Arhitekturni redizajn mobilnih aplikacija za Android

Matijević, Mislav

Master's thesis / Diplomski rad

2023

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:211:426873

Rights / Prava: Attribution-NonCommercial 3.0 Unported/Imenovanje-Nekomercijalno 3.0

Download date / Datum preuzimanja: **2024-05-18**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mislav Matijević

Arhitekturni redizajn mobilnih aplikacija za Android

DIPLOMSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mislav Matijević

JMBAG: 0016136694

Studij: Informacijsko i programsko inženjerstvo

Arhitekturni redizajn mobilnih aplikacija za Android

DIPLOMSKI RAD

Mentor:

Izv. prof. dr. sc. Zlatko Stapić

Varaždin, ožujak 2023.

Mislav Matijević

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 korištene su etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

U radu se tematizira redizajn softverske arhitekture mobilnih aplikacija za operacijski sustav Android radi poboljšanja dizajna arhitekture tih aplikacija. Nekoliko izvora o temi razvoja softverskih proizvoda uspoređeni su međusobno. Ti teorijski opisi modernih standarda dizajna arhitekture primjenjivi su na različite softverske proizvode, kao i na mobilne aplikacije za sustav Android. Detaljno je izložena moderna dokumentacija za razvoj aplikacija za Android. Dokumentacija se uspoređuje s ostalom literaturom. Istražuju se mogućnosti i načini refaktoriranja i arhitekturnog redizajna postojećih programskih proizvoda u svrhu poboljšanja njihove arhitekture i strukture. U praktičnom dijelu rada analiziraju se dobri arhitekturni uzorci, ali i nedostatci dizajna jedne mobilne aplikacije za Android koja u stvarnome svijetu ima tisuće korisnika. Primjenjuje se proces refaktoriranja i arhitekturnog redizajna u svrhu poboljšanja odabrane aplikacije. Cilj je ovoga diplomskog rada teorijski opisati moderne standarde dizajna arhitekture mobilnih aplikacija za Android te istražiti mogućnosti i načine refaktoriranja i arhitekturnog redizajna postojećih programskih proizvoda u svrhu poboljšanja njihove arhitekture i strukture.

Ključne riječi: Android, arhitektura aplikacije, Java, redizajn, refaktoriranje, analiza kôda.

Sadržaj

Sadržaj iii
1. Uvod 1
2. Metode i tehnike rada 4
3. Redizajn softverske arhitekture 5
3.1. Refaktoriranje u kontekstu održavanja softvera 5
3.2. Najbolje prakse refaktoriranja 8
4. Arhitekture mobilnih aplikacija 11
4.1. Dobre arhitekture softverskih proizvoda 11
4.2. Obilježja arhitekture aplikacija za Android 15
4.2.1. Modularnost 17
4.2.2. Predložena troslojna arhitektura 18
4.2.2.1. Sloj korisničkog sučelja 20
4.2.2.2. Sloj domene 23
4.2.2.3. Sloj podataka 24
4.3. Zaključak o dobroj arhitekturi aplikacije za Android 26
4.4. Poteškoće razvoja arhitekture za Android 27
5. Analiza postojeće aplikacije 29
5.1. Pregled funkcionalnosti 29
5.2. Prepoznati uzorci u arhitekturi 33
5.2.1.1. Arhitekturni uzorci 33
5.2.1.2. Uzorci dizajna 34
5.3. Kritike korištenog dizajna 36
5.3.1. Uočeni nedostaci projekta 36
5.3.2. Automatizirana statička analiza projekta 42
6. Refaktoriranje i arhitekturni redizajn aplikacije 50
6.1. Redizajn načina prikaza kartica 51
6.2. Uvođenje modularnosti za prijavu korisnika 67
6.3. Osvrt na proces redizajniranja arhitekture 78
7. Zaključak 79
Popis literature 80
Popis slika 82
Popis tablica 82
Popis programskih kôdova 83

1. Uvod

U ovom poglavlju navodi se motivacija za pisanje rada, opisuju se ciljevi istraživanja literature, opseg istraživanja, metodologija te struktura rada.

Martin Fowler u knjizi „Refactoring: Improving the Design of Existing Code“ (2019) navodi da svatko može pisati kôd koji računalo razumije, ali da dobri programeri pišu kôd koji ljudi mogu razumjeti.

Taj navod objašnjava zašto je važno obaviti refaktoriranje. Fowler pojam refaktoriranja definira kao „proces mijenjanja softverskog sustava na način koji ne mijenja vanjsko ponašanje kôda, a popravlja mu unutrašnju strukturu“ (Fowler et al., 2019). Drugi autori, po uzoru na Fowlera, definiraju refaktoriranje kao „zadatak u sklopu održavanja u kojemu se izvorni kôd restrukturira kako bi mu se poboljšala kvaliteta dok se vanjsko ponašanje sustava održava“ (Al Dallal & Abdin, 2018) ili kao „disciplinirani tehniku za restrukturiranje postojećeg tijela kôda, mijenjajući njegovu internu strukturu bez mijenjanja vanjskog ponašanja“ (Thomas & Hunt, 2019). Naime, riječ je o prolasku programskim kôdom i pronalasku nedostataka u arhitekturi rješenja, što uključuje nazive i smještaj klasa i metoda, smjerove komunikacija među objektima klasa, smještaj i smisao pojedinih komponenti u kontekstu modularnosti rješenja itd. Moglo bi se zaključiti da je glavna premisa refaktoriranja „efektivno ulaganje vremena i novca kako bi se uštedjelo vrijeme i novac u budućnosti“ (Wangberg, 2010).

Bjarne Stroustrup smatra „da bismo trebali tražiti više elegancije u izgrađenim aplikacijama nego u samim jezicima“ (Stroustrup, 2023). Nije potrebno koristiti se najmodernijom tehnologijom, već pomoću korištene tehnologije izraditi sustav koji je održiv i nad kojim se mogu izvršiti promjene, a da ga se ne destabilizira. Upravo to je važno u kontekstu ovoga rada. Nisu jednake dobrobiti korištenja modernog sintaktičkog šećera poput asinkronih metoda i lambda funkcija kao i ispravno praćenje SOLID[1] principa u razvoju programa.

Iz svega navedenog može se zaključiti da je refaktoriranje iznimno važno pri upravljanju velikim projektima. Tu do izražaja dolaze mobilne aplikacije. One su često jasno definirane aplikacije u čvrstim okvirima. Međutim, nerijetko ovise o izrazitoj modularnosti. Za primjer se može navesti prijava u sustav. Današnji pametni uređaji prijavu korisnika mogu izvršiti pomoću

[1] SOLID principi spominju se u poglavlju 4.1.

interneta koristeći se e-poštom ili QR kôdom uz kameru, ali se također mogu koristiti prepoznavanjem lica, otiskom prsta, unosom PIN znamenki itd. Navedeni primjeri prijave korisnika jasno pokazuju koliko je nužno što učinkovitije implementirati modularnost kako bi se neki novi princip prijave što jednostavnije implementirao. Nadalje, nakon prijave, važno je ostvariti jasnu komunikaciju između objekata klasa kako bi se korisničkim podatcima uvijek jednostavno pristupalo bez zahtijevanja mnogo resursa. Specifikacije dijagrama poput UML-a u ovakvim situacijama nisu dovoljne za potpuno razumijevanje kôda pri njegovu održavanju ili nadograđivanju mnogo godina poslije. Potrebno je postaviti uistinu kvalitetnu arhitekturnu podlogu kako bi sustav bio jednostavno održiv uz sve svoje specifičnosti.

Android je odabran kao tehnologija razvoja koja se proučava u kontekstu refaktoriranja kôda i najboljih praksi za ovaj rad. Jedan od razloga jest to što je Android vrlo otvoren sustav, čiji se skup biblioteka za komuniciranje sa sustavom, tzv. SDK (engl. *Software Development Kit*) vrlo transparentno mijenja tijekom više od deset godina. Nadalje, ne manje važan razlog jest iznimna objektna orijentiranost te platforme. U sustavu prevladavaju različiti uzorci dizajna, a sam razvoj zahtijeva vrlo dobro poznavanje objektna orijentiranih principa razvoja programa i mnogo promišljanja o ulozi pojedinih klasa i objekata u sustavu.

Rad je koncipiran na način da se najprije opisuje refaktoriranje u kontekstu redizajna kao postupka, a potom se na izvornom kôdu postojeće aplikacije za Android napisane u programskom jeziku Java analiziraju nedostatci i primjenjuje arhitekturni redizajn dviju osnovnih funkcionalnosti u svrhu jednostavnijeg održavanja i daljnje nadogradnje aplikacije. Jezik Java više nije preporučeni jezik za razvoj aplikacije za Android pokraj modernijeg Kotlina, ali ako se ima na umu ranije spomenuta izjava Bjarna Stroustrupa, to nije problem. Dapače, Java je mnogo stroža što se tiče objektno orijentiranog pristupa te omogućuje mnogo širi pogled na uočene probleme koji se potom može primijeniti i na kôd pisan u okolini koja nije vezana uz mobilni razvoj.

Redizajn obuhvaća refaktoriranje jer u suprotnom nema poboljšanja strukture kôda. Refaktoriranje je vrlo opsežan pojam koji podrazumijeva postupak koji se može provoditi na različite načine. Stoga, prvi cilj ovoga rada jest približiti pojam refaktoriranja. Drugi je cilj rada predočiti arhitekturu postojeće aplikacije koja se koristi u realnom svijetu. Treba naglasiti da u ovom kontekstu cilj nije baviti se sustavom Android niti razvojnom okolinom Android Studio. Naglasak je samo na konceptima koji čine temelj modernog objektno orijentiranog programiranja te modernog modularnog razvoja aplikacija za mobilne uređaje.

Što se tiče strukture rada, nakon uvoda i opisa korištenih metodologija i tehnika, rad čine četiri glavna dijela:

1. Opis pojma refaktoriranja u kontekstu redizajna arhitekture
2. Pregled najboljih praksi za arhitekturu mobilnih aplikacija
3. Analiza arhitekture postojeće aplikacije
4. Arhitekturni redizajn postojeće aplikacije

U prvom dijelu rada daje se pregled literature s temom refaktoriranja kôda aplikacije. Refaktoriranje je nužno detaljno objasniti kako bi bio objašnjen sam smisao arhitekturnog redizajna. U prvome dijelu rada naglasak je na teorijskom okviru. Nudi se pregled najboljih praksi održavanja softverskog proizvoda.

U drugom dijelu rada naglasak je na općoj arhitekturi softvera te se daje pregled moderne arhitekture aplikacija pisanih za sustav Android. Za razliku od prvoga dijela, ovdje je više naglasak na dobrome temelju aplikacije nego na dobroj mogućnosti održavanja njezina kôda. Taj se dio rada u dobrome dijelu odnosi na službenu dokumentaciju za razvoj na Androidovoj platformi, uvod je u modularnost programskog rješenja, u troslojnu arhitekturu u kontekstu razvoja za sustav Android te u specifične poteškoće razvoja za sustav Android.

U trećem dijelu rada analizira se stanje postojeće javno objavljene aplikacije s mnoštvom korisnika koja pripada jednoj organizaciji. Poglavlje upućuje u aplikaciju koja je odabrana da se na njoj izvrši arhitekturni redizajn. Izvršava se pregled same aplikacije i njezinih mogućnosti, navode se dobri arhitekturni uzorci korišteni u aplikaciji te nedostatci dizajna u aplikaciji.

Četvrti dio rada odnosi se na arhitekturni redizajn odabrane i opisane aplikacije. Koristi se sve utvrđeno kako bi se dao bolji uvid u razloge svakog pojedinog koraka redizajna (promjene kôda). Opisuje se provođeni proces refaktoriranja, implementacija uzoraka dizajna, ostvarivanje modularnosti u projekatima za sustav Android te se, konačno, prikazuje rezultat arhitekturnoga redizajna provedenoga na dvjema važnim funkcionalnostima aplikacije.

Nakon svega slijedi zaključak koji sažima istraživačke ciljeve i rezultate ovoga rada, nudi se rasprava o rezultatima rada te konačni zaključci.

2. Metode i tehnike rada

U prethodnom je poglavlju iznesen uvod u obliku motivacije za pisanje rada kao i opis teme i struktura rada. U ovome poglavlju opisuje se kojim je metodama i tehnikama rad sastavljen u logičku cjelinu.

Prije nego što je sadržaj bio razrađen, istražena je okvirna literatura relevantna za ovo područje. Literatura je obrađena tako da je čitana uz stvaranje bilješki, koje su potom uspoređene i izložene u radu. Jedna od glavnih okosnica jest knjiga Martina Fowlera „Refactoring: Improving the Design of Existing Code“, u kojoj autor zastupa tezu da je refaktoriranje ključni element cijelog procesa softverskog razvoja. Drugi važan izvor jest knjiga „Clean Architecture“ autora Roberta Cecila Martina. On opisuje principe dizajniranja softvera te daje detaljna obrazloženja kako bi arhitektura softverskog proizvoda trebala izgledati. Jednako važna jest i službena dokumentacija za razvoj za Android, dostupna na sljedećem mrežnom mjestu: https://developer.android.com/topic/. Ta dokumentacija najažurniji je izvor podataka o tome kako dizajnirati aplikacije za sustav Android. U radu su na više mjesta uspoređeni navodi u dokumentaciji s odabranom literaturom. Za pretraživanje relevantnih izvora koji nisu knjige korišten je sustav Google Scholar.

Što se tiče korištenih istraživačkih metoda, prvi dio rada sadržava izviđajno istraživanje, pri čemu se proučava ono što je već poznato o temi, čime se objašnjava teorija vezana uz kontekst rada. U radu se daje primijenjeno istraživanje prepoznavanja rješenja vezanog za problem usklađivanja dvaju različitih prikaza i ugrađivanja dviju različitih mogućnosti postizanja istoga funkcionalnog zahtjeva prijave u aplikaciju. Nadalje, u radu se daje temeljno istraživanje utjecaja arhitekturnog redizajna na jednostavnost proširenja implementacije bilo kojeg softverskog proizvoda (ne nužno samo mobilnih aplikacija).

Od tehničkih alata korišteno je razvojno okruženje Android Studio (verzija Electric Eel 2022.1.1 Patch 1) i alat Git za kloniranje postojeće aplikacije te verzioniranje kôda tijekom obrade. Korištena je platforma GitHub za javnu pohranu kôda. Kôd koji prati ovaj materijal dostupan je na privatnom repozitoriju: https://github.com/zstapic/obitelji3plus_mislav. Za izradu dijagrama korišteni su alati Draw.io i Visual Paradigm (verzija 17.0). Za obradu slika korišten je alat GIMP (verzija 2.10.22). Za organizaciju relevantne literature korišten je alat Zotero.

3. Redizajn softverske arhitekture

U ovom se poglavlju daje uvid u literaturu koja je korištena za kasnije redizajniranje, a koju je nužno poznavati kako bi kontekst rada bio jasan.

Model kvalitete softvera po standardu ISO/IEC 25010 definira održivost kao „razinu efektivnosti i efikasnosti kojom se proizvod može mijenjati radi poboljšanja, ispravljanja ili prilagodbe promjenama u okolini i u zahtjevima“ (Malavolta et al., 2018). Taj standard definira nekoliko obilježja softvera po kojima mu se može mjeriti održivost: modularnost kao stupanj utjecaja promjene komponente na ostatak softvera, ponovna iskoristivost, količina promjena koje ne dovode do pada kvalitete, mogućnost proučavanja radi identificiranja dijelova koje treba mijenjati te omogućena mjera testiranja (*ISO 25010*, 2022; Malavolta et al., 2018).

Održavanje softvera ne mora uključivati redizajn cijele arhitekture. Ako je arhitektura dobro postavljena, redizajn nije potreban pri promjenama. Međutim, ako je arhitektura postavljena loše, bez promišljanja o budućnosti projekta, što se najčešće događa zbog pritiska rokova, onda je redizajn nužan. U nastavku se objašnjava zašto je tomu tako. Sljedeći odlomak bavi se pojmom refaktoriranja, koje je od iznimne važnosti u održavanju softvera.

3.1. Refaktoriranje u kontekstu održavanja softvera

Ovdje se daje uvid u problematiku arhitekturnog redizajna aplikacija uz prikaz raznih koncepata. Ti su koncepti ono što treba imati na umu prilikom postavljanja i pregleda arhitekture modernih aplikacija, a pomažu programerima jasno prepoznati probleme i pratiti ispravan put njihova rješavanja.

Autori Andrew Hunt i David H. Thomas u knjizi „The Pragmatic Programmer“ (2019) negoduju što je graditeljstvo najčešća metafora za razvoj softvera. Smatraju da je analogija neispravna zbog toga što nakon izgradnje i useljenja stanari samo povremeno zovu radnike za održavanje, onda kada nastane neki problem. Nadalje, smatraju da poslovnim ljudima više odgovara metafora gradnje zgrada jer je znanstvena, ponovljiva, postoji hijerarhija izvještavanja za poslovodstvo itd. Autori, pak, ističu da je softver više poput održavanja vrta, tj. da je više organski nego konkretan. „Posadite mnogo biljki u vrtu s obzirom na inicijalni plan i uvjete. Neke uspiju, neke postanu kompost. (…) Čupate korov i fertilizirate biljke kojima treba dodatna pomoć. Konstantno pratite vitalnost vrta uz dorade (zemlje, biljki, rasporeda) po potrebi.“ (Thomas & Hunt, 2019)

Demeyer i sur. (2009) u svojoj knjizi „Object-Oriented Reengineering Patterns“ navode specifične razloge zbog kojih bi netko želio refaktorirati softver, poput *otpakiranja* monolitnog sustava, poboljšanja performansi, prebacivanja sustava na novu platformu, izoliranja dizajna prije nove implementacije, iskorištavanja nove tehnologije, poput novih standarda ili biblioteka, čime održavanje postaje jednostavnije. Što se tiče poboljšanja performansi, tu nalažu da „prvo treba učiniti, zatim ispraviti, a potom ubrzati“, što se podudara sa stavom Martina Fowlera (2019) o odnosu optimizacije i refaktoriranja. Demeyer spominje izreku „Ako nije pokvareno, ne popravljaj“ i naglašava problem da postoji više načina da nešto bude pokvareno. Navodi i simptome pokvarenosti kôda, poput zastarjele ili nepostojeće dokumentacije, što smatra jasnim znakom da je zastarjeli sustav prošao mnogo promjena te da će se problemi izroditi čim originalni razvojni tim napusti projekt. Zatim spominje nedostajuće testove kao uzrok visokog rizika ili troška tijekom razvitka projekta.

Ostali simptomi pokvarenosti kôda jesu sljedeći: odlazak originalnih razvojnih inženjera, nedostatak razumijevanja unutrašnjosti sustava, ograničeno razumijevanje sustava u cjelini, dug proces puštanja procesa u produkciju, predugo vrijeme potrebno za jednostavne promjene ili za izgradnju sustava, potreba za konstantnim ispravljanjem pogrešaka, ovisnosti održavanja koje uzrokuju nastanak pogrešaka ispravljanjem pogrešaka, poteškoće u razdvajanju proizvoda, duplicirani kôd te *smrdljivi* kôd (engl. *code smell*).

Što se tiče pojma „smrdljivog kôda“, Demeyer se referencira na knjigu Martina Fowlera (2019) i dijela knjige na kojemu je surađivao Kent Beck, a o kojoj će više riječi biti poslije u radu. Demeyer dodatno opisuje citiranu izreku u kontekstu sljedećega problema: „Koje dijelove zastarjelog sustava treba redizajnirati, a koje ostaviti takvima kakvi jesu?“ (Demeyer et al., 2009) Njegovo je rješenje tog problema ispraviti samo dijelove koji više ne mogu biti prilagođeni planiranim promjenama, poput komponenti koje su izložene čestim promjenama, a u isto ih je vrijeme teško mijenjati zbog visoke kompleksnosti i neodgovarajućeg dizajna. Također navodi i kvarne komponente koje nose vrijednost, iako ih možda nije potrebno često mijenjati (Demeyer et al., 2009).

Tema ispravljanja nečega što nije pokvareno nije ključna za srž ovoga rada, kao ni za redizajniranje arhitekture postojeće aplikacije razvijene za Android, ali svakako jest važna radi predstavljanja dodane vrijednosti iza samoga refaktoriranja. Jasno je da je glavna problematika refaktoriranja neuviđanje potrebe za promjenama ili pogrešan pogled na refaktoriranje kao nešto što „oduzima vrijeme“, umjesto kao na nešto što će omogućiti brže daljnje razvijanje sustava. Nadalje, sustav na kojemu dugo nije bilo izvršeno refaktoriranje (primjerice zato što „radi dobro“ ili što se „nema vremena“) može biti odbojan novim

programerima, koji se naknadno uključuju u projekt razvijanja, te oni mogu izgubiti motivaciju za razvijanjem takvog sustava ili čak mogu zbog frustracije napustiti projekt/organizaciju. Ovo je područje u kojemu bi se mogli dalje istraživati stavovi programera koji rade na održavanju i razvijanju zastarjelih projekata sa *smrdljivim* kôdom. No takvo istraživanje izlazi iz okvira ovoga rada te stoga nije provedeno.

Jedna proučena empirijska analiza utjecaja refaktoriranja na kvalitetu softvera, autora Al Dallala i Abdina (2018), provedena na 76 relevantnih istraživanja, upozorava na to da pomicanje metoda, vađenje klasa i vađenje metoda poboljšava održivost i koheziju, ali povećava kompleksnost i povezivanje. Izvršena empirijska analiza istraživanja ističe ponajprije održivost, a potom razumljivost, ponovnu iskoristivost te pouzdanost kao vanjske dobrobiti izvršenog refaktoriranja, dok bi dobrobiti za kôd bile povezivanje komponenti, kohezija i opće smanjenje kompleksnosti kôda (Al Dallal & Abdin, 2018).

Kako bi odredili kvalitativne atribute, autori Al Dallal i Abdin referirali su se na autora Sandra Morascu i njegov znanstveni članak o pristupu mjerenju vanjskih atributa softverskih artefakata, kao i na autore Fentona i Pfleegera i njihov rad „Software Metrics: A Rigorous & Practical Approach“, te su kvalitativne atribute podijelili u dvije kategorije:

- interni kvalitativni atributi
    - mjerljivi, dovoljno je koristiti artefakte kôda, npr. veličina je mjerljiva brojenjem linija kôda (Morasca, 2009)
    - obilježava ih kohezija, povezivanje (engl. *coupling*), strukturna kompleksnost, veličina
- eksterni kvalitativni atributi (Morasca, 2009)
    - nije dovoljno koristiti artefakte kôda, već softverski inženjeri moraju razmatrati komunikaciju između softverske okoline (Al Dallal & Abdin, 2018 prema Morasca, 2009)
    - pouzdani su, korisni, održivi, prenosivi… (Morasca, 2009)
    - promjenjivost softverskog programa ovisi i o programu, ali i o vrsti promjena, timu te okolini promjene – stoga, treba uzeti u obzir više faktora uz softverski artefakt (Morasca, 2009).

Što se tiče ranije spomenute empirijska analize, autori Al Dallal i Abdin (2018) uočili su kontradiktorne utjecaje na atribute kvalitete u kontekstu u kojemu neki scenariji refaktoriranja poboljšavaju jedne aspekte i slabe druge. Međutim, autori vjeruju da nije ispravno ograničiti

istraživanje utjecaja određenog scenarija refaktoriranja na kvalitetu određenog atributa kvalitete, jer bi u tom slučaju utjecaj stvaranja metode u kôdu izravno utjecao na veću kompleksnost kôda, a ne na potencijalno smanjenje vrijednosti vezivanja komponenti i veličine kôda (Al Dallal & Abdin, 2018).

Autori Al Dallal i Abdin upućuju na nekoliko uzroka za koje vjeruju da objašnjavaju slične nelogičnosti na koje su naišli, a koje stoje u kontradikciji toga da refaktoriranje nužno poboljšava kvalitetu softvera. Za prvi uzrok navode nedostatak dogovora oko definicije atributa kvalitete softvera, pristupa mjerenju i razmatranih artefakata kôda. Za taj uzrok autori navode primjer stvaranja podklase iz kôda te kako se s obzirom na različita mjerenja kohezije dobiva drukčiji utjecaj ovog refaktoriranja na koheziju kôda, pa čak i drukčija definicija kohezije. Drugim uzrokom smatraju nedostatne tehnike identifikacije prilika za refaktoriranjem, što vodi do nepotrebnog refaktoriranja te smanjenja kvalitete kôda. Za treći uzrok navode različite tehnike refaktoriranja koje za isti problem nude različito rješenje, a stoga stvaraju opasnost od pogrešne odluke koja može neočekivano narušiti kvalitetu kôda. Konačni četvrti identificirani uzrok kontradikcija u proučavanju rezultata refaktoriranja nalazio bi se u nedostatnim alatima za refaktoriranje koji mogu proizvesti neispravno refaktorirane dijelove kôda i narušiti kvalitetu kôda (Al Dallal & Abdin, 2018).

Autor Ruben D. Wangberg (2010) u svojemu je radu upozorio na nedostatak komunikacije i suradnje između programera razvojnih okruženja koji implementiraju potporu refaktoriranju kôda i znanstvenika koji troše mnogo vremena istražujući potencijalni *smrdljivi* kôd. Iako je od nastanka njegova rada prošlo već trinaest godina, važno je naglasiti da uočeni nedostatak svakako nije bezazlen. Riječ je o usklađivanju znanstvenog djelovanja (istraživanje *smrdljivog* kôda) i vrlo važne komponente postavljanja aplikacije u rad poput statičke analize kôda.

3.2. Najbolje prakse refaktoriranja

U prethodnom je poglavlje dan širi uvid u to što je redizajn softvera. U ovom se poglavlju sužava kontekst na konkretne savjete iz literature oko obavljanja tog procesa.

Kad je riječ o konkretnoj primjeni refaktoriranja kôda, u vezi s tim ističe se već spomenuti autor Martin Fowler (2019) s knjigom „Refactoring: Improving the Design of Existing Code“. U njoj se zalaže da se prije procesa refaktoriranja pripremi niz testova te izvršenje refaktoriranja provede nizom malenih koraka. Testovi će osigurati da funkcioniranje programa ostane nepromijenjeno nakon izmjena, a maleni koraci poduprijet će jednostavno otklanjanje

eventualnih nastalih grešaka. To su promjene poput preimenovanja varijabli i metoda, dodavanje nove lokalne metode, izbacivanje lokalne varijable itd. Upravo te malene promjene autor naglašava kao srž procesa refaktoriranja, uz obvezna testiranja i verzioniranje nakon svake takve promjene. Fowler smatra da nije nužno pisati brz softver, već softver koji se može ubrzati na dovoljnu brzinu, što spominju i autori Demeyer i sur. (2009). Nadalje, naglašava važnost dobrog imenovanja varijabli kako bi kôd što bolje komunicirao svoju svrhu, a i uklanjanje privremenih varijabli zbog toga što „podupiru dugačke i kompleksne“ funkcije. Što se tiče teorijske podloge, Fowler ističe da različita značenja imaju imenica *refaktoriranje* i glagol *refaktorirati*. Po njemu, imenica se odnosi na promjenu na internoj strukturi softvera kako bi ga bilo jednostavnije razumjeti i jeftinije preinačivati bez vidljive promjene u ponašanju, a glagol se odnosi na promjenu strukture softvera korištenjem niza refaktoriranja bez mijenjanja vidljive promjene u ponašanju. „Možete potrošiti nekoliko sati refaktorirajući tijekom kojih primijenite nekolicinu individualnih refaktoriranja“ (Fowler et al., 2019). Fowlerov koautor Kent Beckt, tvorac metodologije ekstremnog programiranja, u knjizi iznosi sljedeće: „Programi imaju dvije vrste vrijednosti: što vam mogu učiniti danas i što će vam moći učiniti sutra. (…) Nije moguće dugo programirati bez shvaćanja da je ono što sustav radi samo dio priče. Ako dovršiš današnji dio posla sutra, ali na način da nećeš moći sutra dovršiti sutrašnji posao, onda gubiš. (…) Refaktoriranje je izlaz iz ovoga ograničenja. Kada shvatiš da jučerašnja odluka nema smisla danas, promijeniš odluku. To omogućava odrađivanje današnjeg posla.“ (Fowler et al., 2019) Fowler ovo spominje u kontekstu izvršenja refaktoriranja tijekom pregledavanja kôda (ostale situacije za refaktoriranje jesu dodavanje funkcionalnosti i ispravak pogreški).

Fowler poručuje da refaktoriranje tijekom pregleda kôda omogućuje istinski uvid u djelovanje kôda uz dolazak do novih ideja, za razliku od samog čitanja kôda i djelomičnog razumijevanja uz davanje savjeta. Naglašava da kod većih pregleda dizajna preferira UML dijagrame. Citirao je Kenta Becka kao tvorca ekstremnog programiranja koje uključuje programiranje u paru (engl. Pair Programming), nešto što omogućuje refaktoriranje prilikom razgledavanja kôda (Fowler et al., 2019). Autor ponovno citira Kenta Becka, ovaj put o tome kako refaktoriranje teži razbiti velike objekte i metode u više manjih, čime se uvodi zaobilaženje. To zaobilaženje smatra da je isplativo jer omogućava dijeljenje logike, odvaja objašnjenje smisla i implementaciju, izolira promjenu i prikriva (engl. *encode*) uvjetnu logiku pomoću polimorfizma. Naravno, navodi se da postoje i rjeđe situacije kada zaobilaženje nije isplativo te se uklanja refaktoriranjem, uz zadržavanje iste razine kvalitete.

Što se tiče sučelja, Fowler ih smatra problematičnima u kontekstu refaktoriranja kada je u pitanju promjena naziva metode (engl. *Rename Method*). Njihova izmjena može uzrokovati probleme u kôdu koji nije dostupan, poput kôda korisnika tog aplikacijskog programskog

sučelja. Fowler takva sučelja naziva objavljenima (engl. *published*) te savjetuje da je nužno zadržati i staro i novo objavljeno sučelje dovoljno dugo da korisnici dobiju priliku reagirati na promjenu (Fowler et al., 2019). Kad je riječ o razvoju za Android, nerijetko se naiđe na zastarjele (engl. *deprecated*) metode unutar službenog SDK[2] (engl. *Software Development Kit*), koje su zaostale iz prethodnih verzija Androida. Razvojni inženjeri u Googleu takve metode označavaju zastarjelima i zadržavaju ih do 12 mjeseci, navodeći da trajno uklanjanje takvih značajki ili metoda omogućuje timu da poboljša kvalitetu proizvoda, smanji održavanje proizvoda te postigne veću vrijednost za korisnika ako je riječ o nekompatibilnosti s novim značajkama koje Google želi objaviti (*Deprecations* | *Android Enterprise*, 2022).

Doduše, Fowler daje i primjer kada ne treba refaktorirati, već ponovno krenuti u izradu aplikacije, a to je onda kada je kôd u tolikom neredu da ne radi. Ističe da je važno da kôd radi prije refaktoriranja. Ali za sve ostale situacije smatra da izlika za izbjegavanje refaktoriranja ne može biti blizina roka projekta, jer nedostatak vremena smatra znakom da je potrebno refaktoriranje (Fowler et al., 2019).

[2] skup biblioteka za nazivni razvoj za Android

4. Arhitekture mobilnih aplikacija

U ovom se poglavlju prikazuju postojeće arhitekture softverskih proizvoda te najbolje prakse u postavljanju arhitekture aplikacije i same strukture direktorija mobilnih aplikacija za Android. Važno je naglasiti da se to ne odnosi samo na mobilne aplikacije, već da je opći uvid u dobre prakse.

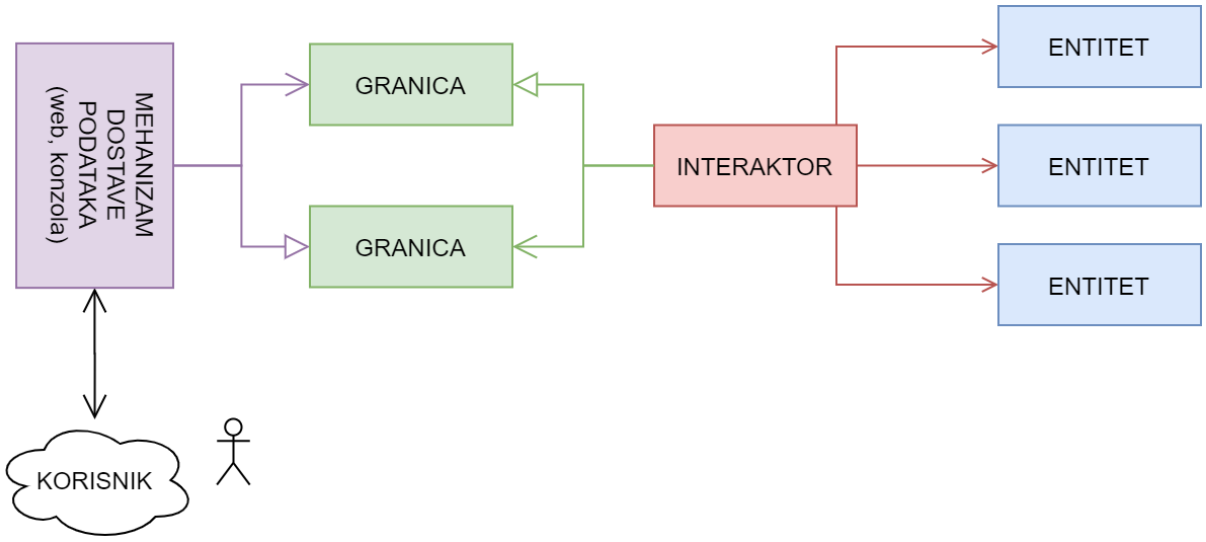
4.1. Dobre arhitekture softverskih proizvoda

Ovdje će biti riječi o osnovnim pojmovima koji se vežu uz arhitekturu softvera. Mnogo citata iz relevantnih izvora koji se bave ovim područjem potkrepljuje definiciju dobre arhitekture softverskih proizvoda. Prije bavljenja samom arhitekturom, dan je kratak opis pojma „dizajn sustava“.

Dizajn sustava jest „aktivnost koju izvodi razvojni inženjer softvera i koja rezultira softverskom arhitekturom sustava“, a izvođenje dizajna softverske arhitekture sastoji se od „tehničkih, metodoloških i procesnih aspekata softverskog inženjerstva“ na način da se „izravno odnosi na potrebe produktivnog razvoja i održavanja softvera te ima velik utjecaj na konačnu kvalitetu softverskog sustava“ (Buschmann et al., 1996). McConnell navodi nekoliko obilježja unutrašnjeg dizajna sustava za koje smatra da čine izazov dizajna, jer ih je potrebno dobro odvagnuti. Ta su obilježja sljedeća: minimalna kompleksnost (tijekom rada na jednom dijelu programa može se ignorirati većina ostalih dijelova), jednostavnost održavanja (dizajniranje za održavanje, tj. tijekom pisanja kôda treba zamišljati pitanja nekoga tko će ga održavati), labavo vezivanje (engl. *loose coupling*, minimalno povezivanje umanjuje posao tijekom integracije, testiranja i održavanja), proširivost (može se poboljšati sustav bez ometanja temeljne strukture), ponovna iskoristivost, visoka raspršenost (engl. *high fan-in*, mnogo klasa koriste jednu klasu), niska do srednja suženost (engl. *low-to-medium fan-out*, jedna klasa koristi najviše 7 drugih), prenosivost, šturost (engl. *leanness*, izbjegavanje dodavanja bilo čega što je višak), stratifikacija (razmatranje sustava na jednoj razini ne zahtijeva prelazak u druge razine) i standardizirane tehnike.

Arhitektura softvera jest „okvir koji drži sve detaljnije dijelove dizajna“ (Steve McConnell, 2004), odnosno „opis podsustava i komponenti softverskog sustava i njihovih odnosa“, s tim da ti odnosi mogu biti statički i izravno upućivati na smještaj komponenti u arhitekturi, ali i dinamički, koji se odnose na temporalne veze i dinamičku interakciju komponenti (Buschmann et al., 1996). Službena dokumentacija o izradi arhitekture za

aplikacije za Android navodi da je svrha arhitekture da „definira granice među dijelovima aplikacije i odgovornosti koje bi svaki dio trebao imati“ (Guide to App Architecture, 2022). Slika 1 prikazuje kako je na konferenciji „Ruby Midwest“ autor Robert C. Martin opisao dobro definiranu granicu sustava, uz objašnjenje da je moguće izolirati sustav od mehanizma dostave podataka (bio to web ili konzola) jer sustav ne ovisi o mehanizmu dostave podataka.



Slika 1: Skica sustava s jasno definiranom granicom (Prema: Confreaks, 2012)

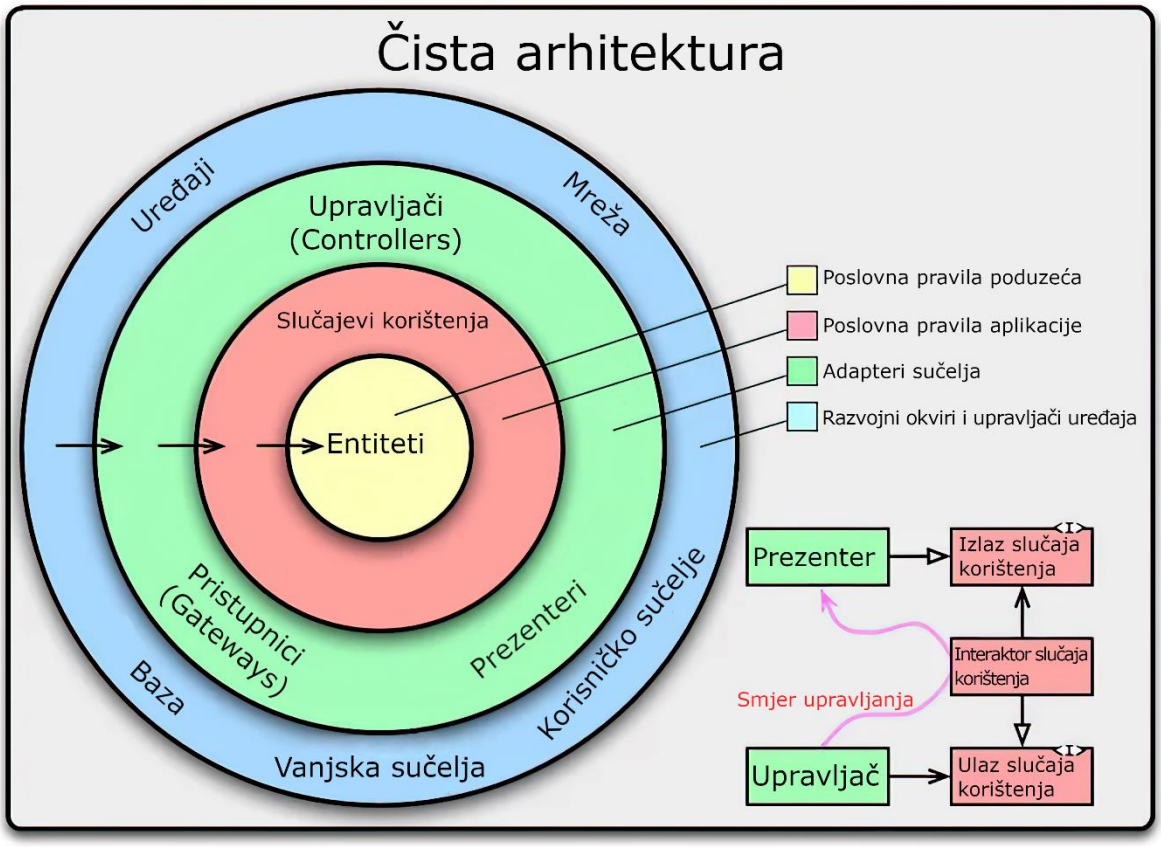
McConnell (2004) iznosi slikovitu usporedbu lanca odgovornosti pri izradi projekta te jednog hranidbenog lanca. Arhitekt koristi korisničke zahtjeve, dizajner sustava koristi arhitekturu te programer koristi dizajn, baš kao što haringe jedu vodene bube, lososi haringe i galebovi lososе. Ako je okolina zagađena, galeb će konzumirati naftu iz lososa, otrove iz haringi te nuklearni otpad iz vodenih buba. Ako su korisnički zahtjevi zagađeni, oni štete arhitekturi i izradi rješenja, ali i programerima i u konačnici rezultiraju defektnim softverom (Steve McConnell, 2004).

Spomenuti autor Robert C. Martin (2018) dobru arhitekturu definira suradnjom dobre ukomponiranosti triju osnovnih stavki (Sanchez et al., 2022):

- **SOLID** – akronim koji označava pet principa primijenjenih u procesu razvoja programskih proizvoda, a koji se koriste u čistoj arhitekturi. Oni su redom princip jedne odgovornosti (engl. *Single Responsibility Principle*), princip otvorenosti-zatvorenosti (engl. *Open-Closed Principle*), princip zamjene od Liskov (engl. *Liskov Substitution Principle*), princip odvajanja sučelja (engl. *Interface Segregation Principle*) i princip obrnute ovisnosti (engl. *Dependency Inversion Principle*). Ti su principi općeniti, ali uglavnom se referiraju na odnose među modulima jednoga rješenja (Martin et al., 2018).

- **Definicija detalja** – softverske komponente koje nisu čvrsto vezane uz poslovna pravila i mogu se mijenjati, poput baze podataka ili weba (Martin et al., 2018).
- **Komponente** – dijelovi softvera koji se moraju moći neovisno razviti i neovisno postaviti. Mogu sadržavati poslovnu logiku ili algoritme (Martin et al., 2018).

Slika 2 prikaz je tzv. „čiste“ arhitekture kako je ilustracijom definira autor knjige.



Slika 2: Čista arhitektura softverskog proizvoda prema Robertu C. Martinu
(Izvor: *Clean Coder Blog*, 2012; Martin et al., 2018)

Ilustracija je prevedena s težnjom da što vjernije prezentira originalne nazive, ali valja istaknuti da prijevodi „upravljač“, „pristupnik“ te „interaktor“ ne zvuče posve prirodno na hrvatskome jeziku, s obzirom na utjecaj engleskog jezika u ovome području.

Robert C. Martin u referenciranoj knjizi (2018) takvu „čistu“ arhitekturu definira s nekoliko karakteristika, poput toga da je neovisna o granicama razvojnih okvira, da joj je moguće izolirati i testirati samo poslovna pravila, da su njezino korisničko sučelje kao i baza podataka u potpunosti zamjenjivi i konačno da ništa iz vanjskoga svijeta ne može utjecati na poslovna pravila. Ipak, autor poručuje da programer može stvarati temelj svoje arhitekture softvera. Ono što je važno jest dobro odabrati izgled i oblik tih elemenata. Nadalje, autor izjavljuje da bi „težina implementacije svake promjene u bilo kojem softveru trebala biti

proporcionalna **samo opsegu promjene, a ne i obliku promjene**“, tj. da refaktoriranje kôda ne smije biti otežano lošim temeljem arhitekture, već eventualno svojom kompleksnošću (Martin et al., 2018).

Robert C. Martin u članku „Design Principles And Patterns“ navodi četiri simptoma „trulog dizajna“ aplikacije, koji jasno upućuju na lošu arhitekturu (Martin, 2000):

1. **Rigidnost** – očitava se u teškoj promjenjivosti softvera, čak i kada su u pitanju jednostavne promjene, zbog toga što svaka promjena uzrokuje niz pratećih promjena u ovisnim modulima kroz aplikaciju. Autor navodi da zbog toga običan propust u dizajnu može postati stroga upravljačka politika tvrtke koja zabranjuje promjene u strahu od beskonačnog rada.
2. **Lomljivost** – usko vezana uz rigidnost, opisuje težnju softvera da se pri svakoj promjeni raspadne na mnogo mjesta koja nisu konceptualno povezana s promijenjenim dijelom kôda.
3. **Nepomičnost** – rezultira ponovnim pisanjem softvera koji već postoji zbog nemogućnosti ponovnog korištenja softvera u drugim projektima ili dijelovima istog projekta.
4. **Viskoznost** – dijeli se u dvije vrste: viskoznost dizajna i viskoznost okoline. Viskoznost dizajna očituje se visokom kada je teže održati postojeći dizajn arhitekture nego izvršiti neku varku koja omogućava uspješnu implementaciju promjene unatoč narušavanja dizajna. Viskoznost okoline opisuje situacije u kojima glomaznost projekta toliko usporava razvoj da su programeri spremni žrtvovati dizajn arhitekture radi što bržeg rada na softveru.

Prepoznavanje bilo kojeg od tih simptoma upućuje na nužnost provedbe prethodno definiranog refaktoriranja.

Ovo je poglavlje bilo uvod u koncept dobre arhitekture softverskog proizvoda. To je arhitektura u kojoj su softverske komponente jasno odvojene granicama odgovornosti. Neke komponente čine ljusku aplikacije jer su vezane uz detalje, poput vrste baze podataka, vrste formata datoteka itd. One ne smiju utjecati na komponente koje su vezane uz poslovnu logiku i čine srž aplikacije. U sljedećem poglavlju domena se sužava na aplikacije pisane za operacijski sustav Android.

4.2. Obilježja arhitekture aplikacija za Android

U ovom se poglavlju nudi pregled modernih najboljih praksi postavljanja strukture direktorija i uspostave komunikacije između objekata klasa za razvoj aplikacija za Android. Poglavlje se pretežito referira Androidovu modernu službenu dokumentaciju sa službenih stranica i s GitHub repozitorija na kojemu se implementiraju najbolje prakse. Navode se i već spomenuti autori i njihovi stavovi o onome što inženjeri u Googleu iznose u službenoj dokumentaciji.

Ovo je područje silno izloženo promjenama. Uzevši u obzir znanstveni rad „Analysis of the Android Architecture“ autora Sefana Brählera iz 2010., prije trinaest godina postojao je potpuno realan problem s virtualnim strojem Dalvik koji je bio realiziran da radi s ukupno 64 MB radne memorije, a taj je problem ograničavao aplikacije da koriste samo do 40 MB radne memorije. Stoga, težilo se ograničavanju aplikacija, da zauzimaju što je manje moguće prostora te da što manje opterećuju ionako ograničene ARM procesore (Brahler, 2010). Danas tomu nije tako. S napretkom u tehnologiji došla je i promjena u vezi s tim. U *Lollipop* verziji sustava Android (peti po redu) virtualni stroj ART naslijedio je Dalvik i pritom je donio 64-bitno adresiranje, bolje prikupljanje smeća te, najvažnije, pomak kompilacije iz JIT (engl. *Just-In-Time*, kompiliranje u trenutku kada se koristi) u AOT (engl. *Ahead-Of-Time*, kompiliranje unaprijed radi uštede baterije) (Fikri et al., 2018; Jonathan, 2015).

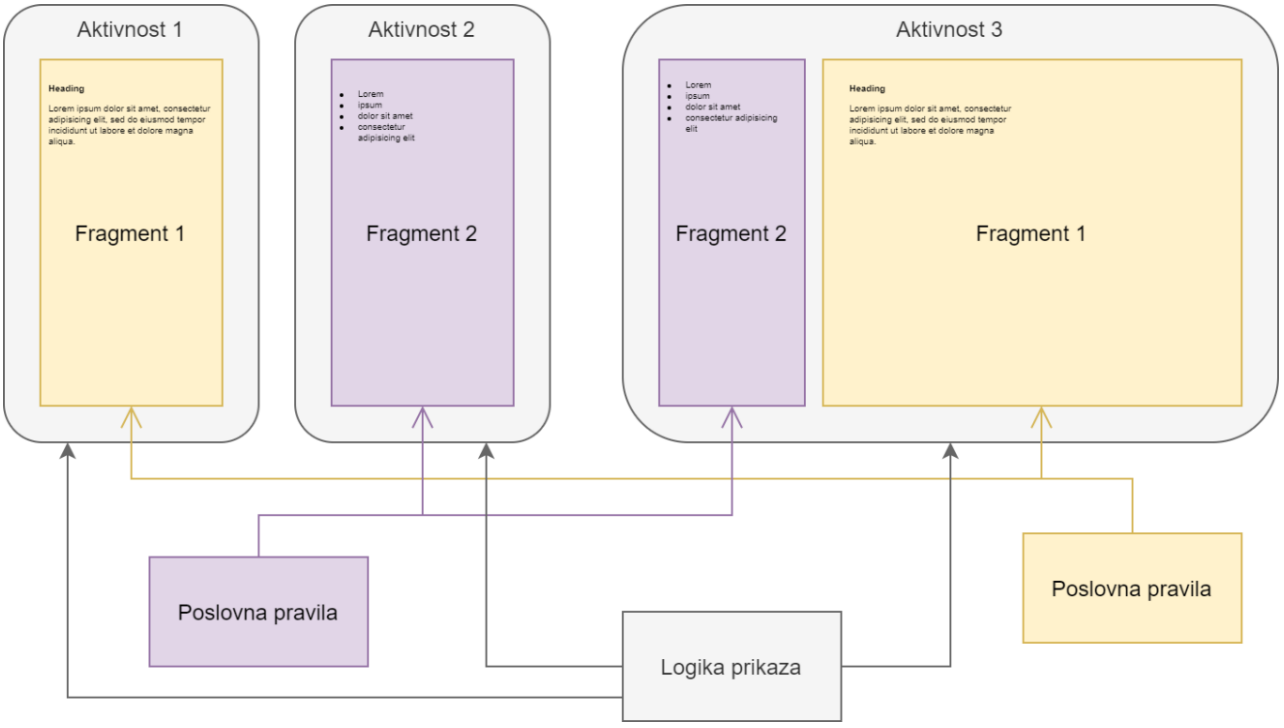
Stoga, osmišljavanje arhitekture aplikacije za Android nije kontinuirano jednak proces, a niti je održavanje Android projekta točno definiran postupak. Te dvije važne stavke razvoja mobilnih aplikacija za sustav Android definirane su mogućnostima najmodernijih, ali i najstarijih uređaja na tržištu, novim najboljim praksama u programiranju te Googleovim preporučenim praksama.

S obzirom na internetsku dokumentaciju (2022) koja čini čvrstu osnovu za bavljenje izradom modernih aplikacija za Android, takve aplikacije trebale bi sadržavati više komponenti poput sljedećih (*Guide to App Architecture*, 2022):

- aktivnosti
- fragmenata
- servisa
- pružatelja sadržaja
- primatelja poruka.

Na mrežnoj stranici „Guide to App Architecture“ (2022) jasno se naglašava da operacijski sustav može odlučiti osloboditi resurse ubijanjem procesa i stvaranjem prostora za nove. To se nadovezuje na ranije spomenute inačice virtualnih strojeva – i dalje treba imati na umu da aplikacija neće vječno raditi dok je korisnik ne ugasi te da joj sustav neće omogućiti korištenje beskonačno resursa. Preporučuje se izbjegavanje pohranjivanja važnih podataka u radnoj memoriji ili stanja u komponentama aplikacije, a naglašava se da komponente aplikacije ne bi trebale ovisiti jedna o drugoj.

Naglašen je princip razdvajanja odgovornosti (engl. *separation of concerns*). Preporučuje se koristiti što manje kôda u klasama koje nasljeđuju klase Activity i Fragment. Ističe se da te dvije klase služe kao ugovor između Androida i aplikacije, poput ljepila, te da radi korisničkog iskustva i održavanja aplikacije treba držati ovisnosti na ove dvije klase pri minimumu. Slika 3 prikazuje tri različite aktivnosti, koje ukupno sadržavaju dva fragmenta. Aktivnosti označene brojevima 1 i 2 namijenjene su mobilnim uređajima, a aktivnost 3 podržava prikaz oba fragmenta istovremeno, na tabletu. Ideja je što više odvojiti programsku logiku iz fragmenata i aktivnosti, čime postaje mnogo jednostavnije testirati aplikaciju.



Slika 3: Prikaz odnosa između programske logike i prikaza

Navodi se da je arhitektura aplikacije bazirana na klasama podatkovnog modela bolja za testiranje i robusnija je, a i da je vezanje grafičkog sučelja za trajno lokalno zapisane modele podataka idealno radi očuvanja korisničkih podataka kada Android čisti resurse ili kada aplikacija nema pristup internetu. Navodi se dalje da je potrebno imati jedinstven „izvor

točnosti“ (engl. *Single source of truth*) tko je vlasnik podataka te da ih jedino on može vraćati i mijenjati, čime se centraliziraju sve promjene određenog tipa podataka, štite se podatci i osigurava se praćenje promjena. Uz taj princip jedinstvenog izvora točnosti dodaje se i uzorak jednosmjernog protoka podataka (engl. *Unidirectional Data Flow*) u kojemu stanje teče u jednom smjeru, a događaji koji mijenjaju podatke u drugome. Navodi se da se time osigurava konzistentnost podataka, smanjuje se mogućnost pogrešaka te je olakšano otklanjanje pogrešaka. Podatci aplikacije najčešće teku iz izvora podataka prema korisničkom sučelju, a zatim korisnički događaji poput pritiska gumba teku do spomenutog jedinstvenog izvora točnosti gdje se podatci mijenjaju i gdje ih se opet prikazuje u nepromjenjivom obliku (*Guide to App Architecture*, 2022). Poslije će još biti riječi o uzorku jednosmjernog protoka podataka pri opisu sloja korisničkog sučelja.

Razvojni inženjeri održavaju dokumentaciju o poželjnoj arhitekturi aplikacija za Android, a dokumentacija je napisana u obliku knjige. Nastavak ovoga poglavlja podijeljen je u dva dijela: prvi dio opisuje modularnost u kontekstu aplikacija za Android, a drugi dio opisuje predloženu troslojnu arhitekturu.

4.2.1. Modularnost

Na najvišoj razini planiranja arhitekture promišlja se o modularnosti cijele aplikacije. Razvojni inženjeri koji rade na bibliotekama za razvoj za Android na platformi GitHub održavaju javno dostupnu aplikaciju „Now in Android“, u kojoj implementiraju najnovije najbolje prakse (*Now in Android App*, 2022). Posebna dokumentacija posvećena je modularnosti u aplikacijama za Android. Modularnost opisuju kao praksu razbijanja koncepta monolitnog kôda s jednim modulom u labavo vezane samoodržive module, čime se postiže skalabilnost, odgovornost pri razvijanju dijelova aplikacije, učahurivanje, brža izgradnja aplikacije, a omogućuje se paralelni rad i ponovna iskoristivost. U dokumentaciji (2022) se dalje navodi da je najvažnije postići što manje vezanje i što veću koheziju kôda. Nisko vezanje odnosi se na to da moduli moraju biti što nezavisniji, neupoznati s ostalim modulima i što manje pod utjecajem vanjskih promjena, a velika kohezija kôda odnosi se na to da modul mora sličiti sustavu, tj. modul mora imati jasne odgovornosti i granice definirane njegovom domenom.

Predlažu se četiri vrste modula (*Now in Android App*, 2022):

- aplikacijski modul – U kontekstu onoga kako Robert C. Martin (2018) opisuje klasu `Main`, ovo se može nazvati „najprljavijim“ modulom jer se veže na sve druge module i svaka vanjska promjena utječe na njega (*Now in Android App*, 2022).

- modul sa značajkom (engl. *feature*) – Imaju jednu jedinstvenu odgovornost u aplikaciji, a može ih se ponovno iskoristiti i u drugim aplikacijama.
- modul jezgre (engl. *core*) – Sadržava ovisnosti koje se trebaju dijeliti među ostalim modulima u aplikaciji, smiju ovisiti samo o drugim modulima iste vrste.
- modul s raznom funkcionalnošću: Mogu biti moduli za obavljanje sinkronizacije, testiranje brzine rada, jedinično testiranje itd.

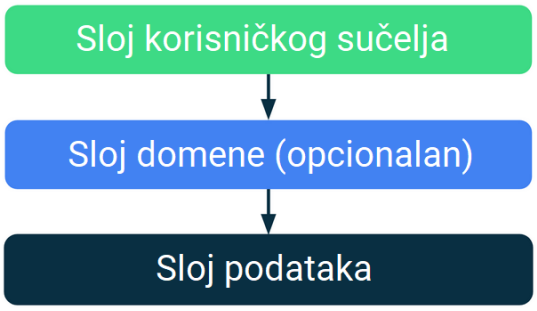
U dokumentaciji (2022) navode se i tri moguća problema s implementiranjem modularnosti: previše modula može uzrokovati preveliku kompleksnost izgradnje projekta, nedovoljno modula ne razrješava problem monolitne aplikacije, a prevelika kompleksnost modula u projektu kojemu nije planiran daljnji rast uklanja dobrobiti skalabilnosti i brzine izgradnje.

U ovom je poglavlju ukratko dan uvid u modularnost aplikacija za Android. Primjena tog vrlo važnog koncepta u praksi opisana je poslije u radu, tijekom postupka redizajna arhitekture.

4.2.2. Predložena troslojna arhitektura

U ovom poglavlju na aplikaciju se gleda iz perspektive protoka podataka. Također, ono se u potpunosti bazira na službenoj Androidovoj dokumentaciji.

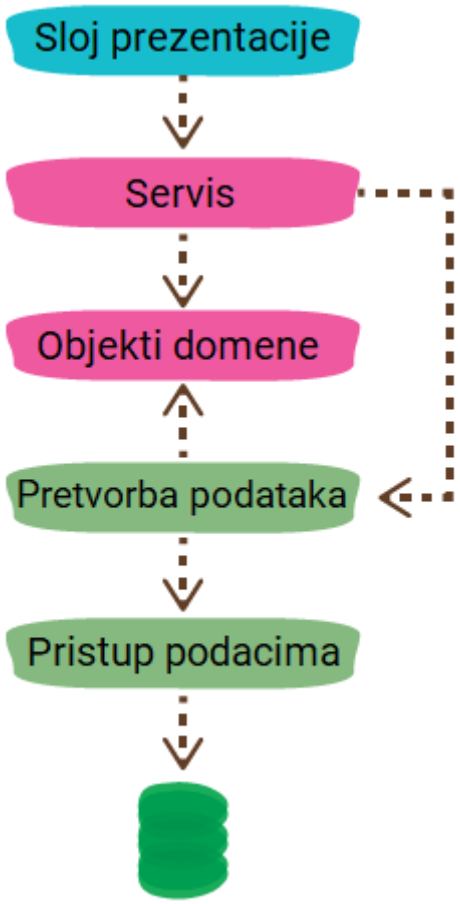
Svaka aplikacija trebala bi imati barem dva sloja, jedan za prikaz podataka na ekranu i jedan koji sadržava poslovnu logiku aplikacije i pruža podatke aplikacije. Moguće je dodati i treći sloj, sloj domene, radi pojednostavnjivanja komunikacije dvaju glavnih slojeva. Slika 4 prikazuje ta tri sloja, s oznakama ovisnosti među klasama.



Slika 4: Dijagram tipične troslojne aplikacijske arhitekture
(Izvor: *Guide to App Architecture*, 2022)

Klase iz sloja korisničkog sučelja ovise o klasama domene koje pak ovise o klasama iz podatkovnog sloja. Ovaj odnos nasljeđivanja definiran je tako vjerojatno radi jasnog određivanja granica arhitekture. To se podudara s onime što piše Robert C. Martin (2018), da

je nužno da su ovisnosti među komponentama (ili u ovome slučaju slojevima) uvijek orijentirane u jednome smjeru. Naime, to je klasičan primjer onoga što se podrazumijeva pod pojmom „troslojna arhitektura“. Riječ je o često korištenom arhitekturnom obrascu koji obuhvaća sloj za prezentaciju podataka, sloj poslovne logike te sloj za upravljanje podatcima/trajnu pohranu podataka. Martin Fowler u tekstu „Presentation Domain Data Layering“ (2015) taj način modularizacije opisuje kao jednostavnu i efektivnu mogućnost zamjene različitih implementacija modula (iako također spominje da se ta mogućnost rijetko iskorištava unatoč tome što služi kao motivacija za implementaciju ovog modela slojevitosti). Fowler još dodaje da je uobičajeno ukloniti ovisnost sloja domene na izvor podataka tako da da se između umetne klasa za mapiranje objekata (engl. *mapper*) te da se to naziva heksagonalnom arhitekturom. Slika 5 prikazuje tu opisanu arhitekturu. (Martin Fowler, 2015)



Slika 5: Alternativni prikaz troslojne arhitekture (Prema: Martin Fowler, 2015)

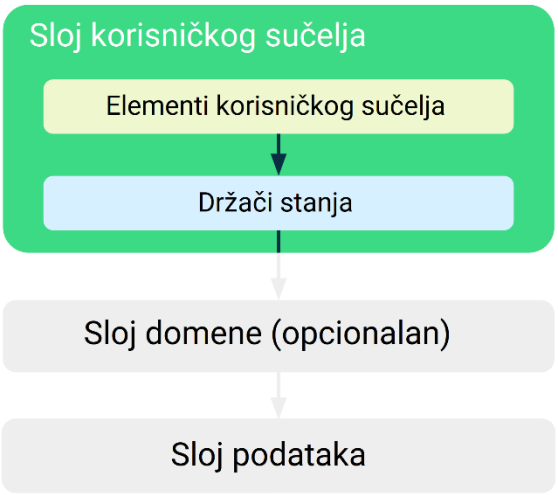
Na ilustraciji sloj prezentacije jest sloj korisničkog sučelja. Ružičasto obojeni servis i objekti domene pripadaju sloju domene, a zeleno obojeni pretvorba podataka i pristup podatcima pripadaju sloju podataka. Ideja jest upravo ono spomenuto, da servis koristi pretvorbu podataka kako bi dohvatio objekte domene koje potom koristi radi vraćanja podataka

prezentaciji, dok pretvorba podataka koristi pristup podatcima za dohvat entiteta nad kojima izvršava pretvorbu u objekte domene (koje također koristi). Valja spomenuti i mišljenje Roberta C. Martina (2018) da baza (koju je Fowler nacrtao na gornjoj ilustraciji) ne označava u arhitekturi ništa osim tehničkog detalja. Stoga bi se, s obzirom na mišljenje Roberta C. Martina, moglo diskutirati o tome bi li baza trebala biti nacrtana na gornjoj ilustraciji.

U nastavku će se definirati svaki od ovih slojeva.

4.2.2.1. Sloj korisničkog sučelja

Ovaj sloj (poznat još kao sloj prezentacije) mora ažurirati korisničko sučelje kako bi ono reflektiralo promjenu podataka nastalu korisničkom interakcijom ili vanjskim događajem poput internetskog odgovora na zahtjev aplikacije (*Guide to App Architecture*, 2022). Riječ je o razdvajanju odgovornosti prikaza podataka u zaseban modul, čime ostatak aplikacije ne ovisi o prikazu podataka.



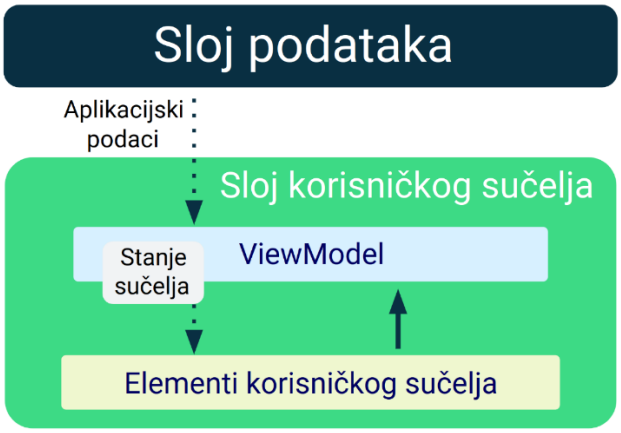
Slika 6: Uloga sloja korisničkog sučelja u arhitekturi aplikacije
(Izvor: *Guide to App Architecture*, 2022)

Na zasebnoj stranici dokumentacije ovog konkretnog sloja on se opisuje kao „vizualna reprezentacija aplikacijskog stanja kakvo je dohvaćeno sa sloja podataka“ koji je „rezultat povezivanja elemenata korisničkog sučelja i stanja korisničkog sučelja“ (*UI Layer*, 2022). Zamjena modula prezentacije može se razmatrati u kontekstu različitog prikaza ekrana malenog pametnog telefona, velikog pametnog telefona, tableta ili primjerice televizora. Ono što je na gornjoj ilustraciji nazvano „držačima stanja“ (engl. *state holders*) jesu klase koje su odgovorne za stvaranje stanja korisničkog sučelja i sadržavaju potrebnu logiku za izvršenje zadatka (*UI Layer*, 2022). Riječ je o medijatoru koji definira logiku između poslovnih pravila i

korisničkog sučelja. Ako se interakcija i njihova logika postavljaju u klase korisničkog sučelja, narušava se odgovornost korisničkog sučelja da samo prikazuje podatke (*UI Layer*, 2022).

Na to se može nadovezati i mišljenje Roberta C. Martina (2018), koji spominje uzorak skromnog objekta (engl. *Humble Object*): riječ je o uzorku koji odvaja ponašanje u dvije klase (u teoriji može i u dva modula, ali to u ovom kontekstu nije relevantno). Robert C. Martin kao primjer za korištenje ovog uzorka daje točno slučaj korisničkog sučelja. Naime, teško je testirati korisničko sučelje automatiziranjem pritisaka gumbova po ekranu i pregledom postavljenih elemenata. Međutim, Robert C. Martin ističe da je logika većine ponašanja korisničkog sučelja jednostavna za testiranje pa predlaže stvaranje objekta Prezentera (engl. *Presenter*), čija je zadaća pripremiti vrijednosti za prikaz na način da ih objekt View (koji upravlja grafičkim prikazom) jednostavno ispiše kao znakovni niz. Kao primjer daje prikaz nekog novčanog iznosa. Presenter prihvaća objekt klase valute te ga formatira u znakovni niz s ispravnim prikazom decimalnih mjesta i oznaka valute. U slučaju da iznos treba biti označen crvenom bojom, Presenter će u modelu prikaza (engl. *View model*) postaviti zastavicu boje teksta na odgovarajuću vrijednost. Autor ovakve odnose smatra arhitekturnim granicama između dijela softvera koji se može testirati i onoga koji nije namijenjen testiranju, a i piše da će se ovaj uzorak vjerojatno pojaviti kod svake arhitekturne granice (Martin et al., 2018). Zanimljivo je da se u knjizi koristi naziv „view model“, a da se u Androidu klasa koja baš osigurava ovakvu funkcionalnost naziva ViewModel. Jasno je da nazivi klasa u Androidu vuku korijene, ako ne baš iz knjige „Clean Architecture“, onda barem iz ostatka literature o ovoj temi.

Prema svemu navedenome, stvara se potreba za klasama koje drže stanje. Dokumentacija za tipičnu implementaciju ovakvih klasa predlaže objekte već spomenute klase ViewModel, čije metode definiraju logiku koja se primjenjuje na aplikacijske događaje te kao rezultat stvara ažurirano stanje. Slika 7 detaljnije prikazuje odnos elemenata korisničkog sučelja i instance klase ViewModel kao opisanog držača stanja.



Slika 7: Dijagram koji opisuje kako jednosmjerni protok podataka funkcionira u arhitekturi aplikacije (Izvor: *UI Layer*, 2022)

Ovo je zapravo primijenjeni uzorak jednosmjernog protoka podataka koji je već spomenut u poglavlju 4.2, u dijelu o modernoj arhitekturi za Android. U ovom slučaju, osim već navedenih dobrobiti takve arhitekture, uzorak implicira i da se sljedeći koraci ponavljaju za svaki događaj koji uzrokuje promjenu stanja (*UI Layer*, 2022):

1. ViewModel transformira aplikacijske podatke i time drži i izlaže stanje korisničkog sučelja.
2. Korisničko sučelje obavještava ViewModel o korisničkim događajima.
3. ViewModel upravlja korisničkim radnjama i ažurira stanje.
4. Ažurirano stanje šalje se natrag korisničkom sučelju na prikaz.

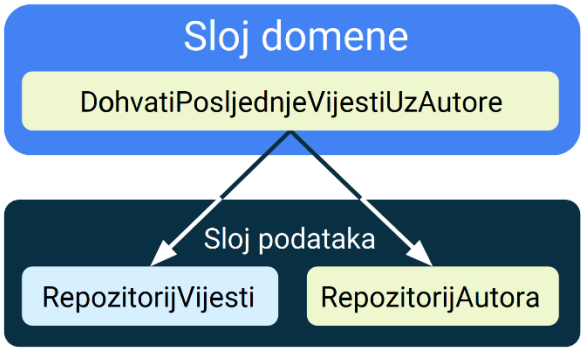
U dokumentaciji vezanoj za temu korisničkog sloja nadalje se objašnjava da postoje dvije logike u aplikaciji: poslovna logika i logika korisničkog sučelja. Što se potonje tiče, ona uključuje dohvat resursa, navigaciju ili prikaz kratke poruke korisniku. Napominje se da je moguće stvoriti jednostavne klase u službi držača stanja koje smanjuju kompleksnost korisničkog sučelja; klasa koje nasljeđuju tipove poput klase Context (*UI Layer*, 2022).

Opisano je upravljanje slojem korisničkog sučelja u kontekstu troslojne arhitekture. Idući sloj jest sloj domene.

4.2.2.2. Sloj domene

Nakon opisa sloja za prikaz podataka slijedi opis središnjega sloja – sloja domene, koji je opcionalan, a zadužen je za sadržavanje kompleksne poslovne logike aplikacije.

U dokumentaciji se naglašava da ga treba koristiti samo kada kompleksnost ili potreba za ponovnom iskoristivosti stvaraju nužnost za njime (*Guide to App Architecture*, 2022). Ako je donesena odluka za njegovim korištenjem, onda mora biti onemogućen pristup sloju podataka izravno iz sloja korisničkog sučelja. Na temelju izmišljenog primjera iz dokumentacije, Slika 8 ilustrira kontrolni tok u arhitekturi sustava pri pozivu metode koja dohvaća posljednje vijesti uz autore.



Slika 8: Graf ovisnosti u slučaju kombiniranja više repozitorija
(Izvor: *Guide to App Architecture*, 2022)

Klase u ovoj domeni zovu se slučajevi korištenja (engl. *use-cases*) ili sudionici (engl. *interactor*), a trebaju imati odgovornost nad jednom funkcionalnošću (*Guide to App Architecture*, 2022). U gornjem je primjeru ta funkcionalnost dohvaćanje posljednje vijesti uz autora. Ilustracija sadržava hrvatske nazive klasa, ali preporučena nomenklatura jest engleska (u dokumentaciji je korišteno za klasu domene `GetLatestNewsWithAuthorsUseCase`, a za klase sloja podataka `NewsRepository`, odnosno `AuthorsRepository`).

Sloj domene služi za smještanje ponavljajuće poslovne logike koja ne zahtijeva višedretvenost (ili je zahtijeva na vrlo ograničenoj razini) radi jednostavnije promjene i testiranja na mjestima gdje se logika primjenjuje.

4.2.2.3. Sloj podataka

Nakon prethodno opisana dva sloja korisničkog sučelja i domene, preostaje definirati sloj podataka. To je sloj koji sadržava vrijednost aplikacije: aplikacijske podatke i pripadnu poslovnu logiku (*Data Layer*, 2023). Slika 9 naglašava poziciju sloja podataka u arhitekturi aplikaciji.

Slika 9: Uloga sloja podataka u arhitekturi aplikacije (Izvor: *Guide to App Architecture*, 2022)

„Sloj podataka čine repozitoriji koji mogu sadržavati nula ili više izvora podataka“ (*Data Layer*, 2023). Službena dokumentacija za ovaj sloj (2023) repozitorije definira kao klase stvorene za svaki podatak kojim se upravlja u aplikaciji u svrhu povezivanja aplikacije i sustava za rad s podatcima, a koje rade sa samo jednom vrstom izvora podataka. Repozitoriji su klase koje su odgovorne za sljedeće (*Guide to App Architecture*, 2022; *Data Layer*, 2023):

- izlaganje podataka ostatku aplikacije
- centraliziranje promjena nad podatcima
- rješavanje sukoba više izvora podataka
- apstrahiranje izvora podataka od ostatka aplikacije
- sadržavanje poslovne logike.

U dokumentaciji (2023) nalaže se da bi podatci koje ovaj sloj izlaže trebali biti nepromjenjivi radi osiguranja stanja njihovih vrijednosti, ali i sigurne implementacije višedretvenosti. Što se tiče repozitorija, preporučuje se da se korištenjem uzorka ubacivanja ovisnosti (engl. *Dependency Injection*) repozitoriju pomoću konstruktora umetnu izvori podataka (*Data Layer*, 2023). Ovdje je dokumentacija kontradiktorna, jer na primjeru kôda prikazuje ubacivanje više izvora podataka u repozitorij, a to nakon definicije koja postavlja da se repozitorij treba baviti samo jednim izvorom podataka. Kako god, u dokumentaciji se

naglašava da svaki repozitorij mora biti „izvor točnosti“ vraćanjem podataka koji su „konzistentni, točni i aktualni“ (*Data Layer*, 2023). Nadalje, u dokumentaciji se preporučuje lokalna baza podataka kao „izvor točnosti“ bez pristupa internetu. Službeno preporučeni pristup korištenju lokalne baze podataka u Androidu jest biblioteka Room (*Save Data in a Local Database Using Room*, 2023), a i biblioteka Retrofit spominje se u kontekstu biblioteke za pristup udaljenim podatcima (*Data Layer*, 2023).

Važno je ne izvršavati dugotrajne operacije dohvata podataka na glavnoj dretvi jer je ona jedina dretva koja može ažurirati korisničko sučelje (*Data Layer*, 2023; *Processes and Threads Overview*, 2021). Njezinim blokiranjem znatno bi se narušilo korisničko iskustvo. Zanimljivo je da biblioteka Room zahtijeva postavljanje posebne zastavice prilikom izgradnje objekta za pristup bazi podataka ako se želi izvršavati dohvat podataka iz glavne dretve. Dakle, natjeralo se programere da koriste višedretvenost za pristup bazi. Moguće je da je takav oštar pristup zabrani korištenja dretve za korisničko sučelje nastupio kako bi se spriječile loše performanse modernih aplikacija za Android. U suprotnom, da većinu programera nije briga ili da nisu svjesni ovog detalja, loše bi performanse mogle navesti korisnike na generalizirana mišljenja da su aplikacije za Android spore, da zapinju itd. Zanimljivo je kako ta ideja vodi do zaključka da sitan propust programera duboko u kôdu može narušiti popularnost cijele platforme, a još je zanimljivije kako se na programere može utjecati tako da ih se natjera da pročitaju dokumentaciju i razumiju što se zapravo zbiva u unutrašnjosti sustava.

U dokumentaciji za sloj podataka (2023) predlaže se razdvajanje klasa tako da bez obzira na to što se dobiva iz izvora podataka, postoji prethodno definiran model koji sadržava samo ono što je aplikaciji potrebno te zbog kojega se štede podatci, prilagođavaju tipovi i po kojemu se potom mogu paralelno razvijati i mrežni servisi i korisničko sučelje.

Definiraju se tri vrste operacija nad kojima sloj podataka može djelovati (*Data Layer*, 2023):

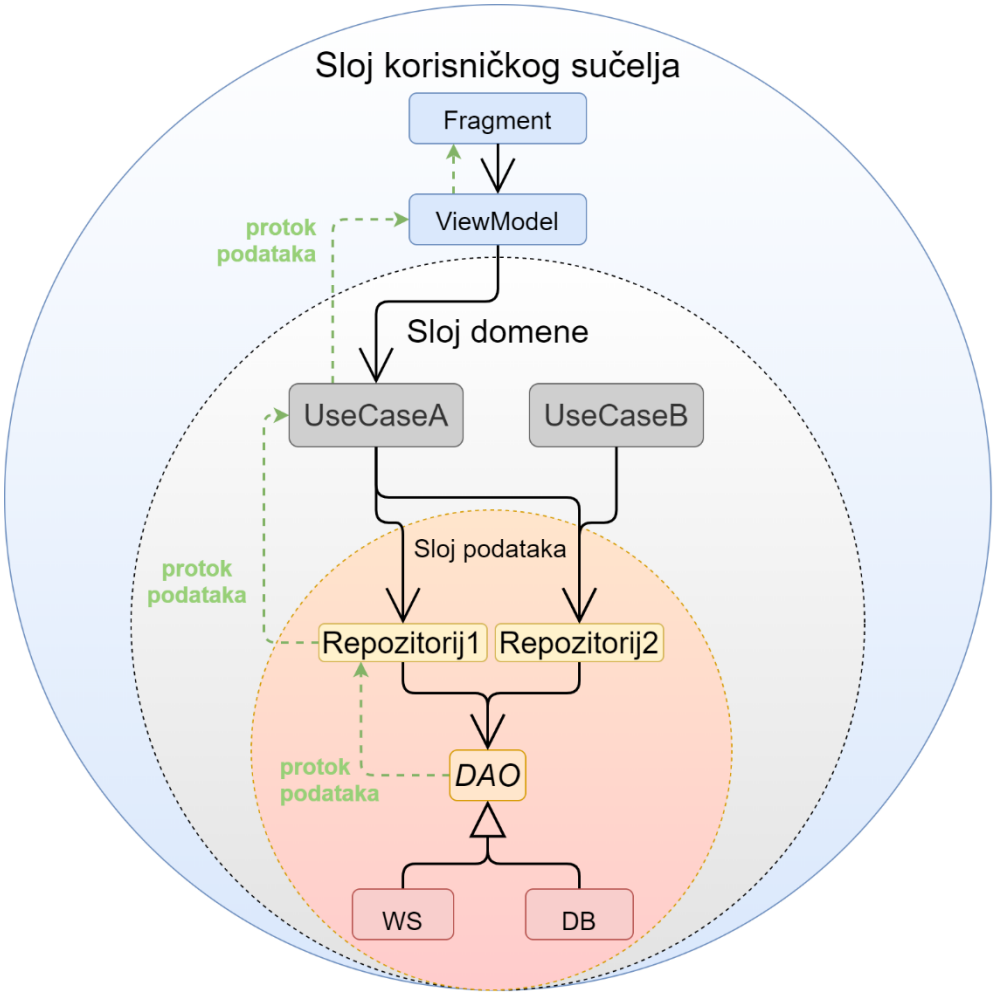
- operacije usmjerene na korisničko sučelje – usko vezane uz vidljiv prikaz, a prekidaju se kada korisnik promijeni prikaz
- operacije usmjerene na aplikaciju – trebaju se izvršavati dok aplikacija radi neovisno o trenutačnom prikazu kako bi osigurale podatke nužne za rad aplikacije
- operacije usmjerene na poslovni proces – ne smiju se prekinuti čak ni pri gašenju aplikacije jer bi mogle rezultirati gubitkom podataka.

Ovdje je bio prikazan treći, posljednji sloj preporučene arhitekture za aplikacije za Android, kakvu je definira službena dokumentacija.

4.3. Zaključak o dobroj arhitekturi aplikacije za Android

Dok su u prethodnom poglavlju definirana sva tri sloja preporučene arhitekture te vodilje kako implementirati aplikaciju da vjerno prati tu specifikaciju, u ovome poglavlju donosi se zaključak vezan za praćenje najbolje prakse u izradi moderne arhitekture aplikacije za Android.

S obzirom na sve izneseno, kao i s obzirom na ilustraciju Roberta C. Martina prikazanu u poglavlju 4.1, autor ovoga rada predlaže izgled dobre arhitekture aplikacije za Android po uzoru na navedenu ilustraciju. Slika 2 umnogome je utjecala na izradu sljedeće ilustracije. Slika 10 autorova je ilustracija primjera u kojem se koriste sve preporučene komponente troslojnog dizajna, s uključenim slojem domene.



Slika 10: Ilustracija preporučene arhitekture aplikacije za sustav Android (izradio autor, prema: *Guide to App Architecture*, 2022; Martin et al., 2018)

Zelenom isprekidanom crtom označen je jednosmjerni tok podataka koji je već više puta do sada naglašen. Fragment (naziv klase osnovnog ponovno iskoristivog elementa grafičkog sučelja u razvoju aplikacija za Android) prikazuje podatke na način da ViewModel poziva odgovarajuću komponentu u sloju domene (nazvanu „*UseCaseA*“), zatim ta komponenta u potrazi za relevantnim podatcima kontaktira oba dostupna repozitorija, a repozitoriji komuniciraju s pristupnim objektom za podatke (DAO, engl. *Data Access Object*) čije sučelje, pak, implementiraju komponente za komunikaciju s mrežnim servisom, odnosno s lokalnom bazom. Primjer je prilagođen situaciji u kojoj je moguće podatke dohvaćati iz dvaju različitih „izvora istine“, recimo s obzirom na to je li uređaj na mreži ili nije. Dizajn ove implementacije mogao bi biti poznati uzorak dizajna Factory Method, koji bi u tom slučaju vraćao objekt s obzirom na to kako se pri početku izvođenja programa odluči.

U sljedećem poglavlju daje se pregled nekih poteškoća u razvoju aplikacija za Android.

4.4. Poteškoće razvoja arhitekture za Android

U prethodnom je poglavlju prikazan jedan primjer dobre implementacije arhitekture za aplikacije razvijene za Android. Međutim, planiranje arhitekture za Android može biti i otežano nekim čimbenicima, o kojima je riječ u ovome poglavlju. Neki su nedostatci navedeni na temelju literature, a neke autor navodi iz vlastitog iskustva izrade aplikacija za Android.

Jedan od velikih nedostataka pri razvoju za Android jest nemogućnost slanja objekata po referenci između elemenata prikaza. Moderna navigacija u mobilnim aplikacijama ovisi o brzoj izmjeni ekrana, po mogućnosti s animacijama, pri čemu se nerijetko treba zadržavati sadržaj. Android se prilagodio ovome zahtjevu uvodeći navigacijske grafove koji definiraju moguće prelaske s jednog fragmenta unutar drugoga u istoj aktivnosti. Primjer može biti pregled artikala, pri čemu dugačak pritisak na jedan artikl otvara poseban ekran za prikaz tog artikla. Osnovno poznavanje objektno orijentiranog pristupa obuhvaća način razmišljanja u kojemu referenca na objekt artikla jednostavno prelazi s jednog prikaza na drugi. Nažalost, u vrijeme pisanja ovog rada ne postoji način za prebacivanje reference objekta između aktivnosti ili fragmenata. Moguće je ponovno izvršiti dohvat iz sloja podataka, izvršiti serijalizaciju objekta, implementirati statičko svojstvo ili objekt uzorka Singleton. Međutim, taj nedostatak svakako je ozbiljno kršenje objektno orijentirane paradigme programiranja. Taj problem ujedno stvara poteškoće u razvoju arhitekture jer se arhitekt mora prilagoditi specifičnosti platforme umjesto specifičnostima jezika. Autori Malavolta i sur. (2018) identificirali su da je visoka razina dupliciranja kôda u projektima za sustav Android povezana s programskim idiomima razvoja

za Android te da na tu razinu utječe orijentiranost aktivnostima i objektima namjere (klase Intent[3]).

Još jedan potencijalni problem jest životni ciklus aktivnosti. Kao primjer uzroka narušavanja životnog ciklusa aktivnosti može se spomenuti rotiranje uređaja. Pri rotiranju uređaja radi promjene orijentacije prikaza zaslona, trenutačno vidljiva aktivnost uništava se te se potom ponovno stvara, čime se ponovno pokreće njezina *onCreate* metoda (*Handle Configuration Changes*, 2022). U službenoj dokumentaciji (2022) preporučuje se prijenos objekta sa stanjem u novu instancu aktivnosti. Postavlja se pitanje zašto bi se cijela aktivnost uništila samo zbog promjene konfiguracije. Jedan od mogućih odgovora jest zato što programer ionako mora računati s time da će se aktivnost uništiti, primjerice, prilikom već spomenutog slučaja oslobađanja resursa sustava. Međutim, to da stanje temeljnih objekata klasa aplikacije može biti u potpunosti narušeno samo zbog toga što se mijenja izgled grafičkog korisničkog sučelja diskutabilno je u kontekstu dobre arhitekture sustava. Međutim, moderni mehanizmi poput već opisanog ViewModela uspješno ublažuju efekte ovakvih odluka razvojnog tima platforme.

Ovisnosti biblioteka za Android (onih treće strane ili službenih Googleovih) mijenjaju se velikom brzinom. Ponekad te „nevidljive“ promjene postaju toliko snažne da ovisnosti međusobnom nekompatibilnošću onemoguće rad aplikacije. U kontekstu ove rasprave, opasnost od naglih promjena važnih biblioteka stavlja velik pritisak na osmišljavanje arhitekture, koji može negativno utjecati na neke arhitekturne odluke. Mnoštvo biblioteka problematično je i zbog mnoštva ovisnosti, kojima može biti kompleksno upravljati kod modularnih rješenja. Problematika tog slučaja jest u tome što treba odrediti koji modul i zašto treba i smije imati ovisnosti na koji drugi modul ili biblioteku. Nadalje, postavlja se i pitanje gdje inicijalizirati sve objekte koji predstavljaju ovisnosti. Na sreću, na to pitanje odgovara Robert C. Martin (2018) navodeći da je glavni objekt (engl. *Main*), koji je ujedno i ulazna točka izvršavanja aplikacije kao računalnog programa, „najprljaviji“ objekt u aplikaciji jer je upravo on zadužen za povezivanje ostatka aplikacije s ovisnostima. U kontekstu Androida, taj bi objekt mogla biti glavna aktivnost koja nerijetko nosi naziv *MainActivity* (tako je imenuje razvojno okruženje Android Studio pri odabiru predloška projekta s aktivnošću). Naravno, ako se aktivnost rekreira, moguće je da se rekreiraju i svi objekti ovisnosti što nipošto nije optimalno.

[3] objekti vrlo važne klase za ostvarivanje suradnje sa sustavom Android

Navedene poteškoće nisu katastrofalne, ali valja ih istaknuti kao odgovor na detaljan opis arhitekture aplikacija za Android.

Slijedi drugi dio rada, u kojem se daje pregled i analiza projekta jedne objavljene aplikacije.

5. Analiza postojeće aplikacije

Dok je tema prethodnoga poglavlja bila najbolja praksa uspostave i održavanja arhitekture mobilne aplikacije, u ovome poglavlju prikazuje se jedna mobilna aplikacija iz stvarnoga svijeta, koja ima tisuće korisnika, te se analizira njezina arhitektura.

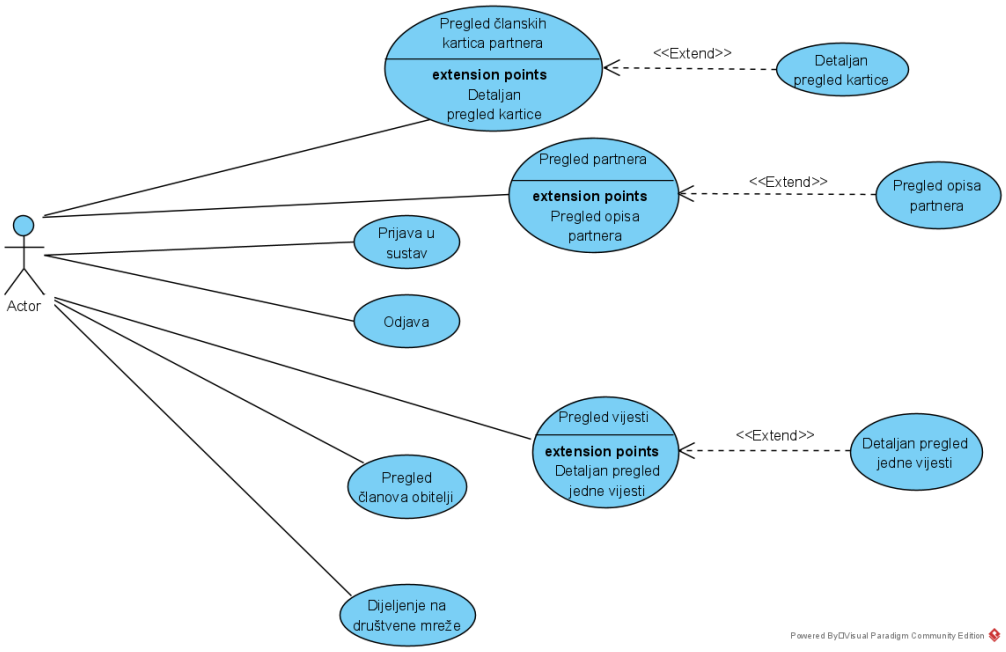
5.1. Pregled funkcionalnosti

Aplikaciju o kojoj je u radu riječ na tržište je postavila organizacija s nekoliko tisuća članova. Aplikacija omogućava članovima organizacije jednostavan pregled ostvarenih benefita kod partnerskih tvrtki. Članovi mogu pregledavati vijesti vezane uz organizaciju i pregledati informacije o povezanim članovima. Aplikacija je prestala biti razvijana 16. ožujka 2021, a najveći broj promjena imala je tijekom 2019. godine. Budući da je dio kôda bio zastario, prije analize stari *android* paketi nativnih[4] biblioteka zamijenjeni su modernim *androidx* paketima nativnih biblioteka. Između ostaloga, kao izvor biblioteka uklonjen je zastarjeli *jcenter* repozitorij, ažurirane su verzije gotovo svih ovisnosti te je aplikacija prilagođena korištenju moderne klase AppCompatActivity umjesto starije klase Activity.

Aplikacija nudi nekoliko osnovnih funkcionalnost koje su poprilično učestale na tržištu mobilnih aplikacija.

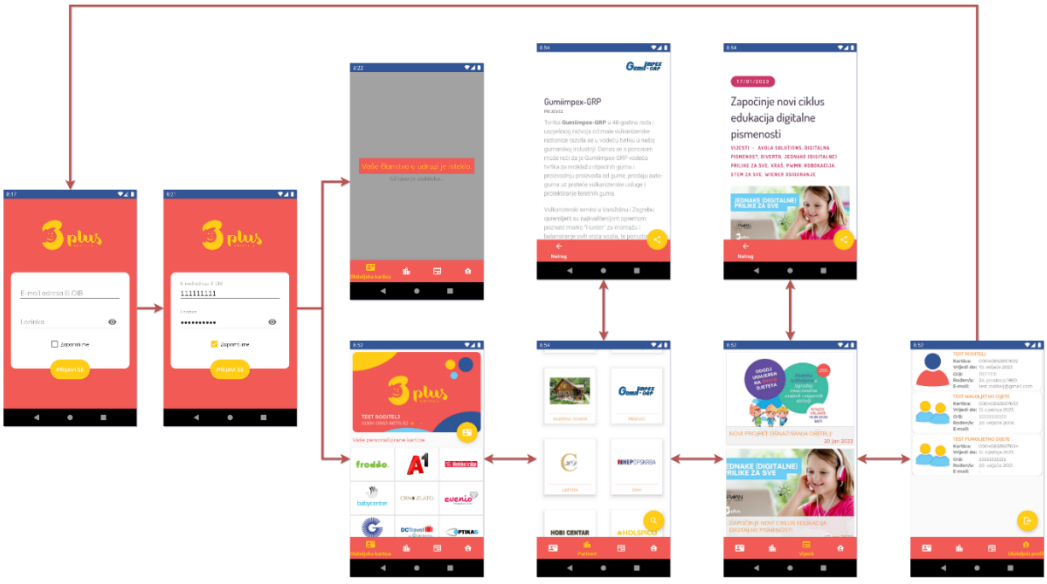
[4] engl. *native*, u ovom kontekstu prirođenih platformi za koju se razvija, tj. Android

Slika 11 prikazuje dijagram slučajeva korištenja aplikacije (engl. *use-case*).



Slika 11: Dijagram slučajeva korištenja postojeće aplikacije

Slika 12 prikazuje dijagram navigacije kroz aplikaciju.



Slika 12: Navigacijski graf aplikacije

Crvene crte prikazuju navigaciju korisnika kroz aplikaciju, a strelice na krajevima crvenih crta prikazuju smjer navigacije. Dvosmjerna navigacija ostvarena je kod pregleda detalja o odabranoj partnerskoj tvrtki i detalja o odabranoj vijesti.

Pri uspješnom pokretanju, aplikacija prikaže ekran za prijavu. Nakon prijave, aplikacija korisnika vodi do glavnoga ekrana. Ako je korisničko članstvo u organizaciji isteklo, ekran je zamračen i prikazana je poruka: „Vaše članstvo u udruzi je isteklo“. U tom je slučaju korisniku omogućeno da pregledava vijesti i informacije o povezanim članovima. Ako je članstvo aktivno, korisnik dobiva mogućnost pregleda članske iskaznice, primjera članskih iskaznica za tvrtke partnere te informacije o tvrtkama partnerima.

Alat Android Studio omogućava pregled broja linija, korištenih datoteka, ostvarenih ovisnosti itd. Projekt ove aplikacije sastoji se od ukupno 248 datoteka, od čega 232 datoteke čine bazu izvršnog kôda. Programski kôd čini 8345 linija kôda raspodijeljenih u ukupno 24 paketa[5], od čega osim korijenskog postoji devet glavnih. Tablica 1 prikazuje popis paketa u ovome projektu. U prvome stupcu podebljano su napisani nazivi glavnih paketa. U kurzivu su napisani nazivi paketa koji pripadaju jednome od glavnih paketa. U drugome stupcu piše broj klasa, odnosno sučelja u paketu. Ovdje su i klase i sučelja okupljeni pod nazivom „klasa“.

Tablica 1: Popis paketa klasa u analiziranoj aplikaciji

Paket	Broj klasa
korijenski paket	2
adapters	6
adapters.diffutil_callbacks	2
adapters.viewholders	3
async	10
contracts	6
data	3
***data**.api*	1
***data**.api.callbacks*	1
***data**.api.interfaces*	1
***data**.models*	20

[5] engl. *package*, pojam vezan za smještaj klasa u jeziku Java

***data**.repositories*	1
***data**.repositories.interfaces*	6
***data**.repositories.local*	6
***data**.repositories.remote*	6
listeners	5
presenters	6
utils	22
***utils**.file_update*	5
***utils**.model_utils*	2
***utils**.sharing*	1
view_models	5
views	0
***views**.activities*	5
***views**.fragments*	8

Iz gornje tablice može se vidjeti da paket *utils* sadržava najviše klasa, čak 22, po veličini je odmah iza njega paket s podatkovnim modelima, a slijedi paket *async*.

Aplikacija ima četiri konkretne implementacije aktivnosti, koje su abecednim redom ove:

1. **ContentActivity** – prikazuje glavni sadržaj aplikacije nakon prijave, sadržava fragmente
2. **MainActivity** – započinje aplikaciju na način da korisnika preusmjeri na prijavu
3. **SignInActivity** – omogućuje prijavu u sustav
4. **WebViewActivity** – omogućuje pregled mrežnih sadržaja te dijeljenje prikazanog sadržaja pomoću društvenih mreža.

Projekt koristi mnoštvo ovisnosti. Ovisnosti su stvorene na 236 vanjske klase, od čega 167 klasa pripada aplikacijskom programskom sučelju sustava Android, a 69 klasa pripada vanjskim bibliotekama. Tih 69 vanjskih ovisnosti usmjerene su prema bibliotekama treće strane (dakle, ne one od Googlea ili iz Androida) koje se koriste u aplikaciji i to su:

- Butterknife za povezivanje elemenata sučelja s objektima koji ih predstavljaju
- Lombok za automatsko generiranje metoda poput *gettera* i *settera*
- OkHttp3, Retrofit za rad s mrežnim servisima

- Glide za jednostavno preuzimanje i prikaz fotografija s mrežnih izvora
- Expandable RecyclerView za prikaz podataka u listi koja se može rastvarati
- Biblioteka *facebook-share* za dijeljenje sadržaja na Facebook.

Slijedi opis prepoznatih arhitekturnih uzoraka u ovoj aplikaciji koji se, u skladu s prethodno obrađenom literaturom, smatraju dobrima.

5.2. Prepoznati uzorci u arhitekturi

Prethodno je poglavlje bilo uvod u odabranu aplikaciju za Android koja će se koristiti za praktični arhitekturni redizajn. U ovome se poglavlju ulazi nešto dublje u taj praktični dio rada, pri čemu se ističu dobri arhitekturni uzorci prepoznati u ovoj aplikaciji.

Pri samom otvaranju aplikacije, raspored i nazivlje aplikacije odaje dojam ozbiljnosti. Naziru se softverske komponente pa se naslućuju i granice arhitekture. Valja razlikovati arhitekturne uzorke i uzorke dizajna. U ovome poglavlju spomenut će se obje vrste, s obzirom na to da rad obuhvaća i arhitekturu i dizajn pod istom domenom strukture projekta.

5.2.1.1. Arhitekturni uzorci

Slijedi opis dobro korištenih arhitekturnih uzoraka aplikacije. Arhitektura aplikacije ispravno uključuje sloj prezentacije. Prikaz čine odgovarajuće aktivnosti i fragmenti, pripadni Presenteri i pozadinski ViewModeli, a sve su te komponente uredno posložene u odgovarajuće pakete:

- **aktivnosti i fragmenti** – paket *views*
- **Presenteri** – paket *presenters*
- **ViewModeli** – paket *view_models*

Tablica 2 sadržava veze između klasa aktivnosti, klasa Presentera i klasa ViewModela.

Tablica 2: Povezanost elemenata sloja korisničkog sučelja u aplikaciji

Prikaz	Presenter	ViewModel
FamilyCardFragment	FamilyCardFragmentPresenter	FamilyCardViewModel
FamilyProfileFragment	FamilyProfileFragmentPresenter	FamilyProfileViewModel
NewsFragment	NewsFragmentPresenter	NewsViewModel
PartnersFragment	PartnersFragmentPresenter	PartnerViewModel
SignInActivity	SignInPresenter	SignInViewModel
SettingsFragment	SettingsFragmentPresenter	

Gornja tablica izrađena je na temelju uvida u reference u kôdu. Od aktivnosti, jedino klasa `SignInActivity` izravno referencira svoj pridruženi Presenter. Od ostalih prethodno navedenih aktivnosti, još samo `ContentActivity` pruža prikaz, ali preko objekata fragmenata. Ne postoji veza u komunikaciji između sloja korisničkog sučelja i sloja podataka.

Još jedna dobra arhitekturna odluka bilo je uvođenje mnoštva sučelja za rad s klasama repozitorija. Naime, tih je sučelja šest, a ona su redom:

- `FamilyCardRepositoryInterface`
- `FamilyProfileRepositoryInterface`
- `NewsRepositoryInterface`
- `PartnerRepositoryInterface`
- `PartnerTypeRepositoryInterface`
- `SignInRepositoryInterface.`

Ovo je dobra odluka jer ispunjava princip obrata ovisnosti (engl. *Dependency Inversion Principle*). Taj je princip Robert C. Martin (2018) definirao na način da je zabranjeno ovisiti o konkretnim implementacijama, osim ako su u pitanju ovisnosti vezane za operacijski sustav ili platformu jer su ondje promjene rijetke. Promjene sučelja repozitorija ne moraju biti rijetke te je stoga pohvalno da je ovakav pristup primijenjen u aplikaciji.

Sljedeće poglavlje obuhvaća detaljniji osvrt na prepoznate korištene uzorke dizajna unutar aplikacije.

5.2.1.2. Uzorci dizajna

Spomenuta fleksibilnost u kontekstu *online* i *offline* repozitorija uspostavljena je korištenjem mehanizma sličnog uzorku dizajna Factory Method (Gamma et al., 1994). Međutim, objekti se ne stvaraju pozivom metode, već se samo dohvaća ispravna instanca jer su svi repozitoriji implementirani po uzorku dizajna Singleton (Gamma et al., 1994). Programski kôd 1 prikazuje jedan primjer tog uzorka u kôdu. Riječ je o odabiru repozitorija s obzirom na stanje mrežne povezanosti.

```
1  public static SignInRepositoryInterface getSignInRepository() {
2      return NetworkUtils.isConnected() ?
3                          SignInRepository.getInstance() :
4                          OfflineSignInRepository.getInstance();
5  }
```

Programski kôd 1: Odabir implementacije s obzirom na mrežnu povezanost

Uzorak dizajna Singleton vrlo je čest u ovoj aplikaciji. Tablica 3 sadržava popis klasa koje su implementirane po ovom uzorku Singleton. Klasa AppDatabase nije navedena u tablici jer nije korištena u kôdu.

Tablica 3: Korištenja uzorka dizajna Singleton u aplikaciji

Singleton klasa	Pretpostavljeni razlog korištenja uzorka
AppUserManager	Jednostavnost privremene pohrane korisničkih podataka, praćenja prijave, čišćenja podataka o prijavi jednoga ili svih korisnika, provjere valjanosti privremeno pohranjenih korisničkih podataka itd.
Obitelji3PlusApplicationMemoryCache	Jednostavan dohvat korisničkog imena i identifikacijske oznake trenutačno aktivnog korisnika.
SharedPrefManager	Jednostavnost korištenja trajne memorije.
APIClient	Pristup istoj referenci na izgrađeni objekt biblioteke Retrofit. Njegova je izrada skupa pa je ovako osigurano da se objekt instancira samo pri prvome zahtjevu za njime.
SignInRepository	Jedinstven pristup odgovoru za pokušaj prijave. Učahuren pristup izgrađenom objektu za pristup mrežnim servisima koji obavlja poziv.
FamilyCardRepository	Jedinstven pristup dijelovima kartica tvrtki partnera te članskoj iskaznici.
OfflineFamilyCardRepository	Jedinstven pristup obiteljskim karticama bez pristupa mreži, korištenjem lokalne datoteke.
OfflineFamilyProfileRepository	Jedinstven pristup svim članovima obitelji pomoću broja kartice bez pristupa mreži, korištenjem lokalne datoteke.

Velik broj klasa izrađenih po uzorku Singleton ponekad je neizbježan, ali svakako treba imati na umu da takve klase čvrsto vežu kôd i da nerijetko otežavaju održavanje, pogotovo kada je potrebno izvesti promjene na njima.

5.3. Kritike korištenog dizajna

U prethodnom poglavlju istaknute su kvalitete arhitekturnog dizajna proučavane aplikacije, a u ovom se poglavlju navode uočeni nedostatci. Poglavlje je podijeljeno u dva dijela:

1. Nedostaci uočeni ručnim pregledom
2. Nedostaci identificirani automatiziranim pregledom
    a. Nedostaci koje je uočio alat SonarLint
    b. Nedostaci koji su uočeni statičkom analizom projekta pomoću platforme SonarQube
    c. Nedostaci uočeni dinamičkom analizom aplikacije tijekom rada pomoću platforme Firebase

Cilj je ovih triju navedenih pristupa prikazati moć modernih alata koji se koriste u praksi u procesu analize i refaktoriranja kôda.

5.3.1. Uočeni nedostaci projekta

U ovom se poglavlju prikazuju nelogičnosti u projektnim datotekama i izvornome kôdu aplikacije odabrane za analizu. Nelogičnosti su identificirane pomnim pregledom kôda i klasa.

Kad je riječ o strukturi projekta, nekoliko sučelja (`FragmentClickListener`, `NewsDownloadListener`, `PartnerCardsDownloadListener`, `PartnersDownloadListener`) nije u paketu *listeners*, već su po raznim paketima gdje se koriste. Moguće je diskutirati o tome je li bolje raspodijeliti klase po komponentama ili po svrsi. Ako postoji paket *listeners*, onda se očekuje da svi slušači događaja budu u njemu. Mišljenje autora ovoga rada jest da paket *listeners* nije opravdan, već da sučelja trebaju biti ondje gdje se koriste te da služe kao apstrakcije za konkretne implementacije istoga paketa. Ako bude potrebno dodavati nove funkcionalnosti unutar modula, neka se to izvršava u paketima gdje funkcionalnosti logički pripadaju. Trenutačno, u projektu se ne poštuje nijedna od dviju ponuđenih mogućnosti. Programski kôd 2 prikazuje tu implementaciju.

```
1    private BottomNavigationView.OnNavigationItemSelectedListener
             mOnNavigationItemSelectedListener
2            = new BottomNavigationView.OnNavigationItemSelectedListener() {
3
4        @Override
5        public boolean onNavigationItemSelected(@NonNull MenuItem item) {
6            switch (item.getItemId()) {
7                case R.id.navigation_family_card:
8                    openViewPagerPage(0);
9                    return true;
10               case R.id.navigation_partners:
11                   openViewPagerPage(1);
12                   return true;
13               case R.id.navigation_news:
14                   openViewPagerPage(2);
15                   return true;
16               case R.id.navigation_family_profile:
17                   openViewPagerPage(3);
18                   return true;
19               //case R.id.navigation_settings:
20               //    openViewPagerPage(4);
21               //    return true;
22           }
23           return false;
24       }
25   };
```

Programski kôd 2: Odabir navigacijske stavke

Jedan od najočitijih nedostataka dizajna arhitekture jest način na koji aplikacija upravlja korisničkim odabirom stavke navigacije. Konkretno, riječ je o odabiru prikaza kartica, partnera, vijesti ili povezanih članova.

Gornja implementacija ima mnogo nedostataka. Za početak, komentirani dio kôda na linijama numeriranima od 19 do 21 potpuno je redundantan i čini kôd teže čitljivim. Programer koji je izvršio komentiranje u svojoj je poruci uz *commit*[6] napisao: „Privremeno onemogućen (skriven) fragment za postavke. Plan je funkcionalnost *logouta* prebaciti na fragment za obiteljski profil.“ Dakle, umjesto izbacivanja kôda, programer je vjerojatno želio „za svaki slučaj“ ostaviti kôd ako ga bude potrebno ponovno aktivirati. Međutim, komentar je postavljen 2019. godine, a u međuvremenu je funkcionalnost odjave uistinu i bila prebačena na obiteljski profil.

Nadalje, alat Android Studio označava da je korištenje sučelja `BottomNavigationView.OnNavigationItemSelectedListener` zastarjelo te da bi trebalo koristiti moderniju alternativu `NavigationBarView.OnItemSelectedListener`. I zastarjelo

[6] Osnovna jedinica verzioniranja stanja kôda. Commit između ostaloga čini i komentar autora o promjeni.

sučelje radi, ali svakako ga u sklopu održavanja projekta treba ukloniti iz kôda kako bi aplikacija bila spremna za modernije uređaje te u tijeku s modernijim programskim sučeljima.

Najveći nedostatak s gornjim kôdom jest arhitekturni nedostatak. Ono što je u kôdu problematično jest razdijeljenost logike kôda. To je najjednostavnije objasniti situacijom: organizacija želi dodati novu značajku u program. Taj izmišljeni zadatak jest dodati novu karticu koja će prikazivati informacije o uplaćenim članarinama. Tu valja ponoviti navod Roberta C. Martina u knjizi „Clean Architecture“: „težina implementacije svake promjene u bilo kojem softveru treba biti proporcionalna samo opsegu promjene, a ne i obliku promjene“ (Martin et al., 2018). U aplikaciji koja je bazirana na navigaciji karticama dodavanje još jedne kartice promjena je relativno malenog *opsega*. Međutim, u gornjem kôdu to je promjena koja zahtijeva mijenjanje mnogo toga! Programski kôd 3 prikazuje dio kôda u kojemu se fragmenti postavljaju u kolekciju iz koje im se potom pristupa spomenutim indeksima.

```
1    private void setupViewPager() {
2        this.adapter = new MainContentPagerAdapter(getSupportFragmentManager());
3        this.adapter.addFragment(FamilyCardFragment.getInstance());
4        this.adapter.addFragment(PartnersFragment.getInstance());
5        this.adapter.addFragment(NewsFragment.getInstance());
6        this.adapter.addFragment(FamilyProfileFragment.getInstance());
7        //this.adapter.addFragment(SettingsFragment.getInstance());
8        vpMainContent.setAdapter(adapter);
9
10       vpMainContent.addOnPageChangeListener(onPageChangeListener);
11
12       //TODO Improve ViewPager's implementation
13       vpMainContent.setOffscreenPageLimit(3);
14
15       vpMainContent.post(new Runnable() {
16           @Override
17           public void run() {
18               currentItem = currentItem == -1 ? 0 : currentItem;
19               storeCurrentPage();
20               vpMainContent.setCurrentItem(currentItem);
21           }
22       });
23
24   }
```

Programski kôd 3: Metoda za postavljanje fragmenata glavnoga prikaza

Dakle, prvo se stvori adapter za prikaz sadržaja. Zatim se fragmenti dodaju jedan po jedan u taj adapter. Pritom se fragmenti instanciraju iza metode *getInstance* (o tome više riječi u poglavlju 6.1). Prethodno spomenuta promjena uključivala bi dodavanje novog fragmenta u adapter, a ukoliko taj novi fragment treba biti u sredini navigacijskog izbornika, utoliko bi trebalo ručno pamtiti indekse fragmenata i mijenjati ih u selekciji tipa *switch*.

Najveći problem s prikazom ovoga glavnog izbornika jest taj da se kartice koje povezuju fragmente definiraju potpuno odvojeno od samih fragmenata. Trenutačno se kartice postavljaju pomoću XML-a (engl. *Extensible Markup Language*) u datoteci *navigation.xml*.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <menu xmlns:android="http://schemas.android.com/apk/res/android">
3      <item
4          android:id="@+id/navigation_family_card"
5          android:icon="@drawable/ic_account_card_details"
6          android:title="@string/menu_family_card_item" />
7      <item
8          android:id="@+id/navigation_partners"
9          android:icon="@drawable/ic_city"
10         android:title="@string/menu_partners_item" />
11     <item
12         android:id="@+id/navigation_news"
13         android:icon="@drawable/ic_newspaper"
14         android:title="@string/menu_news_item" />
15     <item
16         android:id="@+id/navigation_family_profile"
17         android:icon="@drawable/ic_home_account"
18         android:title="@string/menu_family_profile_item"/>
19 </menu>
```

Programski kôd 4: Sadržaj datoteke *navigation.xml*

Nadalje, prezenteri PartnersFragmentPresenter i NewsFragmentPresenter učitavaju podatke s obzirom na cjelobrojnu vrijednost u prostoru za privremenu pohranu.

```
1  @Override
2  public boolean isFragmentActive() {
3      int storedIndex = SharedPrefManager
           .getInstance()
           .readSharedPref()
           .getInt(LocalConstants.SharedPref.KEY_CONTENT_CURRENT_PAGE, -1);
4      return storedIndex == 2;
5  }
```

Programski kôd 5: Metoda *isFragmentActive* klase `NewsFragmentPresenter`

U gornjem isječku kôda sporna je linija 4, jer se u njoj provjerava vrijednost pročitana iz privremene memorije i uspoređuje se s nekom naizgled proizvoljnom brojčanom vrijednošću. O ovome kôdu ovisi programska logika dohvata mrežnoga sadržaja, što znači da o toj prikazanoj usporedbi ovisi gotovo polovina funkcionalnosti aplikacije. Odmah je moguće predložiti korištenje konstante, iako će se u poglavlju 6.1 primijeniti drukčiji pristup.

Ovakvo rješenje nije skalabilno na, primjerice, 100 fragmenata. Čak i uz argument da aplikacija nikada neće imati više od 10 fragmenata, ta implementacija nije korektna jer krši

jedan SOLID princip: princip otvorenosti i zatvorenosti (engl. *Open-Closed Principle*, OCP). OCP nalaže da klasa mora biti otvorena za proširenje, ali zatvorena za promjene (Martin et al., 2018, prema Meyer B., 1988). Po Bertrandu Meyeru, koji je 1988. definirao taj princip, nužno je koristiti nasljeđivanje da bi se ostvario rezultat primjene tog principa. Međutim, po Robertu C. Martinu, koji spominje Meyerov stav (2018), nasljeđivanje nije nužno, već je naglasak na tome da se novim kôdom izbjegne ažuriranje staroga. Svakako je nužno izbjeći ažuriranje već postojećeg kôda. Indeksiranjem fragmenata u selekciji tipa *switch* u potpunosti se krši taj važan princip.

Prethodno je opisan arhitekturni nedostatak. Sljedeći nedostatak manje je arhitekturni, a više dizajnerski. Aplikacija nudi mrežni i izvanmrežni način rada, o čemu je već bilo riječi. Međutim, očekivalo bi se da izvanmrežni način rada usko koristi sloj podataka. Budući da sloj podataka ne postoji, klase za izvanmrežni rad (u projektu su to tzv. repozitoriji) upisuju podatke u lokalne datoteke. Taj pristup nije optimalan, bolje bi bilo koristiti bazu podataka. Klasa za rad s bazom podataka postoji (koristi biblioteku Room), ali nije nigdje korištena u cijeloj aplikaciji.

Također, sličan se nedostatak tiče pristupa mrežnim servisima. Postoji klasa `BaseWebserviceAPI` koja koristi biblioteku Retrofit. Biblioteku Retrofit preporučuje i sâm Google (*Get Data from the Internet*, 2022), ali njezino korištenje u projektu nije konzistentno. Na stranu to što neke metode unutar servisne klase nisu korištene, arhitekturni problem je taj što postoji još jedna klasa za pristup mrežnim servisima koja ne koristi Retrofit, već na niskoj razini upućuje poslužitelju zahtjev HTTP metodom. Ta druga klasa naziva se `NetworkUtils`. Na prvu se čini da ima mnoštvo korisnih metoda koje pružaju funkcionalnost nezamjenjivu Retrofitovim sučeljem. Međutim, nepovezana klasa poslovne logike `PartnerCardsAsyncTask` u metodi `doInBackground` izvršava sljedeću liniju kôda:

```
String responseString = NetworkUtils.getResponseFromUrl(url, cardNumber, true);
```

A potom:

```
PartnerCardResponse responseFromWeb = new Gson()
                        .fromJson(responseString, PartnerCardResponse.class);
```

Takvo ručno dobivanje objekta tipa `PartnerCardResponse` iz odgovora mrežnog servisa potpuno je pogrešno jer zaobilazi već pruženu funkcionalnost biblioteke Retrofit. Takva izravna pretvorba iz znakovnog niza u format JSON nepotrebna je jer biblioteka Retrofit isto izvršava implicitno, a Retrofit se koristi u ostatku projekta. Nadalje, poruka uz *commit* koji je ovaj kôd unio u projekt napisana je 2019., a glasi: „Mijenjanje procedure prikaza loadinga (još nije spremno).“ Očigledno je programer koji je pogriješio i zaobišao Retrofitova sučelja bio u potpunosti svjestan da promjena koju unosi u projekt nije ispravna, ili barem konačna. Nejasno

je zašto je programer implementirao istovjetnu metodu za Retrofit više od pola godine ranije, ali ju je odbio koristiti na ovome mjestu. Svakako, pristup izvoru podataka trebao bi biti jednak u cijelome projektu kako održavanje ne bi uključivalo razumijevanje dvaju potpuno različitih pristupa istome izvoru (istih) podataka.

Još jedan uočen nedostatak jest naziv klase:

`CheckIfPartnerCardDataExistsOrModifiedAsyncTask`.

Naziv klase izgleda poput naziva metode zbog toga što koristi glagol. Opća konvencija imenovanja nalaže da se klase nazivaju imenicama, a metode glagolima. Fowler (2019) koristi nazive metoda „usd“ i „totalAmount“, čime krši ovu konvenciju, a s izlikom da su to metode nastale iz lokalnih varijabli pa su im potrebni kratki i jasni nazivi. Međutim, to u gornjem nazivu nije slučaj te se ni po čemu ne može opravdati kršenje konvencije.

Loša arhitekturna odluka jest i to da se program pokreće aktiviranjem aktivnosti `MainActivity`, a toj aktivnosti jedini je zadatak otvoriti aktivnost `SignInActivity`. Razlog za tu odluku jest prikazivanje oznake organizacije koje traje jednu i pol sekundu. Programski kôd 6 prikazuje jedinu programsku logiku početne aktivnosti aplikacije.

```
1  @Override
2  protected void onCreate(@Nullable Bundle savedInstanceState) {
3      super.onCreate(savedInstanceState);
4      setContentView(R.layout.activity_main);
5      ButterKnife.bind(this);
6
7      Handler h = new Handler();
8      h.postDelayed(new Runnable() {
9          @Override
10         public void run() {
11             openNextView();
12         }
13     }, 1500);
14 }
```

Programski kôd 6: Programska logika klase `MainActivity`

Pitanje je ima li smisla operacijskom sustavu naglasiti da postoji aktivnost s nazivom „glavna“ (engl. *main*), a čiji je jedini smisao postojanje sekundu i pol u sustavu? Prema mišljenju autora ovoga rada, takva je klasa beskorisna te samo šteti korisničkom iskustvu tjeranjem korisnika da vrijeme provede čekajući i gledajući logoznak organizacije čiji je član. Čak i da taj prikaz ostane, prilikom refaktoriranja naziv aktivnosti trebalo bi obvezno promijeniti u „SplashActivity“, čime bi se jasno naznačio smisao aktivnosti.

Za kraj će biti naveden arhitekturni problem sloja korisničkog sučelja. Riječ je o povezivanju elemenata sučelja s programskom logikom. Arhitekt sustava odlučio je koristiti biblioteku Butterknife. Njezin osnovni problem u današnje vrijeme jest to da se više ne održava i da je službeno zastarjela. Opisna datoteka repozitorija biblioteke navodi i da se treba prebaciti na službeno rješenje za povezivanje elemenata korisničkog sučelja s varijablama u kôdu koje je sada standardni dio razvoja za Android: ViewBinding (Wharton, 2013/2020). Čak i da je biblioteka još uvijek aktivna, loša je praksa usko povezati kôd s bibliotekom. Robert C. Martin (2018) izričito ističe da razvojni okviri (u ovome slučaju riječ je o biblioteci) trebaju služiti kao alati, te da ih se ne smije pustiti da obuzmu projekt. Što se tiče ove aplikacije, ButterKnife je poput zloćudnog tumora metastazirao projektom. Također, u kôdu se na mnoštvo mjesta koristi i klasičan pristup elementima sučelja metodom *findViewById* nad objektom aktivnog pogleda, što nije dosljedno te zbog toga čini dizajnersku pogrešku u arhitekturi aplikacije.

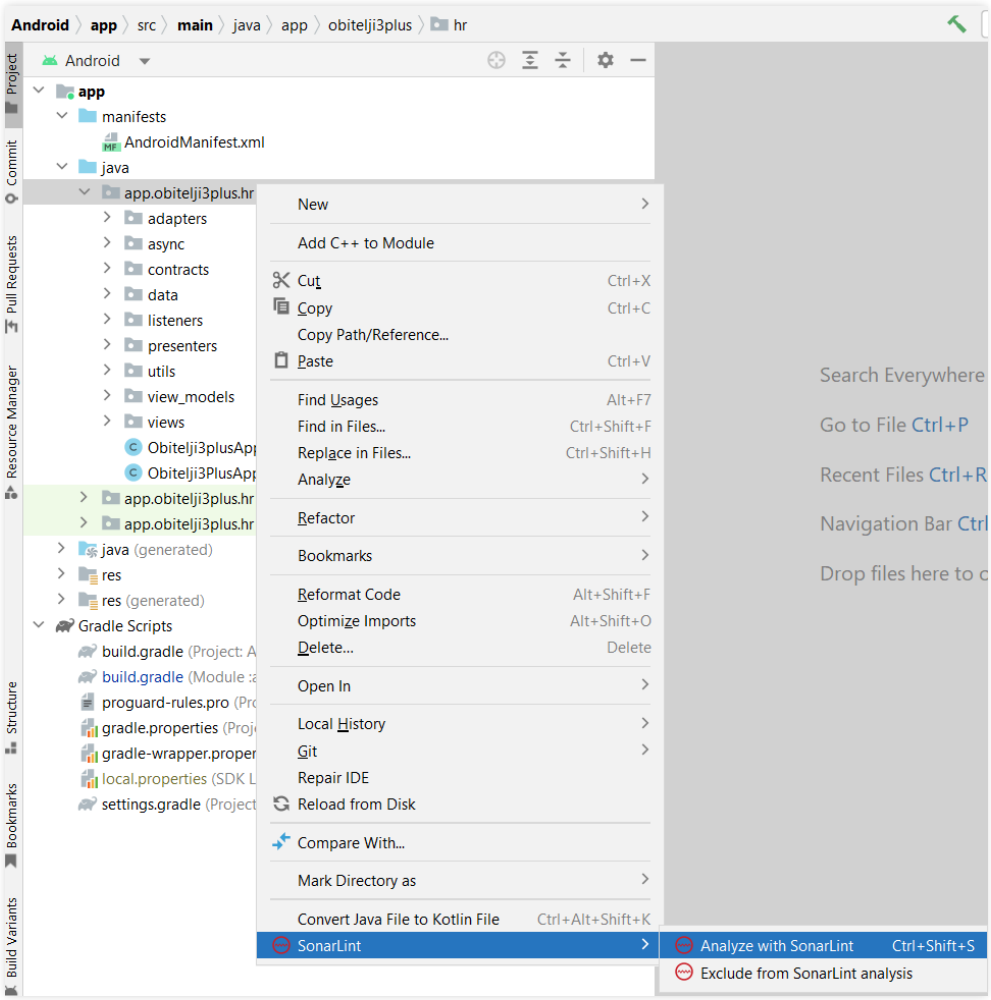
Navedeni su različiti nedostatci dizajna odabrane aplikacije, koji su ručno pregledani te opisani. U sljedećem poglavlju daje se uvid u to kako alati treće strane mogu automatizirati pregled kôda i besplatno ponuditi uvid u stanje aplikacije.

5.3.2. Automatizirana statička analiza projekta

U ovom poglavlju prikazuje se postavljanje i korištenje dvaju alata za analizu kvalitete kôda tvrtke SonarSource: SonarLint i SonarQube.

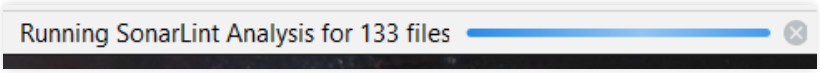
Analiza kôda može biti statička i dinamička, pri čemu se statička analiza odnosi na analiziranje kôda programa, a dinamička analiza proučava program tijekom njegova izvršavanja (Silva, 2013). U kontekstu statičke analize, prikazat će se postavljanja i izvješća alata SonarLint i SonarQube, a za dinamičku analizu tijekom izvršavanja programa koristi se platforma Firebase. Sve prikazane tehnologije besplatne su za korištenje u svrhu prikazanu u ovome radu.

Alat SonarLint besplatno je proširenje za alate za pisanje kôda (između ostaloga, i za Android Studio), a služi za pronalazak i ispravljanje problema, ranjivosti i *smrdljivog* kôda (*SonarLint - IntelliJ IDEs Plugin | Marketplace*, 2023). Proširenje nije komplicirano koristiti. Pritiskom desne tipke miša na odabrani paket otvara se izbornik u kojemu je moguće započeti analizu kôda. Slika 13 prikazuje taj izbornik otvoren nad projektom odabrane aplikacije.



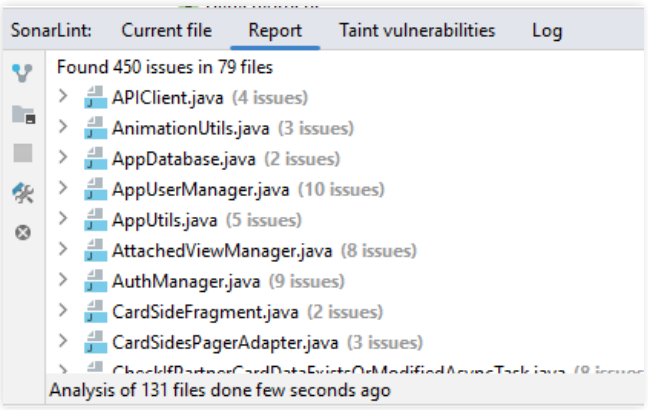
Slika 13: Aktiviranje proširenja SonarLint

Slika 14 prikazuje informaciju o radu SonarLintove analize.



Slika 14: Izvođenje analize kôda alatom SonarLint

Prijavljeno je 450 problema u 79 datoteka. Slika 15 prikazuje rezultat analize alata SonarLint unutar alata Android Studio.



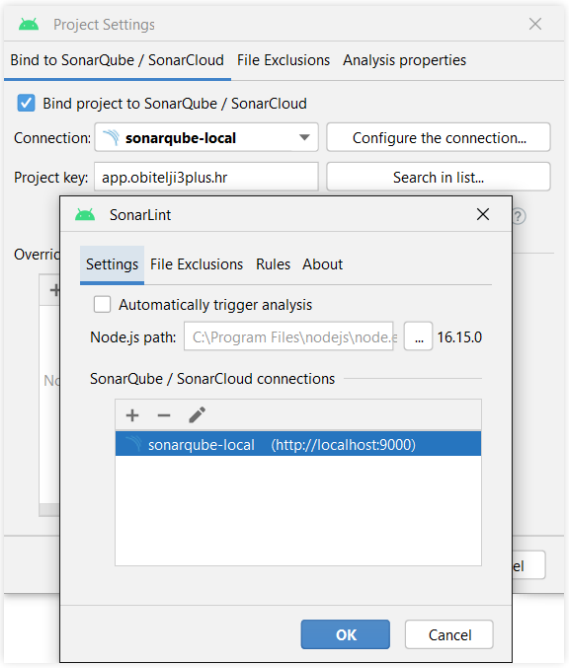
Slika 15: Prijavljeni problemi nakon SonarLintove analize

Iako navedeni problemi nisu usko vezani uz arhitekturu aplikacije, oni svejedno pružaju dobru početnu točku za detaljniji ručni pregled kôda. Na temelju njih moguće je prepoznati te izvesti redizajn povezanih komponenti.

SonarLint izvršava samo djelomičnu analizu. SonarQube je alat kojega je zajedno sa SonarLintom izdala tvrtka SonarSource, a koji djeluje poput platforme za provjeru stanja projekta pri čemu omogućava kontinuirane provjere kôda (*SonarQube 9.9*, 2022). SonarQube mnogo je robusniji od SonarLinta, počevši od toga da se izvršava na poslužitelju. SonarLint može se povezati na poslužitelja od SonarQubea kako bi prikazao rezultate obrade. SonarQube može se aktivirati na projektu za Android jednostavnom naredbom sustavu izgradnje projekata Gradleu:

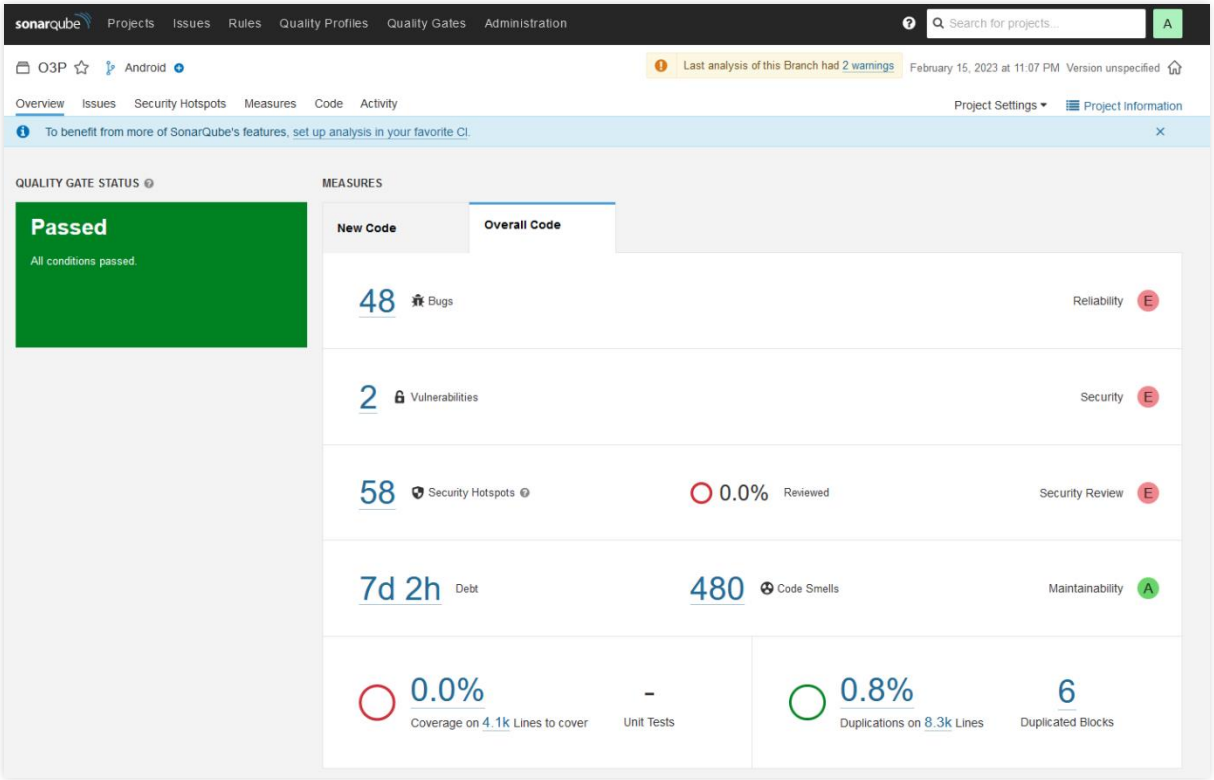
```
.\gradlew sonar
```

Tom naredbom aktivira se automatski pregled strukture projekta koji se izvršava na poslužiteljskoj strani. Tijekom pisanja ovoga rada, SonarQube aktiviran je kao kontejner na platformi Docker, čime je omogućeno brzo i jednostavno lokalno podizanje njegova poslužitelja. Povezivanje proširenja SonarLint s tim poslužiteljem u Android Studiju moguće je izvršiti u postavkama projekta. Slika 16 prikazuje dva prozora unutar razvojnoga okruženja Android Studio kojima se povezuje proširenje SonarLint i poslužitelj na kojemu se izvršava SonarQube (na slici je on pokrenut na lokalnoj adresi *localhost:9000*).



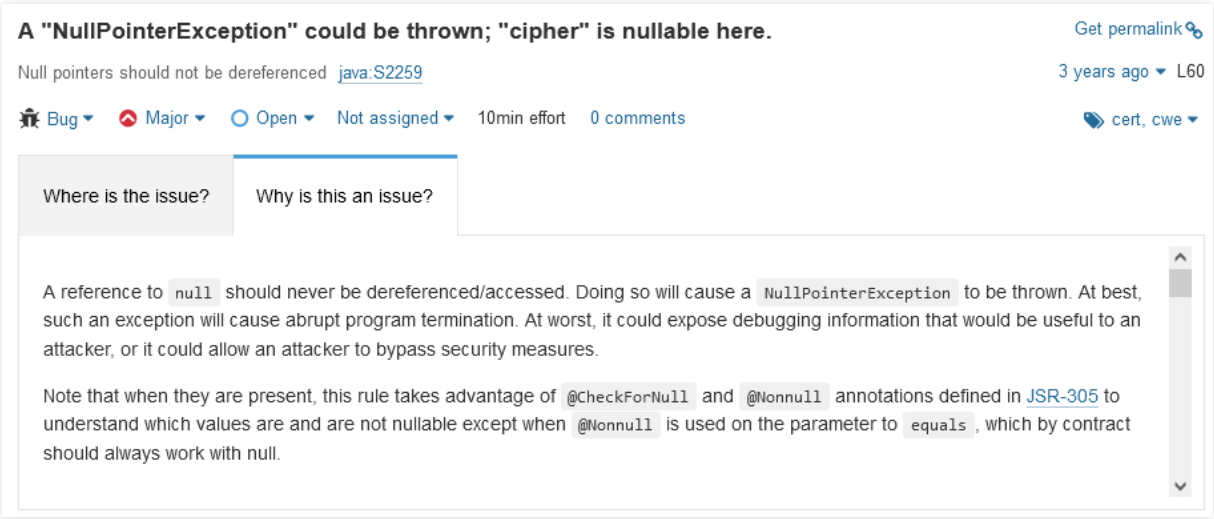
Slika 16: Povezivanje SonarLinta sa SonarQubeom

Slika 17 prikazuje izgled poslužitelja u internetskom pregledniku nakon izvršene prethodno navedene naredbe i odrađene analize kôda odabrane aplikacije.



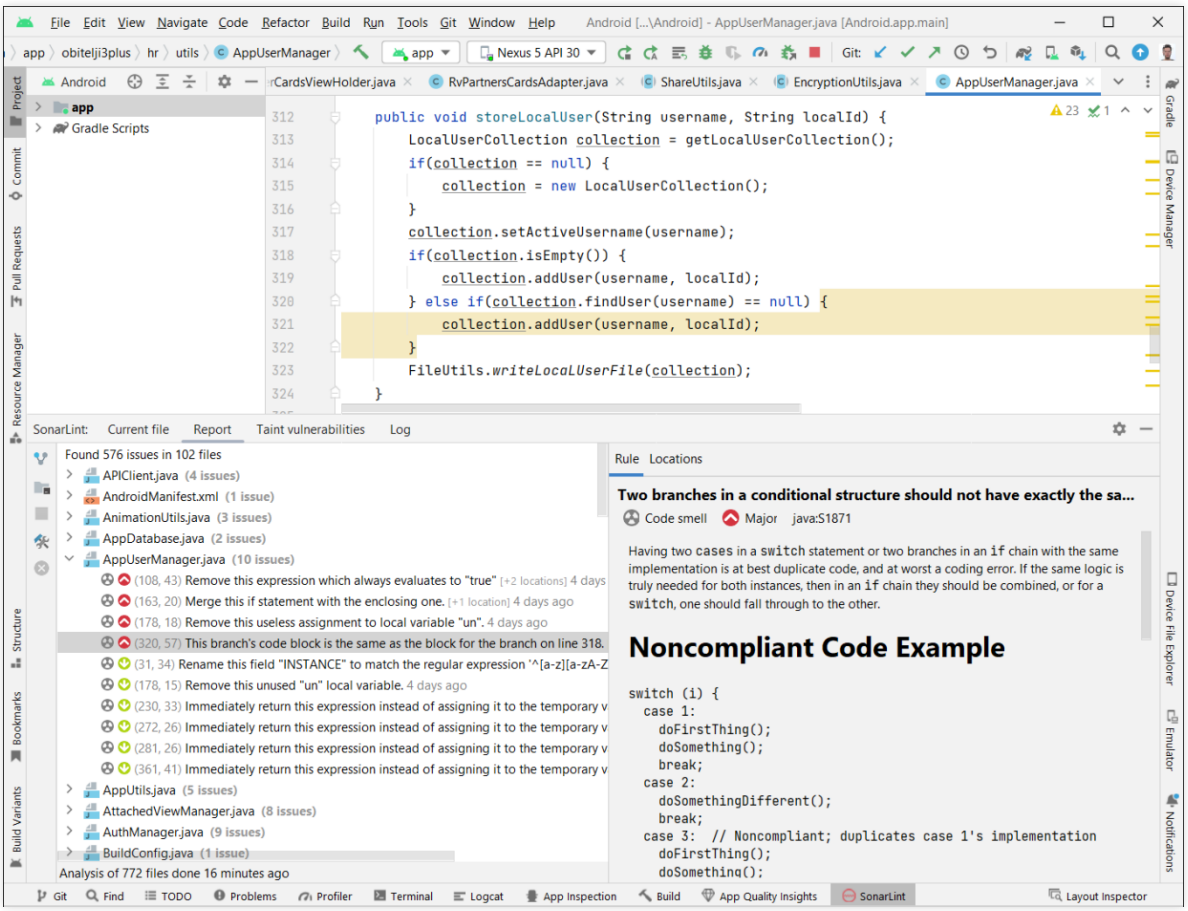
Slika 17: Prikaz SonarQubeovog rezultata analize u internetskom pregledniku.

Korisničko sučelje ovoga mrežnog alata vrlo je intuitivno. Slika 18 prikazuje kako SonarQube opisuje razlog zašto je nešto označio kao smrdljivi kôd.



Slika 18: Razlaganje pogreške u SonarQubeu

Budući da je SonarLint povezan sa SonarQubeom, on također prikazuje ta detaljna objašnjenja pogrešaka u samome razvojnom okružju. Otvaranjem prozora proširenja SonarLint moguće je pregledati 576 problema u 102 datoteke. Slika 19 prikazuje rezultat SonarLintove analize. Na slici je otvoren primjer jednog pronađenog problema uz otvoreni detaljni opis zašto je to problem.



Slika 19: Prikazan izvještaj proširenja SonarLint u razvojnoj okolini Android Studio s jednim otvorenim kritičnim *smrdljivim* kôdom

Od svih prepoznatih problema, u ovome će poglavlju biti istaknuto samo nekoliko njih. Prvi problem prikazan je gore na slici. Alat je uspješno pronašao besmislenu logiku gdje obje grane selekcije rezultiraju obavljanjem istoga posla. Sličan problem pronađen je i malo ranije u istoj klasi, a prikazan je u sljedećem isječku kôda.

```
1  if(!currentUser.getUsername().equals(activeUsername)) {
2      if(isLocalUserSharedPrefExpired(currentUser.getLocalId())){
3          cleanSharedPrefByLocalId(currentUser.getLocalId());
4      }
5  }
```

Programski kôd 7: Prepoznat smrdljiv kôd u ugniježđenim selekcijama

Ovaj kôd dodan je nekoliko dana prije prethodnog. Oba kôda bespotrebno razdvajaju logičke uvjete u više dijelova. Moguće je da je originalni autor kôda odlučio jasno razložiti logiku kako bi bila čitljivija umjesto da povezuje logičke uvjete u cjelinu. Stoga, nije nužno izvesti refaktoriranje nad ovim kôdom unatoč tome što ga je alat označio crvenom bojom.

Zanimljiv je sljedeći obrazac koji je SonarLint prepoznao.

```
1    try {
2        GenericUtils.logger(TAG, "getTokenData:
             ****************************************"
         );
3        GenericUtils.logger(TAG,
             "getTokenData: " + " tokenValue = " + tokenValue
         );
4        GenericUtils.logger(TAG,
             "getTokenData: " + " tokenExpiresIn = " + tokenExpiresIn
         );
5        GenericUtils.logger(TAG,
             "getTokenData: " + " issuedStringDate = " + issuedStringDate
         );
6        GenericUtils.logger(TAG,
             "getTokenData: " + " expiresStringDate = " + expiresStringDate
         );
7        GenericUtils.logger(TAG,
             "getTokenData: ****************************************"
         );
8        authenticationData.setIssued(issuedStringDate == null ? null :
             simpleDateFormat.parse(issuedStringDate));
9        authenticationData.setExpires(expiresStringDate == null ? null :
             simpleDateFormat.parse(expiresStringDate));
10   } catch (ParseException e) {
11       e.printStackTrace();
12       Toast.makeText(
                     context,
                     R.string.error_incorrect_datetime_format,
                     Toast.LENGTH_LONG
             ).show();
13   }
```

Programski kôd 8: Prepoznat smrdljiv kôd u ponavljajućem znakovnom nizu

SonarLint prepoznao je da se na linijama numeriranima od 2 do 7 ponavlja znakovni niz „getTokenData“ pa stoga predlaže vađenje tog niza iz ovog dijela kôd i smještanje u neku globalnu varijablu. Trenutačno je ovaj kôd za provjeru valjanosti žetona korisničke prijave *zagađen* porukama programeru koje čak sadržavaju tajne podatke (vrijednost žetona na 3. liniji). To je vrlo dobro uočen problem, ali njegovo rješenje bilo bi smještanje cijelog dijela ispisa poruka u vanjsku metodu, pa možda čak i u vanjsku klasu koja ima samo odgovornost ispisa poruka. Svakako, SonarLint dobro je uočio taj problem.

Posljednji konkretni problem koji će biti prikazan prema analizi SonarLint djelomično je vezan uz arhitekturu aplikacije. Prepoznato je korištenje konstanti unutar sučelja kao u kôdu ispod. Višak kôda skriven je komentarima s tri točke.

```
1  public abstract class RemoteConstants {
2      public interface API {
3          String BASE_URL = "https://obitelji3plus.hr/";
4          String HEADER_AUTH_KEY = "Authorization";
5          String HEADER_AUTH_VALUE = "Bearer %s";
6          interface URL {
7              String LOGIN_URL = "api/v1/login.php";
8              String NEWS_URL = "api/v1/news.php";
9              // ...
10         }
11         interface Response
12         {
13             String KEY_CARD_VALID_TO = "valid_to";
14             String KEY_CARD_VALID_TO_DATE = "date";
15         }
16         interface RequestValues {
17             String ID = "id";
18             String LIMIT = "limit";
19             // ...
20         }
21     }
22     public interface StringFormat {
23         String FORMAT_DATETIME_FORMAT_TOKEN = "d MMM yyyy HH:mm:ss";
24         String FORMAT_DATETIME_FORMAT_NEWS = "d MMM yyyy HH:mm";
25         // ...
26     }
27 }
```

Programski kôd 9: Prepoznat smrdljiv kôd u apstraktnoj klasi sa sučeljima

Sudeći po poruci uz *commit* iz 2018., koji je uveo gornju klasu u projekt, klasa `RemoteConstants` osmišljena je kao držač svih konstanti vezanih uz komunikaciju s mrežnim servisima u projektu. Tijekom vremena, ova se klasa proširila i obuhvatila razne druge konstante, od zastavica koje se smještaju izravno u poveznicu na servis do formata prikaza datuma. Za početak, SonarLint je ovoj klasi zamjerio njezin status apstraktne klase iako nema potencijal nasljeđivanja jer ne sadržava niti jedno svojstvo ili metodu. Mnogo je više zamjerio sučelja koja umjesto da imaju ulogu pružanja ugovora o implementaciji, sadržavaju već definirane znakovne nizove. Predložio je pomak konstanti iz tih sučelja u drugu klasu ili enumeraciju. Ono što je ispravna odluka jest izvršiti redizajn arhitekture na način da svi konstantni znakovni nizovi budu premješteni u jednu klasu kojoj su potrebni, koja će imati odgovornost pristupa mrežnom servisu. Moguće je i izolirati te znakovne nizove u vanjsku

datoteku (primjerice tipa JSON) i onda ih koristiti iz nje, čime će se kôd u potpunosti osloboditi potrebe za promjenom prilikom mijenjanja domene servisa, naziva atributa zahtjeva ili odgovora itd. Što se tiče znakovnih nizova formata datuma, oni bi mogli pripasti statičkim svojstvima neke klase koja će sadržavati isključivo njih.

Ostala problematična mjesta u kôdu koja je SonarQube prepoznao uglavnom se tiču prijedloga vezanih uz odabir poziva metoda, nazive elemenata kôda, skrivanje konstruktora kod klasa koje sadržavaju samo statičke metode, pretvorbe anonimnih klasa u lambda funkcije, izbjegavanje poziva zastarjelih metoda, upozoravanje na opasnost od iznimki *null* vrijednosti, brisanje komentara itd.

Ova dva alata korištena zajedno iznimno su korisni pri identificiranju problema u kôdu. Oni uvelike olakšavaju prepoznavanje kritičnih točaka u kôdu i daju dobar temelj za planiranje nove arhitekture.

6. Refaktoriranje i arhitekturni redizajn aplikacije

Dok je prethodno poglavlje bilo uvod u odabranu aplikaciju te su u njemu navedeni i različiti nedostatci te aplikacije, tema je ovoga poglavlja redizajn aplikacije s obzirom na sve do sada navedene nedostatke. Cilj je poglavlja predložiti bolju aplikaciju, koju je jednostavnije održavati i nadograđivati.

Martin Fowler (2019) navodi da je promjena vodilja refaktoriranja, tj. ako kôd radi, nema potrebe za promjenom. Međutim, ako treba doradu, treba izvršiti redizajniranje softvera. Stoga je u ovom poglavlju naglasak na promjenama na aplikaciji te objašnjenju kako refaktoriranje i arhitekturni redizajn utječu na implementiranje tih promjena. Promjene koje se trebaju implementirati jesu sljedeće:

- novi prikaz za obavljanje plaćanja
- mogućnost prijave otiskom prsta.

Promjene neće biti implementirane u radu. Međutim, obje promjene obuhvaćaju prikaz sadržaja. Budući da ispravan rad prikaza nije jednostavno jedinično testirati, ručno je provjeren rad aplikacije nakon svakog navedenog koraka promjene u sljedeća dva poglavlja.

6.1. Redizajn načina prikaza kartica

Programski kôd 2 prikazuje trenutačni način upravljanja fragmentima početnog ekrana. U poglavlju 5.3.1 detaljno je objašnjeno zašto je u tom isječku potrebna promjena. Ovo poglavlje bavi se refaktoriranjem načina prikaza kartica.

Aplikacija trenutačno koristi zastarjeli način upravljanja elementom sučelja ViewPager. Taj element sučelja (tzv. pogled, engl. *View*) omogućuje izmjenu fragmenata potezom prsta. Njegov slušač događaja promjene fragmenta postavljen je na sljedeći način.

```
1   @Override
2   protected void implementActivityLogic(@Nullable Bundle savedInstanceState) {
3    this.setupViewPager();
4    btmContentNavigation.setOnNavigationItemSelectedListener(
         mOnNavigationItemSelectedListener
     );
5    if(!isFirstRun) {
6        this.reloadData();
7    }
8
9    isFirstRun = false;
10  }
```

Programski kôd 10: Postavljanje ViewPagera u klasi `ContentActivity`

Metodi *setOnNavigationItemSelectedListener* s linije 4 proslijeđen je objekt slušača događaja odabira nove kartice. Programski kôd 2 prikazuje tog slušača. Metoda je zastarjela, pa je uklonjena (broj linije označen crvenom bojom).

Pogrešno je rješenje da klasa čija je odgovornost upravljanje s aktivnosti također ovako eksplicitno upravlja i cijelim sustavom promjene kartica. Ispravno bi bilo odvojiti aktivnost od upravljanja promjenom fragmenata i kartica u novu klasu čija će to biti jedina odgovornost. To bi odgovaralo definiciji principa jedne odgovornosti (engl. *Single Responsibility Principle*, SRP) koji definira da modul (u ovom kontekstu to može biti i softverska komponenta na dizajnerskoj razini, poput klase) ima odgovornost prema samo jednom izvoru utjecaja koji je ujedno i jedini razlog mijenjanja (Martin et al., 2018). Nova klasa imat će stoga zadatak upravljati odnosom elementa za prikaz fragmenata i elementa za prikaz gumba kartica (element `BottomNavigationView`). Prije definiranja te klase, treba osigurati okolinu u kojoj ona može obavljati svoju dužnost.

Treba pregledati fragmente jer su oni srž funkcionalnosti koja će se redizajnirati. Svi fragmenti iz nepoznatog razloga sadržavaju metodu `getInstance` koja ne radi ništa korisno

osim što sakriva naredbu *new*. Programski kôd 11 prikazuje tu metodu na primjeru klase `FamilyProfileFragment`.

```
1  public static FamilyProfileFragment getInstance(){
2      Bundle args = new Bundle();
3
4      FamilyProfileFragment fragment = new FamilyProfileFragment();
5      fragment.setArguments(args);
6      return fragment;
7  }
```

Programski kôd 11: Metoda *getInstance* u fragmentima

Naziv metode odaje dojam uzorka dizajna Singleton. Takva apstrakcija nad instanciranjem objekta nije objašnjena u komentarima pa je uklonjena iz svih četiriju klasa fragmenata. Nakon testiranja rada aplikacije sve je bilo u redu pa je nastavljen rad stvaranjem klase za prilagodbu fragmenata elementu za njihov prikaz. Element za uzastopan prikaz fragmenata zove se `ViewPager2`. Sufiks „2“ označava moderniju verziju zastarjele klase `ViewPager` koja se koristi u ovome projektu. Stoga, zamjena stare klase novom čini dobar dio posla refaktoriranja prikaza fragmenata. `ViewPager2` zahtijeva adapter. U projektu već postoji prigodna klasa s nazivom `MainContentPagerAdapter` (MCPA) korištena za stari `ViewPager`. Njezina je odgovornost znati stvoriti odgovarajuće fragmente za glavni prikaz, a njezin je kôd ispod.

```
1  public class MainContentPagerAdapter extends FragmentStatePagerAdapter {
2      private List<Fragment> fragments = null;
3
4      public MainContentPagerAdapter(FragmentManager fm) {
5          super(fm);
6          this.fragments = new ArrayList<>();
7      }
8
9      @Override
10     public Fragment getItem(int position) {
11         return fragments.get(position);
12     }
13
14     @Override
15     public int getCount() {
16         return fragments.size();
17     }
18
19     public void addFragment(Fragment fragment) {
20         this.fragments.add(fragment);
21     }
22 }
```

Programski kôd 12: Zastarjeli kôd klase MCPA

Nažalost, klasa u projektu nasljeđuje zastarjelu `FragmentStatePagerAdapter` klasu i zbog toga zahtijeva refaktoriranje. Moderna alternativa je klasa `FragmentStateAdapter`.

```
1  public class MainContentPagerAdapter extends FragmentStateAdapter
2  {
3      public final List<Integer> iconList = new ArrayList<>();
4      public final List<Integer> titleList = new ArrayList<>();
5
6      public MainContentPagerAdapter(@NonNull final FragmentManager fragmentManager,
7                                     @NonNull final Lifecycle lifecycle)
8      {
9          super(fragmentManager, lifecycle);
10         fillIcons();
11         fillTitles();
12     }
13
14     private void fillIcons()
15     {
16         iconList.add(R.drawable.ic_account_card_details);
17         iconList.add(R.drawable.ic_city);
18         iconList.add(R.drawable.ic_newspaper);
19         iconList.add(R.drawable.ic_home_account);
20     }
21
22     private void fillTitles()
23     {
24         titleList.add(R.string.menu_family_card_item);
25         titleList.add(R.string.menu_partners_item);
26         titleList.add(R.string.menu_news_item);
27         titleList.add(R.string.menu_family_profile_item);
28     }
29
30     @NonNull
31     @Override
32     public Fragment createFragment(final int position)
33     {
34         Fragment result;
35         switch (position)
36         {
37             default:
38             case 0: {
39                 result = new FamilyCardFragment();
40                 break;
41             }
42             case 1: {
43                 result = new PartnersFragment();
44                 break;
45             }
46             case 2: {
47                 result = new NewsFragment();
48                 break;
49             }
50             case 3: {
51                 result = new FamilyProfileFragment();
52                 break;
53             }
54         }
55         return result;
56     }
57
58     @Override
59     public int getItemCount()
60     {
61         return 4;
62     }
63 }
```

Programski kôd 13: Prva verzije refaktorirane klase MCPA

Kako bi se ispravio problem definicije kartica nevezane uz fragmente u kôdu, nova verzija klase sadržava popis naslova i ikona za kartice, a ujedno i zna koji fragment pripada kojoj kartici.

Gornji kôd omogućuje potpun zaobilazak datoteke *navigation.xml* (Programski kôd 4). Međutim, situacija nije dobra zbog razdvajanja fragmenata i njima povezanih ikona i naslova. Taj će se problem riješiti kasnije, jer ovo refaktoriranje sustava prikaza fragmenata zahtijeva za početak mnoge promjene u prikazu. Nužno je zamijeniti originalni ViewPager element s ažuriranom verzijom ViewPager2. Ta promjena uključuje refaktoriranje cijelog prethodnog načina prikaza kartica. Prethodni način prikaza ViewPagera i kartica opisan je u programskom kôdu ispod.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout ...>
3
4      <app.obiteljI3plus.hr.utils.MyViewPager
5          android:id="@+id/vpMainContent"
6          android:layout_width="match_parent"
7          android:layout_height="wrap_content"
8          android:layout_alignParentTop="true" />
9
10     <include layout="@layout/component_bottom_nav_bar" />
11
12 </RelativeLayout>
```

Programski kôd 14: Originalna struktura početnoga ekrana

U projektu nije bio korišten „čisti“ ViewPager, već klasa koja ga nasljeđuje – MyViewPager. Ipak, nasljeđivanje je bilo gotovo besmisleno pa ga nije nužno ponoviti. Promjena je izvršena na način da su datoteke *MyViewPager.java*, *component_bottom_nav_bar.xml* i već prikazan *navigation.xml* izbrisane iz projekta. Prikaz glavnog ekrana pretvoren je u strukturu tipa *ConstraintLayout*, a njegov novi opis strukture nalazi se u nastavku.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        android:id="@+id/rlContentContainer"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:fitsSystemWindows="true"
        tools:context=".views.activities.ContentActivity">

    <androidx.viewpager2.widget.ViewPager2
            android:id="@+id/vpMainContent"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_alignParentTop="true" />

    <com.google.android.material.bottomnavigation.BottomNavigationView
            android:id="@+id/btmContentNavigation"
            android:layout_width="match_parent"
            android:layout_height="60dp"
            app:tabTextAppearance="@style/TextAppearance.AppCompat.Small"
            android:background="@color/colorRed"
            app:itemBackground="@color/colorRed"
            app:itemIconTint="@color/bttm_nav_bar_colors"
            app:itemTextColor="@color/bttm_nav_bar_colors"
            app:tabSelectedTextColor="@color/colorYellow"
            android:layout_alignParentBottom="true" />

</RelativeLayout>
```

Programski kôd 15: Poboljšana struktura početnoga ekrana

Konkretno, umjesto povezivanjem XML datoteka, sada je struktura glavnog prikaza u jednoj XML datoteci. Struktura prikaza početnog ekrana u XML formatu ni na koji način više ne definira i sadržaj. Sustav prikaza kartica ispod elementa klase `ViewPager2` mora biti određen programskim kôdom. Za tu je potrebu stvorena nova klasa `BottomNavigationMediator` (BNM).

```
1  public class BottomNavigationMediator
2  {
3      private final BottomNavigationView bottomNavigation;
4      private final ViewPager2 viewPager;
5      private int selectedIndex = 0;
6
7      public BottomNavigationMediator(
           final BottomNavigationView bottomNavigation,
           final ViewPager2 viewPager)
8      {
9          this.bottomNavigation = bottomNavigation;
10         this.viewPager = viewPager;
11     }
12
13     public void attach()
14     {
15         viewPager.registerOnPageChangeCallback(
               new ViewPager2.OnPageChangeCallback()
16         {
17             @Override
18             public void onPageSelected(final int position)
19             {
20                 MenuItem oldItem =
                       bottomNavigation.getMenu().getItem(selectedIndex);
21                 oldItem.setChecked(false);
22
23                 selectedIndex = position;
24
25                 MenuItem nextItem =
                       bottomNavigation.getMenu().getItem(selectedIndex);
26                 nextItem.setChecked(true);
27             }
28         });
29
30         this.bottomNavigation.setOnItemSelectedListener(item -> {
31             selectedIndex = item.getOrder();
32             viewPager.setCurrentItem(selectedIndex);
33             return true;
34         });
35     }
36
37 }
```

Programski kôd 16: Početni kôd klase `BottomNavigationMediator`

Klasa iz gornjeg kôda napisana je po uzorku dizajna Mediator (Gamma et al., 1994), a pisana je po uzoru na klasu `TabLayoutMediator` koja se nalazi u modernom Androidovom SDK-u kao poveznica između ViewPagera i elementa `TabLayout`. Problem s klasom `BottomNavigationView` jest taj da više od pet kartica uzrokuje pucanje sustava zbog iznimke. Ako bi odabrana aplikacija trebala imati više od pet osnovnih navigacijskih kartica na glavnome ekranu, trebalo bi razmisliti o prelasku na element `TabLayout`. Ali tada, također, treba promisliti

o smislu tolikog broja osnovnih mogućnosti zbog opasnosti gubitka intuitivnosti korištenja aplikacije.

Ove promjene svele su metodu *setupViewPager* klase `ContentActivity` na svega četiri linije.

```
1  private void setupViewPager() {
2      this.adapter = new MainContentPagerAdapter(
           getSupportFragmentManager(),
           getLifecycle()
       );
3      vpMainContent.setAdapter(adapter);
4      new BottomNavigationMediator(btmContentNavigation, vpMainContent).attach();
5      vpMainContent.setOffscreenPageLimit(3);
6  }
```

Programski kôd 17: Prva refaktorirana verzija metode *setupViewPager*

U liniji 4 instanciran je novi objekt klase `BottomNavigationMediator` i metodom *attach* povezani su slušači događaja dvaju pogleda. Ovime se sve svelo isključivo na kôd, čime se omogućuje mnogo veća fleksibilnost u prikazu kartica. Primjerice, iste kartice mogu se prikazati kao gumbi, kao padajući izbornik, kao natuknice nekog drugog izbornika, a teoretski se nove kartice mogu i automatski preuzimati s mreže radi ažuriranja funkcionalnosti, čime bi se omogućila znatno veća razina fleksibilnosti nego u prethodnom rješenju. Ono što nedostaje jest način definiranja izgleda pojedinih kartica na pogledu tipa `BottomNavigationView`. To bi svakako bila odgovornost klase BNM, ali pitanje je kako omogućiti toj klasi da u svakom korištenju omogući različito stvaranje izgleda. Rješenje se nalazi u korištenju Strategy uzorka dizajna (Gamma et al., 1994). Riječ je o uzorku dizajna koji omogućuje izmjenjivost korištenog algoritma ovisno o situaciji. Klasa će u konstruktoru prihvatiti objekt strategije ne znajući njegovu implementaciju, već samo mogućnost. Kasnije će dohvaćati informacije iz te strategije. Informacije se mogu učahuriti u internoj klasi unutar klase BNM. Ta interna klasa prikazana je ispod.

```
1  public static class ContentItem
2  {
3      private final int icon;
4      private final int title;
5
6      public ContentItem(final int icon, final int title)
7      {
8          this.icon = icon;
9          this.title = title;
10     }
11
12     public int getIcon()
13     {
14         return icon;
15     }
16
17     public int getTitle()
18     {
19         return title;
20     }
21 }
```

Programski kôd 18: Klasa za jedinstvenu definiciju sadržaja na glavnome zaslonu

Uzorak dizajna Strategy bit će omogućen dodavanjem sučelja, također interno u klasu BNM, s definicijom kao u isječku kôda ispod.

```
1  public interface TabConfigurationStrategy
2  {
3      ContentItem getContentItemForPosition(int position);
4  }
```

Programski kôd 19: Sučelje za ostvarenje izmjenjivog algoritma definiranja kartica

Klasa BNM pohranjuje referencu na objekt toga sučelja, a zatim izvršava sljedeću metodu za punjenje navigacije karticama po uzoru na fragmente ViewPagera2.

```
1  private void fillBottomNavigation()
2  {
3      for (int pos = 0; pos < Objects.requireNonNull(viewPager.getAdapter()).getItemCount(); pos++)
4      {
5          ContentItem contentItem =
               configurationStrategy.getContentItemForPosition(pos);
6
7          MenuItem menuItem = bottomNavigation.getMenu()
                                    .add(Menu.NONE, pos, pos, contentItem.getTitle());
8          menuItem.setIcon(contentItem.getIcon());
9      }
10 }
```

Programski kôd 20: Punjenje navigacijskog izbornika s karticama fragmenata

Prikazana metoda u liniji 3 započinje petlju za svaki fragment u adapteru ViewPagera2. Za svaki se fragment dohvaća objekt stavke sadržaja. Programski kôd 18 prikazuje definiciju klase tih objekata. S podatcima koji su učahureni u tim klasama nije teško izraditi stavku izbornika pozivima metoda u linijama 7 i 8. Sada je potrebno samo implementirati ovu strategiju.

```
1  private void setupViewPager() {
2      this.adapter = new MainContentPagerAdapter(getSupportFragmentManager(), getLifecycle());
3      vpMainContent.setAdapter(adapter);
4      new BottomNavigationMediator(btmContentNavigation, vpMainContent, position ->
5              new BottomNavigationMediator
6                      .ContentItem(adapter.iconList.get(position), adapter.titleList.get(position))
7      ).attach();
8      vpMainContent.setOffscreenPageLimit(3);
9  }
```

Programski kôd 21: Druga refaktorirana verzija metode *setupViewPager*

U ovoj drugoj verziji metode *setupViewPager* lambda funkcijom definira se strategija tako da se odmah vraća novi objekt klase `ContentItem` s ikonom i naslovom iz adaptera za objekt klase `ViewPager2`.

Međutim, uvedene promjene stvorile su problem. Fragment za prikaz partnera i fragment za prikaz vijesti prestali su dohvaćati sadržaj s internetskog servisa. Nakon detaljne analize rada aplikacije, uočeno je da poslovna logika ta dva fragmenta ovisi isključivo o jednoj zastarjeloj metodi: *setUserVisibleHint*. Tu je metodu nad fragmentom pozivao zastarjeli ViewPager kada bi fragment bio prikazan. Međutim, ViewPager2 ne pozva tu metodu pa je stoga nužno izgraditi svoj sustav obavještavanja fragmenata da su prikazani (kako bi u ovom slučaju dva spomenuta fragmenta ažurirala sadržaj s mrežnih izvora). Postoje dva načina kako bi se ovo moglo implementirati:

1. ContentActivity mogao bi preko objekta *supportFragmentManager* izazvati događaj na koji bi se dva fragmenta odazvala ažuriranjem sadržaja. Događaji se hvataju po ključu (znakovni niz). Stoga su suboptimalni u ovoj situaciji jer nisu pretjerano objektno orijentirani. Imali bi smisla kada bi postojalo mnoštvo događaja.
2. Moglo bi se uvesti osluškivanje događaja promjene fragmenta u BNM, a na taj događaj bi se aktivirao objekt neke nove klase čija bi odgovornost bila obavještavanje fragmenata o aktiviranju. Takav pristup objektno je orijentiran i skalabilan.

Zbog razloga izloženih u opisu obaju načina odlučeno je odabrati drugi način. Za početak, potrebno je definirati slušač događaja, po uzoru na uzorak dizajna Observer (Gamma et al., 1994). To će biti interno sučelje klasi BNM čiji objekt klasa BNM prima preko metode za postavljanje atributa (engl. *setter*).

```
1  private ChangePageListener changePageListener = null;
2  ...
3  public interface ChangePageListener
4  {
5      void onChangePage(int oldPosition, int newPosition);
6  }
7  ...
8  public void setChangePageListener(final ChangePageListener changePageListener)
9  {
10     this.changePageListener = changePageListener;
11 }
```

Programski kôd 22: Pregled slušača događaja promjene fragmenta u klasi BNM

Dakle, ta metoda poziva slušača događaja ako je on postavljen *setterom*. Klasa BNM poziva slušača ovog događaja na dva mjesta u metodi *attach*.

```
1  public void attach()
2  {
3      viewPager.registerOnPageChangeCallback(new ViewPager2.OnPageChangeCallback()
4      {
5          @Override
6          public void onPageSelected(final int position)
7          {
8              MenuItem oldItem = bottomNavigation.getMenu().getItem(selectedIndex);
9              oldItem.setChecked(false);
10
11             updateSelectedIndex(position);
12
13             MenuItem nextItem = bottomNavigation.getMenu().getItem(selectedIndex);
14             nextItem.setChecked(true);
15         }
16     });
17
18     this.bottomNavigation.setOnItemSelectedListener(item -> {
19         updateSelectedIndex(item.getOrder());
20         viewPager.setCurrentItem(selectedIndex);
21         return true;
22     });
23 }
```

Programski kôd 23: Pozivanje metode updateSelectedIndex

Linije na kojima se poziva događaj promjene indeksa označene su zelenom bojom na gornjem isječku kôda. Metoda koja se poziva definirana je na sljedeći način.

```
1  private void updateSelectedIndex(int newSelectedIndex)
2  {
3      if (changePageListener != null)
4      {
5          changePageListener.onChangePage(selectedIndex, newSelectedIndex);
6      }
7      selectedIndex = newSelectedIndex;
8  }
```

Programski kôd 24: Metoda koja osigurava obavijest o promjeni stranice

Metoda *setupViewPager* sada izgleda kao u nastavku.

```
1  private void setupViewPager() {
2      this.adapter = new MainContentPagerAdapter(...);
3      vpMainContent.setAdapter(adapter);
4
5      BottomNavigationMediator bnm = new BottomNavigationMediator(
           btmContentNavigation,
           vpMainContent,
6          position ->
7                  new BottomNavigationMediator
                       .ContentItem(
                           adapter.iconList.get(position),
                           adapter.titleList.get(position)
                       )
8      );
9      bnm.setChangePageListener((oldPosition, newPosition) -> {
10         currentItem = newPosition;
11         storeCurrentPage();
12     });
13     bnm.attach();
14
15     vpMainContent.setOffscreenPageLimit(3);
16 }
```

Programski kôd 25: Treća refaktorirana verzija metode *setupViewPager*

Ovako se u klasi `ContentActivity` upravlja događajem promjene fragmenta. Metoda *storeCurrentPage* s linije 11 nije važna u ovome kontekstu – ona samo pohranjuje broj trenutačnog fragmenta u priručnu memoriju aplikacije radi provjere aktivnosti fragmenta.

Ono što nedostaje jest komuniciranje prema fragmentima i zamjena poziva metode koja dohvaća sadržaj s *web* servisa. Dovoljno je definirati sučelje s nazivom `MainContentFragmentContract` (MCFC). To sučelje neće implementirati fragmenti, već neka klasa koju tek treba stvoriti, a koja će za odgovornost imati obavještavanje fragmenata. To sučelje deklarirano je u kôdu u nastavku.

```
1  public interface MainContentFragmentContract
2  {
3      Fragment createFragment() throws IllegalAccessException, InstantiationException;
4      void activate();
5  }
```

Programski kôd 26: Sučelje čija će implementacija omogućavati aktiviranje fragmenata

Prva metoda vratit će objekt određenog fragmenta, a druga će aktivirati isti objekt, tj. javiti mu da je aktivan i da obavi nužnu logiku. Što se tiče prve metode, ona će biti pozvana u adapteru pri odabiru novog fragmenta. Programski kôd 13**Pogreška! Izvor reference nije pronađen.** prikazuje stari kôd adaptera. U tom originalnom kôdu, fragmenti se stvaraju po točno određenom rasporedu. Time se narušilo usko vezanje fragmenata i kartica (ikone i

naslova) koje je u ovom slučaju nužno ostvariti. Adapter zahtijeva temeljite promjene, a novi kôd adaptera prikazan je u nastavku.

```
 1  public class MainContentPagerAdapter extends FragmentStateAdapter
 2  {
 3      private List<MainContentPagerItem> pagerItems = new ArrayList<>();
 4
 5      public MainContentPagerAdapter(@NonNull final FragmentManager fragmentManager,
 6  @NonNull final Lifecycle lifecycle)
 7      {
 8          super(fragmentManager, lifecycle);
 9      }
10
11      public void setPagerItems(final List<MainContentPagerItem> contentPagerItems)
12      {
13          this.pagerItems = contentPagerItems;
14      }
15
16      @SneakyThrows
17      @NonNull
18      @Override
19      public Fragment createFragment(final int position)
20      {
21          return pagerItems.get(position).content.createFragment();
22      }
23
24      @Override
25      public int getItemCount()
26      {
27          return 4;
28      }
29
30      public int getIcon(final int position)
31      {
32          return pagerItems.get(position).icon;
33      }
34
35      public int getTitle(final int position)
36      {
37          return pagerItems.get(position).title;
38      }
39
40      public void onFragmentSelected(final int position)
41      {
42          pagerItems.get(position).content.activate();
43      }
44
45      public static class MainContentPagerItem
46      {
47          private final Integer icon;
48          private final Integer title;
49          private final MainContentFragmentContract content;
50
51          public MainContentPagerItem(
                    final Integer icon,
                    final Integer title,
                    final MainContentFragmentContract contentFragment)
```

```
            {
52              this.icon = icon;
53              this.title = title;
54              this.content = contentFragment;
55          }
56      }
57  }
```

Programski kôd 27: Druga verzija refaktorirane klase MCPA

U gornjem kôdu zelenom bojom označene su dvije najvažnije promjene: ova klasa sada sadržava listu objekata prethodno definiranog sučelja za obavijesti te se fragmenti više ne instanciraju određenim rasporedom unutar *switch* selekcije, već dinamički iz prihvaćene liste.

Za korištenje spomenutog sučelja netko ga treba implementirati. U tu je svrhu stvorena klasa `MainContentFragmentGenericProvider` (MCFGP) kojoj je svrha pružiti generičko sučelje za dinamičko stvaranje fragmenata po potrebi. Njezin kôd je ispod.

```
1   public class MainContentFragmentGenericProvider<T extends Fragment>
        implements MainContentFragmentContract
2   {
3       private Class<T> fragmentClass;
4       private T fragmentInstance = null;
5
6       public MainContentFragmentGenericProvider(final Class<T> fragmentClass)
7       {
8           this.fragmentClass = fragmentClass;
9       }
10
11      @Override
12      public T createFragment() throws IllegalAccessException, InstantiationException
13      {
14          fragmentInstance = fragmentClass.newInstance();
15          return fragmentInstance;
16      }
17
18      @Override
19      public void activate()
20      {
21          if (fragmentInstance instanceof RequestRefreshListener)
22          {
23              ((RequestRefreshListener) fragmentInstance).onRefreshRequest();
24          }
25      }
26  }
```

Programski kôd 28: Kôd klase MCFGP

Ovako definirana klasa koja prihvaća generički tip može biti korištena u raznim situacijama, a njezina ponovna iskoristivost na vrlo je visokoj razini. Metoda *createFragment* na liniji 12 čini uzorak dizajna Factory Method (Gamma et al., 1994) jer se apstrahira stvaranje objekta fragmenta i od klijenta se sakriva konkretna klasa fragmenta. U liniji 21 provjerava se

treba li aktivirati fragment. Naime, samo dva spomenuta fragmenta koji čine problem jer pristupaju mreži po aktivaciji trebaju moći biti osvježeni. Zbog tog su razloga naslijedili već postojeće sučelje iz projekta RequestRefreshListener i nadjačali njegovu metodu *onRefreshRequest* radi reagiranja na događaj aktivacije. U ContentActivityju novom dodanom metodom *getContentPagerItems* unose se učahureni podatci o prikazu. Ti podatci čine „izvor točnosti“ za fragment i pripadnu kartica u navigacijskoj traci.

```
1  private List<MainContentPagerAdapter.MainContentPagerItem> getContentPagerItems() {
2      List<MainContentPagerAdapter.MainContentPagerItem> items = new ArrayList<>();
3      items.add(new MainContentPagerAdapter.MainContentPagerItem(
4              R.drawable.ic_account_card_details,
5              R.string.menu_family_card_item,
6              new MainContentFragmentGenericProvider<>(FamilyCardFragment.class)
7      ));
8      items.add(new MainContentPagerAdapter.MainContentPagerItem(
9              R.drawable.ic_city,
10             R.string.menu_partners_item,
11             new MainContentFragmentGenericProvider<>(PartnersFragment.class)
12     ));
13     items.add(new MainContentPagerAdapter.MainContentPagerItem(
14             R.drawable.ic_newspaper,
15             R.string.menu_news_item,
16             new MainContentFragmentGenericProvider<>(NewsFragment.class)
17     ));
18     items.add(new MainContentPagerAdapter.MainContentPagerItem(
19             R.drawable.ic_home_account,
20             R.string.menu_family_profile_item,
21             new MainContentFragmentGenericProvider<>(FamilyProfileFragment.class)
22     ));
23     return items;
24 }
```

Programski kôd 29: Unos učahurenih podataka o fragmentima i karticama u aplikaciju

Gornji kôd najbolje je ogledalo izvršenog refaktoriranja: jedinstven izvor fragmenata i resursa koji idu uz njih. Ta napunjena lista vraća se u konačnu verziju metode *setupViewPager* prikazanu ispod.

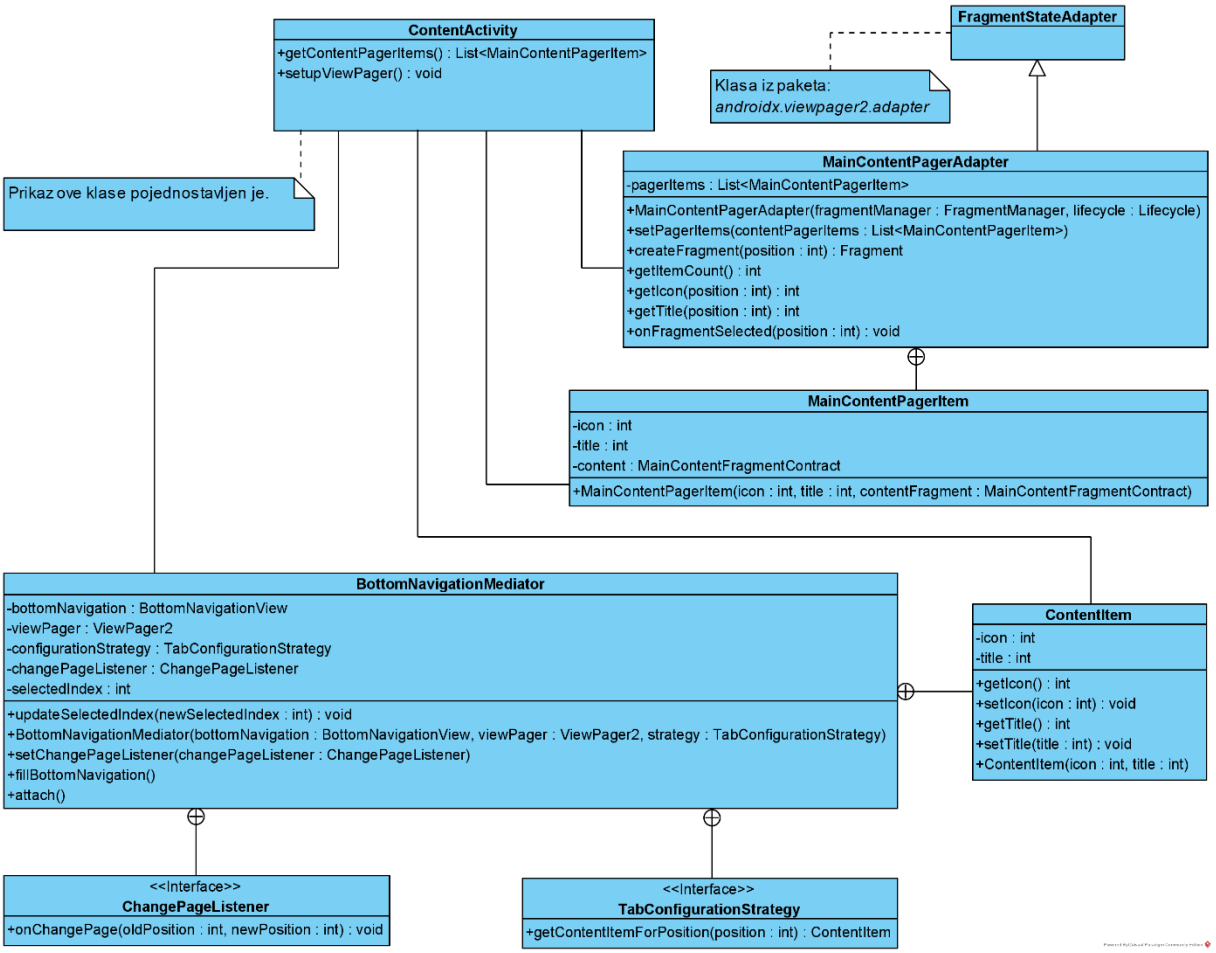
```
1  private void setupViewPager() {
2    adapter = new MainContentPagerAdapter(getSupportFragmentManager(), getLifecycle());
3    adapter.setPagerItems(getContentPagerItems());
4
5    vpMainContent.setAdapter(adapter);
6
7    BottomNavigationMediator bnm = new BottomNavigationMediator(btmContentNavigation,
       vpMainContent, position ->
         new BottomNavigationMediator.ContentItem(
           adapter.getIcon(position),
           adapter.getTitle(position)
8        )
9    );
```

```
10    bnm.setChangePageListener((oldPosition, newPosition) -> {
11      currentItem = newPosition;
12      adapter.onFragmentSelected(newPosition);
13      storeCurrentPage();
14    });
15    bnm.attach();
16
17    vpMainContent.setOffscreenPageLimit(3);
18  }
```

Programski kôd 30: Četvrta verzija metode *setupViewPager*

Ova konačna verzija u 12. liniji nad objektom adaptera izvršava metodu onFragmentSelected kojom se fragment aktivira preko već opisanog sučelja (Programski kôd 27, linija 40). Ovime je refaktoriranje dovršeno.

Slika 20 prikazuje opisan postavljen sustav za prikaz fragmenata.



Slika 20: UML dijagram klasa novog dizajna prikaza fragmenata

Na gornjem dijagramu je pojednostavljen prikaz klase `ContentActivity` izuzimanjem svih nebitnih dijelova te klase. Tablica 4 prikazuje usporedbu potrebnih koraka za izvršenje zadatka dodavanja novog fragmenta u aplikaciju. Tablica vrijedi i za postupak dodavanja fragmenta za plaćanje.

Tablica 4: Usporedba koraka dodavanja fragmenta prije i nakon arhitekturnog redizajna

Prije redizajna	Nakon redizajna
Dodati novu klasu fragmenta u projekt.	
Implementirati funkcionalnost.	
Definirati novi slučaj u selekciji tipa *switch* u klasi `ContentActivity`.	Dodati novi objekt u listu stavki glavnog ekrana u metodi *getContentPagerItems* klase `ContentActivity`.
Definirati novi element u datoteci *navigation.xml*.	Stavci u listi pridružiti resurs ikone i naslova.
Ručno promijeniti indekse u selekciji switch.	
Ažurirati fragmente koji se osvježavaju da provjeravaju nove vrijednosti iz memorije.	
***Koristi se zastarjeli element* `ViewPager`.**	

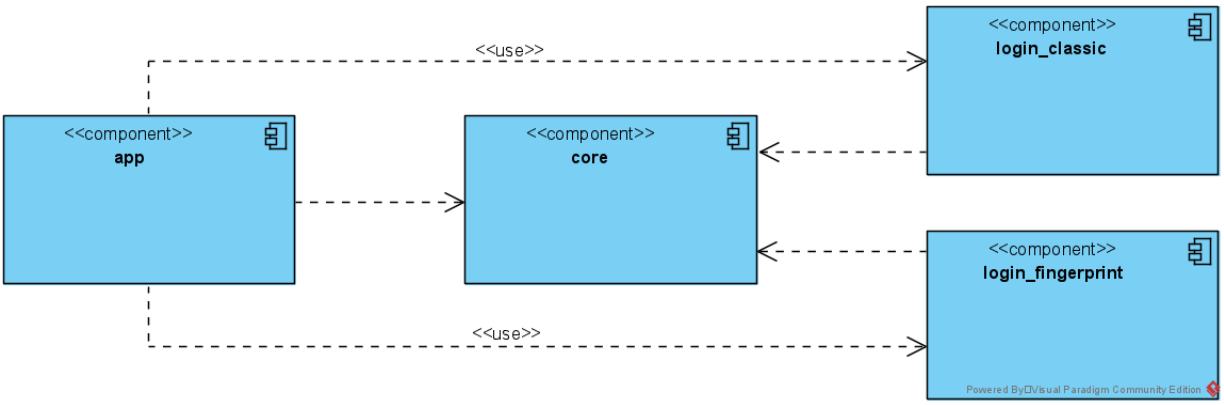
U ovom je poglavlju dan uvid u konkretan postupak arhitekturnog redizajna dijela glavnog prikaza u mobilnoj aplikaciji za Android. Olakšano je buduće održavanje aplikacijom na način da dodavanja ili izmjene fragmenata ili kartica treba izvršiti na samo jednom mjestu, a fragmenti i kartice sada su toliko usko vezani učahurivanjem u klase `MainContentPagerItem` i `ContentItem` da je nemoguće prikazivati kartice za fragmente koji nisu vidljivi i obratno. Upravo je to najveća prednost izvršenog redizajna.

U sljedećem poglavlju perspektiva će se podići na arhitekturne granice uvođenjem novog modula u projekt odabrane aplikacije.

6.2. Uvođenje modularnosti za prijavu korisnika

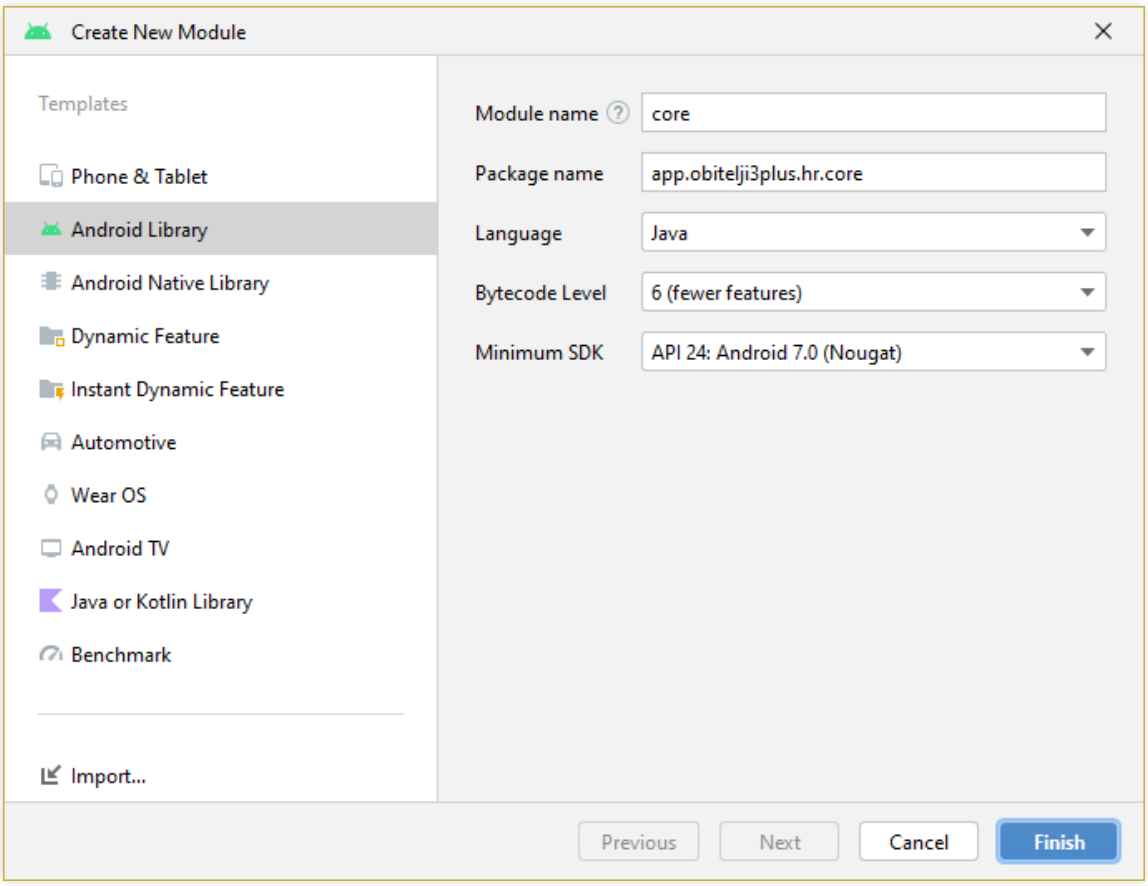
Dok je u prethodnom poglavlju opisan redizajn kôda prikaza fragmenata, u ovome poglavlju redizajnira se arhitektura cijelog projekta uvođenjem modularnosti.

O modularnosti je već bilo riječi u poglavlju 4.2.1. Riječ je o jasno razgraničenim komponentama sustava koje nezavisno učahuruju jedan dio implementacije poslovne logike. U ovom poglavlju uvest će se modul za prijavu, čime će se omogućiti aplikaciji da sadržava više različitih načina prijave. Slika 21 prikazuje kako će se povezati moduli radi ostvarivanja ovog proširenja.



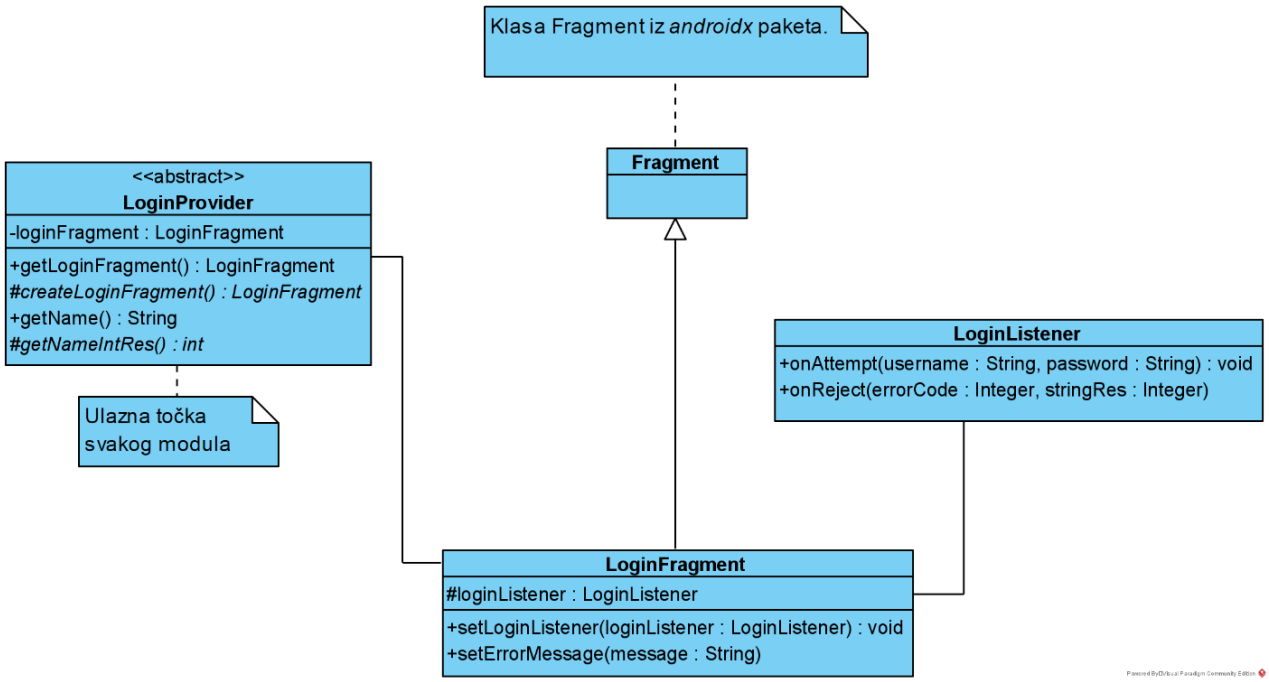
Slika 21: UML dijagram komponenti

Za početak, potrebno je stvoriti modul „core“ koji će sadržavati sva nužna sučelja i zajedničke elemente svim dijelovima projekta. Slika 22 prikazuje stvaranje tog modula.



Slika 22: Prozor za stvaranje novog modula „core“

Budući da glavna aplikacija treba ovisiti o „core“ modulu, u datoteku *build.gradle* dodaje se sljedeća linija kôda: `implementation project(path: ':core')`. To isto radit će se na odgovarajući način na svim ostalim modulima radi ostvarenja svih zamišljenih ovisnosti. Ovaj „core“ modul iznimno je važan jer će definirati prihvatljiva sučelja za sve module za prijavu. Slika 23 prikazuje dijagram klasa ovoga modula. Te klase tvore sustav kojemu se svaki modul za prijavu mora prilagoditi.



Slika 23: UML dijagram klasa novog „core“ modula

Slijede programski kôdovi prikazanih klasa s gornjeg dijagrama. Klasa `LoginProvider` predstavlja ulaznu točku u svaki modul.

```
1  public abstract class LoginProvider
2  {
3      private LoginFragment loginFragment = null;
4
5      public LoginFragment getLoginFragment()
6      {
7          if (loginFragment == null)
8          {
9              loginFragment = createLoginFragment();
10         }
11         return loginFragment;
12     }
13
14     protected abstract LoginFragment createLoginFragment();
15
16     public String getName()
17     {
18         assert loginFragment.getContext() != null;
19         return loginFragment.getContext().getString(getNameIntRes());
20     }
21
22     protected abstract int getNameIntRes();
23 }
```

Programski kôd 31: Klasa `LoginProvider` u modulu „core“

Klasa `LoginProvider` spoj je uzoraka dizajna Template Method zbog pozivanja apstraktnih metoda *createLoginFragment* i *getNameIntRes* i Factory Method zbog apstrahiranja stvaranja objekta fragmenta (Gamma et al., 1994). Način na koji učahuruje objekt fragmenta korištenjem svih triju razina pristupa omogućava ovoj klasi da u potpunosti sakrije implementacijske detalje fragmenta, a da održava objekt fragmenta na način poznat kao lijeno učitavanje (engl. *lazy loading*). Time je osiguran ispravan životni ciklus stvorenog fragmenta kojemu se potom po potrebi pristupa gdje je potrebno.

Svaki fragment koji `LoginProvider` sadržava mora naslijediti apstraktnu klasu `LoginFragment` koja nasljeđuje Androidovu klasu `Fragment`. Kôd te apstraktne klase naveden je u nastavku.

```
1  public abstract class LoginFragment extends Fragment
2  {
3      protected LoginListener loginListener;
4
5      public void setLoginListener(final LoginListener loginListener)
6      {
7          this.loginListener = loginListener;
8      }
9  }
```

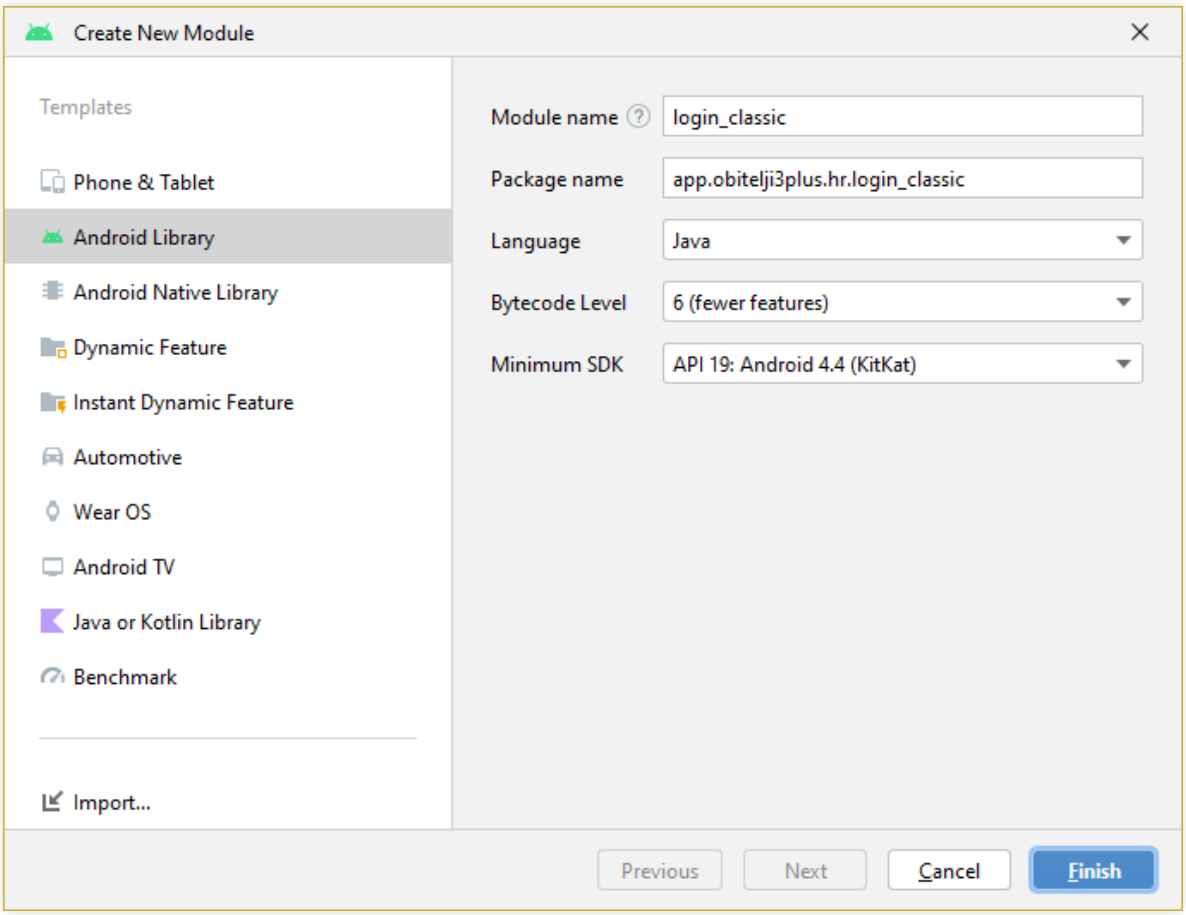
Programski kôd 32: Klasa `LoginFragment` u modulu „core“

Klasa `LoginFragment` predstavlja svaki fragment koji omogućava prijavu. Ono što čini fragment za prijavu jest vanjska implementacija slušača događaja koja definira upravljanje događajem uspješnog ostvarenja pokušaja prijave te neuspješnog pokušaja. Upravo je taj slušač posljednji dio prethodno definiranog dijagrama, a on čini uzorak dizajna Observer kod klase `LoginFragment` (Gamma et al., 1994).

```
1  public interface LoginListener
2  {
3      void onAttempt(String username, String password);
4      void onReject(Integer errorCode, Integer stringRes);
5  }
```

Programski kôd 33: Klasa `LoginListener` u modulu „core“

S ovim posljednjom definicijom kôda u „core“ modulu postavljen je temelj za module za obavljanje prijave. Slijedi posao prebacivanja cijele logike prijave u novi modul. Slika 24 prikazuje stvaranje novog modula s nazivom „login_classic“.



Slika 24: Prozor za stvaranje novog modula „login_classic“

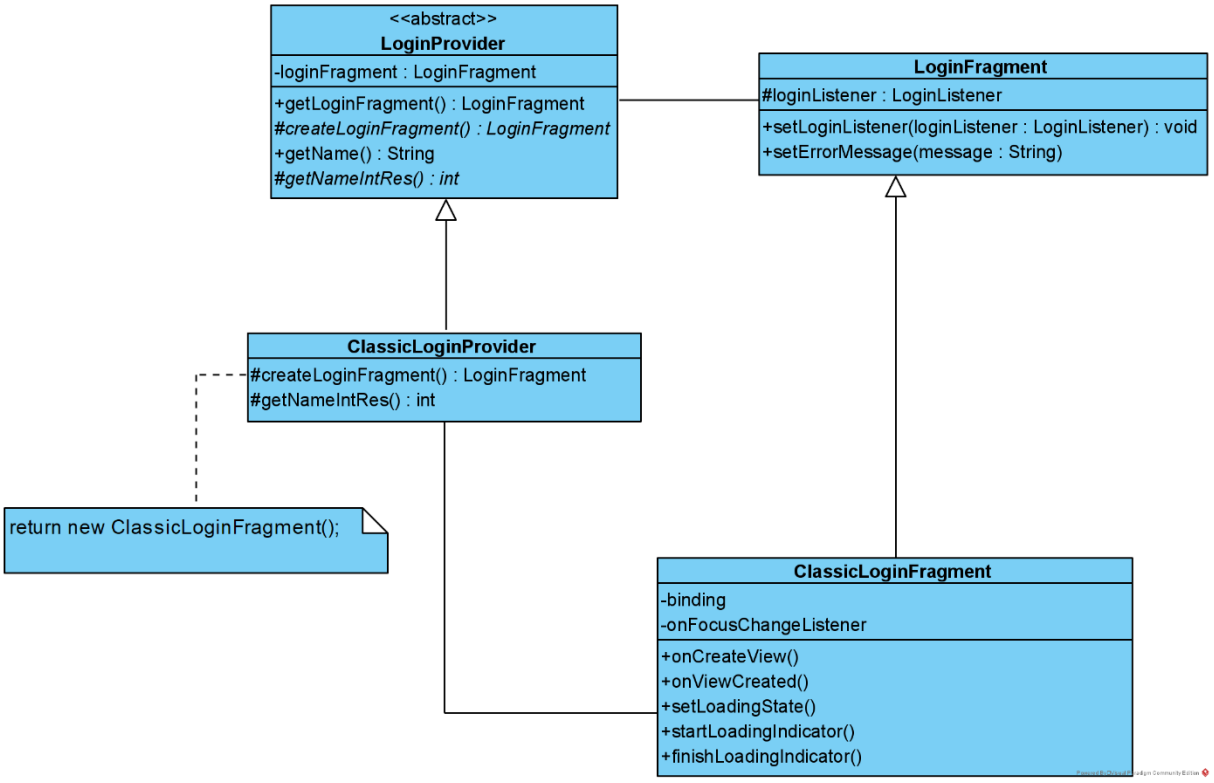
Ovisnosti na novi modul uspostavljene su kao na UML dijagramu komponenti, iznad (Slika 21). Kako bi se iskoristio dosadašnji dizajn, sve XML datoteke resursa (boje, strukture sadržaja itd.) prebačene su iz „app“ modula u „core“. Time je osigurano da bez obzira na to koliko se aplikacija skalira novim modulima i općenito funkcionalnostima, uvijek postoji samo jedan izvor korištenih boja, tekstualnih resursa, animacija itd. u modulu „core“ koji je svugdje dostupan. Nadalje, izdvojen je i dio programske logike iz klasa `StringUtils`, `UI_Utils` iz paketa „utils“ u zasebne klase modula „core“ te su izdvojene klase `SharedPrefManager` i `LocalConstants` u modul „core“. Riječ je o programskoj logici nužnoj za rad prijave, a opet dovoljno odvojenoj od aplikacije da se može izdvojiti u modul kao zaseban, univerzalni dio kôda projekta. Odvojeno je i sučelje za osvježavanje podataka `RequestRefreshListener` koje je već bilo spomenuto u kontekstu fragmenata koji dohvaćaju podatke s mrežnih resursa. Još jedno sučelje koje je prebačeno u „core“ modul jest iznimno važno sučelje (za razmatrani kontekst) s nazivom SignInContract.View (riječ je o ugniježđenom sučelju).

```
1  interface View {
2      void startLoadingIndicator();
3      void finishLoadingIndicator();
4  }
```

Programski kôd 34: Kôd sučelja SignInContract.View

Ovo je sučelje umetnuto u modul „core“ na način da ga klasa `LoginFragment` implementira. To je nužno kako bi prezenter aktivnosti za prijavu mogao komunicirati s fragmentom jednako kao što je prije komunicirao sa samom aktivnošću.

Nakon dovršetka modula za klasičnu prijavu, njegov dijagram klasa izgleda ovako:



Slika 25: UML dijagram klasa „login_classic“ modula

Dvije klase koje se ističu na gornjem dijagramu klasa jesu sljedeće nove klase: `ClassicLoginProvider` i `ClassicLoginFragment`. Kôd za te dvije klase dan je u nastavku. Prvo se prikazuje kôd za pružatelja usluge prijave.

```java
public class ClassicLoginProvider extends LoginProvider
{
    @Override
    protected LoginFragment createLoginFragment()
    {
        return new ClassicLoginFragment();
    }

    @Override
    protected int getNameIntRes()
    {
        return R.string.oib_login;
    }
}
```

Programski kôd 35: Implementirani pružatelj klasične prijave

Sljedeći je kôd implementacija fragmenta za prijavu.

```java
public class ClassicLoginFragment extends LoginFragment
{
  private FragmentClassicLoginBinding binding;

  private final View.OnFocusChangeListener onFocusChangeListener = new View.OnFocusChangeListener()
  {
    @Override
    public void onFocusChange(View view, boolean b)
    {
      if (b)
      {
        binding.tvErrorMessage.setText("");
        binding.tvErrorMessage.setVisibility(View.GONE);
      }
    }
  };

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
  {
    return inflater.inflate(R.layout.fragment_classic_login, container, false);
  }

  @Override
  public void onViewCreated(@NonNull final View view, @Nullable final Bundle savedInstanceState)
  {
    binding = FragmentClassicLoginBinding.bind(view);
    binding.txtUsername.setOnFocusChangeListener(onFocusChangeListener);
    binding.txtPassword.setOnFocusChangeListener(onFocusChangeListener);
    binding.cbZapamtiMe.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener()
    {
      @Override
      public void onCheckedChanged(final CompoundButton buttonView, final boolean isChecked)
      {
        SharedPreferences.Editor editor = new SharedPrefManager(getContext()).getEditor();
        editor.putBoolean(LocalConstants.SharedPref.KEY_REMEMBER_ME, isChecked);
      }
    });
    binding.btnSignIn.setOnClickListener(new View.OnClickListener()
    {
      @Override
```

```
42        public void onClick(final View v)
43        {
44          loginListener.onAttempt(
                  binding.txtUsername.getText().toString(),
                  binding.txtPassword.getText().toString()
          );
45        }
46      });
47    }
48
49    public void setErrorMessage(final String message)
50    {
51      binding.tvErrorMessage.post(new Runnable()
52      {
53        @Override
54        public void run()
55        {
56          binding.tvErrorMessage.setText(message);
57          binding.tvErrorMessage.setVisibility(View.VISIBLE);
58        }
59      });
60    }
61
62    @Override
63    public void startLoadingIndicator()
64    {
65      ProgressBarManager.hideProgressBar(
            binding.llSignInContent, binding.rlSignInFormContainer, false
        );
66    }
67
68    @Override
69    public void finishLoadingIndicator()
70    {
71      ProgressBarManager.showProgressBar(
            binding.llSignInContent, binding.rlSignInFormContainer, false
        );
72    }
73  }
```

Programski kôd 36: Implementirani fragment za klasičnu prijavu

Nakon prikazane implementacije klase `ClassicLoginFragment` preostalo je još samo kôd u modulu „app“ prilagoditi prikazu različitih načina prijave. U klasi `SignInActivity` mnogo je izmjena. Glavna promjena jest dodavanje liste pružatelja usluge prijave te postavljanje konkretnog objekta usluge prijave u tu listu. Jedan kursor uvijek pokazuje na trenutačno odabran pružatelj usluge. Nije prikazano svih 170 linija kôda cijele klase, već su prikazani samo najvažniji dijelovi kôda koje u kontekstu implementacije modularnosti valja istaknuti.

```
1  // ...
2
3  private int currentProviderIndex;
4  private List<LoginProvider> loginProviders;
5
6  // ...
7
```

```
 8  @Override
 9  protected void onCreate(@Nullable Bundle savedInstanceState) {
10    loginProviders = new ArrayList<>(
11        Arrays.asList(
12            new ClassicLoginProvider()
13        )
14    );
15    SharedPrefManager spm = new SharedPrefManager(getApplicationContext());
16    currentProviderIndex = spm.readSharedPref().getInt(KEY_LAST_LOGIN, 0);
17    super.onCreate(savedInstanceState);
18  }
19
20  // ...
21
22  @Override
23  protected void initializeLayout() {
24    LoginFragment initialLoginFragment = getFragmentForIndex(currentProviderIndex);
25    FragmentManager fm = getSupportFragmentManager();
26    fm.beginTransaction().replace(R.id.fcv_login_fragment, initialLoginFragment).commitNow();
27    this.presenter = new SignInPresenter(initialLoginFragment, this);
28
29    this.presenter.setOnLoginSubmitted(new SignInContract.Presenter.OnLoginSubmittedListener()
30    {
31      @Override
32      public void handleSuccessfulLogin()
33      {
34        openNextView();
35      }
36
37      @Override
38      public void handleUnsuccessfulLogin()
39      {
40        Toast.makeText(
                SignInActivity.this,
                R.string.error_offline_login_no_credentials,
                Toast.LENGTH_SHORT
41        ).show();
42      }
43    });
44    // ...
45  }
46
47  // ...
48
49  private LoginFragment getFragmentForIndex(final int currentProviderIndex)
50  {
51    return loginProviders.get(currentProviderIndex).getLoginFragment();
52  }
53
54  // ...
55
56  @Override
57  protected void implementActivityLogic(@Nullable Bundle savedInstanceState) {
58    setLoginListener();
59  }
```

```
60
61  // ...
62
63  private void setLoginListener()
64  {
65    LoginProvider provider = loginProviders.get(currentProviderIndex);
66    provider.getLoginFragment().setLoginListener(new LoginListener()
67    {
68      @Override
69      public void onAttempt(final String username, final String password)
70      {
71        presenter.signIn(username, password);
72      }
73
74      @Override
75      public void onReject(final Integer errorCode, final Integer stringRes)
76      {
77        Toast.makeText(
                SignInActivity.this,
                getString(stringRes),
                Toast.LENGTH_SHORT
78        ).show();
79      }
80    });
81  }
82
83  // ...
84
85  @Override
86  public void onErrorReceived(ApiErrorHolder error) {
87    // ...
88    loginProviders.get(currentProviderIndex)
                    .getLoginFragment()
                    .setErrorMessage(getString(id));
89  }
```

Programski kôd 37: Istaknute promjene radi korištenja modularnosti u klasi SignInActivity

Na liniji 3 nalazi se kursor *currentProviderIndex* za trenutačno odabranu vrstu prijave, a na liniji 4 lista svih pružatelja prijave na koju se kursor odnosi. Na liniji 12 srž je ovakvog modularnog prikaza: polimorfizmom ostvarena lista u koju se pune objekti predstavnici svojih modula. Budući da je u svim kôdovima iznad implementiran samo jedan modul, jedino je njegov objekt klase `ClassicLoginProvider` dodan u listu. Na liniji 16 kursor se postavlja na vrijednost prethodno zapamćenog načina prijave. U trenutačnoj situaciji to će uvijek biti početni, klasični modul. Na liniji 24 pozivom metode prelazi se na izvršenje linije 51. Te dvije linije osiguravaju stvaranje objekta prvog fragmenta za prikaz. Na liniji 26 taj objekt fragmenta postavlja se u element sučelja `FragmentContainerView` i prikazuje se korisniku. Na liniji 29 postavlja se osluškivač događaja uspješne prijave, a na liniji 69 postavlja se osluškivač pokušaja prijave direktno iz fragmenta (koji god on bio).

Tablica 5 prikazuje ostvarene koristi ovog refaktoriranja. Što se tiče stupca s koracima prije redizajna, treba napomenuti da se može ići s mnoštvom loših koraka. Upisani su optimalni koraci u situaciji kakva je bila prije redizajna.

Tablica 5: Usporedba koraka dodavanja načina prijave prije i nakon arhitekturnog redizajna

Prije redizajna	Nakon redizajna
Stvoriti novu aktivnost za prijavu otiskom prsta.	Stvoriti novi modul *login_fingerprint*. Definirati novi fragment unutar njega.
Izvršiti prebacivanje iz osnovne *SignIn* aktivnosti u drugu, treću itd. po potrebi.	Izraditi novi fragment za prijavu otiskom prsta unutar novog modula.
Osigurati ispravan životni ciklus nove aktivnosti za upravljanje prijavom.	U novom modulu postaviti `FingerprintLoginProvider` klasu i njome proširiti klasu `LoginProvider`.
Omogućiti komunikaciju među aktivnostima, kako bi nova aktivnost mogla komunicirati s originalnom `SignInActivity`. U suprotnom je nužno ponavljati kôd koji izvršava prijavu na svakoj sljedećoj aktivnosti za prijavu.	U listu unutar klase `SignInActivity` dodati objekt klase `FingerprintLoginProvider`.
Popuniti neku listu nazivima mogućnosti ovih dviju, ali i teoretski svih ostalih načina prijave. Nazivi su nužni kako bi korisnik znao odabrati način prijave.	
Nakon prijave iz druge aktivnosti osigurati gašenje prve.	

Ovo poglavlje prikazalo je izvršenje redizajna arhitekture cijelog projekta mobilne aplikacije za Android, kako bi projekt bio modularan i više ponovno iskoristiv. Sljedeće poglavlje opisuje zaključke oba prikazana procesa redizajna.

6.3. Osvrt na proces redizajniranja arhitekture

U prethodnim dvama poglavljima detaljno je prikazan proces refaktoriranja postojećeg kôda u svrhu dodavanja nove funkcionalnosti i poboljšanja održivosti aplikacije. Ovo poglavlje obuhvaća osvrt na izvršeno refaktoriranje.

U kôdu je refaktoriranje izvršeno u 23 promjene (gitova *commita*). Ako se uključi i prilagodba projekta novim verzijama ovisnosti i *androidx* bibliotekama, riječ je o 34 promjene, 3933 dodanih linija te 3381 izbrisane linije.

Izvršeno refaktoriranje kôda i redizajniranje arhitekture rezultiralo je mnogo čišćim kôdom u obuhvaćenom području. Refaktoriranje se u oba slučaja oslonilo na korištenje liste različitih objekata. U dijelu s refaktoriranim prikazom fragmenata ta lista koristi generičko programiranje kako bi se prilagodila svakom fragmentu za koji je odlučeno da treba biti u glavnom ekranu. U drugome dijelu, s redizajniranom cijelom projektnom strukturom u koju se uvode moduli, temelj je u polimorfizmu liste objekata različitih klasa. Time je rješenje prilagođeno bilo kojem modulu koji se drži ugovora pružatelja usluge prijave.

Nedostatak je izvršenog redizajna to što ni u kojem dijelu nije stavljen naglasak na testiranje i ispravan tok podataka, odnosno upravljanja. Razlog je to što testiranje nije bilo ostvarivo na ovim promjenama. Kod refaktoriranja prikaza fragmenata, testiranje se nije moglo provesti nad objektom ViewPager2 jer su promjene vezane uz vizualni prikaz. Bilo je važnije osigurati da vizualni prikaz aplikacije ostane dosljedan originalnu nego postaviti testove koji bi pokušali to dokazati programskim putem. Nadalje, testiranje novih modula nije bilo smisleno jer uvođenje novih modula i eventualna reorganizacija smještaja datoteka unutar njih nije nešto što se može jedinično testirati. Krajnja je promjena u toku podataka, a budući da se koristi isti objekt Presentera, poslovna logika nije dirana i stoga je nije bilo smisleno testirati.

7. Zaključak

U prethodnom poglavlju bilo je riječi o implementaciji redizajna, što je u kontekstu rada bio četvrti i posljednji dio rada. Ovo poglavlje donosi zaključak kojim se rad okončava.

U prvome dijelu rada dan je pregled literature koja se odnosi na refaktoriranje i arhitekturu, a ovdje će se istaknuti najvažniji zaključci vezani za iznesene poglede autora.

Autor Martin Fowler preporučuje izvođenje refaktoriranja u malenim koracima s testovima koji će osigurati nepromijenjeno funkcioniranje programa. Također, Fowler naglašava važnost dobrog imenovanja varijabli i uklanjanje privremenih varijabli kako bi se poboljšala komunikacija koda. Preporučuje i refaktoriranje tijekom pregledavanja koda kako bi se dobili novi uvidi i ideje. U ovom se kontekstu spominju UML dijagrami i programiranje u paru.

Autor Kent Beck, tvorac metodologije ekstremnog programiranja, naglašava važnost refaktoriranja kako bi se izbjegla situacija u kojoj danas obavljen posao neće omogućiti obavljanje sutrašnjeg posla. Autor također spominje i važnost komunikacije između članova tima tijekom procesa refaktoriranja, što može biti ostvareno korištenjem zajedničkog koda i redovitog timskog sastanka. Navodi i neke primjere loših praksi refaktoriranja, poput prekasnog odlaganja refaktoriranja, što može dovesti do nakupljanja duga u obliku tehničkog duga te time otežati refaktoriranje u budućnosti. Također upozorava da prevelika opsjednutost refaktoriranjem može dovesti do gubitka fokusa na izvornoj funkcionalnosti softvera te prevelikog trošenja vremena na nevažne detalje. Autor ističe važnost refaktoriranja tijekom pregledavanja kôda, komunikacije između članova tima te upozorava na neke loše prakse refaktoriranja, poput odlaganja refaktoriranja i prevelike opsjednutosti refaktoriranjem. Refaktoriranje podrazumijeva promjenu interne strukture softvera kako bi se olakšalo razumijevanje i preinačivanje bez vidljive promjene u ponašanju.

Nakon pregleda literature, u drugom je dijelu rada dan uvod u odabranu aplikaciju.

Pri refaktoriranju kôda ponekad je potrebno pregledati više razina kôda radi poboljšavanja kvalitete i održivosti. Refaktoriranje se odnosi na proces restrukturiranja postojećeg kôda bez mijenjanja njegova ponašanja kako bi bio čitljiviji, održiviji i skalabilniji, a to se postiže razbijanjem kôda na manje funkcionalne dijelove i poboljšanjem njihove arhitekture. Dobro odrađeno refaktoriranje omogućava daljnji rast i razvoj projekta. Time i programer dokazuje svoju kvalitetu zato što može i računalu prenijeti naredbe, ali i razumjeti sustav te ga pojednostavniti inženjerima koji će na njemu raditi u budućnosti.

Popis literature

Al Dallal, J., & Abdin, A. (2018). Empirical Evaluation of the Impact of Object-Oriented Code Refactoring on Quality Attributes: A Systematic Literature Review. *IEEE Transactions on Software Engineering*, *44*(1), 44–69. https://doi.org/10.1109/TSE.2017.2658573

Brahler, S. (2010). Analysis of the android architecture. *Karlsruhe Institute for Technology*, *7*(8).

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture Volume 1: A System of Patterns* (Volume 1 edition). Wiley.

Clean Coder Blog. (13.8.2012.). [Blog]. The Clean Architecture. Preuzeto 14.2.2023. s http://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Confreaks (Director). (13.8.2012.). *Ruby Midwest 2011 - Keynote: Architecture the Lost Years by Robert Martin*. Preuzeto 4.3.2023. s https://www.youtube.com/watch?v=WpkDN78P884

Data layer. (23.1.2023.). Android Developers. Preuzeto 12.2.2023. s https://developer.android.com/topic/architecture/data-layer

Demeyer, S., Ducasse, S., & Nierstrasz, O. (2009). *Object-Oriented Reengineering Patterns* (9/28/09 edition). Square Bracket Associates. Preuzeto 10.2.2023. s https://www.open.umn.edu/opentextbooks/textbooks/object-oriented-reengineering-patterns

Deprecations | Android Enterprise. (4.11.2022.). [Dokumentacija]. Google Developers. Preuzeto 9.2.2023. s https://developers.google.com/android/work/deprecations

Fikri, A., Presekal, A., Harwahyu, R., & Sari, R. F. (2018). Performance Comparison of Dalvik and ART on Different Android-Based Mobile Devices. *2018 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, 439–442. https://doi.org/10.1109/ISRITI.2018.8864290

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (2019). *Refactoring: Improving the Design of Existing Code. Addison-Wesley.* (2nd ed.). Pearson Education.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Booch, G. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (1st edition). Addison-Wesley Professional.

Get data from the internet. (16.7.2022.). Android Developers. Preuzeto 17.2.2023. s https://developer.android.com/codelabs/basic-android-kotlin-training-getting-data-internet

Guide to app architecture. (15.11.2022.). [Dokumentacija]. Android Developers. Preuzeto 7.2.2023. s https://developer.android.com/topic/architecture

Handle configuration changes. (4.11.2022.). Android Developers. Preuzeto 14.2.2023. s https://developer.android.com/guide/topics/resources/runtime-changes

ISO 25010. (2022). Preuzeto 14.2.2023. s https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

Jonathan, L. (2015). *Dalvik and ART*.

Malavolta, I., Verdecchia, R., Filipovic, B., Bruntink, M., & Lago, P. (2018). How Maintainability

Issues of Android Apps Evolve. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 334–344. https://doi.org/10.1109/ICSME.2018.00042

Martin Fowler. (26.8.2015.). *bliki: PresentationDomainDataLayering*. Martinfowler.Com. Preuzeto 12.2.2023. s https://martinfowler.com/bliki/PresentationDomainDataLayering.html

Martin, R. C. (2000). Design principles and design patterns. *Object Mentor*, *1*(34), 597.

Martin, R. C., Grenning, J., Brown, S., Henney, K., & Gorman, J. (2018). *Clean architecture: A craftsman's guide to software structure and design.* Prentice Hall.

Morasca, S. (2009). A probability-based approach for measuring external attributes of software artifacts. *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 44–55. https://doi.org/10.1109/ESEM.2009.5316048

Now in Android App. (2022). [Kotlin]. Android. Preuzeto 13.2.2023. s https://github.com/android/nowinandroid/blob/abf2b6a7258b364edd25e0bde127eded4844d38d/docs/ModularizationLearningJourney.md

Processes and threads overview. (27.10.2021.). Android Developers. Preuzeto 13.2.2023. s https://developer.android.com/guide/components/processes-and-threads

Sanchez, D., Rojas, A. E., & Florez, H. (2022). Towards a Clean Architecture for Android Apps using Model Transformations. *IAENG International Journal of Computer Science*, *49*(1), 270–278.

Save data in a local database using Room. (7.2.2023.). Android Developers. Preuzeto 13.2.2023. s https://developer.android.com/training/data-storage/room

Silva, C. E. (2013). *Combining Static and Dynamic Analysis for the Reverse Engineering of Web Applications*. ACM.

SonarLint—IntelliJ IDEs Plugin | Marketplace. (23.1.2023). JetBrains Marketplace. Preuzeto 15.2.2023. s https://plugins.jetbrains.com/plugin/7973-sonarlint

SonarQube 9.9. (2022). [Documentation]. SonarQube Documentation. Preuzeto 15.2.2023. s https://docs.sonarqube.org/latest/

Steve McConnell. (2004). *Code Complete* (2nd edition). Microsoft Press.

Stroustrup, B. (2023, December 26). Quotes [Blog]. *Bjarne Stroustrup Quotes*. Preuzeto 6.2.2023. s https://www.stroustrup.com/quotes.html

Thomas, D., & Hunt, A. (2019). *The Pragmatic Programmer: Your Journey To Mastery, 20th Anniversary Edition* (2nd edition). Addison-Wesley Professional.

UI layer. (15.11.2022). [Dokumentacija]. Android Developers. Preuzeto 12.2.2023. s https://developer.android.com/topic/architecture/ui-layer

Wangberg, R. (2010). *A Literature Review on Code Smells and Refactoring* [Master thesis]. https://www.duo.uio.no/handle/10852/8739

Wharton, J. (2020). *Butter Knife* [Java]. https://github.com/JakeWharton/butterknife (originalno objavljeno 2013.)

Popis slika

Slika 1: Skica sustava s jasno definiranom granicom (Prema: Confreaks, 2012) 12
Slika 2: Čista arhitektura softverskog proizvoda prema Robertu C. Martinu (Izvor: *Clean Coder Blog*, 2012; Martin et al., 2018) 13
Slika 3: Prikaz odnosa između programske logike i prikaza 16
Slika 4: Dijagram tipične troslojne aplikacijske arhitekture (Izvor: *Guide to App Architecture*, 2022) 18
Slika 5: Alternativni prikaz troslojne arhitekture (Prema: Martin Fowler, 2015) 19
Slika 6: Uloga sloja korisničkog sučelja u arhitekturi aplikacije (Izvor: *Guide to App Architecture*, 2022) 20
Slika 7: Dijagram koji opisuje kako jednosmjerni protok podataka funkcionira u arhitekturi aplikacije (Izvor: *UI Layer*, 2022) 22
Slika 8: Graf ovisnosti u slučaju kombiniranja više repozitorija (Izvor: *Guide to App Architecture*, 2022) 23
Slika 9: Uloga sloja podataka u arhitekturi aplikacije (Izvor: *Guide to App Architecture*, 2022) 24
Slika 10: Ilustracija preporučene arhitekture aplikacije za sustav Android (izradio autor, prema: *Guide to App Architecture*, 2022; Martin et al., 2018) 26
Slika 11: Dijagram slučajeva korištenja postojeće aplikacije 30
Slika 12: Navigacijski graf aplikacije 30
Slika 13: Aktiviranje proširenja SonarLint 43
Slika 14: Izvođenje analize kôda alatom SonarLint 43
Slika 15: Prijavljeni problemi nakon SonarLintove analize 44
Slika 16: Povezivanje SonarLinta sa SonarQubeom 45
Slika 17: Prikaz SonarQubeovog rezultata analize u internetskom pregledniku. 45
Slika 18: Razlaganje pogreške u SonarQubeu 46
Slika 19: Prikazan izvještaj proširenja SonarLint u razvojnoj okolini Android Studio s jednim otvorenim kritičnim *smrdljivim* kôdom 47
Slika 20: UML dijagram klasa novog dizajna prikaza fragmenata 65
Slika 21: UML dijagram komponenti 67
Slika 22: Prozor za stvaranje novog modula „core“ 68
Slika 23: UML dijagram klasa novog „core“ modula 69
Slika 24: Prozor za stvaranje novog modula „login_classic“ 71
Slika 25: UML dijagram klasa „login_classic“ modula 72

Popis tablica

Tablica 1: Popis paketa klasa u analiziranoj aplikaciji 31
Tablica 2: Povezanost elemenata sloja korisničkog sučelja u aplikaciji 33
Tablica 3: Korištenja uzorka dizajna Singleton u aplikaciji 35
Tablica 4: Usporedba koraka dodavanja fragmenta prije i nakon arhitekturnog redizajna 66
Tablica 5: Usporedba koraka dodavanja načina prijave prije i nakon arhitekturnog redizajna 77

Popis programskih kôdova

Programski kôd 1: Odabir implementacije s obzirom na mrežnu povezanost 34
Programski kôd 2: Odabir navigacijske stavke 37
Programski kôd 3: Metoda za postavljanje fragmenata glavnoga prikaza 38
Programski kôd 4: Sadržaj datoteke *navigation.xml* 39
Programski kôd 5: Metoda *isFragmentActive* klase `NewsFragmentPresenter` 39
Programski kôd 6: Programska logika klase `MainActivity` 41
Programski kôd 7: Prepoznat smrdljiv kôd u ugniježđenim selekcijama 47
Programski kôd 8: Prepoznat smrdljiv kôd u ponavljajućem znakovnom nizu 48
Programski kôd 9: Prepoznat smrdljiv kôd u apstraktnoj klasi sa sučeljima 49
Programski kôd 10: Postavljanje ViewPagera u klasi `ContentActivity` 51
Programski kôd 11: Metoda *getInstance* u fragmentima 52
Programski kôd 12: Zastarjeli kôd klase MCPA 52
Programski kôd 13: Prva verzije refaktorirane klase MCPA 53
Programski kôd 14: Originalna struktura početnoga ekrana 54
Programski kôd 15: Poboljšana struktura početnoga ekrana 55
Programski kôd 16: Početni kôd klase `BottomNavigationMediator` 56
Programski kôd 17: Prva refaktorirana verzija metode *setupViewPager* 57
Programski kôd 18: Klasa za jedinstvenu definiciju sadržaja na glavnome zaslonu 58
Programski kôd 19: Sučelje za ostvarenje izmjenjivog algoritma definiranja kartica 58
Programski kôd 20: Punjenje navigacijskog izbornika s karticama fragmenata 58
Programski kôd 21: Druga refaktorirana verzija metode *setupViewPager* 59
Programski kôd 22: Pregled slušača događaja promjene fragmenta u klasi BNM 60
Programski kôd 23: Pozivanje metode updateSelectedIndex 60
Programski kôd 24: Metoda koja osigurava obavijest o promjeni stranice 60
Programski kôd 25: Treća refaktorirana verzija metode *setupViewPager* 61
Programski kôd 26: Sučelje čija će implementacija omogućavati aktiviranje fragmenata 61
Programski kôd 27: Druga verzija refaktorirane klase MCPA 63
Programski kôd 28: Kôd klase MCFGP 63
Programski kôd 29: Unos učahurenih podataka o fragmentima i karticama u aplikaciju 64
Programski kôd 30: Četvrta verzija metode *setupViewPager* 65
Programski kôd 31: Klasa `LoginProvider` u modulu „core“ 69
Programski kôd 32: Klasa `LoginFragment` u modulu „core“ 70
Programski kôd 33: Klasa `LoginListener` u modulu „core“ 70
Programski kôd 34: Kôd sučelja SignInContract.View 72
Programski kôd 35: Implementirani pružatelj klasične prijave 73
Programski kôd 36: Implementirani fragment za klasičnu prijavu 74
Programski kôd 37: Istaknute promjene radi korištenja modularnosti u klasi SignInActivity 76