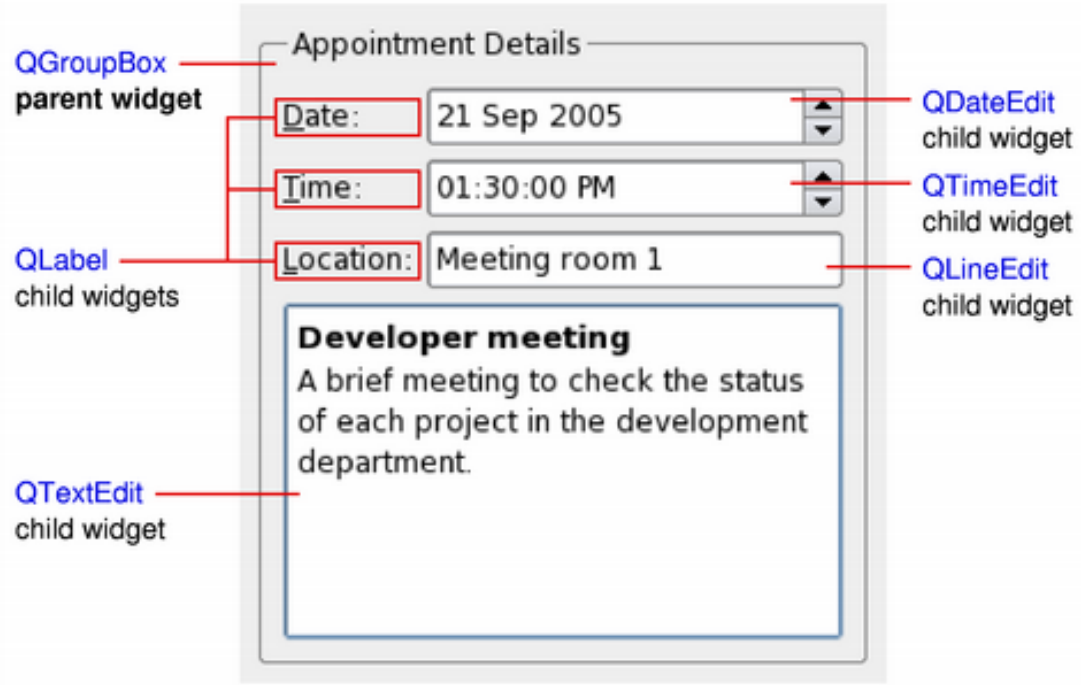
2.3. Biblioteke korištene za izradu grafičkog sučelja

Kada radimo desktop aplikacije s grafičkim sučeljem u Pythonu obično izbor se svodi na dvije biblioteke. Prva je Tkinter i dio je osnovnog Python skupa biblioteka, a druga je Qt (postoje razne inačice Qt4, Qt5). Za potrebe izrade ovog rada izabrao sam PyQt5 biblioteku jer smatram da nudi jako velik broj mogućnosti i elemenata grafičkog sučelja, cross-platform podršku, jednostavnost korištenja te podršku za testiranje.

PyQt5 je skup uvezanih funkcija razvijenih u izvornoj biblioteci Qt koja je nastala u C++. Qt je orginalno skup biblioteka visoke razine koje tvore API za razvoj modernih UI rješenja.[3]

S obzirom na to da je riječ o jednoj stvarno opsežnoj biblioteci krenut ću s analizom njenih dijelova pristupom odozgo prema dolje. Prvi dio biblioteke, a ujedno nama i najvažniji, je QtWidgets iz čijeg ćemo skupa elemenata generirati većinu sučelja. Osnovni dio tog podskupa je klasa QApplication. QApplication je osnovna klasa koja kontrolira izvođenje GUI aplikacije, definira kontrolni tok te postavke, a za svaku GUI aplikaciju postoji samo jedna QApplication instanca klase (Singleton princip).

Nadalje, unutar QtWidgets ima veću količinu klasa koje ima samoobjašnjivo značenja, a definiraju neke od osnovnih elemenata korisničkog sučelja. Takve klase su QMainWindow (klasa koja definira prozor prikazan nakon pokretanja), QPushButton (gumb objekt na čiji se pritisak može dodati slušač kako bi se pozivale druge funkcije), QLabel (tekstualna oznaka), QFrame (okvir unutar kojeg se nalaze drugi elementi), QMessageBox (skočna poruka prema korisniku), QComboBox (element koji služi za odabir jednog elementa iz padajućeg izbornika), QTextEdit (za pisanje veće količine teksta), QGroupBox (za grupiranje više elemenata) i sl.[3]



Slika 3: Prikaz osnovnog PyQt5 GUI-ja (Izvor: RiverbankComputing, 2021)

Od ostalih dijelova QtWidgets važno je izdvojiti QTableView s obzirom na to da nam je prikladan za rad s bazom podataka (za prikaz podataka iz baze te za manipulaciju s podacima zbog skupa metoda definiranih nad recima i stupcima tablice). Naposljetku, važni su nam i layouti. Layouti definiraju način na koji pozicioniramo elemente unutar sučelja. Na primjer, unutar vertikalnog layouta elementi se slažu vertikalno jedan ispod drugog, a unutar grid layouta se elementi preslaguju unutar 2D koordinatne mreže (za svaki element se zadaje pozicija s dva argumenta, x i y). [5]

Unutar PyQt5 postoji skup klasa QtTest koje služe za jednostavnije testiranje grafičkog sučelja. Pruža unit testing mogućnosti zajedno s osnovnim akcijama koje nam služe kako bismo prilikom testiranja mogli simulirati korisnikove akcije (npr. pritisak na dugme). Osim samostalno ovu klasu nam može biti korisno iskoristiti zajedno s nekom drugom klasom za testiranje kako bismo mogli simulirati GUI akcije.

2.4. Biblioteke korištene za testiranje

Osim prethodno spomenutog QtTest-a Python ima brojne dostupne biblioteke za različite vrste testiranja koju ću sada približiti, a vidjet ćemo kako se koriste u praktičnom dijelu. Svaki način testiranja i metoda za sobom povlače drugu testnu biblioteku. Python kao programski jezik razlikuje datoteke s programskim kodom i testne datoteke. Kod testnih datoteka se vrši analiza imena funkcija te one funkcije koje su detektirane kao testne se mogu pokretati zasebno. Korištenjem naprednijih IDE-a možemo iz grafičkog sučelja pokretati testove te dobiti informacije o njihovim izvršavanjima. Treba napomenuti kako ne postoji za svaku biblioteku prepoznavanje od strane IDE-a da je testna te se tom problemu lako može doskočiti drukčijim

pokretanjem izvršne datoteke što će biti prikazano u praktičnom dijelu.

2.4.1. Unittest biblioteka

Unittest je Python testni okvir (eng. framework) koji se koristi za automatizirano jedinično testiranje, a dio je osnovnog skupa Python biblioteka. Sastoji se od četiri osnovna dijela: *test fixture* (radnje koje izvršavamo prije pokretanja testova, primjerice instanciranje klase), *test case* (jedan zaseban test), *test suite* (kolekcija raznih testova koji se pokreću zajedno) i *test runner* (kontrolira pokretanje testova i daje povratnu informaciju korisniku). Sve testne metode se nalaze unutar klase koja nasljeđuje od natklase unittest.TestCase. [6]

Najveća prednost ovog okvira je lakoća i brzina korištenja. Na vrlo jednostavan način se kreiraju testne klase i metode te se poziva putem naredbenog retka:

python -m unittest tests/testsomething.py

Od metoda sadržanih unutar okvira za rad će nam posebno biti važni asssertEqual(izraz1, izraz2) koji provjerava ekvivalenciju dvaju izraza, assertTrue/assertFalse(izraz) koji provjerava nezadovoljivost nekog izraza, setUp() koji priprema podatke prije izvršavanja testova, run() koji pokreće izvršavanje testova i addTest() koji dodaje test unutar test suitea.

Osim glavnog dijela biblioteke unittest korišten će biti i modul unittest.mock. Oponašanje testova nam omogućavaju da testiramo neke dijelove koda koji ovise o dijelovima koji još nisu implementirani. Te dijelove koji nisu implementirani zamijenimo s pretpostavkom ponašanja koju definiramo s *mock* klasom. U modulu unittest.mock imamo dvije glavne klase: Mock i MagicMock. Mock je osnovna klasa koja služi za izbjegavanje testnih "dvojnika" (klasa kreiranih samo za zadovoljavanje testa) te kreiranje i pristupanje različitim atributima klase. MagicMock klasa je potklasa Mock klase koja dolazi s predefiniranim metodama koje se koriste za simuliranje testiranja (tzv. magic methods).[7]

2.4.2. PyTest biblioteka

PyTest je okvir slične namjene kao i unittest uz razliku da ga se mora uvesti korištenjem pipa. Od unittest modula ga razlikuje velika količina funkcionalnosti koji mu daju širi spektar primjenjivosti (ne koristi se samo za jedinično testiranje). Slično kao i unittest koristi *fixture* za pripremu podataka prije testiranja. Također testovi se mogu parametrizirati što znači da im se preko dekoratora proslijedi skup vrijednosti za parametre koje prima te će se test izvrtjeti u petlji koja prolazi iterativno kroz vrijednosti parametara.

Arhitektura testova unutar PyTesta se dijeli na četiri koraka: *arrange* (priprema podataka), *act* (provođenje radnji iz testa), *assert* (provjera zadovoljivosti testa) i *cleanup* (čišćenje nakon testa kako ne bi utjecao na druge testove). [8]

Pokretanje iz terminala se također kao i kod prethodne biblioteka događa na jednostavan način.

testsample1.py F.

Unutar naredbe prvi argument je ime datoteke s testovima, drugi F označava da se očekuje fail nad prvim testom, a točka kao treći argument je da se očekuje success nad drugim testom.

2.4.3. Doctest biblioteka

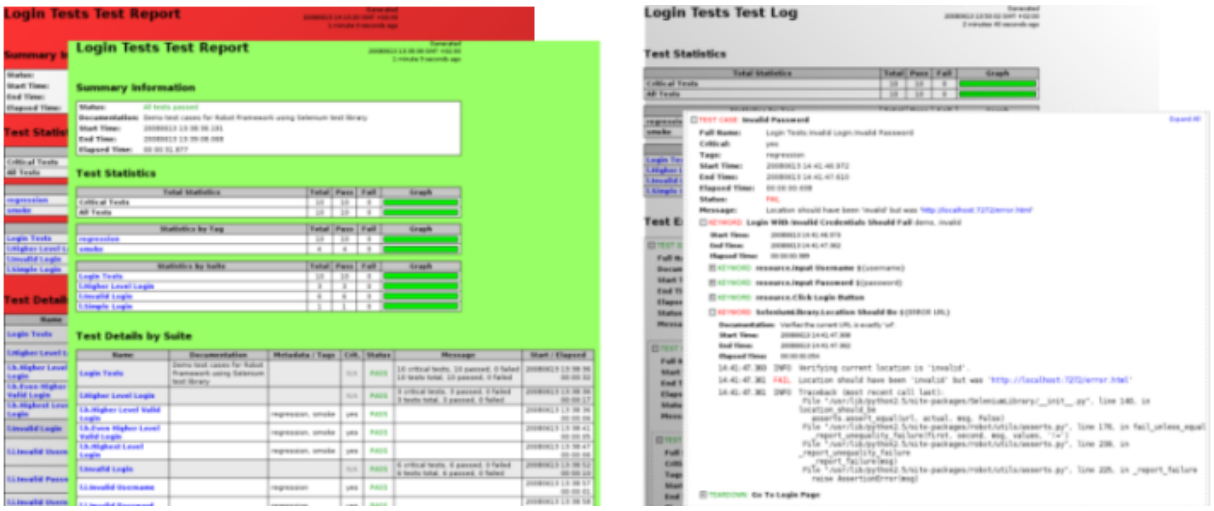
Doctest je modul koji traži dijelove tekstova koji izgledaju kao izvršavanje pokretanja Python skripte (u Python ljusci npr.) te ih izvršava te provjerava rade li kako je zapisano u tekstu. Razlozi za korištenje doctestova su provjeravanje je li dokumentacija ažurirana i rade li se svi testovi kako su zamišljeni, za provođenje regresijskog testiranja i kako bi napisali što precizniju i detaljniju dokumentaciju (zajedno s primjerima izvođenja, očekivanim ulazima i izlazima).[9]

Testovi očekivanog ponašanja se pišu u obliku komentara, a mogu se nalaziti unutar same funkcije ili iznad njene definicije. Osim provjere točnog izlaza funkcije možemo i testirati podiže li se ispravno greška prilikom testiranja i sl.

2.4.4. Robot Framework okvir za testiranje

Robot Framework je okvir za automatizirano testiranje prihvatljivosti, testiranje prihvatljivosti unutar *test driven* koncepta i robotsku procesnu automatizaciju. Kod Robot Frameworka je važno naglasiti da je *keyword driven* što mu omogućava visoku razinu apstrakcije (i bez znanja o samom procesu pisanja automatiziranog testiranja moguće je napisati test). Također omogućen je i informativan izvještaj nakon samog testiranja što je jako korisno u domeni testova prihvatljivosti. [10]

U prethodnom odlomku je spomenut dosad nepoznat pojam, a to je robotska procesna automazicija (RPA). RPA je tehnologija koja se bavi principima na koji se može jednostavno razviti software-ski robot koji će simulirati ponašanje čovjeka u interakciji s nekim drugim računalnim sustavom. [11] Takav oblik ponašanja nam služi u testovima prihvatljivosti kako bi smo simulirali korisničko testiranje svih pojedinih dijelova aplikacije.



Slika 4: Prikaz funkcionalnosti Robot Frameworka (Izvor: RobotFrameworkFoundation, 2021)

2.4.5. Gherkin i Behave biblioteke za ponašanjem pogonjeno testiranje

Unutar ovog odjeljka ću se prvi put dotaknuti biblioteka koje služe za testiranje ponašanja. Taj koncept se naziva *Behavior testing* (o čemu ćemo detaljnije u sljedećoj sekciji), a karakterizira ga visoka razina na kojoj se pišu testovi (testovi se pišu ljudima čitljivim jezikom).

Kako bismo mogli pričati o behave biblioteci u kojoj ćemo pisati testove prvo trebamo spomenuti Gherkin. Gherkin je posebna vrsta zapisa (takozvani *feature file*) koji sadrži scenarije za testiranje zapisane čitkim jezikom (za ljude bez tehničkog predznanja) te sadrže ključne riječi prema kojima se pišu testovi.

Feature je prva ključna riječ koja se nalazi u dokumentu. Označava opis komponente koja se testira. Ne služi kao oznaka za testiranje, ali služi korisniku unutar izvještaja o testiranju. Ključna riječ govori o kojem poslovnom pravilu se radi prilikom testiranja.

Najvažniji dio Gherkin *feature filea* su koraci. Koraci su one ključne riječi koje ćemo koristiti kako bismo odredili koji test testira koji dio funkcionalnosti. Oni se izvode slijedno te se zadaju ključnim riječima Given, When, Then, And, i But. Greška prilikom bilo kojeg koraka uzrokuje da scenarij testiranja pada te se ostali koraci ne izvode nakon što jedan korak padne. [12]

Behave biblioteka je testna biblioteka za implementaciju *behaviour driven developmenta* u Python načinu. Ima slične funkcionalnosti kao i ostale testne biblioteke, ali najveća razlika je što je prilagođena za čitanje oznaka iz datoteke te za automatsko traženje *feature fileova* (relativna putanja od testne datoteke do datoteke je features/steps).

```
Feature: Get greeting

  As a consumer of the greetings resource
  I should be able to get a greeting

  Scenario Outline: Get greeting using appropriate caller
    Given I use the caller <caller>
    When I request a greeting
    Then I should get a response with HTTP status code <status>
    And The response should contain the message <message>
    Examples:
      | caller | status | message           |
      | Duke   | 200    | Hello World, Duke |
      | Tux    | 200    | Hello World, Tux  |

  Scenario: Get greeting using caller 0xCAFEBABE
    Given I use the caller 0xCAFEBABE
    When I request a greeting
    Then I should get a response with HTTP status code 418
```

Slika 5: Prikaz osnovnog PyQt5 GUI-ja (Izvor: Moelholm, 2016)

2.5. Ostale korištene biblioteke

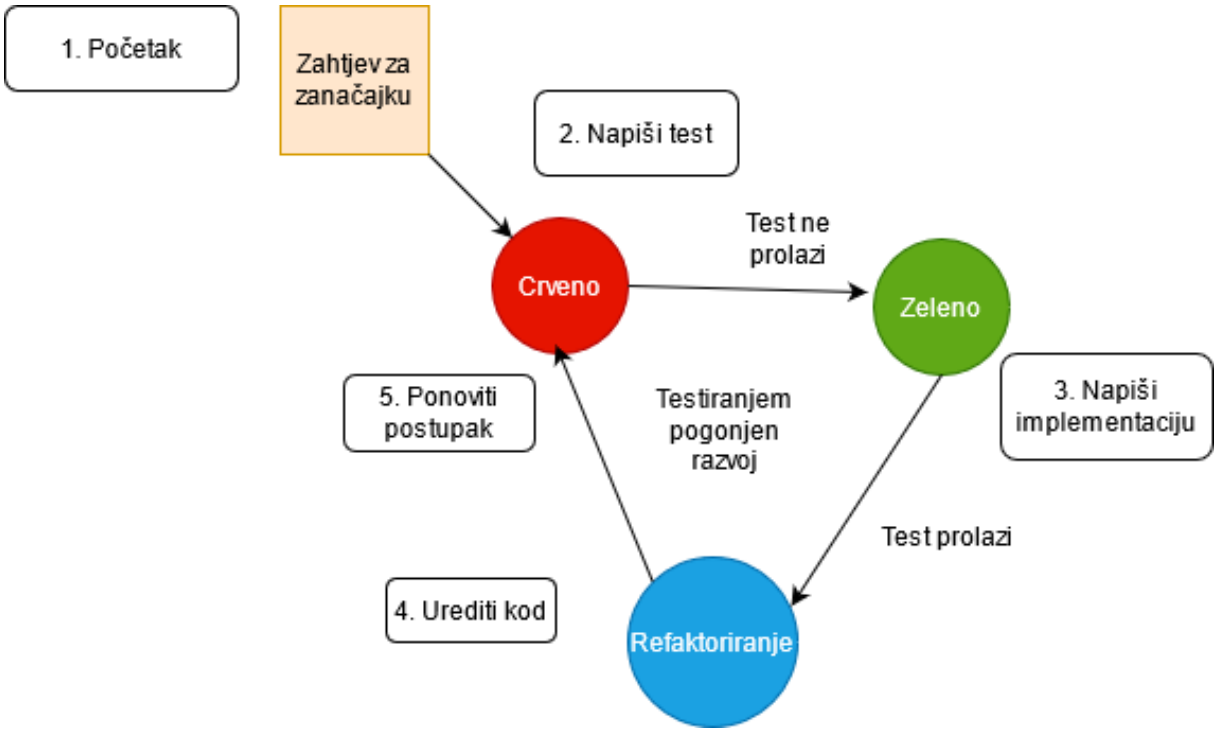
Od pomoćnih biblioteka koje su korištene najvažnije su bile sys, matplotlib, time i sqlite3. Sys je potreban za pokretanje i čitanja argumenata okoline prilikom pokretanja. Matplotlib je popularna biblioteka za grafičko prikazivanje podataka te se koristi za iscrtavanje grafova na ekranu analitike. Time je korišten kako bi se simulirao sustavsko spavanje na n sekundi u svrhu testiranja (za neke promjene treba pričekati sekundu da se osvježe). Sqlite3 je korišten kao jednostavan sustav za upravljanje s bazom podataka unutar samog programa. Korištenjem sqlite3 se izbjegava klijent-server komunikacija (sve je na klijentu) te je zato pogodan za manje količine podataka, a nudi jednostavno sučelje za rad s bazom podataka.

2.6. Testiranjem pogonjen razvoj

Prvi razvojni koncept koji će biti korišten i prikazan je testiranjem pogonjen razvoj (eng. *test driven development*). *Test driven development* kao koncept je nastao u svrhu pisanja čišćeg koda s manje pogrešaka. Najvažnija dva postulata ovog principa razvoja su da se piše kod u svrhu prolaska testova koji su prije pripremljeni te da se ne piše redundantan kod (onaj koji ne utječe na prolaznost testova). [14]

Iz osnovna dva pravila razvija se skup ponašanja koja su tipična za ovaj tip razvoja. Prvo ponašanje je da developer mora pisati sam svoje testove. Ovo ponašanje proizlazi iz toga što bi previše usporavalo razvoj da developer svakodnevno čeka da QA analitičar i tester napišu testove po kojima će on pisati kod (također brzina razvoja je i najveće zamjerke TDD pristupa). Drugo ponašanje je da moramo razvijati na prirodan način (logičkim slijedom) te da nam kod koji pokrećemo mora davati povratnu informaciju između naših promjena. Ovakav pristup nam omogućava da u svakom trenutku prepoznamo željeno i neželjeno ponašanje našeg programskog koda. Iduće ponašanje nam govori da moramo prilagoditi našu razvojnu okolinu tako da nam daje jasne i brze odgovore na najmanje promjene. To nam pravilo omogućava da uočimo sve promjene kako se ne bi dogodilo da nema "nebitna" stavka kasnije utječe na prolaznost testova. Zadnje ponašanje se bavi dizajniranjem samog rješenja, a govori kako bi se rješenje trebalo sastojati od više slabije povezanih komponenti kako bi lakše testirali. [14]

Proces TDD-a se obično provodi u tri faze. Prva faza (još nazivana i crvena faza) je faza pisanja testa koji ne zadovoljava prolaznost (najvjerojatnije se neće moći ni izvršiti jer komponente nad kojima je test pisan ne postoje). Druga faza je zelena faza koja jednostavno označava proces pisanja koda koji će učiniti to da prijašnji test koji ne prolazi sada prođe. Zadnja faza je eliminacijska faza unutar koje eliminiramo sav kod koji nam nije potreban da test prođe. [15] Često se kao faze spominju i početna (koja se označava kao faza jedan i kreće nakon dobivanja zahtjeva) i faza ponovnog iteriranja, ali kako te faze zapravo samo služe da su pozovu tri glavne faze u kojima se događa razvoj i testiranje možda je manje konfuzno kada bismo koristili samo tri faze.



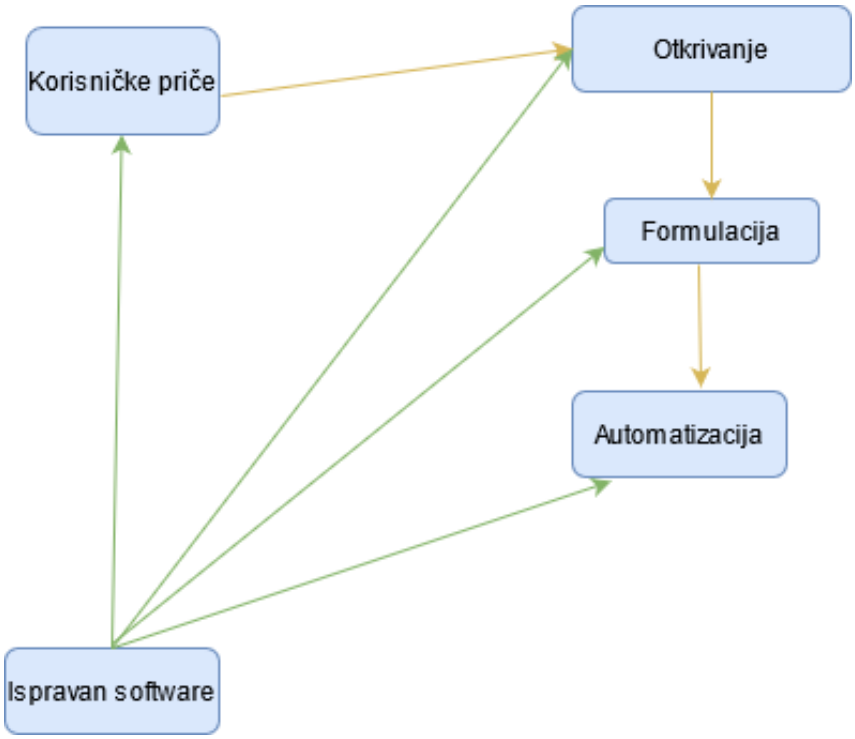
Slika 6: Osnovno načelo TDD-a (Prema: Steinfeld, 2020)

2.7. Ponašanjem pogonjeno testiranje

Pojam ponašanjem pogonjenog testiranja (eng. *behaviour driven development*) je spomenut prilikom upoznavanja s bibliotekom behave koja služi za testiranje ponašanja. BDD možemo više shvatiti kao princip razvoja programskog proizvoda nego kao metodu za implementaciju (TDD se odnosi na postupak pisanja koda i služi isključivo developeru). Kao što ime govori kod BDD najvažnija značajka su ponašanja koja su definirana korisničkim zahtjevima.

Osnovna ideja BDD-a je da premosti razliku u tehničkim i poslovnim znanjima između developera i poslovnih konzultanata te da služi kao svojevrsni mediator u njihovoj komunikaciji. Koncepti na koji se ta ideja temelji su poticanje suradnje između različitih uloga unutar tvrtke kako bi se steklo zajedničko poznavanje problematike, korištenje iterativnog razvoja s konstantnim povratnim informacijama, stvaranje programske dokumentacije koja se odmah provjerava s ponašanjem sustava. [16]

Primjenjivanje načela BDD-a se odvija na sljedeći način: uzme se manja promjena sustava koja je definirana sa strane korisnika (korisnička priča ili slučaj korištenja), dogovora se način na koji će se izvesti ta promjena, dokumentiraju se promjene na način na koji mogu automatizirane i na kraju se implementiraju dogovorene promjene tako da se prvo napišu testovi koji predstavljaju ponašanje sustava, a zatim se implementira i testira rješenje. [16] Iz zadnjeg koraka možemo vidjeti da BDD ne isključuje TDD, štoviše ga i potiče. BDD se može naslanjati na TDD u završnom procesu implementaciju dok u ostalim koracima je samostalan.



Slika 7: Osnovno načelo BDD-a (Prema: cucumber.io, 2021)

3. Testiranje programskih proizvoda

Testiranje softwarea je jako širok pojam jer obuhvaća sve od testiranje od strane korisnika koji prijavljuju nedostatke do testiranja u nekim posebnim domenama (npr. penetracijsko testiranje kod testiranja sigurnosti). Zato kada definiramo pojam testiranja često ga kategoriziramo po više kriterija. Kriteriji mogu biti po načinu na koji se testiranje izvodi, veličini programa koju testiramo, u kojoj fazi izvodimo testiranje, koju domenu testiramo itd.

Iako testiranje ima brojne kategorije po više kriterija sama svrha svakog od njih ostaje ista. Naime, kod svakog testiranja je poanta da se testira izvršava li se program prema dogovorenim specifikacijama (iako osoba koja koristi program možda i nije svjesna da ga testira). Zbog toga različiti načini testiranja se međusobno ne isključuju već nadopunjavaju.

3.1. Testiranja prema razini pristupa programskom kodu

Prva podjela načina testiranja je prema količini uvida koju tester ima u programski kod. Dijeli se na crnu kutiju (bez uvida), bijelu kutiju (s uvidom) i sivu kutiju (hibridni model). Razlog zbog čega se radi podjela prema ovom kriteriju nije samo zbog različitih metoda testiranja (npr. u crnoj kutiji nema mogućnosti statičke analize programskog koda) već zbog različitih domena koje se testiraju.

3.1.1. Testiranje metodom bijele kutije

Bijela kutija (eng. *white box*) testiranje je, kao što je prije spomenuto, tehnika testiranja pri kojoj tester ima potpuni pristup programskom kodu. Naravno, zbog toga ovakav način testiranja zahtjeva testera s visokom razinom tehničkog znanja. Svrha ovakvog načina testiranja je u pronalasku grešaka unutar implementacije, a tester to radi tako da provjerava zasebno svaki dio koda i promatra ponaša li se na zadovoljavajuć način.

Tehnike *white box* testiranja uključuju *basis path testing*. Ovakav način testiranja osigurava da se svaki dio programskog koda izvodi barem jednom (ne postoje dijelovi koda do kojih je nemoguće doći prilikom izvršavanja) kao i da se izvode u traženom redoslijedu. Iduća tehnika je *loop testing*. Unutar ove tehnike je bitno provjeriti konstrukciju petlje, tj. način na koji je ona definirana (zadovoljava li logički zahtjeve). *Control structure testing* je tehnika koja se koristi prilikom testiranja pokrivenosti svih opcija unutar koda. Testira se postoji li grananje za sve uvjete te postoji li ulazak i izlaz iz dijela koda za sve moguće uvjete. [17]

Prednosti ovakvog načina testiranja su da se izvršava prije predaje proizvoda klijentu, da se detaljno testira, da se testira svaki pojedini dio aplikacije, moguće je uočavanje dodatnih manjkavosti i neefikasnosti koda. Nedostatci su da traje jako dugo, zahtjeva velik broj testera koji su upoznati sa sustavom i alatima koji se koriste.

3.1.2. Testiranje metodom crne kutije

Black box testiranje ima dijametralno suprotnu premisu od white box, a to je da tester ne smije imati uvid u programski kod aplikacije. Tester iako nema znanja o programskom kodu mora poznavati arhitekturu aplikacije i korisničke zahtjeve.

Tehnike koje se koriste pri ovom pristupu testiranju su analiza rubnih vrijednosti. To je analiza vrijednosti kod kojih možemo očekivati nekakvo drugačije ponašanje (na primjer želimo utvrditi je li provjera ima li osoba 18 godina obuhvatila 18 kao validnu vrijednosti ili je krivo postavljena granica koja strogo gleda da je vrijednost veća od 18). *Cause effect graph* je tehnika koja na grafu prikazuje uzroke i posljedice nekog ponašanja (može sadržavati osnovne logičke operatore) te se pregledava za svaki uzrok (neka akcija koju se testira) koja će biti posljedica (koja akcija će biti izvedena ili vrijednost vraćena). *State transition testing* je koncept koji se koristi da se testiraju prijelazi iz definiranih stanja (stanja se mogu definirati UML dijagramom stanja). Primjer nekih prijelaza stanja može biti aplikacija za bankomate koja ima jasna stanja (stavi karticu - unesi pin - provjeri pin - isplati novac - ispiši potvrdu) te je bitno testovima obuhvatiti promjene tih stanja. *All pair testing* je tehnika koja se koristi da se ispita kako se aplikacija ponaša kada joj se da određena kombinacija ulaza. Ovakav pristup treba obuhvatiti sve kombinacije ulaza. [18]

Prednosti ovakvog načina testiranja su jednostavnije i brže testiranje, manje potrebno znanje i daje vanjsku perspektivu na sustav. Nedostatci su netemeljitost, nemogućnost testiranja svih mogućih grana u kodu.

3.1.3. Testiranje metodom sive kutije

Testiranje metodom sive kutije se odvija kada tester testira iz uloge korisnika (kao i kod crne kutije), ali ima djelomičan uvid u programski kod. Taj uvid je djelomičan u smislu toga da tester zna na koji je način nešto implementirano (programska logika, koji algoritam je korišten i sl.), ali ne vidi izvorni kod. Zbog većeg uvida u samu implementaciju tester ima više informacija s kojima može raditi od crne kutije testiranja te je stoga testiranje lakše i preciznije.

Neke od tehnika koje se koriste su regresijsko testiranje. Regresijsko testiranje se događa nakon što se dodala neka nova funkcionalnost programa ili se ispravljala prethodna pogreška. Svrha je utvrđivanje je li nova funkcionalnost utjecala na prethodne testove. Možemo birati hoćemo li pokretati samo testove koje smatramo da su se mogli promijeniti (to nam koristi parcijalni uvid u kod) ili ćemo jednostavno pokrenuti sve testove. Testiranje uzoraka je iduća metoda koja se oslanja na dokumentiranju prethodnih problema unutar programa te se provodi daljnja analiza zašto se tu događa pogreška. Također djelomično se ulazi u područje programskog koda te je stoga važno da postoji parcijalan uvid. [18]

Prednosti ovog pristupa su da uzima pozitivne stvari iz crnog i bijelog pristupa te ih spaja, ima i dalje nepristran (vanjski) pristup, ne ulazi preduboku u programski kod te zbog toga nije potrebna tolika ekspertiza, pomaže pri pisanju inteligentnih testova. Negativne stavke su da se previše oslanja na implementaciju (ako aplikacija nema razvijen sustav povratnih informacija dolazi do problema) te kao i black box ima problem što se ne može provoditi temeljita analiza

jer se nema uvida u kompletan programski kod.

Slika 8: Usporedba crne, bijele i sive kutije (Prema: WhiteHackLabs, 2019)

3.2. Vrste testiranja prema razinama

Prema razinama testiranje softwarea se dijeli na četiri grupe: *unit testing* (jedinično testiranje), *integration testing* (integracijsko), *system testing* (testiranje sustava) i *acceptance testing* (testiranje prihvatljivosti rješenja). Razine su definirane tako da svaka razina ima svoj obujam programa s kojim radi. Jedinično testiranje radi samo s pojedinačnim dijelovima koda, a testiranje sustava testira cjelokupnu arhitekturu rješenja. Jedina iznimka je testiranje prihvatljivosti i testiranje sustava. Naime oba testiranja rade na cjelokupnom sustavu, a razlika je u tome što se kod testiranja prihvatljivosti radi na finalnom rješenju, a u testiranju sustava na radnoj verziji zbog čega radimo razliku između ta dva načina.

Slika 9: Razine testiranja (Prema: professionalqa.com, 2019)

3.2.1. Jedinično testiranje

Unit testing ili jedinično testiranje je način testiranja pri kojem nam je domena rada individualna komponenta nekog sustava. Jedinično testiranje je prva razina testiranja koja se odvija prije integracijskog testiranja (komponente prvo moraju dokazano raditi kako bismo ih integrirali u sustav)[21]. Da se nadovežemo na prijašnju podjelu jedinično testiranje obično spada u domenu testiranja bijele kutije i provodi ga developer. Jedinica ili unit kod jediničnog

testiranja se obično računa kao jedna metoda unutar klase. Može se i veći dio koda smatrati jednom jedinicom (npr. dvije metode koje imaju zajedničku zadaću), ali kako bi se dobila što veća pokrivenost testovima treba uzeti što manju granulaciju.

Jedinično testiranje je važno zbog toga što je jeftin način za provođenje uspješnog integracijskog i sustavskog testiranja. Naime uspješnim jediničnim testiranjem možemo na jednostavan način rano otkloniti greške koje ako ostanu mogu se pokazati skupima u kasnijim testiranjima na višoj razini. Testiranja se uobičajeno provode automatizirano, iako je moguće i neke dijelove manualno testirati. Tehnike koje se koriste prilikom testiranja su:

- *Statement coverage* (računa se tako da se podijeli broj izvršenih statementa s ukupnim)
- *Decision coverage* (broj izvršenih ishoda kroz ukupni broj ishoda)
- *Branch coverage* (broj izvršenih grana kroz broj ukupnih grana)
- *Condition coverage* (broj izvršenih logičkih operanada kroz broj ukupnih operanada)
- *Finite state coverage* (broj testiranih prijelaza stanja kroz ukupni broj prijelaza stanja)[21]

3.2.2. Integracijsko testiranje

Nakon što je obavljeno jedinično testiranje može se prijeći na integracijsko testiranje. Spomenuto je kako jedinično testiranje radi s jedinicama koje su zadužene za jednu zadaću (jedna metoda ili više metoda koje zajedno rade jednu funkcionalnost). Integracijsko testiranje je zaduženo da riješi problem spajanja tih jedinica u programski proizvod testirajući njihovu međusobnu komunikaciju. Testira se više jedinica koje rade zajedničku zadaću (spojeni su u isti modul), a njihovo testiranje je bitno da se vidi rade li zajedno na očekivan način jer su često razvijeni od različitih programskih timova. Također moguće je utvrditi greške koje nisu uočene prilikom jediničnog testiranja (moguće zbog neke promjene što se dogodila u međuvremenu).

Postoje dvije glavne tehnike integracijskog testiranja. Prva tehnika je *big bang approach*. Pristup velikog praska je pristup prilikom kojeg se sve komponente integriraju zajedno odjednom te se testiraju kao cjelina. Ovakav pristup je vrlo jednostavan te zbog toga prigodan za male projekte koji se sastoje od nekoliko komponenti. Za sve ostale slučajeve ovakav način je neefikasan jer je teško lokalizirati grešku kad se radi s puno modula, treba čekati završetak implementacije da bi se počelo testirati, lako se preskoči neki modul i slično. [22]

Druga tehnika integracijskog testiranja je inkrementalno testiranje. Za razliku od big bang pristupa kod inkrementalnog pristupa se testira dvije ili više (ali ne sve) cjeline koje su implementirane i logičke ovise jedna o drugoj. Postoje dvije različite metode provođenja inkrementalnog integracijskog testiranja. Prva metoda je odozdo prema gore kod koje se naglasak stavlja na manje jedinice koje testiramo te se spajaju u veće jedinice koje se testiraju. Kod ovakvog pristupa je lakše pronaći grešku i može se odmah započeti s testiranjem jedinica. Zamjerke su da je potrebno puno vremena dok se kreira ukupan sustav koji se može kreirati. Druga metoda je odozgo prema dolje gdje se prvo testiraju veći moduli te se ide prema manjim jedinicama. Prednosti ovog načina su da je moguće lokalizirati greške, da se lako napravi rani

prototip, najvažniji dijelovi se mogu prvi ispraviti. Problem je nedostatno testiranje manjih cjelina i potreba za izrađivanjem *dummy* programa koji simuliraju rad nekih jedinica koje još nisu implementirane.[22]

3.2.3. Sustavsko tesiranje

System testing ili testiranje sustava je vrsta testiranja kod koje se testira cjelokupni sustav s popratnom arhitekturom. Testiraju se komponente, na koji način rade i na koji način komuniciraju jedna s drugom unutar sustava. S obzirom na to da se sustavsko testiranje događa na kraju razvoja testira se i rad aplikacije sa stajališta korisnika.

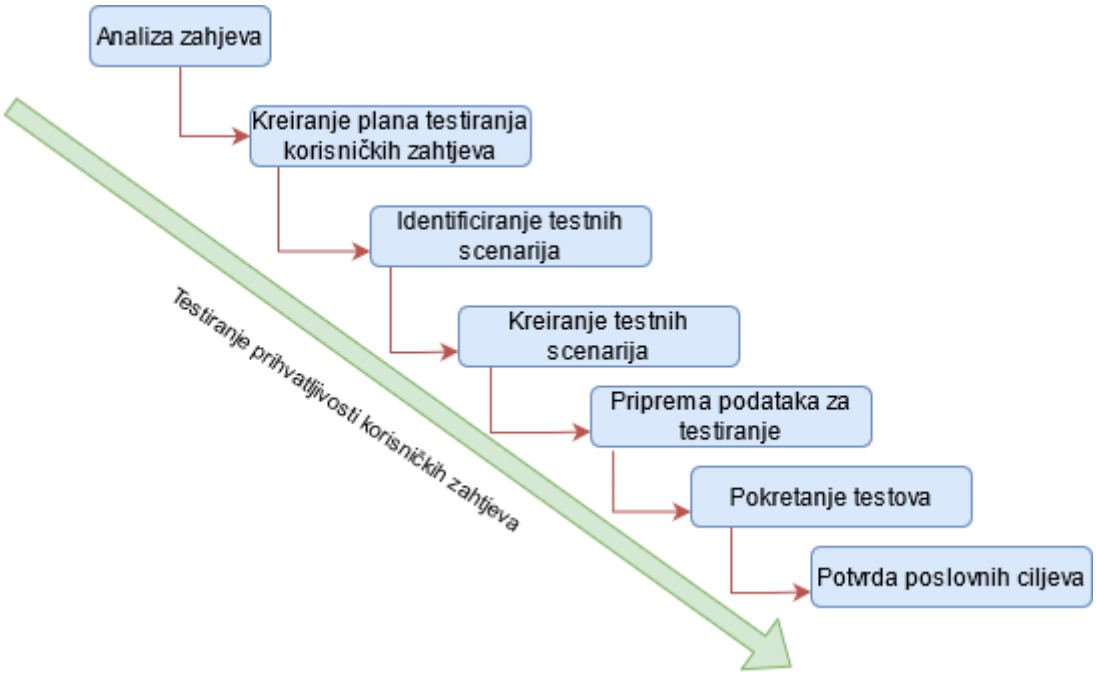
Postoji ogroman broj testiranja sustava i ne postoji neka uvriježena podjela. Naime, koji oblik testiranja će se koristiti dosta ovisi i o domeni aplikacije, načinu rada, korištenoj arhitekturi, broju korisnika koji će ju koristiti i sl. Neki od najčešćih načina testiranja sustava su:

- *Usability testing* (koliko je aplikacija prilagođena za korištenje)
- *Recovery testing* (koliko efikasno se barata s padovima i greškama u sustavu te koliko je brz oporavak)
- *Migration testing* (koliko se efikasno sustav može prebaciti s jedne arhitekture na drugu)
- *Functional testing* (testiranje nedostaje li neka od funkcionalnosti i treba li ju dodati u sustav)
- *Load testing* (kako se sustav ponaša pred opterećenjem određenog broja korisnika)[23]

3.2.4. Testiranje prihvatljivosti

Acceptance test ili test prihvatljivosti je test kojeg izvodi sam korisnik aplikacije kako bi validirao sustav prije nego se krene u produkciju. Radi se poslije testiranja sustava. Glavni fokus testa prihvatljivosti je u zadovoljivosti poslovnih pravila koja su trebala biti implementirana. Potreba za još jednim testiranjem nakon što su odrađena sva prethodna testiranja na svim razinama je u tome da je moguće da je razvojni tim ima drugačije zamisli od samog klijenta te da je potrebno usuglasiti njihove zahtjeve.

Proces testiranja se izvodi prema definiranim koracima kako bi se dala struktura testiranju korisnika. Prvi korak je analiza poslovnih zahtjeva kako bi se znalo što se očekuje od sustava. Drugi korak je kreiranje plana testiranja, testnih scenarija i pristupa testnim slučajevima. Idući korak je kreiranje testnih slučajeva s jasnim testnim koracima. Zatim se pripremaju podaci za testiranje (potrebno je biti upoznat s bazom podataka). Pokreću se testovi te se zapisuju rezultati. Ako test ne prolazi prijavljuje se greška te se ponovno pokreću testovi nakon ispravka. Nakon svega se potvrđuje da su zadovoljeni kriteriji poslovnog testiranja te se ide u produkciju. [24]



Slika 10: Koraci testiranja prihvatljivosti (Prema: softwaretestinghelp.com, 2021)

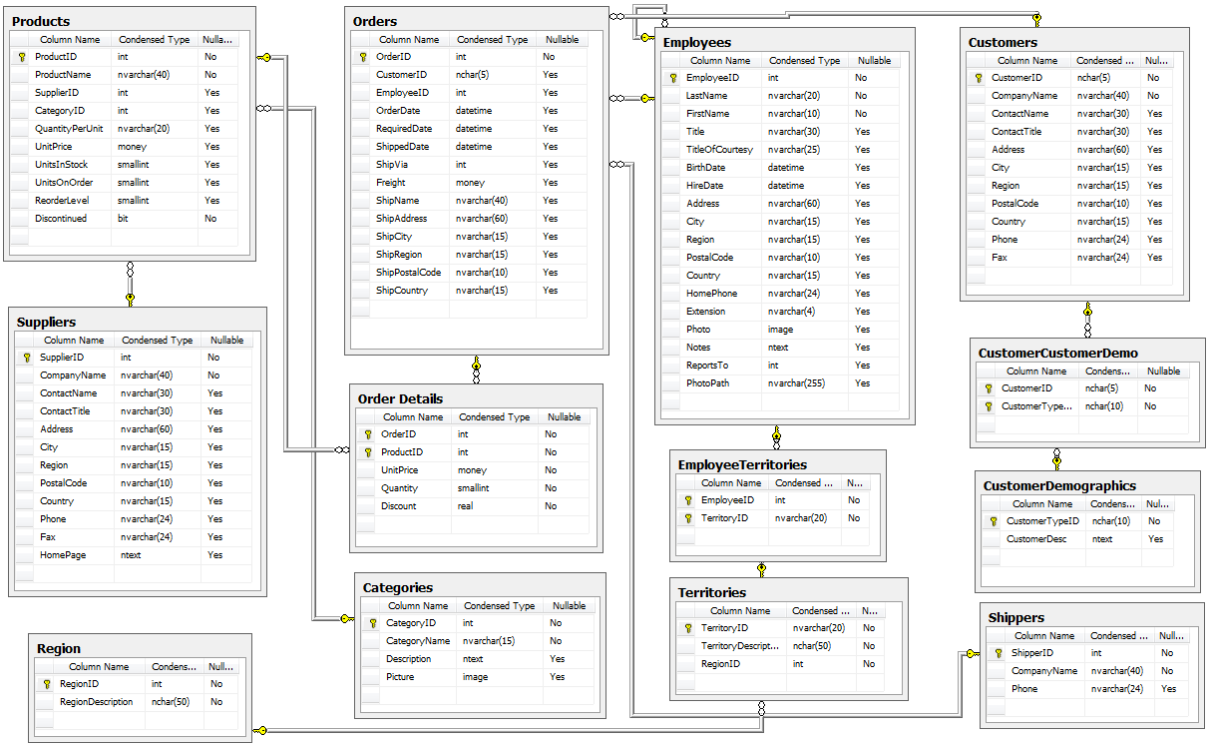
4. Praktični prikaz testiranja u Pythonu

Prije definirani teorijski pristupi testiranju, njihove podjele i alati koji su navedeni će služiti za praktično rješavanje problema testiranja prilikom razvoja aplikacije u Pythonu. Osnovna ideja praktičnog dijela je na cjelovitom primjeru aplikacije prikazati tehnike i metode testiranja. Za potrebe testiranja bit će razvijena Python desktop aplikacija u Qt okviru koja će u pozadini imati bazu podataka, a prema korisniku (front end) će imati sučelje koje će dozvoljavati manipulaciju s podacima u bazi podataka kao i grafički prikaz važnijih podataka iz baze podataka. Baza podataka sadrži podatke o poslovanju trgovine koja ima svoje dobavljače, kupce, zaposlenike itd. Takvi podaci nam koriste jer su lako razumljivi (nije potrebno poznavati tu poslovnu domenu), postoje brojni brojčani prikazi pa se mogu raditi kalkulacije, usporedbe između više poslovnih entiteta i sl.

Testiranja koja će se provodi uključivat će sve od testiranja korisnikovog rada na sučelju (npr. otvori li se neki ekran na pritisak gumba) pa do ispravnosti unosa podataka u bazi. Također testirane će biti pojedine funkcije, grupa funkcija koje zajedno rade, grupa klasa koje se spajaju u istu komponentu te će se na takav način pokriti više razina testiranja.

4.1. Podaci s kojima aplikacija radi

Kao što je spomenuto domena koju će podaci aplikacije predstavljati je trgovina koja se bavi kupoprodajom robe širokog spektra. Model baze podataka je Northwind SQL koji je razvijen kao primjer baze podataka za korištenje i učenje alata. S obzirom na to da je model baze podataka razvijen za potrebe MS Accessa, a korišten je sqlite kao sustav za upravljanje bazom podataka u praktičnoj primjeni potrebno je preuzeti prilagođenu verziju baze podataka.Izgled baze podataka se može vidjeti u sljedećem ERA modelu:



Slika 11: ERA model Northwind baze (Izvor: White, 2021)

Na prethodnom ERA modelu možemo vidjeti koje tablice i veze imamo. Tablice koje su nam najvažnije u modelu su tablice proizvoda, narudžbi, stavki narudžbi (niža granulacija od narudžbi), zaposlenika, kupaca i dobavljača. Te tablice imaju i neke pomoćne tablice kao npr. regije i teritoriji kojima ti entiteti pripadaju. Također treba obratiti pažnju na veze između tablica. Narudžbe imaju vezu 1 na više prema stavkama koje imaju vezu prema proizvodima (1 stavka može imati 1 proizvod, ali proizvod može biti na više stavki), dobavljači prema proizvodima. Ove veze su nam važne kako bismo utvrdili logiku punjenja baze te poslovnu logiku koja se koristi. Razumijevanje baze će nam biti kasnije značajno jer ćemo osim pogrešaka u zadavanju argumenata testirati i pogreške kvalitete podataka. Podaci unutar baze podataka će u inicijalnom stanju biti napunjeni sa skriptom dostupnom na https://github.com/jpwhite3/northwind-SQLite3/blob/master/Northwind.Sqlite3.create.sql.

4.2. Razvoj i testiranje glavnog izbornika

Korisnik kada započne s korištenjem aplikacije prvo se preusmjeri na glavni izbornik iz kojeg dohvaća ekrane koji se bave upravljanjem podataka o pojedinim entitetima u bazi podataka. Glavni izbornik se sastoji od skupine QPushButton-a koji imaju vezane akcije za otvaranje drugih ekrana. Kako bismo pokazali testiranje otvaranja različitih ekrana koristit ćemo se osnovnom procedurom koraka iz TDD-a. Na početku kreiramo datoteku za jediničnо testiranje unutar IDE-a te unutar glavne klase definiramo testove za provjeru navigacije s glavnog izbornika.

Prvi korak je napisati test. Želimo utvrditi koje događaje želimo testirati te koji su potencijalni ishodi tih događaja. S obzirom na to da za početak imamo jednostavan primjer funkcionalnosti možemo jednostavno odrediti scenarije:

- Scenarij 1: Gumb nije pritisnut stoga ekran koji se poziva s tim gumbom nije prikazan
- Scenarij 2: Gumb je pritisnut stoga ekran koji se poziva je prikazan

Kako bismo napisali test za prethodno spomenute scenarije potrebno je iskoristiti funkcionalnost PyQt5.QtTest biblioteke koja nam pomaže da simuliramo navigaciju (s obzirom na to da provodimo automatizirano testiranje nije nam rješenje primijeniti pristup crne kutije pritiskanja gumbova) i unittest-ovu funkciju assertEqual koja uspoređuje jesu li dva parametra jednaka. Prvi parametar će nam biti property koji ćemo definirati klasi glavnog izbornika, a koji će sadržavati boolean vrijednost je li neki ekran aktiviran. Drugi parametar će nam biti True ili False s obzirom želimo li testirati Scenarij 1 ili Scenarij 2.

```
def test_loading_entity_screens(self):
        """Test if activated flags in main menu trigger correctly"""
        QTest.mouseClick(self.MainMenu.btn_employees, Qt.LeftButton)
        QTest.mouseClick(self.MainMenu.btn_orders, Qt.LeftButton)
        QTest.mouseClick(self.MainMenu.btn_customers, Qt.LeftButton)
        QTest.mouseClick(self.MainMenu.btn_shippers, Qt.LeftButton)

        self.assertEqual(self.MainMenu.e_activated, True)
        self.assertEqual(self.MainMenu.o_activated, True)
        self.assertEqual(self.MainMenu.c_activated, True)
        self.assertEqual(self.MainMenu.sh_activated, True)
        self.assertEqual(self.MainMenu.s_activated, False)
        self.assertEqual(self.MainMenu.g_activated, False)
```

Kako bi nam testovi radili i kako ne bi imali redundantne inicijalizacije varijabli trebamo unutar setUp metode glavne klase za unittest definirati objekte klasa koje testiramo kao propertije.

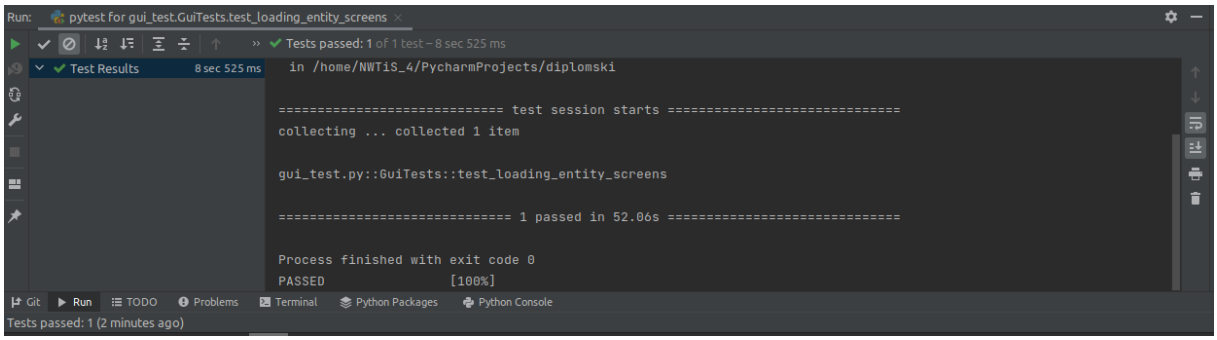
```
def setUp(self):
        self.app = QApplication(sys.argv)
        self.MainMenu = main.MainMenu()
```

Pokrenemo li sada test iz IDE-a dobivamo poruku greške od interpretera kako ne postoji modul main. Podsjetimo li se ulomka o TDD-u onda znamo da je to normalno ponašanje. Sada je potrebno stvoriti klasu Main te u njoj definirati elemente grafičkog sučelja te propertije koji će nam služiti za pozivanje drugih ekrana i sl.

```
class MainMenu(QtWidgets.QMainWindow):
    """
    Class that generates and controls main menu screen.
    """

    def __init__(self):
```

```
        super(MainMenu, self).__init__()
        self.setWindowTitle("Main menu")
        self.setGeometry(300, 300, 640, 680)
        self.e = Employees()
        self.e_activated = False

        label_choose_option = QtWidgets.QLabel(self)
        label_choose_option.resize(300, 100)
        label_choose_option.setText('Choose which entity you want to access')
        label_choose_option.move(200, 20)

        self.btn_employees = QtWidgets.QPushButton("Employees", self)
        self.btn_employees.setGeometry(200, 90, 200, 60)

        self.btn_employees.clicked.connect(self.show_employees)

    def show_employees(self):
        """
        Show employees screen is called and displayed.
        :return:
        """
        self.e_activated = True
        self.e.show()


if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = MainMenu()
    sys.exit(app.exec_())
```

U gore prikazanom kodu definirani su elementi GUI-ja koji služe za poziv klase koja će biti kreirana kao QWidget, zatim konekcija buttona na funkciju koju poziva te main poziv kako bi se pokretanjem aplikacije prikazao glavni izbornik. Također prikazano je mijenjanje vrijednosti propertija u True kada je ekran prikazan (napomena: zbog preglednosti su izbačeni dijelovi source code-a koji su analogni prikazanom, npr. za svaki ekran postoji klasa analogna klasi za prikaz zaposlenika kao i konekcija te klase na button i definiranje same pozicije buttona).



Slika 12: Prikaz rezultata testiranja

Sada kada se vratimo na test možemo ga pokrenuti te vidimo da prolazi u oba scenarija (nepozvani ekrani bit će False, a pozvani True). Zadnja faza u TDD bi bila refaktoriranje nepotrebnog koda, ali s obzirom na to da je ovo prvi primjer te je scenarij jednostavan za tim neće biti potrebe.

4.3. Testiranje dohvata podataka iz baze

Sada kada možemo pokrenuti iz glavnog izbornika ekrane koje prikazuje podatke o pojedinim entitetima trebali bi popuniti podacima te ekrane. Proces će izgledati tako da ćemo kreirati testove koristeći upite na bazu podataka (tako da vidimo koji su podaci i dohvaćamo li ih ispravno unutar programa korištenjem sqlite), zatim ćemo kreirati GUI te ćemo iz GUI pozvati funkciju iz baze podataka koja će popuniti podatke te ćemo ih nakon toga testirati.

Prvi korak je test. Trebamo odrediti što ćemo testirati. S obzirom na to da je ovo Read funkcionalnost CRUD aplikacije podatke treba testirati prenose li se 1:1 iz baze u aplikaciju. To ćemo testirati tako da vidimo prenose li se imena stupaca u aplikaciju, ispisuju li se podaci iz baze u pravom redoslijedu i rade li ispravno JOIN operacije. Join koristimo kako bismo povezane podatke zajedno prikazali i dobili informativniji prikaz. Na primjer proizvode spajamo s kategorijama proizvoda te ih zajedno prikazujemo kako bi odmah vidjeli kojoj kategoriji proizvod pripada.

Prvi korak je pisanje testa koji se neće moći prevesti jer ne postoji još objekt tablice kojeg pozivamo. Testiramo nalazi li se u prvom redu u šestom stupcu kolona "Category Name", nalazi li se ime kategorije "Cheese" u drugom redu tablice itd. Također provjerimo jednu netočnu vrijednost da vidimo da ne postoji neka greška zbog koje su svi testovi lažno pozitivni te na kraju provjeravamo je li jednak broj redaka u bazi i u aplikaciji. Trebamo pripaziti na to da je prvi redak u aplikaciji redak s imenima stupaca, a ne s podacima i da table objekt koristi 0-index (brojanje kreće od 0).

```
    def test_loading_products_screen(self):
        """
        Test if products and categories data loads to main employee
↪   screen.
        :return:
        """
        self.assertEqual(self.prods.table_widget.item(0, 5).text(),
        ↪  "Category Name")
        self.assertEqual(self.prods.table_widget.item(1, 5).text(),
        ↪  "Cheese")
        self.assertEqual(self.prods.table_widget.item(10, 7).text(),
        ↪  str(13.25))
        self.assertEqual(self.prods.table_widget.item(11, 7).text()
        ↪  == "0", False)
        self.assertEqual(self.prods.table_widget.rowCount(), 81)
```

Test naravno ne prolazi jer još nije definiran GUI od product tablice i poziv na bazu podataka. Naravno trebamo prvo GUI napraviti te definirati poziv na bazu podataka. Unutar GUI za ovaj ekran zasad ćemo imati samo tablicu s prikazom podataka jer nemamo Update i Delete funkcionalnosti zasad definirane.

```
class Products(QWidget):
    """
    Main screen for products entity. Shows list of all products and
↪   their categories. Has link to add products and
    categories screen.
    """

    def __init__(self):
        """
        Initialization of elements for products screen.
        """
        super().__init__()
        self.title = 'Product list with belonging categories'
        self.left = 0
        self.top = 0
        self.width = 800
        self.height = 1000

        self.setWindowTitle(self.title)
        self.setGeometry(self.left, self.top, self.width,
        ↪  self.height)
```

```
        self.get_table_data()

        self.layout = QVBoxLayout()
        self.layout.addWidget(self.table_widget)
        self.setLayout(self.layout)

        self.button = QPushButton('Add new product', self)
        self.layout.addWidget(self.button)

    def get_table_data(self):
        """
        Specifies positions of attributes inside of layout. Loads
↪   data from DB to rows of QWidget table and shows
        attributes on their corresponding positions.
        :return:
        """
        s1 = db_singleton.Singleton().get_instance()
        self.table_rows = s1.get_all_products_with_category_names()
        self.table_widget = QTableWidget()
        self.table_widget.horizontalHeader().setVisible(False)
        row_count = len(self.table_rows) + 1

        self.table_widget.setRowCount(row_count)

        self.table_widget.setColumnCount(12)

        self.table_widget.setItem(0, 0, QTableWidgetItem("Product
        ↪  ID"))
        self.table_widget.setItem(0, 1, QTableWidgetItem("Product
        ↪  name"))
        self.table_widget.setItem(0, 2, QTableWidgetItem("Suppliers
        ↪  ID"))
        self.table_widget.setItem(0, 3, QTableWidgetItem("Company
        ↪  Name"))
        self.table_widget.setItem(0, 4, QTableWidgetItem("Category
        ↪  ID"))
        self.table_widget.setItem(0, 5, QTableWidgetItem("Category
        ↪  Name"))
        self.table_widget.setItem(0, 6, QTableWidgetItem("Quantity
        ↪  per Unit"))
        self.table_widget.setItem(0, 7, QTableWidgetItem("Unit
        ↪  price"))
```

```
self.table_widget.setItem(0, 8, QTableWidgetItem("Units in
↪  stock"))
self.table_widget.setItem(0, 9, QTableWidgetItem("Units in
↪  order"))
self.table_widget.setItem(0, 10, QTableWidgetItem("Reorder
↪  level"))
self.table_widget.setItem(0, 11,
↪  QTableWidgetItem("Discontinued"))

counter = 1
for row in self.table_rows:
    self.table_widget.setItem(counter, 0,
    ↪  QTableWidgetItem(str(row[0])))
    self.table_widget.setItem(counter, 1,
    ↪  QTableWidgetItem(row[1]))
    self.table_widget.setItem(counter, 2,
    ↪  QTableWidgetItem(str(row[2])))
    self.table_widget.setItem(counter, 3,
    ↪  QTableWidgetItem(row[3]))
    self.table_widget.setItem(counter, 4,
    ↪  QTableWidgetItem(str(row[4])))
    self.table_widget.setItem(counter, 5,
    ↪  QTableWidgetItem(row[5]))
    self.table_widget.setItem(counter, 6,
    ↪  QTableWidgetItem(str(row[6])))
    self.table_widget.setItem(counter, 7,
    ↪  QTableWidgetItem(str(row[7])))
    self.table_widget.setItem(counter, 8,
    ↪  QTableWidgetItem(str(row[8])))
    self.table_widget.setItem(counter, 9,
    ↪  QTableWidgetItem(str(row[9])))
    self.table_widget.setItem(counter, 10,
    ↪  QTableWidgetItem(str(row[10])))
    self.table_widget.setItem(counter, 11,
    ↪  QTableWidgetItem(str(row[11])))

    counter += 1


↪  self.table_widget.horizontalHeader().setStretchLastSection(True)
self.table_widget.horizontalHeader().setSectionResizeMode(
    QHeaderView.Stretch)
```

Definirano je jednostavno sučelje u kojem zapišemo vrijednosti stupaca koje znamo da ćemo preuzeti, a ostatak koji ćemo dohvatiti iz baze podataka ćemo red po red zapisivati inkrementirajući broj retka.

Gledajući prethodni test možemo uočiti da se sastoji od više dijelova (funkcija). Testira se naime prijenos podataka iz baze i punjenje tih podataka u tablicu. Pogreška na tom putu može nastati na dva mjesta: pri preuzimanju podataka iz baze i pri punjenju tablice u klasi koja obrađuje GUI. Zato ćemo napisati jednostavan test za dohvaćanje podataka iz baze.

```
    def test_get_all_products_with_category_names(self):
        """Initial test of loading from DB. Test should be changed
        ↪  after changes are made to db. Test includes checks
        for first matched row and number of unmatched rows"""

        s1 = db_singleton.Singleton()
        result = s1.get_all_products_with_category_names()
        first_product_name = result[0][1]
        first_product_category_name = result[0][5]
        first_product_supplier_name = result[0][3]
        count_nulls_category = 0
        count_nulls_suppliers = 0
        for row in result:
            if row[2] == "null":
                count_nulls_suppliers += 1
            if row[4] == "null":
                count_nulls_category += 1

        self.assertEqual(first_product_category_name, "Meat/Poultry")
        self.assertEqual(first_product_supplier_name, "Pavlova,
        ↪  Ltd.")
        self.assertEqual(first_product_name, "Alice Mutton")
        self.assertEqual(count_nulls_category, 0)
        self.assertEqual(count_nulls_suppliers, 0)
        self.assertEqual(len(result), 40)
```

Kako bismo dohvatili podatke iz baze podataka potrebno je napraviti klasu koja će kontrolirati upite na bazu. Ta klasa će biti uzorka Singleton što nam ukratko osigurava da imamo samo jednu instancu te klasu unutar sustava. Korištenjem jednostavnog SQL upita dohvatit ćemo podatke iz baze podataka te ih vratiti u obliku liste. Kroz tu listu ćemo iterirati u GUI.

```
def get_all_products_with_category_names(self):
        """
        Select all products joined with categories to provide
↪   category name, joined with suppliers to provide supplier
```

```
        :return: List of all rows from products left joined with
↪   categories and suppliers
        """
        cur = self.con.cursor()
        fetched_rows = []
        command = "SELECT ProductID, ProductName,
         ↪  Products.SupplierID,  CompanyName, Products.CategoryID,
         ↪  CategoryName, QuantityPerUnit,
                  "UnitPrice, UnitsInStock, UnitsOnOrder,
                   ↪  ReorderLevel, Discontinued " \
                  "FROM Products LEFT JOIN \
                  Categories ON Products.CategoryID =
↪   Categories.CategoryID " \
                  "LEFT JOIN Suppliers ON
                   ↪  Products.SupplierID=Suppliers.SupplierID " \
                  "ORDER BY ProductName"

        for row in cur.execute(command):
            fetched_rows.append(row)

        return fetched_rows
```

Koristimo left join kako bi obuhvatili u rezultat sve proizvode, neovisno o tome imaju li definiranu kategoriju. Tako ćemo kao ukupan broj redaka dobiti zapravo broj redaka iz tablice Product (ukoliko bi koristili join potencijalno bi dobili manje redaka). Sada kada imamo funkciju koju vraća retke iz baze podataka možemo napuniti tablicu.

Product list with belonging categories

Product ID	Product name	Suppliers ID	Company Name	Category ID	Category Name	Quantity per Unit	Unit price	Units in stock	Units in order	Reorder level	Discontinued
18	Carnarvon Tigers	7	Pavlova, Ltd.	8	Seafood19	16 kg pkg.	5.3	5	0	0	0
1	Chai	1	Exotic Liquids	1	Beverages	10 boxes x 20 bags	18	39	0	10	0
2	Chang	1	Exotic Liquids	1	Beverages	24 - 12 oz bottles	19	17	40	25	0
39	Chartreuse verte	18	Updated company name	1	Beverages	750 cc per bottle	18	69	0	5	0
4	Chef Anton's Cajun ...	2	New Orleans Cajun ...	2	Condiments	48 - 6 oz jars	22	53	0	0	0
5	Chef Anton's Gumbo Mix	2	New Orleans Cajun ...	2	Condiments	36 boxes	21.35	0	0	0	1
48	Chocolade	22	Zaanse Snoepfabriek	3	Confections	10 pkgs.	12.75	15	70	25	0
38	Côte de Blaye	18	Updated company name	1	Beverages	12 - 75 cl bottles	263.5	17	0	15	0
58	Escargots de Bourgogne	27	Escargots Nouveaux	8	Seafood19	24 pieces	13.25	62	0	20	0
52	Filo Mix	24	G'day, Mate	5	Grains/Cereals	16 - 2 kg boxes	7	38	0	25	0
71	Flotemysost	15	Norske Meierier	4	New cat	10 - 500 g pkgs.	21.5	26	0	0	0
33	Geitost	15	Norske Meierier	4	New cat	500 g	2.5	112	0	20	0
15	Genen Shouyu	6	Mayumi's	2	Condiments	24 - 250 ml bottles	15.5	39	0	5	0
56	Gnocchi di nonna Alice	26	Pasta Buttini s.r.l.	5	Grains/Cereals	24 - 250 g pkgs.	38	21	10	30	0
31	Gorgonzola Telino	14	Formaggi Fortini s.r.l.	4	New cat	12 - 100 g pkgs	12.5	0	70	20	0
6	Grandma's Boysenberr...	3	Grandma Kelly's ...	2	Condiments	12 - 8 oz jars	25	120	0	25	0
37	Gravad lax	17	Svensk Sjöföda AB	8	Seafood19	12 - 500 g pkgs.	26	11	50	25	0
24	Guaraná Fantástica	10	Refrescos Americanas ...	1	Beverages	12 - 355 ml cans	4.5	20	0	0	1
69	Gudbrandsdalsost	15	Norske Meierier	4	New cat	10 kg pkg.	36	26	0	15	0
44	Gula Malacca	20	Leka Trading	2	Condiments	20 - 2 kg bags	19.45	27	0	15	0
26	Gumbär Gummibärchen	11	Heli Süßwaren GmbH ...	3	Confections	100 - 250 g bags	31.23	15	0	0	0
22	Gustaf's Knäckebröd	9	PB Knäckebröd AB	5	Grains/Cereals	24 - 500 g pkgs.	21	104	0	25	0
10	Ikura	4	Tokyo Traders	8	Seafood19	12 - 200 ml jars	31	31	0	0	0
36	Inlagd Sill	17	Svensk Sjöföda AB	8	Seafood19	24 - 250 g jars	19	112	0	20	0

Slika 13: Prikaz tablice proizvoda

Sada kada pokrenemo naš test koji provjera točnost podataka dobivamo da test prolazi. Sličan princip ćemo koristiti i za ostale entitete, a prikazan je entitet Product jer je zanimljiv sa strane da imamo veze između dva entiteta te da postoji provjera ispravnog spajanja. Također

na ovom testiranju možemo uočiti kako izgleda jedinično testiranje koje ne testira jednu već više funkcija koje rade zajedno (funkcija koja dohvaća podatke za tablicu iz GUI-ja i funkcija za dohvaćanje podataka iz baze).

4.4. Funkcionalnost ažuriranja i testiranje točnosti podataka

Nakon što je implementirana i testirana funkcionalnost čitanja iz CRUD-a sljedeća nam je na redu funkcionalnost ažuriranja (eng. *update*). Update će biti izveden tako da korisnik označi n redaka te unese promijenjene vrijednosti. Nakon odabira gumba za update dobiva se poruka o uspješnom/neuspješnom osvježavanju vrijednosti te o mogućim greškama.

Prvi korak nam je kao i dosad napisati test koji će nam služiti da provjerimo radi li osvježavanje na relaciji GUI-baza podataka (testira se izmjena podataka iz GUI-ja u bazi podataka) te također testiramo provjere na sučelju (postojat će i provjere ispravnosti unosa podataka te provjere unosa obaveznih polja). Za prikaz ove funkcionalnosti uzet ćemo entitet dobavljača (eng. *Suppliers*).

Za testiranje ove funkcionalnosti koristit ćemo PyTest umjesto dosadašnjeg unittest-a (i dalje koristimo QTest za simuliranje navigacije). Kako je spomenuto u poglavlju s alatima PyTest ima mogućnost tzv. fixture-a koji služe da pripremimo podatke prije samog izvršavanja testa (nešto kao setUp unutar unittest-a). S obzirom na to da će nam biti potrebni ekrani Suppliers i main, morat ćemo prije izvršavanja pripremiti instance te klase kako se ne bi prilikom svakog poziva ponovo instancirale.

```
@pytest.fixture
def main_menu():
    return main.MainMenu()
@pytest.fixture
def my_singleton():
    return db_singleton.Singleton.get_instance()
@pytest.fixture
def sups():
    return Suppliers.Suppliers()
```

Testirat ćemo uspješno i neuspješno dodavanje. Prvo ćemo testirati neuspješno dodavanje. S obzirom da nam je u bazi podataka polje "Company Name" za dobavljača označeno kao obavezno testirat ćemo hoće li se ispravno obraditi ta greška na strani aplikacije (bez da se poziva baza) te hoće li se ispisati tražena greška.

```
def test_updating_failed_suppliers(app, main_menu, sups,
↪  my_singleton):
    QTest.mouseClick(main_menu.btn_suppliers, Qt.LeftButton)
```

```
    sups.table_widget.item(1, 1).setText("")
    sups.table_widget.selectRow(1)

    prev_sups =
     ↪  my_singleton.get_supplier_by_id(sups.table_widget.item(1,
     ↪  0).text())

    QTest.mouseClick(sups.button_update, Qt.LeftButton)
    new_sups = my_singleton.get_supplier_by_id(prev_sups[0])

    assert (prev_sups[0] == new_sups[0]
            and prev_sups[1] == new_sups[1]
            and new_sups[1] != ""
            and sups.total_updated_rows == 0
            and sups.total_failed_update_rows == 1
            and sups.update_message.find("Please enter required
             ↪  fields (Supplier name)") != -1)
```

Također kao jednu karakteristiku PyTest-a smo spomenuli i mogućnost stavljanja parametara s kojima će se test pozivati. Možemo definirati n parametara koji se onda proslijede funkciji kao argument. Iteriranje kroz parametre funkcionira tako da se svaki sa svakim parametar spajaju, a ne u parovima (npr. param1["1", "2"], param2["2","3"] su četiri testa, a ne dva). To ćemo iskoristiti za uspješno ažuriranje. Testirat ćemo s više vrijednosti promijeni li se vrijednost podataka za redak koji ima traženi ID i postavili se vrijednost na vrijednost parametra koji smo proslijedili u testu.

```
@pytest.mark.parametrize("company_name", ["Import/Export",
 ↪  "Export/Import"])
@pytest.mark.parametrize("contact_name", ["John Doe", "Jane Doe"])
def test_updating_success_suppliers(app, main_menu, sups,
 ↪  my_singleton, company_name, contact_name):
    QTest.mouseClick(main_menu.btn_suppliers, Qt.LeftButton)

    sups.table_widget.item(1, 1).setText(company_name)
    sups.table_widget.item(1, 2).setText(contact_name)
    sups.table_widget.selectRow(1)

    prev_sups =
     ↪  my_singleton.get_supplier_by_id(sups.table_widget.item(1,
     ↪  0).text())

    QTest.mouseClick(sups.button_update, Qt.LeftButton)
    new_sups = my_singleton.get_supplier_by_id(prev_sups[0])
```

```
    assert (prev_sups[0] == new_sups[0]
            and (prev_sups[1] != new_sups[1]
            or prev_sups[2] != new_sups[2])
            and new_sups[1] == company_name
            and new_sups[2] == contact_name
            and sups.total_updated_rows == 1)
```

Dok su nam ostale testne funkcije za provjeru da se ispravan zapis zapisuje analogne jedna drugoj (jedina razlika su koja polja su obavezna za unos), funkcije koje provjeravaju neispravan unos se donekle razlikuju za različite entitete. Na primjer za entitete proizvoda imamo provjeru tipa podataka. Poslovna logika nalaže da je cijena decimalna vrijednost, broj proizvoda u skladištu cijeli broj i sl. Kako koristimo sqlite potrebno je da sami napravimo provjere vrijednosti kako se ne bi neispravne vrijednosti zapisale u bazu. Provjere se ponajviše odnose na datumske zapise i brojčane vrijednosti.

```
def test_updating_failed_products_wrong_num_uprice(app, main_menu,
↪  prod, my_singleton):
    QTest.mouseClick(main_menu.btn_products, Qt.LeftButton)
    prod.table_widget.item(1, 1).setText("New product name")
    prod.table_widget.item(1, 7).setText("Wrong unit price")
    prod.table_widget.selectRow(1)
    prev_pro =
    ↪  my_singleton.get_product_by_id(prod.table_widget.item(1,
    ↪  0).text())
    QTest.mouseClick(prod.button_update, Qt.LeftButton)
    new_prod = my_singleton.get_product_by_id(prev_pro[0])

    assert (prev_pro[0] == new_prod[0]
            and prev_pro[1] == new_prod[1]
            and new_prod[1] != "New product name"
            and new_prod[7] != "Wrong unit price"
            and prod.total_failed_update_rows == 1
            and prod.update_message.find("Unit price should be
            ↪  decimal value") != -1)
```

Kada smo definirali testove potrebno je dodati gumb za ažuriranje retka te na njega spojiti akciju koja obrađuje ažuriranje. Osim samo ažuriranja (s provjerama na front endu) potrebno je definirati *propertye* koji služe za mjerenje broja promijenjenih redaka i broja nepromijenjenih redaka te povratne poruke koja će se korisniku predočiti kao skočni prozor s porukom.

```
self.button_update = QPushButton('Update selected row', self)
self.layout.addWidget(self.button_update)
self.button_update.clicked.connect(self.update_selected_row)
```

```
def update_selected_row(self):
indexes = self.table_widget.selectionModel().selectedRows()
self.total_updated_rows = 0
self.total_failed_update_rows = 0
self.update_message = ""
if len(indexes) > 0:
    for index in sorted(indexes):
        if self.table_widget.item(index.row(), 1).text() == "":
            self.total_failed_update_rows += 1
            self.update_message += f"Row: {index.row() + 1} failed
             ↪  update. Please enter required fields (Supplier
             ↪  name)\n"
        else:
            s = db_singleton.Singleton.get_instance()
            res =
             ↪  s.update_supplier(self.table_widget.item(index.row(),
             ↪  0).text(),

                                          ↪  self.table_widget.item(index.row(),
                                          ↪  1).text(),

                                          ↪  self.table_widget.item(index.row(),
                                          ↪  2).text(),

                                          ↪  self.table_widget.item(index.row(),
                                          ↪  3).text(),

                                          ↪  self.table_widget.item(index.row(),
                                          ↪  4).text(),

                                          ↪  self.table_widget.item(index.row(),
                                          ↪  5).text(),

                                          ↪  self.table_widget.item(index.row(),
                                          ↪  6).text(),

                                          ↪  self.table_widget.item(index.row(),
                                          ↪  7).text(),

                                          ↪  self.table_widget.item(index.row(),
                                          ↪  8).text(),
```

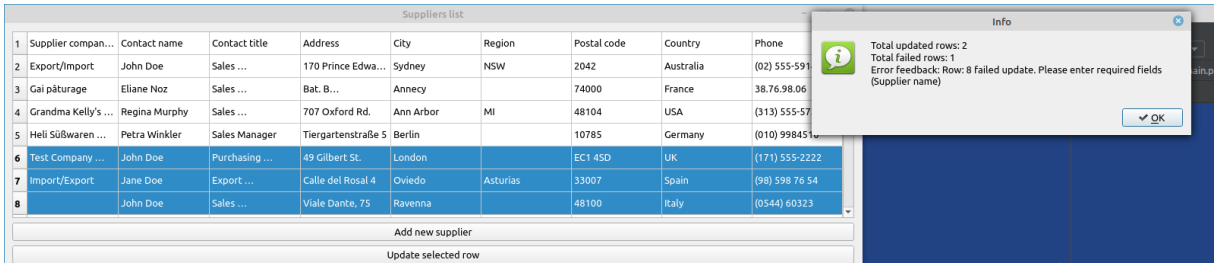
```
                                        ↪  self.table_widget.item(index.row(),
                                        ↪  9).text())

            if res != "Row is successfully updated":
                self.total_failed_update_rows += 1
                self.update_message += f"Error while updating {res}
                 ↪  on row {index.row() + 1} \n"
            else:
                self.total_updated_rows += 1

    self.warning.setIcon(QMessageBox.Information)
    self.warning.setText(f"Total updated rows:
     ↪  {self.total_updated_rows} \n"
                         f"Total failed rows:
                          ↪  {self.total_failed_update_rows} \n"
                         f"Error feedback: {self.update_message}")
    self.warning.setWindowTitle("Info")
    self.warning.setStandardButtons(QMessageBox.Ok)
    self.warning.show()

else:
    self.warning.setIcon(QMessageBox.Information)
    self.warning.setText("Please select rows that you want to
     ↪  update")
    self.warning.setWindowTitle("Update warning")
    self.warning.setStandardButtons(QMessageBox.Ok |
     ↪  QMessageBox.Cancel)
    self.warning.show()
```

Nakon što imamo definirano korisničko sučelje potrebno je samo pripremiti SQL funkciju koja će se povezivati na bazu podataka. S obzirom na to da je prikazana logika spajanja na bazu i vraćanja rezultata u prethodnim primjerima sada će biti prikazana samo SQL naredba napisana u sqlite3 koristeći parametrizirani unos (kao sigurniji način pisanja upita). Neobavezna polja imaju definiranu vrijednost NULL (ako su ostavljena prazna smatra se NULL vrijednosti, dok obavezna polja ne smiju poprimiti NULL vrijednost).

```
    def update_supplier(self, supplier_id, company_name,
     ↪  contact_name="NULL",contact_title="NULL",
     ↪  address="NULL",city="NULL", region="NULL",
     ↪  postal_code="NULL", country="NULL", phone="NULL"):
        updated_supplier = (
            company_name, contact_name, contact_title, address, city,
             ↪  region, postal_code, country, phone, supplier_id)
```

```
sql = '''UPDATE Suppliers SET CompanyName = ?, ContactName =
↪  ?, ContactTitle = ?, Address = ?, City = ?, Region = ?,
                PostalCode = ?, Country = ?, Phone = ?
                WHERE SupplierID = ?'''
```

Nakon što je izvršena naredba funkcija vraća ili poruku o uspješnosti ažuriranja ili poruku o grešci koja se zapisuje u *property* koji se prikazuje korisniku nakon izvršavanja. To nam je potrebno jer je tako zadano u scenariju testiranja da osim samih provjera ispravnosti se da i povratna poruka korisniku. Sada imamo kompletirane funkcionalnosti kako bismo mogli pokrenuti i uspješno izvršiti oba testa. Prvi test će proći jer se vrijednost neće ažurirati zbog provjere na sučelju aplikacije dok drugi test će ažurirati vrijednosti jer su sve kombinacije vrijednosti legitimne.



Slika 14: Prikaz ažuriranja dobavljača

4.5. Integracijsko testiranje komponenti čitanja i ažuriranja

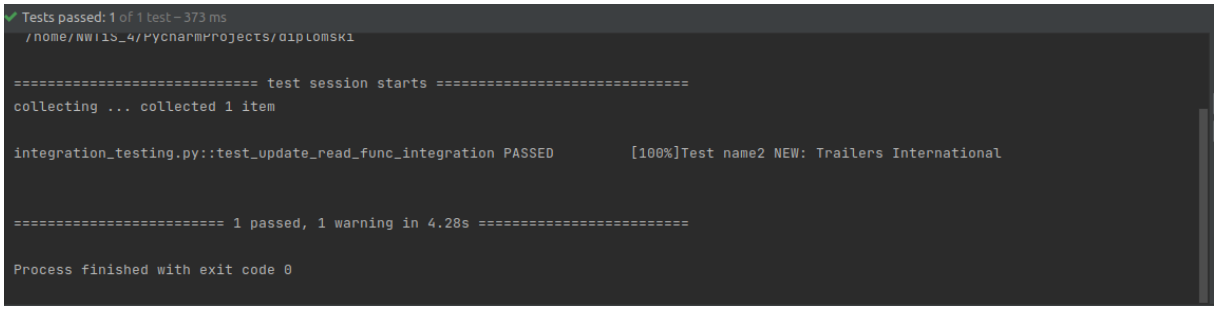
Prethodno smo imali primjer jediničnog testiranja koje se provlačilo nad nekoliko funkcija i klasa. To testiranje je označeno kao jedinično zato što sve te klase radi zajedno i imaju isti cilj. Integracijsko testiranje radimo na cijelim sustavom (dosadašnjim) te gledamo kako komponente (sastavljene od jedinica) zajedno rade. Kako bismo testirali integraciju komponenti za čitanje i prikaz podataka te komponente za osvježavanje podataka koristit ćemo pytest biblioteku koja posjeduje dekorator pymark.test.integration. Taj dekorator označava da je test integracijski, a služi tome da se integracijski testovi ne izvode ako jedinični ne prolaze.

Postavlja se pitanje kakav ćemo scenarij definirati za integracijsko testiranje. S obzirom na to da integracijskim testiranjem testiramo kako komponenta koju dodajemo (ovaj put komponenta ažuriranja) utječe na ostatak sustava, sustav se zasad sastoji od jedne komponente (čitanje) prvo integracijsko testiranje bit će najjednostavnije. Scenarij koji će biti korišten je da je pročitan redak iz baze podataka te spremimo njegovu vrijednost u varijablu. Zatim ćemo tu vrijednost osvježiti te nakon toga provjeriti događa li se provjera na komponenti za čitanje (zapis je promijenjen).

```
@pytest.mark.integration_test
def test_update_read_func_integration(app, main_menu, ships, my_singleton):
    QTest.mouseClick(main_menu.btn_shippers, Qt.LeftButton)
```

```
first_value_of_company_name = ships.table_widget.item(1, 1).text()
expected_changed_value = "Trailers International"
ships.table_widget.item(1, 1).setText(expected_changed_value)
ships.table_widget.selectRow(1)

QTest.mouseClick(ships.button_update, Qt.LeftButton)
changed_value = ships.table_widget.item(1, 1).text()
print(f"{first_value_of_company_name} NEW: {changed_value}")
assert (ships.total_updated_rows == 1
        and first_value_of_company_name != changed_value
        and changed_value == expected_changed_value)
```

Nakon što pokrenemo test dobivamo poruku o uspješnom izvršavanju uz pomoćni ispis u konzolu kako bismo vidjeli da su uistinu ispravne vrijednosti učitane.



Slika 15: Prikaz rezultata integracijskog testiranja read i update komponenti

4.6. Testiranje i razvoj komponente za dodavanje novih zapisa

Nakon što smo dodali read i update funkcionalnosti sljedeća nam je funkcionalnost kreiranja (eng. *create*). Već na prethodnoj snimci zaslona mogli smo vidjeti gumb "Add new" koji zasad ništa ne radi (dodan je kako bi se mogao poravnati gumb update selected rows kojeg smo morali dodati zbog prethodnih testova). Za dodavanje novih redaka bit će nam potreban zaseban ekran (poziva se preko prethodno spomenutog gumba). Unutar novog ekrana bit će omogućen unos podataka potrebnih za novi zapis. Postojat će gumb koji će pozivati operaciju add za bazu podataka.

Testovi koje moramo definirati uključuju sljedeće scenarije:

- Scenarij 1: Nisu ispunjena obavezna polja- ne dodaje se novi zapis, ispisuje se poruka o grešci
- Scenarij 2: Podaci su zadani u krivom formatu (npr. cijena nije decimalna vrijednost)- ne dodaje se novi zapis, ispisuje se poruka o grešci

- Scenarij 3: Krivo su zadani vanjski ključevi (vrijednost vanjskog ključa ne postoji u bazi), ispisuje se poruka o grešci
- Scenarij 4: Ispravno definirani podaci- dodaje se novi zapis, ispisuje se poruka o uspješnom dodavanju

Za primjer je uzet entitet Product jer je zanimljiv jer ima dosta restrikcija vezanih uz kvalitetu podataka. Scenarij 1 za tablicu Product obuhvaća to da su uneseni podaci o imenu proizvoda. U ovom slučaju ispisat će se poruka kako se trebaju upisati obavezna polja (napomena: u kasnijim testovima u scenariju 2 će se događati da vrijednosti moraju biti unesene jer ako nisu ne mogu biti traženog formata, ova dva scenarija su razdvojena jer scenarij 1 obuhvaća samo restrikcije dizajna baze podataka koji ne traži da budu uneseni podaci iz scenarija 2 već to traži poslovna logika).

```
def test_adding_product_failed_product_name(self):
    s = db_singleton.Singleton.get_instance()
    app = QApplication(sys.argv)

    MainMenu = main.MainMenu()
    prod = Products.Products()
    add_prod = AddProducts()

    QTest.mouseClick(MainMenu.btn_products, Qt.LeftButton)
    QTest.mouseClick(prod.button, Qt.LeftButton)

    add_prod.category_id_tb.setText("1")
    add_prod.supp_id_tb.setText("1")

    previous_num = s.get_all_products_with_category_names()
    QTest.mouseClick(add_prod.button_prod, Qt.LeftButton)
    current_num = s.get_all_products_with_category_names()

    self.assertEqual(len(previous_num) == len(current_num), True)
    self.assertEqual(add_prod.label_result_prod.text() == "Please
    ↪  enter required fields", True)
```

Za scenarij 2 postoje tri kritične vrijednosti koje želimo testirati. Prva kritična vrijednost koju želimo testirati je Unit price koju želimo definirati da bude isključivo decimalna. Zatim sljedeće dvije vrijednosti možemo zajedno testirati, a to su Units in Stock i Units in Order koje moraju biti cjelobrojne vrijednosti. Za ove dvije provjere imamo dva testa koja se osim u testnim podacima razlikuju samo u tome da se ispiše drugačija poruka (potrebno je imati dva testa da svaku od provjera zasebno provjerimo kako bi lakše lokalizirali grešku).

```
def test_adding_product_failed_wrong_uprice(self):
    prod = Products.Products()
```

```
        add_prod = AddProducts()

        QTest.mouseClick(MainMenu.btn_products, Qt.LeftButton)
        QTest.mouseClick(prod.button, Qt.LeftButton)

        add_prod.product_name_tb.setText("Test")
        add_prod.category_id_tb.setText("1")
        add_prod.supp_id_tb.setText("1")
        add_prod.unit_price_tb.setText("Wrong unit price")
        add_prod.qty_unit_tb.setText("2")
        add_prod.unit_in_stock.setText("2")
        add_prod.unit_in_order.setText("4")

        previous_num = s.get_all_products_with_category_names()
        QTest.mouseClick(add_prod.button_prod, Qt.LeftButton)
        current_num = s.get_all_products_with_category_names()

        self.assertEqual(len(previous_num) == len(current_num), True)
        self.assertEqual(add_prod.label_result_prod.text() == "Unit
        ↪  price should be decimal value", True)
```

Za tablicu Product u scenariju 3 trebamo paziti da zadamo valjane vrijednosti za sljedeće stupce: CategoryID i SupplierID. Ove dvije funkcije su više-manje podjednake osim što se provjerava ispravnost različitog parametra, ali obje funkcioniraju na isti način (provjeri se postoji li u bazi podataka za tablicu Supplier ili Category postoji li zapis s tim ID, ako ne postoji ispiši pogrešku).

```
    def test_adding_product_failed_supp_id(self):
        prod = Products.Products()
        add_prod = AddProducts()

        QTest.mouseClick(MainMenu.btn_products, Qt.LeftButton)
        QTest.mouseClick(prod.button, Qt.LeftButton)

        add_prod.product_name_tb.setText("Test")
        add_prod.category_id_tb.setText("1")
        add_prod.supp_id_tb.setText("Unknown")

        previous_num = s.get_all_products_with_category_names()
        QTest.mouseClick(add_prod.button_prod, Qt.LeftButton)
        current_num = s.get_all_products_with_category_names()

        self.assertEqual(len(previous_num) == len(current_num), True)
```

```
        self.assertEqual(add_prod.label_result_prod.text() == "Please
        ↪  provide valid Supplier ID", True)
```

Posljednji scenarij nam je onaj u kojem se neće dogoditi greške iz prethodna tri scenarija. To znači da će ispravno biti definiran Company Name, vanjski ključevi će biti validne vrijednosti, a brojčane vrijednosti bit će zapisane u ispravnom formatu. Ako se to dogodi trebala bi se ispisati poruka o ispravnom dodavanju te bi se trebao povećati broj zapisa u bazi za 1.

```
    def test_adding_product_success(self):
        prod = Products.Products()
        add_prod = AddProducts()

        QTest.mouseClick(MainMenu.btn_products, Qt.LeftButton)
        QTest.mouseClick(prod.button, Qt.LeftButton)

        add_prod.product_name_tb.setText("Test test")
        add_prod.category_id_tb.setText("1")
        add_prod.supp_id_tb.setText("1")
        add_prod.qty_unit_tb.setText("2")
        add_prod.unit_in_order.setText("3")
        add_prod.unit_in_stock.setText("3")
        add_prod.unit_price_tb.setText("2.7")

        previous_num = s.get_all_products_with_category_names()
        QTest.mouseClick(add_prod.button_prod, Qt.LeftButton)
        current_num = s.get_all_products_with_category_names()

        self.assertEqual(len(previous_num) + 1 == len(current_num),
        ↪  True)
        self.assertEqual(add_prod.label_result_prod.text() == "New
        ↪  row was added", True)
```

Sljedeći korak nam je kreirati ekran koji će se baviti obrađivanjem ovih podataka. Ekran se mora sastojati od elemenata za upis vrijednosti, labela koje označavaju pojedine elemente, gumba za dodavanje retka te labela za ispisivanje poruku (korištena će biti crvena boja za ekspresivnost). Treba naglasiti kako se ekranu dodavanja novog zapisa pristupa iz pregleda svih zapisa taj entitet te je i taj dio pokriven testom (dio s testiranjem navigacije je obrađen na početku kod main menu-a). Kako su kod tablice Product zajedno prikazani i podaci o kategorijama proizvoda kada pristupamo ekranu za dodavanje proizvoda nudi nam se i dodavanje kategorija. Ta dva entiteta se nalaze na istom ekranu, a svaki ima svoje elemente za dodavanje koji su neovisni o elementima drugog entiteta. Za tablicu Category je isti princip razvoja kao i za Product pa nije potrebno posebno izdvajati neke dijelove te implementacije (jedina razlika je što su ograničenja puno jednostavnija i ne postoje ekvivalenti scenarija 2 i scenarija 3).

```
class AddProducts(QWidget):
```

```
def __init__(self):
    self.layout = QVBoxLayout()
    self.setLayout(self.layout)

    self.button_prod = QPushButton('Add new product', self)
    self.button_prod.resize(120, 50)
    self.button_prod.move(330, 230)
    self.button_prod.clicked.connect(self.add_product_db)

    self.label_result_prod = QtWidgets.QLabel(self)
    self.label_result_prod.setText("")
    self.label_result_prod.move(460, 245)
    self.label_result_prod.adjustSize()

    self.product_name_tb = QtWidgets.QLineEdit(self)
    self.product_name_tb.move(130, 30)
    self.product_name_tb.resize(200, 40)
    self.product_name_tb.setPlaceholderText("Enter new product
    ↪  name")

    label_product_name = QtWidgets.QLabel(self)
    label_product_name.setText('Product name*')
    label_product_name.move(10, 40)

    #izbacena su definiranja ostalih labela i textboxeva zbog
    ↪  preglednosti

def add_product_db(self):
    if self.product_name_tb.text() == "":
        self.label_result_prod.setText("Please enter required
        ↪  fields")
        self.label_result_prod.setStyleSheet("color: red")
    else:
        s = db_singleton.Singleton.get_instance()
        res = s.insert_new_product(self.product_name_tb.text(),
        ↪  self.supp_id_tb.text(), self.category_id_tb.text(),
                                   self.qty_unit_tb.text(),
                                   ↪  self.unit_price_tb.text(),
                                   ↪  self.unit_in_stock.text(),
                                   self.unit_in_order.text())
        self.label_result_prod.setText(res)

def add_category_db(self):
```

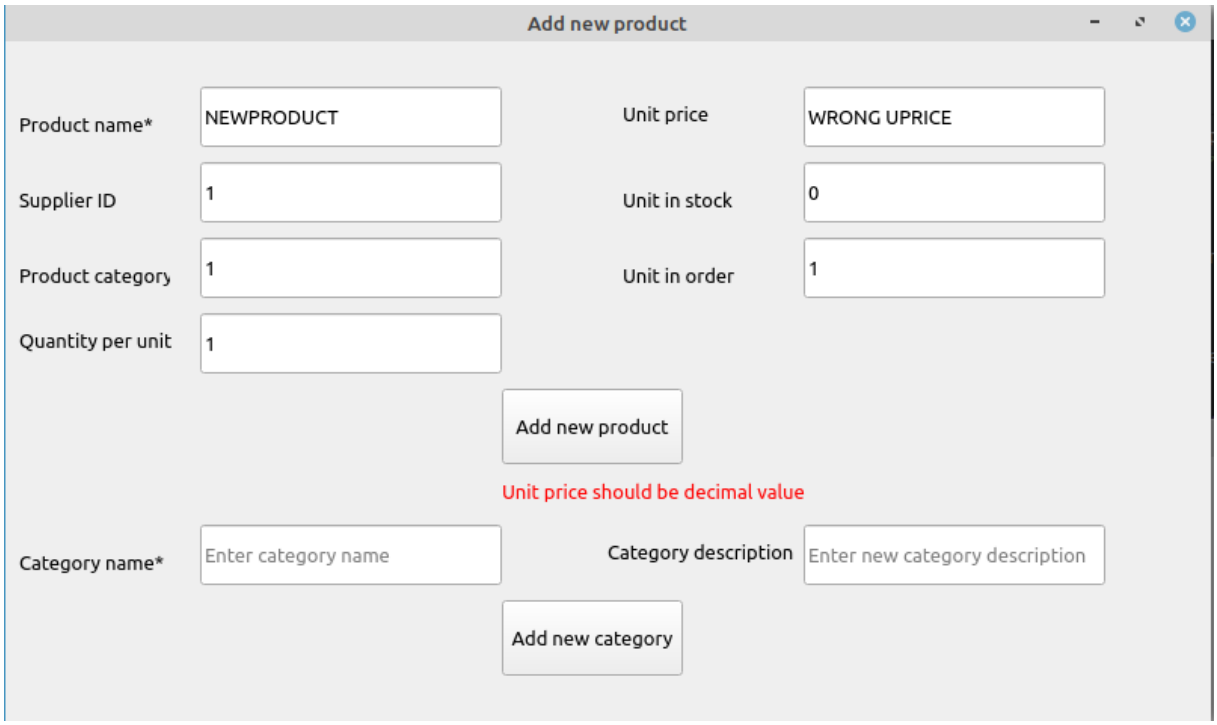
```
        if self.category_name_tb.text() == "":
            self.label_result_cat.setText("Please enter required
            ↪  fields")
            self.label_result_cat.setStyleSheet("color: red")
        else:
            s = db_singleton.Singleton.get_instance()
            res = s.insert_new_category(self.category_name_tb.text(),
            ↪  self.category_desc_tb.text())
            self.label_result_cat.setText(res)
```

Nakon što imamo GUI potrebno je napisati funkciju koja će odraditi dodavanje u bazu podataka te vraćanje rezultata provjere ispravnosti podataka. Funkcije za dodavanje će biti prilično slične funkcijama za ažuriranje te će također koristiti parametrizirane upite.

```
    def insert_new_product(self, product_name, supplier_id="NULL",
    ↪  category_id="NULL", qty_per_unit="NULL",
                           uprice="NULL", ustock="NULL",
                           ↪  uorder="NULL"):
        cat = self.get_category_by_id(category_id)
        if cat == 0: return "Please provide valid Category ID"
        sup = self.get_supplier_by_id(supplier_id)
        if sup == 0: return "Please provide valid Supplier ID"

        if not ustock.isnumeric(): return "Quantity, Unit in stock
        ↪  and order should be numeric values"
        if not qty_per_unit.isnumeric(): return "Quantity, Unit in
        ↪  stock and order should be numeric values"
        if not uorder.isnumeric(): return "Quantity, Unit in stock
        ↪  and order should be numeric values"

        try:
            uprice = float(uprice)
        except ValueError:
            return "Unit price should be decimal value"

        new_product = (product_name, supplier_id, category_id,
        ↪  qty_per_unit, uprice, ustock, uorder, 0)

        sql = '''INSERT INTO Products(ProductName, SupplierID,
        ↪  CategoryID, QuantityPerUnit, UnitPrice
                , UnitsInStock, UnitsOnOrder, Discontinued) VALUES(?,
↪  ?, ?, ?, ?, ?, ?, ?)'''
        try:
            cur.execute(sql, new_product)
```

```
            self.con.commit()
            return "New row was added"
        except Exception as e:
            return str(e)
```

Sada imamo sve elemente da izvršimo testiranja. Nakon izvršenih testiranja možemo pogledati u GUI kako se aplikacija ponaša kada ručno pokušamo neke primjere izvesti. Na primjer, možemo provjeriti kako se ponaša kada unesemo vrijednost krivog formata u polje Unit Price.



Slika 16: Greška prilikom dodavanja novog proizvoda

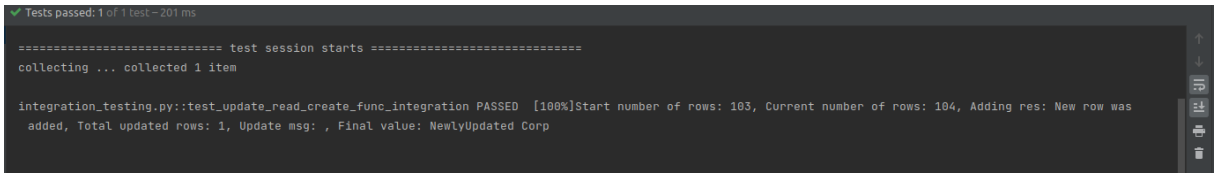
4.7. Integriranje komponente za dodavanje zapisa u sustav

Nakon integriranja komponente update na sustav (koji se u tom trenutku sastojao samo od komponente read) trebamo integrirati komponentu za dodavanje zapisa. Kao što se može primijetiti korišten je iterativni pristup gdje se nakon razvijanja svake komponente provodi integracijsko testiranje pa se zatim ide u novu iteraciju (svaka iteracija je jedna komponenta). Za razliku od prvog integracijskog testnog slučaja gdje smo morali provesti jedan korak sada je ipak situacija kudikamo kompliciranija. Potrebno je razviti test koji će obuhvaćati funkcionalnosti sve tri razvijene komponente te ih testirati kako međusobno rade.

Prvo je potrebno testirati odnos između nove komponente i komponente za čitanje. Njihov odnos bi se trebao svoditi na to da ako dodamo novi zapis preko sučelja trebao bi se prikazati unutar čitanja. Zatim je potrebno *create-update* odnos testirati. Zapis koji smo zadnji dodali trebalo bi biti moguće ažurirati te bi trebalo testirati ažurira li se zapis na zadanu vrijednost. Naposljetku taj zapis bi se trebao ažurirati i pojaviti unutar pregleda zapisa.

```
@pytest.mark.integration_test
def test_update_read_create_func_integration(app, main_menu, cust,
 ↪  my_singleton, add_customer):
    QTest.mouseClick(main_menu.btn_customers, Qt.LeftButton)
    start_number_of_cust = len(my_singleton.get_all_customers())
    QTest.mouseClick(cust.button, Qt.LeftButton)
    add_customer.company_name_tb.setText("NewlyAdded Corp")
    QTest.mouseClick(add_customer.button, Qt.LeftButton)
    adding_result = add_customer.label_result.text()

    QTest.mouseClick(main_menu.btn_customers, Qt.LeftButton)
    current_number_of_cust = len(my_singleton.get_all_customers())
    cust.table_widget.item(current_number_of_cust - 1,
     ↪  1).setText("NewlyUpdated Corp")
    cust.table_widget.selectRow(current_number_of_cust - 1)
    QTest.mouseClick(cust.button_update, Qt.LeftButton)
    total_updated_rows = cust.total_updated_rows
    update_msg = cust.update_message

    QTest.mouseClick(main_menu.btn_customers, Qt.LeftButton)
    changed_value =
     ↪  my_singleton.get_customer_by_id(cust.table_widget.item(current_number_of
     ↪  - 1, 0).text())[1]
    print(f"Start number of rows: {start_number_of_cust}, Current
     ↪  number of rows: {current_number_of_cust}, Adding res:
     ↪  {adding_result}, Total updated rows: {total_updated_rows},
     ↪  Update msg: {update_msg}, Final value: {changed_value}")
    assert (start_number_of_cust + 1 == current_number_of_cust
            and adding_result == "New row was added"
            and total_updated_rows == 1
            and update_msg == ""
            and changed_value == "NewlyUpdated Corp")
```

U prethodnom kodu vidimo da se prvo uzme broj trenutnih redaka iz baze podataka (za tablicu customera). Ta funkcija iz baze podataka je zapravo izvor podataka za tablicu iz read komponente. Nakon toga kreiramo novi zapis kojem moramo definirati "Company Name". Taj zapis dodajemo u bazu te provjeravamo je li uspješno dodan. Nakon toga ponovno uzimamo broj redaka iz baze podataka da vidimo je li se povećao broj za 1. Zatim uzmemo posljednji zapis iz tablice te mu promijenimo ime. Nakon toga ažuriramo taj redak. Na kraju provjerimo u bazi vrijednost promijenjenog atributa te ga usporedimo s ažuriranom vrijednosti. Dodan je ispis za indikativnije pokretanje testa.



Slika 17: Pokretanje integracijskog testa za integraciju dodavanja

4.8. Testiranje i dodavanje komponente za brisanje zapisa

Kako bismo krenuli s implementacijom komponente za brisanje moramo znati nešto o tome na koji način baze podataka funkcioniraju. Restrikcije referencijalnog integriteta nam ne dozvoljavaju brisanje redaka koji imaju vanjski ključ. Postoji način na koji možemo definirati ponašanje prilikom brisanja ovih redaka (jer ipak želimo da se mogu obrisati).

Postoji niz akcija koje možemo napisati prilikom definiranja akcija brisnanja i ažuriranja u bazi podataka:

- NO ACTION: jednostavno ništa se ne dogodi nakon brisanja ili ažuriranja
- RESTRICT: ne dozvoljava nikakve promjene na poljima koja su korištena kao vanjski ključevi u barem jednom zapisu
- SET NULL: vrijednost se postavlja na NULL ukoliko se dogodi neka promjena
- SET DEFAULT: vrijednost se postavlja na defaultnu ukoliko se dogodi neka promjena
- CASCADE: ukoliko se dogodi promjena, promjena se prenosi i na sve retke koji referenciraju taj redak. To znači ukoliko se redak obriše svi reci koji se referenciraju će također biti obrisani.[27]

Prilikom odabira moramo utvrditi kakvo ponašanje očekujemo. No action nam je preliberalan za to što želimo (želimo da se promjene poprate), restrict je preagresivan te nam nad većinom redaka neće dozvoliti nikakve promjene što će nam ograničiti funkcionalnosti, set null nam je validan izbor, ali je nedovoljno indikativan (pogotovo ako neki proizvod promjeni ime ili kategoriju u kojoj se nalazi ne želimo da se u računima kao proizvod pojavi null vrijednost), set default bi nam se praktični sveo na set null (osim što umjesto null bi pisalo nepoznata vrijednost). Stoga čini se cascade kao validan izbor. Ako je nešto uneseno greškom možemo izbrisati, a ako se vrijednosti promjene ažurirat će se u ostalim recima. Lako možemo načiniti promjene na našoj bazi unutar skripte, a to je u ovoj fazi sasvim validno raditi jer koristimo testnu bazu, a ne produkcijsku.

Kako bismo krenuli s testiranjem trebamo definirati scenarije. Scenarij nam je tu prilično jednostavan. Želimo označiti redak i obrisati ga što uzrokuje manji broj redaka u bazi podataka te da redak s tom ID vrijednošću više ne postoji.

```
    def test_delete_customer(self):
        QTest.mouseClick(self.main_menu.btn_customers, Qt.LeftButton)
```

```
        self.cust.table_widget.selectRow(1)
        prev_row_num = len(self.singleton.get_all_customers())
        id_1 = self.cust.table_widget.item(1, 1).text()
        QTest.mouseClick(self.cust.button_delete, Qt.LeftButton)

        cur_row_num = len(self.singleton.get_all_customers())
        prev_cust_1 = self.singleton.get_customer_by_id(id_1)

        assert (prev_row_num - 1 == cur_row_num
                and prev_cust_1 == 0)
```

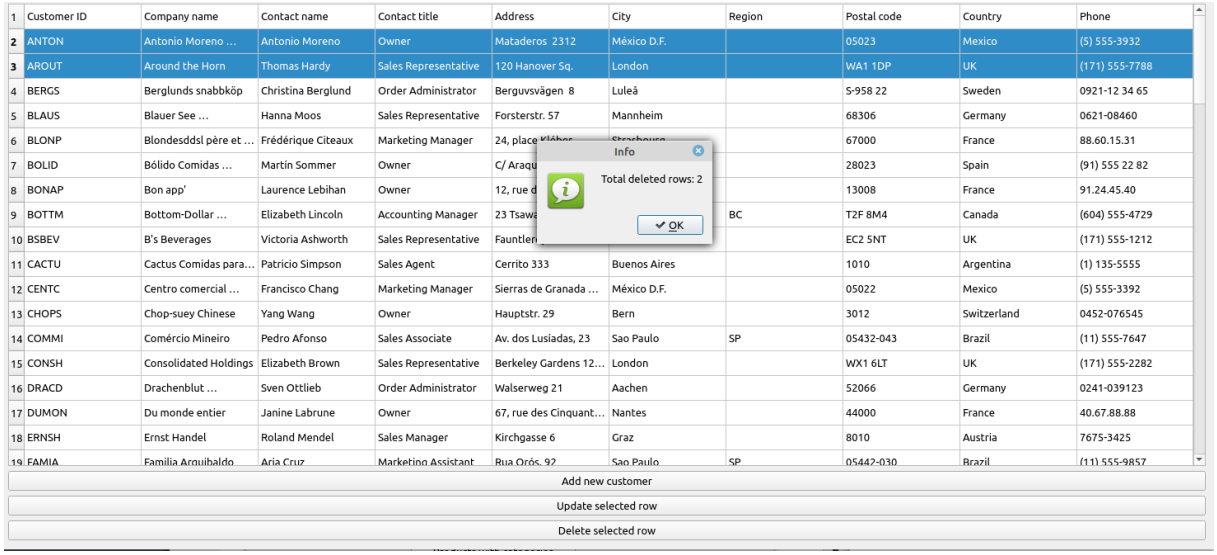
Nakon toga slijedi nam definirati GUI elemente koji će pozivati brisanje. To je prilično jednostavno jer je dodavanje analogno dodavanjima gumbova za ažuriranje i dodavanje. Na gumb povežemo akciju. Akcija je jednostavna, poziva klasu za rad s bazom podataka i daje joj ID koji treba obrisati. Nakon toga korisniku ispiše poruku s brojem redaka koji su obrisani. Također vraća se povratna informacija ako niti jedan redak nije označen.

```
    def delete_rows(self):
        indexes = self.table_widget.selectionModel().selectedRows()

        if len(indexes) > 0:
            for index in sorted(indexes):
                s = db_singleton.Singleton.get_instance()

                ↪  print(s.get_customer_by_id(self.table_widget.item(index.row(
                ↪  0).text()))
                s.delete_customer(self.table_widget.item(index.row(),
                ↪  0).text())
            self.warning.setIcon(QMessageBox.Information)
            self.warning.setText(f"Total deleted rows:
            ↪  {len(indexes)}")
            self.warning.setWindowTitle("Info")
            self.warning.setStandardButtons(QMessageBox.Ok)
            self.warning.show()

        else:
            self.warning.setIcon(QMessageBox.Information)
            self.warning.setText("Please select rows that you want to
            ↪  delete")
            self.warning.setWindowTitle("Delete warning")
            self.warning.setStandardButtons(QMessageBox.Ok |
            ↪  QMessageBox.Cancel)
            self.warning.show()
```

Sada još samo trebamo napraviti poziv na bazu podataka. Ovo nam je jedna od jednostavnijih funkcija i samo je bitno da se pozove naredba za brisanje kojoj se kao parametar proslijedi id retka kojeg želimo obrisati. Nakon toga moramo commit naredbu pozvati kako bi se promjene stvarno dogodile u bazi.

```
get_by_id = f'DELETE FROM Customers WHERE CustomerID = \"{customer_id}\"'
cur.execute(get_by_id)
self.con.commit()
```

Sada imamo potpunu funkcionalnost te je možemo pozvati iz izbornika te vidjeti na koji način se ponaša.



Slika 18: Brisanje redaka iz GUI-ja

4.9. Integracija komponente za brisanje zapisa

Brisanje je zadnja od osnovnih CRUD komponenti te ju stoga treba istestirati s cijelim dosadašnjim sustavom. Prvo je potrebno kreirati scenarij gdje će međusobno komunicirati sve komponente te nakon toga napisati test koji će to obuhvaćati.

Scenarij će se sastoji od više koraka nego dosadašnji. S obzirom na to da je delete funkcionalnost "finalna" te se nakon nje s tim zapisom ne može više ništa raditi teško je pronaći scenarij gdje će zajedno raditi *update* i *delete* funkcionalnosti. Možemo obuhvatiti sve komponente na sljedeći način: pročitamo zapise iz baze (read), dodamo novi zapis (create), pročitamo ponovno zapise da vidimo ima li promjene broja redaka (read), ažuriramo dodani redak (update), izbrišemo ažurirani redak (delete), pročitamo trenutni broj zapisa da vidimo promjenu (read - broj zapisa bi trebao biti manji).

```
@pytest.mark.integration_test
def test_update_read_create_delete_func_integration(app, main_menu,
 ↪  cust, my_singleton, add_customer):
```

```
QTest.mouseClick(main_menu.btn_customers, Qt.LeftButton)
start_number_of_cust = len(my_singleton.get_all_customers())
QTest.mouseClick(cust.button, Qt.LeftButton)
add_customer.company_name_tb.setText("NewlyAdded Corp")
QTest.mouseClick(add_customer.button, Qt.LeftButton)
adding_result = add_customer.label_result.text()

QTest.mouseClick(main_menu.btn_customers, Qt.LeftButton)
second_number_of_cust = len(my_singleton.get_all_customers())
cust.table_widget.item(second_number_of_cust - 1,
↪  1).setText("NewlyUpdated Corp")
cust.table_widget.selectRow(second_number_of_cust - 1)
QTest.mouseClick(cust.button_update, Qt.LeftButton)
total_updated_rows = cust.total_updated_rows
update_msg = cust.update_message
id_updated_row = cust.table_widget.item(second_number_of_cust -
↪  1, 0).text()

changed_value =
↪  my_singleton.get_customer_by_id(cust.table_widget.item(second_number_of_
↪  - 1, 0).text())[1]
cust.table_widget.selectRow(second_number_of_cust - 1)
QTest.mouseClick(cust.button_delete, Qt.LeftButton)
final_number_of_cust = len(my_singleton.get_all_customers())
row_cust_id_found =
↪  my_singleton.get_customer_by_id(id_updated_row)

assert (start_number_of_cust + 1 == second_number_of_cust
        and adding_result == "New row was added"
        and total_updated_rows == 1
        and update_msg == ""
        and changed_value == "NewlyUpdated Corp"
        and final_number_of_cust == start_number_of_cust
        and row_cust_id_found == 0)
```

Vidimo da prvo testiramo startni broj klijenata, zatim dodajemo novi redak te provjeravamo je li dodavanje provedeno na zadovoljavajuć način. Zatim ažuriramo novododani redak te gledamo je li sve prošlo po planu s ažuriranjem. Kasnije nam ostaje uzeti ID retka kako bi znali koji smo redak ažurirali. U sljedećem koraku taj redak brišemo te provjerimo je li smanjen broj redaka za jedan te postoji li u bazi podataka redak s tim ID-om. Mogli bismo dodati još provjeru poruke o broju izbrisanih redaka, ali taj nam je test redundantan jer već imamo provjeru broj izbrisanih redaka, a kako je riječ o istom broju nema potrebe raditi test dvaput.

4.10. Korištenje oponašanja u testiranju

Korištenje oponašanja (eng. *mock*) prilikom testiranja može imati više svrha. Uobičajeno se koristi kako bi se izbjegla ovisnost nekog testa o jednoj komponenti koju zamjenjujemo mockom. Mockom može definirati ponašanje nekog dijela koda te tako izbjeći ovisnost o njegovoj implementaciji. Ovo pogotovo može biti važno ako nisu još razvijeni svi dijelovi. Također ako ne želimo izvršiti sve radnje koje neka metoda definira već nam je potreban samo finalni rezultat možemo simuliranim ponašanjem izbjeći izvršavanje cijele funkcionalnosti.

Prvi *mock* koji ćemo kreirat će nam pokazati kako izgleda osnovna sintaksa. Prvo što definiramo je @patch dekorator koji nam govori koju metodu želimo "zakrpati" (zamijeniti sa mockanim ponašanjem). Zatim u argumentima funkcije proslijedimo ime *mocka* koji ćemo koristiti za simuliranje ponašanja. Prvi slučaj je prilično jednostavan kako bi se pokazalo na koji način funkcionira pozivanje mockane funkcije. Definiramo što nam je traženi rezultat, zatim definiramo rezultat *mock* funkcije kako bismo dobili traženi rezultat. Nakon toga kada tražimo stvarni rezultat pozivamo stvarno funkciju koja bi trebala dohvaćati podatke iz baze podataka. No, kako je ova funkcija zamijenjena simuliranim ponašanjem ona se ne izvršava (što je lako provjeriti s debuggerom).

```
@patch("northwind_db_singleton.Singleton.get_company_name_by_product_name")
    def test_company_by_product_name(self, mock_get_comp_name_by_prod_name):
        wanted_company_name = ['Exotic Liquids']
        mock_get_comp_name_by_prod_name.result_value = ['Exotic Liquids']

        actual_result = self.s.get_company_name_by_product_name("Chang")

        assert (actual_result, wanted_company_name)
```

Drugi primjer će pokazati jednu korisnu komponentu korištenja *mockova*. Ta komponenta je ta da nam omogućavaju testiranje funkcije koje nemaju povratnu vrijednost. Takve funkcije su obično one funkcije koje pozivaju druge funkcije te nemaju definiranu svoju zasebnu funkcionalnost. Ako imamo jednostavnu funkciju koja prema imenu proizvoda utvrdi postoji li i treba li se osvježiti vrijednost ili proizvod ne postoji i treba ga dodati. Ovakva funkcija će pozvati metode iz baze podataka koje će obaviti svoju zadaću te će izaći bez povratne vrijednosti. Ne možemo bez direktnog pogleda u bazu podataka utvrditi je li se vrijednost ispravno promijenila ili dodala. Zbog toga ćemo koristiti svojstvo mockova koje se zove callcount i govori nam koliko puta je neka *mock* funkcija pozvana (poziva se umjesto glavnih funkcija iz baze podataka). Također kako imamo dvije funkcije morat ćemo obje zamijeniti. To radimo tako da dodamo dva patch decoratora koji će definirati obje funkcije koje mijenjamo. Također kao ulazne parametre trebamo definirati oba *mocka*. Treba pripaziti na redoslijed jer će prvi definirani *mock* zamijeniti najdonji *patch* i tako redom. Definiramo listu od tri proizvoda od kojih bi dva trebala ići na ažuriranje te kreiramo test.

```
@patch("northwind_db_singleton.Singleton.update_product")
@patch("northwind_db_singleton.Singleton.insert_new_product")
```

```
def test_correct_function_calling(self, mock_insert, mock_update):
    product_list = ["FAKE", "Chang", "Chai"]
    for product in product_list:
        add_if_exists_else_update(product)

    self.assertEqual(mock_update.call_count, 2)
    self.assertEqual(mock_insert.call_count, 1)
```

4.11. Testiranje i dodavanje ekrana s grafičkim podacima

Iduća komponenta koja će biti dodana je komponenta koja će služiti korisniku kao *dashboard* za pregled podataka u bazi. Na tom *dashboard* bit će prikaz kretanja nekih indikatora poslovanja (eng. *key performance indicators*) kroz određeni vremenski period. S obzirom na to da nam je ključni entitet u bazi proizvod bilo bi potrebno omogućiti filtriranje po proizvodu i kategoriji proizvoda.

Kao indikatori poslovanja uzeti su neke od standardnih mjera koje su ili kumulativne ili po pojedinačnom proizvodu. Odabrane mjere su sljedeće:

- Total number of units sold (broj prodanih jedinica proizvoda u vremenu)
- Total price of units sold (ukupna zarada od prodaje proizvoda u vremenu)
- Average profit (zarada kroz broj prodanih jedninica u vremenu)
- Average price (prosječna cijena prodanih proizvoda u vremenu)
- Cost of goods sold (ukupna nabavna cijena prodanih proizvoda u vremenu)
- Total rebate (ukupan popust dan na prodane proizvode u nekom vremenu)
- Average rebate (prosječan popust dan na prodane proizvode u vremenu)
- Total profit (ukupan profit u vremenu)

Kako bi izračunali vrijednosti ovih indikatora koristit ćemo polja unit price, quantity i discount koja se nalaze unutar tablice Order details (stavke računa). Kao nabavne vrijednosti koristi ćemo podatke o jediničnoj cijeni iz tablice Product.

S obzirom da će konačni rezultat izračuna biti graf koji uzima za x os podatke o mjesecima, a na y os stavlja brojčane vrijednosti unutar testiranja ćemo se posvetiti tome da su vrijednosti koje dolaze do funkcije za iscrtavanje su validne. Na primjer, mjeseci su u rasponu od 1 do 12, godine koje pratimo su od 1900 (uzeta kao standardna vrijednost za račune kod kojih nemamo podataka o vremenu izdavanja jer to nije obavezno polje u bazi) do 2021 (zadnje do koje radimo obračun), određene vrijednosti moraju biti pozitivan broj ili nula(npr. COGS, rebate).

```
    def test_cogs_time(self):
        QTest.mouseClick(self.MainMenu.btn_analytics, Qt.LeftButton)
        result_list = self.s.get_cogs_time('Beverages', 'All')
        data_correct = True
        try:
            for row in result_list:
                cogs = float(row[1])

                if row[0] is not None:
                    year = int(row[0].split('/')[0])
                    month = int(row[0].split('/')[1])
                if row[0] is not None and ((year < 1900 or year >
                ↪  2022) or (month < 1 or month > 12)) or (cogs <
                ↪  0.00):
                    data_correct = False
                    print(row)
                    break
        except Exception:
            data_correct = False
        self.assertEqual(data_correct, True)
```

Također kako bismo napravili poseban ekran u kojem ćemo prikazivati podatke potrebno je definirati navigaciju s glavnog ekrana. Test za to je napravljen analogno prvom prikazan testu u odlomku main menu. Kod kreiranja GUI-ja moramo paziti da su podaci grupirani na ispravan način (naziv kategorije, brojčana vrijednost za cijelo razdoblje, graf kretanja po mjesecima trebaju biti jedno do drugog) i da odabir kategorija i proizvoda radi na ispravan način. Naime, bilo bi ispravno da ako odaberemo neku kategoriju da ne možemo više odabrati sve proizvode već samo proizvode iz te kategorije.

```
class Analytics(QWidget):
    def __init__(self):
        self.figure_quantity = plt.figure()
        self.canvas_quantity = FigureCanvas(self.figure_quantity)

        self.layout = QGridLayout()
        self.layout.addWidget(self.canvas_quantity, 1, 2)
        self.setLayout(self.layout)

        self.select_category = QComboBox()
        self.select_category.insertItem(0, "All")

        self.select_product = QComboBox()
        self.select_product.insertItem(0, "All")
```

```
        label_select_category = QLabel()
        label_select_category.setText("Select category")
        self.layout.addWidget(label_select_category, 0, 0)
        label_select_product = QLabel()
        label_select_product.setText("Select product")
        self.layout.addWidget(label_select_product, 0, 2)

        self.layout.addWidget(self.select_category, 0, 1)
        self.layout.addWidget(self.select_product, 0, 3)

        label_total_products_sold = QLabel()
        label_total_products_sold.setText("Total number of sold
        ↪  products")
        self.layout.addWidget(label_total_products_sold, 1, 0)

        self.total_products_sold = QLabel()
        self.total_products_sold.setText("0")
        self.layout.addWidget(self.total_products_sold, 1, 1)


        ↪  self.select_product.currentIndexChanged.connect(self.product_change)

        ↪  self.select_category.currentIndexChanged.connect(self.category_chang

        self.warning = QMessageBox()

        self.get_table_data()
        self.calculate_fields()
        self.plot()

    def plot(self):
        s = db_singleton.Singleton.get_instance()
        data_quantity =
        ↪  s.get_qty_data_by_months(self.select_category.currentText(),
        ↪  self.select_product.currentText())

        self.figure_quantity.clear()
        ax_qty = self.figure_quantity.add_subplot(111)
```

```
    for qty, total_price, avg_profit, avg_price, cogs,
    ↪  total_rebate, avg_rebate, total_profit in
    ↪  zip(data_quantity, data_total_price, data_avg_profit,
    ↪  data_avg_price, data_cogs, data_total_rebate,
    ↪  data_avg_rebate, data_total_profit):
        if qty[0] is None:
            ax_qty.plot('00/01', qty[1], 'bo')
        else:
            ax_qty.plot(qty[0][2:], qty[1], 'bo')

    plt.setp(ax_qty.get_xticklabels(), visible=False)

    self.canvas_quantity.draw()
def get_table_data(self):
    if self.select_category.currentText() != "All":
        self.select_product.clear()
        self.select_product.insertItem(0, "All")
        prod_names =
        ↪  s1.get_all_product_names(self.select_category.currentText())
        cnt = 1
        for product in prod_names:
            self.select_product.insertItem(cnt, product)
            cnt += 1

    else:
        cat_names = s1.get_all_category_names()
        prod_names = s1.get_all_product_names("All")
        cnt = 1
        for category in cat_names:
            self.select_category.insertItem(cnt, category)
            cnt += 1
        cnt = 1
        for product in prod_names:
            self.select_product.insertItem(cnt, product)
            cnt += 1

def category_changed(self):
    self.get_table_data()
    self.calculate_fields()
    self.plot()

def product_change(self):
    self.calculate_fields()
```

```
        self.plot()

    def calculate_fields(self):
        s = db_singleton.Singleton().get_instance()

        ↪  self.total_sales_number.setText(s.calculate_total_price(self.select_
        ↪  self.select_product.currentText()))
        if self.total_profit.text().startswith("-"):
            self.total_profit.setStyleSheet("color: red")
            self.avg_profit.setStyleSheet("color: red")
        else:
            self.total_profit.setStyleSheet("color: green")
            self.avg_profit.setStyleSheet("color: green")
```

U gornjem kodu je prikazan način na koji definiramo položaj elemenata putem klase QGridLayout(), kako pozivamo funkcije za prikaz podataka i za prikaz grafa. Funkcija za prikaz grafa nam je interesantna za pokazati jer se tu da na zgodan način iskoristiti funkcionalnost Pythona sa tzv. for zip petljom. Naime možemo imati istovremeno n iteratora koji iteriraju kroz n lista što nam koristi kada trebamo ispisati 8 grafova paralelno. Također prikazan je način na koji se parsiraju datumi kako bi se dobio čitljiviji zapis te na koji način će se pozivati funkcija iz baze podataka za učitavanje podataka. Sve je u gornjem kodu prikazano za jednu vrijednost (quantity) jer je za druge vrijednosti analogno, a da je ostavljen cijeli kod bilo bi teže prikazati osnovnu ideju.

Idući korak nam je definirati u bazi podataka upit s kojima ćemo dohvaćati podatke. Prvi put nam SQL upiti neće biti jednostavni već ćemo osim na sintaksu morati gledati na koji način se neka mjera izračunava. Također za svaku mjeru će nam biti potrebne dvije funkcije- jedna koja izračunava ukupnu vrijednost, a druga koja izračunava vrijednost kroz mjesece.

```
    def calculate_avg_profit(self, category_name, product_name):
        cur = self.con.cursor()
        if product_name == "All":
            if category_name != "All":
                get_prods = 'SELECT ((SUM((\"Order
                ↪  Details\".UnitPrice * Quantity) - (\"Order
                ↪  Details\".UnitPrice * Quantity) * Discount)) -
                ↪  SUM(Products.UnitPrice * Quantity)) /
                ↪  SUM(Quantity)' \
                        ' FROM \"Order Details\" JOIN Products ON
                        ↪  \"Order Details\".ProductID =
                        ↪  Products.ProductID' \
                        ' JOIN Categories ON Products.CategoryID
                        ↪  = Categories.CategoryID ' \
```

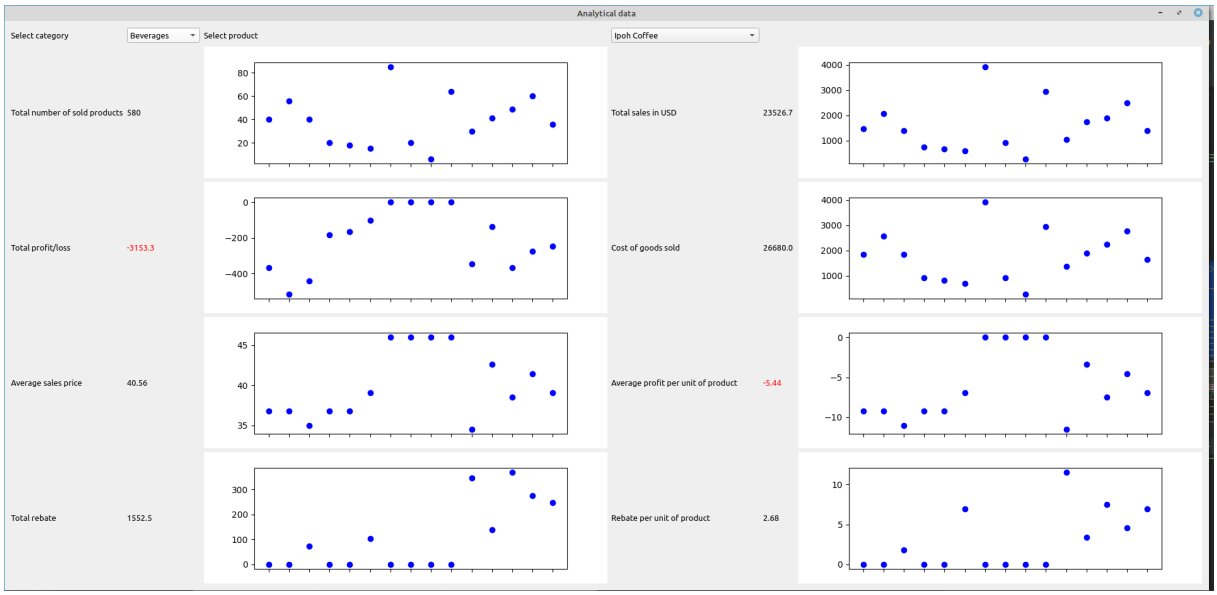
```
                    f' WHERE Categories.CategoryName =
                    ↪  \"{category_name}\"'

        else:
            get_prods = 'SELECT ((SUM((\"Order
            ↪  Details\".UnitPrice * Quantity) - (\"Order
            ↪  Details\".UnitPrice * Quantity) * Discount)) -
            ↪  SUM(Products.UnitPrice * Quantity)) /
            ↪  SUM(Quantity)' \
                    ' FROM \"Order Details\" JOIN Products ON
                    ↪  \"Order Details\".ProductID =
                    ↪  Products.ProductID' \
                    ' JOIN Categories ON Products.CategoryID
                    ↪  = Categories.CategoryID '
    else:
        get_prods = 'SELECT ((SUM((\"Order Details\".UnitPrice *
        ↪  Quantity) - (\"Order Details\".UnitPrice * Quantity)
        ↪  * Discount)) - SUM(Products.UnitPrice * Quantity)) /
        ↪  SUM(Quantity)' \
                ' FROM \"Order Details\" JOIN Products ON
                ↪  \"Order Details\".ProductID =
                ↪  Products.ProductID' \
                f' WHERE Products.ProductName =
                ↪  \"{product_name}\"'
    fetched_ord = []
    for row in cur.execute(get_prods):
        fetched_ord.append(row[0])
    try:
        return str(round(float(fetched_ord[0]), 2))
    except Exception:
        return "N/A"
```

U gornjem kodu vidimo način na koji se izračunava prosječan profit za proizvod ili grupu proizvoda (ovisno o odabiru iz sučelja). Izračuna se ukupna prodajna cijena na koju se izračuna popust te se oduzme nabavna cijena i podijeli se s ukupnim brojem prodanih proizvoda. Ovisno o korisnikovoj selekciji upit se malo modificira (može tražiti sve podatke, određenu kategoriju ili određeni proizvod).

Funkcija za dohvaćanje podataka po vremenskom periodu je dosta slična uz preinaku što se tu grupiraju vrijednosti po vremenskim periodima.

```
get_prods = 'SELECT strftime(\'%Y\', OrderDate) || \'/\' ||
↪  strftime(\'%m\', OrderDate), SUM((\"Order Details\".UnitPrice *
↪  Quantity) - (\"Order Details\".UnitPrice * Quantity) * Discount)
↪  / SUM(Quantity)' \
                                ' FROM \"Order Details\" JOIN Products ON
                                 ↪  \"Order Details\".ProductID =
                                 ↪  Products.ProductID' \
                                ' JOIN Orders ON  \"Order
                                 ↪  Details\".OrderID = Orders.OrderID' \
                                ' JOIN Categories ON Products.CategoryID
                                 ↪  = Categories.CategoryID ' \
                                f' WHERE Categories.CategoryName =
                                 ↪  \"{category_name}\"' \
                                '  GROUP BY 1' \
                                '  ORDER BY 1'
```

Nakon što imamo podatke i sučelje možemo pokrenuti aplikaciju da vidimo kako izgleda sam prikaz.



Slika 19: Prikaz analitičkih podataka

4.12. Integracija grafičkog prikaza mjera u sustav

Kako bismo dodali komponentu prikaza grafičkih podataka u sustav te provjerili kako se ponaša u međusobnom djelovanju s ostalim komponentama trebamo kreirati detaljan scenarij preko kojega ćemo testirati. Scenarij uključuje sve dosadašnje komponente te testira djelovanje prijašnjih komponenti na novu koju dodajemo. S obzirom da postoji velik broj izračuna u komponenti za grafički prikaz trebat ćemo testirati kako se odnose promjene u bazi podataka na promjenu podataka.

Scenarij za testiranje će biti razrađen u pet koraka nakon kojih ide provjera vrijednosti. Koraci prema kojima će se provoditi testiranje su sljedeći:

- Korak 1: Pročitaj inicijalne vrijednosti (za kvantitetu, ukupnu zaradu i cogs, ostale su vrijednosti izvedene iz ovih pa ih nije potrebno, dodatno trebamo dohvatiti nabavnu cijenu proizvoda kojeg ćemo kasnije dodavati)
- Korak 2: Dodati novi redak u tablicu order detail (nabavna cijena proizvoda kojeg dodajemo je dohvaćena u prošlom koraku) te pročitamo vrijednosti koje su se dogodile nakon dodavanja (kvantiteta, cogs, ukupna zarada) te uzmemo vrijednost koja je trenutno na ekranu za kvantitetu (provjeravamo ažurira li se vrijednost u labelima unutar screena)
- Korak 3: Ažurira se prethodno dodani redak i opet se čitaju vrijednosti koje će se kasnije provjeravati
- Korak 4: Izbriše se redak dodan u koraku 2 te se ponovno uzmu vrijednosti nakon brisanja
- Korak 5: Provjerava se ispravnost uzetih podataka. Provjere uključuju ispravno kretanje izračunatih vrijednosti nakon svakog koraka (kretanje cogs, kvantiteta i ukupna zarada) te se provjerava ispravnost podataka prikazanih na ekranu.

```
@pytest.mark.integration_test
def test_analytics_integration(app, main_menu, cust, my_singleton,
 ↪  order, add_orders, analytics):
    #read initial values for beverages category
    QTest.mouseClick(main_menu.btn_analytics, Qt.LeftButton)
    initial_qty = int(my_singleton.calculate_qty_sold("Beverages",
     ↪  "All"))
    initial_cogs =
     ↪  float(my_singleton.calculate_total_cogs("Beverages", "All"))
    initial_total_price =
     ↪  float(my_singleton.calculate_total_price("Beverages", "All"))
    uprice_of_product = my_singleton.get_product_price_by_id(1)

    #add new order detail row with product from beverages category
    add_orders.order_det_ord_id_tb.setText("10253")
    add_orders.prod_det_ord_id_tb.setText("1")
    add_orders.uprice_det_ord_tb.setText("23.50")
    add_orders.qty_tb.setText("2")
    add_orders.dsc_tb.setText("0.10")
    QTest.mouseClick(add_orders.button_add_order_details,
     ↪  Qt.LeftButton)
    after_adding_qty =
     ↪  int(my_singleton.calculate_qty_sold("Beverages", "All"))
    after_adding_cogs =
     ↪  float(my_singleton.calculate_total_cogs("Beverages", "All"))
```

```
after_adding_total_price =
 ↪  float(my_singleton.calculate_total_price("Beverages", "All"))
analytics.select_category.setCurrentIndex(1)
screen_qty_1 = int(analytics.total_products_sold.text())

#update row that was added before, row is added before other
 ↪  order details from same order
order.table_widget.item(8, 19).setText("25.50")
order.table_widget.item(8, 20).setText("3")
order.table_widget.item(8, 21).setText("0.2")
my_singleton.update_ord_details("10253", "Chai", "25.50", "3",
 ↪  "0.2")
after_updating_qty =
 ↪  int(my_singleton.calculate_qty_sold("Beverages", "All"))
after_updating_cogs =
 ↪  float(my_singleton.calculate_total_cogs("Beverages", "All"))
after_updating_total_price =
 ↪  float(my_singleton.calculate_total_price("Beverages", "All"))

#delete created row
my_singleton.delete_order_details("10253", "1")
after_deleting_qty =
 ↪  int(my_singleton.calculate_qty_sold("Beverages", "All"))
after_deleting_cogs =
 ↪  float(my_singleton.calculate_total_cogs("Beverages", "All"))
after_deleting_total_price =
 ↪  float(my_singleton.calculate_total_price("Beverages", "All"))

#Analyze on screen values
analytics.select_category.setCurrentIndex(2)
analytics.select_category.setCurrentIndex(1)
screen_qty_2 = int(analytics.total_products_sold.text())
screen_cogs = float(analytics.cogs.text())
screen_total_price = float(analytics.total_sales_number.text())

assert (after_adding_qty - 2 == initial_qty and
        after_adding_cogs - (uprice_of_product * 2) ==
         ↪  initial_cogs and
        round(after_adding_total_price - ((23.5 * 2) - (23.5 * 2
         ↪  * 0.1)), 2) == initial_total_price and
        after_updating_qty - 3 == initial_qty and
        after_updating_cogs - (uprice_of_product * 3) ==
         ↪  initial_cogs and
```

```
        round(after_updating_total_price - ((25.5 * 3) - (25.5 *
        ↪  3 * 0.2)), 2) == initial_total_price and
        after_deleting_qty == initial_qty and
        after_deleting_cogs == initial_cogs and
        after_deleting_total_price == initial_total_price and
        screen_qty_2 == initial_qty and
        screen_qty_1 == after_adding_qty and
        screen_cogs == initial_cogs and
        screen_total_price == initial_total_price)
```

Testiranjem po ovim koracima prolazimo kroz sve komponente te svaku komponentu uspoređujemo na utjecaj koji ima na novu komponentu. Kod integracijskog testiranja nam je važno da iako jedinično testirana komponenta prikaza podataka radi sama za sebe da uspješno testiramo kako se promjene napravljene na drugim dijelovima aplikacije odražavaju na novu komponentu.

4.13. Testiranje ponašanja pomoću Gherkin i Behave biblioteka

Kao što je spomenuto u teoretskom dijelu testiranje ponašanja nam služi kako bismo na lakši način poslovne zahtjeve preveli u testiranja. Naime za pisanje feature datoteke je potrebno minimalno tehničko znanje (samo paziti na redoslijed oznaka), a mogu se definirati neki slučajevi korištenja koje onda developer pokriva testovima. Poslovni odjel može korak po korak raspisati željeno ponašanje aplikacije kroz scenarije te tako osigurava da se njihovo razmišljanje nije izgubilo u komunikaciji s developerom koji također dobiva preciznije upute nego samo kroz razgovor.

S obzirom na to da više timova radi na testovima ponašanja nije neobično da se prvo definiraju svi scenariji pa se zatim kreće u izradu testova. Scenariji su obuhvaćeni u nadobjekt *feature*.

Prvi scenarij koji je definiran testirat će osnovno ponašanje komponente za prikaz grafičkih podataka. Koraci će se sastojati od navigacije do ekrana za prikaz podataka, odabira kategorija i proizvoda iz padajućih izbornika i testiranja vrijednosti prikazanih podataka. Na ovakvom primjeru može se vidjeti redoslijed obrađivanja oznaka.

```
Feature: Check analytic data calculations

  Scenario: check analytic data values for beverages product category
     Given analytics screen is selected from main menu
      When select beverages in category combo box and all in product
       ↪  combo box
      Then calculations should match data in DB and label should be
       ↪  in right colors
```

Testovi koji će koristiti stepove moraju imati tzv. decorator (anotacija @) i string s opisom koraka kako bi behave automatski prepoznao na koji se korak odnosi. Kako bi mogao uspješno pretražiti korake, *feature file* moramo staviti u direktorij feature/steps. Test za given će biti klasičan test navigacije, test za when nam uključuje odabir vrijednosti iz izbornika te provjeru jesu li vrijednosti ispravno učitane, a posljednji then test će provjeriti jesu li kalkulacije ispravno prenesene iz baze i radi li provjera za bojanje labela za profit/gubitak (ako je profit negativan vrijednosti se ispisuju crveno).

```
@given('analytics screen is selected from main menu')
def step_impl(context):
    MainMenu = main.MainMenu()
    QTest.mouseClick(MainMenu.btn_analytics, Qt.LeftButton)
    is_activated = MainMenu.a_activated
    assert is_activated is True

@when('select beverages in category combo box and all in product combo box')
def step_impl(context):
    MainMenu = main.MainMenu()
    analytics_screen = Analytics.Analytics()
    analytics_screen.select_category.setCurrentIndex(1)
    time.sleep(2)
    analytics_screen.select_product.setCurrentIndex(0)
    time.sleep(1)
    text_curr = analytics_screen.select_category.currentText()
    text_prod = analytics_screen.select_product.currentText()
    assert text_curr == "Beverages" and text_prod == "All"

@then('calculations should match data in DB and label should be in right colors'
def step_impl(context):
    MainMenu = main.MainMenu()
    analytics_screen = Analytics.Analytics()
    analytics_screen.select_category.setCurrentIndex(1)
    analytics_screen.select_product.setCurrentIndex(0)
    total_sale = float(analytics_screen.total_sales_number.text())
    cogs = float(analytics_screen.cogs.text())
    total_profit = float(analytics_screen.total_profit.text())
    qty = int(analytics_screen.total_products_sold.text())
    avg_prof = float(analytics_screen.avg_profit.text())
    avg_price = float(analytics_screen.avg_price.text())
    tot_rebate = float(analytics_screen.total_rebate.text())
    avg_rebate = float(analytics_screen.rebate_per_product.text())
    color_1 = str(analytics_screen.total_profit.styleSheet()).strip()
    color_2 = str(analytics_screen.avg_profit.styleSheet()).strip()
```

```
    assert total_profit == round((total_sale - cogs), 2) \
           and avg_prof == round((total_profit / qty), 2) \
           and avg_price == round((total_sale / qty), 2) \
           and avg_rebate == round((tot_rebate / qty), 2) \
           and color_1 == "color: red" \
           and color_2 == "color: red"
```

Drugi test scenario će služiti da se prikaže na koji način se koristi definiranje parametara unutar Gherkin *feature file-a*. Provjeravamo različite kombinacije vrijednosti i kako one utječu na sam prikaz. Želimo testirati da svaki put kada korisnik odabere drugu vrijednost u izbornicima product i category da se promijene vrijednosti na prikazu. Također želimo provjeriti računaju li se ispravno brojčane vrijednosti za svaki od tih kombinacija (i prikazuju ispravnom bojom). Trebamo paziti na način zapisivanja parametara unutar datoteke (poravnanja i imena stupaca).

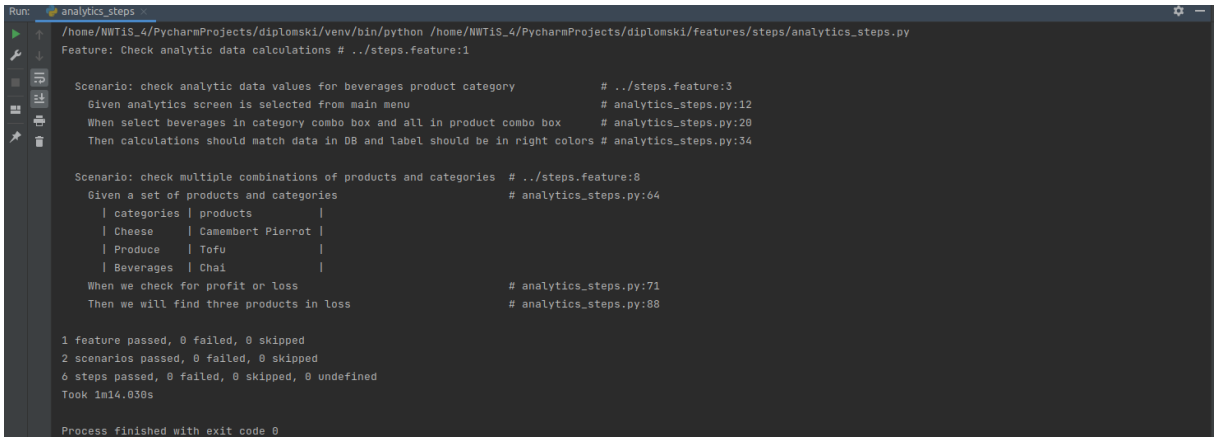
```
Scenario: check multiple combinations of products and categories
  Given a set of products and categories
     | categories | products         |
     | Cheese     | Camembert Pierrot|
     | Produce    | Tofu             |
     | Beverages  | Chai             |

 When we check for profit or loss
 Then we will find three products in loss
```

Testovima prvo provjeravamo učitavaju li se ispravno iz *feature file* vrijednosti (tretira li se npr. prvi redak kao zaglavlje). Zatim testiramo koliko puta se promijene vrijednosti ispisane na ekranu (trebale bi se promijeniti za svaku od kombinacija). Na kraju provjerimo (s obzirom na podatke u bazi) računaju li se mjere ispravno za naše testne parametre.

```
@given('a set of products and categories')
def step_impl(context):
    for row in context.table:
        test_data.append((row['products'], row['categories']))
    assert len(test_data) == 3

@when('we check for profit or loss')
def step_impl(context):
    values_changed = 0
    last_value = -99
    analytics_screen = Analytics.Analytics()
    for test_row in test_data:
        analytics_screen.select_category.setCurrentText(test_row[1])
        time.sleep(2)
        analytics_screen.select_product.setCurrentText(test_row[0])
        time.sleep(1)
        total_profit = float(analytics_screen.total_profit.text()
```

```
        if total_profit != last_value: values_changed += 1
        last_value = total_profit
    assert values_changed == 3

@then('we will find three products in loss')
def step_impl(context):
    number_of_losses = 0
    analytics_screen = Analytics.Analytics()
    for test_row in test_data:
        analytics_screen.select_category.setCurrentText(test_row[1])
        analytics_screen.select_product.setCurrentText(test_row[0])
        total_profit = float(analytics_screen.total_profit.text())
        if total_profit < 0: number_of_losses += 1

    assert number_of_losses == 3
```

Zgodan dodatak behave biblioteci je jako informativan način ispisa. Naime *feature file* se obrađuju korak po korak nakon pozivanja testova te se za svaki korak, scenarij ili *feature* zapisuje je li prošao, nije izvršen ili nije uspješan. Jedan neuspješan utječe da idući koraci neće biti izvršeni, a samim time taj scenarij i feature neuspješni. Prilikom izvršavanja se ispisuje korak koji se trenutno izvršava sa svim informacijama iz *feature file* (tako možemo vidjeti ako je nešto unutar samog feature file definirano).



Slika 20: Prikaz ispisa u terminalu testiranje ponašanja s Behave bibliotekom

4.14. Testiranje unutar dokumentacije

Korištenjem mogućnosti da se unutar dokumentacije kreiraju testovi napravit ćemo funkcije za testiranje ispravnosti podataka prilikom dodavanja proizvoda i stavki narudžbe. Funkcija za dodavanje podataka kao i ta dva entiteta su nam pogodni za ovakav oblik testiranja jer imaju velik broj parametara i provjera ispravnosti podataka.

Prvo ćemo kreirati skup testova za funkciju dodavanja proizvoda. Prvi korak je da testiramo jesu li podaci zadani u ispravnom formatu (inače nam ostali testovi neće ispravno raditi). Kvantiteta, broj zaliha i broj naručenih jedinica moraju biti cjelobrojne vrijednosti, a jedinična cijena mora biti decimalna vrijednost. Testirat ćemo da kvantiteta, broj zaliha i cijena ne mogu biti negativni brojevi. Zatim testiramo vrijednost vanjskih ključeva (postoje li u bazi) za dobavljače i narudžbe. Na kraju dodajemo test za potencijalni "preljev" vrijednosti (eng. *overflow*) kako bi vrijednosti koje dobivamo bile unutar raspona zadanog tipa podataka. Tu nam koristi to što u doctest biblioteci na vrlo jednostavan način možemo zadati koja će se greška dogoditi. Na kraju smo dodali provjeru izvršava li se funkcija ispravno sa stvarnim podacima (ne testiramo samo potencijalne greške već i ispravno ponašanje).

```
def adding_product_with_testable_doc(product_name, supplier_id,
↪  category_id, qty_per_unit, unit_price, units_in_stock,
↪  units_in_order):
        """
        >>> adding_product_with_testable_doc("Some name", 0, 0, -1,
↪  -1, -1, -1)
        Traceback (most recent call last):
            ...
        ValueError: qty_per_unit, unit_price, units_in_stock and
↪  units_in_order must be >= 0

        >>> adding_product_with_testable_doc("Some name", 0, 0, 1,
↪  "a", 3, 4)
        Traceback (most recent call last):
            ...
        ValueError: Quantity, Unit in stock and order should be
↪  numeric values and unit price decimal value

        >>> adding_product_with_testable_doc("Some name", 0, 0, 1,
↪  1.3, 3, 4)
        Traceback (most recent call last):
            ...
        ValueError: CategoryID should exist in category table

        >>> adding_product_with_testable_doc("Some name", 0, 1, 1,
↪  1.3, 3, 4)
```

```
        Traceback (most recent call last):
            ...
        ValueError: SupplierID should exist in suppliers table

        >>> adding_product_with_testable_doc("Some name", 1, 1, 1,
↪   1.3, 33333, 444444)
        Traceback (most recent call last):
            ...
        OverflowError: Units in stock and units in order can go up to
↪   32767

        >>> adding_product_with_testable_doc("Some name", 1, 1, 1,
↪   1.3, 35, 44)
        'New row was added'
        """
    s = db_singleton.Singleton.get_instance()
    try:
        qty_per_unit = int(qty_per_unit)
        units_in_stock = int(units_in_stock)
        units_in_order = int(units_in_order)
        unit_price = float(unit_price)
    except ValueError:
        raise ValueError("Quantity, Unit in stock and order should be
        ↪  numeric values and unit price decimal value")
    if qty_per_unit < 0 or unit_price < 0 or units_in_stock < 0 or
    ↪  units_in_order < 0:
        raise ValueError("qty_per_unit, unit_price, units_in_stock
        ↪  and units_in_order must be >= 0")
    cat = s.get_category_by_id(category_id)
    if cat == 0:
        raise ValueError("CategoryID should exist in category table")
    sup = s.get_supplier_by_id(supplier_id)
    if sup == 0:
        raise ValueError("SupplierID should exist in suppliers
        ↪  table")
    if units_in_stock > 32767 or units_in_order > 32767:
        raise OverflowError("Units in stock and units in order can go
        ↪  up to 32767")

    result = s.insert_new_product_test_doc(product_name, supplier_id,
    ↪  category_id, qty_per_unit, unit_price, units_in_stock,
                                    units_in_order)
    return result
```

Kako smo već unutar dokumentacije funkcije odradili provjeru unosa podataka možemo izbjeći te iste provjere prilikom unosa podataka u bazu. Naime, umjesto da sve provjere ponovno prolazimo dosta je samo proslijediti vrijednosti u SQL naredbu te je izvršiti. Tako smanjujemo implementaciju samog dodavanja u bazu jer odrađujemo provjere van podatkovnog dijela aplikacije.

Drugi test će biti za dodavanje stavke narudžbe. Kod stavke trebamo provjeriti tipove podataka za vrijednosti jedinične cijene, kvantitete i popusta. Zatim provjerimo je li kvantiteta i jedinična cijena vrijednost veća od 0, je li vrijednost popusta u rasponu od 0 do 1 i postoje li vanjski ključevi na tablice order i product. Također na kraju provjeravamo ispisuje li se ispravna poruka nakon uspješnog dodavanja novog retka.

```
def adding_orders_with_testable_doc(order_id, product_id, unit_price,
↪  quantity, discount):
    """
        >>> adding_orders_with_testable_doc(0, 0, "wrong value",
↪  "wrong value", -1)
        Traceback (most recent call last):
            ...
        ValueError: Quantity should be numeric value and unit price
↪  and discount decimal value

        >>> adding_orders_with_testable_doc(0, 0, 0.0, 0, 4.2)
        Traceback (most recent call last):
            ...
        ValueError: Quantity and unit price must be > 0

        >>> adding_orders_with_testable_doc(0, 0, 1.1, 2, 4.2)
        Traceback (most recent call last):
            ...
        ValueError: Discount should be between 0 and 1

        >>> adding_orders_with_testable_doc(0, 0, 1.1, 2, 0.1)
        Traceback (most recent call last):
            ...
        ValueError: ProductID should exist in product table

        >>> adding_orders_with_testable_doc(0, 1, 1.1, 2, 0.1)
        Traceback (most recent call last):
            ...
        ValueError: OrderID should exist in orders table

        >>> adding_orders_with_testable_doc(10251, 1, 1.1, 2, 0.1)
        'New row was added'
```

```
    """
    s = db_singleton.Singleton.get_instance()
    try:
        quantity = int(quantity)
        unit_price = int(unit_price)
        discount = float(discount)
    except ValueError:
        raise ValueError("Quantity should be numeric value and unit
        ↪  price and discount decimal value")
    if quantity <= 0 or unit_price <= 0:
        raise ValueError("Quantity and unit price must be > 0")
    if discount < 0 or discount > 1:
        raise ValueError("Discount should be between 0 and 1")
    prod = s.get_product_by_id(product_id)
    if prod == 0:
        raise ValueError("ProductID should exist in product table")
    orders = s.get_order_by_id(order_id)
    if orders == 0:
        raise ValueError("OrderID should exist in orders table")

    result = s.insert_new_ord_details_test_doc(order_id, product_id,
    ↪  unit_price, quantity, discount)
    return result
```

Pokretanjem *doctesta* iz naredbenog retka (ili iz IDE-a) ne dobivamo nikakvu povratnu informaciju (osim *exit code-a*) u slučaju ispravnog izvršavanja dok u slučaju pogreške se ispisuje log koji daje informaciju koji test nije prošao i koja je vrijednost dobivena umjesto tražene.

4.15. Automatizirano testiranje pomoću Robot Frameworka

Kako je spomenuto u uvodnom dijelu Robot framework se često koristi za automatizaciju testova prihvatljivosti. S obzirom na visoku razinu apstrakcije s kojom se piše sadržaj pogodan je za korištenje vanjskim sudionicima procesa razvoja softwarea, a opet omogućuje automatizaciju testiranja.

S obzirom na to da Robot Framework omogućava više različitih sintaksi za pisanje testne specifikacije moramo se odlučiti koju ćemo koristiti. Najjednostavnija i ona s kojom smo se već upoznali je Gherkin format (iako nisu ni drugi kompliciraniji) koji je malo prilagođen za potrebe Robot-a (postoje neki dodaci u odnosu na behave). Kreirat ćemo tri testa koji će testirati izračune mjere za pojedine kategorije (brojčani izračuni su nam pogodni za testiranja). Prvi test će gledati kvantitetu prodanih jedinica za prvi proizvod, drugi test će gledati ukupan rabat za cijelu kategoriju, a treći test će izračunati rabat po jedinici proizvoda te usporediti s vrijednošću u bazi. Kod rada s Robot Frameworkom najvažnije nam je paziti na sintaksu i na ispravno pisanje imena klasa i ključnih riječi.

Prvi dio .robot datoteke (nema ekstenziju .feature kao u primjeru s Gherkin datotekom i behave bibliotekom jer se samo koraci zadaju u Gherkin formatu, ostalo je specifično za Robot) je Settings odjeljak. Unutar ovog odjeljka se piše dokumentacija (tako da se slijeva napiše Documentation, a sa zdesna tekst dokumentacije). Također jako bitan dio Settings odjeljka je dio gdje se definiraju biblioteke unutar kojih će se tražiti definicije testova. Kraj oznake napišemo ime naše datoteke s testovima (u ovom slučaju RobotTestingData.py). Sljedeći odjeljak koji će biti korišten je odjeljak s testnim slučajevima. Unutar oznake *test cases* pišemo testove tako da bez uvlake napišemo ime testa (npr. Quantity), a onda s uvlakom napišemo testne korake u standardnoj Gherkin sintaksi. Ako želimo nešto koristiti kasnije kao vrijednost napišemo unutar navodnih znakova. Tako možemo pisati više testova jedan ispod drugog samo pazeći na uvlake. Posljednji dio .robot datoteke je sekcija s ključnim riječima *keywords*) koja može biti pomalo konfuzna na prvom korištenju. Naime *keywords* odjeljak će u praksi komunicirati i s testovima koje ćemo pisati i s koracima koje smo definirali (mediator). Bez uvlake se napiše dio teksta koji se nalazi u definiciji koraka (bez taga koraka, npr. given) bez potrebe za pažnjom prema velikim i malim slovima. U drugom retku se nalazi poziv prema testovima (uvučen kako bi se znalo kojem koraku pripada) se napiše ime metoda koja sadrži test (uvažava se razmak, razlika u malim i velik slovima) i s tabom razmaka se piše parametar koji se proslijedi funkciji. Ako je parametar sadržan unutar koraka onda se pomoću oznake dolara definira da je riječ o parametru.

```
*** Settings ***
Documentation     Example test case using the gherkin syntax.
...
...               Tests include checking if calculations are correct and in
...               cooperation with other parts of application.
Library           RobotTestingData.py

*** Test Cases ***
Quantity
    Given data is loaded
```

```
    When user chooses beverages
    and user select chai
    Then quantity is "830"
TotalRebate
    Given data is loaded
    When user chooses Produce
    and user select All
    Then total rebate is "5284.02"
AverageRebate
    Given data is loaded
    When user chooses Seafood
    and user select Ikura
    Then average rebate is "1.72"

*** Keywords ***
Data is loaded
    Check load data    order_details
User chooses Beverages
    Choose category    Beverages
User select Chai
    Choose product     Chai
Quantity is "${quantity}"
    Quantity should be    ${quantity}
User chooses Produce
    Choose category    Produce
User select All
    Choose product     All
Total rebate is "${totalrebate}"
    Totalrebate should be    ${totalrebate}
User chooses Seafood
    Choose category    Seafood
User select Ikura
    Choose product     Ikura
Average rebate is "${averagerebate}"
    Averagerebate should be    ${averagerebate}
```

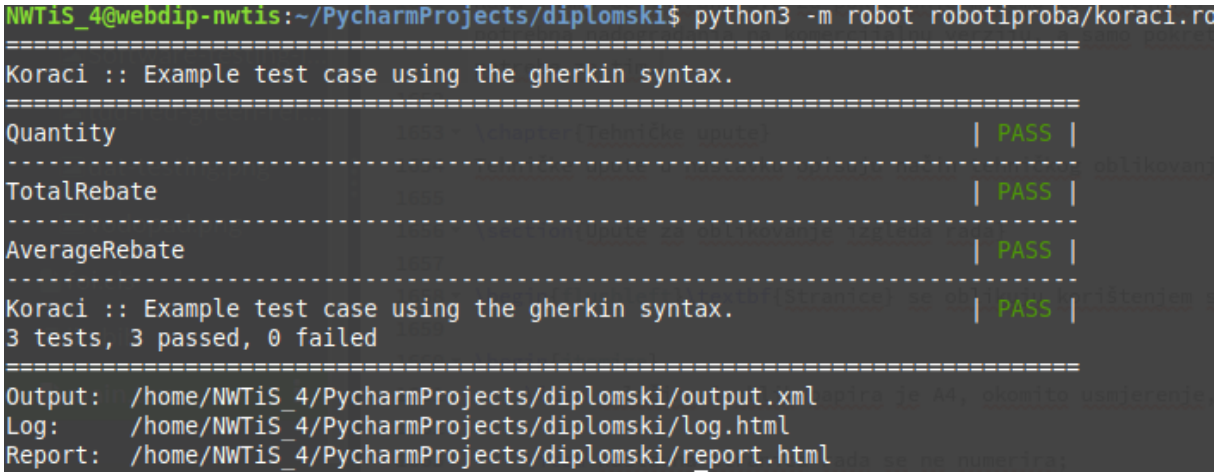
Sada kada imamo datoteku po kojoj trebamo napraviti testove možemo krenuti od kreiranja same klase koja će provoditi testiranje. Kod definiranja moramo paziti da ime klase mora biti isto kao i ime biblioteke koja je definirana u *library* sekciji (za Python obično nije praksa dok kod Jave se ovakav pristup podrazumijeva). Kod metoda treba pripaziti da im se imena poklapaju s imenima zadanim u datoteci s koracima. Također bilo bi poželjno kada bi se unutar komentara dali primjeri za korištenje s testnom datotekom. Sami testovi rade na principu da dok se ne podigne greška sve je u redu. Dok u koracima koje prije radimo može doći do krivo zadane vrijednosti pa se digne ValueError što uzrokuje prekidanjem rada, a u konačnom testu imamo provjeru ispravnosti rezultata.

```
class RobotTestingData(object):
    def __init__(self):
        self._s = db.Singleton.get_instance()
        self._result = ''
```

```
        self._category = ''
        self._product = ''
    def check_load_data(self, table_name):
        """Checks if 'table_name' has loaded rows.
        The given value is passed to the db object directly.
        Examples:
        | Check load data | order_details |
        | Check load data | customers |
        """
        if table_name == "order_details":
            result_list = self._s.get_all_order_details()
            if len(result_list) < 1:
                raise ValueError("Table with given name has no data
                ↪  or doesn't exist.")
        else:
            raise ValueError("Not implemented yet")

    def choose_category(self, category_name):
        """Chooses specific 'category_name' from display
            Example:
            | Choose category | Beverages |
        """
        self._category = category_name

    def choose_product(self, product_name):
        """Chooses specific 'product_name' from display
            Example:
            | Choose product | Chai |
        """
        self._product = product_name

    def quantity_should_be(self, expected):
        """Verifies that the current quantity is 'expected'.

        Example:
        | Choose product     | Chai |
        | Result Should Be | 830       |
        """
        self._result = self._s.calculate_qty_sold(self._category,
        ↪  self._product)
        if self._result != expected:
            raise AssertionError('%s != %s' % (self._result,
            ↪  expected))
```

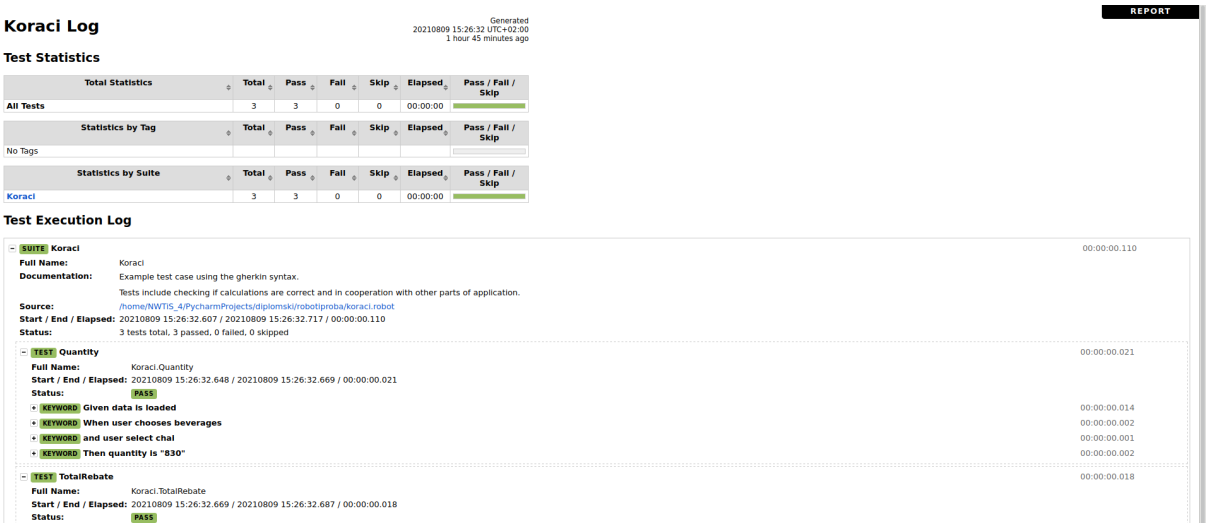
```
def totalrebate_should_be(self, expected):
    """Verifies that the current total rebate is 'expected'.

    Example:
    | Choose category    | Cheese |
    | Choose product     | All |
    | Result Should Be | 830       |
    """
    self._result = self._s.calculate_total_rebate(self._category,
    ↪  self._product)
    if self._result != expected:
        raise AssertionError('%s != %s' % (self._result,
        ↪  expected))

def averagerebate_should_be(self, expected):
    """Verifies that the average rebate is 'expected' as total
    ↪  rebate / qunatity.

    Example:
    | Choose category    | Seafood |
    | Choose product     | Ikura |
    | Result Should Be | 1.72       |
    """
    total_rebate_prod =
    ↪  float(self._s.calculate_total_rebate(self._category,
    ↪  self._product))
    qty_sold_prod =
    ↪  int(self._s.calculate_qty_sold(self._category,
    ↪  self._product))
    if qty_sold_prod == 0:
        raise ZeroDivisionError('Division by zero.')
    res = str(round((total_rebate_prod / qty_sold_prod), 2))
    if res != expected:
        raise AssertionError('%s != %s' % (res, expected))
```

Kada imamo definirano sve za pokretanje možemo to i učiniti iz naredbenog retka. Za pokretanja iz IDE-a je potrebna nadogradnja na komercijalnu verziju, a samo pokretanje iz terminala je jednostavne i brže pa nema potrebe za tim.

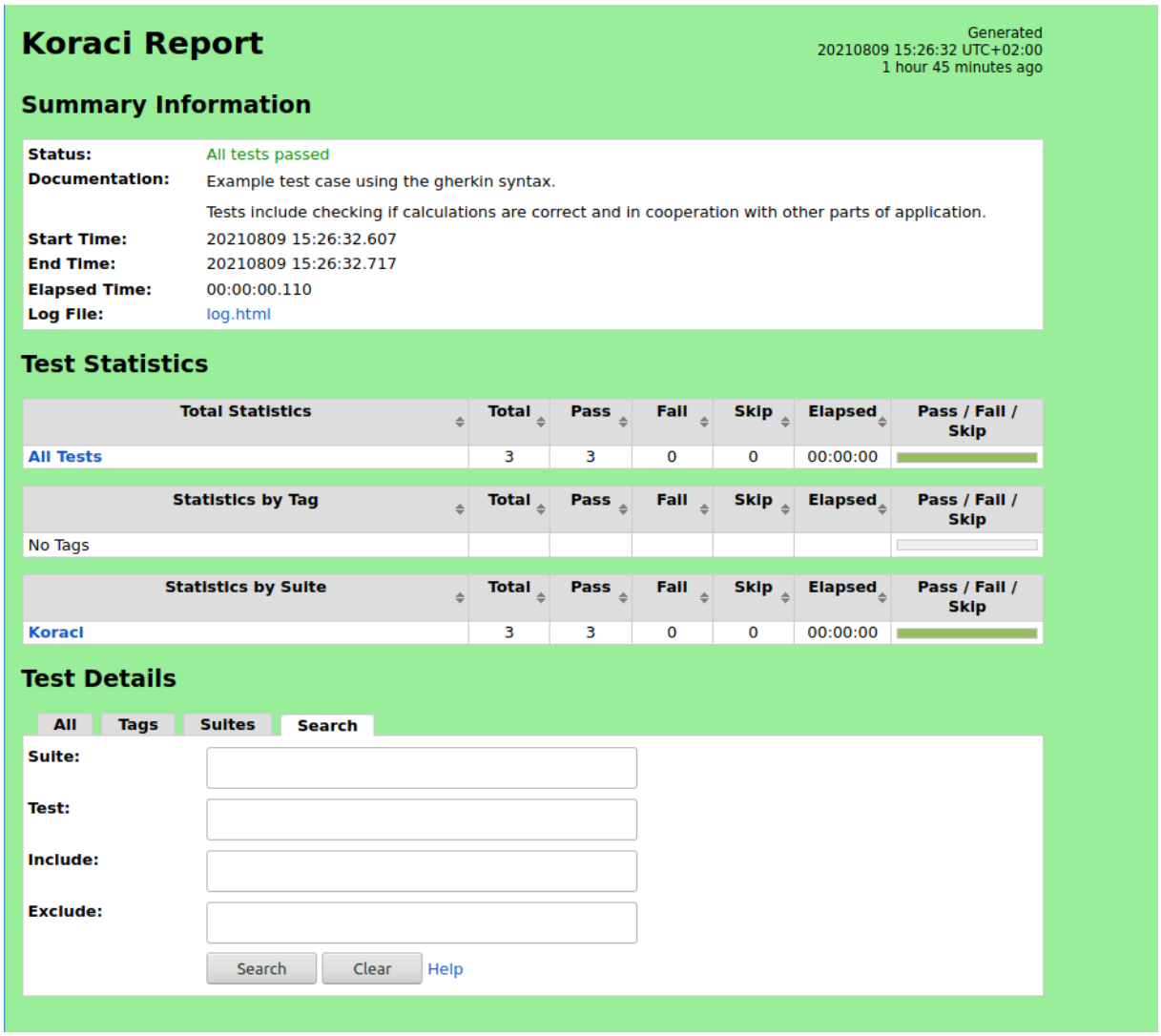


Slika 21: Prikaz ispisa u terminalu prilikom testiranja Robot frameworkom

Kao što vidimo u ispisu testovi prolaze, a ispisuju se i output, log i report podaci. Output je xml datoteke, log i report html. Datoteke se kreiraju u direktoriju iz kojeg pokrećemo testove. Output datoteka sadrži pregled koraka s početnim vremenom i završnim vremenom te s podacima o uspješnom i neuspješnom izvršavanju (logovi). Report i log daju grafički prikaz rezultata testiranja. Log datoteka sadrži više podataka (primarno za developera, dok je report lakše čitljiv (može ga i korisnik shvatiti).



Slika 22: Prikaz log datoteke prilikom testiranja Robot frameworkom



Slika 23: Prikaz report datoteke prilikom testiranja Robot frameworkom

5. Zaključak

Testiranje kao grana i dio ciklusa razvoja programskog proizvoda nije jednoznačan pojam. Kao što se moglo vidjeti u prethodnim primjerima to da je neka komponenta programskog proizvoda testirana ne specificira što je i na koji način testirano (npr. web aplikacija može biti savršeno implementirana ako popratna arhitektura ne zadovoljava *stress* testove aplikacija ne radi). Razvoj testova mora biti sastavni dio kreiranja programa, ali osim developera i testera na njemu moraju sudjelovati i ljudi koji se bave poslovnom problematikom programskog rješenja (kako bi se sve komponente obuhvatile). Isto tako očito je kako ne postoji savršeno i jedinstveno uputstvo za provođenje testiranja nego da je potrebno koristiti više praksi i metodologija kako bi se najbolje prilagodili programu kojeg testiramo. Koncepti testiranja i podjele (na razine testiranja i na uvid u programski kod) ne djeluju jedni protiv drugih već djeluju zajedno u svrhu kreiranja sigurnog i stabilnog programskog proizvoda (npr. integracijsko i jedinično testiranje nisu dva različita pristupa već su jedan pristup koji se međusobno slaže od najmanje do najveće cjeline).

Metodologija testiranja definira različite načine na koje bi se trebalo ukomponirati testiranja u životni ciklus razvoja programa. Svaki od metodologija ima svoje prednosti i nedostatke te će se rijetko u praksi naći primjer gdje se doslovno može implementirati neki princip te tako riješiti problem testiranja koda. Uzme li se TDD za primjer tada možemo reći da stvarno kreiranjem testova za sve potencijalne scenarije te komponente te zatim implementacijom da se zadovolje testovi koji su definirani te naposljetku eliminiranjem nepotrebnog koda (koji ne služi za prolazak testa) dobivamo optimiziran i siguran kod koji zadovoljava uvjete (ako su testovi dobro definirani). Ali u praksi takav pristup možda i nije najprimjenjiviji (barem ne u svom doslovnom obliku) jer najveći protivnik testiranju i razvoju stabilnog proizvoda je manjak vremena. Kada bismo maksimalno puristički pristupili TDD-u onda bismo trošili jako puno vremena na definiranje svih mogućih scenarija i svih testova koji će se napraviti. Nakon toga moramo implementirati sve te dijelove koji zadovoljavaju prethodne testove. Vrlo moguće da se prilikom implementacije sjetimo nekog dodatnog zahtjeva ili uvjeta te zatim moramo ponovno pisati test za taj zahtjev (već smo izašli iz okvira TDD-a jer smo iz implementacije kreirali test, a ne obrnuto). U zadnjem dijelu gdje se refaktorira kod koji nije potreban da se zadovolje testovi lako možemo izbaciti neki dio koji nije nužan, ali je koristan dodatak za UI/UX ili je dodatna funkcionalnost koja se čini korisnom. Ako želimo i dalje zadržati i te elemente i TDD ponovno moramo pisati testove. Ovakav pristup zahtjeva jako puno vremena za razvojni tim te se ne može primjenjivati u situacijama s bliskim rokom za predaju (kakvih je često). Zato je važno iz ove metodologije uzeti dobre principe pisanja jednostavnog, optimiziranog i testiranog koda, a sada hoće li se doslovno primjenjivati prema teoretskom predlošku će ovisiti od projekta do projekta.

Alati koje možemo koristiti prilikom testiranja nam olakšavaju sam proces. Kreiranje grupa testova, automatsko pronalaženje testova, automatsko pokretanje metoda kao testnih i druge funkcionalnosti koje nude sva naprednija razvojna okruženja danas ubrzavaju testiranja i smanjuju razinu tehničkog znanja potrebnu za pisanje testova. Nažalost i dalje pisanje testova moramo obavljati sami (iako će vrlo vjerojatno kroz neko dogledno vrijeme razvojem AI se moći

iz zahtjeva ili iz već implementiranog koda generirati testove), ali ako imamo dobro definirane zahtjeve i ograničenja koja je potrebno testirati tada nije problem kreirati kvalitetne testove. Razne biblioteke koje nam Python omogućuje za različite oblike testiranja su od velike pomoći te velika većina njih je kvalitetna te je stvar osobne preference koju ćemo odabrati. Unittest i PyTest biblioteke su u dosta stvari jako slične: obje su primjenjive, primarna zadaća im je jedinično testiranje, a koju ćemo izabrati čisto ovisi o projektu (PyTest nudi više mogućnosti, a unittest je brži i ne mora se posebno dohvaćati). Kako budućnost programiranja vodi prema *low code* (nudi se mogućnost automatizacije pisanjem skripti, ali većina operacije se može provesti bez potrebe za programiranjem) i *no code* (razvoj bez pisanja koda) platformi razvojni okviri poput Behave i Robot Frameworka (nude višu razinu apstrakcije, nije potreba biti developer za definirati testove) su od velikog značaja. Robot Framework pogotovo može biti jako koristan alat jer uvodi automatizaciju u testiranje prihvatljivosti bez da značajno podiže razinu tehničkog znanja potrebnog za provođenje.

Kao nekakav finalni osvrt na temu i alate korištene rekao bih da Python svakako nudi velik broj mogućnosti za testiranja svih oblika (s obzirom na to da je *community driven* postoji gomila samostalnih projekata koji se bave danom tematikom). Najveći izazov nije u samoj implementaciji testova (iako i ona može biti kompliciranа za neke veće sustave, pogotovo integracijska i sustavska testiranja) već da je u ispravnom definiranju zahtjeva i ponašanja. Smatram da bi se svaki malo bolji developer koristeći prethodno spomenutim bibliotekama mogao osigurati da mu se kod izvršava bez greške ukoliko bi dobio jasnu i detaljnu specifikaciju sustava. Baš zbog takvih slučajeva od posebnog značaja su biblioteke koje testiraju ponašanje (behave) ili koje se bave sustavom s više razine (robot framework) jer osiguravaju jednostavnu komunikaciju između timova koji definiraju zahtjeve i oni koji ih provode.

Popis literature

[1] IBM.com, „Software testing," 2021. adresa: https://www.ibm.com/topics/software-testing.

[2] V. Rastogi, „Software development life cycle models-comparison, consequences," *International Journal of Computer Science and Information Technologies*, sv. 6, br. 1, str. 168–172, 2015.

[3] RiverbankComputing, „PyQt5 5.15.4," 2021. adresa: https://pypi.org/project/PyQt5/.

[4] ——, „Qt Widgets," 2021. adresa: https://doc.qt.io/qt-5/qtwidgets-index.html.

[5] M. Fitzpatrick, „Layout management," 2020. adresa: https://www.pythonguis.com/tutorials/pyqt-layouts/.

[6] python.org, „Unit testing framework," 2021. adresa: https://docs.python.org/3/library/unittest.html.

[7] ——, „unittest.mock," 2021. adresa: https://docs.python.org/3/library/unittest.mock.html.

[8] H. Kregel, „pytest Documentation," 2021. adresa: https://buildmedia.readthedocs.org/media/pdf/pytest/latest/pytest.pdf.

[9] python.org, „Test interactive Python examples," 2021. adresa: https://docs.python.org/3/library/doctest.html.

[10] RobotFrameworkFoundation, „TRobot Framework User Guide," 2021. adresa: http://robotframework.org/robotframework/4.1/RobotFrameworkUserGuide.html.

[11] uipath.com, „What is robotic process automation?," 2021. adresa: https://www.uipath.com/rpa/robotic-process-automation.

[12] cucumber.io, „Gherkin Reference," 2021. adresa: https://cucumber.io/docs/gherkin/reference/.

[13] N. Moelholm, „Spring Boot: Gherkin tests," 2016. adresa: https://moelholm.com/blog/2016/10/15/spring-boot-gherkin-tests.

[14] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[15] G. Steinfeld, „5 steps of test-driven development," 2020. adresa: https://developer.ibm.com/articles/5-steps-of-test-driven-development/.

[16] cucumber.io, „Behaviour-Driven Development," 2021. adresa: https://cucumber.io/docs/bdd/.

[17] S. Acharya i V. Pandya, „Bridge between Black Box and White Box–Gray Box Testing Technique," *International Journal of Electronics and Computer Science Engineering*, sv. 2, br. 1, str. 175–185, 2012.

[18] M. E. Khan, F. Khan i dr., „A comparative study of white box, black box and grey box testing techniques," *Int. J. Adv. Comput. Sci. Appl*, sv. 3, br. 6, 2012.

[19] WhiteHackLabs, „Penetration Testing Black Box vs White Box," 2019. adresa: https://whitehacklabs.com/penetration-testing-types-black-box-vs-white-box/.

[20] professionalqa.com, „Software Testing Levels," 2019. adresa: https://www.professionalqa.com/levels-of-testing.

[21] guru99.com, „Unit Testing Tutorial: What is, Types, Tools and Test EXAMPLE," 2021. adresa: https://www.guru99.com/unit-testing-guide.html.

[22] ——, „Integration Testing: What is, Types, Top Down and Bottom Up Example," 2021. adresa: https://www.guru99.com/integration-testing.html.

[23] ——, „What is System Testing? Types and Definition with Example," 2021. adresa: https://www.guru99.com/system-testing.html.

[24] ——, „What is User Acceptance Testing (UAT)? with Examples," 2021. adresa: https://www.guru99.com/user-acceptance-testing.html.

[25] softwaretestinghelp.com, „What Is The Difference Between SIT Vs UAT Testing?," 2021. adresa: https://www.softwaretestinghelp.com/sit-vs-uat/.

[26] J. White, „Northwind-SQLite3," 2021. adresa: https://github.com/jpwhite3/northwind-SQLite3.

[27] sqlite.org, „SQLite Foreign Key Support," 2009. adresa: https://www.sqlite.org/foreignkeys.html.

Popis slika

1. Testiranje u vodopadnom razvoju (Prema: Rastogi, 2015) 2
2. Testiranje u iterativnom razvoju (Prema: Rastogi, 2015) 3
3. Prikaz osnovnog PyQt5 GUI-ja (Izvor: RiverbankComputing, 2021) 6
4. Prikaz funkcionalnosti Robot Frameworka (Izvor: RobotFrameworkFoundation, 2021) 8
5. Prikaz osnovnog PyQt5 GUI-ja (Izvor: Moelholm, 2016) 9
6. Osnovno načelo TDD-a (Prema: Steinfeld, 2020) 11
7. Osnovno načelo BDD-a (Prema: cucumber.io, 2021) 12
8. Usporedba crne, bijele i sive kutije (Prema: WhiteHackLabs, 2019) 15
9. Razine testiranja (Prema: professionalqa.com, 2019) 15
10. Koraci testiranja prihvatljivosti (Prema: softwaretestinghelp.com, 2021) 18
11. ERA model Northwind baze (Izvor: White, 2021) 20
12. Prikaz rezultata testiranja 23
13. Prikaz tablice proizvoda 28
14. Prikaz ažuriranja dobavljača 34
15. Prikaz rezultata integracijskog testiranja read i update komponenti 35
16. Greška prilikom dodavanja novog proizvoda 41
17. Pokretanje integracijskog testa za integraciju dodavanja 43
18. Brisanje redaka iz GUI-ja 45
19. Prikaz analitičkih podataka 54
20. Prikaz ispisa u terminalu testiranje ponašanja s Behave bibliotekom 60
21. Prikaz ispisa u terminalu prilikom testiranja Robot frameworkom 69
22. Prikaz log datoteke prilikom testiranja Robot frameworkom 69

23. Prikaz report datoteke prilikom testiranja Robot frameworkom 70