Usporedba pristupa i obrazaca prilikom implementacije paralelizma na Android platformi

Takač, David

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Diplomski sveučilišni studij Računarstvo, izborni blok Programsko inženjerstvo

Usporedba pristupa i obrazaca prilikom implementacije paralelizma na Android platformi

Diplomski rad

David Takač

Osijek, 2021.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 26.08.2021.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime studenta:	David Takač
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1100R, 06.10.2019.
OIB studenta:	92780543146
Mentor:	Doc.dr.sc. Zdravko Krpić
Sumentor:	Doc. dr. sc. Bruno Zorić
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Krešimir Nenadić
Član Povjerenstva 1:	Doc. dr. sc. Bruno Zorić
Član Povjerenstva 2:	Dr. sc. Krešimir Romić
Naslov diplomskog rada:	Usporedba pristupa i obrazaca prilikom implementacije paralelizma na Android platformi
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U teorijskom dijelu rada opisati mehanizme paralelizma u razvoju programske podrške s posebnim naglaskom na Android platformu. Detaljnije prikazati ustaljene preporuke i obrasce koji se primjenjuju prilikom pisanja paralelizirane programske podrške kao i uz njih vezane biblioteke. U praktičnom dijelu rada potrebno je izraditi aplikaciju koja rabi odgovarajuće mehanizme i obrasce prikazane u teorijskom dijelu rada te ju prikladno testirati. Tema rezervirana za: David Takač Sumentor s FERIT-a: Bruno Zorić
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/bodaPostignuti rezultati u odnosu na složenost zadatka: 3 bod/bodaJasnoća pismenog izražavanja: 3 bod/bodaRazina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	26.08.2021.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 17.09.2021.

Ime i prezime studenta:	David Takač
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1100R, 06.10.2019.
Turnitin podudaranje [%]:	1

Ovom izjavom izjavljujem da je rad pod nazivom**: Usporedba pristupa i obrazaca prilikom implementacije paralelizma na Android platformi**

izrađen pod vodstvom mentora Doc.dr.sc. Zdravko Krpić

i sumentora Doc. dr. sc. Bruno Zorić

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija. Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

Sadržaj

1. Uvod 1

2. Razvoj paralelizirane programske podrške 2

2.1. Paralelizam 2

2.1.1. Novo normalno 2

2.2. Istodobnost 4

2.3. Višedretvenost 6

2.4. Asinkronost u Androidu 7

2.4.1. Zastajkivanje u Android aplikacijama 8

2.4.2. Glavna dretva 9

2.4.3. Asinkronost 10

2.5. Dretve u Javi. 11

2.5.1. Osnovni pristup. 12

2.5.2. Android dijeljenje poruka 14

2.5.3. HandlerThread i AsyncTask 16

2.5.4. ExecutorService i više od jedne pozadinske dretve 18

2.5.5. Česti problemi i anti-obrasci 19

2.5.6. Preporučeni obrazac korištenja 22

2.6. RxJava 25

2.6.1. Promotrivi tok 26

2.6.2. Raspoređivači, istodobnost i asinkronost 28

2.6.3. Paralelni promotrivi tokovi 31

2.6.4. Otkazivanje pretplate 34

2.6.5. Česti problemi i anti-obrasci 36

2.6.6. Preporučeni obrazac korištenja 39

2.7. Korutine u Kotlinu 41

2.7.1. Izravni stil i stil prosljeđivanja kontinuacija 41

2.7.2. Korutina u računarstvu 42

2.7.3. Pregled terminologije 44

2.7.4. Jednostavna korutina 45

2.7.5. Korutinski kontekst 45

2.7.6. Dispečeri i upravljanje asinkronošću 47

2.7.7. Istodobne korutine 49

2.7.8. Otkazivanje i strukturirana istodobnost. 50

2.7.9. Česti problemi i anti-obrasci 51

2.7.10. Preporučeni obrazac korištenja 54

2.8. Testiranje korutina 55

2.8.1. Jedinično testiranje u Androidu 56

2.8.2. Pokretanje korutina u testovima. 57

2.8.3. Testni korutinski opseg i dispečer 57
2.8.4. Testiranje zaustavnih funkcija. 58
2.8.5. Testiranje koda koji pokreće korutine 59
2.9. Usporedba dretvi, RxJave i korutina 61
3. Primjena mehanizama paralelizma u izradi Android aplikacije . . . 63
3.1. Opis problema i prikaz rješenja na visokoj razini. 63
3.2. Specifikacija zahtjeva 63
3.3. Korišteni alati i tehnologije. 67
3.4. Prikaz programskog rješenja 69
3.4.1. Dohvaćanje i spremanje podataka 69
3.4.2. Preslikavanje iz entiteta baze u objekte sučelja 72
3.4.3. Postavljanje podataka na sučelje 74
3.4.4. Način rada aplikacije 75
3.5. Testiranje programskog rješenja 76
3.5.1. Testni duplikati. 76
3.5.2. Testiranje ViewModela današnje prognoze 78
3.5.3. Testiranje preslikavanja u sažetak dana. 81
4. ZAKLJUČAK 83
Literatura. 84
Sažetak 90
Abstract 91
Životopis 92
Prilog 1 93

1. UVOD

Danas je gotovo nemoguće napraviti brzu i privlačnu Android mobilnu aplikaciju bez korištenja nekog oblika paralelizma, jer skoro svaka aplikacija sadrži neku blokirajuću operaciju, kao komuniciranje s mrežom i/ili lokalnom bazom podataka. Ako se ovakve operacije izvrše bez brige o paralelizmu, dolazi do zastajkivanja korisničkog sučelja što odbija korisnike i uzrokuje nepovjerenje u sposobnost aplikacije da obavi svoj posao. Srećom, postoji mnoštvo mehanizama paralelizma koji se mogu upotrijebiti na Android platformi, ali nažalost, većina ih sa sobom nosi relativno strmu krivulju učenja i ustaljene loše obrasce korištenja. Zato razvojni programeri mogu postati preplavljeni izborom tijekom odabira pravog mehanizma, pri čemu im ni mrežni izvori nekad ne mogu pomoći jer i oni šire zastarjele i loše prijedloge i obrasce. Čak je i službena dokumentacija bila zaslužna za neke od njih, ali i za čitave loše i kratkovidno dizajnirane mehanizme. Cilj ovog rada je predstaviti tri najvažnija mehanizma paralelizma za razvoj Android aplikacija, ukazati na njihove nedostatke, česte probleme i loše obrasce te dati preporučeni obrazac korištenja koji ih minimizira.

Drugo poglavlje ukratko daje teorijsku podlogu paralelizma i uvodi čitatelja u pojmove istodobnosti i asinkronosti, koji su potrebni kako bi se iskoristila paralelna arhitektura današnjih mobilnih uređaja. Zatim opisuje dretve, RxJavu i korutine kao tri najvažnija mehanizma paralelizma. Za svakog ističe njihove česte probleme i anti-obrasce te daje preporučene obrasce korištenja. Na kraju poglavlja opisuje se postupak testiranja korutina i daje kvalitativna usporedba obrađenih mehanizama. Treće poglavlje opisuje implementaciju potpune Android aplikacije za vremensku prognozu koja koristi korutine za svoje potrebe paralelizma. Pokriva njezine zahtjeve, korištene tehnologije, implementacijske detalje i testiranje. Četvrto poglavlje je zaključak u kojem se kratko navode i komentiraju prednosti i nedostaci obrađenih mehanizama te daje preporučeni mehanizam i smjernice za daljnji rad.

2. RAZVOJ PARALELIZIRANE PROGRAMSKE PODRŁKE

Paralelna obrada zadataka nije svojstvena samo računarstvu. Ljudi u svakodnevnom životu paralelno obrađuju zadatke bez da su toga i svjesni. Osoba koja ujutro pristavi čaj će, dok voda ne proključa, čitati vijesti, praviti doručak ili pričati s ukućanima. Ona ne mora aktivno čekati i provjeravati je li voda gotova jer će čajnik dati obavijest o završetku – zviždati će, i tada zna da je voda proključala i može prekinuti ono što trenutno radi, napraviti čaj i nastaviti sa svojim danom. Nasuprot tome, mogla je stajati ispred čajnika nekoliko minuta i strpljivo čekati dok voda ne proključa i tek onda napraviti čaj, pa doručak, pa čitati vijesti, itd. Tada bi izgubila nekoliko minuta na čekanje i njezin postupak spremanja za dan trajao bi duže, a vrijeme koje je izgubila na čekanje mogla je iskoristiti na učinkovitiji način. Računarstvo se suočava sa sličnim problemom.

2.1. Paralelizam

Paralelno računalo je skup procesora koji surađuju u rješavanju računalnog problema [1, str.4]. Takva definicija dovoljno je općenita da uključuje raznovrsne oblike paralelizma poput međusobno nezavisnih, geografski distribuiranih računala koji su spojeni mrežom i surađuju na nekom zadatku (npr. volontersko računarstvo, BOINC). Uključuje i grozdove računala smještenih na jednoj geograskoj lokaciji koji su povezani brzom mrežom kako bi surađivali na zadatku kao da su jedno računalo, superračunala s nekoliko tisuća procesora, stolna računala s višejezgrenim procesorima i, ono na što se ovaj rad usredotočuje, prijenosna računala s višejezgrenim procesorima.

Danas se skoro svako osobno računalo sastoji od nekolicine procesora u obliku jednog višejezgrenog procesora, ali nije uvijek bilo tako. U prošlom stoljeću kad su se procesori tek pojavili i tijekom njihovog razvoja, imali su samo jednu jezgru, a poboljšanja su se ostvarivala ubrzanjem jednojezgrenih performansi. Iako performanse računala ovise o vremenu potrebnom da izvrši jednu operaciju te količini operacija koje može izvršavati istodobno [1, str.7], početni napredak procesora fokusirao se na smanjenje vremena potrebnog za izvršavanje jedne operacije. Veličine tranzistora smanjivale su se prema Mooreovom zakonu, pa ih je bilo moguće staviti više na jedan čip. Povećavale su se brzine ciklusa, a time i broj ciklusa u sekundi, čime je procesor mogao izvršiti sve više primitivnih operacija u istom vremenu. Skupovi instrukcija napredovali su od 4-bitnog (prvi komercijalni procesor Intel 4004) do 8-bitnog, 16-bitnog, 32-bitnog, te 64-bitnog što je omogućilo sve veće oblike paralelizma na razini bitova čime su se iste operacije izvršavale u manje primitivnih operacija i time brže. Dok je bilo područja za rast u jednojezgrenim performansama, proizvođači su ga iskorištavali, a dolaskom sve bržih procesora obrađivali su se sve zahtjevniji problemi, što je motiviralo sve brže procesore, i tako u krug.

2.1.1 Novo normalno

Ovakvom trendu rasta došao je kraj jer se povećanjem brzine ciklusa procesora povećavaju i njegovi energetski zahtjevi, a time i disipacija topline. Previsoki energetski zahtjevi natjerali

su Intel da 2004. godine otkaže razvoj Tejas i Jayhawk jednojezgrenih procesora. Naveli su da će umjesto njih prebaciti fokus na dvojezgrene procesore [2] jer je tržiste sada spremno za njih time što su razvojni programeri počeli pisati programe s paralelizmom na umu. Novi je način ubrzanja računala postao dodavanje jezgri procesorima i sve veća podrška paralelizmu, odnosno izvršavanju više zadataka istodobno. Problem koji se pojavio, a kojeg su Intelovi predstavnici također spomenuli [2], je što napredak u ovom smjeru ne znači ništa ako programeri ne pišu programe koji mogu iskoristiti višejezgrenu arhitekturu. Ubrzavanjem jednojezgrenih procesora programeri nisu morali mijenjati ništa u vezi svojih programa (osim prilagoditi ih višebitnim arhitekturama). Njihov bi se program samo pokretao na bržem procesoru, što je značilo da bi se brže izvodio. Kako bi programer iskoristio paralelnu arhitekturu, mora prilagoditi svoje algoritme istodobnom izvođenju. Pri tome treba paziti na [1, str.24]:

- Lokalnost – rad s lokalnim podacima (čitanje i pisanje) bolji je od rada s udaljenim podacima (primanje i slanje). Ovo svojstvo se odnosi na paralelna računala velikih razmjera, primjerice distribuirana računala ili grozdove gdje se podaci ponekad trebaju slati i primati preko mreže, ali i na međuprocesnu komunikaciju, iako se ona možda odvija na istom računalu.

- Skalabilnost – algoritmi i programi trebaju biti otporni na buduće povećanje broja dostupnih procesora.

- Istodobnost – algoritmi i programi trebaju se moći izvršavati na više procesora u isto vrijeme bez utjecaja na krajnji rezultat.

- Modularnost – algoritmi i programi trebaju se rastaviti u jednostavne komponente koje se mogu dodavati ili uklanjati po potrebi.

Kako bi se napisao paralelni program (ili paralelizirao serijski) treba se voditi brige o rastavljanju problema kojeg program rješava na dijelove koji se mogu izvršavati istodobno i pritom dati iste rezultate kao da se izvršavaju slijedno. Rastavljanje problema na takve dijelove zove se particioniranje i postoje dva pristupa koja se pritom koriste [1, str.29-31]:

- Domenska ili podatkovna dekompozicija koja je fokusirana na podjelu podataka, a zatim definiciju procedura koji ih obrađuju. Primjer ovakve dekompozicije je problem množenja vektora i matrice suglasnih dimenzija, kao što jednadžba 2-1 prikazuje. Može se uočiti da svaka ćelija rezultata ovisi o jednom stupcu matrice i zato se dekompozicija može izvršiti podjelom matrice u skupove nezavisnih stupaca.

$$\begin{bmatrix} v_1 & v_2 \end{bmatrix} \times \begin{bmatrix} m_{1,1} & m_{1,2} \\ m_{2,1} & m_{2,2} \end{bmatrix} = \begin{bmatrix} v_1 * m_{1,1} + v_2 * m_{2,1} & v_1 * m_{1,2} + v_2 * m_{2,2} \end{bmatrix} \tag{2-1}$$

  Svakom će rasponu stupaca (čija veličina najčešće ovisi o broju dostupnih procesora) biti dodijeljena programska procedura koja ih obrađuje. Ona će u ovom slučaju biti računska operacija koja računa vrijednost ćelije rezultirajućeg vektora.

- Funkcionalna dekompozicija koja je fokusirana na podjelu procedura, a zatim definiciju podataka koje svaka obrađuje. Čest primjer takve dekompozicije je rastavljanje aplikacije s korisničkim sučeljem na procedure koje se brinu o I/O operacijama i one koje se brinu o iscrtavanju elemenata sučelja što je vrlo često u Android aplikacijama (i programima s korisničkim sučeljem općenito).

Osim toga, treba znati za kakav oblik paralelnog računala se piše program. Michael J. Flynn je 1972. godine podijelio računala u četiri tipa u ovisnosti o njihovim sposobnostima obradbe toka instrukcija i toka podataka [3]. Rekao je da računala ili mogu obrađivati više takvih tokova u jednom ciklusu ili jedan, što vodi do četiri kombinacije računala koji su prikazani u tablici 2.1.

Podaci\instrukcije	Jedan tok	Više tokova
Jedan tok	SISD	MISD
Više tokova	SIMD	MIMD

Tab. 2.1: *Flynnova podjela računala prema tokovima instrukcija i podataka*

Računala koja tijekom jednog ciklusa mogu obrađivati samo jedan tok instrukcija i pritom mogu koristiti samo jedan tok podataka zovu se SISD (engl. *single instruction-stream, single data-stream*) računala. Još se zovu serijska ili neparalelna računala, a primjer su računala s jednojezgrenim procesorima. Računala koja tijekom jednog ciklusa mogu obrađivati više tokova instrukcija i pritom koristiti više tokova podataka zovu se MIMD (engl. *multple instruction-stream, multiple data-stream*) računala. Primjer su današnja računala s višejezgrenim procesorima. Svaka od jezgri procesora može istodobno izvršavati posebne instrukcije (npr. jedna se bavi programom za e-poštu, druga se bavi korisničkim sučeljem, treća preglednikom) i pritom svaka može čitati svoje podatke. Bitno je napomenuti da SISD računala mogu rukovati višestrukim procesima u isto vrijeme, ali ih ne mogu izvršavati u isto vrijeme. SIMD i MISD računala imaju posebnu primjenu i van su opsega ovog rada. Ukratko, SIMD računala se koriste kod vektorskih procesora kad je potrebno izvršiti isti tok instrukcija nad više podataka. MISD računala su još neobičnija time što se više tokova instrukcija treba izvršiti na istom toku podataka, za što nema puno primjena u stvarnom životu.

Android uređaji su SISD ili MIMD računala. Prije uređaja LG Optimus 2X koji je izašao 2011. godine i bio prvi Android uređaj s dvojezgrenim procesorom [4], Android uređaji su imali jednojezgrene procesore i time bili SISD računala. Danas su to gotovo isključivo MIMD računala i stoga je potrebno iskoristiti njihove mogućnosti pisanjem prikladnog programskog koda.

2.2. Istodobnost

U poglavlju 2.1 pojmovi paralelizam i istodobnost spomenuti su nekoliko puta. Paralelizam se odnosi na izvršavanje više operacija u isto vrijeme na nezavisnim obradbenim jedinicama. Primjerice, u jednom djeliću vremena, jedan procesor MIMD računala izvršava program za e-mail dok drugi izvršava program za reprodukciju glazbe. Još jedan primjer su klijent-poslužitelj

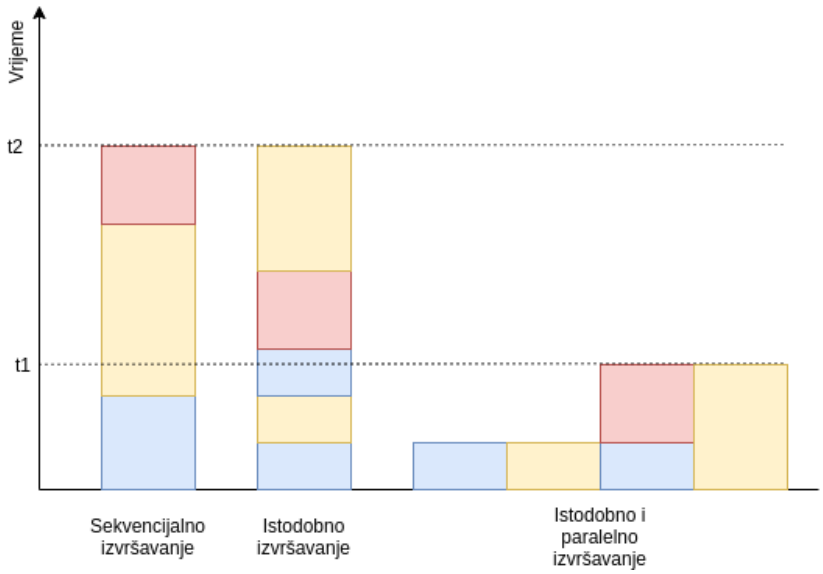
sustavi, gdje klijent napravi zahtjev za udaljenim resursom poslužitelja i dok poslužitelj ne dostavi podatke, klijent se bavi iscrtavanjem sučelja ili nekom drugom operacijom, a poslužitelj poslužuje ostale klijente. U oba slučaja, odvojeni procesori u istom djeliću vremena izvršavaju drugačije operacije.

Dok se paralelizam bavi izvršavanjem zadataka, istodobnost se bavi njihovom strukturom; to je kompozicija operacija koje se nezavisno izvršavaju [5]. Istodobnost i paralelizam se često koriste zajedno, ali potrebno je naglasiti kako to nisu sinonimi, niti je jedan pojam podskup drugog. Moguće je imati paralelizam bez istodobnosti, kao paralelizam na razini bitova, i istodobnost bez paralelizma, kao na SISD računalima pomoću dijeljenja vremena (engl. *time-sharing*). Rastavljanjem programa na takve dijelove on se ne mora izvršavati paralelno ako računalo na kojemu se izvršava ne podržava paralelizam, ali ako podržava, programer ne mora raditi daljnje izmjene što čini program skalabilnim.

Osim paralelizma, cilj istodobnosti je bolja struktura programa i međusobna nezavisnost dijelova od kojih je sastavljen. Dobar primjer toga nalazi se u svakoj Android aplikaciji gdje je u nekom trenu potrebno dohvatiti podatke s Interneta, npr. u aplikacijama za razgovore. Povlačenjem liste razgovora prema dolje pojavljuje se ikona za osvježavanje i puštanjem ona se krene animirati. To ukazuje na dohvaćanje sadržaja s Interneta koji će biti prikazan kad ona nestane. U pozadini, operativni sustav upravlja s dva nezavisna dijela pokrenute aplikacije: jednim koji iscrtava korisničko sučelje zbog čega se vidi animacija i drugim koji rukuje internetskim zahtjevom. Na MIMD računalu, ti će dijelovi možda biti raspoređeni tako da svaki dobije svoj procesor, a možda će se svejedno svaki dio izvršavati na jednom procesoru. Na SISD računalu će operativni sustav svakom dijelu naizmjence dati djelić vremena u kojemu se može izvršavati na njegovom jednom procesoru, pa će se korisniku činiti kao da se izvršavaju u isto vrijeme.

Oba slučaja ne bi bila moguća bez istodobnosti, jer kad bi sve to bio jedan dio, korisničko bi sučelje bilo smrznuto dok ne stigne odgovor od poslužitelja. Slika 2.1 prikazuje razliku između ovih pojmova u ovisnosti o vremenu potrebnom za njihovo izvršavanje. Na toj slici, program se sastoji od tri zadatka: crvenog, žutog i plavog. Slijednim izvršavanjem oni se u cijelosti izvršavaju jedan za drugim. Istodobnim je moguće prekinuti izvršavanje jednog, prebaciti se na neki drugi, te se vratiti onom prekinutom. Ključna stvar za primijetiti je da istodobno izvršavanje samo po sebi ne daje brže izvođenje programa. Tek se kombinacijom istodobnosti i paralelizma dobiva ubrzanje jer se sada nezavisni dijelovi mogu izvršavati na više procesora (četiri, u ovom slučaju) paralelno.

U sažetku, razlika između istodobnosti i paralelizma je u tome što se istodobnost bavi kompozicijom programa, a paralelizam se bavi njegovim izvršavanjem. Nakon što se problem rastavi na dijelove koji se mogu izvršavati nezavisno, potrebno je implementirati rješenje tog problema u programskom kodu za što služe razni mehanizmi kojima se takvi dijelovi mogu predstaviti.



Sl. 2.1: *Usporedba istodobnosti i paralelizma*

2.3. Višedretvenost

Na najvišoj razini, zadaci koje operativni sustav izvršava su programi kao preglednik, program za e-poštu, upravitelj datotekama, upravitelj prozorima i sl. Program se prije izvršavanja nalazi u statičkoj memoriji, npr. na tvrdom ili SSD (engl. *solid state drive*) disku i sastoji se od slijeda instrukcija u strojnom kodu koje računalo može izvršiti. Kad korisnik želi pokrenuti program klikom na njegovu ikonicu, operativni sustav ga mora učitati iz statičke u radnu memoriju te započeti izvršavanje njegovih instrukcija. Učitani program u izvršavanju zove se **proces** [6, str.25].

Operativni sustav rukuje višestrukim procesima istodobno raspoređivanjem procesorskog vremena tako da svaki proces dobije djelić vremena (engl. *time slot* ili *quantum*) u kojem može izvršavati svoje dijelove posla. Dijelovi posla se u procesima organiziraju u obliku **dretvi** (niti, engl. *threads*) koje su najmanji izvedivi tokovi instrukcija koje procesor može izvršiti. Svaki proces ima jednu glavnu dretvu koja se stvara zajedno s njim. Ona predstavlja ulaz u program i, ako razvojni programer ne stvori niti jednu drugu dretvu, ona je jedina dretva u tom procesu. Primjer koda 2.1 prikazuje najjednostavniji primjer jednodretvenog programa s ispisom imena dretve u kojoj se napisani kod pokreće, odnosno glavne (engl. *main*) dretve.

Primjer koda 2.1: *Jednodretveni program s ispisom imena dretve*

```
1  fun main() {
2      println("Hello world!")
3      println("Thread name: ${Thread.currentThread().name}")
4  }
5
6  /*
7  Output:
8      Hello world!
9      Thread name: main
10 */
```

Iz glavne dretve je moguće pokrenuti proizvoljnu količinu sporednih dretvi i tada program postaje višedretven (engl. *multithreaded*). Upravo se stvaranjem novih dretvi postiže istodob-

nost. Svaka dretva dijeli memorijski prostor procesa u kojemu se nalazi, što znači da će objekti u dosegu (engl. *scope*) glavne dretve biti i u dosegu stvorene. Primjer koda 2.2 prikazuje kako je jedinstveni identifikator objekta (engl. *hash code* [7]) jednak u obje dretve koje su stvorene iz glavne dretve, što znači da obje imaju referencu na isti objekt. Također, jer se objektima koji su stvoreni unutar dretvi jedinstveni identifikatori međusobno razlikuju, takvi su objekti svojstveni samo onoj dretvi koja ih je stvorila.

Primjer koda 2.2: *Demonstracija dijeljenja memorijskog prostora između dretvi*

```
1  val globalObject = Any()
2
3  fun main() {
4      demonstrateMemorySharing()
5      demonstrateMemorySharing()
6  }
7
8  private fun demonstrateMemorySharing() {
9      Thread {
10         println("${Thread.currentThread().name} - global object hash code: ${
               ↪ globalObject.hashCode()}")
11
12         val localObject = Any()
13         println("${Thread.currentThread().name} - local object hash code: ${localObject.
               ↪ hashCode()}")
14     }.start()
15 }
16
17 /*
18 Output:
19 Thread-0 - global object hash code: 2027534214
20 Thread-0 - local object hash code: 856805482
21 Thread-1 - global object hash code: 2027534214
22 Thread-1 - local object hash code: 923478855
23 */
```

Višestrukim pokretanjem koda iz primjera 2.2 redoslijed ispisa nije konstantan. Ovisno o redoslijedu kojim operativni sustav rasporedi izvršavanje dretvi, moguća je bilo koja kombinacija. Moguće je ispreplitano izvršavanje gdje se dretva `Thread-0` izvrši do linije 10. Nakon toga ju operativni sustav zaustavi i dodijeli procesorsko vrijeme dretvi `Thread-1` u kojem se ona izvrši. Na kraju dodijeli vrijeme dretvi `Thread-0` u kojemu se ona izvršava do kraja. Upravo to se dogodilo u primjeru 2.3, a mogući su i ostali redoslijedi kao `0101`, `1100`, `1010` itd. Zbog toga je nemoguće raditi pretpostavke o redoslijedu izvršavanja dretvi niti o vremenu u kojem će se dretva izvršiti. Sve ovisi o tome na koji će način operativni sustav rasporediti njihovo izvršavanje.

Primjer koda 2.3: *Neodređenost redoslijeda izvršavanja nezavisnih dretvi*

```
1 /*
2 Output:
3 Thread-0 - global object hash code: 132793263
4 Thread-1 - global object hash code: 132793263
5 Thread-1 - local object hash code: 1399812740
6 Thread-0 - local object hash code: 2062442815
7 */
```

2.4. Asinkronost u Androidu

Razvijanjem aplikacija s korisničkim sučeljem nastoje se apstrahirati kompleksne računalne operacije jednostavnom, funkcionalnom i ugodnom fasadom koja predstavlja vrijednost korisniku.

Funkcionalnost i jednostavnost aplikacije su kvalitete koje se definiraju pri dizajnu korisničkog sučelja i definiciji njezinih funkcionalnosti. Neke od takvih kvaliteta su:

- Paleta boja, pristupačnost, tamna i svijetla tema, margine itd.
- Pomni odabir izloženih opcija i funkcionalnosti prema korisniku i njihov oblik u aplikaciji.
- Minimizacija skupih operacija kao pristup mreži (potrošnja mobilnih podataka) ili lokaciji (potrošnja baterije).
- Poštivanje preporučenih obrazaca dizajna platforme (npr. Material Design [8] za Android) što vodi jednostavnijem, prepoznatljivijem i ugodnijem korisničkom iskustvu.

Stvaranjem aplikacije koja se slaže sa svime navedenim u specifikaciji i dizajnu može pružiti funkcionalnost, jednostavnost i privlačnost, ali ne nužno i ugodno korisničko iskustvo. Iako se u fazi specifikacije i dizajna mogu definirati kvalitete aplikacije koje poboljšavaju korisničko iskustvo, potreban je pažljiv razvoj kako bi aplikacija imala performanse koje ga omogućavaju.

2.4.1 Zastajkivanje u Android aplikacijama

Jedan od najvećih krivaca za loše korisničko iskustvo u Android aplikacijama je zastajkivanje sučelja. Dugoročni korisnici Android mobilnih uređaja sigurno su se susreli s time bezbroj puta. Očituje se isprekidanim pomicanjem u listama, povećanom vremenu odziva na radnje korisnika (npr. klikovi, pomicanje, navigacija), isprekidanim animacijama i sporim iscrtavanjem elemenata sučelja. U najgorem slučaju, ako aplikacija nije u mogućnosti reagirati na unos korisnika duže od pet sekundi, Android platforma prikazuje poruku greške o neresponzivnosti (engl. *Application Not Responding*, ANR) koja korisniku daje mogućnost prisilnog zaustavljanja aplikacije [9].

Ovakvi i slični problemi s performansama mogu znatno utjecati na korisničko iskustvo. Čak i ako su zatrzavanja rijetka, tijekom vremena mogu uzrokovati nezadovoljstvo i nestrpljivost, pogotovo u današnje vrijeme gdje su ljudi naviknuti na, i očekuju, skoro trenutan odziv (unutar 16 milisekundi [10]) na svaku radnju. Dokaz ovom trendu je prelazak proizvođača mobilnih uređaja na korištenje:

- Ekrana s visokom brzinom osvježavanja (frekvencije 90Hz i više) koji omogućavaju veću brzinu ažuriranja prikazane slike. Takvi ekrani pružaju glađi prikaz pomičnog sadržaja što je najprimjetnije tijekom pomicanja lista, tranzicijskih animacija, reprodukcije videa i animacija.
- Ekrana s visokom brzinom uzorkovanja dodira (frekvencije iznad 120Hz) koji omogućavaju veću brzinu reakcije na dodir. Utjecaj toga očituje se pomicanjem lista i interakcijom sa sučeljem, ali i u igrama, gdje se doima kao da reakcije sučelja bliže prate korisnikov prst.

- Velike količine radne memorije (4GB i više) što omogućava veću količinu istovremeno učitanih aplikacija. Ovo smanjuje slučajeve u kojima se aplikacija mora učitati potpuno iznova, što smanjuje vrijeme čekanja na paljenje aplikacija.
- Posebnog procesora koji posvećenog obradi slike u Google Pixel 2, 3 i 4 uređajima [11] koji omogućavaju brzu HDR+ obradu slike i time brzo fotografiranje. Korisnici su se požalili na sporu obradu fotografija u novim Pixel 5 uređajima koji zbog manje cijene nisu imali ovaj procesor [12].

Zbog toga je danas korisnička tolerancija na zastajkivanje znatno manja nego prije dok su pametni uređaji još sazrijevali kao tehnologija. U današnje, zastajkivanje ukazuje jedino na nemarno i loše izrađenu aplikaciju. U najboljem slučaju može uzrokovati blago nezadovoljstvo i iritiranost korisnika, a u najgorem nepovjerenje u sigurnost, privatnost, pouzdanost i integritet aplikacije i podataka koje joj korisnik povjerava na korištenje i obradu [13, str.16].

2.4.2 Glavna dretva

Pokretanjem aplikacije Android za nju stvara novi Linux proces s glavnom dretvom. Proces se stvara račvanjem `Zygote` procesa koji je stvoren paljenjem sustava i sadrži predučitan platformski kod i resurse [14], a svrha mu je smanjenje početnog vremena pokretanja aplikacija. Osim glavne, stvaraju se `Binder` dretve koje rukuju zahtjevima ostalih procesa i ostale dretve virtualnog stroja čijim životnim ciklusom upravlja sustav [15, str.30,34].

Pokrenuti procesi i dretve na uređaju mogu se analizirati pomoću ADB (engl. *Android Debug Bridge*) alata komandne linije koji omogućava komunikaciju sa spojenim Android uređajem. Za svrhu primjera, stvorena je prazna Android aplikacija imena `VoidApp`. Pokrenuta je na uređaju s upaljenim ispravljačem grešaka (engl. *debugger*) te su njezine dretve analizirane. Rezultati eksperimenta nalaze se u tablici 2.2 iz koje se može uočiti nekoliko stvari:

1. Kod aplikacijskog procesa je identifikator roditeljskog procesa jednak identifikatoru procesa `Zygote`, što znači da je aplikacijski proces nastao njegovim račvanjem.
2. Glavnu dretvu procesa prepoznaje se prema imenu dretve koje odgovara imenu procesa [15, str.33]. Osim toga, njezin identifikator jednak je identifikatoru procesa.
3. Pri instalaciji, svim je aplikacijama dodijeljen Linux korisnik s jedinstvenim identifikatorom, čime su aplikacije međusobno nezavisne i odvojene. Jedan korisnik, odnosno aplikacija, nema dopuštenje pristupa podacima drugog korisnika. U tablici, identifikator korisnika `VoidApp` aplikacije je `u0_a936` i on je vlasnik aplikacijskog procesa kao i njegovih dretvi.

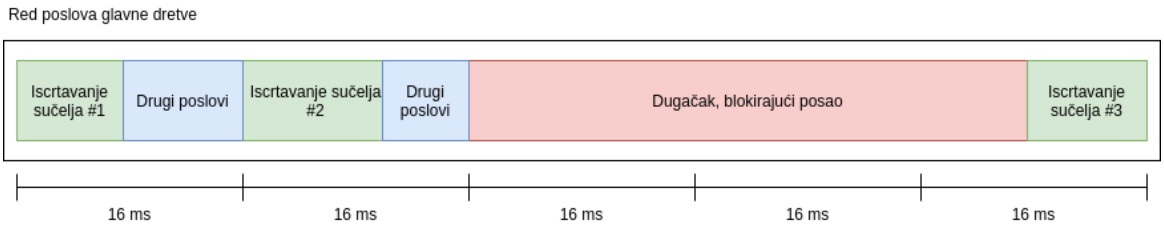
Glavna dretva Android aplikacije ima jednostavan dizajn. Njezin jedini posao je obraditi zadatke iz svog reda zadataka. Istodobno može izvršavati samo jedan zadatak i tek kad završi s njim, povlači novi i ponavlja postupak. Većinu zadataka u njezinom redu čekanja generira platforma, a oni uključuju sustavske događaje (npr. `BroadcastReceiver`), ulazne događaje (npr.

Korisnik	Proces	Roditeljski proces	Dretva	Ime procesa	Ime dretve	Komentar
root	3445	1	–	zygote64	–	*Zygote* proces
u0_a936	11051	3445	–	hr.voidapp	–	Aplikacijski proces
u0_a936	11051	3445	11051	hr.voidapp	hr.voidapp	Glavna dretva
u0_a936	11051	3445	11063	hr.voidapp	Binder:11051_1	*Binder* dretva
u0_a936	11051	3445	11064	hr.voidapp	Binder:11051_2	*Binder* dretva
u0_a936	Desetak izostavljenih ART dretvi kao *FinalizerDaemon*, *HeapTaskDaemon* i sl.					

Tab. 2.2: *Uređeni ispis dretvi VoidApp aplikacije*

klikovi), aplikacijske povratne pozive (engl. *callback*) (npr. slušatelji događaja), kod iz drugih aplikacijskih komponenti (npr. servisi), periodički zakazane događaje (npr. `AlarmManager`) te iscrtavanje korisničkog sučelja [16] što je najvažniji zadatak za ugodno korisničko iskustvo.

Pri iscrtavanju sučelja, njegova ciljana okvirna stopa (engl. *frame rate*) koju korisnici smatraju glatkom i koju Google preporučuje [10] je 60 okvira po sekundi (engl. *frames per second*). Kako bi se postigla takva brzina, potrebno je iscrtati korisničko sučelje svakih 16.67 milisekundi. Ako iscrtavanje sučelja postane odgođeno zbog nekog drugog zadatka na glavnoj dretvi dolazi do propuštanja okvira što korisnik vidi kao zastajkivanje. Jedan propušteni okvir je neprimjetan jer tada okvirna stopa pada na 59fps, ali ako ih se propusti nekoliko, okvirna stopa pada do razine koju korisnik primjećuje. U primjeru na slici 2.2 dugačak, blokirajući zadatak uzrokuje dva i pol propuštena okvira. Tijekom blokirajućeg zadatka novi zadaci iscrtavanja sučelja stižu u red čekanja, ali ne mogu se izvršiti zbog njega. Tek kad on završi, glavna dretva može uzeti prvi novi zadatak iz reda čekanja i krenuti s njegovim izvršavanjem, čime sučelje opet postaje responzivno.

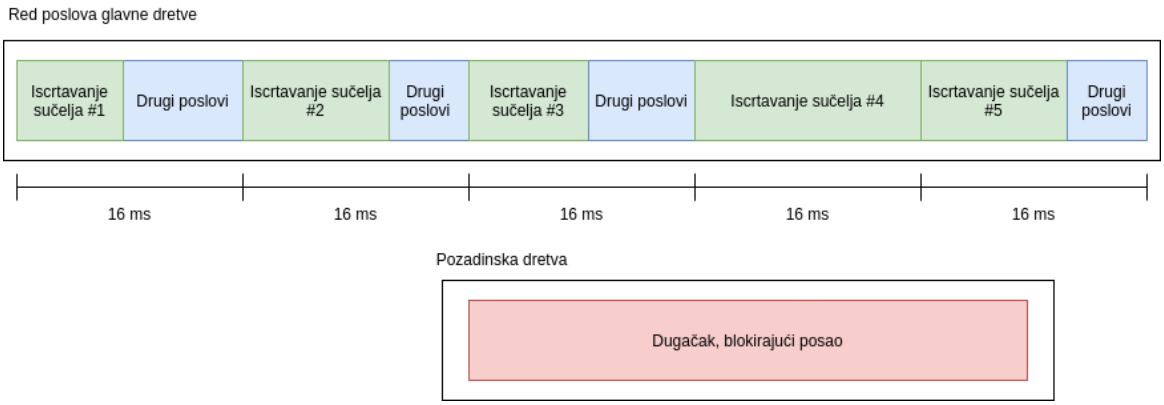


Sl. 2.2: *Blokiranje glavne dretve*

2.4.3 Asinkronost

Blokiranje glavne dretve može se izbjeći prebacivanjem blokirajućih zadataka na pozadinske dretve kako bi se mogli izvršavati istodobno. Time joj se omogućava da se bavi samo svojom svrhom – iscrtavanjem sučelja i rukovanjem događajima. Čest primjer uporabe pozadinskih dretvi su mrežni pozivi. Potrebno je napraviti zahtjev za nekim udaljenim resursom te obraditi rezultate i prikazati ih korisniku. Kad bi se mrežni poziv radio na glavnoj dretvi, blokirao bi ju sve dok rezultat ne stigne, što može trajati desetke sekundi. Način obrade zadataka gdje se jedna operacija može izvršiti tek kad druga završi, a do tog trenutka program blokira i nema priliku raditi ništa drugo (pa ni iscrtavati sučelje) zove se **sinkronost**. Na slici 2.2 crveni posao se izvršava sinkrono s ostalim poslovima dretve.

Kako bi korisničko iskustvo bilo glatko i responzivno, potrebno je prebaciti blokirajuće zadatke s glavne na pozadinske dretve kao na slici 2.3. Stvaranje pozadinskih dretvi je jednostavno (primjer 2.2), ali glavni problem je u komunikaciji između dretvi. Nakon završetka pozadinskog zadatka treba prikazati dobivene podatke za što treba pristupiti referencama elemenata sučelja. Za to je potrebna obavijest o završetku zadatka koja može (ali ne mora) sa sobom nositi te podatke. Takvu se obavijest treba konzumirati na mjestu u kodu iz kojega je moguće pristupiti elementima sučelja kako bi ga se moglo ažurirati rezultatima pozadinske operacije. Način obrade zadataka u kojem se ovisi o obavijestima ili događajima koji javljaju o završetku pozadinske operacije zove se **asinkronost**. Na slici 2.3 crveni zadatak se izvršava asinkrono s ostalim zadacima dretve.



Sl. 2.3: *Oslobođena glavna dretva*

Problem s asinkronošću na Androidu je zabrana pristupa elementima sučelja s bilo koje dretve osim glavne. Kad se to dogodi, platforma podiže iznimku `CalledFromWrongThreadException` koja prisiljava programere da se vrate na glavnu dretvu prije pristupanja elementima sučelja. Time glavno pitanje pri implementaciji asinkronosti na Android platformi nije kako obraditi zadatke, nego kako se nakon toga vratiti na glavnu dretvu i prikazati njihove rezultate. Postoje razni mehanizmi asinkronosti u Javi, Kotlinu i Android API-ju (engl. *Application Programming Interface*) koji mogu olakšati (ili otežati) implementaciju asinkronosti i komunikacije između dretvi.

2.5. Dretve u Javi

Jedan od načina na koji se može postići asinkronost je višedretvenost, a najjednostavniji alat za to su dretve u Javi. Danas je preporučeni programski jezik za razvoj Android aplikacija Kotlin, ali on održava potpunu kompatibilnost s Javinim API-jem, pa se njezine dretve mogu koristiti i u Kotlinu. Iako je danas snažno preporučeno korištenje korutina (engl. *coroutine*) za asinkrone poslove [17], ponekad je potrebno početi od osnovnijih mehanizama kako bi se vidjela šira slika problema. Ovo poglavlje prikazuje načine na koje se asinkronost može postići dretvama u Androidu, koji su pristupi nepoželjni i zastarjeli te preporučeni obrazac korištenja.

2.5.1 Osnovni pristup

Za jednostavne i osnovne slučajeve kad je potrebno samo napraviti blokirajuću operaciju i prikazati njezine rezultate, moguće je to napraviti pomoću obične dretve. Dretva je u Javi implementirana klasom `Thread` koja izvršava zadatak definiran kodom. Nakon što taj kod završi, završava i dretva. Osim toga, dretva se završava i ako se njezin aplikacijski proces završi, jer je život dretve vezan uz život procesa (osim kod *daemon* dretvi). Za demonstraciju je stvorena jednostavna aplikacija s gumbom, trakom za napredak i prekidačem. Klikom na gumb, simulira se dohvaćanje teksta s Interneta, koji se zatim prikazuje na sučelju. Kod aplikacije prikazan je primjero 2.4, a nebitni dijelovi su izostavljeni zbog sažetosti.

Primjer koda 2.4: *Blokiranje glavne dretve*

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        binding.button.setOnClickListener {
            binding.result.text = getStringFromApi()
        }
    }

    private fun getStringFromApi(): String {
        // Pretend we are fetching from an API
        for (i in 1..5) {
            Log.d("GetStringFromApiRunnable", "getStringFromApi: $i")
            Thread.sleep(1000)
        }
        return "A nice string ${Random.nextInt(10)}"
    }
}
```

Klikom na gumb, animacija trake za napredak se zaustavlja i prekidač ne mijenja svoje stanje kad ga se dodirne. Dodirivanjem sučelja nekoliko puta prikazuje se ANR dijalog. Ako se odabere zaustavljanje aplikacije, bit će prisilno zaustavljena, ali ako ne, prikazuje se dobiveni tekst. Dakle, nije stvar u tome što se *ne smije* blokirati glavnu dretvu, nego što to daje izuzetno loše korisničko iskustvo jer korisnik u tih nekoliko sekundi ne zna što se događa ili hoće li se aplikacija oporaviti. To se može popraviti premještanjem posla na pozadinsku dretvu kao u primjeru 2.5.

Primjer koda 2.5: *Asinkronost s rušenjem*

```
binding.button.setOnClickListener {
    Thread {
        binding.result.text = getStringFromApi()
    }.start()
}
```

Sad se klikom na gumb traka napretka nastavlja animirati i sučelje ostaje responzivno na klikove, ali se dolaskom odgovora aplikacija sruši zbog `CalledFromWrongThreadException` iznimke koja glasi: *"Only the original thread that created a view hierarchy can touch its views"*. Ovo se događa jer je u Androidu zabranjen pristup elementima sučelja s bilo koje dretve osim glavne, a kod u primjeru 2.5 radi upravo to. Razlog iz kojega se unutar `Thread`a može pristupiti elementu sučelja je što se u pozadini stvara anonimna `Runnable` klasa koja je u opsegu vanjske klase, odnosno aktivnosti. Međutim, kod koji se nalazi unutar te klase izvršava se u stvorenoj

dretvi (koja nije glavna). Zbog toga je potreban mehanizam kojim se manipulacija sučeljem prebacuje na glavnu dretvu. Postoje tri jednostavna pristupa za to koja su prikazana primjerom 2.6.

Primjer koda 2.6: *(Skoro) sigurna asinkronost*

```
private val resultLiveData = MutableLiveData<String>()

override fun onCreate(savedInstanceState: Bundle?) {
    ...
    binding.button.setOnClickListener {
        Thread {
            val result = getStringFromApi()
            // Approach #1
            binding.result.post {
                // Main thread
                binding.result.text = result
            }
            // Approach #2
            runOnUiThread {
                // Main thread
                binding.result.text = result
            }
            // Approach #3
            resultLiveData.postValue(result)
        }.start()
        // Required for #3
        resultLiveData.observe(this) {
            // Main thread
            binding.result.text = it
        }
    }
}
```

Implementacijom bilo kojeg od ova tri pristupa aplikacija radi kako je predviđeno. Pristupi 1 i 2 u pozadini koriste `Handler` mehanizam Android API-ja za dijeljenje poruka koji omogućava izvršavanje linije koda unutar lambda izraza na glavnoj dretvi (pogledati [18, lin.7064] i [19, lin.18758]). Pristup 3 koristi `LiveData` klasu Android Jetpack API-ja [20] koja implementira oblikovni obrazac promatrača svjesnog životnog ciklusa svoje roditeljske komponente (aktivnosti, u ovom slučaju). Aktivnost se pretplaćuje na događaje `resultLiveData` objekta i svakim događajem mijenja tekst rezultata na sučelju. Metoda `postValue` u pozadini koristi `TaskExecutor` klasu za dostavljanje novog zadatka koji poziva `setValue` na glavnoj dretvi. Sva tri pristupa su samo fasadne metode koje apstrahiraju ključnu funkcionalnost Android asinkronosti, a to je dostavljanje zadataka glavnoj dretvi korištenjem Android APIja za dijeljenje poruka. Iako je ovaj pristup jednostavan i ispunjava svrhu rasterećivanja glavne dretve, ima nekoliko problema koji ga čine neprikladnim i previše jednostavnim za korištenje u razvoju:

- Time što dretva koristi anonimnu unutarnju klasu `Runnable` sučelja, ona ima čvrstu referencu na aktivnost, što sprječava skupljača smeća (engl. *garbage collector*) od oslobađanja memorije koju je aktivnost zauzela ukoliko dretva postane blokirana (curenje memorije, engl. *memory leak*).
- Nema kontrole nad brojem dretvi koje se mogu pokrenuti, što može dovesti do previše dretvi i smanjenja performansi.
- Kod postaje ugniježđen i kad bi se trebalo napisati više takvih dretvi, ubrzo bi postao

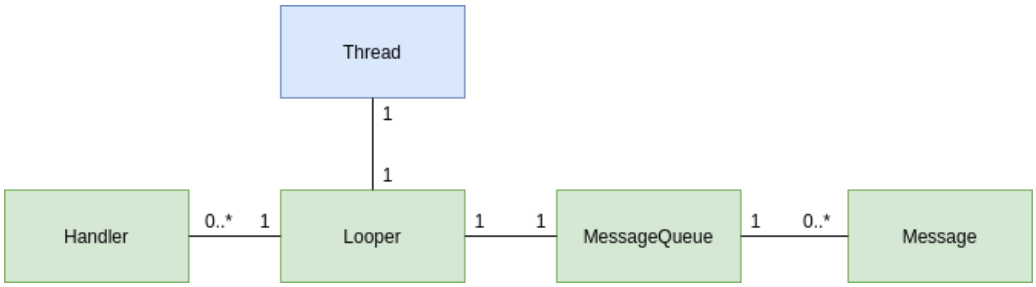
nepregledan i neodrživ.

- Aktivnost bi trebala biti odgovorna samo za prikazivanje podataka, a ne i za prebacivanje na glavnu dretvu kako bi ih mogla prikazati.

Kako bi se kod razdvojio prema nekom od arhitekturnih obrazaca i time apstrahirala asinkronost i izbjeglo curenje memorije, implementacija `Runnable` sučelja mora se izdvojiti u posebnu klasu. Zatim se mora uspostaviti komunikacija između aktivnosti i dretve jer je sada aktivnost izvan njezinog opsega i ne može pristupiti njezinim članovima. Java ima mehanizme međudretvene komunikacije koji se mogu koristiti za ovo u obliku klasa `Pipe`, `BlockingQueue` i dijeljene memorije s mehanizmima sinkronizacije (muteksi, semafori, ključna riječ `synchronized` itd.), ali oni su skloni blokiranju glavne dretve [15, str.47]. Android API umjesto njih pruža neblokirajuće mehanizme dijeljenja poruka kojima se može postići željena funkcionalnost.

2.5.2 Android dijeljenje poruka

Biblioteka za dijeljenje poruka u Android API-ju zasniva se na neblokirajućem obrascu proizvođač-potrošač. Kako bi se obična dretva koristila zajedno s ovim obrascem, potrebno joj je dodijeliti `Looper` koji je odgovoran za neprestano ponavljanje petlje koja izvršava zadatke definirane porukama. Poruke su objekti `Message` klase koji mogu sadržavati podatke i/ili zadatke (`Runnable` objekti), a `Looper` ih izvlači jednog za drugim iz svog reda poruka koji je implementiran objektom `MessageQueue` klase. Poziv kojim se poruka stavlja u red nikad ne blokira i zato je pogodan za komunikaciju između glavne i pozadinskih dretvi. Klasa odgovorna za punjenje reda je `Handler`. Svaki je povezan s jednim `Looper`om koji mu se prosljeđuje u konstruktoru, a time se može reći da je povezan s jednom dretvom jer `Looper` ne može postojati bez dretve. Dijagram 2.4 prikazuje vezu između navedenih komponenti.



Sl. 2.4: *UML dijagram Android APIja za dijeljenje poruka , izrađeno prema [15, str.48]*

Ulogu proizvođača može poprimiti bilo koja dretva s referencom na `Handler`a povezanog s `Looper`om neke dretve, koja je u tom slučaju potrošač. Ovako je građena i glavna dretva Android procesa – njezin `Looper` objekt može se dobiti pozivanjem statičke metode `Looper#getMainLooper` kojeg je moguće povezati s bilo kojim `Handler` objektom i tako slati poruke glavnoj dretvi iz pozadinskih dretvi. Poslane poruke stižu u njezin red poruka gdje se izvršavaju jedna za drugom kao na slici 2.2. Glavna dretva je, dakle, ništa drugo nego obična dretva koja koristi Androidov API za dijeljenje poruka kako bi slijedno izvršavala svoje zadatke. Primjer koda 2.7 prikazuje kako se primjer 2.6 može napisati dijeljenjem poruka.

Primjer koda 2.7: *Sigurna asinkronost pomoću dijeljenja poruka*

```
class MainActivity : AppCompatActivity() {
    ...
    private val workerThread = WorkerThread()

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        workerThread.start()
        binding.button.setOnClickListener {
            workerThread.post(
                GetStringFromApiRunnable(
                    callback = {
                        binding.result.text = it
                    }
                )
            )
        }
    }

    override fun onDestroy() {
        super.onPause()
        workerThread.quit()
    }

    companion object {
        private class GetStringFromApiRunnable(
            private val callback: (String) -> Unit
        ) : Runnable {
            private val mainHandler = Handler(Looper.getMainLooper())

            override fun run() {
                val result = getStringFromApi()
                mainHandler.post {
                    callback(result)
                }
            }

            private fun getStringFromApi(): String {...}
        }
    }
}

class WorkerThread : Thread() {
    private var handler: Handler? = null

    override fun run() {
        Looper.prepare()
        handler = Handler(Looper.myLooper()!!)
        Looper.loop()
    }

    fun post(runnable: Runnable) {
        handler?.post(runnable)
    }

    fun quit() {
        handler?.looper?.quit()
    }
}
```

Dretva `WorkerThread` pretvorena je u `Looper` dretvu pozivom `Looper#prepare` metode koja joj dodjeljuje `Looper` objekt i red poruka [15, str.50]. Tek je nakon toga moguće inicijalizirati njezin `Handler` koji je apstrahiran metodom `post` kako se pozivatelji ne bi brinuli o unutarnjoj strukturi dretve. Dijeljenje poruka pokreće se `Looper#loop` blokirajućom metodom koja sprječava završetak dretve i pokreće petlju koja dohvaća poruke iz reda i izvršava ih. Dretvu se završava pozivom metode `quit` koja apstrahira poziv `Looper#quit` metode koja uklanja sve poruke u redu koje nisu započele s izvršavanjem i zaustavlja petlju. Za mrežnu operaciju

stvoren je `GetStringFromApiRunnable` objekt odgovoran za izvršavanje operacije i za poziv svog `callback` povratnog poziva na glavnoj dretvi. Ovakvim pristupom izbjegnuto je curenje memorije i odgovornost aktivnosti za dretvenu sigurnost.

Ovim je pristupom sada lako napraviti dodatne `Runnable` klase koje će implementirati proizvoljne zadatke i izvršavati ih na stvorenoj dretvi. Ako se svi oni šalju jednom `WorkerThread` objektu, izvršavat će se jedan za drugim, što nije idealno, ali barem neće blokirati glavnu dretvu. Iako je lako napisati nove `Runnable` i `Thread` klase, ubrzo bi postalo zamorno i neefikasno jer bi se svaki put morao pisati isti kod, a i `WorkerThread` klasa ne implementira sve najbolje prakse pri proširivanju `Thread` klase. Za tu se svrhu u Android API-ju nalaze dvije gotove klase: `HandlerThread` i `AsyncTask`.

2.5.3 HandlerThread i AsyncTask

`HandlerThread` je proširenje `Thread` klase i služi kao omotač mehanizama opisanih u primjeru 2.7, ali s dodatnim mehanizmima koji osiguravaju sigurnu inicijalizaciju klase. Razlikuje se od navedenog primjera u tome što izlaže svoj `Looper` kojeg se može koristiti u inicijalizaciji `Handler`a kako bi joj se dostavljale raznovrsne poruke, a ne samo `Runnable` objekti. Primjer 2.8 prikazuje potrebne promjene kako bi se `HandlerThread` koristio umjesto `WorkerThread`. Iako s ovom klasom nema potrebe ručno proširivati `Thread`, svejedno se moraju pisati `Runnable` objekti i rukovati životom dretve unutar aktivnosti.

Primjer koda 2.8: *Primjer 2.7, ali s HandlerThread umjesto WorkerThread*

```
class MainActivity : AppCompatActivity() {
    ...
    private val workerThread = HandlerThread("Worker Thread")
    private val workerThreadHandler by lazy { Handler(workerThread.looper) }

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        binding.button.setOnClickListener {
            workerThreadHandler.post(
                GetStringFromApiRunnable(
                    callback = {
                        binding.result.text = it
                    }
                )
            )
        }
    }
    ...
}
```

Drugi i popularniji mehanizam koji se često koristio za asinkrone poslove je `AsyncTask`. Njegova je svrha bila apstrakcija prebacivanja poslova na pozadinsku dretvu i povratnih poziva na glavnu dretvu. Idealni obrazac korištenja `AsyncTask`a bio je za kratke poslove ne duže od nekoliko sekundi [21] koje nakon završetka prikazuju rezultate na sučelju. Izvorno zamišljen kao oblik "bezbolne višedretvenosti" tijekom vremena pokazao se vrlo zbunjujućim i sklon greškama zbog čega je od API razine 30 službeno zastario (engl. *deprecated*). Primjer 2.9 prikazuje potrebne promjene kako bi se `AsyncTask` koristio umjesto `WorkerThread`.

Primjer koda 2.9: *Primjer 2.7, ali s AsyncTask umjesto WorkerThread*

```
class MainActivity : AppCompatActivity() {
    ...
    private val asyncTasks = mutableListOf<AsyncTask<Void, Void, String>>()

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        binding.button.setOnClickListener {
            val task = GetStringFromApiAsyncTask(
                    callback = {
                        binding.result.text = it
                    }
            )
            asyncTasks.add(task)
            task.execute()
        }
    }

    override fun onDestroy() {
        super.onDestroy()
        asyncTasks.forEach {
            it.cancel(false)
            executorService.shutdown()
        }
    }

    companion object {
        private class GetStringFromApiAsyncTask(
                private val callback: (String) -> Unit
        ) : AsyncTask<Void, Void, String>() {
            override fun doInBackground(vararg params: Void?): String {
                return getStringFromApi()
            }

            override fun onPostExecute(result: String?) {
                callback(result ?: "")
            }

            private fun getStringFromApi(): String {...}
        }
    }
```

Jedna se instanca `AsyncTaska` ne može pokrenuti više puta zbog čega je svaki put potrebno instancirati i pokrenuti novi objekt. Kako bi se spriječilo curenje memorije, čuva se lista svih pokrenutih zadataka i završetkom aktivnosti ih se otkazuje. Prednost nad dosadašnjim pristupima je što nije potrebno ručno prebaciti povratni poziv na glavnu dretvu jer se metoda `onPostExecute` izvršava na njoj. Iako je ova klasa napravljena kako bi apstrahirala kompleksnost asinkronosti, dokumentacija `AsyncTaska` često spominje dretve i potrebno ih je poznavati kako bi ga se koristilo na pravilan način. Osim toga, potrebno je poznavati i njegov API. Programer treba:

1. Znati da se svaki `AsyncTask` mora pokrenuti s glavne dretve.
2. Znati da se sve zaštićene metode osim `doInBackground` pozivaju na glavnoj dretvi.
3. Znati kad će svaka od tih metoda biti pozvana.
4. Za svaki novi zadatak pisati novu klasu proširivanjem `AsyncTask` klase.
5. Pri tome joj definirati čak tri generična parametra `Params`, `Progress` i `Results`.
6. Znati da svaka instanca `AsyncTaska` u aplikaciji dijeli jednu pozadinsku dretvu osim ako se ne definira drugačije pozivom `executeOnExecutor` metode. Time su svi asinkroni zadaci

u aplikaciji ograničeni na jednu pozadinsku dretvu što nedovoljno iskorištava današnje višejezgrene procesore. Više o razlogu odabira ove implementacije može se pronaći u [21, metoda `execute`] i [22].

Prema tome se vidi da `AsyncTask` u stvari ne pojednostavljuje asinkronost, nego ju zamagljuje i čini kompliciranom i ograničenom. Prije službenog statusa zastarjelosti Android programeri su ga smatrali *de-facto* zastarjelim i većinom su izbjegavali njegovo korištenje. Umjesto njega pribjegavali su bibliotekama za asinkronost kao RxJava, Android dijeljenje poruka ili Javinom `Executor` okviru u kombinaciji s dijeljenjem poruka.

2.5.4 ExecutorService i više od jedne pozadinske dretve

U primjerima do sada svi su zadaci bili pokretani na jednoj pozadinskoj dretvi. Iako je ovo poželjnije od pokretanja tih zadataka sinkrono s glavnom dretvom, ograničenje na samo jednu dretvu nedovoljno iskorištava današnje procesore koji imaju po osam ili 16 jezgri. Moguće je kompenzirati za to stvaranjem dodatnih instanci dretve i ručnim rukovanjem, ali to ubrzo postaje komplicirano i sklono greškama. Umjesto toga može se koristiti Javin `ExecutorService` [23] čija je svrha izvršavanje zadataka na svom internom skupu dretvi (engl. *thread pool*). Tada nema potrebe za ručnim instanciranjem `Thread`a i ostalih podklasa. Treba se pozvati samo `submit` metoda sa željenim zadatkom, a on će rukovati njegovim izvršavanjem na dostupnim dretvama. Granularnije rukovanje poslanim zadatkom (npr. otkazivanje) može se ostvariti vraćenim `Future` objektom. Primjer 2.10 prikazuje potrebne promjene kako bi se `ExecutorService` mogao koristiti umjesto `WorkerThread`a.

Primjer koda 2.10: *Primjer 2.7, ali s ExecutorService umjesto WorkerThread*

```
class MainActivity : AppCompatActivity() {
    ...
    private val executorService = Executors.newFixedThreadPool(1)

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        binding.button.setOnClickListener {
            executorService.submit(
                    GetStringFromApiRunnable(
                            callback = {
                                binding.result.text = it
                            }
                    )
            )
        }
    }

    override fun onDestroy() {
        super.onDestroy()
        executorService.shutdownNow()
    }
    ...
}
```

Glavna razlika u usporedbi s prethodnim primjerima je što se višestrukim klikovima gumba u isto vrijeme pokreće i izvršava više zadataka. Ovo se događa jer `ExecutorService` raspolaže s više od jedne dretve (u ovom primjeru, točno onoliko koliko ima dostupnih obradbenih jedinica) koje se mogu izvršavati istodobno i međusobno nezavisno. Otkazivanje zadataka uslijed izlaska

iz aktivnosti može se postići metodama `shutdown` (koja će izvršiti sve do sada dodane zadatke, ali spriječiti dodavanje drugih) i `shutdownNow` (koja će odbaciti sve dodane zadatke i pokušati završiti onaj koji se trenutno izvodi). Svrha `ExecutorService`a je apstrakcija stvaranja i rukovanja dretvama. Njime je moguće usredotočiti se samo na zadatak kojeg treba izvršiti, a ostaviti vrijeme i metodu njegovog izvršavanja kao implementacijski detalj. U primjeru se za stvaranje `ExecutorService`a koristila pomoćna metoda `Executors#newFixedThreadPool` koja stvara skup dretvi konstantnog broja. Ako neka dretva završi, nova će biti pokrenuta da popuni njezino mjesto. Ako ima više zadataka nego dostupnih dretvi, oni će čekati dok se jedna od njih ne oslobodi. Postoji još nekolicina gotovih skupova dretvi koji se mogu koristiti, a moguće je stvoriti i svoje nasljeđivanjem `ExecutorService` sučelja.

2.5.5 Česti problemi i anti-obrasci

Dretve u Javi i njihova proširenja jednostavni su oblici asinkronosti koji se mogu koristiti pri razvoju za Android, ali pri tome postoji nekoliko problema koji mogu uzrokovati rušenje aplikacija, spore performanse, zastajkivanje sučelja i sl. Oni najčešći dani su u ovom potpoglavlju.

Ignoriranje životnog ciklusa roditeljske komponente Jedan od glavnih razloga curenju memorije u Android aplikacijama je što komponente kao fragmenti, aktivnosti i servisi ostaju učitani u memoriju duže nego što je potrebno. Stvaranjem anonime klase povratnog poziva i referiranjem na element sučelja `result` u primjeru 2.7 red poruka `WorkerThread`a ima snažnu referencu na aktivnost. Ako dretva postane blokirana, izlaskom iz aktivnosti ta dretva je još uvijek pokrenuta i u njezinom redu poruka se još uvijek nalazi referenca na aktivnost. Time je njezina memorija iscurila. Izlaskom iz komponente treba se prekinuti svaki asinkroni posao kojem je svrha jednokratno prikazivanje podataka jer se izlaskom iz komponente ti podaci neće imati na čemu prikazati. Upravo to se događa u `onDestroy` metodi, prazni se red poruka dretve čime se uklanjaju reference na aktivnost (pogledati [24, lin.362] i [25, lin.425,876]) i sprječava curenje memorije. Otkazivanje poslova kad više nisu potrebni ključno je za sve metode asinkronosti u Androidu.

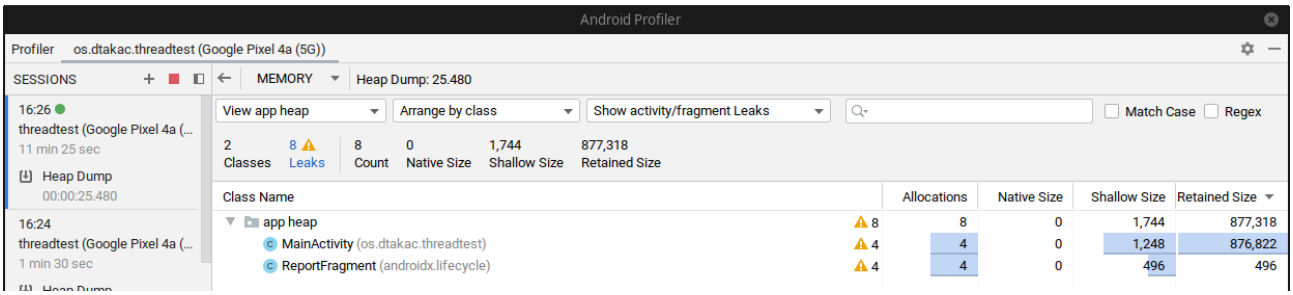
Isto se događa u primjerima 2.8 i 2.9. Iako se `AsyncTask`u pripisuje curenje memorije, češće su za to krivi programeri koji ne vode računa o otkazivanju poslova. To je još jedan primjer neadekvatnosti `AsyncTask`a i apstrakcije koju pokušava postići – programer svejedno mora znati da je potrebno otkazati zadatak što mu je teško za razumjeti bez dubljeg poznavanja dretvi i dijeljenja poruka na Androidu. Pražnjenjem reda poruka nije u potpunosti izbjegnuto curenje memorije. U slučaju izlaska iz aktivnosti usred blokirajućeg zadatka, njezina će se memorija smatrati iscurenom sve dok taj blokirajući zadatak ne završi, za što nažalost nema lijeka. Srećom, većina biblioteka za komunikaciju s mrežom i bazom podataka ima mehanizme vremenskog ograničenja operacije nakon čega podižu iznimku. Rukovanjem tom iznimkom moguće je završiti dretvu i spriječiti curenje memorije.

Ignoriranje mehanizama za otkazivanje dretvi Postoji nekoliko mehanizama za rano otkazivanje dretvi koji mogu znatno smanjiti vrijeme koje ona provodi nepotrebno pokrenuta i time troši memoriju. Osim prirodnog završetka koji se događa završetkom programskog koda, dretve se mogu kooperativno završiti provjerom zastavice `isInterrupted`. U običnoj se dretvi ta zastavica postavlja pozivom metode `Thread#interrupt`, a u ostalim klasama kao `Future` ili `AsyncTask` može se postaviti metodom `cancel` s parametrom `mayInterruptIfRunning` postavljenim na `true`. Kooperativan završetak znači da se u kodu treba povremeno provjeriti vrijednost zastavice i u ovisnosti o njoj odlučiti o daljnjem izvršavanju ili završetku. Ako se zadatak sastoji od jednog blokirajućeg poziva (npr. mrežni poziv), onda neće postojati prozor vremena u kojemu se njezina vrijednost može provjeriti, pa se time zadatak ne može završiti ranije. Ali ako se sastoji od više blokirajućih operacija (npr. preuzimanje datoteka u petlji), tada se može u svakoj iteraciji provjeriti zastavicu i zaustaviti petlju ranije ako je ona postavljena.

Ako nije moguće ručno zaustaviti izvršavanje zadatka, neke biblioteke omogućavaju postavljanje vremenskog ograničenja nakon kojeg podižu iznimku što prisilno zaustavlja dretvu. Npr. HTTP klijent OkHttp3 [26] često korišten uz biblioteku za mrežne pozive Retrofit 2 [27] ima podršku za vremensko ograničenje poziva pomoću metode `callTimeout`. Naravno, korištenjem takvih biblioteka potrebno je pravilno rukovati iznimkama kako se aplikacija ne bi srušila. Ako biblioteka ne podržava ni ovo, jednostavno se mora čekati do završetka zadatka.

Ignoriranje konfiguracijskih promjena Primjeri 2.7, 2.8, 2.9 i 2.10 dobro podnose izlazak iz aktivnosti otkazivanjem poslova u `onDestroy` metodi, ali ima jedan sličan problem kojeg ne podnose, a to su konfiguracijske promjene. Konfiguracija se odnosi na konfiguraciju uređaja kao orijentacija (okomita ili vodoravna), tema sustava (svijetla ili tamna), način rada s višestrukim prozorima, promjena jezika itd. Kad se konfiguracija uređaja promijeni, Android sustav ponovno stvara pokrenutu aktivnost kako bi dobila priliku prilagoditi se novoj konfiguraciji. Primjerice, promjenom iz vertikalne u vodoravnu orijentaciju aktivnost može učitati drugačije sučelje ili promjenom iz svijetle u tamnu temu može učitati drugačije boje ili resurse. Prošle se reference na elemente sučelja uništavaju i aktivnost prolazi kroz svoju `onDestroy` i zatim `onCreate` metodu. Kad bi se uklonila nadjačana `onDestroy` metoda, memorija aktivnosti bi iscurila. Ovo se može vidjeti korištenjem Profiler alata u Android Studiju. Ako se četiri puta promijeni orijentacija i nakon toga klikne gumb koji pokreće dretvu te se zatim pokrene ispis Javine hrpe (engl. *Java heap dump*), može se vidjeti kako je `MainActivity` klasa iscurila četiri puta (slika 2.5).

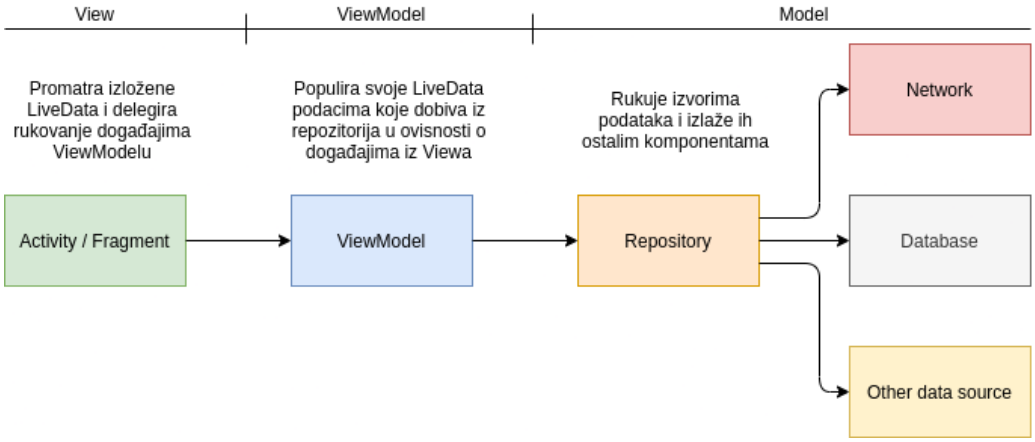
Curenje memorije se u ovom slučaju izbjeglo otkazivanjem poslova u `onDestroy` metodi, ali to nije ono što korisnik očekuje. Ako je kliknuo gumb, očekuje da će se rezultati dugačke operacije prikazati i nakon promjene orijentacije. Kako bi se to postiglo, asinkronom zadatku je potreban način preživljavanja konfiguracijskih promjena. Prije se to radilo nadjačavanjem metoda aktivnosti `onRetainNonConfigurationInstance` i `getLastNonConfigurationInstance` ili pomoću fragmenata kao u [15, str.114], ali danas se ti pristupi smatraju zastarjelima. Pozadinski se poslovi umjesto toga premještaju iz aktivnosti u druge komponente koje mogu



Sl. 2.5: *Ispis memorijske hrpe nakon četiri promjene orijentacije i klika na gumb primjera 2.9*

preživjeti konfiguracijske promjene (npr. `ViewModel`, `Application` ili sustav za ubrizgavanje ovisnosti), a povratni se pozivi stvoreni u aktivnostima zamjenjuju promotrivim objektima na koje se aktivnost pretplaćuje (npr. `LiveData`). Time se uklanja potreba za izravnim rukovanjem konfiguracijskim promjenama.

Pogled odgovoran za asinkronost U aplikacijama u kojima nije implementiran neki arhitekturni obrazac sav kod se nalazi u aktivnosti ili fragmentu, kao u primjerima 2.7, 2.8, 2.9 i 2.10. Aktivnosti i fragmenti smatraju se prikaznim slojem aplikacije koji prema najboljim praksama ne bi trebao biti odgovoran za logiku kojom dohvaća ili manipulira podatcima, nego bi ih samo trebao prikazati kad mu se na neki način dostave iz druge komponente. Preporučeni arhitekturni obrazac za razvoj Android aplikacija je MVVM (engl. *Model-View-ViewModel*) [28]. On rastavlja monolitni prikazni sloj na tri komponente koje su prikazane slikom 2.6.



Sl. 2.6: *Android MVVM arhitektura*

Za razliku od MVP (engl. *Model-View-Presenter*), MVVM arhitektura ima jednosmjeran lanac komunikacije. Pogled (engl. *view*) ima referencu na ViewModel kojem delegira rukovanje događajima. Osim toga, promatra njegove izložene promotrive `LiveData` objekte i prikazuje podatke koji stižu. ViewModel nema referencu na pogled i njegova odgovornost je reagirati na događaje i gurati nove podatke u svoje `LiveData` objekte. Što se njega tiče, s druge strane ne mora ni postojati potrošač tih podataka. On ima referencu na repozitorij (engl. *repository*) iz kojeg dohvaća podatke kojima manipulira kako bi ih pripremio za pogled. Repozitorij nema

referencu na ViewModel i njegova odgovornost je izlaganje podataka koje može dohvatiti iz raznih izvora kao mreže, baze podataka ili nekog drugog izvora. Ovime se postiže razdvajanje odgovornosti koje osim čistog i čitkog koda omogućava višu razinu testabilnosti, štednju resursa i otpornost na konfiguracijske promjene.

Otpornost na konfiguracijske promjene posljedica je toga što ih ViewModel preživljava, odnosno tek se završetkom aktivnosti uništava i njezin ViewModel. To znači da će i pokrenute dretve u ViewModelu preživjeti konfiguracijske promjene. Nadalje, time što ViewModel nema referencu na aktivnost i samo gura podatke u svoje `LiveData` objekte, neće se pojaviti problem gdje se rezultati ne mogu prikazati jer su reference nevažeće. Aktivnost će se pri ponovnom stvaranju opet pretplatiti na `LiveData` objekte i biti u mogućnosti prikazati podatke koji stignu. Ovo sprječava i curenje memorije aktivnosti jer asinkroni kod više nema referencu na nju. Zbog navedenih prednosti je prisutnost bilo kakve poslovne logike u prikaznom sloju anti-obrazac, a time i prisutnost logike asinkronosti koja ona se treba apstrahirati ViewModelom.

2.5.6 Preporučeni obrazac korištenja

Izvršena je analiza prednosti i nedostataka pristupa asinkronosti pomoću običnih Javinih dretvi na Androidu. U ovom dijelu daje se njihov preporučeni obrazac korištenja pri razvoju Android aplikacija [29]. Iako je danas preporučeno pisati asinkroni kod pomoću korutina, nisu svi projekti prešli na Kotlin, a neki možda preferiraju jednostavnost Javinih dretvi. Kod je organiziran prema MVVM arhitekturnom obrascu koji osigurava sigurnost pri konfiguracijskim promjenama. Za asinkrono izvođenje koristi se `ExecutorService` konfiguriran prema dostupnom broju obradbenih jedinica i globalno dostupan pomoću Koin okvira za ubrizgavanje ovisnosti [30]. Aplikacija je jednostavna i sastoji se od gumba koji pokreće blokirajući zadatak koji vraća tekst s informacijom o svom rednom broju. Počinje se od aktivnosti čiji je kod prikazan primjerom 2.11. Kao što primjer prikazuje, ona se sad ne mora brinuti o asinkronosti, dretvama ili logici, već samo o prikazu podataka i korisničkoj interakciji. Bitno je primijetiti i to da se svakim pozivom `onCreate` funkcije (dakle i nakon konfiguracijskih promjena) ona ponovo pretplaćuje na ViewModelove `LiveData` objekte iz kojih dobiva zadnje podatke. Ovime se postiže:

- Otpornost na konfiguracijske promjene.
- Rasterećivanje aktivnosti od poslovne logike.

Primjer koda 2.11: *Aktivnost u MVVM arhitekturi*

```
1  class MainActivity : AppCompatActivity() {
2      ...
3      private val viewModel: MainActivityViewModel by viewModel()
4
5      override fun onCreate(savedInstanceState: Bundle?) {
6          ...
7          passUserActionsToViewModel()
8          observeViewModel()
9      }
10
```

```
    private fun passUserActionsToViewModel() {
        binding.btnGetResult.setOnClickListener {
            viewModel.getNextResult()
        }
    }

    private fun observeViewModel() {
        viewModel.result.observe(this) {
            binding.tvResult.text = it
        }
        viewModel.progressVisible.observe(this) {
            if (it) {
                binding.piProgress.show()
            } else {
                binding.piProgress.hide()
            }
        }
    }
}
```

Sljedeća komponenta u lancu je ViewModel čiji je kod prikazan primjerom 2.12. Njegova metoda `getNextResult` poziva istoimenu metodu repozitorija i stvara tri povratna poziva: `onStart` u kojemu prikazuje traku napretka, `onError` u kojemu postavlja vrijednost rezultata na pogrešku i skriva traku napretka te `onResult` u kojemu postavlja vrijednost rezultata i skriva traku napretka. Bitno je primijetiti da se nigdje ne spominje aktivnost niti se referiraju njezini `View`ovi. Podatci se postavljaju na `MutableLiveData` objekte koji su izloženi u obliku `LiveData` objekata, a aktivnost (ili neka druga komponenta kao fragment) konzumira te podatke. Ako nitko ne promatra te objekte u trenutku kad su im podaci postavljeni, nema veze. Promatrač će dobiti zadnji postavljeni podatak u trenutku pretplate. Metoda `onCleared` poziva se kad ViewModel više nije u uporabi i kad će biti uništen tj. kad njegova aktivnost završava. Kao u prošlim primjerima, tada je potrebno otkazati preostale asinkrone zadatke, što se i radi pozivom repozitorijeve metode `cleanUp`. Ovime se postiže:

- Otkazivanje zadataka u ovisnosti o životnom ciklusu komponente.
- Rasterećivanje ViewModela od dretvene sigurnosti.
- Izbjegavanje curenja memorije uzrokovano povratnim pozivima.

Primjer koda 2.12: *ViewModel u MVVM arhitekturi*

```
class MainActivityViewModel(
    private val resultRepository: ResultRepository
) : ViewModel() {
    private val _result = MutableLiveData<String>()
    val result: LiveData<String> get() = _result

    private val _progressVisible = MutableLiveData<Boolean>(false)
    val progressVisible: LiveData<Boolean> get() = _progressVisible

    override fun onCleared() {
        super.onCleared()
        resultRepository.cleanUp()
    }

    fun getNextResult() {
        resultRepository.getNextResult(
            onStart = {
                _progressVisible.value = true
            },
```

```
            onError = {
                _result.value = "Failed to get result."
                _progressVisible.value = false
            },
            onResult = {
                _result.value = it
                _progressVisible.value = false
            }
        )
    }
}
```

Repozitorij je odgovoran za dostavu podataka ViewModelu i prikazan je primjerom 2.13. Prima tri parametra: `resultApi` odgovoran za dohvaćanje podataka s mreže, `executorService` odgovoran za izvršavanje asinkronih zadataka i `callbackHandler` odgovoran za pozivanje povratnih poziva na određenoj dretvi (u ovom slučaju glavnoj). Njegova metoda `getNextResult` odgovorna je za ne-blokirajuće dohvaćanje rezultata od `resultApi` i za pozivanje proslijeđenih povratnih poziva. Osim toga, sprema svaki podnijeti zahtjev u listu kako bi ih mogao otkazati u metodi `cleanUp`. Ovime se postiže:

- Fleksibilnost pri izvršavanju asinkronih zadataka.
- Otkazivanje asinkronih zadataka.

Primjer koda 2.13: *Repozitorij u MVVM arhitekturi*

```
class DefaultResultRepository(
    private val resultApi: ResultApi,
    private val executorService: ExecutorService,
    private val callbackHandler: Handler
) : ResultRepository {
    private val futures = mutableListOf<Future<*>>()

    override fun getNextResult(
        onStart: () -> Unit,
        onError: (Throwable) -> Unit,
        onResult: (String) -> Unit
    ) {
        val future = executorService.submit {
            callbackHandler.post { onStart() }
            try {
                val result = resultApi.fetchResult()
                callbackHandler.post { onResult(result) }
            } catch (t: Throwable) {
                callbackHandler.post { onError(t) }
            }
        }
        futures.add(future)
    }

    override fun cleanUp() {
        futures.forEach {
            it.cancel(false)
        }
        futures.clear()
    }
}
```

Konkretni `ExecutorService`, čije sučelje koristi repozitorij, definiran je pomoću okvira za ubrizgavanje ovisnosti. Ovo je dobra praksa jer repozitorij ne bi trebao imati znanje o tome kako će se zadatak izvršiti, na koliko dretvi, kakve prioritete imaju te dretve i slično. Što se njega tiče, može biti implementiran i tako da izvršava sve na pozivajućoj dretvi. Isto

tako i s `callbackHandler`om. Koin modul s definiranim ovisnostima prikazan je primjerom 2.14. Može se vidjeti da u trenutnoj implementaciji repozitorij dobiva `ExecutorService` koji je skup dretvi stalne veličine jednak broju dostupnih obradbenih jedinica i kao `callbackHandler` dobiva `Handler` spojen na `Looper` glavne dretve. Ovime se postiže:

- Fleksibilnost i testabilnost jer se konkretni objekti mogu proizvoljno mijenjati.
- Istodobno izvođenje i iskorištenost svih obradbenih jedinica.
- Dretvenu sigurnost (s pretpostavkom da je proslijeđen prikladan `callbackHandler`).

Primjer koda 2.14: *Koin modul u MVVM arhitekturi*

```
private val defaultExecutorServiceQualifier = named("default")
private val mainHandlerQualifier = named("main")

val mainModule = module {
    single<ResultApi> {
        FakeResultApi()
    }
    factory<ResultRepository> {
        FakeResultRepository(
            resultApi = get(),
            executorService = get(defaultExecutorServiceQualifier),
            callbackHandler = get(mainHandlerQualifier)
        )
    }
    viewModel {
        MainActivityViewModel(get())
    }
    single<ExecutorService>(defaultExecutorServiceQualifier) {
        Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors())
    }
    single(mainHandlerQualifier) {
        Handler(Looper.getMainLooper())
    }
}
```

2.6. RxJava

U poglavlju 2.5 asinkrone operacije bile su implementirane pomoću dretvi, a njihovi su se rezultati prosljeđivali povratnim pozivima u kojima su ih pozivajuće komponente mogle obraditi. Ovo je jedan od osnovnih oblika reaktivnog programiranja kojem je fokus reagirati na promjene u obliku novih podataka, događaja i sl. [31, str.1]. Povratni pozivi predstavljaju reaktivno programiranje na imperativan način koji je pogodan za jednostavne slučajeve, ali povećanjem kompleksnosti koda teži problemu koji se naziva "paklom povratnih poziva" (engl. *callback hell*). Primjer takvog pakla je slučaj kad je potrebno izvršiti lanac asinkronih operacija gdje svaka ovisi o rezultatu prijašnje. Tad se povratni pozivi trebaju ulančavati čime kod postaje kompleksan i nepregledan, kao što je opisano u [32].

Suprotni način imperativnom je deklarativni način programiranja. On je usredotočen na rezultat posla kojeg se treba obaviti, a ne na njegove sastavne korake. Razliku između njih prikazuje primjer 2.15. Obje metode pretvaraju listu malih slova u riječ velikih slova. Razlika je u tome što se imperativni način bavi rješavanjem problema točnom implementacijom algoritma

dok deklarativni način apstrahira implementacijske detalje i usredotočuje se na jednostavno i čitko sučelje.

Primjer koda 2.15: *Razlika između imperativnog i deklarativnog programiranja*

```
fun main() {
    val letters = listOf("a", "b", "c", "d", "e", "f")

    val uppercaseWordImperative = toUppercaseWordImperative(letters)
    val uppercaseWordDeclarative = toUppercaseWordDeclarative(letters)

    println(uppercaseWordImperative == uppercaseWordDeclarative) // true
}

fun toUppercaseWordImperative(letters: List<String>): String {
    val builder = StringBuilder()
    for (letter in letters) {
        builder.append(letter.toUpperCase())
    }
    return builder.toString()
}

fun toUppercaseWordDeclarative(letters: List<String>): String {
    return letters.map { it.toUpperCase() }.joinToString(separator = "")
}
```

Reaktivno programiranje u vidu reaktivnih ekstenzija predstavlja apstrakciju koja omogućava deklarativno pisanje asinkronih, događajima uvjetovanih (engl. *event-driven*) programa bez brige o podležećim mehanizmima za dretvenu sigurnost, blokiranje, integritet podataka i ostalo [31, str.2]. Zasniva se na promotrivim tokovima (engl. *Observable streams*) i na propagiranju njihovih promjena (događaja) svojim promatračima. Implementacije reaktivnih ekstenzija postoje za mnoštvo popularnih jezika kao C++, JavaScript, Python, Scala, Swift itd. Implementacija u Javi zove se RxJava [32]. Ovo će poglavlje pokriti mogućnosti asinkronosti s RxJavom na Androidu, česte probleme pri implementaciji takve programske podrške i preporučenu metodu njezinog korištenja.

2.6.1 Promotrivi tok

Reaktivne ekstenzije zasnovane su na oblikovnim obrascima Promatrač i Iterator. Promatrač je obrazac kojemu je svrha definirati zavisnost jedan-na-više između objekata tako da kad se stanje jednog promijeni, ostali budu obaviješteni i ažurirani automatski [33, str.326]. Ključni elementi ovog obrasca su promatrač i subjekt. Subjekt je element kojega se promatra i on čuva listu svojih promatrača koje obavještava o svakoj promjeni njegovog unutarnjeg stanja. Ovaj model se još naziva i objavi-pretplati (engl. *publish-subscribe*) model jer subjekt objavljuje promjene svojeg stanja na koje se promatrači pretplaćuju čime bivaju ažurirani bez ručne provjere promjene stanja.

Iterator je obrazac kojemu je svrha apstrahirati slijedni pristup elementima kolekcije bez otkrivanja njezine unutarnje strukture [33, str.289]. Svrha ovog obrasca je odvojiti pohranu podataka u kolekciji od metode pristupa tim podacima za što odgovornost preuzima Iterator. Njegovo sučelje, između ostalih metoda, sadrži metodu za dohvaćanje sljedećeg elementa kolekcije. Korisnik Iteratora ručno poziva tu metodu i tako dohvaća (povlači, engl. *pull-based*) nove elemente.

Promotrivi tok se može gledati kao spoj Promatrača i Iteratora zasnovanog na guranju (engl. *push-based*). Umjesto da korisnici Iteratora ručno pozivaju njegove metode za povlačenje elemenata, oni se pretplaćuju na njega i dobivaju nove podatke kako pristižu. Implementacija promotrivog toka se u RxJavi zove `Observable`, a promatrači (u daljnjem tekstu pretplatnici) se pretplaćuju na njega metodom `subscribe` kojoj moraju proslijediti tri povratna poziva:

- `onNext` – poziva se svaki put kad tok emitira novi događaj.
- `onError` – pozove se kad u toku dođe do iznimke ili nekog drugog stanja od kojeg se ne može oporaviti. Nakon ovog poziva, `onNext` se više neće pozivati. Daje mogućnost oporavka od grešaka na jednom mjestu bez žongliranja višestrukim `try-catch` blokovima.
- `onComplete` – pozove se nakon zadnjeg `onNext` poziva u slučaju da nije pozvan `onError`. Označava završetak toka.

Na promotrivom toku se mogu primijeniti razni operatori koji ga transformiraju, slično Kotlin operatorima za rad s kolekcijama. Glavna je razlika u tome što su svi elementi Kotlin kolekcije dostupni u trenutku poziva operatora. Stvaranjem promotrivog toka i pozivom njegovih operatora zapravo se definira lanac transformacija za svakog od buduće emitiranih događaja, koliko god ih bude bilo i kad god dođu. Konačni, transformirani događaj dospijeva u `onNext` povratni poziv gdje ga pretplatnik može obraditi. Primjeri 2.16 i 2.17 prikazuju ovu sličnost. Oba primjera filtriraju brojeve manje od 3, množe ih s 2 te ih ispisuju na standardni izlaz.

Primjer koda 2.16: *Primjer jednostavnih operatora nad Kotlin kolekcijom*

```
listOf(1, 2, 3, 4, 5)
    .filter { it <= 3 }
    .map { it * 2 }
    .forEach {
        println("Number: $it")
    }
```

Primjer koda 2.17: *Primjer jednostavnog Rx promotrivog toka s operatorima*

```
Observable.fromIterable(listOf(1, 2, 3, 4, 5))
    .filter { it <= 3 }
    .map { it * 2 }
    .subscribeBy(
            onNext = {
                println("Number: $it")
            },
            onError = {
                println("An error occurred.")
            },
            onComplete = {
                println("All done!")
            }
    )
```

Osim navedenih operatora `filter` i `map`, RxJava pruža mnoštvo operatora čiji se popis može vidjeti na [34]. Kako Kotlin već ima operatore za rad s kolekcijama slične RxJavi, ona se ne koristi ako je potrebno samo raditi s običnim kolekcijama. Glavna motivacija korištenja RxJave je deklarativna i jednostavna implementacija asinkronosti i istodobnosti.

2.6.2 Raspoređivači, istodobnost i asinkronost

RxJava prema zadanim postavkama izvršava poslove promotrivog toka sinkrono s dretvom na kojoj se pretplatnik pretplatio na njega [35]. To je i jedna od smjernica u programiranju reaktivnih ekstenzija; promotrivi tokovi se trebaju implementirati sinkrono tako da su odgovorni samo za logiku generiranja događaja i pri tome pružiti pretplatnicima mogućnost istodobnog raspoređivanja ako to žele [31, str.153]. Odvajanjem istodobnosti od generiranja događaja dobiva se na fleksibilnosti, testabilnosti i jednostavnosti konačnog programa. Način izvršavanja kontrolira se raspoređivačima (engl. *schedulers*) koji se promotrivom toku mogu postaviti `subscribeOn` i `observeOn` operatorima. Namjerno je rečeno "način izvršavanja" jer se raspoređivačima ne mijenjaju samo dretve. Ovisno o odabranom raspoređivaču, dijelovi promotrivog toka mogu se izvršavati na jednoj pozadinskoj dretvi, skupu dretvi, sinkrono s pozivajućom dretvom, bez promjena itd. Raspoređivač kao takav je odgovoran samo za definiciju radnika (`Worker`) koji izvršava predani posao čime je moguće definirati razne oblike istodobnosti. RxJava pruža gotove raspoređivače kojima se može pristupiti klasama `Schedulers` i `AndroidSchedulers` (iz RxAndroid biblioteke [36]) i njezinim metodama tvornicama:

1. `io` – raspoređivač koji raspoređuje poslove na neograničenom broju ponovno-iskoristivih dretvi. Pogodan je za blokirajuće IO operacije kao rad s bazom podataka ili datotečnim sustavom, dohvaćanje udaljenih resursa i sl.

2. `computation` – raspoređivač koji raspoređuje poslove na onoliko ponovno-iskoristivih dretvi koliko ima dostupnih obradbenih jedinica. Pogodan je za neblokirajuće, obradbeno zahtjevne poslove.

3. `mainThread` – raspoređivač iz `AndroidSchedulers` biblioteke koji dostavlja poslove na izvršavanje glavnoj dretvi.

4. `newThread` – raspoređivač koji stvara novu dretvu za svaki posao. Nije preporučen jer nema mehanizam za ponovno korištenje stvorenih dretvi, a stvaranje nove dretve je skupo. Osim toga, stvaranjem prekomjerne količine dretvi može doći do usporenja programa i grešaka zbog manjka radne memorije. Zbog toga je skoro uvijek bolje koristiti nekog od prethodno navedenih raspoređivača.

5. `single` – raspoređivač koji slijedno raspoređuje svaki posao na jednoj pozadinskoj dretvi. Koristan ako je redoslijed izvršavanja poslova bitan.

6. `trampoline` – raspoređivač koji raspoređuje poslove na pozivajućoj dretvi tako da sljedeći posao može započeti tek kad svi prethodni završe.

7. `from` – metoda koja stvara raspoređivač koji je omotač (engl. *wrapper*) oko predanog `Executor`a i koristi njegove dretve za izvršavanje poslova.

Operator subscribeOn Pretplatom na promotrivi tok prvo se poziva kod koji ga stvara, odnosno koji je odgovoran za emitiranje njegovih događaja. Raspoređivač stvaranja promotrivog toka kontrolira se `subscribeOn` operatorom. Ako se nakon njega ne postavi drugi raspoređivač, cijeli će lanac operatora kao i povratni pozivi pretplatnika biti izvršeni na njemu. Višestruki pozivi neće imati utjecaj jer se uvijek primjenjuje poziv najbliži izvornom toku. Primjer 2.18 prikazuje stvaranje promotrivog toka i utjecaj kojeg operator `subscribeOn` ima na dretvu na kojoj se on stvara. Unatoč višestrukim pozivima `subscribeOn` operatora stvaranje toka, `map` operator i `onNext` povratni poziv izvršavaju se na onom raspoređivaču koji je prvi postavljen. Isto se događa i s `filter` operatorom, iako je `subscribeOn` pozvan nakon njega.

Primjer koda 2.18: *Način rada subscribeOn operatora*

```
val observable = Observable.create<Int> { emitter ->
        println("Thread in onSubscribe: ${Thread.currentThread().name}")
        Thread.sleep(1000)
        emitter?.onNext(1)
        Thread.sleep(1000)
        emitter?.onNext(2)
        emitter?.onComplete()
    }
observable
    .filter {
        println("Thread in filter: ${Thread.currentThread().name}")
        it > 0
    }
    .subscribeOn(Schedulers.computation())
    .subscribeOn(Schedulers.io())
    .map {
        println("Thread in map: ${Thread.currentThread().name}")
        it * 2
    }
    .subscribeOn(Schedulers.single())
    .subscribe {
        println("Thread in onNext: ${Thread.currentThread().name}")
    }
// Output:
// Thread in onSubscribe: RxComputationThreadPool-1
// Thread in filter: RxComputationThreadPool-1
// Thread in map: RxComputationThreadPool-1
// Thread in onNext: RxComputationThreadPool-1
// Thread in filter: RxComputationThreadPool-1
// Thread in map: RxComputationThreadPool-1
// Thread in onNext: RxComputationThreadPool-1
```

Ovime je riješen problem prebacivanja na pozadinsku dretvu (ili više njih, ovisno o raspoređivaču), ali javlja se problem naknadnog prebacivanja. Kao što je primjer pokazao, `subscribeOn` operatorom ne može se naknadno promijeniti dretva na kojoj se operatori ili povratni pozivi izvršavaju. Može se samo promijeniti za čitavi tok, što je nedovoljno ako je potrebno, primjerice, napraviti mrežni poziv na `io` raspoređivaču i zatim preslikati dobivene podatke na `computation` raspoređivaču. Specifično za Android, nakon tih je operacija često potrebno i prikazati preslikane podatke za što je potrebno prebacivanje na glavnu dretvu pomoću `mainThread` raspoređivača.

Operator observeOn Naknadnu promjenu raspoređivača radi `observeOn` operator koji postavlja raspoređivača na kojem se izvršavaju nizvodni operatori i povratni pozivi. Za razliku od `subscribeOn`, mjesto na kojem se poziva je bitno i svaki poziv ima utjecaj na nizvodno

raspoređivanje. Primjer 2.19 prikazuje stvaranje promotrivog toka i utjecaj `observeOn` operatora na njega. Bitno je primijetiti da se stvaranje promotrivog toka izvršava na pozivajućoj (ovdje glavnoj) dretvi, odnosno da `observeOn` operator nema utjecaj na taj dio koda. Osim toga, svakim pozivom `observeOn` operatora mijenja se dretva na kojoj se izvršavaju naknadni operatori, a zadnji poziv prije pretplate definira i na kojoj će dretvi biti pozvani povratni pozivi pretplatnika.

Primjer koda 2.19: *Način rada observeOn operatora*

```
1  val observable = Observable.create<Int> { emitter -> ...}
2  observable
3      .filter {
4          println("Thread in filter: ${Thread.currentThread().name}")
5          it > 0
6      }
7      .observeOn(Schedulers.computation())
8      .map {
9          println("Thread in map: ${Thread.currentThread().name}")
10         it * 2
11     }
12     .observeOn(Schedulers.io())
13     .subscribe {
14         println("Thread in onNext: ${Thread.currentThread().name}")
15     }
16 // Output:
17 // Thread in onSubscribe: main
18 // Thread in filter: main
19 // Thread in map: RxComputationThreadPool-1
20 // Thread in onNext: RxCachedThreadScheduler-1
21 // Thread in filter: main
22 // Thread in map: RxComputationThreadPool-1
23 // Thread in onNext: RxCachedThreadScheduler-1
```

Kombinacijom operatora `subscribeOn` i `observeOn` u potpunosti se kontroliraju raspoređivači na kojima se izvršava posao promotrivog toka. Primjer 2.20 prikazuje ova dva operatora zajedno i kako utječu jedan na drugog. Može se vidjeti kako raspoređivač postavljen `subscribeOn` operatorom vrijedi do prvog `observeOn` poziva nakon čega vrijedi njegov raspoređivač. Kad linije 4 ne bi bilo, filtriranje bi se također izvršilo na `io` raspoređivaču nakon čega bi se preslikavanje izvršilo na `computation` raspoređivaču kako je definirano linijom 14.

Primjer koda 2.20: *Način zajedničkog rada subscribeOn i observeOn operatora*

```
1  val observable = Observable.create<Int> { emitter -> ...}
2  observable
3      .subscribeOn(Schedulers.io())
4      .observeOn(Schedulers.single())
5      .filter {
6          println("Thread in filter: ${Thread.currentThread().name}")
7          it > 0
8      }
9      .observeOn(Schedulers.computation())
10     .map {
11         println("Thread in map: ${Thread.currentThread().name}")
12         it
13     }
14     .observeOn(AndroidSchedulers.mainThread())
15     .subscribe {
16         println("Thread in onNext: ${Thread.currentThread().name}")
17     }
18 // Output:
19 // Thread in onSubscribe: RxCachedThreadScheduler-1
20 // Thread in filter: RxSingleScheduler-1
21 // Thread in map: RxComputationThreadPool-1
22 // Thread in onNext: main
23 // Thread in filter: RxSingleScheduler-1
```

```
24 // Thread in map: RxComputationThreadPool-1
25 // Thread in onNext: main
```

RxJava sve svoje mogućnosti istodobnosti izlaže kroz raspoređivače i operatore `subscribeOn` i `observeOn` [31, str.159]. Kao što primjeri pokazuju, a i kakva je priroda deklarativnog programiranja, pri tome nije potrebno voditi računa o životnom ciklusu dretvi, njihovoj komunikaciji, unutarnjim mehanizmima i sl. Potrebno je samo (informirano i pažljivo) navesti raspoređivač na kojemu se nešto treba izvršiti. RxJava je postala izuzetno popularna među Android programerima [31, str.277] jer se njezinim korištenjem izbjegava pakao povratnih poziva i omogućava elegantno prebacivanje s dretve na dretvu bez većih glavobolja. Kombinacijom `subscribeOn` i `observeOn` operatora može se postići većina obrazaca asinkronosti na Androidu od kojih je vrlo popularan obrazac dohvaćanja podataka s Interneta (`io` raspoređivač), preslikavanje tih podataka u prikladan oblik za pogled (`computation` raspoređivač) te prikaz preslikanih podataka na sučelju (`mainThread` raspoređivač). Ako je potrebno nešto više od toga, RxJava omogućava i pravo istodobno izvršavanje.

2.6.3 Paralelni promotrivi tokovi

Primjeri do sada bavili su se implementacijom asinkronosti u RxJavi gdje se u slijednom toku prebacivalo izvršavanje poslova s jedne na drugu dretvu. Ovo je dovoljno za jednostavne upotrebe na Androidu gdje je potrebno dohvatiti podatke s jednog mjesta (baza podataka, Internet, datotečni sustav), transformirati ih te prikazati na sučelju. Za kompleksnije upotrebe koje zahtijevaju istodobnu obradu na više pozadinskih dretvi ovaj model je nedostatan. Primjerice, ako je u nekoj aplikaciji potrebno dohvatiti podatke s više nezavisnih API ruta kako bi se stvorilo stanje trenutnog prikaza, to se može jednostavno napraviti lančanjem API poziva. Kad stigne odgovor jedne rute, pozove se druga i tako dalje dok se lanac ne završi. To bi imalo smisla da je svaki API poziv ovisan o prethodnom, ali jer su nezavisni, bilo bi efikasnije pokrenuti svakog istodobno i na kraju kombinirati njihove rezultate. Ovakve i slične operacije mogu se postići operatorima `flatMap`, `zip`, `merge` i nekim drugim transformacijskim i kombinacijskim operatorima [34].

Operator flatMap Transformacijski operator koji emitirane događaje pretvara u promotrive tokove čije događaje usmjerava u jedan promotrivi tok. Pogodan je za podatkovnu dekompoziciju gdje se neku operaciju treba primijeniti na više podataka istodobno, a zatim ih sve opet spojiti u isti tok. Primjer 2.21 prikazuje jedan takav slučaj. Bitno je primijetiti poziv operatora `subscribeOn` unutar `flatMap`a jer upravo on omogućava izvršavanje svih stvorenih tokova istodobno. U primjerima do sada, ime dretvi `io` raspoređivača bilo je `RxCachedThreadScheduler-1` gdje `1` označava broj dretve. Sad ima više dretvi koje se izvršavaju istodobno na što ukazuju imena čiji se sufiksi kreću od `1` do `5` kao i ukupno proteklo vrijeme koje iznosi nešto više od jedne sekunde iako je stvoreno pet promotrivih tokova od kojih svaki blokira svoju dretvu jednu sekundu (kod slijednog izvršavanja bi očekivana vrijednost bila nešto više od pet sekundi). Treba primijetiti i da redoslijed događaja više nije slijedan. Za to se umjesto `flatMap` operatora može

koristiti `concatMapEager` koji se ponaša isto, ali čuva redoslijed emitiranih događaja.

Primjer koda 2.21: *Istodobnost pomoću operatora flatMap*

```
val timestampBeforeSubscription = System.currentTimeMillis()
Observable.just(1, 2, 3, 4, 5)
    .flatMap {
        Observable.create<Int> { emitter ->
            Thread.sleep(1000)
            emitter.onNext(it)
            emitter.onComplete()
        }
            .subscribeOn(Schedulers.io())
            .map { inner ->
                println("Thread in inner map: ${Thread.currentThread().name}")
                inner * 2
            }

    }
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(
        { println("onNext: $it") },
        { /*Handle errors*/ },
        { println("Elapsed time: ${System.currentTimeMillis() -
            ↪ timestampBeforeSubscription}") }
    )
// Output:
// Thread in inner map: RxCachedThreadScheduler-5
// Thread in inner map: RxCachedThreadScheduler-1
// Thread in inner map: RxCachedThreadScheduler-2
// Thread in inner map: RxCachedThreadScheduler-3
// Thread in inner map: RxCachedThreadScheduler-4
// onNext: 10
// onNext: 2
// onNext: 4
// onNext: 6
// onNext: 8
// Elapsed time: 1039
```

Operator zip Kombinacijski operator koji kombinira prve sljedeće događaje sastavnih promotrivih tokova u jedan s pravilom da se kombinacijska funkcija poziva tek kad je svaki od njih emitirao sljedeći događaj. Ovaj se operator može koristiti kad je potrebno istodobno napraviti više blokirajućih poziva i reagirati tek kad su svi gotovi, slično operaciji `join` kod običnih dretvi gdje se prije nastavka čeka da svaka dretva završi. Primjer 2.22 prikazuje takav slučaj. Kao kod `flatMap`, glavna stvar koju treba zapamtiti je `subscribeOn` na sastavnim promotrivim tokovima bez kojih se oni ne bi izvršavali istodobno. Ispis pokazuje da se sastavni tokovi izvršavaju istodobno na nezavisnim dretvama `io` raspoređivača, a dodatni dokaz je proteklo vrijeme koje je jednako vremenu trajanja najsporijeg toka (pet sekundi), a ne zbroju trajanja svih (7.5 sekundi).

Primjer koda 2.22: *Istodobnost pomoću operatora zip*

```
val networkCall = Observable.create<String> { emitter ->
    Thread.sleep(5000)
    emitter.onNext("burger")
    emitter.onComplete()
}
val databaseCall = Observable.create<Int> { emitter ->
    Thread.sleep(2500)
    emitter.onNext(2)
    emitter.onComplete()
}
val timestampBeforeSubscription = System.currentTimeMillis()
Observable.zip(
```

```
    networkCall.subscribeOn(Schedulers.io())
        .map {
            println("Thread in network map: ${Thread.currentThread().name}")
            "Ham$it"
        },
    databaseCall.subscribeOn(Schedulers.io())
        .map {
            println("Thread in database map: ${Thread.currentThread().name}")
            it * 12
        }
) { networkResponse, databaseResponse ->
    "$networkResponse combined with $databaseResponse"
}
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(
        { println("onNext: $it") },
        { /*Handle errors*/ },
        { println("Elapsed time: ${System.currentTimeMillis() -
            ↪ timestampBeforeSubscription}") }
    )
// Output:
// Thread in database map: RxCachedThreadScheduler-2
// Thread in network map: RxCachedThreadScheduler-1
// onNext: Hamburger combined with 12
// Elapsed time: 5024
```

Operator merge Kombinacijski operator koji kombinira događaje sastavnih promotrivih tokova u jedan. Sličan je `flatMap`u, a razlika je u tome što `flatMap` osim spajanja više promotrivih tokova u jedan i *stvara* te promotrive tokove, dok ih `merge` dobiva kao ulazne varijable. Koristan je kad je potrebno istodobno izvršavanje više nezavisnih promotrivih tokova i rukovanje njihovim događajima s jednog mjesta umjesto odvojenom pretplatom na svaki tok. Primjerice, ako događaji tokova imaju zajedničko sučelje, mogu se kombinirati ovim operatorom što omogućava poziv zajedničkog ponašanja s jednog mjesta. Događaji će biti kombinirani tako da konačni promotrivi tok emitira događaje koji su tip zajedničkog sučelja. Primjer 2.23 prikazuje takav slučaj. Slično kao `flatMap` i `zip`, ključni element je `subscribeOn` poziv koji raspoređuje tokove istodobno.

Primjer koda 2.23: *Istodobnost pomoću operatora merge*

```
interface Vocal { fun makeSound() }
class Cat : Vocal {...}
class Dog : Vocal {...}
...
val clowder = Observable.create<Cat> { emitter ->
    Thread.sleep(1000)
    emitter.onNext(Cat())
    Thread.sleep(2000)
    emitter.onNext(Cat())
    emitter.onComplete()
}
val kennel = Observable.create<Dog> { emitter ->
    Thread.sleep(1500)
    emitter.onNext(Dog())
    emitter.onComplete()
}
Observable.merge(
    clowder.subscribeOn(Schedulers.io()),
    kennel.subscribeOn(Schedulers.io())
)
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe { vocal ->
        vocal.makeSound()
    }
// Output:
```

```
26 // Meow!
27 // Woof!
28 // Meow!
```

2.6.4 Otkazivanje pretplate

Iako RxJava ima jednostavan model definicije asinkronosti i istodobnosti, nije čaroban alat koji uklanja potrebu za sprječavanjem curenja memorije. U primjerima do sada moglo se vidjeti da se RxJava oslanja na lambda izraze (anonimne klase) kako bi stvorila nove promotrive tokove, nad njima primijenila operatore te definirala povratne pozive prilikom pretplate. Kao kod dretvi, ovo predstavlja problem na Androidu jer je često u tim lambda izrazima potrebno pristupiti elementima sučelja radi prikaza rezultata. Ako tada komponenta kao aktivnost ili fragment prođe kroz `onDestroy` metodu (npr. uslijed konfiguracijske promjene) dolazi do curenja memorije. Ovaj nedostatak imali bi svi dosadašnji primjeri kad bi se u njima pristupalo elementima sučelja, a rješenje je otkazivanje pretplate (engl. *dispose*) na promotrivi tok. Pretplata na promotrivi tok vraća jednokratni `Disposable` objekt s metodom `dispose` koja omogućava prekid pretplate [37, str.88]. Može ju se shvatiti kao reaktivnu analogiju metodama `Thread#interrupt`, `Future#cancel` ili `AsyncTask#cancel`. njezinim se pozivom postižu dvije stvari:

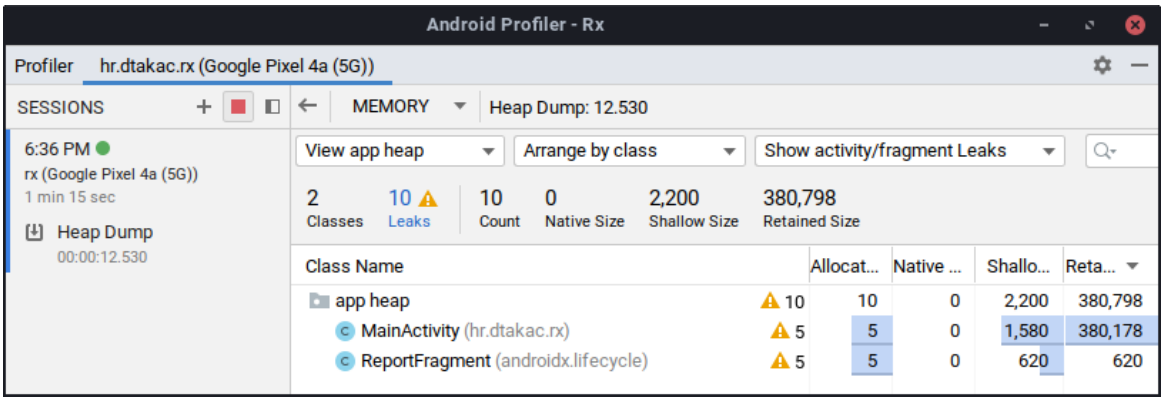
1. Odašiljaču (engl. *emitter*) promotrivog toka postavlja se zastavica `isDisposed` koja omogućava rani, kooperativni prekid emitiranja novih događaja. Prilikom stvaranja promotrivog toka iz nule (npr. `create` metodom) loša je praksa zanemariti ovu zastavicu. Nemarno je i potrošno nastaviti s emitiranjem događaja ako više nema pretplatnika kojima se oni mogu dostaviti. Analogna je zastavicama `isInterrupted` dretve ili `isCancelled` `Future`a i `AsyncTask`a.

2. Povratni poziv pretplatnika uklanja se iz popisa pretplatnika promotrivog toka, čime se uklanja i implicitna referenca na Android komponentu što sprječava curenje memorije. Ovo je analogno uklanjanju poruke iz reda poruka kod Android dijeljenja poruka.

Primjer 2.24 prikazuje čest slučaj curenja memorije. U povratnom pozivu pretplatnika pristupa se `tvText` elementu sučelja kojem se za vrijednost teksta postavlja proteklo vrijeme. Curenje memorije nastaje kad aktivnost prođe kroz `onDestroy` metodu jer promotrivi tok ne zna za taj događaj. Izostavljanjem otkazivanja, promotrivi tok nastavlja emitirati događaje, a pretplatnički povratni poziv ostaje u promotrivom toku. Time skupljač smeća ne može osloboditi memoriju aktivnosti i događa se curenje. Slika 2.7 prikazuje ispis memorijske hrpe nakon pet orijentacijskih promjena gdje se vidi da je memorija aktivnosti iscurila pet puta. Potrebno je naglasiti da ovdje nije bitno što je to promotrivi tok tipa `interval`. Isto bi se dogodilo s bilo kojim promotrivim tokom ako se negdje u njegovom lancu operatora pristupi elementu roditeljske komponente.

Primjer koda 2.24: *Curenje memorije u aktivnosti zbog ne-otkazane pretplate*

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        demonstrateObservableLeak()
    }

    private fun demonstrateObservableLeak() {
        Observable.interval(0, 1, TimeUnit.SECONDS)
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe {
                binding.tvText.text = it.toString()
            }
    }
}
```



Sl. 2.7: *Ispis memorijske hrpe nakon pet orijentacijskih promjena primjera 2.24*

Rješenje ovog problema slično je rješenju istog problema kod dretvi. Potrebno je otkazati posao i ukloniti povratni poziv koji drži referencu na komponentu. Oboje se postiže spremanjem rezultirajućeg jednokratnog objekta i njegovim otkazivanjem pri završetku života komponente. Ako se neka komponenta pretplaćuje na više promotrivih tokova, može za svakog stvoriti posebnu varijablu i onda ih na kraju života sve otkazati. Rastom broja promotrivih tokova ovo postaje nepraktično, pa se umjesto toga koristi kolekcija jednokratnih objekata `CompositeDisposable` koja omogućava zbirno dodavanje i otkazivanje svojih jednokratnih objekata (slično listi `Future` objekata iz primjera 2.13). Primjer 2.25 prikazuje potrebne promjene s kojima nema curenja memorije nakon završetka aktivnosti. Nakon pretplate, jednokratni objekt dodaje se kolekciji jednokratnih objekata koja se pri završetku aktivnosti otkazuje, čime interno otkazuje svakog od svojih jednokratnih objekata.

Primjer koda 2.25: *Točno rukovanje životnim ciklusom i promotrivim tokom*

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding
    private var disposables = CompositeDisposable()

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        demonstrateObservableProper()
    }

    private fun demonstrateObservableProper() {
        val disposable = Observable.interval(0, 1, TimeUnit.SECONDS)
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
```

```
14              .subscribe {
15                  binding.tvText.text = it.toString()
16              }
17          disposables.add(disposable)
18      }
19
20      override fun onDestroy() {
21          super.onDestroy()
22          disposables.dispose()
23      }
24  }
```

Osim curenja memorije, ignoriranje jednokratnog objekta lako uzrokuje i nepotrebno trošenje resursa. U primjeru 2.24 nakon pet konfiguracijskih promjena postoji pet promotrivih tokova koji zauzimaju pet dretvi `io` raspoređivača. Primjer 2.25 nema ovaj problem jer se svakom konfiguracijskom promjenom promotrivi tok otkazuje i resursi postaju slobodni za ponovno korištenje. Zbog ovih je nedostataka metoda `subscribe` označena `@CheckReturnValue` anotacijom (engl. *annotation*) koja dodatno upozorava programera da ne ignorira jednokratni objekt nakon pretplate. Najbolja praksa je sačuvati ga i otkazati kad više nije potreban.

2.6.5 Česti problemi i anti-obrasci

RxJava pruža deklarativnu apstrakciju asinkronosti, prebacivanja dretvi i rukovanja promotrivim tokovima. Njezinim korištenjem može doći do problema koji mogu uzrokovati zastajkivanje i nepotrebno korištenje resursa, ali i nečitak i neodrživ kod. Oni najčešći dani su u ovom poglavlju.

Ignoriranje životnog ciklusa i jednokratnog objekta Kako je navedeno u poglavlju 2.5, ignoriranje životnog ciklusa i mehanizama otkazivanja poslova asinkronih zadataka vodi nepotrebnom trošenju resursa te sporijoj i manje responzivnoj aplikaciji. Zato je potrebno čuvati reference na jednokratne objekte nakon pretplate i otkazati ih kad je to prikladno. Ako se promotrivi tokovi koriste u prikaznom sloju (npr. u aktivnosti kako bi se događaji klikova predstavili promotrivim tokom [38]) onda se pretplatu može otkazati u nekom od povratnih poziva životnog ciklusa (npr. `onDestroy`). Ako se promotrivi tokovi koriste za operacije koje trebaju nadživjeti komponente prikaznog sloja (npr. dohvaćanje podataka s API-ja) i njihove konfiguracijske promjene, onda se treba koristiti ViewModel u kojem se nalazi kolekcija jednokratnih objekata koje se može otkazati u `onCleared` metodi.

Apstrakcija observeOn i subscribeOn operatora Promotrivi tokovi reaktivnih ekstenzija osmišljeni su da, u svom zadanom obliku, budu agnostični po pitanju istodobnosti [31, str.144]. Svaki bi pretplatnik trebao individualno odlučiti je li mu, i u kojoj mjeri, potrebno istodobno izvršavanje poslova promotrivog toka. Na ovaj način promotrivi tokovi održavaju jednostavnost, a pružaju fleksibilnost u slučaju da je nekom pretplatniku potrebno istodobno izvršavanje. Programiranjem Android aplikacija (i aplikacija s korisničkim sučeljem općenito), često je potrebno prebaciti izvršavanje blokirajuće operacije na pozadinsku dretvu te prikazati rezultat iz glavne dretve. Takvih poziva u jednoj Android aplikaciji može biti puno, a

za većinu je potrebna ista kombinacija operatora: `subscribeOn` koji postavlja izvršavanje promotrivog toka na `io` raspoređivač i `observeOn` koji postavlja izvršavanje povratnih poziva na raspoređivač glavne dretve. Motivirani željom smanjenja količine dupliciranog koda, programeri mogu apstrahirati ove pozive na razne načine.

Prvi je način Kotlin funkcija proširenja (engl. *extension function*) koja omogućava proširenje klase s novom mogućnosti bez nasljeđivanja ili oblikovnih obrazaca kao Dekorater [39]. Napravi se funkcija proširenja kao u primjeru 2.26 koja se onda poziva na bilo kojem mjestu gdje se koriste promotrivi tokovi. Na projektima gdje radi više ljudi se onda zna poticati programere da "samo pozovu `defaultSchedulers`" na svakom promotrivom toku. Za vrlo jednostavne slučajeve gdje je potrebno samo pozvati blokirajuću operaciju i prikazati njezine rezultate ovo je dostatno, ali povećanjem kompleksnosti pojavljuje se nekoliko problema:

1. Novi programeri na projektu mogli bi pomisliti da je za sve promotrive tokove na koje se pretplaćuju dovoljno pozvati ovu funkciju i njihove su muke s asinkronosti gotove, a to je daleko od istine. Što ako je nekad u budućnosti potrebno dodati zahtjevnu `map` operaciju na tok? Tada će se i ona izvršavati na glavnoj dretvi, što će ju usporiti, a uzrok (implicitni poziv `observeOn` za prebacivanje na glavnu dretvu) apstrahiran je funkcijom proširenja.

2. Što znači ime funkcije `defaultSchedulers` ili njemu slična imena? Koji su to raspoređivači? Koji im je redoslijed? Rješenje maglovitosti ove funkcije je preimenovati ju u `subscribeOnIoObserveOnMain` ili dokumentirati što ona točno radi, ali oboje promašuje poantu apstrakcije raspoređivača.

3. Uzimanjem prošle dvije točke u obzir, može se u posebnim, jednostavnim slučajevima koristiti ova funkcija, a u drugima ne, ali to nije dosljedno i može uzrokovati nečitak i neodrživ kod.

Primjer koda 2.26: *Apstrakcija raspoređivačkih operatora funkcijom proširenja*

```
fun <T> Observable<T>.defaultSchedulers(): Observable<T> {
    return subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.mainThread())
}
...
repository.networkCallObservable
    .defaultSchedulers()
    .subscribe {...}
```

Drugi je način izlaganje promotrivog toka metodom repozitorija (ili neke druge komponente) gdje ta metoda implicitno primjenjuje raspoređivačke operatore. Ovime se smanjuje fleksibilnost pretplatnika jer ne mogu napraviti svoju odluku o istodobnosti izvršavanja promotrivog toka. Ako nekom pretplatniku više odgovara drugačiji raspoređivač, ne može ga postaviti gdje želi. Implicitan poziv `subscribeOn` operatora može biti prihvatljiv jer je istodobnost pri generiranju događaja odgovornost repozitorija, ali poziv `observeOn` operatora ne. Čist i reaktivan način je ostaviti ovaj odabir pretplatnicima i ukloniti implicitno raspoređivanje [31, str.153].

Provlačenje toka od modela do prikaznog sloja U ovom se anti-obrascu promotrivi tok iz repozitorija provlači kroz ViewModel i zatim završava u prikaznom sloju koji se pretplaćuje na rezultate. Primjer takvog koda prikazan je u 2.27. Time se krši sigurnost ViewModela s obzirom na životni ciklus jer promotrivi tokovi nisu svjesni životnog ciklusa Android komponenti kao `LiveData`. Osim toga, krši se razdvajanje odgovornosti (engl. *separation of concerns*) jer je prikazni sloj sada odgovoran za odabir raspoređivača, pretplatu na događaj iz modela i otkazivanje poslova, što bi sve trebale biti odgovornosti ViewModela. Naravno, promotrivi se tokovi smiju stvarati u prikaznom sloju onda kad se to tiče prikaznog sloja. Primjerice, nema ništa krivo u stvaranju promotrivog toka koji omata događaje klikova na gumb ili promjene teksta komponente `EditText` i rukovanja rezultirajućim jednokratnim objektima unutar prikaznog sloja.

Primjer koda 2.27: *Provlačenje toka od repozitorija do prikaznog sloja*

```
class Repository(private val api: Api) {
    fun getNetworkCall(): Observable<Result> {
        return api.getNetworkCallObservable()
    }
}

class ViewModel(private val repository: Repository) {
    val networkCall = repository.getNetworkCall()
}

class ExampleActivity: AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        viewModel.networkCall
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe {
                binding.tvText.text = it.toString()
            }
    }
}
```

Fiksiranje raspoređivača Raspoređivači su se u dosadašnjim primjerima postavljali izravnim referencama na konkretne raspoređivače, odnosno bili su fiksirani (engl. *hard-coded*). To znači da se ne mogu promijeniti izvana, što predstavlja problem jer se tijekom testiranja moraju koristiti testni dispečeri. Zato je preporučeno ubrizgavati dispečere u komponente koje ih koriste, što će biti prikazano u poglavlju 2.6.6.

RxJava kao zlatni čekić Iako RxJava pruža sažeto, deklarativno i čitko sučelje, njezina krivulja učenja vrlo je strma [40, str.1]. Početnici trebaju značajno promijeniti imperativni način razmišljanja na deklarativan koji je zasnovan na promotrivim tokovima i apstrahiranim mehanizmima istodobnosti. U velikim projektima s kompleksnim potrebama međudretvene komunikacije, ulančavanja asinkronih zadataka, čestih prebacivanja dretvi i predstavljanja podataka u obliku tokova, njezino korištenje je opravdano. Implementacija istih funkcionalnosti konvencionalnim mehanizmima asinkronosti u "suhoj" Javi paklen je zadatak pun isprepletenih povratnih poziva. Međutim, za jednostavne slučajeve treba se zapitati je li potrebno uvoditi kompleksnost i veličinu biblioteke kao RxJava u projekt? Ako je potrebno napraviti nekoliko

jednostavnih, nezavisnih mrežnih zahtjeva ili operacija s bazom podataka, to se jednako lako može napraviti običnim dretvama bez potrebe za učenjem potpuno nove paradigme programiranja te povećanjem veličine aplikacije i kompleksnosti pisanja, čitanja i održavanja koda.

2.6.6 Preporučeni obrazac korištenja

Izvršena je analiza prednosti i nedostataka implementacije asinkronosti pomoću RxJave na Androidu. U ovom dijelu daje se njezin preporučeni obrazac korištenja u prikaznom i prezentacijskom sloju. Aplikacija ima iste funkcionalnosti kao ona iz poglavlja 2.5.6 i koristi MVVM arhitekturni obrazac. Osim ovdje navedenih promjena, kod je isti. Repozitorij, ViewModel i aktivnost su prepravljeni i sada koriste promotrive tokove i raspoređivačke operatore kako bi postigli asinkronost. Primjer 2.28 prikazuje aktivnost u kojoj se umjesto slušatelja klikova (engl. *click listener*) koristi promotrivi tok `clicks` iz RxBinding biblioteke koja pruža omotače za Androidov API koji je zasnovan na povratnim pozivima [38]. Ova promjena je napravljena samo kako bi se pokazalo da je sasvim sigurno i točno koristiti promotrive tokove u prikaznom sloju ako se pritom pazi na otkazivanje pretplate. Sučelje ViewModela prema aktivnosti nije se promijenilo i stoga nije bilo potrebno mijenjati kod aktivnosti. ViewModel je morao promijeniti način na koji stavlja događaje u svoje `LiveData` objekte, ali aktivnost nije osjetila tu promjenu, što je jedna od glavnih prednosti arhitekturnih obrazaca. Jedna komponenta može drastično promijeniti svoju unutarnju implementaciju, a ostale nastavljaju kao da se ništa nije dogodilo.

Primjer koda 2.28: *Aktivnost iz poglavlja 2.11 prilagođena na RxJavu*

```
class MainActivity : AppCompatActivity() {
    ...
    private val viewModel: MainActivityViewModel by viewModel()
    private val disposables = CompositeDisposable()

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        passUserActionsToViewModel()
        observeViewModel()
    }

    override fun onDestroy() {
        super.onDestroy()
        disposables.dispose()
    }

    private fun passUserActionsToViewModel() {
        val disposable = binding.btnGetResult.clicks()
            .subscribe {
                viewModel.getResult()
            }
        disposables.add(disposable)
    }

    private fun observeViewModel() {
        viewModel.result.observe(this) {
            binding.tvResult.text = it
        }
        viewModel.progressVisible.observe(this) {
            if (it) {
                binding.piProgress.show()
            } else {
                binding.piProgress.hide()
            }
        }
    }
```

```
}
```

ViewModel je promijenjen tako da upravlja događajima promotrivog toka iz repozitorija i u ovisnosti o njima gura podatke u svoje `LiveData` objekte. Osim toga, nadjačava svoju `onCleared` metodu kako bi sigurno otkazao pretplatu na sve pokrenute tokove. Popratni operator `doOnSubscribe` poziva se u trenutku pretplate i analogan je `onStart` povratnom pozivu iz primjera 2.12. Skrivanje trake napretka prebacilo se iz `onError` i `onResult` u popratni operator `doOnEach` koji se poziva svakom emisijom događaja ili greške. Postavljanje teksta rezultata prebačeno je u `onNext` i `onError` povratne pozive. Ovakva je podjela reakcija na događaje promotrivog toka samo pitanje semantike i postoji mnogo različitih načina. Primjerice, pretplati su se mogli predati prazni lambda izrazi, a operacije prebaciti u popratne operatore `doOnSubscribe`, `doOnEach`, `doOnNext` i `doOnError`. Pritom je dobra misao vodilja u popratne operatore staviti kod koji se tiče popratnih efekata promotrivog toka (kao upravljanje trakom napretka), a u povratne pozive kod koji se tiče glavnog efekta (kao prikazivanje traženih podataka). Važno je primijetiti i da se raspoređivači ne referiraju izravno, nego kroz `SchedulerProvider` objekt. Tako se u produkcijskom kodu može koristiti zadani `SchedulerProvider` objekt s normalnim raspoređivačima (prikazan primjerom 2.30), a za testiranje se svi raspoređivači mogu postaviti na `TestScheduler`. Ovakva implementacija ViewModela ima sljedeće prednosti:

- Jednostavno rukovanje asinkronošću.
- Nema ručnog pisanja povratnih poziva.
- Omogućeno testiranje.
- Jednostavna i deklarativna implementacija posla kojeg se treba napraviti i reakcije na njegove događaje.

Primjer koda 2.29: *ViewModel iz primjera 2.12 prilagođen na RxJavu*

```
class MainActivityViewModel(
    private val resultRepository: ResultRepository,
    private val schedulerProvider: SchedulerProvider
) : ViewModel() {
    private val _result = MutableLiveData<String>()
    val result: LiveData<String>
        get() = _result
    private val _progressVisible = MutableLiveData<Boolean>(false)
    val progressVisible: LiveData<Boolean>
        get() = _progressVisible

    private val compositeDisposable = CompositeDisposable()

    override fun onCleared() {
        super.onCleared()
        compositeDisposable.dispose()
    }

    fun getResult() {
        val disposable = resultRepository.getResult()
            .subscribeOn(schedulerProvider.io)
            .observeOn(schedulerProvider.main)
            .doOnSubscribe {
                _progressVisible.value = true
            }
```

```
            .doOnEach {
                _progressVisible.value = false
            }
            .subscribe(
                { result ->
                    _result.value = result
                },
                { throwable ->
                    _result.value = "Failed to get result."
                }
            )
        compositeDisposable.add(disposable)
    }
}
```

Primjer koda 2.30: *Aplikacijski davatelj raspoređivača*

```
class AppSchedulerProvider : SchedulerProvider {
    override val main: Scheduler get() = AndroidSchedulers.mainThread()
    override val io: Scheduler get() = Schedulers.io()
    override val computation: Scheduler get() = Schedulers.computation()
}
```

Repozitorij je, u usporedbi s primjerom 2.13, vrlo jednostavan. On samo poziva metodu API usluge koja izlaže dohvaćanje podataka s Interneta u obliku promotrivog toka. Retrofit 2 ima podršku za promotrive tokove i radi na sličan način. Ako nije moguće koristiti biblioteku koja to već ima implementirano, moguće je napraviti omotač oko mrežnog poziva metodom `Observable#create`.

Primjer koda 2.31: *Repozitorij iz primjera 2.13 prilagođen na RxJavu*

```
class DefaultResultRepository(
    private val api: ResultApi
) : ResultRepository {
    override fun getResult(): Observable<String> {
        return api.fetchResult()
    }
}
```

2.7. Korutine u Kotlinu

RxJava rješava kompleksnost međudretvene komunikacije i pruža deklarativan, slijedan oblik asinkronosti koji je lakši za čitanje i pisanje, ali nosi strmu krivulju učenja, potpuno novu paradigmu programiranja, kompleksnost i zadržava povratne pozive. S druge strane, korutine u Kotlinu omogućavaju pisanje asinkronog koda u potpunosti bez povratnih poziva.

2.7.1 Izravni stil i stil prosljeđivanja kontinuacija

Podrutina (engl. *subroutine*) je nezavisni dio programa kojeg je moguće koristiti u drugim programima [41, str.1]. Implementirane su u obliku funkcija, metoda, operacija, potprograma i sl., odnosno nezavisnih strukturnih cjelina programskog koda. Uobičajeni način pozivanja podrutina kojeg se uči kad se tek počne programirati je slijedni u kojem se glavna rutina (engl. *main routine*) izvršava dok ne dođe do točke u kojoj se poziva podrutina. Tada se izvršava podrutina i kad završi, njezin se rezultat vraća glavnoj rutini koja nastavlja dalje. Takav se stil programiranja zove izravni stil (engl. *direct style*, DS). On je jednostavan za čitanje jer redoslijed linija

koda odražava način na koji se program izvršava. Nedovoljan je za slučajeve u kojima su neke podrutine blokirajuće jer glavna rutina ostaje blokirana sve do kraja podrutine. Rješenje ovom problemu je asinkronost gdje se pojavljuje problem međudretvene komunikacije. Mehanizam komunikacije koji se koristio u poglavljima 2.5 i 2.6 je prosljeđivanje podrutine koju dretva poziva kako bi dostavila rezultate pozivajućem opsegu. Ovakva se podrutina zove povratni poziv, a stil programiranja koji ju koristi stil prosljeđivanja kontinuacija (engl. *continuation-passing style*, CPS) [42, str.19]. Kontinuacija je dio koda u kojem se rezultat obrađuje, odnosno tijelo povratnog poziva. Iako CPS rješava problem međudretvene komunikacije, uvodi svoje probleme i smanjuje čitkost koda. U CPS se više ne može, kao u DS, slijedno pratiti kod i zaključiti kako se program izvršava. Može ih se slijedno pratiti jedino ako se ulančavaju, ali to vodi problemu pakla povratnih poziva. RxJava izravnava ovu hijerarhiju povratnih poziva, ali svejedno ne pruža tako jednostavan kod kao DS. S druge strane, korutine i njihova implementacija u Kotlinu rješavaju oba problema jer omogućavaju pisanje asinkronog koda jednostavnim, izravnim stilom.

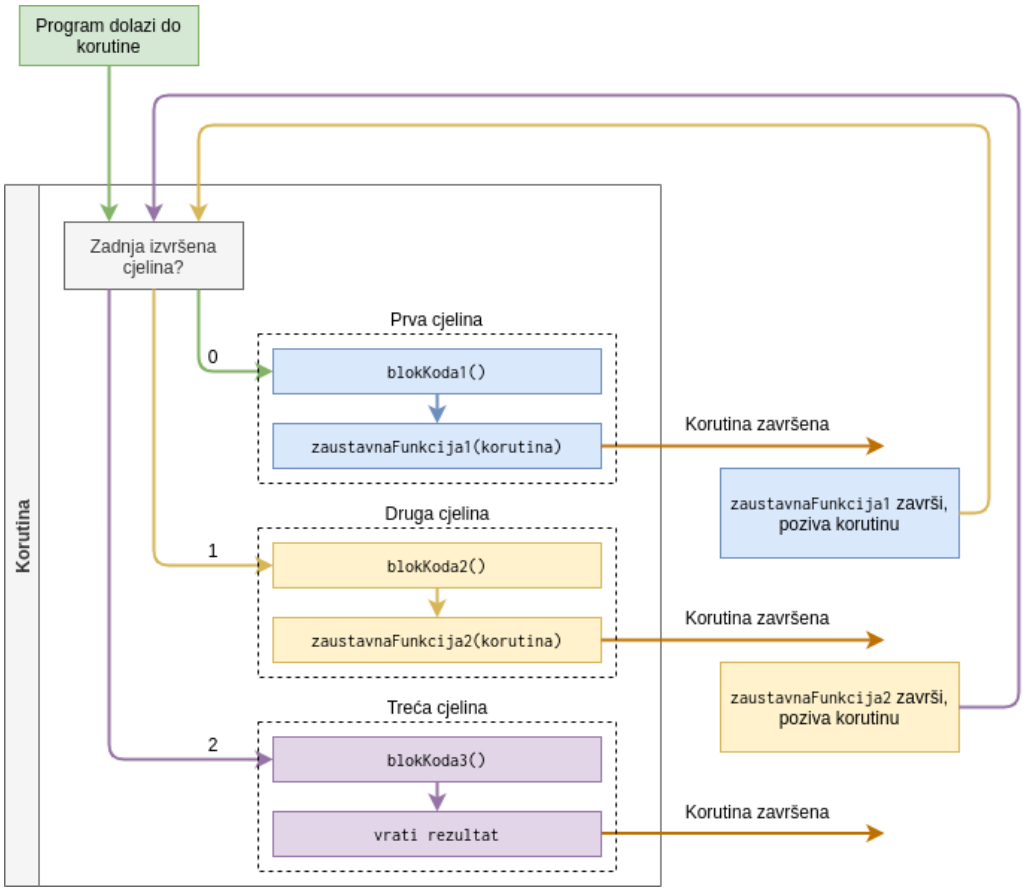
2.7.2 Korutina u računarstvu

Kad glavna rutina tijekom izvršavanja naiđe na poziv podrutine, staje dok podrutina ne završi, nakon čega nastavlja odakle je stala. Ovo se ponašanje postiže stogom poziva (engl. *call-stack*) na kojemu se nalaze sve rutine koje još nisu završile. Rutine se stavljaju (engl. *push*) na stog u trenutku poziva, a uklanjaju (engl. *pop*) sa stoga kad završe. Nakon uklanjanja, sljedeća rutina na stogu nastavlja odakle je stala [43, str.76]. Pri tome podrutine kreću s izvršavanjem od početka, obavljaju svoj posao i završe te naknadnim pozivima opet kreću od početka. S druge strane, korutine su rutine koje počinju od mjesta na kojem su prošli put završile [44, str.194]. One pamte gdje su prošli put stale i sljedećim pozivom se izvršavaju odatle, a ne od samog početka.

Korutina počinje s izvršavanjem kad ju se pozove iz neke druge rutine, kao i obična podrutina. Razlika je u tome što se sastoji od slijednih cjelina od kojih se sljedeća izvršava svakim uzastopnim pozivom korutine. Granice između ovih cjelina zovu se zaustavne točke (engl. *suspension points*) na kojima se nalaze pozivi posebnih podrutina zvane zaustavne funkcije (engl. *suspending functions*). Pozivom zaustavne funkcije, korutina završava (uklanja se sa stoga poziva) i tada se za nju kaže da je zaustavljena (engl. *suspended*). Nakon što zaustavna funkcija završi, opet poziva korutinu nakon čega se ona izvršava od sljedeće cjeline i proces se ponavlja do zadnje cjeline. Korutine se u programskim jezicima mogu implementirati na različite načine, međutim bitno je štedljivo korištenje resursa za što su se efikasnim pokazali konačni automati (engl. *state machine*) [45]. Dijagram 2.8 prikazuje korutinu u takvom obliku, a slijed izvršavanja je sljedeći:

1. Program svojim izvršavanjem dolazi do mjesta na kojem se poziva korutina. Ona se pokreće i stavlja na stog poziva.

2. Korutina provjerava je li prethodno izvršena neka cjelina. Kako je ovo je prvi put da je



Sl. 2.8: *Dijagram korutine u obliku konačnog automata, nadahnut sličnim dijagramima iz [46]*

pozvana, nijedna se cjelina prethodno nije izvršila, pa kreće od prve.

3. Prva cjelina izvršava običan kod koji je predstavljen funkcijom `blokKoda1`. Zamisliti primitivne operacije, pozive običnih podrutina, grananja, petlje itd.

4. Nakon toga, korutina dolazi do prve zaustavne točke. U tom trenutku zabilježava do koje je cjeline došla kako bi znala odakle nastaviti sljedeći put. Zamisliti da postavlja internu varijablu `lastSection` na `1`.

5. Poziva zaustavnu funkciju i prosljeđuje joj instancu sebe. Ono što zaustavnu funkciju razlikuje od obične je što, osim svojih parametara, prima objekt koji joj omogućava ponovno pozivanje korutine s njezinim očuvanim stanjem. Zbog jednostavnosti, ovdje se zaustavnoj funkciji prosljeđuje instanca korutine.

6. Korutina završava i uklanja se sa stoga poziva.

7. Kad zaustavna funkcija završi, poziva predanu korutinu i predaje joj svoj rezultat (ako ga ima). Na ovaj se način vrijednosti mogu prenositi iz cjeline u cjelinu. Ovaj je korak sličan pozivanju **povratnog poziva** nakon završetka pozadinske dretve.

8. Korutina provjerava je li prethodno izvršena neka cjelina. Jer je `lastSection` varijabla sada postavljena na `1`, korutina kreće s izvršavanjem od druge cjeline.

9. Postupak se ponavlja od koraka 3 do 8 sve dok cjeline završavaju zaustavnim funkcijama.

10. Nakon kraja zadnje cjeline, korutina vraća svoj izračunati rezultat i završava u potpunosti jer više nema zaustavnih funkcija koje ju mogu aktivirati. Sljedećim pozivom ona kreće ispočetka.

Kad bi se korutine ručno implementirale u obliku konačnih automata, suština koda se ne bi značajno razlikovala od običnih povratnih poziva, osim znatno povećane kompleksnosti i smanjene čitkosti. Ključ je u tome što programski jezici mogu transformirati kod pisan izravnim stilom u konačni automat tijekom prevođenja (engl. *compile*), što se zove CPS transformacija (engl. *CPS transformation*) [45]. Jedan takav programski jezik je Kotlin, koji osim toga omogućava i asinkrono izvršavanje zaustavnih funkcija.

2.7.3 Pregled terminologije

Kotlin ima korutine od verzije 1.3 [47] koje, nakon prevođenja, imaju oblik konačnih automata. Prije detaljnijeg pregleda unutarnjih mehanizama i načina rada korutina u Kotlinu, dobro je poznavati pojmove koji se pri tom koriste i kakav im je međusobni odnos. Popis pojmova s objašnjenjima dan je ovdje [45]:

- Korutina – instanca zaustavljivog izvršavanja. Konceptualno je slična dretvi po tome što prima blok koda kojeg izvršava i može biti stvorena i pokrenuta, ali se razlikuje u tome što nije vezana za jednu dretvu, već može biti zaustavljena na jednoj i nastavljena na nekoj drugoj. Osim toga, završetkom može vratiti rezultat.

- Zaustavna funkcija – funkcija označena ključnom riječju `suspend`. Može zaustaviti izvršavanje koda bez blokiranja dretve u kojoj je pozvana. Ne može biti pozvana iz običnog koda, već samo iz drugih zaustavnih funkcija i lambda izraza.

- Zaustavna lambda (engl. *suspending lambda*) – blok koda koji mora biti pokrenut u korutini. Označen je ključnom riječju `suspend` što ga razlikuje od običnog lambda izraza i slično njemu, on je anonimna zaustavna funkcija.

- Korutinski graditelj (engl. *coroutine builder*) – funkcija koja prima zaustavnu lambdu, stvara korutinu i, po želji, vraća njezin rezultat.

- Zaustavna točka – trenutak izvršavanja korutine u kojem ona može biti zaustavljena. Sintaktički, ovo je točka u kojoj se poziva zaustavna funkcija.

- Kontinuacija (engl. *continuation*) – stanje zaustavljene korutine na zaustavnoj točki. Konceptualno predstavlja ostatak njezinog izvršavanja nakon zaustavne točke. Korutina koja je stvorena, ali ne i pokrenuta, predstavljena je inicijalnom kontinuacijom koja predstavlja njezino cijelo izvršavanje. Sadrži povratni poziv čijim pozivom korutina nastavlja s ostatkom izvršavanja. Kad se korutina nastavi na ovaj način, kaže se da je aktivirana (engl. *resumed*).

- Korutinski kontekst (engl. *coroutine context*) – skup objekata pridruženih korutini. Ovaj skup može sadržavati objekte koji su odgovorni za dretvenost, zabilježbu (engl. *logging*), sigurnost, ime korutine itd. Drugim riječima, to je skup objekata koji obilježava korutinu i određuje njezin način rada.

- Korutinski opseg (engl. *coroutine scope*) – omotač korutinskog konteksta koji služi kao temelj za definiciju korutinskih graditelja u obliku funkcija proširenja. Omogućava pokretanje korutina s istim korutinskim kontekstom.

2.7.4 Jednostavna korutina

Nakon usvajanja osnovnih pojmova i načina na koje korutine funkcioniraju, jednostavno ih je pisati, čitati i pokretati. Primjer 2.32 prikazuje jednostavnu korutinu koja postavlja tekst na sučelje, čeka pet sekundi i opet postavlja tekst na sučelje. Glavna privlačnost korutina je što zaustavna funkcija `delay` ne blokira pozivajuću dretvu, iako konceptualno predstavlja vremensku odgodu nastavka koda u bloku. Umjesto toga, ona zaustavlja pokrenutu korutinu i aktivira ju za pet sekundi. Ova korutina je pokrenuta u opsegu `lifecycleScope` koji pokreće svoje korutine na glavnoj dretvi i definiran je u biblioteci proširenja životnog ciklusa [48]. Na njemu se poziva korutinski graditelj `launch` koji gradi i pokreće korutinu definiranu proslijeđenom zaustavnom lambdom.

Primjer koda 2.32: *Jednostavna kortuina u aktivnosti*

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    lifecycleScope.launch {
        binding.tvResult.text = "Text before delay"
        delay(5000)
        binding.tvResult.text = "Text after 5 seconds"
    }
}
```

Jer je korutina pokrenuta u opsegu koji je vezan za glavnu dretvu, normalan kod postavljanja teksta na sučelje također se izvršava na glavnoj dretvi. Međutim, zaustavne funkcije ne moraju pratiti ovo pravilo. One jamče samo da neće blokirati pozivajuću dretvu, a mehanizam kojim to rade se razlikuje od funkcije do funkcije. Primjerice, `delay` u pozadini koristi običan poziv `Handler#postDelayed` kako bi aktivirao korutinu za pet sekundi (pogledati implementaciju `delay` funkcije i [49, lin.135]). Neke druge zaustavne funkcije mogu koristiti skup dretvi koje svojim završetkom aktiviraju korutinu (pogledati [50, lin.94]). Dakle, korutine samo pametno apstrahiraju asinkronu komunikaciju povratnim pozivima. Komponenta koja omogućava asinkronost u korutinama je dispečer (engl. *dispatcher*) koji je dio konteksta korutine.

2.7.5 Korutinski kontekst

Korutinski kontekst je skup objekata koji definiraju svojstva korutine pomoću kojih korutinski okvir (engl. *coroutine framework*) upravlja njezinim izvršavanjem [45]. Konkretni konteksti se implementiraju nasljeđivanjem sučelja `CoroutineContext` koje sadrži mapu elemenata `Element` koji predstavljaju navedene objekte. Novoj se korutini dodjeljuje kontekst prosljeđivanjem

njegove instance korutinskom graditelju (npr. `launch`). Ključni elementi konteksta su [43, str.104]:

- `CoroutineExceptionHandler` – klasa koja rukuje neuhvaćenim iznimkama tijekom izvršavanja korutine. Nalik je `onError` povratnom pozivu RxJave. Kad se njegova metoda `handleException` pozove, više se nije moguće oporaviti od nastale pogreške jer je korutina završila. Koristi se za rukovanje popratnim efektima nakon iznimke, npr. zabilježba nastale pogreške, prikaz poruke ili ponovno pokretanje korutine [51].

- `Job` – dio posla s mogućnošću otkazivanja čiji životni ciklus završava kad završi posao kojeg predstavlja. Ako posao sadrži druge poslove (djecu), oni se otkazuju njegovim otkazivanjem. Graditelj `launch` vraća instancu posla koji predstavlja posao korutine u izvršavanju. Posao je predstavljen sučeljem `Job` koji omogućava čekanje na završetak korutine metodom `join`, otkazivanje metodom `cancel` i pristup informacijama o životnom ciklusu varijablama `isActive`, `isCancelled` i `isCompleted`. Osim toga, omogućava pristup svojoj djeci na kojima se mogu raditi iste ove operacije. Kad se korutinskom kontekstu dodijeli posao, sve korutine pokrenute s njim su djeca tog posla [52]. Tim se mehanizmom može upravljati njihovim izvršavanjem s jednog mjesta. Primjerice, za otkazivanje svih pokrenutih korutina konteksta dovoljno je pozvati `cancel` na njegovom poslu.

- `ContinuationInterceptor` – sučelje koje definira mehanizam presretanja kontinuacije. Sastoji se od metode `interceptContinuation` koja prima i vraća kontinuaciju, a interno se implementira kao omotač oko ulazne kontinuacije koji joj mijenja ponašanje. Osim običnog omotavanja kontinuacija, ovaj mehanizam omogućava izvršavanje cjelina korutine na dretvi koja je određena presretačem u njezinom kontekstu. Za ovo su ponašanje odgovorni dispečeri (engl. *dispatcher*) o kojima će biti riječi u potpoglavlju 2.7.6.

Elementi konteksta se mogu jednostavno kombinirati operatorom `+`. Tako se deklarativno mogu prikazati svojstva korutine koju se pokreće. Kontekst može biti implicitan ako je definiran u korutinskom opsegu čime sve njegove korutine dijele taj kontekst. Ovo se ponašanje može nadjačati eksplicitnom definicijom konteksta u graditelju korutine. Primjer 2.33 prikazuje stvaranje konteksta kompozicijom elemenata i njegovo prosljeđivanje korutinskom graditelju, čime se korutina izvršava s njegovim svojstvima. Korutinskom se kontekstu može pristupiti varijablom `coroutineContext`, a njegovim se elementima može pristupiti ključem koji odgovara sučelju elementa jer se elementi čuvaju u mapi. Tako se ovdje pristupa imenu i presretaču i može se vidjeti da odgovaraju postavljenim elementima. Podizanjem iznimke, ona stiže u postavljenog rukovatelja iznimkama (engl. *exception handler*) koji ju samo ispisuje.

Primjer koda 2.33: *Postavljanje konteksta korutini*

```
1 val exceptionHandler = CoroutineExceptionHandler { coroutineContext, throwable ->
2     Log.e(TAG, "An unhandled error occurred: ${throwable.message}", )
3 }
4 val job = Job()
5 val continuationInterceptor = Dispatchers.Default
6 val customContext = exceptionHandler + job + continuationInterceptor + CoroutineName("
      ↪ Custom context")
```

```
lifecycleScope.launch(customContext) {
    Log.d(TAG, "Context name: ${coroutineContext[CoroutineName]?.name}")
    Log.d(TAG, "Context continuation interceptor: ${coroutineContext[
        ↪ ContinuationInterceptor]}")
    throw RuntimeException("Whoops!")
}

// Output:
// Context name: Custom context
// Context continuation interceptor: Dispatchers.Default
// An unhandled error occurred: Whoops!
```

Za preinaku elemenata trenutnog konteksta koristi se zaustavna funkcija `withContext` koja izvršava predani joj kod s kontekstom koji je rezultat zbrajanja vanjskog i predanog [53]. Tako se može nadjačati bilo koji element trenutnog konteksta, ali uglavnom se koristi za nadjačavanje dispečera na kojem se predani kod izvršava. Nakon završetka ove funkcije, korutinski kontekst se vraća na prijašnje stanje. Primjer 2.34 prikazuje kako se `withContext` može koristiti za izmjenu elemenata korutinskog konteksta. Bitno je primijetiti da se, osim imena, presretač promijenio iz `Dispatchers#Default` u `Dispatchers#Main`. Ova funkcija omogućava proizvoljno prebacivanje presretača unutar korutine, što može značiti i proizvoljno prebacivanje dretvi na kojima se korutina izvršava, što je jedna od glavnih privlačnosti korutina.

Primjer koda 2.34: *Primjer 2.33 s preinakom konteksta*

```
...
lifecycleScope.launch(customContext) {
    Log.d(TAG, "Context name: ${coroutineContext[CoroutineName]?.name}")
    Log.d(TAG, "Context continuation interceptor: ${coroutineContext[
        ↪ ContinuationInterceptor]}")
    withContext(Dispatchers.Main + CoroutineName("Modified custom context")) {
        Log.d(TAG, "Context name: ${coroutineContext[CoroutineName]?.name}")
        Log.d(TAG, "Context continuation interceptor: ${coroutineContext[
            ↪ ContinuationInterceptor]}")
    }
    throw RuntimeException("Whoops!")
}

// Output:
// Context name: Custom context
// Context continuation interceptor: Dispatchers.Default
// Context name: Modified custom context
// Context continuation interceptor: Dispatchers.Main
// An unhandled error occurred: Whoops!
```

2.7.6 Dispečeri i upravljanje asinkronošću

Korutinski dispečeri određuju dretvu ili skup dretvi na kojoj se korutina izvršava [43, str.114]. Predstavljeni su klasom `CoroutineDispatcher` koja nasljeđuje `ContinuationInterceptor` sučelje. Nadjačava njegovu `interceptContinuation` metodu gdje kontinuaciju omotava u `DispatchedContinuation` koja poziv svoje `invokeSuspend` metode prosljeđuje konkretnom dispečeru. Time se ponovna aktivacija korutine izvršava na dretvi koja je njime definirana. Gotovi se dispečeri nalaze u klasi `Dispatchers` [54]:

1. `Main` – dispečer koji ponovno aktivira korutinu na glavnoj dretvi odgovornoj za rad s elementima sučelja. Na Androidu, ovo je glavna dretva, a ovaj se dispečer koristi svaki put kad je potrebno iz korutine pristupiti elementima sučelja.

2. `Default` – zadani dispečer kojeg koriste svi korutinski graditelji ako u pozivu ili kontekstu nije postavljeno drugačije. Na Java virtualnom stroju (engl. *Java virtual machine*, JVM) ovaj dispečer koristi dijeljeni skup dretvi veličine 2 ili broja dostupnih procesorskih jezgri, što god je veće. Pogodan je za obradbeno zahtjevne operacije.

3. `IO` – dispečer predviđen za obradu blokirajućih poslova. Koristi dijeljeni skup dretvi veličine 64 ili broja dostupnih procesorskih jezgri, što god je veće.

4. `Unconfined` – dispečer koji nema unutarnji mehanizam dretvenosti. On koristi dretvu na koju je koristila prošla zaustavna funkcija.

Kako se istodobnošću u RxJavi u potpunosti upravlja raspoređivačima, tako se u korutinama upravlja dispečerima. S tog su gledišta dispečeri slični raspoređivačima iz RxJave; koriste se kad je potrebno neki posao unutar korutine prebaciti s jedne na drugu dretvu. Za razliku od RxJave, ovdje se ne mora brinuti o redoslijedu promjene dispečera niti o popratnim efektima na ostatak korutine jer se dispečer odnosi samo na blok koda na kojem je primijenjen. Primjer 2.35 prikazuje kako se na Androidu mogu napraviti dvije međuzavisne blokirajuće IO operacije čiji se rezultati prikazuju na sučelju. Jer je `lifecycleScope`u zadano postavljen `Main` dispečer, nije se potrebno prebacivati na njega kad se želi pristupiti elementima sučelja. Završetkom `withContext` bloka, kontekst se vraća u prijašnje stanje, a time i dispečer. Dubina ugniježđenosti bi ostala ista da je u primjeru bilo još desetke takvih međuzavisnih operacija, dok bi se s običnim povratnim pozivima oni nizali sve dublje.

Primjer koda 2.35: *Promjena dispečera na kojem se korutina izvršava*

```
lifecycleScope.launch {
    val userId = 10
    val userName = withContext(Dispatchers.IO) {
        getUserNameBlocking(id = userId)
    }
    binding.tvResult.text = "User id: $userId, user name: $userName"
    val usersWithName = withContext(Dispatchers.IO) {
        getUsersWithNameBlocking(name = userName)
    }
    binding.tvResult.text = "All user ids with name $userName: ${usersWithName.
        ↪ joinToString()}"
}
```

Kako bi se od navedenih blokirajućih poziva napravile funkcije koje se mogu pozvati iz korutine bez brige o dispečeru na kojem se trebaju izvršavati, potrebno ih je izdvojiti u posebne zaustavne funkcije i postaviti im dispečera tamo. Ovdje se to može napraviti jednostavnim kopiranjem `withContext` blokova u zasebne zaustavne funkcije, što prikazuje primjer 2.36. Kod u korutini sada izgleda kao običan blokirajući kod, ali ne blokira jer su zaustavne funkcije implementirane tako da prebace svoje blokirajuće poslove na `IO` dispečer.

Primjer koda 2.36: *Izdvajanje blokirajućih poziva u zaustavne funkcije*

```
lifecycleScope.launch {
    val userId = 10
    val userName = getUserName(id = userId)
    binding.tvResult.text = "User id: $userId, user name: $userName"
    val usersWithName = getUsersWithName(name = userName)
    binding.tvResult.text = "All user ids with name $userName: ${usersWithName.
        ↪ joinToString()}"
```

```
}
...
private suspend fun getUserName(id: Int): String {
    return withContext(Dispatchers.IO) {
        getUserNameBlocking(id = id)
    }
}

private suspend fun getUsersWithName(name: String): List<Int> {
    return withContext(Dispatchers.IO) {
        getUsersWithNameBlocking(name = name)
    }
}
```

2.7.7 Istodobne korutine

Kombinacijom `withContext` poziva može se lako prebacivati između dretvi na kojima se izvršava slijedni kod korutine. Ovo je pogodno za slučajeve u kojima postoji više međuzavisnih operacija jer ih se tada može predstaviti slijedno, ali nepogodno za one u kojima postoji više nezavisnih operacija za koje bi efikasnije bilo istodobno izvršavanje. U korutinama se ovaj obrazac ostvaruje `async` korutinskim graditeljem i funkcijom `await`. Njihov način rada dijeli dosta s `Future` objektima koji nastaju dodjeljivanjem poslova `Executoru`, kao što je objašnjeno u poglavlju 2.5. Tamo se `Future` koristio samo za otkazivanje poslova, ali on omogućava i sinkronizaciju istodobnih poslova pozivom `get` metode koja blokira pozivajuću dretvu dok njegov posao ne završi. Tako se `Executoru` može dodijeliti nekoliko poslova, sačuvati reference na rezultirajuće `Future` objekte i pozvati `get` metodu na njima u trenutku kad su rezultati potrebni. Jer su se oni do tog trenutka istodobno izvršavali na pozadinskim dretvama, imali su vremena napraviti dio posla ili čak završiti u vremenu dok nisu bili potrebni, što je skratilo vrijeme koje je pozivajuća dretva trebala blokirati. Prednost ovog mehanizma je što nije potrebno koristiti povratni poziv kako bi se dobio rezultat asinkrone operacije; metoda `get` izravno vraća njezin rezultat kao bilo koja metoda. Nažalost, nepogodan je za korištenje na Androidu jer ipak blokira dretvu dok rezultat nije dostupan. Kako je uvijek potrebno izbjeći blokiranje glavne dretve, opet bi se morali uvoditi mehanizmi međudretvene komunikacije, a time i povratni pozivi.

Korutine su do sad bile pokretane korutinskim graditeljem `launch`. On vraća instancu sučelja `Job` koji predstavlja korutinu u izvršavanju, a svrha mu je omogućiti otkazivanje pokrenute korutine i njezine djece. Osim njega, postoji i `async` koji vraća instancu sučelja `Deferred`. Na površini, jedina razlika između ova dva graditelja je što jedan vraća `Job`, a drugi `Deferred`. Naime, `Deferred` nasljeđuje `Job` i proširuje ga s mogućnošću čekanja na rezultat pokrenute korutine [55]. Čekanje se ostvaruje njegovom zaustavnom funkcijom `await`, koja je analogna metodi `Future#get`, ali se razlikuje u tome što ne blokira pozivajuću dretvu, nego zaustavlja korutinu. Kao što `Job` ima metodu `join` koja zaustavlja korutinu dok posao ne završi, `Deferred` ima metodu `await` koja zaustavlja korutinu dok posao ne završi i vraća njegov rezultat. Primjer 2.37 prikazuje kako se ovaj mehanizam može koristiti za istodobno izvršavanje korutina. Njegovim se pokretanjem tekst na sučelju prikazuje nakon što najduža od dvije zaustavne funkcije u `async` blokovima završi.

Primjer koda 2.37: *Istodobno izvršavanje korutina pomoću async graditelja*

```
val userId = 10
val userNameDeferred = lifecycleScope.async { getUserName(id = userId) }
val userAddressDeferred = lifecycleScope.async { getUserAddress(id = userId) }
lifecycleScope.launch {
    val userName = userNameDeferred.await()
    val userAddress = userAddressDeferred.await()
    binding.tvResult.text = "User name: $userName, user address: $userAddress"
}
...
private suspend fun getUserName(id: Int): String {
    return withContext(Dispatchers.IO) {
        getUserNameBlocking(id = id)
    }
}

private suspend fun getUserAddress(id: Int): String {
    return withContext(Dispatchers.IO) {
        getUserAddressBlocking(id = id)
    }
}
```

Istodobno izvršavanje korutina ne mora se implementirati samo pomoću `async` graditelja. Na isti se način mogao koristiti `launch` graditelj ako nije potrebno obraditi rezultat korutine, nego samo čekati njezin završetak. Jedina promjena koja bi tada bila potrebna je poziv metode `join` umjesto `await`.

2.7.8 Otkazivanje i strukturirana istodobnost

Kao što je pokriveno u poglavljima 2.5 i 2.6, asinkrone je poslove potrebno otkazati ako njihovi rezultati više nisu potrebni. Korutine se otkazuju pomoću sučelja `Job` i njegove metode `cancel` koja propagira događaj otkazivanja svojoj djeci. Međutim, često se korutine ne otkazuju tako granularno, nego na višoj razini. Od najviše prema najnižoj to su:

1. Otkazivanje korutinskog opsega. Radi se pozivom metode `cancel` koja delegira otkazivanje unutarnjem korutinskom kontekstu.

2. Otkazivanje korutinskog konteksta. Također se radi pozivanjem metode `cancel` koja delegira otkazivanje svojem `Job` elementu. Ako on ne postoji, podiže iznimku.

3. Otkazivanje konkretne instance sučelja `Job`, čime se otkazuju i sva njegova djeca. Ovakva ovisnost posla o njegovom roditelju zove se strukturirana istodobnost [43, str.97] i predstavlja središnji mehanizam korutina koji s jednog mjesta omogućava otkazivanje čitave hijerarhije poslova.

Kao kod ostalih mehanizama asinkronosti predstavljenih u ovom radu, otkazivanje poslova je kooperativno. Otkazivanjem posla, njegova zastavica `isActive` se postavlja na `false` što omogućava korutinama izvanredan prekid svog posla. Analogna je zastavicama `isInterrupted`, `isCancelled` ili `isDisposed`. Njoj se u zaustavnoj funkciji može pristupiti referencom na opseg ili kontekst. Može se pristupiti i izravno sintaksom `coroutineContext[Job]`, međutim bolje je pomoću prve dvije reference jer imaju sigurnosne mehanizme u slučaju da kontekst nema posao. Primjer 2.38 prikazuje jednostavan slučaj otkazivanja zaustavne funkcije na klik gumba. Nekoliko napomena u vezi ovog primjera:

- Da se umjesto `cancellableWorkJob` otkazao `lifecycleScope`, otkazale bi se sve korutine pokrenute u tom opsegu. Čuvanjem instance posla dobiva se granulacija upravljanja otkazivanjem koja nije uvijek potrebna, pogotovo na Androidu kad će u većini slučajeva biti potrebno samo otkazati cijeli `lifecycleScope` ili `viewModelScope`.
- Zastavica `isActive` mogla se provjeriti i izvan `withContext` bloka preko konteksta zaustavne funkcije kodom `coroutineContext.isActive`.
- Mogla se provjeriti i u zaustavnoj lambdi graditelja na isti način. Općenito, ona je dostupna u svakoj zaustavnoj funkciji ili lambda izrazu i najbolja je praksa provjeravati ju često.
- Otkazivanje instance sučelja `Deferred` radi se na isti način.

Primjer koda 2.38: *Jednostavno kooperativno otkazivanje zaustavne funkcije*

```
private var cancellableWorkJob: Job? = null

override fun onCreate(savedInstanceState: Bundle?) {
    ...
    binding.btnStop.setOnClickListener {
        cancellableWorkJob?.cancel()
    }
    cancellableWorkJob = lifecycleScope.launch {
        doCancellableWork()
    }
}

private suspend fun doCancellableWork() {
    withContext(Dispatchers.Default) {
        while (isActive) { // or coroutineContext.isActive
            Log.d(TAG, "doCancellableWork: working...")
            delay(1000)
        }
    }
}
```

2.7.9 Česti problemi i anti-obrasci

Korutine pružaju mogućnost pisanja asinkronog koda izravnim stilom koji izgleda kao običan blokirajući kod. Kako bi se koristile pravilno i bez neželjenih popratnih efekata, potrebno je poznavati neke od njihovih unutarnjih mehanizama i načela rada i izbjegavati česte uzroke problema i anti-obrasce. Neki od takvih problema i anti-obrazaca dani su u ovom potpoglavlju.

Ignoriranje životnog ciklusa roditeljske komponente Korutine ne uklanjaju potrebu za otkazivanjem pozadinskih poslova kad njihovi rezultati više nisu potrebni. Završetkom Android komponentne, najčešće je potrebno završiti i pokrenute korutine. Zato je dobra praksa stvoriti korutinski opseg u dotičnoj komponenti (ViewModel, aktivnost, fragment itd.), pokretati korutine u njemu i otkazati ga završetkom životnog ciklusa komponente. Primjer 2.39 prikazuje kako se ovo može napraviti u ViewModelu, ali isti se pristup može primijeniti u aktivnostima ili fragmentima, samo se otkazivanje opsega treba pomaknuti u `onDestroy`.

Primjer koda 2.39: *Stvaranje i otkazivanje korutinskog opsega u ViewModelu*

```
class MainActivityViewModel : ViewModel() {
    private val scope = CoroutineScope(Job() + Dispatchers.Main)
    ...
    override fun onCleared() {
        super.onCleared()
        scope.cancel()
    }
}
```

Eksplicitno otkazivanje u prošlim primjerima nije bilo potrebno jer se koristio ugrađeni `lifecycleScope` koji se automatski otkazuje završetkom aktivnosti. Slično se ponaša i ugrađeni `viewModelScope` koji se otkazuje završetkom ViewModela. Osim toga, kao kod ostalih mehanizama asinkronosti, prebacivanjem korutina u ViewModel i pokretanjem u `viewModelScope`u one preživljavaju konfiguracijske promjene, što vodi boljem korisničkom iskustvu.

Kršenje strukturirane istodobnosti Ako je u korutini potrebno pokrenuti korutinu, važno je osigurati da se ona pokreće u istom opsegu kao roditeljska korutina [43, str.98]. Ovo osigurava točnu propagaciju otkazivanja svim zavisnim korutinama. U suprotnom bi se događaj otkazivanja primijenio samo na korutine u opsegu kojega se otkazalo, ali ne i unutarnje korutine, čime bi ostale nepotrebno aktivne. Primjer 2.40 prikazuje ovo ponašanje. Pokretanjem koda i otkazivanjem `lifecycleScope`a prestaje se ispisivati samo *"Outer coroutine: working..."*, dok se *"Inner coroutine: working..."* nastavlja ispisivati beskonačno. Tek se otkazivanjem `GlobalScope`a prestaje ispisivati tekst unutarnje korutine, odnosno dotična korutina završava. Ovo predstavlja i veći problem od samog ispisivanja jer kad bi se u korutini pristupalo elementu aktivnosti, njezina bi memorija iscurila uslijed konfiguracijskih promjena, kao i kod ostalih mehanizama asinkronosti.

Primjer koda 2.40: *Pogrešno pokretanje ugniježđene korutine*

```
lifecycleScope.launch {
    GlobalScope.launch {
        while(isActive) {
            delay(1000)
            Log.d(TAG, "Inner coroutine: working...")
        }
    }
    while(isActive) {
        delay(1000)
        Log.d(TAG, "Outer coroutine: working...")
    }
}
```

Ovi se problemi rješavaju pokretanjem korutine iz opsega roditeljske korutine. U ovom se primjeru to može napraviti uklanjanjem koda `GlobalScope.` nakon čega ostaje samo `launch` koji se veže uz vanjski opseg. To je moguće jer svaki korutinski graditelj nasljeđuje korutinski opseg, pa mu se unutar zaustavnog lambda izraza može pristupiti varijablom `this`. Ako je unutarnja korutina izdvojena u posebnu funkciju, treba joj proslijediti instancu korutinskog opsega u kojem će biti pokrenuta. Primjer 2.41 prikazuje ovaj slučaj. Naravno, isto se ovo može napraviti graditeljem `async`, samo se tada treba rukovati `Deferred` objektima.

Primjer koda 2.41: *Pravilno pokretanje ugniježđene korutine, nadahnuto kodom iz [43, str.97]*

```
lifecycleScope.launch {
    launchInnerCoroutine(parentScope = this)
    while(isActive) {
        delay(1000)
        Log.d(TAG, "Outer coroutine: working...")
    }
}
...
private fun launchInnerCoroutine(parentScope: CoroutineScope): Job {
    return parentScope.launch {
        while(isActive) {
            delay(1000)
            Log.d(TAG, "Inner coroutine: working...")
        }
    }
}
```

Strukturiranu istodobnost krši i ignoriranje zastavice `isActive`, jer se tada korutina ne može kooperativno završiti. Zato ju je potrebno provjeravati na ključnim mjestima i završiti posao što je brže moguće kako bi se oslobodili resursi i izbjeglo curenje memorije. Primjerice prije početka blokirajuće ili zahtjevne operacije, svakom iteracijom petlje, prije poziva zaustavne funkcije itd. Načelo je isto kao u poglavljima 2.5 i 2.6.

Zaustavna funkcija bez dretvene sigurnosti Konvencija pisanja zaustavnih funkcija je da one ne smiju blokirati pozivajuću dretvu [56]. Na Androidu se ovo preslikava u činjenicu da zaustavne funkcije ne smiju blokirati glavnu dretvu, što omogućava bezbrižno pozivanje zaustavnih funkcija iz opsega vezanog glavne dretve, kao u primjeru 2.36. Dakle, odgovornost asinkronog izvršavanja pada na zaustavnu funkciju, a ne na pozivatelja, što je slično obrascu iz poglavlja 2.5.6 gdje repozitorij apstrahira unutarnju asinkronost i izlaže povratne pozive.

Fiksiranje konteksta i opsega Konteksti i opsezi u kojima su se korutine pokretale su do sada bili fiksirani, što znači da ih je nemoguće izmijeniti izvana. Ovo predstavlja problem prilikom testiranja jer se u tom slučaju trebaju koristiti `TestCoroutineScope` i `TestCoroutineDispatcher`, o kojima će više riječi biti u poglavlju 2.8. Kako bi se osigurala testabilnost, potrebno je:

- U klasama koje definiraju zaustavne funkcije (npr. repozitorij) omogućiti ubrizgavanje konteksta koji se koriste za poslove. Primjerice, umjesto izravnog referiranja na `Dispatchers#IO`, koristiti kontekst iz konstruktora [43, str.236].
- U klasama koje pokreću korutine (npr. ViewModel) omogućiti ubrizgavanje korutinskog opsega. Primjerice, umjesto izravnog referiranja na `viewModelScope`, koristiti opseg iz konstruktora [57].

Ako se u nekoj klasi radi oboje, treba ubrizgati i opseg i kontekst. Primjer implementacije ovih promjena prikazan je u poglavlju 2.7.10.

Korištenje GlobalScopea Opsezi se trebaju definirati tako da su vezani za konkretni životni ciklus neke komponente [43, str.99]. `GlobalScope` je globalni korutinski opseg koji završava kad

i aplikacijski proces. Može mu se pristupiti iz bilo koje klase jer je implementiran kao globalno dostupan Singleton objekt. Prava motivacija za njegovo korištenje su operacije koje se trebaju izvršavati tijekom cijelog života aplikacije, ali ga početnici znaju zloupotrebljavati i koristiti za pokretanje svih korutina, čime se nepotrebno troše resursi, krši strukturirana istodobnost i otežava testiranje. Za takve je potrebe u Androidu preporučeno koristiti ručno definiran opseg čija se referenca nalazi u `Application` klasi [57] ili u okviru za ubrizgavanje ovisnosti.

2.7.10 Preporučeni obrazac korištenja

Napravljena je analiza unutarnjih mehanizama korutina, načina njihovog korištenja te čestih problema i anti-obrazaca. U ovom se potpoglavlju daje preporučeni obrazac korištenja prema najboljim praksama iz [57] i [43]. Za potrebe demonstracije, primjer iz 2.5.6 je preoblikovan tako da umjesto dretvi koristi korutine. Aktivnost je ostala ista jer su odgovornosti između aplikacijskih slojeva razdvojene, pa ona ne mora znati da se unutarnja implementacija ViewModela u potpunosti promijenila, niti reagirati na tu promjenu. Promjene u ViewModelu prikazane su primjerom 2.42. Korutinski opseg se ubrizgava u konstruktoru i otkazuje u `onCleared` metodi. `getNextResult` metoda sada koristi korutinu kako bi postigla svoj zadatak. Ova implementacija ima sljedeće prednosti:

- Omogućeno testiranje jer se može ubrizgati proizvoljan korutinski opseg.
- Otkazivanje korutina završetkom ViewModela.
- Algoritam obradbe rezultata asinkrone operacije napisan čitkim, izravnim stilom koji izgleda kao običan, blokirajući kod.
- Rukovanje iznimkama blokom `try-catch`.

Primjer koda 2.42: *ViewModel iz primjera 2.12 prilagođen na korutine*

```
class MainActivityViewModel(
    private val resultRepository: ResultRepository,
    coroutineScope: CoroutineScope? = null
) : ViewModel() {
    ...
    private val coroutineScope = coroutineScope ?: viewModelScope

    override fun onCleared() {
        super.onCleared()
        coroutineScope.cancel()
    }

    fun getNextResult() {
        _progressVisible.value = true
        coroutineScope.launch {
            val result = try {
                resultRepository.getNextResult()
            } catch (e: Throwable) {
                "Failed to get result."
            }
            _result.value = result
            _progressVisible.value = false
        }
    }
}
```

Novi repozitorij je prikazan primjerom 2.43. Sada je odgovoran samo za izlaganje dretveno sigurne zaustavne funkcije koju druge komponente mogu pozivati u svojim korutinama. Dretvenu sigurnost osigurava primjenom `io` konteksta iz `ContextProvider` pružatelja konteksta. Ova implementacija ima sljedeće prednosti:

- Omogućeno testiranje jer se može ubrizgati proizvoljan davatelj konteksta, a time i proizvoljni dispečeri. Aplikacijski davatelj konteksta prikazan je primjerom 2.44.
- Jednostavno upravljanje dretvom na kojoj se izvršava blokirajući kod.

Primjer koda 2.43: *Repozitorij iz primjera 2.13 prilagođen na korutine*

```
class DefaultResultRepository(
    private val resultApi: ResultApi,
    private val contextProvider: CoroutineContextProvider
) : ResultRepository {
    override suspend fun getNextResult(): String {
        return withContext(contextProvider.io) {
            resultApi.fetchResult()
        }
    }
}
```

Primjer koda 2.44: *Aplikacijski davatelj konteksta*

```
class AppCoroutineContextProvider : CoroutineContextProvider {
    override val ui: CoroutineContext get() = Dispatchers.Main
    override val io: CoroutineContext get() = Dispatchers.IO
    override val default: CoroutineContext get() = Dispatchers.Default
}
```

2.8. Testiranje korutina

Kako su korutine danas preferirani i službeno preporučeni način implementacije asinkronosti u Android aplikacijama, u ovom se potpoglavlju pokriva i njihovo testiranje. Testiranje programskog koda je ključan dio razvoja softvera jer omogućava strukturiran, ponovljiv i pregledan proces pronalaska grešaka koje treba ispraviti prije dostave krajnjim korisnicima. Kad su funkcionalnosti aplikacije pokrivene testovima, oni služe i kao njezina dokumentacija te ulijevaju samopouzdanje programeru prilikom implementacije novih funkcionalnosti. Ako napravi promjenu koja utječe na stari kod, testovi će pasti, što pokazuje da je novi kod imao utjecaj na stari. Zbog ovih i ostalih prednosti, testiranje je ključan alat u izgradnji projekata svih veličina. Postoje tri općenita pristupa testiranju s obzirom na razinu uvida u izvorni kod. Testiranje crne kutije (engl. *black box*) je ono u kojemu ispitivač nema uvid u izvorni kod, a njemu suprotno je testiranje bijele kutije (engl. *white box*). Treći pristup je spoj ova dva i zove se testiranje sive kutije (engl. *gray box*) u kojem ispitivač ima uvid u neki izvorni kod, ali ne sav. U slučaju bijele kutije, testiraju se metode, klase, kohezija aplikacijskih slojeva, grananja programske logike i sl., što je moguće jedino kad je dostupan izvorni kod. Zbog toga su za pisanje ovih testova prikladni upravo autori aplikacije, jer imaju uvid u potpuni izvorni kod, njegov način rada i moguće slabe točke.

Testiranje bijele kutije dijeli se na tri razine s obzirom na opseg: opsežno (engl. *end-to-end*), integracijsko (engl. *integration*) i jedinično (engl. *unit*) [58, min.3]. Opsežno testiranje se bavi provjerom ponašanja ključnih korisničkih putanja kroz aplikaciju, npr. od ulaska u aplikaciju za online kupovinu do kupovine proizvoda. Takve se putanje sastoje od više komponenti, navigacije između njih, prosljeđivanja podataka itd. Na nižoj razini nalazi se integracijsko testiranje koje se bavi testiranjem suradnje između aplikacijskih komponenti, npr. suradnja između repozitorija i ViewModela. Na najnižoj razini nalazi se jedinično testiranje koje se bavi testiranjem najmanjih smislenih cjelina koda, npr. metode neke klase. Takav pristup omogućava detaljno testiranje ponašanja neke cjeline s obzirom na ulazne vrijednosti, iznimke i rubna stanja. Vodilja prilikom pisanja ovakvih testova je gotovo potpuna pokrivenost funkcionalnosti cjeline koju se testira i tada služi kao temelj ostalim razinama testiranja koje se bave širim opsegom i pretpostavljaju ispravnost temeljnih klasa. Ovo se potpoglavlje bavi jediničnim testiranjem klasa koje koriste korutine.

2.8.1 Jedinično testiranje u Androidu

Jedinični se testovi u Androidu pišu pomoću okvira JUnit4 [59][60]. To je jednostavan JVM testni okvir za pisanje ponovljivih testova. Testovi su predstavljeni metodama koje su označene anotacijom `@Test`. Njih okvir pokreće i, u ovisnosti o uspješnosti sastavnih testnih tvrdnji (engl. *assertions*), izvještava o uspješnosti testa. Jedan od načina strukturiranja jediničnih testova je postavi-izvrši-utvrdi (engl. *Arrange, Act, Assert*, AAA) koji se sastoji od tri faze:

1. Postavljanje inicijalnih uvjeta testa. Ovdje se osigurava da je stanje prije izvršavanja testa očekivano i da su sve varijable i objekti prikladno inicijalizirani.

2. Izvršavanje radnje koju se testira. Primjerice, ako se test odnosi na testiranje neke metode, poziva se ta metoda s prethodno pripremljenim parametrima i sprema se njezin rezultat.

3. Utvrđivanje ispravnosti rezultata. U ovoj se fazi utvrđuje slaže li se dobiveni rezultat s očekivanim. Ako se slaže, test prolazi, ako ne, test pada.

Primjer 2.45 prikazuje jednostavan test s ovakvom strukturom. Prvo se postavljaju varijable potrebne za izvršavanje metode koju se testira, ona se izvršava, nakon čega se utvrđuje slaže li se njezin rezultat s očekivanim. Testni okvir će u ovisnosti o rezultatu tvrdnje izvijestiti je li test uspješan ili ne.

Primjer koda 2.45: *Jednostavan AAA test*

```
class MultiplicationTest {
    @Test
    fun twoTimesTwo_equalsFour() {
        // Arrange
        val x = 2
        val y = 2
        val expectedResult = 4
        // Act
        val actualResult = x * y
        // Assert
        assertEquals(actualResult, expectedResult)
    }
}
```

Na ovaj se jednostavan način pišu lokalni jedinični testovi za čiste JVM klase bez ovisnosti o Android platformi. U MVVM aplikaciji, svi slojevi osim prikaznog su čiste JVM klase. Prikazni sloj je poseban i nije čista JVM klasa jer zahtjeva Android platformu kako bi radio. Za njegovo testiranje su potrebni instrumentirani testovi (engl. *instrumented tests*) u kojima je dostupna Android platforma, bilo to pokretanjem testova na stvarnom uređaju ili emulatoru, ili korištenjem okvira kao Robolectric [61] za simulaciju platforme. Pisanjem monolitnih aplikacija bez arhitekturnog obrasca, sav se kod nalazi u prikaznom sloju, čime i svi jedinični testovi moraju biti instrumentirani, što predstavlja problem jer su oni puno sporiji od lokalnih [62]. Jediničnih testova u većem projektu može biti nekoliko desetaka (ili stotina), a ako svi ovise o Android platformi, vrijeme izvršavanja testova će znatno porasti. Osim toga, kod monolitnih je aplikacija teško identificirati dovoljno malene i izolirane cjeline koje su kandidati za jedinično testiranje, jer se poslovna logika isprepliće s prikazom njezinih rezultata, što je još jedan od razloga zašto su arhitekturni obrasci tako važni. Omogućavaju razdvajanje poslovne logike od prikaza, čime se gotovo sva poslovna logika može testirati brzo i jednostavno u obliku čistih JVM klasa, bez ovisnosti o Android platformi. Pri tom se testira ispravnost podataka koji dospijevaju do prikaznog sloja (npr. vrijednosti iz `LiveData` objekata) s pretpostavkom da će ih ispravno prikazati.

2.8.2 Pokretanje korutina u testovima

Korutine su se u poglavlju 2.7 pokretale tako da ne blokiraju pozivajuću dretvu, čime ona nezavisno nastavlja sa svojim izvršavanjem. Ovakvo ponašanje je nepoželjno kod testiranja jer se utvrđivanje uspješnosti testa izvršava na pozivajućoj dretvi. Kad se obična korutina pokrene u testu, pozivajuća dretva nastavlja sa svojim izvršavanjem i odmah dolazi do utvrđivanja uspješnosti. Pokrenuta korutina je pri tom još uvijek aktivna i, ovisno o njezinoj implementaciji, može završiti prije ili poslije ovog dijela. Ako se slučajno dogodi da završi prije, test će proći, a ako ne, test će pasti. Za ovakav se test kaže da je neponovljiv (engl. *non-deterministic*, *flaky*) jer se svakim pokretanjem može dobiti drugačiji rezultat, što ga čini beskorisnim. Kako bi test bio ponovljiv, potrebno je osigurati završetak korutine prije utvrđivanja uspješnosti testa, što se postiže korutinskim graditeljima `runBlocking` i `runBlockingTest`. Graditelj `runBlocking` pokreće novu korutinu i blokira pozivajuću dretvu dok ne završi. Ako se u njemu pokrene još korutina, čekat će i njih, ali samo ako su pokrenute s njegovim opsegom. Graditelj `runBlockingTest` se ponaša isto, ali osim navedenog, preskače sve `delay`ove i sprječava curenje korutina. Općenito, testne metode trebaju koristiti `runBlockingTest` ako testiraju zaustavne funkcije, a lažni objekti koji simuliraju stvarno ponašanje nekog objekta (testni duplikati, engl. *test double*, *fake*) trebaju koristiti `runBlocking` za pokretanje svojih zaustavnih funkcija [63].

2.8.3 Testni korutinski opseg i dispečer

Osim pokretanja testova u blokirajućim korutinama, treba pripaziti i na činjenicu da Androidov dispečer glavne dretve nije dostupan u testnom okruženju. Pokretanje testa koji bilo gdje u sebi poziva `withContext(Dispatchers.Main)` podiže iznimku koja upozorava na nedostatak

`Looper`a glavne dretve. Ovaj se problem rješava pokretanjem korutina s testnim dispečerom koji je implementiran klasom `TestCoroutineDispatcher`. Prema zadanim postavkama, on trenutačno izvršava dostavljene mu poslove na pozivajućoj dretvi i pruža mogućnost zaustavljanja i pokretanja korutina. Pozivom metode `pauseDispatcher`, sljedeće pokretanje korutine će biti zaustavljeno dok se ne pozove metoda `resumeDispatcher`. Ovo će u testovima biti korisno za utvrđivanje vrijednosti prije i poslije izvršavanja korutina. Osim toga, metodom `cleanupTestCoroutines` omogućava podizanje iznimke ako nakon završetka testa postoje aktivne korutine [64]. Testnog dispečera koristi testni korutinski opseg `TestCoroutineScope`. Omata njegov API i omogućava stvaranje i pokretanje korutina funkcijom proširenja `runBlockingTest`, koja poziva statičku metodu `runBlockingTest` kojoj predaje svoj kontekst [65]. Za podizanje iznimke u slučaju iscurenih korutina, statički graditelj `runBlockingTest` poziva metodu `cleanupTestCoroutines` testnog opsega, koji poziva istoimenu metodu svog testnog dispečera, što znači da se pozivom bilo koje od ovih metoda na kraju poziva metoda unutarnjeg testnog dispečera. Prevencija curenja korutina je u testovima važna jer bi takve korutine mogle imati popratne efekte na ostale, nepovezane testove.

2.8.4 Testiranje zaustavnih funkcija

Klasa u MVVM arhitekturi koja izlaže svoje zaustavne funkcije višim slojevima je repozitorij. Primjer 2.46 prikazuje testnu klasu repozitorija iz poglavlja 2.7.10 koja se sastoji od dva jednostavna testa. Prvi (`getNextResult_returnsResult`) testira ponašanje repozitorija kad API sloj vrati vrijednost, a drugi (`getNextResult_throwsExceptionIfApiThrowsException`) njegovo ponašanje kad API podigne iznimku. Komentari primjera navedeni su u sljedećim točkama:

- Testni korutinski opseg se inicijalizira s testnim korutinskim dispečerom. Graditelj `runBlockingTest` ovog opsega se koristi za pokretanje svakog testa koji se sastoji od zaustavnih funkcija, jer se one moraju pokrenuti u blokirajućoj korutini.
- Apstrahiranje API-ja sučeljem omogućilo je stvaranje testnog duplikata API-ja koji, umjesto komunikacije s mrežom, vraća rezultat odmah. To je prihvatljivo jer ova klasa testira ponašanje repozitorija, a ne API-ja i jer bi stvarni API poziv rezultirao dužim vremenom izvršavanja testa. Općenito se preporučuje korištenje stvarnih objekata kad god je to moguće i prikladno [63], ali kod operacija koje produžuju vrijeme testiranja je bolje koristiti testni duplikat.
- Slično tome, apstrahiranje davatelja konteksta sučeljem omogućilo je stvaranje testnog duplikata koji za sve dispečere koristi testni dispečer. Ovime se, zajedno s prvom točkom i preporučenom implementacijom repozitorija, osigurava ponovljivost testova i strukturirana istodobnost.
- Metoda označena anotacijom `@Before` je odgovorna za inicijalizaciju početnog stanja svakog testa kako se isti kod ne bi morao pisati u zasebnim `@Test` metodama. U njoj se prije svakog testa ponovno inicijalizira repozitorij kojem se prosljeđuju testni duplikati

API-ja i davatelja konteksta. Osim toga, prije svakog testa postavi zastavicu testnom API-ju da ne podiže iznimke. Slično ponašanje nakon testova može se postići metodom označenom anotacijom `@After`.

- Test `getNextResult_returnsResult` jednostavno poziva metodu repozitorija i utvrđuje je li vratila očekivani rezultat koji je izložen kao statička varijabla u `FakeResultApi`.
- Test `getNextResult_throwsExceptionIfApiThrowsException` postavlja zastavicu testnog duplikata API-ja koja uvjetuje podizanje iznimke svakim pozivom njegovih metoda. Test utvrđuje podiže li tu iznimk i repozitorij.

Primjer koda 2.46: *Testna klasa repozitorija iz 2.7.10*

```
@ExperimentalCoroutinesApi
class DefaultResultRepositoryTest {
    private val testCoroutineDispatcher = TestCoroutineDispatcher()
    private val testCoroutineScope = TestCoroutineScope(testCoroutineDispatcher)
    private val testResultApi = FakeResultApi()
    private val testCoroutineContextProvider = FakeCoroutineContextProvider(
        ↪ testCoroutineDispatcher)
    // Class under test
    private lateinit var repository: DefaultResultRepository

    @Before
    fun setup() {
        repository = DefaultResultRepository(
            resultApi = testResultApi,
            contextProvider = testCoroutineContextProvider
        )
        testResultApi.shouldThrowError = false
    }

    @Test
    fun getNextResult_returnsResult() = testCoroutineScope.runBlockingTest {
        // Arrange
        val expectedResult = FakeResultApi.fakeResult
        // Act
        val actualResult = repository.getNextResult()
        // Assert
        assertEquals(expectedResult, actualResult)
    }

    @Test
    fun getNextResult_throwsExceptionIfApiThrowsException() = testCoroutineScope.
        ↪ runBlockingTest {
        // Arrange
        testResultApi.shouldThrowError = true
        // Act
        val threwException = try {
            repository.getNextResult()
            false
        } catch (e: Exception) {
            true
        }
        // Assert
        assertTrue(threwException)
    }
}
```

2.8.5 Testiranje koda koji pokreće korutine

Klase u MVVM arhitekturi koje stvaraju korutine su ViewModel i prikazni sloj. Primjer 2.47 prikazuje testnu klasu ViewModela iz poglavlja 2.7.10 koja se sastoji od tri testa. Prvi

(`getNextResult_showsResult`) se bavi ponašanjem ViewModela kad repozitorij vrati očekivani rezultat. Drugi (`getNextResult_showsAndHidesProgressBar`) se bavi ponašanjem trake napretka prilikom dohvaćanja rezultata. Treći (`getNextResult_showsError`) se bavi ponašanjem ViewModela kad repozitorij podigne iznimku. Komentari primjera navedeni su u sljedećim točkama:

- Kako bi se testirale komponente iz arhitekturnih komponenti Android Jetpack API-ja, potrebno je u svakoj testnoj klasi definirati `InstantTaskExecutorRule` koji osigurava da se, primjerice, postavljanje vrijednosti `LiveData` objektu izvršava sinkrono s pozivajućom dretvom, što osigurava ponovljivost testova [63].

- I ovdje se inicijaliziraju testni dispečer i opseg, ali se ne koriste isto kao u testu repozitorija. ViewModel ne izlaže zaustavne funkcije, nego obične metode koje pokreću svoje korutine, zbog čega se ViewModelu prosljeđuje testni opseg kojeg koristi za njihovo pokretanje. Ovime se, uz preporučenu implementaciju ViewModela, osigurava ponovljivost testova i strukturirana istodobnost. Kad bi ViewModel izlagao zaustavne funkcije, njihovi bi se testovi morali pokretali kao u poglavlju 2.8.4.

- Na testnom se dispečeru nakon svakog testa eksplicitno poziva `cleanupTestCoroutines` kako bi se osiguralo da nijedna korutina nije iscurila. Kod testiranja repozitorija ovo nije bilo potrebno jer `runBlockingTest` implicitno poziva tu metodu.

- Stanje sučelja se testira utvrđivanjem vrijednosti ViewModelovih `LiveData` objekata, a pretpostavlja se da će ih dotični prikazni sloj prikazati ispravno.

- Prvi test utvrđuje postavlja li se rezultat na stanje sučelja.

- U drugom se testu koristi zaustavljanje i pokretanje dispečera kako bi se utvrdilo stanje trake napretka. Zaustavljanjem dispečera, sljedeća korutina neće započeti izvršavanje dok se dispečer ne pokrene. Zato se može pozvati metoda `getNextResult` koja staje na pozivu korutine. U tom se trenutku utvrđuje stanje trake napretka i ponovno pokreće dispečer, čime se korutina izvršava i na svom kraju skriva traku napretka. Ovim se testom utvrđuje je li traka napretka skrivena prije, prikazana tijekom i skrivena nakon dohvaćanja rezultata.

- Treći je test sličan prvom, ali se u njemu postavlja zastavica repozitorija koja uvjetuje podizanje iznimke pozivom njegovih metoda. Ovim se testom utvrđuje prikazuje li se poruka greške na stanje sučelja.

Primjer koda 2.47: *Testna klasa ViewModela iz 2.7.10*

```
1 @ExperimentalCoroutinesApi
2 class MainActivityViewModelTest {
3     @get:Rule
4     val instantTaskExecutorRule = InstantTaskExecutorRule()
5     private val testCoroutineDispatcher = TestCoroutineDispatcher()
6     private val testCoroutineScope = TestCoroutineScope(testCoroutineDispatcher)
```

```
7       private val testRepository = FakeResultRepository()
8       // Class under test
9       private lateinit var viewModel: MainActivityViewModel
10
11      @Before
12      fun setup() {
13          viewModel = MainActivityViewModel(testRepository, testCoroutineScope)
14          testRepository.shouldThrowError = false
15      }
16
17      @After
18      fun teardown() {
19          testCoroutineScope.cleanupTestCoroutines()
20      }
21
22      @Test
23      fun getNextResult_showsResult() {
24          // Arrange
25          assertNull(viewModel.result.value)
26          val expectedResult = FakeResultRepository.fakeResult
27          // Act
28          viewModel.getNextResult()
29          val actualResult = viewModel.result.value
30          // Assert
31          assertEquals(expectedResult, actualResult)
32      }
33
34      @Test
35      fun getNextResult_showsAndHidesProgressBar() {
36          // Arrange
37          assertFalse(viewModel.progressVisible.value!!)
38          testCoroutineScope.pauseDispatcher()
39          // Act
40          viewModel.getNextResult()
41          // Assert
42          assertTrue(viewModel.progressVisible.value!!)
43          testCoroutineScope.resumeDispatcher()
44          assertFalse(viewModel.progressVisible.value!!)
45      }
46
47      @Test
48      fun getNextResult_showsError() {
49          // Arrange
50          assertNull(viewModel.result.value)
51          testRepository.shouldThrowError = true
52          val expectedResult = "Failed to get result."
53          // Act
54          viewModel.getNextResult()
55          val actualResult = viewModel.result.value
56          // Assert
57          assertEquals(expectedResult, actualResult)
58      }
59  }
```

2.9. Usporedba dretvi, RxJave i korutina

Poglavlja 2.5, 2.6 i 2.7 predstavila su mehanizme asinkronosti na Androidu, njihove prednosti i nedostatke te preporučene načine korištenja. Jer se nativne Android aplikacije mogu razvijati Kotlinom i Javom, postoji dosta mehanizama asinkronosti od kojih se može birati, toliko da programerima može biti problem odlučiti se za jednog od njih. U ovom su poglavlju, tablicom 2.3, predstavljena ključna obilježja prema kojima se kvalitativno procjenjuje svaki od predstavljenih mehanizama.

U Java aplikacijama se mogu koristiti obične dretve ili RxJava. Obične dretve su prikladne kad projekt nema kompleksne asinkrone zahtjeve. Primjerice, ako ne postoji potreba za

Obilježje	Dretve	RxJava	Korutine
Proces učenja	srednje lagan	težak	srednje težak
Čitkost koda	loša	dobra	odlična
Složenost prebacivanja dretvi	srednje niska	srednja	niska
Složenost slijednih asinkronih poslova	visoka	srednja	niska
Podrška za Javu	da	da	ne
Ručno rukovanje skupovima dretvi	da	ne	ne
Složenost otkazivanja asinkronih poslova	visoka	niska	niska
Složenost i opširnost biblioteke	srednje niska	visoka	srednja
Složenost skaliranja	visoka	niska	niska
Službena preporuka	ne	ne	da
Podrška za asinkrone tokove	ne	da	da, uz Flow

Tab. 2.3: *Tablična usporedba dretvi, RxJave i korutina*

dubokim ulančavanjem asinkronih poziva, kompleksnom istodobnošću ili skaliranjem, sasvim su dovoljne i drže kod jednostavnim i čitkim. Inače se može koristiti RxJava jer pojednostavljuje kompleksan asinkroni kod, izbjegava pakao povratnih poziva i lako se skalira. U Kotlin aplikacijama se, osim dretvi i RxJave, mogu koristiti i korutine. One kombiniraju prednosti dretvi i RxJave zbog čega se, u pravilu, mogu koristiti umjesto njih. Za asinkrone tokove se u korutinama može koristiti biblioteka Kotlin Flow [66].

3. PRIMJENA MEHANIZAMA PARALELIZMA U IZRADI ANDROID APLIKACIJE

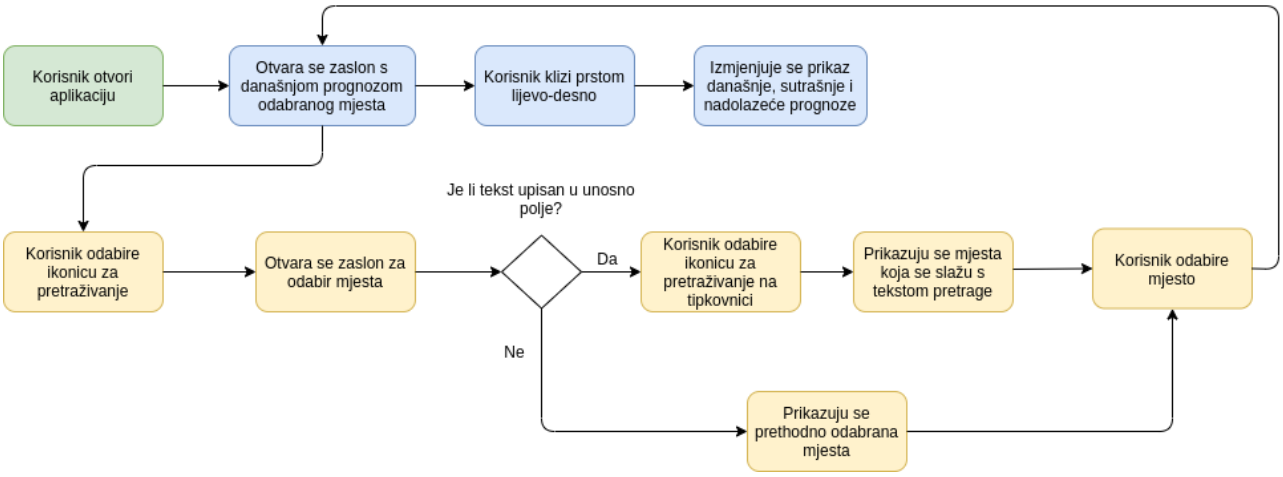
Poglavlje 2 daje pregled mehanizama istodobnosti i asinkronosti koji se mogu koristiti na Androidu. Na kraju ih uspoređuje prema kvalitativnim kriterijima i zaključuje da su korutine preporučeni mehanizam za aplikacije pisane u Kotlinu. Ovo poglavlje opisuje i prikazuje implementacijske detalje i testove potpune aplikacije za vremensku prognozu koja koristi korutine za rasterećivanje glavne dretve i bolje performanse.

3.1. Opis problema i prikaz rješenja na visokoj razini

Najveća trgovina Android aplikacijama Google Play Store sadrži mnoštvo aplikacija za vremensku prognozu. One najpopularnije često imaju lijep, jednostavan i pregledan dizajn [67] zbog kojeg su privlačne širokoj lepezi korisnika, čemu su dokaz brojevi preuzimanja koji idu i u milijune. Njihov nedostatak, u očima korisnika kojima je stalo do privatnosti, je nedostupnost izvornog koda, reklame i moguća zloupraba osobnih podataka. Naime, aplikacija može ispunjavati svoju svrhu i pružati vrijednost korisniku, a pokraj toga prikazivati reklame, skriveno obrađivati osobne podatke, prodavati ih nepoznatim uslugama, skupljati analitičke podatke i sl. Alternativa takvim aplikacijama su aplikacije s trgovine F-Droid koja sadrži samo aplikacije izgrađene iz otvorenog koda [68], čime osigurava njihovu transparentnost i zadovoljava korisnike kojima je privatnost u interesu. Osim toga, označava tzv. protu-obilježja (engl. *anti-features*) poput ne-slobodnih (engl. *non-free*) mrežnih usluga, skupljanja analitičkih podataka, ne-slobodnih ovisnosti, reklama itd. Izbor aplikacija za vremensku prognozu na F-Droidu, u usporedbi s Google Play Storeom, znatno je ograničen (oko 20, u trenutku pisanja) i većina ih ima neintuitivan i nepregledan dizajn i/ili protu-obilježja poput ne-slobodnih mrežnih usluga ili ne-slobodnog uzvodnog (engl. *upstream*) koda. Zbog tih je problema u sklopu ovog rada stvorena slobodna aplikacija otvorenog koda (engl. *Free and Open-Source Software*, FOSS) za prikaz vremenske prognoze i nazvana je "Prognoza". Po svojim je značajkama i mogućnostima slična ostalim aplikacijama za vremensku prognozu, ali ju razlikuje korištenje slobodnih mrežnih usluga, materijalni dizajn (engl. *Material design*) i otvoren kod što omogućava buduće postavljanje na F-Droid bez protu-obilježja. Dijagram 3.1 prikazuje tok korištenja navedene aplikacije na visokoj razini. Podatke o vremenskoj prognozi nekog mjesta potrebno je dohvatiti s neke mrežne usluge i, po mogućnosti, spremiti podatke lokalno kako bi se izbjeglo prekomjerno trošenje resursa. Korutine se koriste kako bi se tijekom ovih aktivnosti spriječilo zastajkivanje i iskoristile dostupne obradbene jedinice uređaja.

3.2. Specifikacija zahtjeva

Aplikacija je namijenjena prosječnim korisnicima koji žele vidjeti osnovne podatke o trenutnim i nadolazećim vremenskim uvjetima kako bi se znali pripremiti za njih prije, npr., izlaska iz kuće. Trebala bi, na brzi pregled, dati odgovore na pitanja kao: što obući, treba li ponijeti kišobran,



Sl. 3.1: *Dijagram toka aplikacije Prognoza*

hoće li za tri dana vrijeme biti pogodno za kupanje, je li vjetar previše jak za bicikliranje i sl. Nije namijenjena detaljnom pregledu vremenskih uvjeta koje bi trebali meteorolozi ili drugi znanstvenici. U skladu s tim grubim opisom, ovo poglavlje navodi zahtjeve na svaku funkcionalnost aplikacije.

Današnja prognoza Prikazana u obliku prvog zaslona kojeg korisnik vidi ulaskom u aplikaciju. Svrha mu je predstaviti trenutne i njima neposredne vremenske uvjete koji trebaju biti lako i brzo pregledni kako bi korisnik što brže dobio potrebne informacije. One najvažnije trebaju biti istaknute na vrhu i opisane su tablicom 3.1.

Informacija	Opis	Format	Komentar
Vrijeme	Vrijeme podataka	12. kolovoza, 16:23	Format ovisi o sistemskom lokalitetu
Temperatura zraka	Trenutna temperatura zraka	20°	*Bez komentara*
Osjetna temperatura	Osjet temperature zraka u ovisnosti o vlazi i brzini vjetra	23°	Izračunati lokalno prema formulama za *wind chill* i *heat index*
Prognoza oborina	Prognoza oborina za sljedeća dva sata.	"Bez oborina u sljedeća dva sata" ili "4.3 mm oborina u sljedeća dva sata"	Granica iznad koje se količina oborina smatra značajnom je 0.1 mm.
Ikonica	Ikonica koja predstavlja trenutne vremenske uvjete	Slika oblaka, sunca i sl.	*Bez komentara*

Tab. 3.1: *Istaknute informacije današnje prognoze*

Informacije o budućim satima trebaju biti dostupne ispod trenutnih uvjeta u obliku liste kartica. Trebaju početi od trenutnog sata i završiti u 6:00 sljedećeg jutra. Informacije koje kartica treba sadržavati opisane su tablicom 3.2. I ova lista treba biti lako i brzo pregledna, zbog čega kartice u zadanom stanju trebaju prikazati samo najvažnije informacije. Klikom na karticu, trebaju se otkriti detaljne informacije koje su u tablici označene zvjezdicom.

Sutrašnja prognoza Drugi najvažniji zaslon koji treba predstaviti vremenske uvjete sutrašnjeg dana. Kao i današnja prognoza, treba se sastojati od istaknute cjeline koja predstavlja

Informacija	Opis	Format	Komentar
Sat	Sat podataka	16:00	Uvijek prikazan u nultoj minuti
Temperatura zraka	Temperatura zraka u tom satu	20°	*Bez komentara*
Ikonica	Ikonica koja predstavlja trenutne vremenske uvjete	Slika oblaka, sunca i sl.	*Bez komentara*
* Opis ikonice	Tekstualni opis prikazane ikonice	"Pretežito oblačno", "Sunčano" i sl.	*Bez komentara*
* Osjetna temperatura	Osjet temperature zraka u ovisnosti o vlazi i brzini vjetra	23°	Izračunati lokalno prema formulama za *wind chill* i *heat index*
* Vjetar	Brzina vjetra i smjer iz kojeg puše	4.3 km/h, S	*Bez komentara*
* Vlaga	Udio vlažnosti zraka	43%	*Bez komentara*
* Pritisak	Pritisak zraka	1009 hPa	*Bez komentara*

Tab. 3.2: *Informacije u kartici sata*

najvažnije informacije i liste sati, koja treba biti iste građe, ali gdje sati trebaju početi od 7:00 ujutro i završiti u 6:00 sljedećeg jutra. Informacije koje trebaju biti u istaknutoj cjelini prikazane su tablicom 3.3, a informacije koje trebaju biti u karticama sati prethodno su opisane tablicom 3.2.

Prognoza nadolazećih dana Ovaj zaslon treba dati jednostavan pregled sažetaka vremenske prognoze za nadolazeće dane. Treba se sastojati od liste kartica koje, kao i kartice sati, u zadanom stanju prikazuju samo najvažnije informacije. Klikom na karticu, trebaju se otkriti detaljne informacije. Ove su informacije prikazane tablicom 3.4 i detaljne su označene zvjezdicom.

Pretraga i odabir mjesta Podaci o vremenskoj prognozi se vežu za određenu geografsku dužinu i širinu. Kako korisnik ne bi morao ručno tražiti i upisivati ove podatke za željeno mjesto, treba moći pretražiti mjesta prema imenu grada, države, četvrti, ulice i sl. Nakon odabira mjesta, treba se prikazati njegova prognoza. Prethodno odabrana mjesta trebaju biti dostupna bez ponovnog pretraživanja. Mjesta trebaju biti prikazana listom kartica čije su informacije opisane tablicom 3.5.

Način rada bez mreže i spremanje podataka Način rada bez mreže treba biti omogućen spremanjem podataka o vremenskoj prognozi i mjestima u lokalnu bazu podataka. Značajke koje su nedostupne bez mreže su ažurna prognoza i pretraživanje novih mjesta. Kad se korisniku prikazuju zastarjeli podaci, treba biti obaviješten o razlogu u obliku poruke, npr. ako su podaci zastarjeli jer se zahtjev ne može napraviti bez spajanja na Internet.

Rukovanje pogreškama Tijekom rada aplikacije mogu se pojaviti pogreške u komunikaciji s odabranim mrežnim uslugama i bazom podataka ili preslikavanju iz entiteta baze u objekte

Informacija	Opis	Format	Komentar
Vrijeme	Vrijeme na kojeg se podaci odnose	četvrtak, 12. kolovoza	Format ovisi o sistemskom lokalitetu
Najviša temperatura zraka	Najviša temperatura zraka od svih sati dana	33°	*Bez komentara*
Najniža temperatura zraka	Najniža temperatura zraka od svih sati dana	21°	*Bez komentara*
Ikonica pretežitih vremenskih uvjeta	Ikonica koja opisuje pretežite vremenske uvjete dana	Slika oblaka, sunca i sl.	Uzeti u obzir samo ne-noćne ikonice.
Opis pretežitih vremenskih uvjeta	Tekstualni opis pretežitih vremenskih uvjeta	"Pretežito sunčano", "Sunčano" i sl.	*Bez komentara*
Prognoza oborina	Prognoza oborina cijelog dana	"Bez oborina" ili "4.3 mm oborina"	Granica iznad koje se količina oborina smatra značajnom je 0.1 mm.

Tab. 3.3: *Istaknute informacije sutrašnje prognoze*

Informacija	Opis	Format	Komentar
Vrijeme	Vrijeme na kojeg se podaci odnose	četvrtak, 12. kolovoza	Format ovisi o sistemskom lokalitetu
Najviša temperatura zraka	Najviša temperatura zraka od svih sati dana	33°	*Bez komentara*
Najniža temperatura zraka	Najniža temperatura zraka od svih sati dana	21°	*Bez komentara*
Ikonica pretežitih vremenskih uvjeta	Ikonica koja opisuje pretežite vremenske uvjete dana	Slika oblaka, sunca i sl.	Prilikom određivanja pretežite ikonice uzeti u obzir samo ne-noćne ikonice.
Opis pretežitih vremenskih uvjeta	Tekstualni opis pretežitih vremenskih uvjeta	"Oblačno", "Sunčano" i sl.	*Bez komentara*
Prognoza oborina	Prognoza oborina cijelog dana	"Bez oborina" ili "4.3 mm oborina"	Granica iznad koje se količina oborina smatra značajnom je 0.1 mm.
* Najveća brzina vjetra	Najveća brzina vjetra tijekom dana i smjer iz kojeg puše	10.1 km/h, N	*Bez komentara*
* Najveći udio vlage	Najveći udio vlage tijekom dana	80%	*Bez komentara*
* Najveći pritisak	Najveći pritisak zraka tijekom dana	1050 hPa	*Bez komentara*

Tab. 3.4: *Informacije prognoze nadolazećih dana*

Informacija	Opis	Format	Komentar
Ime mjesta	Jednostavno ime mjesta	Osijek, New York, Hong Kong itd.	*Bez komentara*
Puno ime mjesta	Puno ime mjesta s državom, poštanskim brojem, županijom itd.	"Kutina, Grad Kutina, Sisačko-moslavačka županija, 44320, Hrvatska"	*Bez komentara*
Oznaka odabranog mjesta	Ikonica koja označava je li ovo mjesto prethodno odabrano	Slika sata	*Bez komentara*

Tab. 3.5: *Informacije o mjestu*

sučelja. Poruke greške se trebaju prikazati samo kad je to nužno i trebaju omogućiti ponovan pokušaj, primjerice kad ne postoje podaci koji se mogu prikazati zbog davnog ažuriranja ili pogreške u komunikaciji s uslugama. Uvijek je bolje oporaviti se od pogrešaka prikazivanjem dostupnih podataka iz baze, računajući pritom na mirno razrješenje pogreške u budućnosti.

Podrška za konfiguracijske promjene Aplikacija treba podržavati tamnu i svijetlu temu i orijentacijske promjene. Nakon konfiguracijske promjene, podaci trebaju ostati isti i ne smiju se ponovno dohvaćati ili preslikavati u podatke sučelja, čime se izbjegava nepotrebno trošenje resursa.

Slobodne mrežne usluge i otvoren kod Zbog budućeg postavljanja na trgovinu aplikacijama F-Droid, korištene mrežne usluge za prognozu i pretraživanje mjesta moraju biti slobodne za korištenje i sav kod aplikacije treba biti otvoren.

3.3. Korišteni alati i tehnologije

Opisana aplikacija napisana je u Kotlinu i strukturirana prema MVVM arhitekturnom obrascu. Pritom su korištene tehnologije:

- Room – biblioteka za bazu podataka. Apstrakcijski sloj relacijske baze podataka SQLite koji omogućava čitak i potpun pristup bazi. Od glavnih značajki nudi verifikaciju SQL upita tijekom prevođenja i manje ponavljanja koda pomoću pred-definiranih SQL upita dostupnih u anotacijama [69].
- Coroutines Android – biblioteka za korutine na Androidu koja, osim korutina, daje i dispečer glavne Android dretve [47].
- Retrofit, OkHttp i Gson [70] – biblioteke za komunikaciju s mrežom i pretvaranje rezultata iz JSON (engl. *JavaScript Object Notation*) formata u Kotlin klase i obratno.
- Koin – okvir za ubrizgavanje ovisnosti koji omogućava jednostavno testiranje i fleksibilnost prilikom pisanja, promjene i nadogradnje koda.

- Desugar JDK Libs – biblioteka koja omogućava korištenje podskupa klasa i metoda novijih verzija Jave na starijim [71]. U ovom se projektu koristi za pristup Javinoj "Time" biblioteci iz verzije 8 koja je potrebna za rukovanje datumima i vremenskim zonama.
- JUnit4 okvir za testiranje.
- Biblioteke za testiranje Rooma, Koina i Androidovih arhitekturnih komponenti.
- Ostale biblioteke uključujući ekstenzije arhitekturnih komponenti, Material, ViewPager2, ViewBinding itd.

Za podatke o vremenskoj prognozi odabran je API meteorološke organizacije MET Norway [72] koja svoje podatke licencira Norveškom licencom za otvorene vladine podatke [73]. Ona dozvoljava slobodno dijeljenje i mijenjanje podataka pod uvjetom da se jasno navede odakle podaci potiču. Za pristup ovim podacima nije potrebno imati račun, ali postoji nekoliko pravila koja se moraju slijediti prilikom korištenja njihovog API-ja [74]:

- Na mjestima gdje se prikazuju podaci jasno navesti da potiču od MET Norway.
- Uz zahtjeve priložiti identifikacijsko zaglavlje (engl. *header*) `User-Agent`. Treba se sastojati od informacija o imenu aplikacije i organizacije, kontaktnoj e-mail adresi i (po volji) broju mobitela radi kontaktiranja u slučaju prekomjernog korištenja API-ja.
- Biti štedljiv sa zahtjevima. Poimence:
  - Raditi novi zahtjev tek ako je prošlo vrijeme isteka podataka naznačeno zaglavljem `Expires` zadnjeg zahtjeva.
  - Spremati podatke lokalno.
  - Novim zahtjevima priložiti zaglavlje `If-Modified-Since` s vrijednosti zaglavlja `Last-Modified` prošlog zahtjeva kako bi se izbjeglo nepotrebno preuzimanje podataka. Ako je ovo prvi zahtjev, ne priložiti navedeno zaglavlje.
  - Ako je potrebno periodično pozivati zahtjeve, rasporediti ih u nejednake vremenske intervale.
  - Zaokružiti geografsku širinu i dužinu na najviše četiri decimale.
  - Ne ažurirati kontinuirano mobilne aplikacije dok se ne koriste, npr. u svrhu obavijesti.
- Držati se unutar 20 zahtjeva po sekundi što se broji na osnovi cijele aplikacije, ne po klijentu. Za sve preko toga se treba tražiti odobrenje. S ovom aplikacijom to vjerojatno neće biti problem jer je skup korisnika F-Droida relativno malen u usporedbi s Google Play Storeom.

Za podatke o geografskim mjestima odabran je API organizacije OpenStreetMap Nominatim [75]. Podaci su licencirani ODbL licencom koja također dopušta slobodno dijeljenje i mijenjanje podataka sve dok se jasno naglasi njihovo porijeklo [76]. Za pristup ovim podacima također nije potrebno imati račun, ali postoji nekoliko pravila korištenja [77]:

- Najviše jedan zahtjev po sekundi po aplikaciji.
- Zahtjevima priložiti zaglavlje `User-Agent`.
- Jasno navesti da podaci potiču od OpenStreetMap Nominatim.
- Spremati podatke lokalno kako bi se spriječilo prekomjerno pozivanje API-ja.
- Ne periodično raditi zahtjeve.
- Ne implementirati automatsku pretragu tijekom unosa teksta. Umjesto toga, praviti zahtjev tek na pritisak gumba za pretragu.

Glavno ograničenje navedenih API-ja je zabrana prekomjernih zahtjeva, ali to će biti kompenzirano lokalnim spremanjem podataka i ažuriranjem samo kad je to potrebno. Preporučeni i pravilniji način bio bi postavljanje vlastitog poslužitelja koji se spaja na API-je MET Norwaya i Nominatima i sprema rezultate koje distribuira klijentima, odnosno instancama aplikacije Prognoza. Time bi se uvelike smanjilo opterećenje na poslužitelje navedenih organizacija. Takva arhitektura je izvan opsega ovog rada i nepotrebna s obzirom na relativno malen broj predviđenih korisnika. Međutim, ako se zahtjevi aplikacije povećaju do spomenutih granica, bit će potrebno stupiti u kontakt s navedenim organizacijama i dogovoriti se oko uvjeta daljnjeg korištenja.

3.4. Prikaz programskog rješenja

Aplikacija je stvorena prema zahtjevima pomoću navedenih tehnologija. Ovo će poglavlje prikazati najvažnije implementacijske detalje od dohvaćanja podataka do njihovog prikaza na sučelju. Cijeli programski kod je dostupan u prilogu.

3.4.1 Dohvaćanje i spremanje podataka

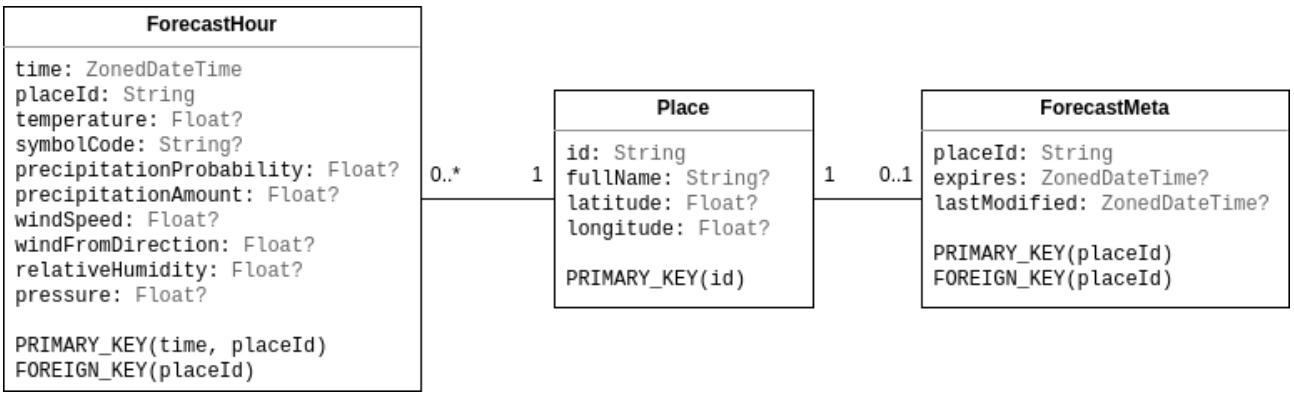
Komponenta odgovorna za dohvaćanje podataka prognoze s REST API-ja je sučelje `ForecastService`, koje je prikazano isječkom 3.1. Ruta s koje se dohvaćaju podaci je `locationforecast/2.0/compact` koja vraća smanjen opseg podataka od potpune prognoze, ali on je sasvim dovoljan za potrebe prosječnog korisnika. Struktura povratnog JSON objekta može se vidjeti na poveznici [78]. Korijenska podatkovna klasa je `LocationForecastResponse` i sastoji se od ugniježđenih podatkovnih klasa koje odgovaraju strukturi JSON-a. Pretvorbu iz JSON-a u Kotlin objekt Retrofit radi automatski pomoću Gson biblioteke, tako da je odgovornost programera samo navesti rutu, parametre i povratni tip u kojeg se JSON treba serijalizirati. Retrofit

ima i podršku za korutine koja se iskoristila dodavanjem ključne riječi `suspend`. Sada se zahtjev može sigurno i bez blokiranja pozvati iz bilo koje korutine. Parametri se, osim obavezne geografske dužine i širine, sastoje od propisanih zaglavlja iz uvjeta korištenja API-ja.

Primjer koda 3.1: *API usluga za dohvaćanje prognoze*

```
interface ForecastService {
    @GET("locationforecast/2.0/compact")
    suspend fun getCompactLocationForecast(
        @Header("User-Agent") userAgent: String,
        @Header("If-Modified-Since") ifModifiedSince: String?,
        @Query("lat") latitude: String,
        @Query("lon") longitude: String
    ): Response<LocationForecastResponse>
}
```

Dobiveni se podaci preslikavaju u entitete i spremaju u bazu koja se sastoji od tri tablice čiji su odnosi predstavljeni UML dijagramom 3.2. Podaci prognoze predstavljeni su entitetima `ForecastHour` koji sadrže vremenske informacije nekog mjesta u određenom satu. Mjesta su predstavljena entitetima `Place` i, osim svog punog imena, sadrže geografsku dužinu i širinu koji se koriste za dohvaćanje prognoze. Metapodaci prognoze sadrže podatke o vremenu njezinog isteka i zadnje promjene. Predstavljeni su entitetom `ForecastMeta`.



Sl. 3.2: *UML dijagram baze podataka*

Komponenta odgovorna za ažuriranje baze s novim podacima je `DefaultForecastRepository` i njezine najvažnije metode prikazane su isječkom 3.2. Njezino sučelje izlaže metode za dohvaćanje današnje, sutrašnje i nadolazeće prognoze i brisanje isteklih podataka. Navedene metode za dohvaćanje prognoza u pozadini sve koriste metodu `getForecastHours` koja je zadužena za odluku o ažuriranju baze i vraćanje spremljenih podataka. Kao što isječak prikazuje, metoda koristi metapodatke kako bi provjerila jesu li podaci prognoze istekli. Ako jesu, radi zahtjev i ažurira bazu prije dohvaćanja i preslikavanja podataka u objekt `ForecastResult` kojeg gornji slojevi mogu obrađivati bez brige o iznimkama. Zahtjevu se u metodi `updateForecastDatabase` prosljeđuje zaglavlje `If-Modified-Since` kako bi API mogao vratiti prazan odgovor ukoliko podaci nisu promijenjeni od vremena naznačenog zaglavljem. Ovim se mehanizmima štedi resurse poslužitelja i korisnika jer se izbjegavaju nepotrebni mrežni pozivi i preuzimanje podataka. Asinkronost osiguravaju Room i Retrofit biblioteke koje imaju podršku za korutine.

Primjer koda 3.2: *Najvažniji kod DefaultForecastRepository klase*

```
class DefaultForecastRepository(
    private val forecastService: ForecastService,
    private val forecastDao: ForecastHourDao,
    private val placeRepository: PlaceRepository,
    private val metaRepository: MetaRepository,
    private val dispatcherProvider: DispatcherProvider
) : ForecastRepository {
    ...
    private suspend fun getForecastHours(
        start: ZonedDateTime,
        end: ZonedDateTime,
        placeId: String
    ): ForecastResult {
        var meta = try {
            metaRepository.get(placeId)
        } catch (e: Exception) {
            null
        }
        var error: ForecastError? = null
        if (meta?.hasExpired() != false) {
            try {
                updateForecastDatabase(placeId, meta?.lastModified)
                meta = metaRepository.get(placeId)
            } catch (e: HttpException) {
                error = handleHttpException(e)
            } catch (e: SQLiteException) {
                error = DatabaseError(e)
            } catch (e: Exception) {
                error = Unknown(e)
            }
        }
        return try {
            val hours = forecastDao.getForecastHours(start, end, placeId)
            hours.toForecastResult(meta, error)
        } catch (e: SQLiteException) {
            Empty(DatabaseError(e))
        } catch (e: Exception) {
            Empty(Unknown(e))
        }
    }

    private suspend fun updateForecastDatabase(placeId: String, lastModified:
        ↪ ZonedDateTime?) {
        val forecastPlace = placeRepository.get(placeId) ?: placeRepository.
            ↪ getDefaultPlace()
        val lastModifiedTimestamp = ForecastMetaDateTimeConverter.toTimestamp(
            ↪ lastModified)
        val forecastResponse = forecastService.getCompactLocationForecast(
            userAgent = USER_AGENT,
            ifModifiedSince = lastModifiedTimestamp,
            latitude = format("%.2f", forecastPlace.latitude),
            longitude = format("%.2f", forecastPlace.longitude)
        )
        updateForecastMeta(forecastResponse.headers(), forecastPlace.id)
        updateForecastHours(forecastResponse.body(), forecastPlace.id)
    }

    ...
}
```

Sučelje `ForecastResult` prikazano je isječkom 3.3. Ključna riječ `sealed` ga označava kao zapečaćeno sučelje čije su sve podklase poznate u vrijeme prevođenja. U Kotlinu se ovakve klase većinom koriste za iscrpne `when` blokove jer, ako su u njemu pokrivene sve podklase, nije potrebno uključiti i `else` izraz. Postoje tri podklase, odnosno tri slučaja rezultata repozitorija:

- `Success` – normalan slučaj kad su podaci dohvaćeni iz baze jer još nisu istekli ili kad su ažurirani i zatim dohvaćeni iz baze.

- `CachedSuccess` – slučaj kad su podaci dohvaćeni iz baze jer je došlo do pogreške prilikom ažuriranja podataka. Razlog pogreške predstavljen je `ForecastError` parametrom prema kojem gornji slojevi mogu prikazati poruku pogreške.
- `Empty` – slučaj kad nema podataka koji se mogu prikazati s neobaveznim razlogom.

Primjer koda 3.3: *Zapečaćeno sučelje ForecastResult*

```
sealed interface ForecastResult

data class Success(
    val meta: ForecastMeta?,
    val hours: List<ForecastHour>
) : ForecastResult

data class CachedSuccess(
    val success: Success,
    val reason: ForecastError?
) : ForecastResult

data class Empty(
    val reason: ForecastError?,
) : ForecastResult
```

3.4.2 Preslikavanje iz entiteta baze u objekte sučelja

Kako pogled ne bi ovisio o entitetima baze, trebaju se preslikati u objekte sučelja. Tada se može u potpunosti promijeniti unutarnji način rada baze, pružatelja prognoze ili ViewModela, a korisničko sučelje neće znati za to, što daje fleksibilnost tijekom razvoja. Podaci vremenske prognoze prikazuju se u fragmentima za današnju, sutrašnju i nadolazeću prognozu. Svaki od njih ima svoj ViewModel koji dohvaća podatke iz repozitorija, preslikava ih u objekte sučelja i rukuje pogreškama. Preslikani objekti im se dostavljaju putem `LiveData` objekata na koje se pretplaćuju i reagiraju postavljanjem podataka na svoje poglede. Tijekom razvoja postalo je očito da ViewModeli dijele većinu logike, pa je ona izdvojena u posebnu nadklasu `BaseForecastFragmentViewModel` koja je prikazana isječkom 3.4. Ona je odgovorna za rukovanje uspješnim, spremljenim i praznim prognozama, odlučivanje o ponovnom učitavanju podataka, prikazivanje trake napretka i prikazivanje poruke o spremljenim podacima. Podklase su odgovorne samo za dohvaćanje (metoda `getNewForecast`) i preslikavanje (metoda `mapToForecastUiModel`) podataka, jer se jedino ova ponašanja razlikuju od zaslona do zaslona.

Primjer koda 3.4: *Nadklasa svih ViewModela za prikaz prognoze*

```
abstract class BaseForecastFragmentViewModel<T: ForecastUiModel>(
    coroutineScope: CoroutineScope?,
    protected val preferencesRepository: PreferencesRepository
): CoroutineScopeViewModel(coroutineScope) {
    private var currentMeta: ForecastMeta? = null
    abstract val _forecast: MutableLiveData<T>
    private val _emptyScreen = MutableLiveData<EmptyForecastUiModel?>()
    private val _cachedResultsMessage = MutableLiveData<Event<Int?>>()
    private val _isLoading = MutableLiveData(false)

    // public LiveDatas of MutableLiveDatas omitted for brevity

    fun getForecast() {
        coroutineScope.launch {
            if (isReloadNeeded()) {
```

```
                _isLoading.value = true
                when (val result = getNewForecast()) {
                    is Success -> handleSuccess(result)
                    is CachedSuccess -> handleCachedSuccess(result)
                    is Empty -> handleEmpty(result)
                }
                _isLoading.value = false
            }
        }
    }

    protected abstract suspend fun getNewForecast(): ForecastResult

    protected abstract suspend fun mapToForecastUiModel(success: Success): T

    private suspend fun handleSuccess(success: Success) {
        _forecast.value = mapToForecastUiModel(success)
        currentMeta = success.meta
        _emptyScreen.value = null
    }

    private fun handleEmpty(empty: Empty) {
        _emptyScreen.value = EmptyForecastUiModel(empty.reason?.toErrorResourceId())
    }

    private suspend fun handleCachedSuccess(cachedResult: CachedSuccess) {
        handleSuccess(cachedResult.success)
        _cachedResultsMessage.value = Event(cachedResult.reason?.toErrorResourceId())
    }

    private suspend fun isReloadNeeded(): Boolean {
        return currentMeta?.hasExpired() != false ||
                currentMeta?.placeId != preferencesRepository.getSelectedPlaceId() ||
                _forecast.value == null
    }
}
```

Korutine dolaze do izražaja upravo u ViewModelu. Pozivom metode `getForecast` pokreće se korutina koja osigurava da se svo dohvaćanje podataka, bilo s mreže ili iz baze, odvija na pozadinskim dretvama i time ne blokira glavnu dretvu. Podklase definiraju preslikavanje entiteta u objekte sučelja pri čemu koriste istodobnost kako bi smanjile ukupno potrebno vrijeme izvršenja operacija. Jedan jednostavan primjer prikazan je isječkom 3.5 gdje se podaci preslikavaju u objekte sučelja današnje prognoze. Definiraju se tri korutinska graditelja `async` koji se odmah počinju izvršavaju istodobno. Pozivom metode `await` prilikom izgradnje objekta sučelja, na njihove se rezultate čeka samo onoliko koliko je potrebno najdužem bloku da završi (u slučaju uređaja s barem tri jezgre). Pri tom je osigurana i strukturirana istodobnost jer je korišten `coroutineScope` definiran u nadklasi.

Primjer koda 3.5: *Preslikavanje podataka za današnju prognozu*

```
override suspend fun mapToForecastUiModel(success: Success): TodayForecastUiModel {
    val currentHourAsync = coroutineScope.async(dispatcherProvider.default) {
        success.hours[0].toHourUiModel().copy(time = ZonedDateTime.now())
    }
    val otherHoursAsync = coroutineScope.async(dispatcherProvider.default) {
        success.hours.map { it.toHourUiModel() }
    }
    val precipitationForecastAsync = coroutineScope.async(dispatcherProvider.default) {
        val total = success.hours.subList(0, 2).totalPrecipitationAmount()
        if (total <= 0f) null else total
    }
    return TodayForecastUiModel(
        currentHour = currentHourAsync.await(),
        otherHours = otherHoursAsync.await(),
        precipitationForecast = precipitationForecastAsync.await()
    )
```

```
}
```

Kompleksniji primjer istodobnosti nalazi se u metodi `toDayUiModel` koja preslikava listu sati u sažetak dnevne prognoze. Prikazana je isječkom 3.6 gdje se može vidjeti da stvara čak sedam `async` blokova koji se izvršavaju istodobno. Ovo je moguće jer su izračuni svakog od parametara međusobno nezavisni, pa se cijeli postupak izvršava u vremenu najdužeg izračuna (u slučaju uređaja s barem sedam jezgri). Strukturirana istodobnost je ovdje omogućena prosljeđivanjem korutinskog opsega, a izdvajanje u posebnu metodu bilo je potrebno jer se ovo preslikavanje koristi na više mjesta u kodu.

Primjer koda 3.6: *Preslikavanje podataka za sažetak prognoze jednog dana*

```
suspend fun List<ForecastHour>.toDayUiModel(coroutineScope: CoroutineScope): DayUiModel {
    val weatherIconAsync = coroutineScope.async { representativeWeatherIcon() }
    val lowTempAsync = coroutineScope.async { minTemperature() }
    val highTempAsync = coroutineScope.async { maxTemperature() }
    val precipitationAsync = coroutineScope.async { totalPrecipitationAmount() }
    val hourWithMaxWindSpeedAsync = coroutineScope.async { hourWithMaxWindSpeed() }
    val maxHumidityAsync = coroutineScope.async { maxHumidity() }
    val maxPressureAsync = coroutineScope.async { maxPressure() }
    val firstHour = get(0)
    return DayUiModel(
        id = "${firstHour.placeId}-${firstHour.time}",
        time = firstHour.time,
        representativeWeatherIcon = weatherIconAsync.await(),
        lowTemperature = lowTempAsync.await(),
        highTemperature = highTempAsync.await(),
        totalPrecipitationAmount = precipitationAsync.await(),
        maxWindSpeed = hourWithMaxWindSpeedAsync.await()?.windSpeed,
        windFromCompassDirection = hourWithMaxWindSpeedAsync.await()?.windFromDirection
            ?.toCompassDirection(),
        maxHumidity = maxHumidityAsync.await(),
        maxPressure = maxPressureAsync.await()
    )
}
```

3.4.3 Postavljanje podataka na sučelje

Fragmenti prognoza reagiraju na `LiveData` objekte svojih ViewModela. Jedina netipična stvar kod postavljanja preslikanih podataka na sučelje je način na koji se liste postavljaju u `RecyclerView`. Umjesto klasičnog adaptera koji ručno poziva metode osvježavanja liste, ova aplikacija koristi `ListAdapter` s klasom `DiffUtil` koja sama izračunava potrebne operacije kako bi se stara lista pretvorila u novu. Time se dobiva na jednostavnosti implementacije adaptera jer je potrebno samo proslijediti novu listu, nakon čega se potrebne promjene automatski izračunavaju i primjenjuju. Dobiva se i na efikasnosti, jer algoritam uvijek pronalazi najmanji skup takvih promjena [79]. Takav je adapter implementiran za listu sati i prikazan isječkom 3.7. Klasa koja nasljeđuje `DiffUtilCallback` odgovorna je za određivanje jednakosti dva elementa i jednakosti njihovog sadržaja. U metodi `areContentsTheSame` dozvoljeno je upotrijebiti jednostavan operator `==` jer je klasa `HourUiModel` podatkovna klasa čija je `equals` metoda implementirana tako da vraća istinu samo ako svi parametri imaju jednaku vrijednost, za razliku od normalnih klasa koje bi prema zadanom ponašanju vratile istinu samo kad su reference objekata iste.

Primjer koda 3.7: *Adapter za prikaz satne prognoze s klasom DiffUtil*

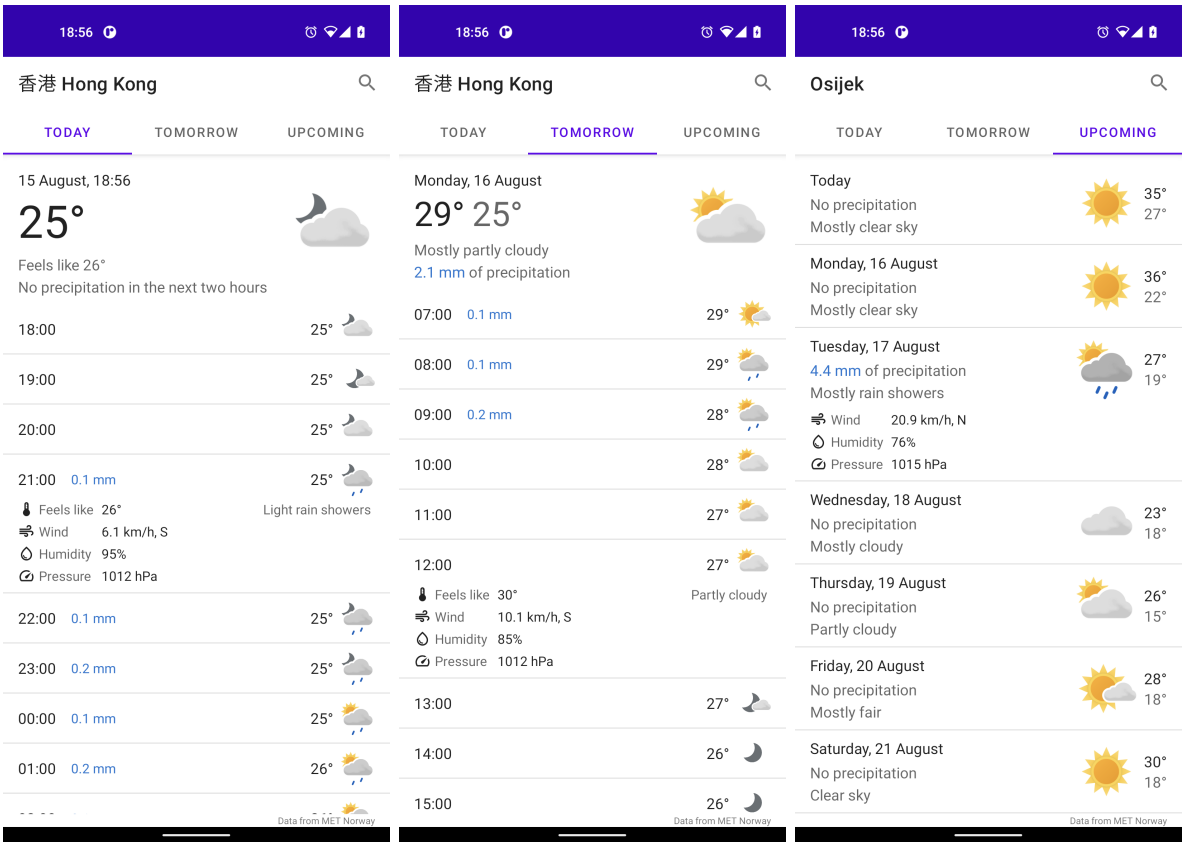
```
class HoursRecyclerViewAdapter : ListAdapter<HourUiModel, HourViewHolder>(
    ↪ HourDiffCallback()) {
    private val onItemClickCallback = object : (Int) -> Unit {
        override fun invoke(position: Int) {
            val itemAtPosition = getItem(position)
            itemAtPosition.isExpanded = !itemAtPosition.isExpanded
            notifyItemChanged(position)
        }
    }

    override fun onBindViewHolder(holder: HourViewHolder, position: Int) {
        holder.bind(getItem(position))
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): HourViewHolder {
        val binding = CellHourBinding.inflate(LayoutInflater.from(parent.context),
            ↪ parent, false)
        return HourViewHolder(binding, onItemClickCallback)
    }
}

class HourDiffCallback : DiffUtil.ItemCallback<HourUiModel>() {
    override fun areContentsTheSame(oldItem: HourUiModel, newItem: HourUiModel): Boolean
        ↪  {
        return oldItem == newItem
    }

    override fun areItemsTheSame(oldItem: HourUiModel, newItem: HourUiModel): Boolean {
        return oldItem.id == newItem.id
    }
}
```

3.4.4 Način rada aplikacije

Nakon otvaranja aplikacije, prikazuje se današnja prognoza koja se može vidjeti na slici 3.3a. Prema zahtjevima, u prvom planu se nalazi vrijeme, temperatura, osjetna temperatura, ikonica i prognoza oborina. Preglednost sljedećih sati postignuta je jednostavnim karticama koje sadrže samo osnovne informacije. Informacija o količini oborina istaknuta je plavom bojom i prikazana samo ako je predviđena značajna količina oborina. Klikom na karticu, ona se proširuje i otkriva dodatne informacije o osjetnoj temperaturi, vjetru, vlazi i pritisku. Sutrašnja prognoza izgleda slično i prikazana je slikom 3.3b. Jedina je razlika u istaknutim informacijama u kojima se nalazi najviša i najniža temperatura, prognoza oborina i pretežiti vremenski uvjeti. Prognoza nadolazećih dana prikazuje sažetke koji prikazuju slične informacije istaknutim informacijama sutrašnje prognoze, ali imaju i mogućnost proširenja čime se otkrivaju dodatne informacije o najvećim vrijednostima brzine vjetra, vlage i pritiska. Prikazana je slikom 3.3c. Kao i do sad, oborine su istaknute plavom bojom zbog lakše preglednosti.

Zaslon pretraživanja mjesta otvara se u skočnom prozoru pritiskom ikonice za pretraživanje i prikazan je slikom 3.4a. Kad tekst pretrage nije unesen, prikazuju se prethodno spremljena mjesta za brz odabir. Odabirom gumba za pretraživanje na tipkovnici, prikazuju se mjesta koja API vrati. Korisnik ih može odabrati, čime se zatvara skočni prozor i ažuriraju zasloni prognoza s podacima za to mjesto. Korisnik može i otkazati odabir mjesta navigacijom unazad ili klikom izvan skočnog prozora.

Kad dođe do pogreške od koje se aplikacija može oporaviti prikazom spremljenih rezultata, uz njih se prikazuje i skočni `Snackbar` pogled s gumbom koji omogućava prikaz razloga zbog



(a) *Današnja prognoza* **(b)** *Sutrašnja prognoza* **(c)** *Nadolazeća prognoza*

Sl. 3.3: *Prikaz vremenske prognoze*

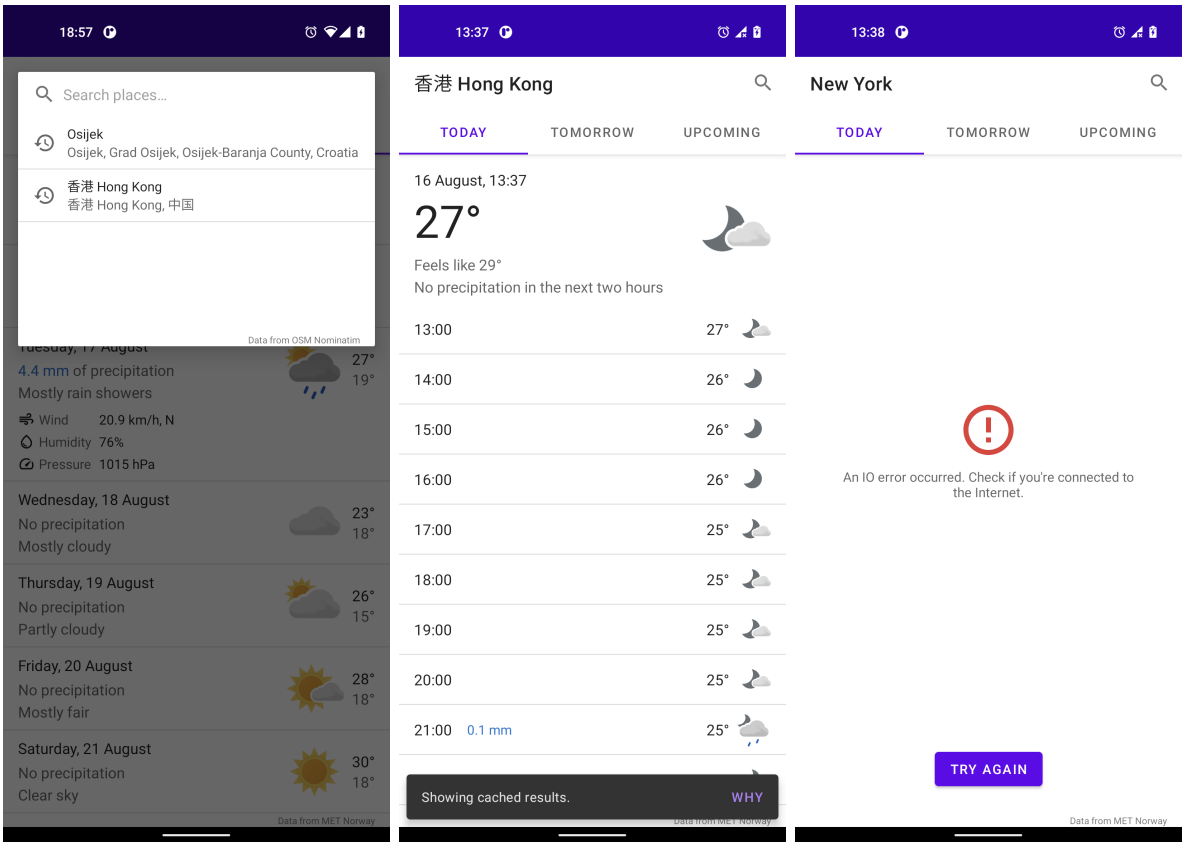
kojeg se prikazuju spremljeni, zastarjeli rezultati. Ovaj je slučaj prikazan slikom 3.4b. Kad se aplikacija ne može oporaviti od pogreške, prikazuje se pogled praznog stanja preko cijelog zaslona s porukom pogreške i gumbom za ponovni pokušaj, prikazan slikom 3.4c.

3.5. Testiranje programskog rješenja

Iako se aplikacija ne sastoji od puno zaslona i nema puno mogućnosti, ovisi o bazi podataka i dva nezavisna pružatelja mrežnih usluga, što daje mnoštvo rubnih slučajeva za testiranje. Međutim, jer je za testiranje Room baze podataka potrebno pisati instrumentirane testove i jer programeri nemaju pristup testnim API-jima navedenih mrežnih usluga, takvi slučajevi neće biti prikazani u ovom radu. Umjesto toga, fokus će biti na jediničnom testiranju ViewModela i preslikavanja gotovih podataka iz testnog duplikata klase `ForecastRepository`. Ovo su jedne od najvažnijih funkcionalnosti aplikacije, pokraj dohvaćanja i spremanja podataka, i u njima se pokrivaju dva slučaja koja mogu nastati kod testiranja korutina: kad test poziva kod koji interno pokreće korutinu i kad test pokreće korutinu kako bi ispitao zaustavnu funkciju.

3.5.1 Testni duplikati

Kako bi se ponašanje ViewModela testiralo bez ovisnosti o stvarnoj bazi podataka ili mreži, potrebno je ubrizgati testne duplikate odgovarajućih komponenti, što je lako jer svi napisani



(a) *Pretraživanje mjesta* **(b)** *Spremljeni rezultati* **(c)** *Nema podataka*

Sl. 3.4: *Prikaz pretrage mjesta i pogrešaka*

ViewModeli ovise o apstrakcijama. Duplikat sučelja `PreferencesRepository` implementiran je tako da uvijek vraća isti identifikator odabranog mjesta i da metoda spremanja novog odabranog mjesta ne radi ništa. Duplikat sučelja `DispatcherProvider` implementiran je tako da su svi dispečeri jednaki onom koji je ubrizgan preko konstruktora, što omogućava ubrizgavanje testnog dispečera. Najkompleksniji testni duplikat je `FakeForecastRepository`, koji je prikazan isječkom 3.8.

Dužnost ovog repozitorija je generiranje podataka o prognozi koji su predstavljeni objektima klase `Success`, `CachedSuccess` ili `Empty`. Prva ideja koja se može pojaviti je instanciranje lažnih podataka ručnom konstrukcijom podatkovnih klasa, ali to nije praktično. Klasa `Success` sadrži objekt klase `ForecastMeta` kojeg nije problem ručno instancirati jer se sastoji od tri podatka, ali sadrži i listu objekata klase `ForecastHour` koja se sastoji od puno podataka. Ručno instanciranje ovih objekata oduzelo bi previše vremena, ne bi vjerodostojno predstavilo stvarne podatke i bilo bi nečitko. Umjesto toga, ruta se ručno pozvala koristeći web-stranicu [78] i dobiveni JSON objekt se kopirao u datoteku, koja je kopirana u `resources` direktorij testnog direktorija. Takvog ga se može otvoriti i učitati u niz znakova kojeg Gson serijalizira u Kotlin objekte, što se događa u metodi `getData`. Nakon toga ostaje preslikavanje u objekte klase `ForecastHour` i omotavanja u jednu od podklasa zapečaćenog sučelja `ForecastResult`, što prikazuje metoda `getForecastHours`. Omogućeno je i vraćanje spremljenog i praznog rezultata, što se kontrolira varijablom `typeOfResultToReturn`. Ovim se postupcima

vjerodostojno duplicira ponašanje stvarnog repozitorija, izbjegava rukovanje bazom i mrežom i ubrzava vrijeme izvršavanja testova.

Pri vrhu klase, može se vidjeti statička varijabla `now`. Ona služi kao referentna točka koja omogućava ponovljivost testova. Stvarni repozitorij dohvaća podatke o prognozi koji počinju od trenutnog vremena određenog metodom `ZonedDateTime#now`, što je u redu jer će korisnik (vjerojatno) biti spojen na Internet relativno često, pa će se baza imati priliku ažurirati svježim, trenutnim podacima. Međutim, testni repozitorij koristi fiksirane podatke koji se nikad neće automatski ažurirati. Zato je testovima koji će koristiti ovaj duplikat potrebna informacija o referentnom početnom vremenu podataka. U suprotnom bi test mogao za referentno vrijeme koristiti trenutni datum koji bi u trenutku pisanja davao očekivane rezultate, a za dva-tri tjedna padao.

Primjer koda 3.8: *Važan kod testnog duplikata repozitorija*

```
class FakeForecastRepository : ForecastRepository {
    companion object {
        val now: ZonedDateTime = ZonedDateTime.parse("2021-08-16T20:00:00Z")
    }
    private val hoursAfterMidnightToShow = 6L
    var typeOfResultToReturn: Class<*> = Success::class.java
    ...
    private suspend fun getForecastHours(
        start: ZonedDateTime,
        end: ZonedDateTime,
        placeId: String
    ): ForecastResult {
        val response = getData()
        val hours = response.body()!!.forecast.forecastTimeSteps
            .filter {
                val time = ZonedDateTime.parse(it.time)
                time in start..end
            }
            .map {
                it.toForecastHour(TEST_PLACE_ID)
            }
        val success = Success(null, hours)
        return when(typeOfResultToReturn) {
            CachedSuccess::class.java -> CachedSuccess(success, null)
            Empty::class.java -> Empty(null)
            Success::class.java -> success
            else -> throw IllegalStateException("Result type $typeOfResultToReturn not
                ↪ recognized.")
        }
    }

    private fun getData(): Response<LocationForecastResponse> {
        val json = readFileFromResources("osijek_16_08_21_response.json")
        val body = Gson().fromJson(json, LocationForecastResponse::class.java)
        val headers = Headers.headersOf(
            "Expires", "Mon, 16 Aug 2021 21:18:38 GMT",
            "Last-Modified", "Mon, 16 Aug 2021 20:47:40 GMT"
        )
        return Response.success(body, headers)
    }

    private fun readFileFromResources(fileName: String): String {...}
}
```

3.5.2 Testiranje ViewModela današnje prognoze

Testiranjem ViewModela ispituje se jesu li podaci postavljeni na izložene `LiveData` objekte u skladu s očekivanim podacima nakon neke operacije. Ovdje je odabran ViewModel današnje

prognoze i prikazana su tri najvažnija slučaja: rukovanje uspješnim, spremljenim i praznim rezultatom. Klasa u kojoj se nalaze testovi postavljena je s testnim duplikatima i metodama koje se izvršavaju prije i poslije svakog testa, što je prikazano isječkom 3.9. Jer ViewModel sam pokreće svoje korutine u ubrizganom korutinskom opsegu, ovi testovi neće koristiti `runBlockingTest`. Umjesto toga, slijedno izvršavanje korutinskog koda omogućeno je ubrizgavanjem testnog korutinskog opsega s testnim dispečerom preko konstruktora ViewModela.

Primjer koda 3.9: *Klasa za testiranje ViewModela današnje prognoze*

```
@ExperimentalCoroutinesApi
class TodayFragmentViewModelTest {
    @get:Rule
    val instantTaskExecutorRule = InstantTaskExecutorRule()
    private val coroutineDispatcher = TestCoroutineDispatcher()
    private val coroutineScope = TestCoroutineScope(coroutineDispatcher)
    private val forecastRepository = FakeForecastRepository()
    private val preferencesRepository = FakePreferencesRepository()
    private val dispatcherProvider = FakeDispatcherProvider(coroutineDispatcher)
    // class under test
    private lateinit var viewModel: TodayFragmentViewModel

    @Before
    fun setup() {
        viewModel = TodayFragmentViewModel(
            coroutineScope,
            forecastRepository,
            dispatcherProvider,
            preferencesRepository
        )
    }

    @After
    fun teardown() {
        coroutineScope.cleanupTestCoroutines()
    }
    ...
}
```

Rukovanje uspješnim rezultatom Kod rukovanja uspješnim rezultatom, provjerava se samo vrijednost `forecast` objekta ViewModela. Kako bi se utvrdila ispravnost vremenske prognoze, potrebno je utvrditi da su podaci u skladu s očekivanim vremenskim rasponom. Odnosno, prvi sat u listi treba biti postavljen na trenutni sat i zadnji na sutra u 6:00 ujutro. Primjer 3.10 prikazuje implementaciju ovog testa. U postavi se provjerava je li `forecast` prethodno inicijaliziran i repozitorijeva se varijabla `typeOfResultToReturn` postavlja na `Success` kako bi vratio uspješan rezultat. Zatim se poziva metoda `getForecast` i provjerava opisani vremenski raspon podataka.

Primjer koda 3.10: *Testiranje rukovanja uspješnim rezultatom*

```
@Test
fun getForecast_whenSuccess_showsForecast() {
    // Arrange
    assertTrue {
        viewModel.forecast.value == null
    }
    forecastRepository.typeOfResultToReturn = Success::class.java
    // Act
    viewModel.getForecast()
    // Assert
    val now = FakeForecastRepository.now
    val hours = viewModel.forecast.value?.otherHours
```

```
    val firstHour = hours?.getOrNull(0)
    val lastHour = hours?.getOrNull(hours.lastIndex)
    assertTrue("First hour is start of now") {
        firstHour?.time == now.withMinute(0)
    }
    assertTrue("Last hour is tomorrow at 6AM") {
        val isTomorrow = lastHour?.time?.minusDays(1)?.toLocalDate() == now.toLocalDate
            ↪ ()
        val is6Am = lastHour?.time?.hour == 6
        isTomorrow && is6Am
    }
}
```

Rukovanje spremljenim rezultatom Testiranje rukovanja spremljenim rezultatom vrši se sličan način, samo se ovdje dodatno provjerava vrijednost objekta `cachedResultsMessage` kojem je svrha obavijestiti korisnika o razlogu iz kojeg mu se prikazuju spremljeni podaci. Primjer 3.11 prikazuje kod ovog testa.

Primjer koda 3.11: *Testiranje rukovanja spremljenim rezultatom*

```
@Test
fun getForecast_whenCached_showsMessageAndForecast() {
    // Arrange
    assertTrue {
        viewModel.cachedResultsMessage.value == null
    }
    assertTrue {
        viewModel.forecast.value == null
    }
    forecastRepository.typeOfResultToReturn = CachedSuccess::class.java
    // Act
    viewModel.getForecast()
    // Assert
    assertTrue("Is forecast shown") {
        viewModel.forecast.value != null
    }
    assertTrue("Is cached results notification shown") {
        viewModel.cachedResultsMessage.value != null
    }
}
```

Rukovanje praznim rezultatom Testiranje rukovanja praznim rezultatom samo ispituje je li `emptyForecast` objekt postavljen na neku vrijednost nakon operacije, a pretpostavlja se da će odgovarajući pogled reagirati u skladu s tim i prikazati prazan zaslon. Primjer 3.12 prikazuje kod ovog testa.

Primjer koda 3.12: *Testiranje rukovanja praznim rezultatom*

```
@Test
fun getForecast_whenEmpty_showsEmptyScreen() {
    // Arrange
    assertTrue {
        viewModel.emptyScreen.value == null
    }
    forecastRepository.typeOfResultToReturn = Empty::class.java
    // Act
    viewModel.getForecast()
    // Assert
    assertTrue("Is forecast not shown") {
        viewModel.forecast.value == null
    }
    assertTrue("Is empty screen shown") {
        viewModel.emptyScreen.value != null
    }
```

```
}
```

3.5.3 Testiranje preslikavanja u sažetak dana

Metoda preslikavanja u sažetak dana, prikazana isječkom 3.6, je zaustavna metoda koja koristi predani korutinski opseg za stvaranje i čekanje na rezultat internih `async` blokova. Zbog toga je testove potrebno pokrenuti korutinskim graditeljem `runBlockingTest` u testnom korutinskom opsegu i proslijediti taj opseg metodi kako bi interni `async` blokovi također bili vezani za njega. Prikazana su tri testa: preslikavanje normalnih, ništavnih (engl. *null*) i miješanih vrijednosti. Testna klasa prikazana je isječkom 3.13. Prikazane (sažete) metode stvaraju testne objekte klase `ForecastHour` i njima odgovarajuće očekivane objekte klase `DayUiModel` kako bi se preslikavanje moglo testirati i kako se testne metode ne bi zagađivale instanciranjem objekata.

Primjer koda 3.13: *Testiranje rukovanja praznim rezultatom*

```
@ExperimentalCoroutinesApi
class DayUiModelMappingTest {
    @get:Rule
    val instantTaskExecutorRule = InstantTaskExecutorRule()
    private val coroutineDispatcher = TestCoroutineDispatcher()
    private val coroutineScope = TestCoroutineScope(coroutineDispatcher)
    ...
    private fun getForecastHours_withNormalValues(start: ZonedDateTime) = listOf(
        ForecastHour(
            time = start,
            placeId = TEST_PLACE_ID,
            temperature = 18f,
            symbolCode = "clearsky_day",
            precipitationProbability = null,
            precipitationAmount = 0f,
            windSpeed = 15.4f,
            windFromDirection = 230f,
            relativeHumidity = 90f,
            pressure = 1020f
        ),
        ForecastHour(...),
        ForecastHour(...),
    )

    private fun getForecastHours_withAllNullValues(time: ZonedDateTime): List<
        ↪ ForecastHour> {
        val nullHour = ForecastHour(...)
        return listOf(nullHour, nullHour, nullHour)
    }

    private fun getExpectedDayUiModel_forNormalValues(start: ZonedDateTime): DayUiModel
        ↪ =
        DayUiModel(
            id = "$TEST_PLACE_ID-$start",
            time = start,
            representativeWeatherIcon = RepresentativeWeatherIcon(
                weatherIcon = WEATHER_ICONS["clearsky_day"]!!,
                isMostly = false
            ),
            lowTemperature = 18f,
            highTemperature = 33f,
            maxWindSpeed = 15.4f,
            windFromCompassDirection = R.string.direction_sw,
            totalPrecipitationAmount = 6.3f + 7.87f + 0f,
            maxHumidity = 90f,
            maxPressure = 1020f,
            isExpanded = false
        )

    private fun getExpectedDayUiModel_forAllNullValues(start: ZonedDateTime) =
        ↪ DayUiModel(...)
```

```
}
```

Preslikavanje normalnih i ništavnih vrijednosti Za ispitivanje preslikavanja normalnih i ništavnih vrijednosti potrebno je samo odrediti je li stvarni rezultat preslikavanja jednak očekivanom za dani ulaz. Ulazne podatke pruža metoda `getForecastHours_withNormalValues`, a njima odgovarajuće izlazne podatke metoda `getExpectedDayUiModel_forNormalValues`. Podatke za ništavne vrijednosti pružaju njima analogne metode. Primjer 3.14 prikazuje ova dva testa.

Primjer koda 3.14: *Testiranje preslikavanja normalnih i ništavnih vrijednosti*

```
@Test
fun toDayUiModel_mapsNormalValues() = coroutineScope.runBlockingTest {
    // Arrange
    val start = ZonedDateTime.now()
    val hours = getForecastHours_withNormalValues(start)
    // Act
    val actual = hours.toDayUiModel(this)
    // Assert
    val expected = getExpectedDayUiModel_forNormalValues(start)
    assertTrue { actual == expected }
}

@Test
fun toDayUiModel_mapsNullValues() = coroutineScope.runBlockingTest {
    // Arrange
    val start = ZonedDateTime.now()
    val nullHours = getForecastHours_withAllNullValues(start)
    // Act
    val actual = nullHours.toDayUiModel(this)
    // Assert
    val expected = getExpectedDayUiModel_forAllNullValues(start)
    assertTrue { actual == expected }
}
```

Otpornost na ništavne vrijednosti Prema dokumentaciji API-ja prognoze na poveznici [78], skoro svaki od parametara može poprimiti ništavnu vrijednost. Važno je osigurati da se takvi podaci ignoriraju pri izračunu maksimalne temperature, brzine vjetra i slično. Upravo tome služi ovaj test, koji je prikazan isječkom 3.15. On kombinira normalne podatke s ništavnim i na kraju utvrđuje je li preslikani objekt jednak onom koji bi se dobio kad bi svi podaci bili normalni.

Primjer koda 3.15: *Testiranje preslikavanja normalnih i ništavnih vrijednosti*

```
@Test
fun toDayUiModel_isResistantToNullValues() = coroutineScope.runBlockingTest {
    // Arrange
    val start = ZonedDateTime.now()
    val hours = getForecastHours_withNormalValues(start) +
            getForecastHours_withAllNullValues(start)
    // Act
    val actual = hours.toDayUiModel(this)
    // Assert
    val expected = getExpectedDayUiModel_forNormalValues(start)
    assertTrue { actual == expected }
}
```

4. ZAKLJUČAK

Ovim su radom predstavljene dretve u Javi, biblioteka RxJava i korutine u Kotlinu kao tri najvažnija mehanizma paralelizma za razvoj Android aplikacija. U dretvama su, osim osnovnog pristupa, pokriveni i kompleksniji pristupi kao Javin `ExecutorService` i Androidov API za dijeljenje poruka gdje je bilo riječi o njihovim čestim problemima i anti-obrascima. Glavni problemi s ovim mehanizmima su klasa `AsyncTask`, pakao povratnih poziva, manjak skalabilnosti i nespretno rukovanje otkazivanjem zadataka uslijed promjene u životnom ciklusu roditeljske komponente. Unatoč tome, kao preporučeni obrazac korištenja predstavljena je arhitektura s globalno dostupnim `ExecutorService`om kojem se dostavljaju svi asinkroni zadaci. Ovakav je obrazac pogodan za korištenje u aplikacijama s malenim potrebama istodobnosti.

Nakon toga, predstavljena je RxJava kao deklarativna apstrakcija dretvenih mehanizama i prebacivanja između dretvi, gdje su također pokriveni česti problemi i anti-obrasci. Njezin glavni problem je strma krivulja učenja, nova paradigma programiranja i zlouporaba API-ja. No, to ne znači da je loš mehanizam, samo da je potrebno pomno razmisliti o njegovom uključenju u projekt i razviti plan učenja za sudionike na projektu. Kad se ova biblioteka koristi pravilno, rješava pakao povratnih poziva i predstavlja kompleksnu istodobnost i prebacivanje dretvi na jednostavan i čitak način. Preporučeni obrazac korištenja ju smješta u MVVM arhitekturu gdje se pretplata na promotrive tokove vrši u ViewModelu.

Kao zadnji mehanizam, predstavljene su korutine u Kotlinu koje kombiniraju apstrakciju dretvenosti s normalnim, direktnim stilom blokirajućeg koda. Pokriven je njihov unutarnji način rada i terminologija i opisan je način korištenja kao i česti problemi i anti-obrasci. Njihov je glavni problem relativno strma krivulja učenja koja, iako ne tako strma kao RxJavina, zahtijeva edukaciju i plan učenja. Nakon usvajanja načina korištenja, problemi se mogu pojaviti u obliku krivog korištenja API-ja čime se kod čini težim za čitanje i testiranje. Nakon toga je pokriveno i testiranje korutina jer su one službeno preporučeni mehanizam istodobnosti, a testiranje je važan sastavni dio svake aplikacije.

Na kraju su ova tri mehanizma uspoređena prema kvalitativnim kriterijima gdje je zaključeno da su korutine preporučeni mehanizam istodobnosti za razvoj modernih Android aplikacija u Kotlinu. U skladu s time napravljena je i testirana potpuna Android aplikacija za vremensku prognozu koja za svoje potrebe preslikavanja podataka i komunikacije s mrežom i lokalnom bazom podataka koristi korutine. Time rasterećuje glavnu dretvu i koristi dostupne obradbene jedinice kako bi se operacije izvršile što brže. Kao rezultat, izbjegava se zastajkivanje sučelja i postiže brz prikaz podataka, što su sastavni dijelovi dobrog korisničkog iskustva.

U daljnjem radu, cjelokupna arhitektura aplikacije mogla bi se poboljšati stvaranjem posebnog poslužitelja kako bi se smanjilo opterećenje na poslužitelje odabranih organizacija. Aplikaciji bi se mogle dodati nove funkcionalnosti, kao brisanje odabranih mjesta ili prikaz alatnog bloka (engl. *widget*) na početnom zaslonu. Mogla bi se obraditi i Kotlin Flow biblioteka i njezini česti problemi, anti-obrasci te preporučeni način korištenja i usporedba s RxJavom.

LITERATURA

[1] Ian T Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley, 1995.

[2] The New York Times: Intel Halts Development Of 2 New Microprocessors. https://www.nytimes.com/2004/05/08/business/intel-halts-development-of-2-new-microprocessors.html. Pristup: 29.04.2021.

[3] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[4] Guinness World Records: First dual-core smartphone. https://www.guinnessworldrecords.com/world-records/first-dual-core-smartphone. Pristup: 29.04.2021.

[5] Rob Pike: Concurrency Is Not Parallelism. https://vimeo.com/groups/waza2012/videos/49718712. Pristup: 04.05.2021.

[6] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC, 2018.

[7] Kotlin dokumentacija hashCode() metode. https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-any/hash-code.html. Pristup: 14.04.2021.

[8] Material Design. https://material.io. Pristup: 17.04.2021.

[9] Android Developers: ANRs. https://developer.android.com/topic/performance/vitals/anr. Pristup: 17.04.2021.

[10] Android Developers: Slow Rendering. https://developer.android.com/topic/performance/vitals/render. Pristup: 17.06.2021.

[11] Anandtech: Hot Chips 2018: The Google Pixel Visual Core Live Blog (10am PT, 5pm UTC). https://www.anandtech.com/show/13241/hot-chips-2018-the-google-pixel-visual-core-live-blog. Pristup: 17.04.2021.

[12] GSMArena: Google Pixel 5 long-term review. https://www.gsmarena.com/google_pixel_5_long_term-review-2225p5.php. Pristup: 17.04.2021.

[13] Jonathan Anderson, John McRee, Robb Wilson, et al. *Effective UI: The art of building great user experience in software*. " O'Reilly Media, Inc.", 2010.

[14] Android Developers: Overview of memory management. https://developer.android.com/topic/performance/memory-overview. Pristup: 17.04.2021.

[15] Anders Goransson. *Efficient Android Threading: Asynchronous Processing Techniques for Android Applications.* " O'Reilly Media, Inc.", 2014.

[16] Android Developers: Better performance through threading. https://developer.android.com/topic/performance/threads. Pristup: 17.04.2021.

[17] Android Developers: Guide to background processing. https://developer.android.com/guide/background. Pristup: 05.05.2021.

[18] Android SDK: Izvorni kod Activity.java klase. https://github.com/AndroidSDKSources/android-sdk-sources-for-api-level-30/blob/master/android/app/Activity.java. Pristup: 07.05.2021.

[19] Android SDK: Izvorni kod View.java klase. https://github.com/AndroidSDKSources/android-sdk-sources-for-api-level-30/blob/master/android/view/View.java. Pristup: 07.05.2021.

[20] Android Developers: Jetpack. https://developer.android.com/jetpack. Pristup: 01.07.2021.

[21] Android SDK: Dokumentacija AsyncTask klase. https://developer.android.com/reference/android/os/AsyncTask. Pristup: 10.05.2021.

[22] Reddit.com: Menadžer Android Toolkit/Jetpack tima o AsyncTasku i motivaciji iza nekih odluka. https://www.reddit.com/r/androiddev/comments/bty3au/whats_wrong_with_asynctask/ep4dsv4/. Pristup: 14.05.2021.

[23] Java: Executor dokumentacija. https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/Executor.html. Pristup: 19.05.2021.

[24] Android SDK: Izvorni kod Looper.java klase. https://github.com/AndroidSDKSources/android-sdk-sources-for-api-level-30/blob/master/android/os/Looper.java. Pristup: 07.05.2021.

[25] Android SDK: Izvorni kod MessageQueue.java klase. https://github.com/AndroidSDKSources/android-sdk-sources-for-api-level-30/blob/master/android/os/MessageQueue.java. Pristup: 07.05.2021.

[26] Okhttp3: Dokumentacija. https://square.github.io/okhttp/4.x/okhttp/okhttp3/. Pristup: 17.05.2021.

[27] Retrofit 2: Dokumentacija. https://square.github.io/retrofit/2.x/retrofit/. Pristup: 17.05.2021.

[28] Android developers: Guide to app architecture. https://developer.android.com/jetpack/guide. Pristup: 18.05.2021.

[29] Android Developers: Running Android tasks in background threads. https://developer.android.com/guide/background/threading. Pristup: 19.05.2021.

[30] Koin – The Kotlin Injection Framework. https://insert-koin.io/. Pristup: 19.05.2021.

[31] Tomasz Nurkiewicz and Ben Christensen. *Reactive Programming with RxJava: Creating Asynchronous, Event-based Applications.* " O'Reilly Media, Inc.", 2016.

[32] ReactiveX: Uvod. http://reactivex.io/intro.html. Pristup: 08.06.2021.

[33] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. Elements of reusable object-oriented software. *Design Patterns. massachusetts: Addison-Wesley Publishing Company*, 1995.

[34] RxJava: Alphabetical List of Observable Operators. https://github.com/ReactiveX/RxJava/wiki/Alphabetical-List-of-Observable-Operators. Pristup: 15.06.2021.

[35] ReactiveX: Scheduler. http://reactivex.io/documentation/scheduler.html. Pristup: 15.06.2021.

[36] RxAndroid: Reactive Extensions for Android. https://github.com/ReactiveX/RxAndroid. Pristup: 16.06.2021.

[37] Christoper Arriola and A Huang. Reactive programming on android with rxjava. *MYNAH Software*, 2017.

[38] Jake Wharton: RxBinding - RxJava binding APIs for Android's UI widgets. https://github.com/JakeWharton/RxBinding. Pristup: 20.06.2021.

[39] Kotlin: Extensions. https://kotlinlang.org/docs/extensions.html. Pristup: 20.06.2021.

[40] K Matt Dupree. *RxJava for Android App Development.* O'Reilly (USA), 2016.

[41] David J Wheeler. The use of sub-routines in programmes. In *Proceedings of the 1952 ACM national meeting (Pittsburgh)*, pages 235–236, 1952.

[42] Gerald Jay Sussman and Guy L Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.

[43] F. Babić and N. Srivastava. *Kotlin Coroutines by Tutorial.* Razeware LLC, 2019.

[44] Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., third edition, 1997.

[45] GitHub: KEEP - Kotlin Evolution and Enhancement Process: Službeni dokument prijedloga implementacije korutina u Kotlinu. https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md. Pristup: 24.06.2021.

[46] labs.pedrofelix.org: Suspending functions, coroutines and state machines. https://labs.pedrofelix.org/guides/kotlin/coroutines/coroutines-and-state-machines. Pristup: 25.06.2021.

[47] Android Developers: Kotlin coroutines. https://developer.android.com/kotlin/coroutines. Pristup: 27.06.2021.

[48] Android Developers: Lifecycle. https://developer.android.com/jetpack/androidx/releases/lifecycle. Pristup: 03.07.2021.

[49] GitHub: kotlinx.coroutines: HandlerDispatcher.kt. https://github.com/Kotlin/kotlinx.coroutines/blob/master/ui/kotlinx-coroutines-android/src/HandlerDispatcher.kt. Pristup: 03.07.2021.

[50] GitHub: kotlinx.coroutines: Executors.kt. https://github.com/Kotlin/kotlinx.coroutines/blob/master/kotlinx-coroutines-core/jvm/src/Executors.kt. Pristup: 01.07.2021.

[51] kotlinx.coroutines: CoroutineExceptionHandler. https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-exception-handler/. Pristup: 29.06.2021.

[52] kotlinx.coroutines: Job. https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html. Pristup: 28.06.2021.

[53] kotlinx.coroutines: withContext. https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/with-context.html. Pristup: 29.06.2021.

[54] GitHub: kotlinx.coroutines. https://github.com/Kotlin/kotlinx.coroutines. Pristup: 27.06.2021.

[55] kotlinx.coroutines: Deferred. https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-deferred/. Pristup: 30.06.2021.

[56] Roman Elizarov: Blocking threads, suspending coroutines. https://elizarov.medium.com/blocking-threads-suspending-coroutines-d33e11bf4761. Pristup: 01.07.2021.

[57] Android Developers: Best practices for coroutines in Android. https://developer.android.com/kotlin/coroutines/coroutines-best-practices. Pristup: 01.07.2021.

[58] Build testable apps for Android (Google I/O'19). https://www.youtube.com/watch?v=VJi2vmaQe6w. Pristup: 04.07.2021.

[59] JUnit4. https://junit.org/junit4/. Pristup: 06.07.2021.

[60] Android Developers: Testing. https://developer.android.com/training/testing. Pristup: 06.07.2021.

[61] Robolectric: test-drive your Android code. http://robolectric.org/. Pristup: 06.07.2021.

[62] Android Developers: Build instrumented tests. https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests. Pristup: 06.07.2021.

[63] Advanced Android in Kotlin 05.3: Testing Coroutines and Jetpack integrations. https://developer.android.com/codelabs/advanced-android-kotlin-training-testing-survey. Pristup: 06.07.2021.

[64] kotlinx.coroutines.test: TestCoroutineDispatcher. https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-test/kotlinx.coroutines.test/-test-coroutine-dispatcher/. Pristup: 06.07.2021.

[65] kotlinx.coroutines.test: TestCoroutineScope. https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-test/kotlinx.coroutines.test/-test-coroutine-scope/. Pristup: 06.07.2021.

[66] Asynchronous Flow. https://kotlinlang.org/docs/flow.html. Pristup: 14.07.2021.

[67] Android Central: Best Weather Apps for Android 2021. https://www.androidcentral.com/best-weather-apps-android. Pristup: 09.08.2021.

[68] F-Droid: Docs. https://f-droid.org/docs/. Pristup: 09.08.2021.

[69] Android Developers: Save data in a local database using Room. https://developer.android.com/training/data-storage/room/. Pristup: 12.08.2021.

[70] GitHub: Gson. https://github.com/google/gson. Pristup: 12.08.2021.

[71] GitHub: desugar_jdk_libs. https://github.com/google/desugar_jdk_libs. Pristup: 12.08.2021.

[72] MET Norway: developer.yr.no. https://developer.yr.no/. Pristup: 12.08.2021.

[73] Norwegian Digitalisation Agency: Norwegian Licence for Open Government Data (NLOD) 2.0. https://data.norge.no/nlod/en/2.0/. Pristup: 12.08.2021.

[74] MET Norway: API Terms of Service. https://developer.yr.no/TermsOfService. Pristup: 12.08.2021.

[75] Nominatim: Open-source geocoding with OpenStreetMap data. https://nominatim.org/. Pristup: 12.08.2021.

[76] Open Data Commons Open Database License (ODbL). https://opendatacommons.org/licenses/odbl/. Pristup: 12.08.2021.

[77] Nominatim Usage Policy. https://operations.osmfoundation.org/policies/nominatim/. Pristup: 12.08.2021.

[78] MET Norway: LocationForecast 2.0. https://api.met.no/weatherapi/locationforecast/2.0/#/. Pristup: 13.08.2021.

[79] Android Developers: DiffUtil. https://developer.android.com/reference/androidx/recyclerview/widget/DiffUtil. Pristup: 15.08.2021.

SAŽETAK

Rad prvo uvodi pojam paralelizma, istodobnosti i asinkronosti te opisuje zašto su važni za korisničko iskustvo u Android aplikacijama i općenito u aplikacijama s korisničkim sučeljem. Zatim opisuje tri najvažnija mehanizma istodobnosti na Androidu: dretve u Javi, biblioteka RxJava i korutine u Kotlinu. Daje opis njihovog rada, načina korištenja, čestih problema i anti-obrazaca te preporučenog načina korištenja. Kako su korutine trenutno službeno preporučeni mehanizam istodobnosti na Androidu, pokriva se i njihovo jedinično testiranje. Na kraju teorijskog dijela dana je usporedba navedenih mehanizama prema kvalitativnim kriterijima i preporuka mehanizma u ovisnosti o aplikacijskim zahtjevima. U praktičnom dijelu je opisana, implementirana i testirana potpuna aplikacija za vremensku prognozu koja za svoje potrebe istodobnosti koristi korutine. Korutinama se izbjeglo blokiranje glavne dretve i, u slučaju uređaja s više obradbenih jedinica, ubrzalo preslikavanje podataka za prikaz na sučelju.

Ključne riječi: Android, dretve, istodobnost, korutine, paralelizam

ABSTRACT

This thesis first introduces the concepts of parallelism, concurrency and asynchronicity and then describes what makes them so important for user experience in Android applications and GUI applications in general. After that, it describes the three most important concurrency mechanisms on Android: Java threads, the RxJava library and Kotlin coroutines. It describes the way they work, the way they should be used, their common issues and anti-patterns and finally the recommended usage patterns. Being as coroutines are currently the officially recommended concurrency mechanism on Android, their unit testing is also covered. After coroutine testing, a comparison of the three mechanisms is given based on qualitative criteria, and a recommended mechanism is given based on application requirements. A complete weather application that uses coroutines for it's concurrency needs is then described, implemented and tested. With coroutines, the main thread remained unblocked and, when running on a device with multiple processing units, the raw data was more quickly mapped to be displayed on the user interface.

Keywords: Android, concurrency, coroutines, parallelism, threads

ŽIVOTOPIS

David Takač rođen je 22.07.1997. u Beogradu. S tri godine doselio se s roditeljima u Osijek gdje je pohađao osnovnu i srednju školu. U Elektrotehničkoj i prometnoj školi Osijek stekao je strukovnu kvalifikaciju elektrotehničara nakon čega upisuje preddiplomski sveučilišni studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek gdje je stekao akademski naziv sveučilišnog prvostupnika inženjera računarstva. Na istom je fakultetu trenutno student druge godine diplomskog sveučilišnog studija računarstva i radi kao razvojni programer za Android u osječkoj Plavoj tvornici.

PRILOG 1

Na CD-u:

1. Diplomski rad "Usporedba pristupa i obrazaca prilikom implementacije paralelizma na Android platformi" u .zip arhivi LaTeX projekta.

2. Diplomski rad "Usporedba pristupa i obrazaca prilikom implementacije paralelizma na Android platformi" u .pdf formatu.

3. Programski kod aplikacije "Prognoza".