Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Sandi Pangerc

Avtomatizirano testiranje aplikacije Android

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Testiranje je pomemben del razvoja programske opreme. V diplomskem delu predstavite načine in metode testiranja programske opreme. Opisane koncepte uporabite na praktičnem primeru aplikacije Android in preučite ter opišite, kako posamezni načini in metode pozitivno oziroma negativno vplivajo na razvoj produkta.

Zahvaljujem se mentorju doc. dr. Tomažu Dobravcu za vložen čas in pomoč pri pripravi diplomskega dela. Zahvaljujem se tudi staršem za vso podporo med študijem in podjetju MESI, d.o.o., še posebej Vodji razvoja Tomotu Krivcu, za priložnost in izkušnje, ki sem jih pridobil. Zahvaljujem se še Jani Penca za vzpodbudo in pomoč pri pisanju diplomskega dela in prav tako Luciji Kotar, ki mi je pomagala pri odpravljanju slovničnih napak v delu.

Kazalo

Seznam uporabljenih kratic

kratica	**angleško**	**slovensko**
IDE	Integrated Development Environment	Integrirano razvojno okolje
XP	Extreme Programming	Ekstremno programiranje
TDD	Test-Driven Development	Testno voden razvoj
BDD	Behavior-Driven Development	Vedenjsko voden razvoj
ATDD	Acceptance Test Driven Development	Sprejemno voden razvoj
FDD	Feature-Driven Development	Funkcionalno voden razvoj
API	Application Programming Interface	Aplikacijski programski vmesnik
JVM	Java Virtual Machine	Java navidezni stroj
MVC	Model-View-Controller	Model-Pogled-Krmilnik
DVM	Dalvik Virtual Machine	Dalvik navidezni stroj
SDK	Software Development Kit	Orodja za razvoj programske opreme

Povzetek

Naslov: Avtomatizirano testiranje aplikacije Android

Avtor: Sandi Pangerc

V tem diplomskem delu so predstavljeni postopki testiranja aplikacije Android. Namen dela je spoznati osnovne koncepte in načine testiranja programske opreme ter jih nato uporabiti v praksi. Glavni cilj je ustvariti avtomatizirano testno okolje za programsko opremo in spoznati, kakšni so učinkoviti načini testiranja. V okviru diplomskega dela se bomo osredotočili na testiranje aplikacije Android znotraj produkta podjetja MESI, d.o.o. Izbrana orodja za testiranje so JUnit, Mockito in Espresso. Orodja bomo uporabljali za testiranje tako same aplikacije, kot integracijo aplikacije znotraj platforme. Z izvajanjem testnih primerov želimo zagotoviti zanesljivo delovanje. Iz rezultatov ugotovimo, kako uspešni so določeni načini testiranja in kakšni so potrebni ukrepi za reševanje pomanjkljivega testiranja.

Ključne besede: Testiranje programske opreme, avtomatsko testiranje, testno orodje JUnit, testno orodje Mockito, testno orodje Espresso, Android, programska oprema.

Abstract

Title: Android automated testing

Author: Sandi Pangerc

The thesis presents testing methods for the Android application. The purpose of the thesis is to learn the basic concepts and approaches to software testing and applying them in practice. The main objective is to create a software automated testing environment and to identify which testing methods are most effective. In the thesis we have focused on testing the Android application within the company product made by MESI, Ltd. The chosen testing tools have been JUnit, Mockito, and Espresso. These tools have been used to test the application's stability and to verify that the application works as intended when integrated into the company's platform. By implementing these testing methods we hope to ensure a reliable and running operation. The end results help us discover how effective certain testing methods are, and what measures are necessary to resolve inadequate testing.

Keywords: Software testing, automated software testing, test tool JUnit, test tool Mockito, test tool Espresso, Android, software.

Poglavje 1

Uvod

Dandanes je programska oprema prisotna v vedno več vsakdanjih napravah, kot so računalniki, telefoni in razni vgrajeni sistemi. Ti so prisotni v digitalnih urah, pametnih hladilnikih ter novejših vozilih. Računalniki in telefoni niso več edine naprave, od katerih pričakujemo brezhibno delovanje. Od pametnih hladilnikov in vse do avtomobilov pričakujemo delovanje brez napak, saj te lahko povzročijo škodo ter nepotrebne stroške samemu proizvajalcu kot tudi strankam. Določene napake v programski opremi lahko stranke tudi smrtno ogrozijo. Zaradi teh razlogov je vedno bolj popularno testiranje v sami industriji, saj je napake, odkrite v začetnih fazah izdelave produkta, ceneje popraviti takoj kot kasneje pri stranki.

V tem diplomskem delu se bomo posebej osredotočili na avtomatsko testiranje platforme Android, ki je v zadnjih letih postala zelo razširjena v izdelovanju vgrajenih sistemov [1, 7]. Sam Android je odprtokoden operacijski sistem, kar omogoča vsem proizvajalcem vgrajenih sistemov ustvariti svojo različico le-tega. Med razvojem aplikacije za platformo Android se moramo, poleg njenega brezhibnega delovanja, zavedati tudi vseh različnih velikosti naprav, na katerih lahko teče. Tu razvijalci pridejo tudi do izzivov, kako svojo aplikacijo prilagoditi za različno velike naprave in različne verzije operacijskega sistema Android. Njihove rešitve potrjujejo preizkuševalci programske opreme skupaj z ročnimi in avtomatiziranimi testi. Cilj tega dela je

predstaviti, kaj je testiranje in kako postaviti avtomatizirani sistem testiranja aplikacije Android.

1.1 Motivacija za izdelavo diplomskega dela

Pri podjetju MESI, razvoj medicinskih naprav, d.o.o sem se kot del testne ekipe pobližje spoznal s področjem in avtomatizacijo testiranja programske opreme. Do sedaj je testiranje potekalo preko ročnega preverjanja produkta. S povečanjem števila razvijalcev, se je pojavila potreba po hitrejšem, zanesljivejšem in cenejšem načinu testiranja. Sam sem se z osnovami testiranja, glavnimi metodologijami in praksami razvijanja programske opreme spoznal tekom študijskih let. Napisal sem že nekaj praktičnih primerov testiranja programske opreme in sem bil vesel sodelovanja s podjetjem. Sprva sem razvijalce podjetja seznanil s testiranjem in pomembnostjo le tega. Prav tako sem skupaj s podjetjem sestavil avtomatiziran sistem testiranja produkta, na katerem teče aplikacija Android. Avtomatiziran sistem testiranja je bil ustvarjen z namenom enostavnega vzdrževanja in natančnega definiranja rezultata testiranja.

Poglavje 2

Koncepti testiranja programske opreme

V tem poglavju si bomo podrobneje ogledali različne pristope testiranja programske opreme. Spoznali bomo metode testiranja in kako se te razlikujejo glede na vpletenost in vloge same osebe, ki izvaja testiranje. Nato bomo spoznali, kako se testira programsko opremo v različnih stopnjah razvoja in kakšni so priporočeni pristopi testiranja v posameznih fazah. Spoznali bomo tudi, kateri način testiranja bolj ustreza naši programski opremi. Odvisno od uporabnikov produkta so nekatera podjetja lahko zadovoljna le z ročnim testiranjem programske opreme ali celo brez njega. Nazadnje si bomo ogledali še metodologije za izdelavo programske opreme, ki dajejo poudarek na pisanje testnih primerov. Temelji teh metodologij so razumeti, kaj želimo od naše kode, da naredi, preden začnemo z njeno implementacijo.

2.1 Metode testiranja programske opreme

Nekateri izvajalci testov zelo dobro poznajo, kaj program dela in kakšne vnose od nas program pričakuje, vendar lahko ravno zaradi tega dodatnega znanja zgrešijo kritične in ponavadi tudi očitne napake. Medtem ko bodo lahko drugi izvajalci testov, ki programa ne poznajo, zlahka odkrili kritične

napake s samim vnašanjem naključnih vhodov, ki jih programska oprema ne pričakuje (npr. namesto številke se vnese črka). Na podlagi teh dveh primerov izhaja ideja, da praviloma razvijalci svoje kode ne testirajo. Poglejmo si nekaj metod testiranja programske opreme. Od testiranja, ki ga izvajajo že razvijalci med samim pisanjem kode, in vse do testiranja programske opreme kot oseba, ki ne pozna, kako deluje programska koda in je le zunanji opazovalec.

2.1.1 Statično testiranje

Najbolj osnovni pristop testiranja je že preprosto pregledovanje kode (ang. software walkthrough) [2]. Pri pregledovanju kode nam zelo pomagajo sama integrirana razvojna okolja (IDE), saj nas že ta opozorijo o sintaktičnih napakah (npr. klicanje metod z napačnimi argumenti) ali strukturnih napakah (npr. neizvedljive vrstice kode). Ta pristop spada pod tako imenovane statične metode testiranja in se izvaja pred izvršitvijo programske kode. Najboljše je, če napake najdemo v tej fazi, saj prej ko jih najdemo, manjša je cena za njihovo popravilo. To je tudi prednost statičnega testiranja, ker iščemo napake, preden začnemo pisati testne primere. Primer programov statičnega testiranja so Android Studio, Visual Studio oziroma kakršenkoli IDE, ki nas že med pisanjem kode opozori na sintaktične napake. Še en primer statičnega testiranja je odpravljanje napak z gumijasto račko (ang. rubber duck debugging) [8]. Testiranje poteka tako, da razvijalec pregleduje kodo od vrstice do vrstice in hkrati poskuša na glas razložiti njeno delovanje. Med samim procesom pojasnjevanja, kako deluje, in opazovanjem, kaj koda dejansko dela, lahko razvijalec hitro opazi neskladnosti in popravi svoje napake.

2.1.2 Dinamično testiranje

Za razliko od statičnega testiranja se pri dinamičnem testiranju [2] zanašamo, da bomo napake odkrili pri zagonu programa in njegovih testnih primerov,

saj opazujemo obnašanje programa med njegovim izvajanjem. To testiranje se uporablja, ko spremenljivke v programu niso konstantne in ne moremo statično preveriti, kako se bo program obnašal. Dinamično testiranje vključuje dodajanje vhodnih podatkov programu in preverjanje, ali se ujemajo s pričakovanimi izhodnimi podatki. Primer dinamičnega testiranja, katerega vsi uporabljamo, vendar se tega mogoče ne zavedamo, so preproste permutacije kode (npr. sprememba pogoja) in nato ponoven zagon programa.

2.1.3 Metode škatel

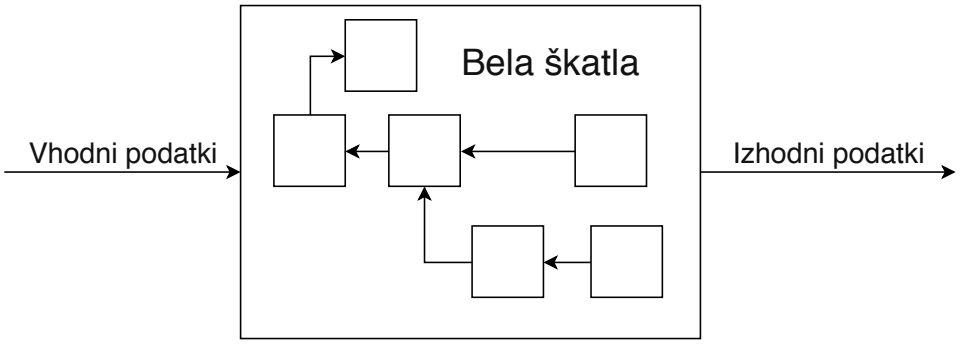
Načini testiranja z uporabo škatel [3] so starejši in širše uporabljeni izrazi v testiranju programske opreme. Najbolj pogosta postopka tega testiranja sta z uporabo bele in črne škatle. Ti pristopi opisujejo stališče, iz katerega se pišejo testni primeri.

Bela škatla

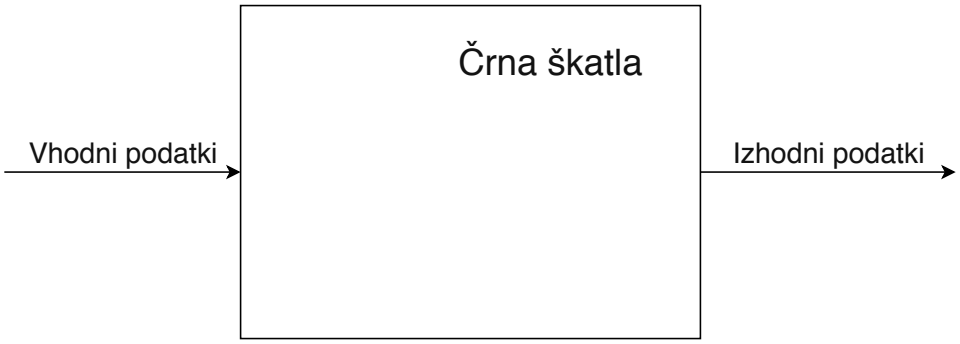
Pristop bele škatle (Slika 2.1) temelji na testiranju notranjih struktur programske opreme. Izvajalec testov ima vpogled v kodo, ki jo testira in ve, kako ta deluje. S tem znanjem tudi sestavi testne primere. To testiranje je nizkonivojsko, kar pomeni, da se izvaja na nivoju testiranja enot in njihove integracije. Več o posameznih nivojih testiranja bomo spoznali v poglavju 2.2.

Črna škatla

Pri pristopu črne škatle izvajalci ne potrebujejo izkušenj s programiranjem in ne vedo, kako testirana koda deluje. Na podlagi pričakovanih rezultatov preizkušajo programsko opremo. Njihov cilj je odkriti napake pri procesiranju, funkcionalnih zahtevah, napake vmesnika ali napake pri inicializaciji. Testiranje je višjenivojsko in se izvaja na nivoju testiranja sistema.



Slika 2.1: Bela škatla



Slika 2.2: Črna škatla

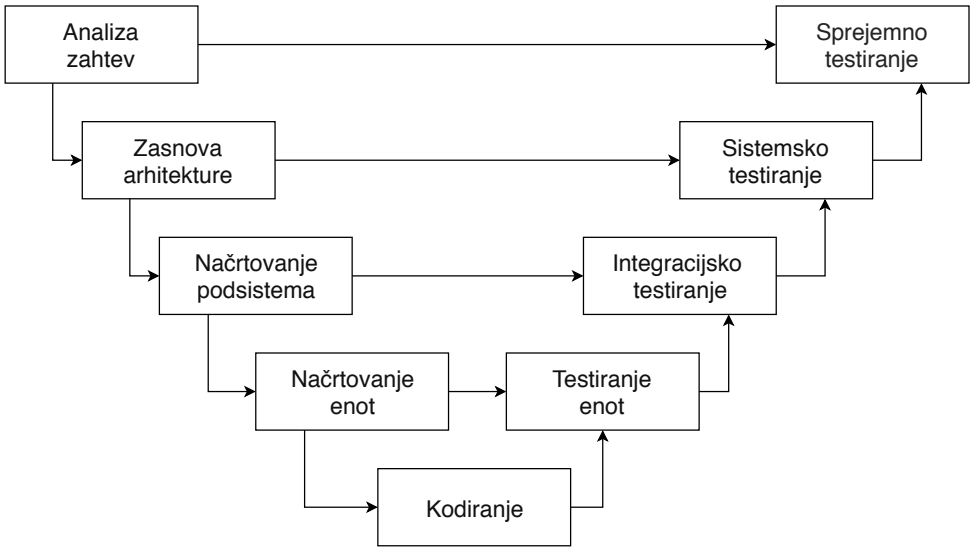
Siva škatla

Poleg metod bele in črne škatle, obstaja tudi metoda testiranja sive škatle. Ta metoda združi praksi bele in črne škatle. Izvajalcem testov je znano delovanje programa v ozadju, kar jim omogoča pripraviti boljše testne primere, tudi ko testirajo v načinu črne škatle.

2.2 Stopnje testiranja

Razvoj programske opreme poteka v več fazah (Slika 2.3) - od analize zahtev, zasnove arhitekture do kodiranja. Za vsako od razvojnih faz imamo

določeno stopnjo testiranja - od sprejemnega, sistemskega testiranja do testiranja enot [4]. V tem diplomskem delu se bomo bolj natančno osredotočili



Slika 2.3: Stopnje razvoja programske opreme in testiranja [5]

na stopnji testiranja enot in integracijskega testiranja, ostale stopnje bomo združili v sistemsko testiranje.

2.2.1 Testiranje modulov

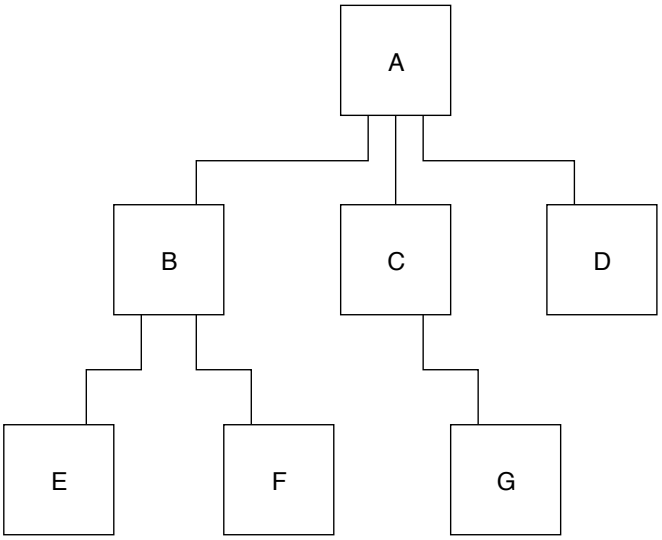
Takoj, ko imamo napisane module ali enote[1], lahko začnemo s testiranjem modulov [6]. Te testiramo v izolaciji (neodvisno od preostalih modulov), ker nas zanima, kako delujejo posamezno. Želimo se prepričati, da se izvajajo pravilno in vračajo pričakovane rezultate. Module, odvisne od podatkov drugih modulov, testiramo s pomočjo umetnih podatkov oz. lažnih objektov (ang. mock objects). Lažni objekti so zelo priročni, ko želimo simulirati določeno stanje in preveriti, če se modul obnaša pričakovano. Prav tako lahko z lažnimi objekti simuliramo tudi komunikacijo s strežnikom ali podatkovno

[1]modul je najmanjši del kode, ki ga lahko testiramo (primer v Javi: metoda)

bazo, saj ne želimo, da imajo naši testni primeri dostop do resničnih podatkov in posledično pokvarijo obstoječe podatke.

2.2.2 Integracijsko testiranje

Ko preverimo delovanje posameznih modulov, jih lahko začnemo združevati v podsisteme [6]. Namen integracijskega testiranje je preveriti, ali moduli med seboj pravilno komunicirajo. Predpostavljamo, da samostojni moduli delujejo pravilno. Poznamo več načinov integracije [5]:

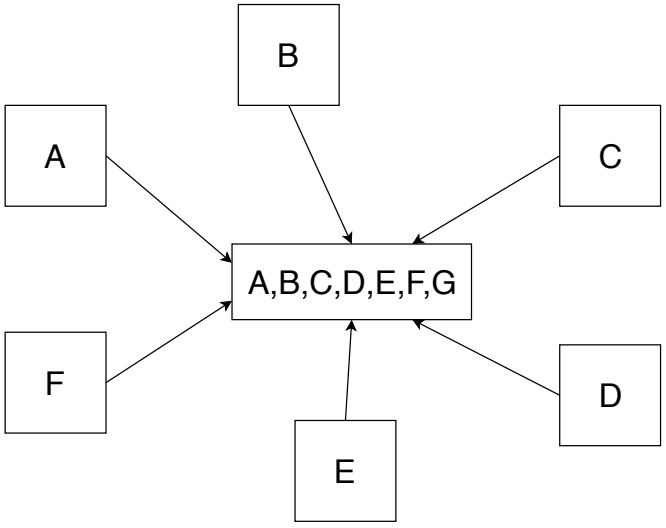


Slika 2.4: Arhitektura podsistema

- **Od spodaj navzgor (ang. bottom up)**: Začnemo s testiranjem najnižjih komponent (E, F, G in D iz Slike 2.4). Nato testiramo višje komponente (B, C in A iz Slike 2.4), tako da posamezno kličemo nižje. Če tu pride do napake vemo, da imamo problem v nižji komponenti oziroma v komunikaciji med višjo in nižjo komponento. S testiranjem

nadaljujemo, dokler ne pridemo do vrha podsistema. V primeru, ko višje komponente še niso pripravljene za integracijo, se jih nadomesti z gonilniki[2] (ang. drivers).

- **Od zgoraj navzdol (ang. top bottom)**: V nasprotju s prejšnjo metodo, s pristopom zgoraj navzdol, najprej testiramo integracije med najvišjimi komponentami (A, B, C in D iz Slike 2.4). Postopoma jih združujemo z vsemi komponentami nivo nižje in integracijo ponavljamo dokler ne pridemo do dna. V primeru, ko nižje komponente še niso pripravljene za integracijo, se jih nadomesti z uporabo vtičnikov[3] (ang. stub).



Slika 2.5: Integracija veliki pok

- **Veliki pok (ang. big-bang)**: Pri tej metodi združimo vse komponente naenkrat in jih testiramo kot celoto (Slika 2.5). Če celoten sistem

[2]gonilniki umetno predstavljajo višje nivojsko komponento in vračajo lažne podatke
[3]vtičniki umetno predstavljajo nižje nivojsko komponento in vračajo lažne podatke

deluje, smo zaključili z integracijo. V primeru, da se najdejo napake, je integracija neuspešna. Potrebno je odkriti, kje je nastala napaka (med integracijo katerih komponent), zato moramo celoten sistem razbiti na komponente in jih preveriti.

2.2.3 Testiranje sistema

Sistemsko testiranje

Sistemsko testiranje je sestavljeno tako, da preveri, ali sestavljen sistem ustreza svojim specifikacijam [6]. Domnevamo delovanje posameznih delov in se prepričamo, ali sistem deluje kot celota. Tu se ponavadi iščejo oblikovalske in specifikacijske težave. Testiranje vodi ekipa, ki zagotavlja kakovost produkta in od katere ni pričakovano znanje programiranja. Odkrivanje nizkonivojskih napak v tej stopnji je nesprejemljiva in zelo draga akcija.

Sprejemno testiranje

Na koncu imamo še sprejemno ali funkcionalno testiranje, s katerim želimo preveriti funkcionalne zahteve stranke [6]. To so višjenivojski testi, ki preverjajo pristnost uporabniških zgodb, več v podpoglavju 2.4.2, in funkcionalnih zahtev. Ti testi so primarno ustvarjeni s pomočjo strank, saj bodo le te znale oceniti ali je produkt narejen tako, kot so si ga zaželele. Primer testa: uporabnik se lahko registrira, ko izpolni vsa potrebna polja. Tu je postal tudi popularen vedenjsko voden razvoj (več v podpoglavju 2.4.2), ki omogoča tako razvijalcem kot strankam skupen jezik, s katerim lažje razumeta zahteve produkta.

2.3 Načini testiranja

Glede na velikost projekta in čas izdelave, se je potrebno odločiti, ali se bo naša programska oprema testirala ročno ali avtomatizirano. Za manjše projekte je bolj primerno ročno testiranje, saj na njih ponavadi dela manj

razvijalcev in ker je za preveriti manj funkcionalnosti. Za večje projekte, kjer več razvijalcev piše kodo za en produkt, so primernejši avtomatizirani testi. V primeru, da nimamo časa za pisanje avtomatiziranih testov, je obvezno izvajanje vsaj ročnega testiranja, saj se moramo prepričati, da naš izdelek deluje, preden ga predamo v uporabo strankam.

2.3.1 Ročno testiranje

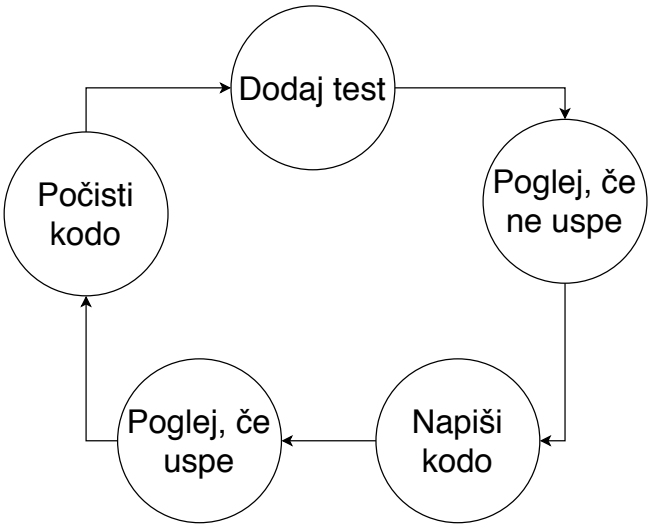
Ročno testiranje je proces testiranja programske opreme kot končni uporabnik. Med testiranjem se ne uporablja nobenega orodja, temveč se le preverja, ali ima programska oprema ustrezne funkcionalnosti in deluje po pričakovanjih. Dobra praksa ročnega testiranja je sestaviti teste glede na uporabniške zgodbe. S tem želimo pokriti čim več funkcionalnosti in odkriti vse morebitne napake. Velika prednost ročnega testiranja je hitrost testiranja. Ko želimo preveriti novo funkcionalnost, le zaženemo program in preverimo ali deluje pričakovano. Nasprotno za avtomatizirano testiranje, bi sprva morali napisati teste in šele nato bi jih zagnali. Ročni testi so tudi boljši pri iskanju uporabniških napak, saj so izvedeni bolj "človeško".

2.3.2 Avtomatizirano testiranje

Pri večjih projektih je boljša praksa pisanja avtomatiziranih testov. Večji kot je projekt, večje je število ljudi, ki sodeluje pri projektu in večje je število modulov v projektu. Na dolgi rok je cenejše izvajanje avtomatiziranih testov, saj je izvajanje ponavljajočih akcij hitrejše izvedeno avtomatsko. Prednost se še posebej opazi, če je koda programske opreme večkrat spremenjena in je vsakič potrebno ponovno preveriti vse funkcionalne zahteve. Potrebno se je seveda zavedati, da avtomatizirano testiranje ni primerno za vsa podjetja. Programska oprema, kjer se zelo pogosto zamenjuje tehnologija in logika same kode, ni primerna za avtomatizirano testiranje. S tem bi le dodatno izgubljali čas pri implementaciji testov, ki bi morali biti ponovno postavljeni, ob vsaki naslednji spremembi kode.

2.4 Metodologije za izdelavo programske opreme

Obstaja več metodologij za izdelavo programske opreme. Vsaka izmed njih ima svoje načine razvijanja programske kode in različne poglede na testiranje programske opreme. Poznamo agilne metodologije, ki so bolj usmerjene v hitrejše razvijanje kode, kot sta Scrum in exstremno programiranje (XP), kot tudi metodologije, ki specifično poudarjajo testiranje programske opreme [9]. To so na primer: testno voden razvoj (TDD), vedenjsko voden razvoj (BDD), sprejemno voden razvoj (ATDD) in funkcijsko voden razvoj (FDD). V tem diplomskem delu se bomo natančneje posvetili metodologijama TDD in BDD.



Slika 2.6: Vizualizacija: TDD

2.4.1 Testno voden razvoj (TDD)

Testno voden razvoj [6] je ena izmed metodologij, ki daje poudarek na pisanje testov, preden začnemo pisati samo kodo. Tak način pisanja prepleta načrtovanje kode s testiranjem, kar pomaga razvijalcem pri pisanju kvalite-

tnejše kode. Osredotočimo se na specifikacijske zahteve in manj na funkcionalne zahteve. Testno voden razvoj poteka ciklično (Slika 2.6) in se začne z dodajanjem testa specifikacije. Nato zaženemo vse teste in pogledamo, če novi test uspe. Če test uspe, testirano specifikacijo že imamo in zaključimo cikel. Če test ne uspe, napišemo kodo, da zadosti novemu testu. Ponovno zaženemo vse teste. Če testi ne uspejo, popravimo kodo in postopek ponavljamo. Ko so vsi testi uspešni, je na vrsti čiščenje kode. Odstranimo podvojitve, smiselno poimenujemo spremenljivke in konstante, preoblikujemo samo strukturo kode ter dodamo komentarje, kjer je potrebno. Cikel ponavljamo za vsako specifikacijo, ki jo želimo dodati. Testno vodeno razvijanje razvijalce osredotoči le na specifikacije, ki jih morajo implementirati in sami ne dodajajo specifikacije, za katere mislijo, da bodo potrebne v prihodnosti.

2.4.2 Vedenjsko voden razvoj (BDD)

Vedenjsko voden razvoj [10] se je razvil iz metodologije TDD. Velikokrat so razvijalci v dvomih, kdaj začeti s testiranjem, kaj naj testirajo, česa naj ne testirajo, koliko naj testirajo in tudi zakaj je bil test neuspešen. Osredotočiti se želimo na celoten sistem in ne na posamične enote. Test je lahko neuspešen zaradi napačnega vhodnega podatka, ampak mi želimo izvedeti, zakaj je tale podatek povzročil napako v sistemu. Uspešnost sistema visi na vedenjskih oziroma funkcionalnih zahtevah. Poskrbeti moramo za razumevanje zahtev vseh razvijalcev, vsi morajo razumeti celotno sliko in kako bo sistem deloval. Kako naj se program obnaša, določi stranka s pomočjo uporabniških zgodb.

Uporabniške zgodbe

Uporabniške zgodbe [10] so pričakovane funkcionalne zahteve programske opreme. Podjetje jih mora definirati skupaj s svojimi strankami v uporabniške zgodbe, da so čimbolj razumljive in natančno zapisane po njihovih željah. Velikokrat se zgodi, da stranke ne znajo opisati, česa si želijo za končni produkt. Uporabniške zgodbe izrazijo funkcionalne zahteve s pomočjo treh faktorjev: vloga, cilj in motivacija. Predpostavimo primer: kot študent

si lahko izposojam knjige v namen študija. Vlogo v uporabniški zgodbi predstavlja študent, s ciljem izposoje knjig in z motivacijo študija. Zaradi enostavnega besedilnega jezika, lahko tako stranke kot razvijalci točno razumejo, kaj je od produkta pričakovano.

Poglavje 3

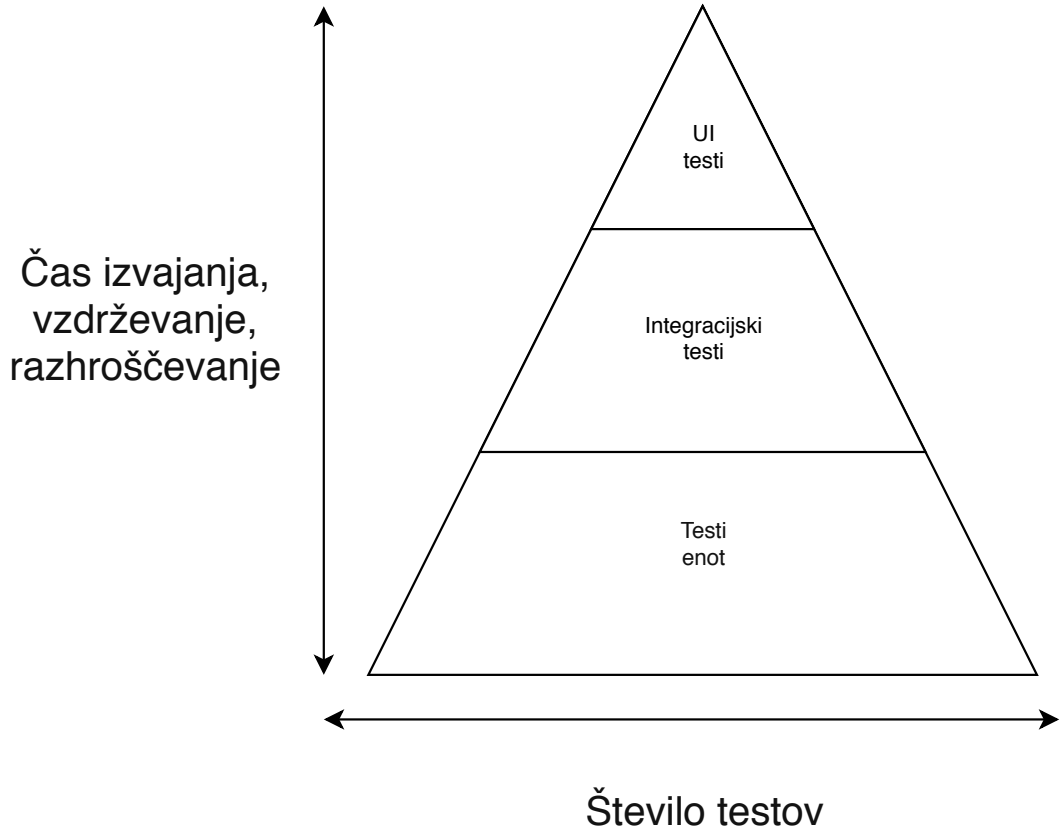
Testiranje aplikacij Android

V tem poglavju si bomo ogledali, kako poteka testiranje v platformi Android. Osredotočili se bomo na samo kodo in kako jo prevajalnik pretvori v končni zaganljivi skupek. Izpostavili bomo, kako lahko hitrejše in učinkovitejše testiramo v platformi Android s pravilno implementacijo kode. Spoznali bomo tudi nekaj različnih orodij za izvajanje testov in kako se posamezna orodja razlikujejo med seboj. Na koncu se bomo osredotočili na sestavljanje praktičnih primerov testiranja in kako lahko z izbranimi orodji sestavljamo učinkovite testne primere.

3.1 Testiranje v platformi Android

Aplikacije se uporabljajo za različne akcije, kot so izračun seštevka dveh števil, premikanje po uporabniškem vmesniku, branje perifernih naprav, prenašanje in pošiljanje spletnih virov preko naše aplikacije. Za vsako interakcijo ali akcijo, ki jo uporabnik lahko naredi, moramo biti prepričani, da le ta deluje.

Kot smo že omenili v podpoglavju 2.2 imamo tri stopnje testov, ki so predstavljeni v testni piramidi (Slika 3.1) [11]. Piramida nam prikazuje najprimernejšo količino testov na posameznih stopnjah. Vedno si želimo imeti največ testov modulov (testi enot), manj integracijskih testov in najmanj sistemskih testov (testi uporabniškega vmesnika). Primerno je, če najprej

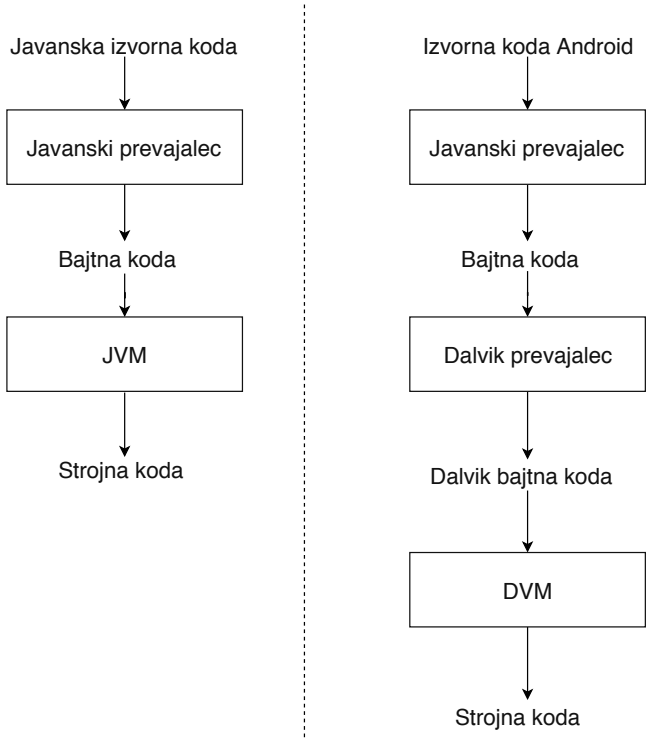


Slika 3.1: Piramida testiranja [11]

celotno kodo razdelimo na module. Te teste imenujemo majhni testi in so zagnani večkrat med razvijanjem programske opreme. Ko module združujemo, začnemo z integracijskim testiranjem. Tu želimo potrditi komunikacije med moduli. To so srednji testi in so že pognani na napravi. Na koncu imamo še velike teste, ki potrjujejo delovanje uporabniškega vmesnika. Teh testov je najmanj in se prav tako izvajajo na napravi.

Struktura map v projektu Android nam že sama pokaže glavni veji testiranja: Java del in Android del. Že vnaprej vemo, da bo potrebno za posamezni veji napisati svoje teste. Seveda se lahko javansko kodo še vedno testira skupaj z Android aplikacijskim programskim vmesnikom (API), ven-

dar s tem izgubimo veliko fleksibilnosti in na dolgi rok tudi dragoceni čas. V naslednjih podpoglavjih bomo predstavili, zakaj pri testiranju ločujemo javansko kodo in Android API.

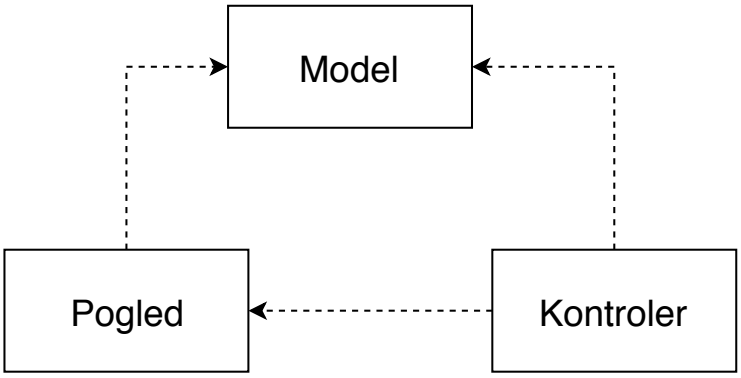


Slika 3.2: Postopek prevajanja Android API in javanske kode [12]

3.1.1 Prevajanje javanske kode

V nasprotju z Android API, je javanska koda lahko izvedena neodvisno od naprave, kar nam bistveno zmanjša čas zaganjanja programske kode in zato je javanska koda zelo primerna za poganjanje testov enot. Ko prevajamo (Slika 3.2) javansko kodo s pomočjo javanskega prevajalca (ang. Java Compiler), se nam ta prevede v bajtno kodo [12]. To potem razčleni javanski navidezni stroj (JVM) v strojno kodo, katero nam JVM ustrezno spremeni glede na operacijski sistem, na katerem teče. JVM je podprt s strani vseh vodilnih

operacijskih sistemih, kot so Linux, MacOS in Windows, kar pomeni, da je podprt na praktično vseh napravah.



Slika 3.3: MVC model

Model-Pogled-Krmilnik (MVC)

Ravno zato, ker lahko javansko kodo testiramo hitreje, če jo posebej ločimo od Android API, je postalo popularno tudi programiranje v načinu Model-Pogled-Krmilnik (MVC) [6]. Bistvo MVC je ločevanje javanske kode v modele, Android API v poglede in ustvariti krmilnike, ki modele in poglede povezujejo (Slika 3.3). Prednosti MVC so tudi na področju razvijanja programske opreme: koda je razčlenjena in je lahko uporabljena na več mestih, prav tako omogoča, da čelni in zaledni razvijalci ločeno razvijajo svoje podsisteme.

3.1.2 Prevajanje Android API

Kot je že prikazano na Sliki 3.2, se javanska in Android koda prevajata na različna načina [12]. Tako kot javanska koda, se Android sprva prevede s pomočjo javanskega prevajalca v bajtno kodo, vendar jo za tem prevede Dalvik prevajalec v Dalvik bajtno kodo. Dalvik prevajalec se nahaja znotraj Android SDK. To bajtno kodo nato Dalvik navidezni stroj (DVM) razčleni

v strojno kodo. V nasprotju z JVM, DVM ni razširjen za uporabo na drugih operacijskih sistemih in se primarno uporablja samo za operacijski sistem Android.

3.2 Orodja za testiranje v platformi Android

Kot smo že spoznali, testiranje aplikacij Android razdelimo na dva osnovna dela. Javansko kodo, ki večinoma vsebuje module, bomo testirali z metodo bele škatle. Android API, ki vsebuje tudi uporabniške vmesnike, bomo testirali z metodami bele, črne ali sive škatle. Vse to je možno doseči z orodji, ki jih bomo predstavili v naslednjih podpoglavjih. Naj omenimo še, da so to le nekatera orodja, ki jih imamo na voljo za testiranje.

3.2.1 JUnit

JUnit [4] je testno okolje za programski jezik Java. Če imamo v aplikaciji Android enoto, ki ne vsebuje Android API, je priporočljiva uporaba JUnit za testiranje. JUnit ponuja statične metode za preverjanje rezultatov preko razreda `Assert`. Te metode se imenujejo trdilne izjave (ang. assert statements). Argumenti teh izjav nam omogočajo, da za vsak test opišemo njegovo delovanje, kar izboljša pregled rezultatov testiranja. V primeru, da trdilna izjava ni resnična, se sproži trdilna izjema (ang. assert exception), ki označi test kot neuspešen. Kot primer trdilne izjave bomo predstavili testiranje metode `mnozenje(a,b)`, ki za vhodna parametra vzame dve celi števili in ju zmnoži.

```
1 public int mnozenje(int a, int b) {
2       return a*b;
3 }
4 assertEquals("2 x 0 mora biti 0", 0, mnozenje(2,0));
```

Izsek kode 3.1: Metoda `mnozenje`

3.2.2 Mockito

Med izvajanjem testov v fazi testiranja enot se želimo prepričati, da naše enote delujejo, preden jih začnemo združevati z drugimi. Težava nastane, ko testiramo enote, ki so odvisne od drugih enot (pričakujejo določeno akcijo ali rezultat druge enote). V tem primeru lahko ustvarimo lažne objekte in z njimi testiramo našo enoto. Eno izmed orodij, ki nam to omogoča je Mockito [13]. Mockito je primarno uporabljeno samo za testiranje javanske kode, lahko pa se ga uporablja tudi za testiranje Android API. Tu lahko pride do zapletov zaradi samega procesa prevajanja Android API, kot smo že navedli v podpoglavju 3.1.2. Kot primer si bomo pogledali izdelavo in uporabo lažnega objekta med testiranjem.

```
1 // ustvarimo navidezen objekt Kalkulator
2 Kalkulator kalkulator = mock(Kalkulator.class);
3 // ob katerem koli paru celih števil nam vrni 13
4 when(kalkulator.zmnozi(anyInt(), anyInt()).thenReturn(13);
5 // uporabimo navidezen objekt v testu
6 assertEquals(kalkulator.zmnozi(1,2), 13);
```

Izsek kode 3.2: Uporaba orodja Mockito

3.2.3 Espresso

Eno izmed bolj razširjenih orodij za testiranje Android API in uporabniških vmesnikov je Espresso [14]. Espresso lahko uporabljamo na fizičnih in navideznih napravah, kar nam ponuja večjo fleksibilnost pri testiranju. Vključevanje orodja v naš projekt je zelo enostavno, saj moramo le dodati ustrezne odvisnosti v konfiguracijsko datoteko in Android Studio bo s pomočjo orodja Gradle poskrbel za vse ostalo. Espresso lahko uporabljamo z metodama črne in bele škatle. Prav tako dovoljuje razvijalcem testiranje uporabniškega vmesnika preko trdilnih izjav, ki se pišejo na podoben način kot povedi. Kot primer bomo preverili, ali se ob kliku na gumb z besedilom `Pokazi pogled`, pokaže pogled, ki smo mu določili unikatno ime (ang. identifica-

tion) `skritiPogled`, z vsebino `Tukaj sem`.

```
1 onView(withText("Pokazi pogled"))
2     .perform(click());
3 onView(withId(R.id.skritiPogled))
4     .check(matches(withText(("Tukaj sem"))))
5     .check(matches(isDisplayed()));
```

Izsek kode 3.3: Uporaba orodja Espresso

3.2.4 UiAutomator

UiAutomator [15] je še eno orodje za testiranje uporabniškega vmesnika. V primerjavi z Espresso, testira z metodo črne škatle, saj nima dostopa do internih pogledov (ang. views). Ima druge prednosti, saj lahko upravlja z napravo, na kateri se izvajajo testi. Imamo možnost pritiskov na fizične gumbe naprave, prižiganje in ugašanje zaslona, omrežja, Bluetooth... Kot primer bomo z orodjem pritisnili na gumb *Domov* na napravi.

```
1 // poišči napravo
2 UiDevice mDevice = UiDevice.getInstance
3       (InstrumentationRegistry.getInstrumentation());
4 // pritisni na gumb Domov
5 mDevice.pressHome();
```

Izsek kode 3.4: Uporaba orodja UiAutomator

3.2.5 Pokritost kode

Pokritost kode nam pove, kolikšen odstotek kode je bil izveden med testiranjem [2]. V testiranju si želimo imeti čim večjo pokritost kode. Pokritost ne pomeni, da koda deluje pravilno, pomeni le, da je bila ta testirana in vrača pričakovane rezultate. Med pisanjem testnih primerov si težje zapomnimo ali potrdimo, katere vrstice kode so bile preverjene. Android Studio ima že integrirano orodje, s katerim nam izračuna pokritost kode. Vse, kar je potrebno

narediti, je omogočiti to orodje in pognati teste.

3.3 Javanski testi

Javansko kodo aplikacije lahko testiramo hitreje, saj se testi izvajajo neodvisno od naprave. Tu testiramo posamezne module aplikacije. V tem poglavju se bomo pobližje spoznali s knjižnicama JUnit in Mockito, ki smo ju opisali že v prejšnjem poglavju 3.2.

3.3.1 Dodajanje JUnit in Mockito v aplikacijo

Najprej v projekt dodamo knjižnici JUnit in Mockito. To naredimo z dodajanjem novih odvisnosti (ang. dependencies) v projektno konfiguracijsko datoteko okolja Gradle (`build.config`).

```
1 dependencies {
2     // JUnit
3     testImplementation 'junit:junit:4.12'
4     // Mockito
5     testImplementation 'org.mockito:mockito-core:1.10.19'
6 }
```

Izsek kode 3.5: Dodajanje orodij JUnit in Mockito v aplikacijo

3.3.2 Deklaracija JUnit razreda

Android Studio nam ob ustvarjanju novega projekta sam ustvari predlogo JUnit razreda. Ustvarjena predloga je zelo strnjeno sestavljena, saj vsebuje le en osnoven test in nam ne ponuja nobenih indikacij, kako testni razred povežemo s kodo, ki jo želimo testirati.

```
1 package com.test.package;
2
3 import org.junit.Test;
4 import static org.junit.Assert.*;
```

```
5
6 public class ExampleUnitTest {
7     @Test
8     public void addition_isCorrect() {
9         assertEquals(4, 2 + 2);
10    }
11 }
```

Izsek kode 3.6: Primer JUnit razreda

Izvajanje JUnit razreda je kompleksnejše, kot je prikazano na zgornjem primeru kode. Razred je sestavljen iz več metod, ki se na podlagi anotacij (ang. annotation) izvedejo ob različnem času. Te anotacije so:

- **@BeforeClass**: Metoda se bo izvedla, preden se ustvari instanca testnega objekta in preden se bo izvedel katerikoli izmed testov. Zaradi tega mora biti metoda statična[1]. Ker je to prva metoda, ki se bo izvedla in ker se bo izvedla samo enkrat, je primerna za izvajanje dražjih in daljših operacij. Primer: prenos in odprtje toka z datoteko iz spleta ali odpiranje povezave s podatkovno bazo.
- **@AfterClass**: Metoda se bo izvedla, ko bodo zaključeni vsi testi in bodo vse metode zaključile svoj življenjski cikel. Tudi ta metoda mora biti statična. Primer: zapiranje toka z datoteko in zapiranje povezave na strežnik ali zapiranje povezave s podatkovno bazo.
- **@Before**: Uporabljena metoda se izvede pred začetkom vsakega testa in je zato primerna za nastavljanje okolja testiranja. Ponavadi se v tej metodi ustvari instanco testiranega objekta, na katerega se potem sklicujemo ob vsakem testu.
- **@After**: Uporabljena metoda se izvede, ko se test zaključi. Primerna uporaba te metode je čiščenje okolja, ki smo ga vzpostavili v `@Before`. S tem si zagotovimo čisto in neodvisno okolje za izvajanje testov.

[1] statične metode pripadajo razredu (ne objektu) in so lahko izvedene brez inicializacije objekta

- **@Test**: Metoda, v kateri se izvajajo testi. Skupaj s trdilnimi izjavami potrjuje delovanje kode.

Če dopolnimo celoten vzorec JUnit razreda, dobimo naslednje:

```
1 public class ExampleUnitTest {
2     @BeforeClass
3     public static void beforeClass() {
4     }
5     @Before
6     public void setUp() {
7     }
8     @After
9     public void tearDown() {
10    }
11    @AfterClass
12    public static void afterClass() {
13    }
14    @Test
15    public void addition_isCorrect() {
16        assertEquals(4, 2 + 2);
17    }
18 }
```

Izsek kode 3.7: Predloga JUnit razreda

3.3.3 Ustvarjanje prvega JUnit testa

Predpostavimo, da opravljamo delo razvijalca, ki je dobil naslednjo nalogo. Zasnuj program s sledečimi zahtevami:

1. Program zna zmnožiti dve celi števili
2. Program zna seštevati nedoločeno število decimalnih vrednosti
3. Program vedno vrne rezultat v obliki navzdol zaokroženega celega števila

Iz zgornjih zahtev razberemo, da bo naš program izvajal tri operacije: seštevanje, množenje in zaokroževanje rezultatov. Ustvarili bomo dve metodi, ki ju bomo

poimenovali `zmnozi`, za izvajanje operacije množenja dveh celih števil, in `sestej`, za izvajanje operacije seštevanja nedoločenega števila decimalnih vrednosti. Rezultate bomo za sedaj zaokroževali kar znotraj omenjenih metod. Preden začnemo z implementacijo, sprva ustvarimo grob obris razreda in napišemo teste novih zahtev.

```
1 public class Kalkulator() {
2       public int zmnozi(int a, int b) {
3             return -1;
4       }
5       public int sestej(double... vrednosti) {
6             return -1;
7       }
8 }
```

Izsek kode 3.8: Implementacija razreda `Kalkulator`

Implementirajmo še teste, ki preverjajo dane zahteve.

```
1 @Test
2 public void mnozenjeCelihStevil_uspe() {
3     Kalkulator kalkulator = new Kalkulator();
4     assertEquals(kalkulator.zmnozi(4,2), 8);
5 }
6 @Test
7 public void sestevanjeDvehCelihStevil_uspe() {
8     Kalkulator kalkulator = new Kalkulator();
9     assertEquals(kalkulator.sestej(1,1), 2);
10 }
11 @Test
12 public void sestevanjeVecDecimalnihStevil_uspe() {
13     Kalkulator kalkulator = new Kalkulator();
14     assertEquals(kalkulator.sestej(
15       1.2, 0.3, 2.57, 0.03, 1.8, -0.8), 5);
16 }
```

Izsek kode 3.9: Implementacija testov razreda `Kalkulator`

S prvim testom potrjujemo množenje dveh celih števil, z drugim seštevanje celih števil in s tretjim seštevanje več decimalnih števil. Testi so poimenovani po vzorcu `kajTestPočne_pričakovaniRezultat`, ki nam omogoča hitro preverjanje, kaj posamezni test dela.

Zgornje definirane teste lahko optimiziramo z uporabo notacij `@Before` in `@After`.

```
1 @Before
2 public void setUp() {
3     kalkulator = new Kalkulator();
4 }
5 @After
6 public void tearDown() {
7     kalkulator = null;
8 }
9 @Test
10 public void mnozenjeCelihStevil_Uspe() {
11     assertEquals(kalkulator.zmnozi(4,2), 8);
12 }
13 @Test
14 public void sestevanjeDvehCelihStevil_Uspe() {
15     assertEquals(kalkulator.sestej(1,1), 2);
16 }
17 @Test
18 public void sestevanjeVecDecimalnihStevil_Uspe() {
19     assertEquals(kalkulator.sestej(
20       1.2, 0.3, 2.57, 0.03, 1.8, -0.8), 5);
21 }
```

Izsek kode 3.10: Optimizacija implementacije testov

Vsak test še vedno uporablja svojo instanco objekta Kalkulator, hkrati smo se znebili tudi podvojene kode. Če sedaj teste poženemo, bodo neuspešni, saj programske kode še nismo implementirali. Implementacija metod bi lahko bila sledeča.

```
1 public int zmnozi(int a, int b) {
```

```
2         return a*b;
3 }
4 public int sestej(double... vrednosti) {
5       double sestevek = 0;
6       for (double vrednost : vrednosti) {
7             sestevek += vrednost;
8       }
9       return (int)sestevek;
10 }
```

Izsek kode 3.11: Implementacija metod razreda `Kalkulator`

Za testiranje zgornjih funkcionalnosti bi lahko napisali ogromno število testov, saj bi za oba vhoda metode `zmnozi`, lahko uporabili vsako celo število. Celo število je v Javi predstavljeno z 32 bit-i, kar pomeni, da imamo samo za en vhod 2^{32} različnih možnosti. Za naš primer nam bo zadostovalo le par kombinacij vhodov (dva pozitivna, dva negativna, kombinacija negativnega in pozitivnega, kombinacija pozitivnega vhoda in ničle ter kombinacija negativnega vhoda in ničle).

Predpostavimo, da je imelo vodstvo ponoven sestanek s stranko in do nas pride nova zahteva:

4. Program zna seštevati in množiti samo pozitivne vrednosti

Napišemo naslednje nove teste, da zadostijo novih zahtevam.

```
1 @Test(expected = IllegalArgumentException.class)
2 public void mnozenjeNegativnihStevil_VrzeIzjemo() {
3       kalkulator.zmnozi(5, -2);
4 }
5 @Test(expected = IllegalArgumentException.class)
6 public void sestevanjeZNegativnimiStevili_VrzeIzjemo() {
7       kalkulator.sestej(0.3, -0.4, 1, -2, -9.16);
8 }
```

Izsek kode 3.12: Implementacija testov novih zahtev

V primeru napačno vpisanih argumentov (v tem primeru so to negativne vrednosti) želimo prejeti izjemo napačnih argumentov (`IllegalArgumentException`). Poženemo teste in na novo dodani testi se ne izvedejo uspešno. Ustrezno prilagodimo še izvorno kodo kalkulatorja in dodamo proženje izjeme napačnih argumentov. Proženje izjeme implementiramo z dodajanjem pogoja na začetek metode `zmnozi` in v zanko metode `sestej`, preden seštejemo število h končnemu rezultatu. Če ponovno poženemo teste, nam novo dodana testa uspeta, vendar se nam sedaj test `sestevanjeVecDecimalnihStevil_Uspe` izvaja neuspešno. Težava leži v negativnemu številu, ki je dodan med argumente testa. Test popravimo tako, da ponovno zastavimo trdilni stavek.

```
1 int sestevek = kalkulator.sestej(1.2, 0.3, 2.57, 0.03, 1.8, 1);
2 assertEquals(sestevek, 6);
```

Izsek kode 3.13: Popravek trdilnega stavka

Sedaj se nam vsi testi pravilno izvedejo in hkrati sledijo zastavljenim zahtevam. Na tak način razvijalci svojo kodo in teste prilagajajo novih zahtevam. Ko so spremembe zahtev manjše in je sprememba obnašanja metod majhna, so takšna popravila hitra in obvladljiva. Da taka ostanejo je pomembno natančno načrtovati specifikacije in definirati obnašanje sistema pred implementacijo, saj nam večje neskladnosti lahko prinesejo veliko dodatnega dela in časa, ki bi lahko bil bolje izkoriščen (npr. napisali bi več kode in več testov).

3.3.4 Integracijsko testiranje z JUnit

Najbolj preprosto je, ko razvijamo posamezne enote in preverjamo le njihovo delovanje. Šele ko jih začnemo združevati, lahko odkrijemo ključne napake implementacije. Predpostavimo, da nadaljujemo z implementacijo aplikacije iz prejšnjega podpoglavja 3.3.3 in želimo metodo `sestej` nadgraditi. Preden vhodne vrednosti seštejemo, jih bomo sprva zaokrožili na najbližje celo število. To bomo naredili s pomočjo nove metode `zaokrozi`. Sprva napišemo teste za novo metodo in jo nato še implementiramo.

```
1 // testi
2 @Test
3 public void zaokrozevanjeNavzdol_Uspe() {
4     assertEquals(kalkulator.zaokrozi(1.4), 1);
5     assertEquals(kalkulator.zaokrozi(5.2), 5);
6     assertEquals(kalkulator.zaokrozi(-7.6), -8);
7     assertEquals(kalkulator.zaokrozi(10.49), 10);
8 }
9 @Test
10 public void zaokrozevanjeNavzgor_Uspe() {
11     assertEquals(kalkulator.zaokrozi(1.6), 2);
12     assertEquals(kalkulator.zaokrozi(5.8), 6);
13     assertEquals(kalkulator.zaokrozi(-7.4), -7);
14     assertEquals(kalkulator.zaokrozi(10.51), 11);
15 }
16 // implementacija metode zaokrozi
17 public int zaokrozi(double vrednost){
18     return (int)Math.round(vrednost);
19 }
```

Izsek kode 3.14: Primer integracijskega testiranja

Testi lahko vsebujejo več trdilnih izjav. Test bo uspešen le v primeru, da se vse izjave uspešno izvedejo. Z vso potrebno kodo lahko sedaj začnemo z integracijskimi testi. Vse teste za metodo `sestej` moramo prilagoditi novi implementaciji. Napisali bomo dodatne teste, s katerimi bomo potrjevali pravilno delovanje integracije in popravili stare testne primere.

```
1 @Test
2 public void sestevanjeVecDecimalnihStevil_Uspe() {
3     assertEquals(kalkulator.sestej(
4       1.2, 0.3, 2.57, 0.03, 1.8, 1), 7);
5     assertEquals(kalkulator.sestej(
6       30.1234, 6.39, 0.10, 4.67), 41);
7     assertEquals(kalkulator.sestej(17.7, 87, 10.4, 7.36), 122);
8     assertEquals(kalkulator.sestej(8, 2, 7, 4, 0, 9), 30);
9 }
```

```
10 @Test
11 public void sestevanjeDvehDecimalnihStevil_Uspe() {
12     assertEquals(kalkulator.sestej(0.6, 1.7), 3);
13     assertEquals(kalkulator.sestej(2.1, 6.04), 8);
14     assertEquals(kalkulator.sestej(1.31, 5.72), 7);
15     assertEquals(kalkulator.sestej(10.33, 100.14), 110);
16 }
```

Izsek kode 3.15: Integracijski testi z več trdilnimi izjavami

Ustrezno moramo tudi spremeniti implementacijo metode `sestej`, v kateri vrednosti, preden jih seštejemo, zaokrožimo na najbližje celo število.

```
1 public int sestej(double... vrednosti) {
2     int sestevek = 0;
3     for (double vrednost : vrednosti) {
4         if (vrednost < 0) {
5             throw new IllegalArgumentException();
6         }
7         sestevek += zaokrozi(vrednost);
8     }
9     return sestevek;
10 }
```

Izsek kode 3.16: Poprava metode `sestej`

Na tak način se testiranje javanske kode nadaljuje, dokler ne pokrijemo vseh metod in preverimo njihovega delovanja med seboj. Teste lahko med razvojem dopolnjujemo in jim dodajamo dodatne pogoje za preverjanje. Navada je tudi pisanje testov za specifično odkrite napake. Odkrije se napaka v metodi, napako se popravi in napiše test, ki preverja odsotnost te napake.

Delo z navideznimi objekti

Predpostavimo sedaj primer, ko testiramo metodo `sestej`, ampak metoda `zaokrozi` še ni dokončana. Metodo `zaokrozi` bomo predstavili s pomočjo navideznih objektov. Za integrirane metode, v našem primeru `zaokrozi`, nas zanimajo samo, kakšni so pričakovani vhodi in izhodi (metoda `sestej`

ima pričakovan vhod decimalnih vrednosti in pričakovan izhod celih vrednosti). Sama implementacija in logika v integrirani metodi za testirano ni pomembna. Preprosto navidezno predstavo metode dosežemo že z najmanjšo implementacijo metode. Določimo na primer, da ne glede na prejet vhod, metoda `zaokrozi` vedno vrača isto vrednost.

```
1 public int zaokrozi(double vrednost){
2     return 1;
3 }
```

Izsek kode 3.17: Implementacija metode `zaokrozi`

Tako bo lahko testirana koda izvedena skupaj z integrirano. Težava nastane, ko želimo testirati nepričakovane dogodke (npr. napačno zaokroženo število ali sprožitev izjeme). Za doseg takšnega obnašanja, bi za vsak test morali implementirati novo različico metode, kar bi bilo zelo potratno. Tu nam pomagajo orodja, ki so bila posebej narejena za ustvarjanje navideznih objektov. V našem primeru si bomo pomagali z orodjem Mockito, ki smo ga že dodali kot odvisnost v naš projekt v podpoglavju 3.3.1. Ustvarjanje navideznega razreda je enostavno, saj le s pomočjo metode `mock(Object.class)` ustvarimo naš navidezen objekt. Ustvarjanje navideznih objektov s to metodo, nam celoten razred pretvori v navideznega. Za vsako metodo navideznega razreda, ki jo želimo uporabiti, moramo prej določiti, kaj naj vrača. Metoda `mock(Object.class)` je primernejša, ko želimo testirati druge razrede s pomočjo navideznih. V našem primeru bomo uporabili metodo `spy(Object.class)`. Ta nam ustvari kopijo celotnega razreda in ohrani implementirano kodo posameznih metod, kar tudi želimo. Izgled implementacije za naš primer:

```
1 Kalkulator navidezniKalkulator = spy(Kalkulator.class);
```

Izsek kode 3.18: Ustvarjanje navideznega razreda

Metodi navideznega objekta lahko še določimo, kaj naj vrne glede na vhodne argumente.

```
1 // ob vhodu 2.4, vrni 2
2 when(navidezniKalkulator.zaokrozi(2.4))
3                     .thenReturn(2);
4 // ob vhodu 0, vrni -1
5 when(navidezniKalkulator.zaokrozi(0))
6                     .thenReturn(-1);
7 // ob kateremkoli decimalnem vhodu, vrni 4
8 when(navidezniKalkulator.zaokrozi(anyDouble()))
9                     .thenReturn(4);
```

Izsek kode 3.19: Uporaba navideznega razreda

Če sedaj kličemo metodo `zaokrozi` posredno v testih ali pa neposredno v metodi, ki jo vsebuje, nam bo ta vedno vrnila vrednost, ki smo jo definirali. Potrebno se je le še zavedati, da moramo uporabiti isti ustvarjen navidezni razred. Primer testa:

```
1 Kalkulator navidezniKalkulator = spy(Kalkulator.class);
2 when(navidezniKalkulator.zaokrozi(anyDouble()))
3         .thenReturn(2);
4 assertEquals(navidezniKalkulator.zaokrozi(4.6), 2);
```

Izsek kode 3.20: Testiranje z navideznim razredom

Ko določamo, kaj nam bo metoda vračala, lahko uporabimo tudi metodo `doReturn(Object)`, s katero vračamo katerikoli objekt iz metode (npr. namesto celega števila vrnemo decimalno ali pa niz). Primer uporabe:

```
1 doReturn("2.4").when(
2     navidezniKalkulator.zaokrozi(anyDouble()));
```

Izsek kode 3.21: Uporaba metode `doReturn(Object)`

Na ta način preverjamo delovanje metod v nepričakovanih dogodkih. Metode nam ne bi smele vračati napačnih podatkovnih tipov, vendar je lahko tak način testiranja zelo priročen pri testiranju integracij z drugimi programi. Prav tako, kot lahko vračamo druge podatkovne tipe, lahko prožimo tudi izjeme.

```
1 when(navidezniKalkulator.zaokrozi(anyDouble()))
2                     .thenThrow(new IllegalArgumentException());
```

Izsek kode 3.22: Proženje izjeme z navideznim razredom

3.4 Android testi

Android API lahko testiramo le s pomočjo fizične ali navidezne naprave (ang. virtual machine), ki jo lahko ustvarimo s pomočjo IDE Android Studio. Obe možnosti imata svoje prednosti in slabosti. Na voljo imamo več različnih navideznih naprav, ki pa lahko dodatno obremenijo naše računalnike. Testiranje na fizičnih napravah je manj obremenjujoče za naše računalnike, saj je potrebno le napravo povezati z računalnikom s pomočjo USB kabla in Android Studio bo s pomočjo ADB [16] poskrbel za vso komunikacijo. Problem je, da imamo ponavadi na voljo le omejeno število različnih fizičnih naprav. Pri Android testiranju se je potrebno še zavedati, katero najstarejšo podprto verzijo Android podpira naša aplikacija. V primeru, da na naši naprava teče starejša verzija Android od podprte v aplikaciji, na napravi naše aplikacije ne bomo mogli poganjati in testirati.

3.4.1 Priprava testnega okolja

Preden lahko začnemo z Android testi na naši aplikaciji, si moramo pripraviti testno okolje. To pomeni priprava ustreznih knjižnic in priprava testne naprave.

Dodajanje knjižnic Espresso in UiAutomator v aplikacijo

Podobno kot pri javanskih testih v podpoglavju 3.3.1, moramo sprva dodati v aplikacijo potrebne knjižnice, s katerimi bomo pisali testne primere. Te so dodane v projektno konfiguracijsko datoteko okolja Gradle (`build.config`).

```
1 dependencies {
```

```
2     androidTestImplementation
3       'com.android.support.test:runner:1.0.2'
4     androidTestImplementation
5       'com.android.support.test:rules:1.0.2'
6     androidTestImplementation
7       'com.android.support.test.espresso:espresso-core:3.0.2'
8 }
```

Izsek kode 3.23: Dodajanje knjižnic za testiranje platforme Android

Konfiguracija navidezne naprave

Če bomo testiranje izvajali na navidezni napravi, je le to potrebno še dodati v naš IDE Android Studio.

3.4.2 Konfiguracija zaganjača testov

Za zaganjanje Android testov potrebujemo določiti zaganjača testov. Da lahko aplikacijo testiramo, jo moramo pripraviti na poseben način, ki se imenuje instrumentacija (ang. instrumentation). Ko je aplikacija v tem načinu, lahko testi dostopajo do internih metod aplikacije in jih poganjajo med testiranjem. Zato moramo definirati instrumentalnega zaganjača testov (ang. InstrumentationTestRunner) v konfiguracijski datoteki.

```
1 android {
2     defaultConfig {
3         testInstrumentationRunner
4           "android.support.test.runner.AndroidJUnitRunner"
5     }
6 }
```

Izsek kode 3.24: Izbira zaganjača testov

3.4.3 Osnovno testiranje pogledov

Ustvarili bomo aplikacijo, ki bo delovala s pomočjo že obstoječega razreda `Kalkulator` iz prejšnjega poglavja. Najprej na hitro obnovimo glavne funkcionalnosti omenjenega razreda: seštevanje neomejenega števila decimalnih vrednosti, množenje dveh celih števil, sprejemanje samo pozitivnih števil in kot rezultat vračanje zaokrožene vrednosti. Pomembno je še izpostaviti, da smo v naši implementaciji razreda dodali proženje izjem, ko `Kalkulator` ne zna delati z danimi vrednostmi. Da bomo lažje sledili implementaciji, si zamislimo aplikacijo, ki bo imela sledeče komponente:

- Dve vnosni polji, ki sprejemata decimalna števila
- Dva gumba, prvi za operacijo `sestej`, drugi za `zmnozi`
- Eno oznako, v katero bomo izpisovali rezultat izračuna ali morebitne napake pri izračunu

Posamezne komponente, na katere se bomo v testih sklicevali, bomo poimenovali:

- Prvo vnosno polje: `prviOperand`
- Drugo vnosno polje: `drugiOperand`
- Gumb za seštevanje: `gumbSestej`
- Gumb za množenje: `gumbZmnozi`
- Oznaka za prikaz rezultatov: `oznakaRezultat`

Sprva bomo sestavili par testnih primerov, s katerimi bomo preverili osnovno delovanje aplikacije. Želimo se prepričati, da ob vnesenih dveh parametrih, dobimo pravilen izpis rezultata. Za testiranje bomo uporabili knjižnico `Espresso`, katero smo v krajšem primeru že spoznali v podpoglavju 3.2.3. S testnimi primeri bomo sprva le preverili, ali lahko seštejemo in zmnožimo dve celi števili.

```
1 @RunWith(JUnit4.class)
2 public class KalkulatorTest {
3   Kalkulator kalkulator = null;
```

```
4
5       @Rule
6       public ActivityTestRule<KalkulatorActivity> mActivityRule =
7         new ActivityTestRule<>(KalkulatorActivity.class);
8
9       @Before
10      public void setUp() {}
11
12      @After
13      public void tearDown() {}
14
15      @Test
16      public void sestejInZmnoziDveStevili(){
17          onView(withId(R.id.prviOperand))
18                  .perform(typeText("5"));
19
20          onView(withId(R.id.drugiOperand))
21                  .perform(typeText("3"));
22
23          onView(withId(R.id.gumbSestej))
24                  .perform(click());
25
26          onView(withId(R.id.oznakaRezultat))
27                  .check(matches(withText("8")));
28
29          onView(withId(R.id.gumbZmnozi))
30                  .perform(click());
31
32          onView(withId(R.id.oznakaRezultat))
33                  .check(matches(withText("15")));
34      }
35  }
```

Izsek kode 3.25: Testiranje aplikacije

Testni razredi za testiranje Androida so strukturno zelo podobni javanskim. Razlikujejo se samo v dodatnih notacijah, kot je na primer notacija `@Rule`. S pomočjo te notacije, lahko med testiranjem pridobimo instanco aktivnosti

(ang. Activity), ki ga testiramo. Na ta način si omogočimo dostop do katerekoli javne metode in celotnega konteksta aplikacije, s katerim imamo moč dostopati do vseh komponent znotraj aktivnosti. Implementiran test je samorazložljiv. Najprej v vnosni polji vnesemo dve števili in preverjamo, ali se ob pritisku na gumba za seštevanje in množenje izpisujejo pravilne vrednosti v oznaki.

3.4.4 Optimizacije pri pisanju testnih primerov s knjižnico Espresso

Za hitrejšo implementacijo testov, smo si za testno okolje pripravili tudi uporabnostne (ang. Utility) razrede, znotraj katerih smo implementirali kodo knjižnice Espresso. Namen tega razreda je zmanjšanje količine pisanja nepotrebne kode in posledično tudi zmanjšanje možnosti človeške napake. Poglejmo prvi testni primer iz prejšnjega podpoglavja. Da smo lahko pritisnili na gumb, smo morali prej natančno definirati, kaj želimo narediti na katerem pogledu. Če bomo imeli test, ki bo pritiskal na več gumbov, bo na koncu edini del kode, ki bo drugačen od ostale, samo ime gumba. Ravno zaradi tega smo za vse osnovne akcije nad pogledi sestavili njihove enostavnejše različice.

```
1 public static void klikNaGumb(int id) {
2   onView(withId(id))
3     .perform(click());
4 }
```

Izsek kode 3.26: Implementacija metode `klikNaGumb`

Na ta način so naši končni testi izgledali bolj podobni navadnemu jeziku kot programski kodi. Dodatne prednosti takšne implementacije so v tem, da lahko sedaj kdorkoli z manjšim znanjem o imenih komponent sestavi testne primere aplikacije. Lažje tudi svoje teste prilagajamo novejšim implementacijam uporabljenih knjižnic (predpostavimo, da je bila narejena večja sprememba in moramo sedaj posodobiti vse teste, ki klikajo na poglede) ali pa jih enostavno zamenjamo z drugimi, saj v večini primerov potrebujemo

samo ime pogleda, da lahko z njim delamo. Dodatno se seveda lahko tudi nadgradi metodo `klikNaGumb(int)` in sprejmemo kot vhodni parameter ali identifikacijsko ime komponente ali besedilo, ki ga nosi.

```
1 public static void klikNaGumb(Object komponenta) {
2   Matcher identifikacija = null;
3   if(komponenta instanceof Integer){
4     identifikacija = withId((int)komponenta);
5   } else if (komponenta instanceof String){
6     identifikacija = withText((String)komponenta);
7   }
8
9   onView(identifikacija)
10    .perform(click());
11 }
```

Izsek kode 3.27: Izboljšava metode `klikNaGumb`

Sedaj bo metoda sama predpostavila, kateri pogled želimo testirati. Če bo vhodni parameter besedilo, bo pogled poiskala s pomočjo akcije `Matcher` za ujemanje besedila. Če bo vhodni parameter število, bo poiskala pogled z isto identifikacijo. Takšen način implementacije testov ima svoje prednosti, saj v končnem rezultatu hitreje implementiramo teste, lažje popravljamo napake implementacije in povečamo berljivost same kode. Če ponovno izpostavimo prvotno napisani test, Izsek 3.25, lahko vidimo, da je končni rezultat preglednejši in zlahka razumljiv tudi za tehnično nepodkovane ljudi.

```
1 @Test
2 public void sestejInZmnoziDveStevili(){
3   vpisiNiz(R.id.prviOperand, "5");
4
5   vpisiNiz(R.id.drugiOperand, "3")
6
7   klikNaGumb(R.id.gumbSestej);
8
9   preveriVsebinoPogleda(R.id.oznakaRezultat, "8");
10
```

```
11    klikNaGumb(R.id.gumbZmnozi);
12
13    preveriVsebinoPogleda(R.id.oznakaRezultat, "15");
14 }
```

Izsek kode 3.28: Optimizacija testa `sestejInZmnoziDveStevili`

3.4.5 Ustvarjanje testnih poročil

Android Studio že sam ponuja načine ustvarjanja testnih primerov in računanja, koliko kode je bilo izvedene med testiranjem, vendar ne omogoča veliko svobode pri spreminjanju in dodajanju novih informacij v testna poročila. Vsak test znotraj naše aplikacije je označen s posebno kodo, preko katere lahko ob koncu testiranja natančno vemo, kateri testi so bili izvedeni in kaj posamezni testi delajo. S pomočjo objekta `RunListener`, ki se izvaja med testiranjem, lahko zlahka pridemo do potrebnih podatkov, iz katerih potem ustvarimo svoja testna poročila. `RunListener` vsebuje sledeče metode:

- `testRunStarted(Description)`
- `testRunFinished(Result)`
- `testStarted(Description)`
- `testFinished(Description)`
- `testFailure(Failure)`
- `testIgnored(Description)`

Z naštetimi metodami lahko zlahka izvemo, kateri test se izvaja in kakšen je njegov rezultat ter te informacije hranimo, dokler ne zaključimo s testiranjem. Na takšen način smo tudi implementirali ustvarjanje svojih poročil testiranja znotraj projekta. Testno poročilo vsebuje podatke o tem, koliko testov je bilo izvedenih, kateri testi so bili izvedeni, koliko časa je trajalo testiranje in informacijo o uspešnosti testov. Dodatno smo si tudi pomagali pri razreševanju napak neuspešnih testov, saj smo dodali zajemanje zaslonskega okna, ko je zaznan neuspešen test. Na ta način lahko pregledovalec testiranja skupaj z izpisano napako lažje oceni, kje in zakaj je bil test neuspešen.

Poglavje 4

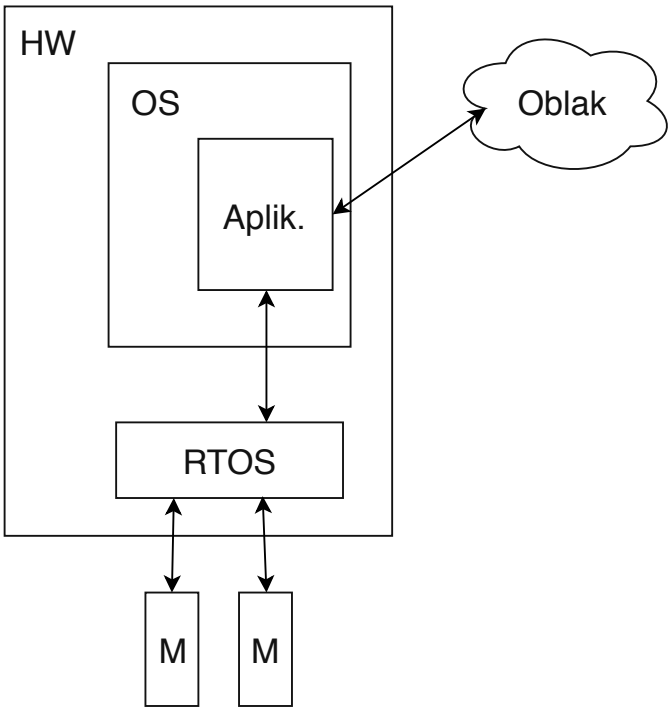
Testiranje v praksi

V sledečem poglavju želimo pokazati, kako se do sedaj zapisane koncepte uporablja v praksi med razvojem programske opreme. Predstavili bomo, kako smo na podjetju MESI, d.o.o. začrtali in zapisali testne primere za aplikacijo Android. V tem diplomskem delu se posebej sklicujemo na testiranje znotraj aplikacij Android, ampak kot bomo izvedeli v podpoglavju 4.1, so bili končni rezultati testiranja razširjeni izven same aplikacije in vsebujejo integracijsko testiranje znotraj platforme.

4.1 Predstavitev platforme

Za bolj natančno razumevanje stališča iz katerega smo sestavljali testne primere, bomo najprej predstavili, kako izgleda platforma produkta podjetja MESI, d.o.o. in kakšno strategijo smo uporabili za testiranje. Kot je razvidno na Sliki 4.1 je glavna izhodiščna točka testiranja aplikacija Android. Kratice na omenjeni sliki pomenijo: HW - strojna oprema, OS - operacijski sistem, RTOS - Real Time Operating System (operacijski sistem v realnem času), M - medicinske diagnostične naprave. RTOS se izvaja poleg operacijskega sistema Android in je odgovoren za takojšno komunikacijo z medicinskimi diagnostičnimi napravami. Aplikacija je bila narejena z namenom olajšanja dela zdravnikov, saj jim omogoča shraniti rezultate odčitovanja EKG, krv-

nega tlaka ali prostornine pljuč s spirometrijo v elektronsko kartoteko bolnika. Vsak zdravnik, in tudi sestre, si lahko ustvarijo svoj uporabniški račun, na katerem so shranjeni njihovi pacienti in meritve. Te si lahko delijo s tem, da se pridružijo istim skupinam in pridobijo istočasen dostop do pacientov in meritev ali preprosto posamezne rezultate meritev delijo z drugimi zdravniki. Vse meritve, skupaj s podatki pacientov, se hranijo v elektronski kartoteki na oblaku. Kot smo že omenili je aplikacija posredno povezana še s spletnim vmesnikom, operacijskim sistemom Android in RTOS, preko katerega komunicira z medicinskimi merilnimi napravami s pomočjo brezžične tehnologije Bluetooth. Zagotoviti, da celotna platforma deluje brez napak, pomeni najprej testirati vse dele platforme posamezno, nato testirati integracije med deli platforme in na koncu še celoten sistem.



Slika 4.1: Platforma produkta podjetja MESI, d.o.o., ki smo jo testirali

Na ta način lahko tudi natančneje določimo, iz kje izvirajo posamezne napake, ki se odkrijejo med testiranjem. Primer: nedelujoča povezava s spletnim vmesnikom lahko pomeni nedelovanje v kateremkoli delu platforme. Težava bi bila lahko v nedosegljivem spletnem vmesniku, implementaciji povezave s spletnim vmesnikom, implementaciji gonilnikov znotraj operacijskega sistema ali nedelujoči strojni opremi.

4.2 Testiranje aplikacije

Testiranje aplikacije je potekalo večinoma v načinu bele in sive škatle. Odločitev implementacije avtomatiziranih testov v aplikacijo je prišla v izvedbo med razvojem samega produkta. To je pomenilo, da programska koda ni bila napisana za testiranje in je bila vedno preverjena le z ročnim testiranjem s strani razvijalcev. Zaradi teh razlogov smo se sprva odločili za testiranje funkcionalnih zahtev aplikacije. Aplikacijo smo preverjali s sistemskimi testi, ki predstavljajo uporabo naprave s strani strank. Vpogled v programsko kodo, skupaj s poznavanjem logike kode in testiranjem s stališča strank, je pomenilo, da bomo izvajali testiranje v načinu sive škatle.

4.2.1 Testiranje v načinu sive škatle

Testni primeri za aplikacijo so bili sestavljeni s pomočjo znanja, kaj se od programske kode pričakuje in kakšni so željeni rezultati. Iz funkcionalnih zahtev so se začeli ustvarjati testni primeri. Na ta način smo pokrili večji del kode in kot ekipa smo imeli večjo samozavest v delovanje same aplikacije, saj so najpogostejše uporabniške poti bile preverjene. Tu so nastali testni primeri, s katerimi smo preverjali osnovne funkcionalne zahteve kot so: prijava uporabnika, registracija novega uporabnika, ustvarjanje novih pacientov in izvedbe meritev z medicinskimi napravami. Primer testa:

```
1 @Test
2 public void ustvariPacienta() {
```

```
3    vpisiUporabnika();
4
5    odpriPogled(USTVARI_PACIENTA);
6
7    izpolniPodatkePacienta("Peter", "Polivc", SPOL_MOSKI,
8      new Date("24.04.1990"), "ZDRST12345, RASA_BELA);
9
10   ustvariPacienta();
11
12   assertTrue(obstajaPacientVBazi("Peter", "Polivc",
13     SPOL_MOSKI, new Date("24.04.1990"),
14     "ZDRST12345, RASA_BELA));
15
16   assertTrue(obstajaPacientNaOblaku("Peter", "Polivc",
17     SPOL_MOSKI, new Date("24.04.1990"),
18     "ZDRST12345, RASA_BELA));
19
20   odjaviUporabnika();
21 }
```

Izsek kode 4.1: Test za ustvarjanje novega pacienta

Sprva je testiranje zelo pomagalo pri samem razvoju produkta. Odkrili smo tudi veliko napak, ki jih z ročnim testiranjem nismo opazili. Sčasoma so se vseeno pokazale pomanjkljivosti takšnega načina testiranja. Izvedeno testiranje je bilo integracijsko testiranje načina veliki pok (glej poglavje 2.2.2), kar pomeni, da je bila celotna platforma produkta sestavljena skupaj in na njej smo preverjali pravilnost izvajanja. Vse odkrite nepravilnosti smo popravljali na ta način, da smo posamezno preverjali, v katerem delu platforme leži težava. Te so bile najpogosteje povezane z ničelnimi kazalci (`NullPointers`) znotraj aplikacije, pogosto smo imeli težave tudi s komunikacijo spletnega vmesnika, ko se vsi podatki niso uspešno sinhronizirali. Popravljanje teh napak je bila najdražja operacija, saj nismo nikoli točno vedeli, kje leži težava. Ko je bila napaka zaznana, je bilo potrebno zagotoviti njeno ponovljivost in hkrati preveriti izvor napake, ali je napaka znotraj aplikacije, ali znotraj samega operacijskega sistema, ali je povezana z nedelujočo strojno opremo,

ali s spletnim vmesnikom.

Ravno tu so ležale največje pomanjkljivosti našega testiranja. Če se vrnemo na piramido (Slika 3.1), smo implementirali samo sistemske teste, kar nam je piramido porušilo. Testni postopek je bilo potrebno popraviti. Bolj natančno, zakaj je število testov na posameznih nivojih pomembno in kako lahko to vpliva na celotno testiranje, bomo analizirali na spodnjem resničnem primeru.

Med testiranjem smo se preveč osredotočili v preverjanje pravilnega delovanja programske opreme. Vsi testni primeri so preverjali osnovno oziroma pričakovano delovanje in ne tudi nepravilne uporabe. Trden primer nepravilne uporabe smo zaznali prav med samim postopkom ustvarjanja pacienta. Nekatere stranke so dodajale nedovoljene znake pred imeni pacientov. Na strani aplikacije smo dovolili ustvarjanje takšnih pacientov in smo poslali zahtevek dodajanja pacienta v oblak. Ta nam tudi ni dodatno preverjal, kakšne podatke pacient vsebuje, in če je zahtevek bil pravilno zgrajen, je dovolil ustvarjanje pacienta. Tako smo si nakopali večje glavobole, saj aplikacija ni bila zmožna delati s pacienti, ki so vsebovali nedovoljene znake. Rezultat takih pacientov je bilo sesutje aplikacije, ko smo jih želeli prikazati. Ta napaka je bila odkrita s pomočjo knjižnice za pošiljanje napak tako da, ko nas je stranka obvestila o napaki, smo sami že delali na rešitvi. Knjižnica za pošiljanje napak deluje tako, da nas vsakič, ko se sproži izjema znotraj aplikacije, obvesti o njej, skupaj z dodatnimi informacijami izjeme (npr. iz kje izvira ta izjema). Konkretna rešitev napake je zelo enostavna, kodo odgovorno za ustvarjanje pacientov, je bilo potrebno razbiti na več testov in to smo tudi naredili. Kodo smo ustrezno razdelili in preuredili, da je postala primerna za testiranje. Napisali smo več testov za različne stopnje testiranja. Sprva so bili napisani testi enot, ki so vse parametre pacientov preverjali posamično s pomočjo različnih vhodnih podatkov (npr. preveri ali pacientov priimek vsebuje nedovoljene znake). Nato smo začeli z integracijski testi. Tu smo šli nivo višje in smo preverjali, kaj nam celoten objekt za ustvarjanje pacientov vrne ob kombinaciji različnih vhodnih podatkov ter tudi, ali dovolimo

ustvariti zahtevek za ustvarjanje pacienta s pomanjkljivimi in nepravilnimi podatki. Na koncu smo dodali tudi več sistemskih testov, s katerimi smo preverjali iste postopke (ustvarjanje pacientov z več različnimi vhodnimi podatki), le da smo sedaj te vnašali iz perspektive končne stranke. S pomočjo razbitja testov na več delov smo hitreje pokrili več testnih postopkov. V končni fazi bi lahko še vedno s sistemskimi testi preverjali, kako se aplikacija obnaša, ko ji podamo različne vhodne podatke, vendar bi na ta način izgubili preverjanje posameznih delov celotnega postopka, kar bi nam lahko prikrilo napake in težje bi iskali vzroke napake.

4.2.2 Testiranje v načinu bele škatle

Zaradi težav in pomanjkljivega testiranja, izpostavljenega v prejšnjem podpoglavju, smo začeli z intenzivnejšim testiranjem posameznih enot znotraj aplikacije. Začeli smo s testi podatkovne baze, natančneje so bili to testi migracije podatkov podatkovne baze. Želeli smo se prepričati, da sprememba strukture v podatkovni bazi ne bo privedla do izgube podatkov uporabnikov. Sploh, ker če ti podatki ne bi bili sinhronizirani, bi stranke lahko izgubile lokalne podatke, kot so narejene meritve ali celo pacienti. Vsakič, ko spremenimo strukturo podatkovne baze, ji definiramo novo verzijo. Migracija podatkov podatkovne baze poteka tako, da preverja ali je trenutna verzija podatkovne baze enaka novi. Podatkovna baza je zgrajena s pomočjo knjižnice OrmLite [17], ki nam sama omogoča, da določimo implementacijo, ko se verziji ne ujemata. V vsaki verziji posebej določimo, kako bomo strukturo spremenili, ali bomo dodali nov stolpec, novo vrstico, ali celo novo tabelo (glej Izsek 4.2), tako da vse, kar moramo v testih preveriti je, ali se struktura ujema v posameznih verzijah (glej Izsek 4.3).

```
1 @Override
2 public void onUpgrade(SQLiteDatabase baza,
3   ConnectionSource povezava, int staraV, int novaV) {
4   if (staraV < 2) {
5       // dodali smo stolpec starost v tabelo pacient
```

```
6       baza.executeRaw("ALTER TABLE 'pacient'
7         ADD COLUMN starost INTEGER;");
8       staraV = 2;
9     }
10    if (staraV < 3) {
11        // dodali smo stolpec spol v tabelo pacient
12        baza.executeRaw("ALTER TABLE 'pacient'
13          ADD COLUMN spol INTEGER;");
14        staraV = 3;
15    }
16  }
```

Izsek kode 4.2: Metoda `onUpgrade`

```
1  @Test
2  public void onUpgrade_Verzija3 () {
3    SQLiteDatabase baza = nadgradiBazo(0, 3);
4
5    // preveri ali obstajata stolpec starost v tabeli pacient
6    assertTrue(obstajaStolpec(Pacient.starost.getName(),
7      Pacient.imeTabele, db));
8
9    // preveri ali obstajata stolpec spol v tabeli pacient
10   assertTrue(obstajaStolpec(Pacient.spol.getName(),
11     Pacient.imeTabele, db));
12 }
```

Izsek kode 4.3: Test metode `onUpgrade`

Glavni del testa je seveda priprava njegova okolja, ki nam tabelo ustvari do željene verzije. To naredimo s pomočjo metode `nadgradiBazo(int od, int do)`. Vse kar nam ta metoda naredi je, da nam ustvari prvo različico podatkovne baze s pomočjo SQL skripte, nad katero potem poženemo metodo `onUpgrade(int, int)`, ki nam to nadgradi do željene verzije. V SQL skripti so napisani ukazi za ustvarjanje tabele z vsemi vrsticami in stolpci, ki so bili v prvi različici podatkovne baze.

```
1 private SQLiteDatabase upgradeDatabase(int from, int to) {
2   PodatkovnaBaza baza = new PodatkovnaBaza();
3   baza = izvediSQLSkripto("bazaV1.sql");
4     baza.onUpgrade(from, to);
5     return baza;
6 }
```

Izsek kode 4.4: Metoda `upgradeDatabase`

Želeli bi izpostaviti, da je lahko priprava podatkovne baze različna za druge platforme. Vse je seveda odvisno od uporabljene knjižnice in implementacije kode. Vendar glavna ideja testiranja ostaja, ne glede na način implementacije podatkovne baze. Vedno pripravimo svojo instanco podatkovne baze, ki jo uporabljamo samo za testiranje, in jo zgradimo do željene verzije ter šele nato začnemo z izvajanjem testov. Zelo je tudi pomembna ponovna uporabnost podatkovne baze, saj je postavljanje baze dolg postopek, ravno zato sprva podatkovno bazo zgradimo in jo nato po vsakem testu počistimo za novo uporabo. Na ta način preverimo, ali so vse spremembe podatkovne baze pravilne in s tem tudi potrjujemo migracije podatkov med različnimi verzijami podatkovnih baz. Za končni test smo dodali še sistemski test, s katerim potrjujemo migracijo podatkov s strani uporabnika. Test poteka tako, da najprej prenesemo prejšnjo verzijo aplikacije s portala git [18] in jo namestimo na napravo. Nato vstavimo nekaj podatkov v podatkovno bazo, to zlahka izvedemo že s prijavo uporabnika. Sledi namestitev trenutne verzije aplikacije. Če se sedaj lahko brez težav premikamo po napravi in so se podatki znotraj podatkovne baze ohranili, smo potrdili uspešno migracijo podatkov. Na ta način smo že odkrili več napak migracij, ki jih med samo implementacijo razvijalci pozabijo preveriti. Rešitev je lahko zelo enostavna in kratka, vendar so rezultati napake lahko katastrofalni za uporabnike.

Testiranje pošiljanja spletnih zahtevkov

Prav tako smo začeli s pisanjem testov enot za vse komunikacijske klice na oblak. Večkrat se nam je zgodilo, da nismo znali natančno diagnosticirati,

na katerem koncu je vzrok za neuspešen spletni klic. Velikokrat je bila težava v aplikaciji, saj smo dovolili pošiljanje zahtevkov tudi brez zahtevanih parametrov. Zato smo začeli več različnih testnih primerov za vsak spletni klic. Testirali smo grajenje zahtevkov s pomanjkljivimi podatki in tudi prejemanje in procesiranje odgovorov iz oblaka. Preverjali smo tudi prejemanje napak iz oblaka in posredovanje teh napak znotraj aplikacije. Lažje delo z napakami nam je omogočila knjižnica `Mockito`, s katero smo lahko simulirali vse možne odgovore oblaka na dane klice. Primer testa:

```
1 @Test
2 public void dodajKomentarMeritvi_VrniNapakoIzOblaka() {
3   EKGMeritev meritev = ustvariLaznoEKGMeritev();
4
5   // ustvari lažni razred PovezavaApi
6   PovezavaApi mApi = spy(PovezavaApi.class);
7
8   String sporocilo = "Napaka pri povezovanju";
9
10  RetrofitError error = ustvariLaznoIzjemo(sporocilo);
11
12  doThrow(error).when(mApi)
13    .dodajKomentarMeritvi(any(EKGMeritev.class));
14  try {
15    mApi.addMeasurementComment(meritev, "Prvi komentar");
16    fail("Izjema ni bila sprozena");
17  } catch (RetrofitError e) {
18    assertEquals(e.getMessage(), sporocilo);
19  }
20 }
```

Izsek kode 4.5: Test za dodajanje komentarja meritvi

4.2.3 Testiranje v načinu črne škatle

Integracijo aplikacije z drugimi moduli smo testirali v načinu črne škatle. Komunikacija poteka s pomočjo RTOS, kateremu aplikacija pošilja definirane

ukaze, sam pa nam sporoča katere pakete je poslal in prejel preko Bluetooth povezave. Namen komunikacije preko Bluetooth povezave v aplikaciji je čim hitrejši in enostaven brezžični dostop do različnih medicinskih diagnostičnih naprav, ki so povezane na napravo. Kot smo že omenili, ima aplikacija dostop do vseh poslanih in prejetih paketov, vendar, ker smo testiranje izvajali v načinu črne škatle, samih paketov nismo preverjali s testi. Preverjali smo le, kako je videti komunikacija z napravami iz uporabniškega vidika. Vsi napisani testi komunikacije Bluetooth so sledili enostavnemu receptu, kot lahko vidimo v Izseku 4.6.

```
1  @Test
2  public void izvediKratkoEKGMeritev() {
3      vpisiUporabnika();
4
5      izberiAplikacijo(APLIKACIJA_EKG);
6
7      izberiPacienta("Peter");
8
9      assertTrue(jeModulVDosegu());
10
11     // izvedi 10 sekundno meritev
12     izvediMeritev(10);
13
14     assertTrue(jePrikazanPogled(REZULTAT_EKG));
15
16     // preveri ali smo prejeli vse podatke
17     assertTrue(preveriPodatke());
18
19     assertTrue(jePrikazanPogled(REZULTAT_EKG_GRAF));
20
21     zapriMeritev();
22
23     odjaviUporabnika();
24 }
```

Izsek kode 4.6: Test za izvajanje EKG meritve

Testirali smo predvsem, ali smo prejeli vse podatke ob pregledu rezultata meritve. Preverjanje, ali so bili vsi podatki prejeti, je zelo enostavno, saj napravi EKG določimo, naj nam pošilja enostaven trikotni signal med izvajanjem meritve. V datoteki rezultatov preverimo, ali je bil shranjen trikotni signal in ali so kakšne neskladnosti v rezultatu. Na ta način smo že odkrili in popravili več primerov, ko smo izgubljali podatke med izvajanjem meritev. Nadaljnji testi komunikacije so bili izvedeni z različnimi filtri in parametri, ter tudi, ali ob različnih klikih, premikih in drugih nenadnih operacijah nad aplikacijo lahko povzročimo, da se podatki neuspešno shranijo v rezultat. Testirali smo enostavno uporabo in izvajanje meritev naprave EKG. Čeprav smo na ta način rešili veliko začetnih težav, smo ponovno padli v lažno samozavest. Težave so se začele pojavljati pri zelo dolgih meritvah, ko se je lahko RTOS, odgovoren za komunikacijo z medicinskimi napravami, znašel v nepričakovanem stanju in je prenehal delovati. Odziv na to napako je bil nenaden, saj je izvajanje meritev ena izmed pomembnejših funkcij naprave in le to nedelovanje lahko privede do nepravočasnih pregledov, ki se lahko končajo za paciente tragično. Vzrok težave smo odkrili v programski kodi RTOS. Zakaj pride do te težave, ponovno zaradi narave testiranja, nismo znali natančno identificirati, kaj šele ponoviti. Prvi korak pri odkrivanju napake so bili testi, ki so izvajali daljše meritve z napravo EKG v bližini. Kasneje smo testiranje še dodatno stopnjevali s premikanjem modula v prostor, kjer je bila slabša povezljivost in tudi, kjer je bilo več komunikacijskih motenj. Izvajali smo neprestane meritve iz dneva v dan. Napako smo nato le odkrili in pojavila se je, ko podatki prvotno niso bili uspešno prejeti in smo zahtevali od naprave EKG, da nam jih ponovno pošlje. Na tej točki, smo zaradi implementacije kode lahko celoten RTOS spravili v kritično stanje, iz katerega se ni znal rešiti. Brezhibno delovanje RTOS smo nato dodatno zagotovili s testi enot znotraj njegove programske kode, prav tako pa tudi na strani aplikacije z intenzivnejšimi stresnimi testi. Stresni testi so dejavnosti, ki določajo robustnost programske opreme s preizkušanjem izven meja normalnega izvajanja [3]. Na ta način smo poskušali odkriti še kakršnokoli

drugo težavo, ki bi se lahko pojavila ob nenadno velikem številu poslanih ukazov, naključnih prekinitev povezave in z izvajanjem različno dolgih meritev. Tako smo odkrili še nekaj komunikacijskih težav in tudi zagotovili pravilno delovanje RTOS preko aplikacije. Dodatno bi lahko še testirali integracijo aplikacije z RTOS na nivoju prejetih paketov, kjer bi preverjali prejete in poslane pakete. Tega nam trenutno sama implementacija kode še ne omogoča, ampak smo na pravi poti, da pokrijemo tudi ta del kode.

4.3 Težave

Ena izmed večjih težav, s katero smo se spopadali med testiranjem produkta, je bila sama programska koda aplikacije. Koda v osnovi ni bila napisana, da bi podpirala njeno testiranje, kar je večkrat oteževalo sestavljanje testnih primerov. Specifično največ težav smo imeli z globalnimi spremenljivkami in strukturo kode. Posamezni testni primeri se morajo izvajati neodvisno od ostalih in čim bolj izolirano. Zelo težko dosežemo neodvisno izvajanje testov, če je potrebno predhodno nastaviti celotno okolje, namesto, da se osredotoči samo na programski del, ki ga želimo testirati.

```
1 public void izvediOperacijo() {
2   if (operacija == OP_SESTEJ) {
3     napaka.setVisibility(View.INVISIBLE);
4     rezultat.setVisibility(View.VISIBLE);
5     rezultat.setText((prviOperand + drugiOperand) + "");
6   } else {
7     rezultat.setVisibility(View.INVISIBLE);
8     napaka.setText("Neznana operacija");
9     napaka.setVisibility(View.VISIBLE);
10  }
11 }
```

Izsek kode 4.7: Metoda z globalnimi spremenljivkami

Izsek 4.7 prikazuje vse uporabljene spremenljivke so globalno dostopne in struktura kode nam meša uporabo pogledov s samo logiko. Napisan testni

primer za zgornjo metodo `izvediOperacijo()` bi zato moral biti zagnan preko navidezne ali fizične naprave in priprave testiranja metode bi morale poskrbeti, da so vse uporabljene spremenljivke ustrezno nastavljene.

```
1 @Test
2 public void testirajOperacijoSestej() {
3   int prvi = 3, drugi = 5;
4   Kalkulator kalk = new Kalkulator();
5   kalk.rezultat = (TextView)getActivity()
6     .findViewById(R.id.rezultat);
7   kalk.napaka = (TextView)getActivity()
8     .findViewById(R.id.napaka);
9
10  kalk.prviOperand = prvi, kalk.drugiOperand = drugi;
11  kalk.operacija = Kalkulator.OP_SESTEJ;
12
13  kalk.izvediOperacijo();
14
15  preveriPrikazanPogled(rezultat)
16    .vidnost(View.VISIBLE)
17    .zVsebino((prvi + drugi) + "");
18  preveriPrikazanPogled(napaka)
19    .vidnost(View.INVISIBLE);
20 }
```

Izsek kode 4.8: Test metode z globalnimi spremenljivkami

Taki programski izseki nam dodatno podaljšajo čas testiranja in implementacijo testa. Kot primer je uporabljen le manjši izsek preproste kode. Predstavljamo si lahko stovrstične implementacije metod, ki sledijo podobnemu receptu in pri testiranju ne vemo več, kje začeti in kje končati z nastavljanjem vseh spremenljivk. Prav tako ne moremo biti prepričani in testirati, na katerem koncu in na kaj se bo določena globalna spremenljivka lahko nastavila med resničnim izvajanjem aplikacije. S testom, glej Izsek 4.8, smo želeli preveriti samo operacijo `OP_SESTEJ`, s katero program sešteje dve števili, a da smo test zagnali, smo sprva morali nastaviti vse poglede in uporabljene

globalne spremenljivke. Čistejša implementacija bi razbila celotno metodo v dva testa, saj želimo preveriti delovanje operacije in pravilno prikazovanje pogledov. Veliko bi pripomogla k sami implementaciji že uporaba metod za dostop do spremenljivk, saj bi na ta način sami določili kakšno instanco spremenljivk naj metoda uporablja med testiranjem.

4.4 Rezultati

Z uvajanjem avtomatiziranega testiranja v aplikacijo smo se veliko naučili. Glavne težave, na katere smo naleteli med pisanjem testnih primerov, smo izpostavili in delali na njihovih rešitvah. Vse nove funkcionalnosti so razvite vzporedno s testiranjem, tako v aplikaciji, kot na celotni platformi. Razvita programska oprema je odprta za testiranje in pisanje testnih primerov ni več tako oteženo, kot je bilo. Skupaj se tudi odločamo, kateri deli novih funkcionalnosti potrebujejo več pozornosti in časa za testiranje. Dodatno smo začeli tudi dodajati vedno več integracijskih testov in tudi testov z navideznimi objekti, kjer zmanjšamo potencialne težave pri integraciji delov platforme. Avtomatizirani testi modulov in integracijski testi se izvedejo ob vsaki spremembi kode. Ob večjih spremembah in pred objavo nove različice programske opreme, se izvedejo tudi sistemski testi, ki nam ustvarijo tudi testna poročila, na katere se kasneje sklicujemo. Še vedno se izvaja ročno testiranje, kjer avtomatizirano ni primerno, primarno so to testi strojne opreme. Tudi te smo začeli olajševati, saj smo pripravili posebne interakcijske teste, kjer posamezna oseba, ki napravo sestavi, sledi navodilom in preverja njeno delovanje. Ob koncu testiranja ima prav tako možnost pregleda uspešnosti testov, kjer se potem presodi, ali je naprava primerna za uporabo. V celotnem obsegu implementacije avtomatiziranega testiranja v aplikacijo je bilo odkritih vsaj dvesto zabeleženih napak. Od tistih, ki so lahko pokvarile uporabniško izkušnjo, do tistih, za katere uporabnik niti ni vedel, da obstajajo. Za vse te napake je bilo ustvarjenih več testnih primerov, s katerimi poskrbimo, da se te napake nikoli več ne ponovijo. Kot je bilo že omenjeno v prejšnjih poglav-

jih, smo v aplikacijo dodatno implementirali knjižnico za beleženje napak. Takšne knjižnice so za razvijalca in osebe, ki testira, ter tudi uporabnika, zelo pomembne, saj nikoli ne moremo predvideti v kakšnem stanju se lahko zgodijo napake na naši napravi. Veliko je tudi izoliranih primerov napak, ki jih popravimo, preden se lahko pojavijo še na drugih napravah. Na ta način smo še dodatno zagotovili, da naša programska oprema deluje brez napak, hkrati daje tudi celotni ekipi samozavest, da se gibljemo v pravo smer.

Poglavje 5

Zaključek

Testiranje programske opreme je zelo težek izziv. Menim, da vsa podjetja, ki razvijajo za širše občinstvo, potrebujejo temeljito testirati svojo programsko kodo. Tu ne gre le za boljšo uporabniško izkušnjo, temveč tudi za ugled in spoštovanje podjetja. Kot smo že izpostavili v podpoglavju 4.3 je testiranje zelo težko implementirati v projekte, če le ta ni bil zasnovan s testiranjem v mislih. Pomembno se je zavedati, kako pristopiti k testiranju in kakšni načini najbolj ustrezajo različnim delom projekta. Prav tako je pomembna tudi količina testov na posameznih stopnjah, kot sem izpostavil v podpoglavju 4.2.1, saj so za različne teste primerni različni pristopi. Cilj testiranja nam ne sme biti določen procent pokrite kode, temveč kvaliteta testirane kode. Pokritost kode nam res poda lažjo predstavo, koliko kode smo testirali, ampak nam žal ne pove, kako natančno je bila koda testirana in, ali napisani testi testirajo pričakovano delovanje kode. Na splošno menim, da je pri razvijanju produkta vedno najboljše imeti vizijo in razmišljati na dolgotrajni razvoj produkta. Tu ne gre le za lažjo in cenejšo implementacijo testov, temveč tudi za učinkovitejšo kodo. Zavedati se moramo, da ne moremo testirati vseh možnih poti znotraj programske opreme. Testirati moramo vsaj večino, v skrajni sili le uporabniške poti, ki so pričakovane, kot najbolj uporabljene pri strankah produkta. Predvsem se želimo s testi prepričati, da programska koda deluje tako, kot smo načrtovali in da bo delala to, kar naše stranke

pričakujejo.

Zaključil bi s citatom Edsger W. Dijkstra, ki je trdil:

> Program testing can be used to show the presence of bugs, but never to show their absence! [19]

oziroma v slovenskem jeziku: testiranje programske opreme naj nam potrjuje prisotnost hroščev in ne njihove odsotnosti.

Literatura

[1] Reto Meier. *Professional Android 4 Application Development*. J. Wiley & Sons, 2012.

[2] Lisa Crispin, Janet Gregory. *Agile Testing: a practical guide for testers and agile teams*. Addison-Wesley, 2009.

[3] Glenford J. Myers Tom Badgett, Corey Sandler. *The Art of Software Testing, 3rd Edition*. John Wiley & Sons, 2011.

[4] Paul Ammann, Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[5] Anne Mette Jonassen Hass. *Guide to Advanced Software Testing*. Artech House Publishers, 2017.

[6] Ian Sommerville. *Software Engineering, Ninth Edition*. Addison-Wesley, 2011.

[7] Gartner Says Worldwide Sales of Smartphones Grew 7 Percent in the Fourth Quarter of 2016. Dosegljivo: `https://www.gartner.com/newsroom/id/3609817`, 2017. [Dostopano: 26. 05. 2018].

[8] The Psychology Underlying the Power of Rubber Duck Debuggin. Dosegljivo: `http://pressupinc.com/blog/2014/06/psychology-underlying-power-rubber-duck-debugging/`, 2014. [Dostopano: 19. 05. 2018].

[9] Agile Testing - Methodologies. Dosegljivo: https://www.tutorialspoint.com/agile_testing/agile_testing_methodologies.htm, 2018. [Dostopano: 28. 05. 2018].

[10] Behavior-Driven Development. Dosegljivo: http://www.codemag.com/article/0805061/, 2008. [Dostopano: 20. 05. 2018].

[11] Fundamentals of Testing. Dosegljivo: https://developer.android.com/training/testing/fundamentals.html, 2018. [Dostopano: 20. 05. 2018].

[12] How Android Code İs Compiled? Dosegljivo: https://medium.com/android-stars/how-android-code-i%CC%87s-compiled-5557cf82ebe9, 2017. [Dostopano: 14. 05. 2018].

[13] Unit tests with Mockito - Tutorial. Dosegljivo: http://www.vogella.com/tutorials/Mockito/article.html, 2017. [Dostopano: 20. 05. 2018].

[14] Espresso. Dostopano: https://developer.android.com/training/testing/espresso/, 2018. [Dostopano: 20. 05. 2018].

[15] UI Automator. Dosegljivo: https://developer.android.com/training/testing/ui-automator, 2018. [Dostopano: 20. 05. 2018].

[16] Android Debug Bridge. Dosegljivo: https://developer.android.com/studio/command-line/adb, 2018. [Dostopano: 22. 06. 2018].

[17] OrmLite - Lightweight Object Relational Mapping. Dosegljivo: http://ormlite.com/, 2018. [Dostopano: 5. 07. 2018].

[18] Git. Dosegljivo https://git-scm.com/, 2018. [Dostopano: 5. 07. 2018].

[19] Edsger W. Dijkstra. *Notes on Structured Programming*. Dosegljivo http://www.cs.utexas.edu/users/EWD/ewd02xx/, 1970. [Dostopano: 20. 07. 2018].