Inverzija ovisnosti kao arhitekturalni konstrukt u dizajnu mobilnih aplikacija

Jalžabetić, Rikardo

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Rikardo Jalžabetić

Inverzija ovisnosti kao arhitekturalni konstrukt u dizajnu mobilnih aplikacija

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Rikardo Jalžabetić

JMBAG: 0016152067

Studij: Informacijski i poslovni sustavi

Inverzija ovisnosti kao arhitekturalni konstrukt u dizajnu mobilnih aplikacija

ZAVRŠNI RAD

Mentor:

Izv. prof. dr. sc. Zlatko Stapić

Varaždin, rujan 2024.

Rikardo Jalžabetić

Izjava o izvornosti

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

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovom temom završnog rada obrađujem što je inverzija ovisnosti (eng. Dependency Inversion), što je injekcija ovisnosti (eng. Dependency Injection) i dajem programski primjer korištenja. Pod inverziju ovisnosti opisujem stvari tipa što je to, čemu služi i koje su joj prednosti i nedostaci. U poglavlju o injekciji ovisnosti dodajem još i primjer korištenja za desktop, web i android aplikacije. Opisujem njen temelj i načine na koje se može koristiti. Isto tako obraćam pažnju na automatsku i ručnu injekciju. Nakon toga govorim o razlikama inverzije i injekcije ovisnosti, zašto one nisu isto, i zašto govorim o injekciji ovisnosti.

Kroz programski dio ovog rada objašnjavam funkcionalne zahtjeve. Prikazujem kako korisnik može koristiti aplikaciju i što sve ona nudi.
Nadalje pažnju stavljam na njen dizajn i njene bitne elemente preko dijagrama klase. I za kraj osvrćem se na bitne stvari iz programskog dijela ovog rada.

Ključne riječi: Dependency inversion principle; DIP; Inverzije ovisnosti; Injekcija ovisnosti; Dependency injection; DI; Android Studio; Kotlin; Interface; Sučelje

Sadržaj

1. Uvod

Svako tko je programirao neko vrijeme susreće se s novim stvarima, pa tako i ja. Ovo je neka nova cjelina u koju prvi puta ulazim i istražujem. Prvenstveno radi širine znanja koju stječem učenjem novih koncepata kao i zabavu primjene naučenog na neku aplikaciju.

Inverzija ovisnosti kao arhitekturalni konstruktor u dizajnu mobilnih aplikacija poslužiti će mi u slučaju problema prilikom programiranja za lakšu izradu aplikacije koja ima više zadataka. Oduvijek sam htio isprobati napraviti aplikaciju kao „Duolingo“, ali nisam znao koje koncepte koristiti. Zato mi je moj mentor izvanredni prof. dr. sc. Zlatko Stapić preporučio koncept inverzije ovisnosti. S ovim konceptom mogu naučiti nešto novo i ostvariti svoju ideju, a ujedno će mi biti od pomoći u rastu i razvoju mene kao programera.

Ovom završnom radu prvenstveno je cilj složiti dobar temelj aplikacije za učenje i ponavljanje (kako olakšati učenje programiranja putem repetitivnog učenja programiranja kroz aplikaciju za mobilne uređaje). Sama primjena inverzije ovisnosti kao arhitekturalnog konstrukta u dizajnu mobilnih aplikacija u ovom slučaju su zadaci koje ostatak koda ima na raspolaganju. Cilj je da se aplikacija s lakoćom može nadograditi i složiti u napredniju i kompleksniju aplikaciju u budućnosti. Ova tema je interesantna i iz razloga:

- da se zadaci mogu jednostavno kreirati u posebnom dijelu i da ga drugi dijelovi koda mogu lako koristiti
- da je netko osmislio koncept inverzije ovisnosti jer mu je trebao i sama primjena tog koncepta
- samo repetitivno učenje kroz ponavljanje istih kratkih zadataka svaki dan može puno pomoći kod učenja, a može biti i zabavno

Glavna poglavlja ovog rada su inverzija ovisnosti, injekcija ovisnosti i kreiranje aplikacije. Svako novo poglavlje se nadovezuje na prethodno na način da prvo shvatimo teorijski što je to inverzija ovisnosti, a zatim što je injekcija ovisnosti i zašto nam ona pomaže u kreiranju i korištenju inverzije ovisnosti. Kroz primjere u poglavlju injekcije ovisnosti možemo shvatiti razne kutove i primjene ovih koncepata. I kroz praktični primjer ovog rada učimo koristiti ovaj koncept u pravom svijetu. Kroz ove tematike u radu prikazat ćemo sva potrebna znanja koja će ostvariti cilj kreiranja ovakve aplikacije.

Sama riječ inverzija označava preokretanje, izvrtanje, preokret stvari/ideja, itd. Iz toga se vidi da se radi o nekom neuobičajenom stilu programiranja. Taj stil je da preokrenemo ovisnost od većih i manjih dijelova na temeljni dio. Ovaj stil se tek uči kada osoba ima dobre temelje u programiranju, pošto je zahtjevan. O konceptu inverzije ovisnosti opisati ćemo više u našim glavnim poglavljima i kroz primjenu prikazati upotrebu tih koncepata.

2. Metode i tehnike rada

Rad je započet pisanjem dokumentacije, odnosno pisanjem teorijskog dijela rada. Programski kod unutar rada formatiran je pomoću hilite.me web aplikacije. Nakon dovršetka teorijskog dijela rada slijedi praktični dio rada gdje je kreirana aplikacija.

Istraživanje za ovaj rad provedeno je pronalaženjem postojećih radova na ovu temu koristio sam web stranice: Google, Bing, GitHub, ChatGPT, Internet Archive, geeksforgeeks, Stack Overflow, Medium i dr. Pretraživao sam ključne riječi, određene stavke uz pojam, tražio primjere, objašnjenja određenih stavki, knjige o konceptu injekcije ovisnosti i njenoj primjeni itd.

Što se tiče primjera za desktop aplikacije korišten je Visual Studio 2022. Provjera desktop aplikacije napravljena je tako da se program uključi i prikaže rješenje koje se tražilo. Za web aplikaciju korišten je Visual Studio Code i provjeren na način da se aplikacija uključi i prikaže traženo rješenje.

Što se tiče praktične primjene inverzije ovisnosti korišteni su slijedeći programski alati i aplikacije:

- za programiranje korišten je alat Android Studio i jezik Kotlin
- za verzioniranje koda korišten je GitHub
- za opis programskog koda korišten je Microsoft Word
- za prikaz ekrana i kako aplikacija funkcionira korišten je Microsoft Word

Za traženje dijelova koda koristio sam Google, Stack Overflow, you.com, bing, bingAI, ChatGPT, GitHub Copilot. Ovi alati ubrzali su proces programiranja na način da su prikazali kako se nešto radi. Same ideje i koncepti morali su biti predefinirani i proučeni. Pošto ovi alati rade najbolje tek onda kada osoba zna točno ono što želi i točno na koji način to želi. Što to znači? Moje ideje došle su iz iskustva ili proučavanja. Nakon ideja, programski kod došao je iz ChatGPT i GitHub Copilot alata na način da su bili postavljani upiti. Nakon što su ovi alati prikazali kod, taj kod je proučen i vidjelo se što s njime. Dosta puta nije generirao korištenje DI i DIP koncepata koje sam obradio u radu, pa je taj programski kod morao biti uređen. Kada je programski kod bio neuspješno generiran više puta, rješenje sam tražio na razne načine (tipa YouTube, Stack Overflow, Google..). Nakon pronalaska rješenja to rješenje je trebalo prilagoditi tako da radi za koncepte obrađene u radu i da radi ono što se traži u funkcionalnim zahtjevima. Sav programski kod rađen je na ovakav način. Što se tiče samih ideja, one su prvenstveno došle iz iskustva programiranja, ali kada sam zapeo dolazile su na način da sam koristio Google, Stack Overflow, YouTube… i upiti mentoru.

3. Inverzija ovisnosti

Autor Thorben (2024) u svojem članku „*SOLID Design Principles Explained: Dependency Inversion Principle with Code Examples*“ navodi da je inverzija ovisnosti jedan od temelja SOLID principima. Svako slovo označava jedan princip, pa tako i inverzija ovisnosti. Ona je u SOLID zadnji princip, a to je D, tj. eng. Dependency Inversion. Jedna od prvih osoba koja je sugerirala da bi se taj koncept mogao tako zvati je bio Robert C. Martin.

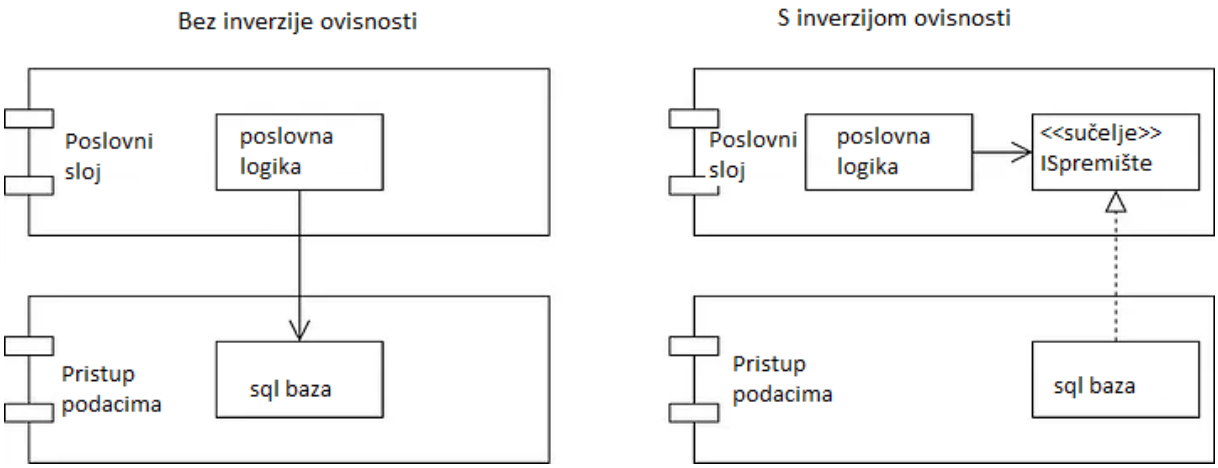
Razlog pojave ovog koncepta je vrlo jednostavan. Prije je programski kod bio dosta neuredan i nije se dao lako nadograditi. Kako bi to razriješili, Robert C. Martin je u raznim knjigama pisao o čistom kodu, čistoj arhitekturi, agilnom razvoju softvera itd. kako bi osvijestio ljude i dao im potrebne informacije o tome da se programski kod ne piše samo za danas, nego i za budućnost (Dependency inversion principle, bez dat.).

3.1. Što je inverzija ovisnosti

Kako bi jednostavnije objasnili ovaj koncept koristit ćemo jednostavan primjer. Zamislite uređaj za kavu. Zamislite da taj uređaj koristite svaki dan, i da radi samo s jednom markom kapsula kave (eng. coffee pod). Jednog dana odete u kupovinu kako bi kupili još takvih kapsula kave i na policama ih više nema. Pitate prodavače, a oni vam kažu da se takve kapsule više ne proizvode. Što sada? Imate uređaj, a ne radi za druge vrste marka kapsula kave? Kako bi izbjegli ovaj izazov, taj uređaj bi trebao raditi za bilo koji brend kapsula kave. Tako da, ukoliko se te kapsule kave izbace iz proizvodnje i prodaje, možemo i dalje koristiti taj uređaj.

Ovaj koncept se naziva inverzija ovisnosti. Naš uređaj za kavu nije ovisan o jednoj marki kapsule kave, nego radi za bilo koju marku tih kapsula kave. Tako vrijedi i za klase, klasa nije vezana za drugu klasu, nego za sučelje (eng. interface). Nije joj bitno kakva je marka kapsule, bitno joj je da je to određena kapsula. To znači da naša klasa nije ovisna o specifičnoj implementaciji sučelja, nego možemo koristiti bilo koju implementaciju našeg sučelja. Na taj način klasa može koristiti bilo koju implementaciju koja zadovoljava sučelje. I tako smo osigurali neovisnost klase o implementaciji sučelja. Isto kao naš uređaj za kavu, koji može koristiti bilo koju marku kapsule koja odgovara standardu uređaja za kavu.

Grafički je to prikazao BuketSenturk (2024), a to izgleda ovako:



Slika 1. Grafički prikaz inverzije ovisnosti (BuketSenturk, 2024)

Bez inverzije ovisnosti, naša logika direktno pristupa bazi, a s inverzijom ovisnosti, ona koristi sučelje „IRepository“ kako bi pristupila bazi. To znači da „Buisness Logic“ nasljeđuje sučelje „IRepository“ i implementira ga. Postoji više izvora koji opisuju dva najbitnija principa inverzije ovisnosti. Svi ti izvori govore o istim stvarima, isto tako Thorben (2024) u svojem članku „*SOLID Design Principles Explained: Dependency Inversion Principle with Code Examples*“ govori: „Robert C. Martin-ova definicija inverzije ovisnosti sastoji se od dva dijela:

1 Moduli više razine ne bi smjeli ovisiti o modulima niže razine. Obje bi trebale ovisiti o apstrakcijama.
2 Apstrakcije ne bi smjele ovisiti o detaljima. Detalji bi trebali ovisiti o apstrakciji“ (Thorben 2024, „*SOLID Design Principles Explained: Dependency Inversion Principle with Code Examples*“)

Ova dva principa govore nam da bilo koje klase, bile to klase više razine ili niže razine, ne komuniciraju međusobno. One komuniciraju preko sučelja. To znači da klase ne znaju kakav je programski kod ili implementacija druge klase i što one rade, samo znaju da će koristiti sučelje. One ne ovise o detaljima, nego o sučelju. Klase niže razine rade se na način da su lako zamjenjive, jer su one uglavnom implementacija sučelja. Tako da, ukoliko imamo više implementacija sučelja, ona nisu ovisna o tome kakva je implementacija klasa više razine, tj. klase više razine više ne ovise o klasama niže razine, nego sada ovise o sučelju.

3.2. Čemu služi i za što se koristi inverzija ovisnosti

Uobičajeno jednostavno programiranje je takvo da unutar glavne klase stavimo implementaciju našeg sučelja. Na taj način nemamo slobodu nad time, tj. kao sa primjerom aparata za kavu, ukoliko će nam u budućnosti trebati nešto drugo, moramo mijenjati puno toga.

Inverzija ovisnosti osigurava da taj izazov ne nastane, pošto je implementacija odvojena od mjesta gdje se koristi. To znači da kada pozivamo metodu, pozivamo ju preko sučelja, a ne preko implementacije. Tek kasnije u kodu „spajamo“ te dvije stavke tako da naša klasa zna koju implementaciju sučelja koristiti. Na taj način osigurali smo fleksibilnost naše aplikacije, lakšu nadogradnju, lakše testiranje, mogućnost ponovnog korištenja komponenti itd.

Što se tiče testiranja koda koji koristi koncept inverzije ovisnosti, puno ga lakše jedinično testirati. Razlog tomu je da kada testiramo module više razine, oni tada ovise o apstrakciji. Pošto ovise o apstrakciji možemo puno lakše koristiti lažne tj. izmišljene implementacije modula nižih razina, pa iz tog razloga testiranje postaje brže i pouzdanije. Pogotovo ukoliko imamo bazu podataka (eng. Database) ili neke druge vanjske povezanosti našeg programa, pa ne moramo testirati preko prave baze podataka nego samo napravimo lažnu implementaciju baze. Time dobijemo i sigurnost nad podacima, pošto znamo da ih nećemo mijenjati, dodavati, brisati itd. Na taj način omogućili smo programerima da se fokusiraju na funkcionalnosti bez da brinu o tome što će vanjski izvori davati i kako će utjecati na aplikaciju i njeno testiranje (Remya Mohanan, 2024; Dependency inversion principle, bez dat.; Beribey, 2019).

Mogućnost ponovnog korištenja komponenti je jedna velika prednost ovog koncepta. Kada više puta koristimo module, oni se lakše ponovno iskoriste, pošto više nisu ovisni o jednoj implementaciji. Isto tako module niže razine, tj. implementacije možemo lako koristiti na drugim mjestima. I lakše se daju ponovno koristiti u raznim drugim kontekstima u programu. Tako sprječavamo dupliciranje programskog koda, pošto nemamo potrebu za kreiranjem novih klasa koje rade gotovo identične stvari, a razlikuju se u jednoj liniji koda (Remya Mohanan, 2024; Dependency inversion principle, bez dat.; Beribey, 2019).

Održavanje koda je još jedan veliki plus ovog koncepta. Puno je lakše nešto mijenjati i nadodati pošto nije sve čvrsto povezano jedno s drugim. Postoji taj odmak od komponenti, a i time dobivamo puno lakše razumijevanje programskog koda. Ovo je vrlo bitno za aplikacije koje će se nadograđivati u budućnosti, pošto su sada jasnije i lakše za modifikaciju. U bilo kojem trenu možemo promijeniti korištenje nekih od implementacija sučelja koje smo isprogramirali i to nam daje veliku slobodu nad programskim kodom (Remya Mohanan, 2024; Dependency inversion principle, bez dat.; Beribey, 2019).

Inverzija ovisnosti nam daje prilagodbu za različite zahtjeve. Određena implementacija sučelja ne ovisi o ostatku programa. To znači da, ako promijenimo našu implementaciju nekog sučelja u neku drugu implementaciju i ne promijenimo ostatak koda, naš kod bi trebao i dalje raditi kako treba. Jedino što će biti drugačije je sama implementacija toga sučelja. Isto tako nam to daje slobodu, ukoliko naš kod postane biblioteka koju drugi korisnici mogu koristiti. Daje nam slobodu da promijene implementaciju sučelja na onaj način koji žele. Realni primjer bi bio ukoliko naš klijent želi na neki drugi način ispis npr. da smo imali ispis na ekran, a on

želi u pdf. Mi vrlo lako možemo zamijeniti taj dio. Pošto samo kod korištenja i poziva u glavnom programu ne stvorimo ispis na ekran nego ispis u pdf i dodijelimo ga programu (Remya Mohanan, 2024; Dependency inversion principle, bez dat.; Beribey, 2019).

Thorben (2024) u svojem članku „*SOLID Design Principles Explained: Dependency Inversion Principle with Code Examples*“ govori da zapravo inverzija ovisnosti koristi dva principa iz SOLID principa. Prvi princip je otvorenost-zatvorenost (eng. Open-Closed principle). Otvorenost-zatvorenost govori o tome da komponente trebaju biti u mogućnosti nadograditi, a ne mijenjati. Ne mijenjamo metode sučelja nego stvaramo implementaciju sučelja. Drugi princip je Liskovo načelo zamjene (eng. Liskov Subtitution Principle) koji govori o tome da moramo biti u mogućnosti promijeniti implementacije sučelja.

3.3. Prednosti i nedostaci inverzije ovisnosti

3.3.1. Prednosti inverzije ovisnosti

Većina prednosti navedena je i opisana u poglavlju „3.2. Čemu služi i za što se koristi inverzija ovisnosti“, a ima ih još par koje su bitne za naglasiti. Navedeno je lako jedinično testiranje, mogućnost ponovnog korištenja komponenti, lakša održivost koda u budućnosti i laka promjena i prilagodba, a ono što je još bitno za spomenuti je mogućnost kasnijeg kreiranja implementacije. Mi direktno možemo napraviti program koji poziva sučelje, nakon što smo napravili program i znamo kako bi trebao raditi, tu implementaciju sučelja i metoda dodamo nakon. Na taj način imamo više informacija o tome što program radi i što bi trebao koristiti. Imamo širu sliku i možemo programirati prvo složenije pa jednostavnije komponente (Remya Mohanan, 2024; Thorben, 2024, „Design Patterns Explained – Dependency Injection with Code Examples“; Milica Dancuk, 2023).

Iduća prednost je mogućnost lakšeg paralelnog programiranja softvera. Ovo je jedna velika prednost ukoliko radimo u timovima. Svaki tim se može dogovoriti što će raditi. Kada se jednom definiraju sučelja, rad može krenuti. Pošto sve komponente ovise o sučelju, nije potrebno znati kakva će biti implementacija određenog sučelja. Na taj način kada timovi završe svoj dio, mogu spojiti kod (eng. merge code). Nakon spajanja koda, mogu odlučiti koju implementaciju sučelja koristiti. Kao što smo imali na primjeru za aparata za kavu, možemo napraviti kapsule i aparat, i nije bitno kakva je kava unutar kapsula, bitna je kapsula, tj. okvir. I bitan je aparat koji koristi određene kapsule, tj. dogovoreni okvir (Remya Mohanan, 2024; Thorben, 2024, „Design Patterns Explained – Dependency Injection with Code Examples“; Milica Dancuk, 2023).

Prednost lake skalabilnosti koda nam daje slobodu da dodajemo nove komponente u naš kod, bez briga da li ćemo ga potrgati. Svaka komponenta je za sebe i ne ovisi toliko o

drugim komponentama kao u standardnom programiranju. To znači da ukoliko dobijemo zahtjev da dodamo novu funkcionalnost unutar aplikacije, tu novu funkcionalnost će biti puno lakše dodati i implementirati nego uobičajenim tehnikama programiranja (Remya Mohanan, 2024; Thorben, 2024, „Design Patterns Explained – Dependency Injection with Code Examples“; Milica Dancuk, 2023).

Sada, kada znamo čemu služi i koje su prednosti ovog koncepta, polako možemo vidjeti zašto ćemo baš njega koristiti u praktičnom dijelu ovog rada. Fokus ćemo staviti na lakoću promjene komponenti. Napravit ćemo glavno sučelje za zadatak i od njega granati dalje posebne implementacije svakog zadatka. Na taj način ukoliko želimo u budućnosti dodati novi zadatak ili promijeniti postojeći, to ćemo s lakoćom moći izvesti.

Pošto svaka stvar ima svoje prednosti i nedostatke, tako i inverzija ovisnosti. Pa je jednako bitno znati i nedostatke ovog koncepta.

3.3.2. Nedostaci inverzije ovisnosti

Sami nedostaci inverzije ovisnosti su više u stranu težine i iskustva u programiranju. Povećanje kompleksnosti koda nam govori više stvari. Sam programski kod ukoliko imamo jednostavan program, postaje kompliciraniji. Ukoliko želimo jednostavan ispis na ekran to postaje program s više datoteka, klasa, sučelja. To za sobom povlači potrebu za više vremena kako bi shvatili kako programski kod funkcionira. Potreba dokumentacije raste, kako bi se smanjilo vrijeme potrebno za shvaćanje programskog koda, ukoliko nova osoba dolazi na projekt. Isto tako dokumentacija će povećati razumijevanje što se radi u programu i kako to funkcionira (Remya Mohanan, 2024; Thorben, 2024, „Design Patterns Explained – Dependency Injection with Code Examples“; Milica Dancuk, 2023).

Iduća stavka je to što stvara izazov kod uklanjanja pogrešaka u kodu. Teže je procijeniti gdje je točno greška, pošto programski kod ima puno raznih poziva. Može nam se učiniti da je problem u klasi koju gledamo, a zapravo je u nekoj drugoj. Pogreške se skrivaju od nas i teže ih uočavamo (Remya Mohanan, 2024; Thorben, 2024, „Design Patterns Explained – Dependency Injection with Code Examples“; Milica Dancuk, 2023).

Sam koncept je lagan, ali je potrebno pažnje kako bi se ušlo i shvatilo što on točno radi. Pogotovo kod projekata, uvelike otežava razumijevanje projekta. Bilo to za novo dodane programere koji se prvi puta susreću s ovim konceptom ili ljude koji su već upoznati i imaju neko iskustvo na tom projektu. Isto povećava vrijeme potrebno za shvaćanje programskog koda. Bitno je imati dobru dokumentaciju koja će ovaj izazov značajno smanjiti i ubrzati rad (Remya Mohanan, 2024; Thorben, 2024, „Design Patterns Explained – Dependency Injection with Code Examples“; Milica Dancuk, 2023).

Trošak rada povećava se u usporedbi s potrebnim vremenom koje uložimo u shvaćanje tog koncepta, pogotovo u programskom kodu. Time povećavamo cijeli trošak razvoja i održavanja softvera. Ovo je pogotovo vidljivo kada aktivno nadograđujemo i mijenjamo naš programski kod (Remya Mohanan, 2024; Thorben, 2024, „Design Patterns Explained – Dependency Injection with Code Examples“; Milica Dancuk, 2023).

3.4. Zaključak o inverziji ovisnosti

Iz ovog poglavlja dobili smo sliku o tome što je inverzija ovisnosti. Kroz primjer aparata za kavu nastojali smo bolje razjasniti ovaj koncept inverzije ovisnosti. Navedena su dva temeljna dijela od kojih se inverzija ovisnosti sastoji. Objašnjeni su razlozi korištenja ovog koncepta, navedene su prednosti i nedostaci te su pojašnjeni.

Kako bi mogli koristiti koncept inverzije ovisnosti u programiranju stvorena je injekcija ovisnosti. U poglavlju „4. Injekcija ovisnosti“ razraditi će se korištenje inverzije ovisnosti preko injekcije ovisnosti kroz teoriju i razne primjere i primjene u kodu.

4. Injekcija ovisnosti

U ovom poglavlju razjašnjavam koncept injekcije ovisnosti kroz primjere, isto tako dajem neke primjere implementacije za desktop aplikacije u Visual Studio 2022 koristeći C#, za Web aplikacije koristeći JavaScript i za Android Studio koristeći Kotlin. Opisujem razliku unutar programskih jezika i njihove primjene. Dotičem se osnovnih koncepata od kojih se injekcija ovisnosti sastoji kako bi dobili širu sliku tog koncepta. Za što se koristi taj koncept? Koje su najčešće korištene tehnike implementacija inverzije ovisnosti? Objašnjena je svaka od najčešće korištenih tehnika, dano je par primjera, dane su prednosti i nedostaci svake. Opisana je uporaba injekcije ovisnosti kod android aplikacija i njenog dizajna preko automatske i ručne injekcije i dan je primjer kako bi nam ti koncepti bili lakši za shvatiti. I za kraj uspoređena je inverzija ovisnosti s injekcijom ovisnosti i objašnjeno je da to nisu isti koncepti i opisana je njihova razlika.

4.1. Koncept injekcije ovisnosti

Mark Seemann (2011, str. 3, 4) u svojoj knjizi „*Dependency Injection in .NET*“ opisuje koncept injekcije ovisnosti kao teškim konceptom za korištenje. Zašto teškim? Razlog je jednostavan, ukoliko pogriješimo, morat ćemo početi ispočetka. Znači li to da je težak? Ne, zapravo je jednostavan i jednostavno se može ući u njega.

To nam govori da koncept zahtjeva puno vremena, jer ukoliko pogriješimo moramo krenuti ispočetka. Ljudi ga izbjegavaju i baš iz tog razloga malo ga ljudi opće i pokuša koristiti. Pošto sam koncept može zaplašiti ljude da ga uopće i probaju. No i to ima svoje prednosti. Makar kada pogriješimo, i moramo sve započeti ispočetka, na toj grešci ćemo naučiti i bit će nam lakše kada pokušamo ponovo. Ali s druge strane moramo biti svjesni vremena, pažnje i volje koji nam trebaju kako bi uspješno izvršili taj koncept.

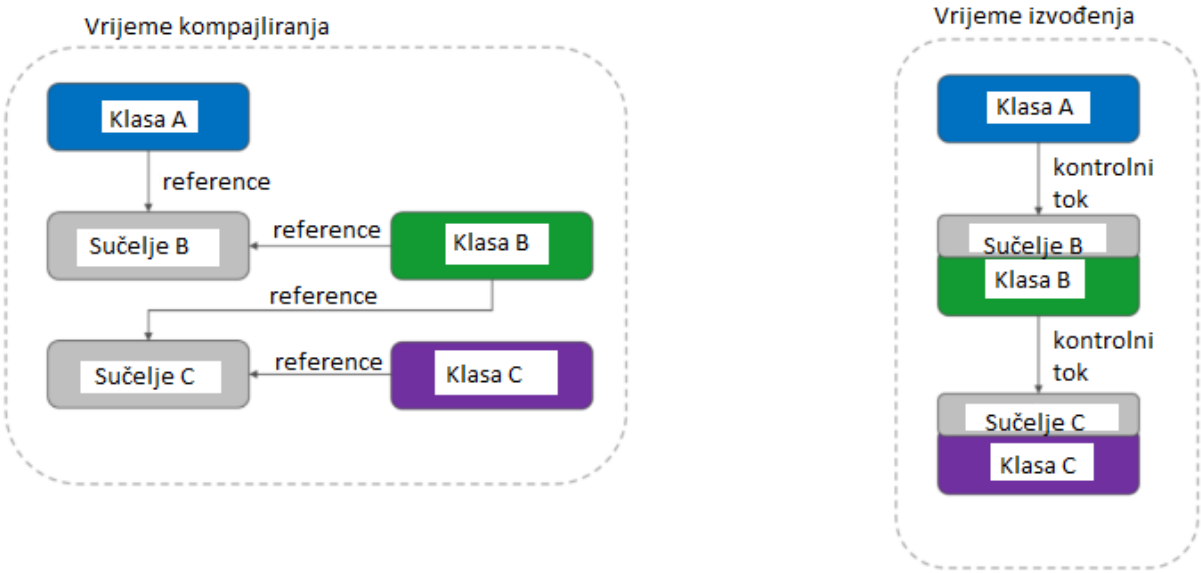
Mark Seemann (2011, str. 4) isto govori kako najjednostavnije opisati injekciju ovisnosti? Na stack overflow-u pod naslovom „How to explain dependency injection to a 5-year-old?“ odgovor od John-a Munsch-a govori: „Kad odete sami uzeti stvari iz hladnjaka, možete izazvati probleme. Možda ostavite otvorena vrata, možda uzmete nešto što vaši roditelji nisu htjeli da uzmete. Možda potražite čega ni nema u hladnjaku ili čemu je rok uporabe istekao. Ono što bi zapravo trebali je izraziti potrebu „trebam nekakvo piće uz ručak“ i onda ćemo osigurati da imate nešto za piti kada sjednete jesti“ (Brad Larson, 2009).

Ova analogija jednostavno opisuje injekciju ovisnosti i kako se koristi. Petogodišnjak predstavlja sve klase koje surađuju, roditelji predstavljaju sučelje. To znači da petogodišnjak ovisi o roditeljima, tj. da klase ovise o sučelju.

Sada znamo da klase ovise o sučelju i da trebamo znati što želimo, ali nastaje pitanje, da li je to dovoljno? Što točno je koncept injekcije ovisnosti? Objasnimo to na nekom primjeru. Zamislite da imamo kostur igračke u obliku kuće. Pod time mislim samo na okvir, bez zidova, prozora, vrata, krova… čisti okvir, okvir u obliku kvadra, taj okvir je u našem slučaju sučelje. On predstavlja neku osnovu kako će kuća izgledati i kako će biti sagrađena. Na taj okvir kvadra kod svake plohe možemo staviti razne oblike i dizajne zidova, mogu biti plavi, kockasti, trokutasti, malo izbočeni prema van da dobe na dizajnu, prema unutra… isto tako krov kuće može biti napravljen na razne načine, mogu se četiri strane spajati u jednu točku, mogu se dvije, a može biti i ravan. Sve to nam predstavlja našu implementaciju sučelja. Mi na naše sučelje tj. okvir kvadra imamo slobodu da ispunimo te plohe na način koji želimo. Sada slijedi jedan veliki „ali"… Ali moramo se držati pravila našeg okvira, to znači da ako želimo piramidu, ne možemo pretvoriti naš okvir kvadra u piramidu, zbog toga jer to više nije kvadar nego piramida, to onda više nije taj okvir, to je postao neki drugi okvir, to više nije naše sučelje. Prazne plohe u našem slučaju mogle još predstavljati ulazne i izlazne elemente. Ukoliko naše sučelje ima neku metodu, ona može primati argumente, i vraćati ih po želji. Ali opet moramo ostati u tim granicama. Ukoliko naša metoda u sučelju prima brojke, ne možemo joj ubaciti varijablu tipa bool ili string. Zaključak ovoga je da imamo okvir tj. sučelje, i držimo se toga okvira, ali da ga implementiramo onako kako nama treba u nekom programskom kodu i kontekstu.

Sada kada smo dobili malo bolju sliku sučelja, pitanje je, što dalje, je li je tu kraj, imamo sučelje, imamo implementaciju sučelja i sada napravimo program koji koristi to sučelje. Tako je, sučelje, ne implementaciju, naš program je neovisan o implementaciji. Neovisna o implementaciji znači da imamo naš okvir kvadra kao okvir igračke kuće, pitanje glasi, treba li to biti isključivo kuća? Što ako mi u budućnosti želimo igračku kontejnera, želimo kutiju za cipele, ormar ili ciglu? To su sve različiti kvadri i jedina bitna stvar kod njih je da koriste naš originalan okvir, tj. metode sučelja i da se drže njegovog pravila, a to je da su kvadar. To znači da program ne ovisi o implementaciji sučelja, nego o sučelju. Sada što se tiče programa, naš program samo poziva metode od sučelja dajući mu traženi argument, ukoliko ga ta metoda ima. Na taj način ta implementacija i korištenje je neovisno jedno o drugom, to znači da implementacija može biti napravljena u jednoj datoteci i radi nešto, dok implementacija sučelja u drugoj i radi nešto. Odvojeni su, a spajamo ih preko sučelja, pošto oba koriste sučelje. Jedna klasa ga implementira, a druga ga klasa poziva. Napravili smo sučelje, implementirali ga, i koristimo ga u nekoj klasi. U glavnom programu možemo kreirati varijablu sučelja i instancu klase i povezati te dvije stvari na razne načine. Nakon povezivanja, možemo pozvati metodu te klase gdje se sučelje poziva, i ono će indirektno zvati i koristiti tu implementaciju, pošto smo kreirali varijablu te implementacije i dodijelili ju joj.

Grafički prikaz ovog koncepta izgleda ovako:



Slika 2. Grafički prikaz injekcije ovisnosti Mohammad Ramezani (2020)

„Slika 2. Grafički prikaz injekcije ovisnosti Mohammad Ramezani (2020)“ prikazuje dijagram odnosa između klasa i sučelja u inverziji ovisnosti. Stavka „Vrijeme kompajliranja“ to najbolje prikazuje. Klasa A koristi Sučelje B. Klasa B implementira Sučelje B. Klasa B isto tako implementira sučelje C. I na kraju Klasa C implementira sučelje C.

Sada ćemo pogledati primjer primjene za desktop aplikacije kako bi nam ovaj dio bio potpuno jasan.

4.2. Primjena za desktop aplikacije

Kako bi demonstrirali inverziju ovisnosti kod desktop aplikacija upotrjebljavamo koncept injekcije ovisnosti. Za programiranje koristimo Visual Studio 2022. Projekt je Console App (.NET Framework) i jezik C#.

Prvi primjer koristi konstruktor kao prijenosnik injekcije ovisnosti. Mark Seemann (2011, str. 13, 14, 15) je na jednostavan način prikazao kako se koristi injekcija ovisnosti preko konstruktora. Drugi primjer koristi svojstvo kao prijenosnik. I za kraj treći primjer koristi metodu kao prijenosnika. Drugi i treći primjeri su uređivanje prvog primjera i originalan su rad.

4.2.1. Prvi primjer – C# – injekcija pomoću konstruktora

Prvi korak je stvaranje sučelja `IMessageWriter` i unutar njega metodu imena `Write` koje će primati poruka tipa `string` unutar sebe. To izgleda ovako:

```
1 public interface IMessageWriter
2 {
3     void Write(string message);
4 }
```

Programski Kod 1 – prikaz klase IMessageWriter – injekcija preko konstruktora – C# (Mark Seemann, 2011, str. 15.)

Nakon toga kreiramo implementaciju sučelja `IMessageWriter`. Pošto ona ima samo jednu metodu, samo tu metodu implementiramo. Naziv nove klase je **`ConsoleMessageWriter`** i obavezno moramo naslijediti toj klasi sučelje `IMessageWriter`. Metoda **`Write`** u sebi ima `Console.WriteLine(message)`, što nam omogućuje ispisivanje dobivene poruke na zaslon. To izgleda ovako:

```
1 public class ConsoleMessageWriter : IMessageWriter
2 {
3     public void Write(string message)
4         {
5             Console.WriteLine(message);
6         }
7 }
```

Programski Kod 2 – prikaz klase ConsoleMessageWriter – injekcija preko konstruktora – C# (Mark Seemann, 2011, str. 15)

Sada imamo i sučelje i implementaciju. Kako bi demonstrirali injekciju ovisnosti, napravimo još jednu klasu zvanu **`Salutation`**. Ona će koristiti sučelje i neće biti direktno povezana ni na koji način sa implementacijom **`Write`** u klasi **`ConsoleMessageWriter`**. To znači, ukoliko promijenimo implementaciju metode **`Write`** unutar **`ConsoleMessageWriter`** klasa **`Salutation`**, ona će i dalje raditi što radi, ako dodamo drugu implementaciju sučelja `IMessageWriter` tipa da zapisuje u .txt dokument, klasa **`Salutation`** će i dalje raditi to što radi. Jednostavno rečeno, to su kao dvije različite osobe koje ne znaju jedna za drugu. Što ih onda spaja? Kako komuniciraju jedna sa drugom? Kako je korištena injekcija ovisnosti? Na sva ova pitanja dobit ćemo odgovor vrlo brzo.

Pa krenimo sa implementacijom klase **`Salutation`**. Prva stvar, kreiramo deklaraciju `IMessageWriter` -a zvanom **`Write`**. Nakon toga kreiramo konstruktor. Naš konstruktor primat će parametar. Znate li možda koji? To nije **`ConsoleMessageWriter`**, pošto smo bili rekli da klase **`ConsoleMessageWriter`** i **`Salutation`** ne komuniciraju međusobno i ne znaju jedna za drugu. Sada dolazi do ubacivanja injekcije ovisnosti u konstruktor, a to je da je konstruktor ovisan o `IMessageWriter` sučelju. Pa je parametar koji prima `IMessageWriter` **`Write`**. Unutar konstruktora kreiramo **`if`** uvjet i provjeravamo da **`Write`** nije prazan tj. dali je **`null`**. Ukoliko je **`null`**, vraćamo `ArgumentNullException`. I nakon toga postavljamo `Writer = writer`. Sada

nam jedino preostaje koristiti metodu `Write` unutar klase `Salutation`. Kreiramo metodu `Exclaim` koja poziva unutar sebe metodu `Write` i predaje joj string tipa `"Hello DI!"`. Na taj način osigurano je korištenje `Write` metode i ovisnost klase `Salutation` o sučelju `IMessageWriter`. Isto tako na taj način je kreirano korištenje injekcije ovisnosti. Tako da klasa `Salutation` i klasa `ConsoleMessageWriter` ovise o `IMessageWriter` -u tj. o sučelju, a ne ovise jedna o drugoj. Implementacija `Salutation` klase izgleda ovako:

```
1  public class Salutation
2      {
3          private readonly IMessageWriter Writer;
4          public Salutation(IMessageWriter writer)
5          {
6              if (writer == null)
7                  throw new ArgumentNullException("writer");
8              Writer = writer;
9          }
10         public void Exclaim()
11         {
12             Writer.Write("Hello DI!");
13         }
14     }
```

Programski Kod 3 – prikaz klase Salutation – injekcija preko konstruktora – C# (Mark Seemann, 2011, str. 14)

Jedina preostala stvar je korištenje naše metode `Exclaim`. Unutar našeg glavnog programa `Main` deklariramo varijablu `writer` koja je tipa `IMessageWriter`. Ona nam je jednaka našoj implementaciji, a u ovom slučaju je to `ConsoleMessageWriter()`. Nakon toga kreiramo instancu `Salutation` i prosljeđujemo njenom konstruktoru `writer`. Na taj način smo „spojili“ klase `ConsoleMessageWriter` i `Salutation` indirektno, tj. rekli smo klasi `Salutation` da koristi implementaciju od `ConsoleMessageWriter`. I naposlijetku pozovemo našu metodu `Exclaim`. Implementacija `Main` programa izgleda ovako:

```
1  internal class Program
2      {
3          private static void Main(string[] args)
4          {
5              IMessageWriter writer = new ConsoleMessageWriter();
6              var salutation = new Salutation(writer);
7              salutation.Exclaim();
8
9              Console.ReadLine();
10         }
11     }
```

Programski Kod 4 – prikaz glavnog programa – injekcija preko konstruktora – C# (Mark Seemann, 2011, str. 13)

Napomena: Kako bi vidjeli ispis na ekran, dodajmo još jednu liniju a to je `Console.ReadLine()`. Ova linija će čekati da korisnik pritisne tipku enter, jer bez nje program se odmah nakon pokretanja i zatvara, pa je teško vidjeti da li program radi ili ne.

4.2.2. Drugi primjer – C# – injekcija pomoću svojstva

U drugom primjeru koristimo svojstvo kao pomoćni element u injekciji ovisnosti. Klase `IMessageWriter` i **`ConsoleMessageWriter`** ostaju iste. Promjene se dešavaju unutar klase **`Salutation`** i glavnom programu. Prvo kasa **`Salutation`** više neće sadržavati konstruktor, nego će naš `IMessageWriter Writer` biti taj kojem ćemo direktno pristupati. Pored njega nadodajemo **`get`** i **`set`**. Metoda **`Exclaim`** je malo drugačija. Iz koda „Programski Kod 5 – prikaz klase **`Salutation`** – injekcija preko svojstva – C#“ linija broj 7 sa **`if`**-om stavljamo unutar **`Exclaim`** metode prije pozivanja `Writer.Write`. To izgleda ovako:

```
 1 public class Salutation
 2     {
 3         public IMessageWriter Writer { get; set; }
 4
 5         public void Exclaim()
 6         {
 7             if (Writer == null) throw new
 8             ArgumentNullException("writer");
 9             Writer.Write("Hello DI with property Injection!");
10         }
11     }
```

Programski Kod 5 – prikaz klase Salutation – injekcija preko svojstva – C#

Na taj način osigurali smo da se korištenje inverzije ovisnosti dešava preko svojstva, a ne konstruktora.

I na kraju naš glavni program umjesto da kod kreiranja instance **`Salutation`** i prosljeđivanje parametra u konstruktor, to radimo prizivanjem parametra `Writer` iz klase **`Salutation`**. I to izgleda ovako:

```
 1 internal class Program
 2     {
 3         private static void Main(string[] args)
 4         {
 5             IMessageWriter writer = new ConsoleMessageWriter();
 6             var salutation = new Salutation();
 7             salutation.Writer = writer;
 8             salutation.Exclaim();
 9
10             Console.ReadLine();
11         }
12     }
```

Programski Kod 6 – prikaz glavnog programa – injekcija preko svojstva – C#

Razlika prvog i drugog primjera je u tome što smo promijenili klasu `Salutation`, maknuli smo konstruktor i dodali `Writer` s `get` i `set` metodom. Na taj način osigurali smo da koristimo svojstvo, a ne konstruktor. U glavnom programu pozivamo samo to svojstvo i dodijelili mu novi `ConsoleMessageWriter`.

4.2.3. Treći primjer – C# – injekcija pomoću metode

I za kraj treći primjer injekcije ovisnosti, koristimo metodu umjesto konstruktora ili svojstva. Programski kod klasa `IMessageWriter` i `ConsoleMessageWriter` je isti kao i prije. Kod klase `Salutation` izbacujemo van svojstvo `IMessageWriter Writer` i ostaje nam samo `Exclaim` metoda. Inverziju ovisnosti možemo urediti na vrlo jednostavan način, dodamo `IMessageWriter writer` u njen parametar. To izgleda ovako:

```
1 public class Salutation
2     {
3         public void Exclaim(IMessageWriter writer)
4         {
5             if (writer == null) throw new
6             ArgumentNullException("writer");
7             writer.Write("Hello DI!");
8         }
9     }
```

Programski Kod 7 – prikaz klase Salutation – injekcija preko metode – C#

I za kraj naš glavni program je vrlo sličan prva dva primjera. Deklariramo varijablu `writer` koja je tipa `IMessageWriter`. Nakon toga kreiramo instancu `Salutation`. Pozivamo metodu `Exclaim` i prosljeđujemo joj `writer` koji je tipa `IMessageWriter`. Na taj način smo osigurali inverziju ovisnosti. Kod glavnog programa izgleda ovako:

```
 1 internal class Program
 2     {
 3         private static void Main(string[] args)
 4         {
 5             IMessageWriter writer = new ConsoleMessageWriter();
 6             var salutation = new Salutation();
 7             salutation.Exclaim(writer);
 8
 9             Console.ReadLine();
10         }
11     }
```

Programski Kod 8 – prikaz glavnog programa – injekcija preko metode – C#

Za razliku od prvog i drugog primjera, ovaj primjer koristi metodu. Isto smo promijenili klasu `Salutation`, ali ovoga puta unutar naše metode `Exclaim` dodali `Writer` tipa sučelja `IMessageWriter`. Na taj način osiguran je rad preko metode. Što se tiče glavnog programa

proslijeđen je novo kreirani `Writer` našoj metodi `Exclaim` tako da zna koju će implementaciju sučelja koristiti.

4.3. Na čemu se bazira injekcija ovisnosti tj. koje principe koristi

Mark Seemann (2011, str 8, 9, 10, 11, 12, 13) opisuje ovo preko utičnice na zidu. Ovaj koncept opisan je preko USB-C utora. Zamislite scenu, imate mobitel i tablet i želite ih povezati na laptop. Kako bi izgledalo da svaki uređaj ima svoje utore? Da mobitel ima poseban utor, da tablet ima poseban utor i k tome da laptop ima svoje posebne utore. Kako ih povezati žično? Trebali bi koristiti ili nastavke koji pretvaraju jedan port u drugi ili imati svaki mogući kabel s posebnim portom, ali to je neefikasno. Zašto? Zamislite da još nadodate mobitele roditelja, koji naravno imaju svoje posebne portove. Morali bi za svaki port imati nastavak koji bi to pretvarao u onaj koji laptop podržava ili imati svaki mogući kabel sa svim portovima. Kako bi ovo izbjegli koristimo standardizirani port, USB-C. Svaki noviji mobitel ima USB-C port, i svaki laptop ima barem 1 utor za USB-C port. Na taj način je osigurano da možemo uzeti minimalno samo 1 kabel, a ne hrpu raznih nastavka i kablova za svaki uređaj. Kakve to ima veze s bazom injekcije ovisnosti? Izvrsno pitanje. Baza se zove „Liskovo načelo zamjene“ i ono nam govori da možemo rješavati zahtjeve koji će se pojaviti u budućnosti makar ih danas ne možemo predvidjeti. Kakve to ima veze sa portovima? Vrlo jednostavno, nakon standardiziranja portova, osigurano je da bilo kakav uređaj, ne samo mobiteli, tableti nego i miševi, tipkovnice, kamere, punjenje uređaja, itd. možemo spojiti u USB-C i raditi će, a da ne moramo brinuti koje nastavke koristiti, koji tipovi utora postoje, koliko različitih kablova moramo nositi sa sobom. U ovom primjeru nam USB-C glumi sučelje, a ostali uređaji (mobiteli, laptopi, tipkovnice, itd.) klase koje koriste to sučelje. I na taj način, bez obzira ukoliko promijenimo uređaj ili nadodamo novi (klasu), s lakoćom ćemo ju moći koristiti. Na ovaj način smo osigurali, ukoliko je nešto spojeno radit će kako treba. Nadalje korištenjem konekcije tipkovnica – laptop, osiguravamo da tipkovnica samo radi jednu stvar, a to je da bude tipkovnica. I ukoliko se nešto desi tipkovnici, laptop će i dalje raditi, ukoliko se nešto desi laptopu, tipkovnica će i dalje raditi na drugom laptopu. To se naziva „Jedna odgovornost“ (eng. Single Responsibility). Ona nam osigurava da jedna stvar, u ovom primjeru tipkovnica, radi samo jednu stvar, a to je da bude tipkovnica koja nam daje unos znakova u računalo. I zadnja stavka je što ako imamo samo jedan USB-C port na laptopu a više vanjskih uređaja? Napravimo nastavak koji daje dodatne USB-C portove koji se uključi u USB-C port. Taj koncept naziva se „Dekorater“ (eng. Decorator). On koristi i dalje USB-C port, ali je nadograđen na način da se više uređaja može spojiti. Što to točno znači? Nastavak (klasa) i dalje koristi sučelje od USB-C porta i nadograđuje

ga s nečim (u ovom slučaju više portova) i daje nam van ono što nam je potrebno, a to je u ovom slučaju USB-C port. Što smo time dobili? Ulazni USB-C port s dodatnim specifikacijama koji ima izlazni USB-C utor. Zadovoljava li ovaj nastavak naše navedene stavke? Da, ima jednu funkcionalnost, ostaje istinit za USB-C sučelje i rješava problem sada i za ubuduće, a to je da imamo više USB-C utora. Sve navedeno stvara svoje nadogradnje na sučelje i ostaje istinito sučelju kakvo je i bilo. Taj koncept se zove „Kompozit“ (eng. Composite). Kao zadnji primjer, stavit ćemo USB-Micro u priču. Što ako imamo stariji uređaj koji ima na sebi USB-Micro i trebamo ga spojiti na laptop? Isto kao i prije, koristimo sučelje USB-C, kreiramo nastavak tj. pretvornik USB-Micro na USB-C. Dali ovo odgovara našim bazama inverzije ovisnosti? Da, jer su sve stavke zadovoljene.

4.4. Čemu služi, za što i kako se koristi injekcija ovisnosti

Kako bi bolje savladali injekciju ovisnosti opisujemo čemu ona služi, za što i kako se koristi. Pod čemu služi navodimo i objašnjavamo prednosti. A kako i za što služi objašnjavamo ručno korištenje i automatizirano korištenje ovog koncepta. Objašnjavamo svaku stavku, dajemo izazove s kojima se suočavaju i kako ih riješiti. Isto tako dajemo jednostavan primjer kako bi vidjeli korištenje ovih koncepata u Android Studiu s jezikom Kotlin.

4.4.1. Čemu služi i za što se koristi injekcija ovisnosti

Pod čemu služi zapravo spadaju njene prednosti. Pošto će ljudi koristiti nešto znajući prvo za što je to dobro i gdje se može primijeniti. Glavna stvar zašto koristimo injekciju ovisnosti je kako bi mogli koristiti inverziju ovisnosti. Injekcija ovisnosti zapravo ima prednosti kao i inverzija ovisnosti. Te prednosti su nabrojane i objašnjene u lekcijama „3.2. Čemu služi i za što se koristi inverzija ovisnosti“ i „3.3. Prednosti i nedostaci inverzije ovisnosti“. Ukratko u njima je spomenuto kako je aplikacija fleksibilnija, lakše ju je nadograditi, lakše jedinično testirati i lakše održavati. Isto tako bolja je iskoristivost koda pošto ga možemo ponovo koristiti. Funkcionalnosti klasa su odvojene jedna od druge i svaka obavlja svoj posao. Kada imamo timove lakše je raditi paralelno programiranje i nije nam potrebna sva implementacija svega nego samo sučelja, pa možemo kasnije implementirati neke stvari. Te prednosti nam daju do znanja za što bi sve mogli koristiti injekciju ovisnosti. Uzmimo lakšu nadogradnju. Ona u sebi ima urednost u kodu, kada je kod uredniji i čitljiviji lakše je nadogradiv. Isto tako to nam daje do znanja da bi ovaj koncept mogli koristiti kod većih projekata gdje se traži stalna nadogradnja softvera. Ako firma koja je zatražila softver poslije zatraži još neke funkcionalnosti, ovaj način će učiniti taj manevar puno lakšim. Svaka klasa radi svoju stvar, ne smeta jedna drugoj. Ukoliko imamo novi tim ljudi koji je došao u firmu i radi na nekom programu ovaj koncept će reći, hej,

ne moraš imati ovaj dio koda, to trenutno nije bitno. Samo se ti pobrini da ono što radiš napraviš kako treba. I na taj način se indirektno povećava fokus programera i njihova učinkovitost. To za sobom povlači da su programeri sretniji i rade brže, a ako je zaposlenik sretniji i voli to što radi, taj posao će biti znatno bolje napravljen nego netko tko nije. Sa testiranjem svaki programski kod profesionalnijih softvera mora biti testiran, mora proći sve testove koji su bili određeni bez grešaka. Ovaj koncept znatno olakšava to testiranje, pogotovo jedinično testiranje. Ukoliko imamo na umu napraviti program kako bi mogli vježbati jedinično testiranje, ovaj koncept bi bio savršen za tako nešto. Kao što smo mogli primijetiti, ovog koncepta prikladno je koristiti u velikim, složenim softverima, pogotovo gdje je bitna modularnost i testiranje (Remya Mohanan, 2024; Thorben, 2024, „Design Patterns Explained – Dependency Injection with Code Examples“; Milica Dancuk, 2023; „Dependency Injection“, bez. dat; tarunsh648, 2024).

4.4.2. Kako i zašto se koristi injekcija ovisnosti

Sada kada imamo ideje o tome čemu služi i za što bi se koristila injekcija ovisnosti, uzmimo u obzir ručno i automatsko korištenje ovog koncepta. Zašto bi koristili ručni, a zašto automatizirani koncept injekcije ovisnosti? Originalna android studio dokumentacija za programere govori o automatizaciji injekcije ovisnosti. Kaže: „Ručna injekcija ovisnosti također predstavlja nekoliko problema:

- Za velike aplikacije, uzimanje svih ovisnosti i njihovo ispravno povezivanje može zahtijevati veliku količinu standardnog koda. U višeslojnoj arhitekturi, kako bismo stvorili objekt za gornji sloj, moramo osigurati sve ovisnosti slojeva ispod njega. Kao konkretan primjer, za izradu pravog automobila možda će vam trebati motor, prijenosnik, šasija.. i drugi dijelovi; a motor zauzvrat treba cilindre i svjećice.
- Kada niste u mogućnosti konstruirati ovisnosti prije nego što ih proslijedite — na primjer kada koristite lijene inicijalizacije (eng. lazy initializations) ili obuhvaćene objekte tokovima vaše aplikacije — trebate napisati i održavati prilagođeni spremnik (ili graf ovisnosti) koji upravlja životnim vijekom vaše ovisnosti u memoriji.

Postoje biblioteke koje rješavaju ovaj problem na način da automatiziraju proces stvaranja i pružanja ovisnosti. One spadaju u dvije kategorije:

- Rješenja temeljena na refleksiji koja povezuju ovisnosti tijekom izvođenja (eng. runtime).
- Statička rješenja koja generiraju programski kod kako bi povezale ovisnosti tijekom vremena stvaranja (eng. compile time)“ („Android Studio“, 2024).

Oni za automatiziran rad preporučaju Dagger biblioteku za Android Studio, bio to jezik Java ili Kotlin. Razlog tomu je da Dagger-a održava Google. O Dagger-u još govore „Dagger

olakšava korištenje injekcije ovisnosti u vašoj aplikaciji na način da stvara i upravlja grafom ovisnosti umjesto vas. Omogućuje potpuno statičke ovisnosti i ovisnosti o vremenu izvođenja rješavajući mnoga pitanja razvoja i performansi rješenja temeljenih na refleksiji kao što je Guice“ („Android Studio“, 2024).

Osim Dagger-a preporučuju Hilt kojemu je osnova Dagger. „Hilt je Jetpack-ova preporučena biblioteka za injekciju ovisnosti u Android-u. Hilt definira standardni način rada za injekciju ovisnosti u vašoj aplikaciji pružajući spremnik za svaku Android klasu u vašem projektu i automatski upravljajući njihovim životnim ciklusima za vas“ („Android Studio“, 2024).

Sada znamo da postoji ručno i automatizirano korištenje. Dobili smo osnovne informacije o tome zašto bi koristili injekciju ovisnosti automatizirano i zašto ju ne bi koristili ručno. Isto tako dobili smo informaciju o tome da postoje biblioteke za automatizirano korištenje i da je jedna koju najviše preporučaju Hilt pošto se bazira na Dagger-u. Sada ćemo pojasniti ručno korištenje injekcije ovisnosti.

4.4.3. Ručno korištenje injekcije ovisnosti

Ukoliko imamo velike aplikacije ručno postavljanje ovisnosti može postati naporno i kaotično. Unutar android dokumentacije kažu: „Postoje problemi s ovim pristupom:

- Postoji mnogo dupliciranog koda. Ako želite stvoriti još jednu instancu `LoginViewModela` u drugom dijelu programskog koda, imat ćete dupliciranje programskog koda.
- Zavisnosti se moraju deklarirati redom. Morate instancirati `UserRepository` prije `LoginViewModela` kako biste ga stvorili.
- Ponovo korištenje objekata je teže. Ako želite ponovo koristiti `UserRepository` u više značajki, morali biste ga natjerati da slijedi uzorak jednog elementa (eng. singleton pattern). Uzorak jednog elementa otežava testiranje jer svi testovi dijele jednu istu instancu“ („Android Studio“, 2024).

Na našem C# primjeru to izgleda da je njihov `LoginViewModela` kao naš `Salutation`. Njihov `UserRepository` je kao naš `ConsoleMessageWriter`. Ako stvorimo još instanca `Salutation` klase, u nekom drugom dijelu koda, taj kod će se nepotrebno duplicirati. Druga stvar je da moramo kreirati prvo novi `ConsoleMessageWriter` pa tek onda `Salutation`, ali što ako nemamo implementaciju ni klasu `ConsoleMessageWriter` -a? I ponovo korištenje istog objekta je otežano. Kako se boriti s ovime? Njihov prijedlog je: „Kako biste riješili problem ponovnog korištenja objekata, možete stvoriti vlastitu klasu spremnika ovisnosti koju koristite za dobivanje ovisnosti“ („Android Studio“, 2024). Stavljaju sve ovisnosti koje se koriste preko cijele aplikacije u tu klasu, tj. spremnika. Nova kreirana klasa koju su nazvali „`AppContainer`“ u sebi bi sadržavala `ConsoleMessageWriter` i ostale klase sličnog tipa. „Na ovaj način nemate

eng. singleton `ConsoleMessageWriter`. Nego imate `AppContainer` koji se dijeli na sve aktivnosti koji sadrži objekte iz grafa i stvara instance tih objekata koje druge klase mogu koristiti“ („Android Studio“, 2024).

Ukoliko bi klasa `Salutation` trebala biti na više mjesta, možemo kreirati klasu koja će ju stvoriti na jednom mjestu, tj. prebaciti njeno kreiranje u spremnik. Ako imamo više jednakih klasa kao `Salutation`, onda ona zapravo kreiramo opći način kreiranja instance bilo koje proslijeđene klase. Sa ovakvim rješenjem android developeri kažu: „Ovaj je pristup bolji od prethodnog, ali još uvijek postoje neki izazovi oko kojih trebamo promisliti:

- Morate sami upravljati `AppContainer`-om, ručno stvarajući instance za sve ovisnosti.
- Još uvijek postoji mnogo ponavljajućeg koda. Morate ručno stvoriti eng. factory ili parametre ovisno o tome želite li ponovno koristiti objekt ili ne.“ („Android Studio“, 2024).

Još upozoravaju o tome da: „Kada vaša aplikacija postane veća i počnete uvoditi različite funkcionalnosti, javlja se još više problema:

- Kada imate različite tokove, možda biste željeli da objekti samo žive u opsegu toka. Na primjer, kada stvarate `LoginUserData` (koji se može sastojati od korisničkog imena i lozinke koji se koriste samo kod prijave) ne želite zadržati podatke iz starog tijeka prijave od drugog korisnika. Želite novu instancu za svaki novi tijek…
- Optimiziranje aplikacijskog grafa i toka spremnika također može biti teško. Morate imati na umu da ne zaboravite izbrisati instance koje vam ne trebaju, ovisno o tijeku u kojem se nalazite“ („Android Studio“, 2024).

Moramo biti jako oprezni na životni vijek naših dijelova programskog koda, kada obrisati nešto i kada stvoriti nešto. Ukratko na cijelu ovu situaciju sa ručnim korištenjem injekcije ovisnosti kažu: „Injekcija ovisnosti dobra je tehnika za stvaranje skalabilnih i lako testiranih Android aplikacija. Koristite spremnik kao način dijeljenja instanci klasa u različitim dijelovima vaše aplikacije kao glavno (centar) mjesto za stvaranje instanci klasa pomoću eng. factories. Kada vaša aplikacija postane veća, primjećivati ćete da pišete puno dupliciranog koda (kao što su eng. factories), što može biti sklono pogreškama. Također morate upravljati opsegom i životnim ciklusom spremnika sami, optimizirajući i odbacujući spremnike koji vam više nisu potrebni kako biste oslobodili memoriju tj. RAM. Ako ovo radite na pogrešan način, to može dovesti do suptilnih grešaka i curenja memorije u vašoj aplikaciji“ („Android Studio“, 2024).

Kako bi bolje shvatili ovaj koncept dajem primjer. Koristimo njihov koncept za `AppContainer` da pokažemo ručno korištenje ovog koncepta u Android Studi-u. Jezik koji se koristi je Kotlin. Prvo kreiramo prazan projekt. Izaberite „Empty Views Activity“. Odaberite Minimum SDK „API 29 (“Q“; Android 10.0)“ i Build configurations language je „Kotlin DSL (build.gradle.kts)“. Ostalo možemo popuniti po izboru. Kreiran activity_main.xml stavite

`TextView` id. Nakon toga stvaramo sučelje `IMessageWriter` sa funkcijom **`write`**. Kod za to sučelje izgleda ovako:

```
1 interface IMessageWriter {
2     fun write(message: String)
3 }
```

Programski Kod 9 - IMessageWriter sučelje – ručna injekcija ovisnosti – Kotlin

Nakon sučelja implementiramo sučelje, ono izgleda ovako:

```
1 import android.widget.TextView
2
3 class ScreenMessageWriter (var textView: TextView) : IMessageWriter {
4     override fun write(message: String) {
5         textView.text=message
6     }
7 }
```

Programski Kod 10 – ScreenMessageWriter – ručna injekcija ovisnosti – Kotlin

Kao što možemo primijetiti, ovi dijelovi koda su vrlo slični primjeru sa C#-om. Naš `ConsoleMessageWriter` sada je **`ScreenMessageWriter`**, pošto prikazujemo na ekranu, a ne u konzoli. Isto tako dodajemo `textView` kako bi mogli prikazati dani tekst na ekran.

Nakon te implementacije slijedi klasa **`Salutation`**. Ona je jednaka kao i kod primjera sa C#-om, a izgleda ovako:

```
1 class Salutation(private val writer: IMessageWriter) {
2     fun exclaim() {
3         writer.write("Hello DI!")
4     }
5 }
```

Programski Kod 11 – Salutation – ručna injekcija ovisnosti – Kotlin

U glavnoj dokumentaciji za android studio, korištena je klasa `AppContainer`, naša klasa se zove **`DependencyContainer`**, a radi isto kao i njihova. Kreira sve potrebne komponente tako da se smanji duplikacija koda. Ona u sebi ima `textView` koji kasnije koristimo za spajanje s našim `textView`-om koji se nalazi u activity_main.xml datoteci. Isto tako stvaramo `writer` i `salutation` varijable. Ona izgleda ovako:

```
1 import android.content.Context
2 import android.widget.TextView
3
4 class DependencyContainer (context: Context) {
5     private val textView: TextView = TextView(context).apply{
6         text="Waiting for a message..."
7     }
```

```
 8     val writer: IMessageWriter = ScreenMessageWriter(textView)
 9     val salutation: Salutation = Salutation(writer)
10 }
```

Programski Kod 12 – DependencyContainer – ručna injekcija ovisnosti – Kotlin

I za kraj imamo u glavnom programu (`MainActivity`) naš kod koji sve ove klase spaja. Kreiramo instancu `DependencyContainer`. Nakon toga dohvaćamo `textView`-a po ID-u. Spajamo ih i pozivamo metodu `exclaim` iz `Salutation`-a. To izgleda ovako:

```
 1 class MainActivity : AppCompatActivity() {
 2     private lateinit var container: DependencyContainer
 3
 4     override fun onCreate(savedInstanceState: Bundle?) {
 5         super.onCreate(savedInstanceState)
 6         enableEdgeToEdge()
 7         setContentView(R.layout.activity_main)
 8
 9         var textView =
10                 findViewById<TextView>(R.id.textView)
11         /*val writer =
12                 ScreenMessageWriter(textView)
13         var salutation= Salutation(writer)
14         salutation.exclaim()*/
15
16         container =
17                 DependencyContainer(this)
18         (container.writer as ScreenMessageWriter)
19                         .textView = textView
20         container.salutation.exclaim()
21
22         ViewCompat
23                 .setOnApplyWindowInsetsListener(
24                 findViewById(R.id.main)
25                 ) { v, insets ->
26             val systemBars =
27                         insets.getInsets(
28                                 WindowInsetsCompat
29                                 .Type
30                                 .systemBars()
31                         )
32             v.setPadding(
33                                 systemBars.left,
34                                 systemBars.top,
35                                 systemBars.right,
36                                 systemBars.bottom
37                         )
38             insets
39         }
40     }
41     override fun onDestroy() {
42         super.onDestroy()
43         container
44     }
```

```
45 }
```

Programski Kod 13 – Prikaz glavnog programa – ručna injekcija ovisnosti – Kotlin

U komentarima je programski kod koji ne koristi `DependencyContainer` klasu. Pa ukoliko želite isprobati kako to funkcionira samo od komentirajte taj kod i za komentirajte linije 16, 17, 18, 19 i 20.

Sada kad smo dobili sliku o tome kako se koristi injekcija ovisnosti ručno, koji su njeni problemi, kako ih razriješiti, na što moramo paziti i primjer programskog koda, iduća stavka je kako se ovaj koncept koristi automatski. Pošto kao što je navedeno, bolje je koristiti automatski da se ne moramo brinuti oko puno drugih stavki.

4.4.4. Automatizirano korištenje injekcije ovisnosti

Za automatizirano korištenje injekcije ovisnosti koristimo Hilt. Objašnjavamo i dajemo primjer korištenja. Kao primjer koristimo codelab iz android studio domumentacije zvan „Using Hilt in your Android app“ kojeg je napravio Manuel Vivo (Manuel Vivo, 2023). Prolazimo samo osnovne lekcije od „2. Getting set up“ do „8. Providing interfaces with `@Binds`“.

Što je Hilt? Kao što android developer dokumentacija kaže „Hilt je biblioteka inverzije ovisnosti za Android koja smanjuje ponavljanje izvođenja inverzije ovisnosti ručno u projektu. Izvođenje inverzije ovisnosti ručno zahtijeva da stvorite svaku klasu i njezine ovisnosti i da koristite spremnike za ponovnu upotrebu i upravljanje ovisnostima“ („Android Studio“, 2024).

Što Hilt pruža? „Hilt pruža standardni način korištenja injekcije ovisnosti u vašoj aplikaciji tako da osigura spremnike za svaku Android klasu u vašem projektu i upravlja njihovim životnim ciklusima“ („Android Studio“, 2024).

Na čemu je Hilt sagrađen? „Hilt je izgrađen na vrhu popularne biblioteke injekcije ovisnosti zvane Dagger kako bi dobio benefite od ispravljanja vremena pokretanja, performanse tijekom izvođenja, skalabilnost i podršku za Android Studio koju Dagger pruža“ („Android Studio“, 2024).

Što je nužno za Hilt? Svaka klasa koja ima aplikacijsku klasu, mora u sebi imati `@HiltAndroidApp`. „`@HiltAndroidApp` pokreće Hilt-ovo generiranje koda, uključujući osnovnu klasu za vašu aplikaciju koja služi kao spremnik ovisnosti na razini aplikacije. Ova generirana Hilt komponenta je spojena na životnom ciklusu `Application` objekta i pruža mu ovisnosti. Osim toga, to je roditeljska komponenta aplikacije, što znači da druge komponente mogu pristupiti ovisnostima koje ona pruža“ („Android Studio“, 2024).

Kako se ubacuje i pružaju ovisnost za druge klase? „Nakon što je Hilt postavljen u vašoj aplikacijskoj klasi i komponenta na razini aplikacije je dostupna, Hilt može pružiti ovisnosti drugim klasama koje imaju napomenu `@AndroidEntryPoint`“ („Android Studio“, 2024).

„Hilt podržava klase:

Slika 3. Prikaz podržavanih aktivnosti klase

Ako označite Android klasu s `@AndroidEntryPoint`, onda morate označiti Android klase koje ovise o njoj. Na primjer, ako označite fragment, morate označiti i sve aktivnosti u kojima koristite taj fragment. Za dobivanje ovisnosti o komponenti upotrijebite komponentu `@Inject` za izvođenje injekcije ovisnosti. Klase koje Hilt injektira mogu imati druge osnove koje isto koriste injektiranje. Takve klase ne trebaju `@AndroidEntryPoint` ako su apstraktne. Za izvođenje injekcije ovisnosti, Hilt mora znati kako osigurati instance potrebnih ovisnosti iz odgovarajuće komponente. Vezanje sadrži potrebne informacije za pružanje instanci tipa kao ovisnosti.

Jedan od načina pružanja obvezujućih informacija Hiltu je injekcija konstruktora. Upotrijebite `@Inject` na konstruktoru klase tako da Hilt zna kako osigurati instance te klase.“ („Android Studio“, 2024).

Klase koje su označene s `@AndroidEntryPoint` automatski generiraju i ubacuju potrebne Hilt komponente tako da se injekcija ovisnosti može uključiti. Unutar tih klasa koristimo `@Inject` i on dohvaća povezane klase. Klase se povezuju preko konstruktora, ali što ako to ne možemo? Kako onda povezati te klase? Korištenjem `@Module` i `@InstallIn`. Kažu: „Ponekad se zadan tip ne može ubaciti preko konstruktora. To se može dogoditi iz više razloga. Na primjer, ne možete ubaciti konstruktor na sučelje. Također ne možete ubaciti konstruktor na tip koji ne posjedujete, kao što je klasa iz vanjske biblioteke. U tim slučajevima možete Hilt-u pružiti povezujuće informacije pomoću Hilt modula“ („Android Studio“, 2024).

Pa što je to `@Module`? „Hilt modul je klasa koja je označena s `@Module`. Poput modula Dagger, obavještava Hilt kako osigurati instance određenih vrsta. Za razliku od Dagger modula, Hilt module morate označiti s `@InstallIn` da kažete Hilt-u kojoj će klasi svaki modul biti korišten ili instalirati. Ovisnosti koje date Hilt-u, njegovi moduli dostupne su u svim generiranim komponentama koje su povezane s klasom u koju instalirate Hilt modul“ („Android Studio“, 2024).

Za lakše ubacivanje instanci sučelja koristi se `@Binds`. „`@Binds` govori Hilt-u koju implementaciju treba koristiti kada treba osigurati instancu sučelja“ („Android Studio“, 2024). Što `@Binds` radi Hilt-u?

- „Povratni tip funkcije govori Hilt-u koje instance sučelja funkcija pruža.
- Funkcijski parametar govori Hilt-u koju implementaciju treba osigurati“ („Android Studio“, 2024)

Što se tiče vanjskih biblioteka, za njih koristimo `@Provides`. Što on radi Hilt-u?

- „Povratni tip funkcije govori Hilt-u koju vrstu funkcija pruža instance.
- Funkcijski parametri govore Hilt-u o ovisnostima odgovarajućeg tipa.
- Tijelo funkcije govori Hilt-u kako osigurati instancu odgovarajućeg tipa. Hilt izvršava tijelo funkcije svaki put kada treba osigurati instancu tog tipa“ („Android Studio“, 2024).

Sada smo obuhvatili neke osnovne dijelove Hilt-a. Kako bi ovo bolje sjelo dajemo primjer „Using Hilt in your Android app“ od autora Manuel Vivo (Manuel Vivo, 2023). U ovom primjeru prikazujemo kako se koristi injekcija ovisnosti preko Hilt-a. Prikazuje se kako krenuti i klonirati kod, anotacije `@Inject`, `@HiltAndroidApp`, `@AndroidEntryPoint` i `@HiltViewModel` za korištenje aktivnosti, fragmenata i `ViewModel`-a. Isto tako `@Module` i `@InstallIn` anotacije za vanjske ovisnosti. Aplikacija prikazuje tri gumba, kod klika svaki dohvaća datum i vrijeme kada je pritisnut i njegovo ime. Te informacije sprema u log i vidljive su kada kliknemo gumb „See all logs“. Isto možemo obrisati sve logov-e sa „Delete Logs“. Svaka klasa unutar com.example.android.hilt radi slijedeće:

Paket „dana“ imamo:

- `AppDatabase` – klasa baze podataka za Room. Pružanje baze preko SQLite-a. Pohrana zapisa logov-a.
- `Log` – tablica u bazi podataka. Tri svojstva: poruka, vrijeme i id. Id je automatski generiran
- `LogDao` – sučelje koje definira metode `getAll, insertAll i nukeTable`.
- `LoggerLocalDataSource` – klasa koja upravlja između sučelja i SQLite baze. Koristi `LogDao` sučelje i implementira njihove metode.

Paket „navigator“ imamo:

- `AppNavigator` – sučelje za navigaciju u aplikaciji. Metoda `nagivateTo` služi za navigaciju različitih zaslona u aplikaciji.
- `AppNavigatorImpl` – implementacija sučelja `AppNavigator`. Implementira navigaciju različitih zaslona.

Paket „ui“ imamo:

- `ButtonsFragment` – prikazuje fragment korisničkog sučelja. Sadrži gumbove čije se interakcije zabilježe i pohrane u bazu. Isto tako koristi `AppNavigator` za navigaciju zaslona. Isto ima gumb za brisanje podataka iz baze.
- `LogsFragment` – prikazuje fragment korisničkog sučelja. Dohvaća i prikazuje podatke iz baze u log-u.
- `MainActivity` – prikazuje sve što se nalazi na ekranu. Upravlja navigacijom aplikacije. Kreira sve kada se aplikacija otvori, i upravlja tipkom nazad.

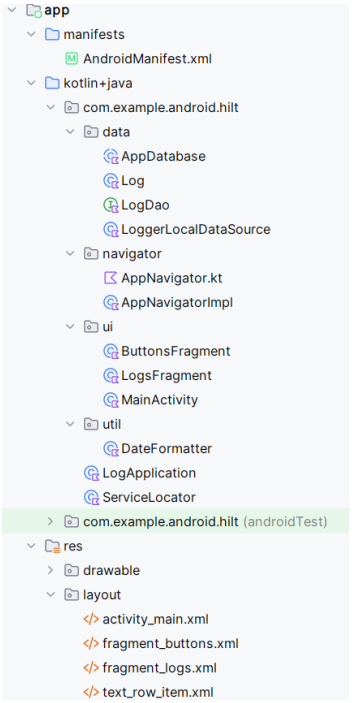
Paket „util“ imamo:

- `DateFormatter` – formatira datum na: dan mjesec godina sati:minute:sekunde

I klase:

- `LogApplication` – održava globalno stanje aplikacije. Inicijalizira `ServiceLocator` koja pruža ovisnosti cijeloj aplikaciji.
- `ServiceLocator` – pruža ovisnosti cijeloj aplikaciji. Kreira instance `LoggerLocalDataSource`, `DateFormatter` i `AppNavigator`.

Kako to izgleda u Android Studi-u:



Slika 4. Prikaz com.example.android.hilt mapa

Ovaj programski kod je dobar primjer ručne injekcije ovisnosti. Ali ćemo ga urediti tako da koristi automatsku injekciju ovisnosti pomoću Hilt biblioteke. Prvi korak je klonirati projekt. U uputama pod „2. Getting set up“ možete ga klonirati preko github-a ili direktno skinuti ZIP i raspakirati ga (Manuel Vivo, 2023). Nakon toga otvorite projekt i pričekajte da se sve otvori i

složi kako treba. Nakon nekog vremena trebali bi ste moći pokrenuti aplikaciju. Drugi korak je podešavanje `build.gradle` -a. Prvi `build.gradle` koji ima u zagradi (Project: codelab-android-hilt-main) ima starije verzije kotlina i hilt-a. Možete ih promijeniti na novije verzije. Kod ovog primjera kotlin verzija je stavljena na 1.9.0 i hilt verzija na 2.51.1. Što se tiče ostalih promjena, promijenjen je unutar buildscript-a unutar dependencies, classpath koji u sebi ima „`com.android.tools.build.gradle: 7.3.0`“ na verziju 7.4.2. Sada je taj `build-gradle` podešen. Drugi `build-gradle`, tj. onaj koji u zagradi ima (Module:app) unutar android, unutar `compileOptions`, verzije `sourceCompatibility` i `targetCompatibility` stavljena je na 1.9 (a kod kloniranja je 1.8). Sada kada smo podesili oba 2 `build-gradle`-a možemo krenuti na primjer.

Njihov `LogApplication` ovisi o `ServiceLocator` -u. To se vidi u liniji 27. Kako bi prebacili to na Hilt skroz gore prije kreiranja klase `LogApplication` stavljamo `@HiltAndroidApp`. Zapravo želimo kompletno maknuli ovisnosti sa klasom `ServiceLocator`. Klasu **`LogsFragment`** želimo automatizirati, pošto trenutno ručno stavlja podatke s `onAttach()`. Kako bi ju prebacili da radi s Hilt-om, prije njenog kreiranja stavit ćemo `@AndroidEntryPoint`. Taj dio koda izgleda ovako:

```
1 @AndroidEntryPoint
2 class LogsFragment : Fragment() {
3
4     @Inject lateinit var logger: LoggerLocalDataSource
5     @Inject lateinit var dateFormatter: DateFormatter
```

Programski Kod 14 – primjer s Hilt-om – početak klase LogsFragment (Manuel Vivo, 2023)

„S `@AndroidEntryPoint`, Hilt će stvoriti spremnik ovisnosti koji je pridružen životnom ciklusu **`LogsFragment`** i moći će ubaciti instance u **`LogsFragment`**“ (Manuel Vivo, 2023) Kako bi osigurali da je **`LogsFragment`** neovisan o `ServiceLocator` -u, stavljamo `@Inject` pored `logger`-a i `dateFormatter` -a. Isto tako brišemo funkcije `onAttach` i `populateFields`. Sada naš **`LogsFragment`** više ne ovisi o `ServiceLocator` -u, nego radi preko Hilt-a i on popunjava njegova polja. Ali još nismo gotovi. Što smo bili rekli u teoriji? Što nam još fali? Otvorite klasu `DataFormatter` i dodajte joj `@Inject` **`constructor()`**, isto napravite i za **`LoggerLocalDataSource`**. Taj kod izgleda ovako:

```
1 class
2 DateFormatter @Inject
3 constructor()
4 { ... }
5 class
6 LoggerLocalDataSource @Inject
7 constructor(private val logDao: LogDao)
8 { ... }
```

Programski kod 15 – primjer s Hilt-om – dodavanje konstruktora i Inject kod DateFormatter i LoggerLocalDataSource klasa (Manuel Vivo, 2023)

Sada klasa **LogsFragment** više ne koristi stvari iz ServiceLocator i radi preko Hilt-a. „Ako ponovno otvorite klasu ServiceLocator, vidjet ćete da imamo javno **LoggerLocalDataSource** polje. To znači da će ServiceLocator uvijek vratiti istu instancu **LoggerLocalDataSource** kad god se pozove. To se naziva postavljanje instance na spremnik. Za Hilt to možemo učiniti na način da možemo koristiti anotacije za traženje instanci u spremniku. Anotacija koja obuhvaća instancu u spremniku aplikacije je @Singleton“ (Manuel Vivo, 2023).

Što @Singleton radi? „Ova anotacija će napraviti da spremnik aplikacije uvijek pruža istu instancu bez obzira na tip koji je korišten kao ovisnost drugog tipa ili ga je potrebno umetnuti poljem“ (Manuel Vivo, 2023). To u programskom kodu izgleda ovako:

```
1 @Singleton
2 class LoggerLocalDataSource
3 @Inject
4 constructor(private val logDao: LogDao)
5 { ... }
```

Programski kod 16 – primjer s Hilt-om – dodavanje @Singleton kod LoggerLocalDataSource klasa (Manuel Vivo, 2023)

„Sada Hilt zna kako osigurati instance **LoggerLocalDataSource**. Međutim, tip ima tranzitivne ovisnosti! Da bi osigurao instancu **LoggerLocalDataSource**, Hilt također mora znati kako osigurati instancu LogDao“ (Manuel Vivo, 2023). Pošto je LogDao sučelje, ne možemo koristiti @Inject. Što bi mogli koristiti umjesto toga?

Za rješenje ovog izazova kažu: „Hilt modul je klasa označena s @Module i @InstallIn. @Module govori Hilt-u da je ovo modul, a @InstallIn govori Hilt-u spremnike u kojima su povezivanja dostupna određivanjem Hilt komponente. Možete zamisliti Hilt komponentu kao spremnik. Za svaku Android klasu koju može ubaciti Hilt, postoji pridružena Hilt komponenta. Na primjer, spremnik Application povezan je sa SingletonComponent, a spremnik Fragment je povezan s FragmentComponent“ (Manuel Vivo, 2023). Sada možemo kreirati novi objekt zvan DatabaseModule, a njega možemo smjestiti u novi paket zvan „di“.

„Budući da je LoggerLocalDataSource obuhvaćen spremnikom aplikacije, LogDao vezanje mora biti dostupno u spremniku aplikacije. Mi specificiramo taj zahtjev pomoću @InstallIn prosljeđivanjem klase komponente Hilt koja je s njom povezana (tj. SingletonComponent::class)“ (Manuel Vivo, 2023). To izgleda ovako:

```
@InstallIn(SingletonComponent::class)
@Module
object DatabaseModule {

}
```

Programski kod 17 – primjer s Hilt-om – stvoren objetk DatabaseModule (Manuel Vivo, 2023)

„U implementaciji klase `ServiceLocator`, instanca `LogDao` se dobiva pozivanjem `logsDatabase.logDao()`. Stoga, kako bismo osigurali instancu `LogDao` -a imamo tranzitivnu ovisnost nad klasom `AppDatabase`“ (Manuel Vivo, 2023).

Sada ćemo na našu funkciju koja će biti unutar `DatabaseModule` objekta staviti `@Provides`.

„Tijelo funkcije funkcije koja je označena s `@Provides` izvršit će se svaki put kada Hilt treba pružiti instancu te vrste. Povratni tip `@Provides` -označene funkcije govori Hiltu tip vezivanja, tip čije instance daje funkcija. Parametri funkcije su ovisnosti tog tipa“ (Manuel Vivo, 2023).

```
@Provides
    fun provideLogDao(database: AppDatabase): LogDao {
        return database.logDao()
    }
```

Programski kod 18 – primjer s Hilt-om – funkcija provideLogDao unutar objekta DatabaseModule (Manuel Vivo, 2023)

„Gornji kod (Programski kod 18 – primjer s Hilt-om – funkcija **`provideLogDao`** unutar objekta DatabaseModule (Manuel Vivo, 2023)) govori Hiltu da se baza podataka.logDao() mora izvršiti kada se daje instanca `LogDao`. Budući da imamo `AppDatabase` kao tranzitivnu ovisnost, također moramo reći Hiltu kako osigurati instance te vrste. Naš projekt također ne posjeduje klasu `AppDatabase` jer ju generira Room. Ne možemo konstruktor ubaciti `AppDatabase`, ali možemo upotrijebiti funkciju `@Provides` da to isto tako pružimo. Ovo je slično načinu na koji gradimo instancu baze podataka u klasi `ServiceLocator`“ (Manuel Vivo, 2023).

```
Provides
    @Singleton
    fun provideDatabase(
        @ApplicationContext appContext
        : Context
        ): AppDatabase {
        return Room.databaseBuilder(
            appContext,
            AppDatabase::class.java,
            "logging.db"
        ).build()
```

```
12      }
```

Programski kod 19 – primjer s Hilt-om – funkcija provideDatabase unutar objekta DatabaseModule (Manuel Vivo, 2023)

„Budući da uvijek želimo da Hilt pruži istu instancu baze podataka, označavamo metodu `@Provides` **`provideDatabase`** () s `@Singleton`. Svaki Hilt spremnik dolazi sa skupom zadanih povezanosti koja se mogu umetnuti kao ovisnosti u vaša prilagođena (eng. custom) povezivanja. Ovo je slučaj s `applicationContext`. Da biste joj pristupili, morate označiti polje s `@ApplicationContext`. Prije pokretanja aplikacije otvorite ui/MainActivity.kt datoteku i stavite `@AndroidEntryPoint` prije kreiranje klase **`MainActivity`**“ (Manuel Vivo, 2023). Pošto je još **`MainActivity`** povezan s **`ServiceLocator`**-om, to želimo maknuti. On koristi taj servis preko `AppNavigator`-a. Pošto je `AppNavigator` sučelje, ono treba koristiti `@Binds`. Kreiramo novu datoteku naziva **`NavigationModule`** u mapi „di“. Razlog kreiranja nove datoteke je:

- „Radi bolje organizacije, naziv modula trebao bi prenositi vrstu informacija koje pruža...
- Modul `DatabaseModule` instaliran je u `SingletonComponent`, tako da su povezivanja dostupna u spremniku aplikacije. Naše nove informacije o navigaciji (tj. `AppNavigator`) trebaju informacije specifične za aktivnost jer `AppNavigatorImpl` ima `Activity` kao ovisnost. Stoga se mora instalirati u spremnik Activity umjesto u spremnik `Application`, budući da su tamo dostupne informacije o `Activity`.
- Moduli hilt-a ne mogu sadržavati i ne statičke i apstraktne metode vezanja, tako da ne možete postaviti `@Binds` i `@Provides` komentare u istu klasu“ (Manuel Vivo, 2023).

Taj kod izgleda ovako:

```
1 @InstallIn(ActivityComponent::class)
2 @Module
3 abstract class NavigationModule {
4
5     @Binds
6     abstract fun bindNavigator(impl: AppNavigatorImpl): AppNavigator
7 }
```

Programski kod 20 – primjer s Hilt-om – apstraktna klasa NavigationModule (Manuel Vivo, 2023)

Sada još trebamo dati klasi `AppNavigatorImpl` do znanja da želimo raditi s Hilt-om. To radimo tako da dodamo `@Inject constructor(…)`. „`AppNavigatorImpl` ovisi o `FragmentActivity`. Zbog `AppNavigatora` instance koja se nalazi u spremniku aktivnosti, `FragmentActivity` je već dostupan kao unaprijed definirano vezanje. Sada kada Hilt ima sve informacije o ubacivanju za `AppNavigator` instance. Otvorite **`MainActivity`** i dodajte

`@Inject` na njega i obrišite navigator iz `onCreate` funkcije“ (Manuel Vivo, 2023). Taj kod izgleda kao:

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {

    @Inject lateinit var navigator: AppNavigator

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        if (savedInstanceState == null) {
            navigator.navigateTo(Screens.BUTTONS)
        }
    }

    ...
}
```

Programski kod 21 – primjer s Hilt-om – MainActivity (Manuel Vivo, 2023)

Preostala klasa nam je `ButtonsFragment`. Ona i dalje koristi `ServiceLocator`. Njoj dodajemo `@AndroidEntryPoint`, stavljamo `@Inject` na `LoggerLocalDataSource` i `AppNavigator`. Isto tako mičemo metode `onAttach` i `populateFields`. To izgleda ovako:

```
@AndroidEntryPoint
class ButtonsFragment : Fragment() {

    @Inject lateinit
        var logger: LoggerLocalDataSource
    @Inject lateinit
        var navigator: AppNavigator

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater
                .inflate(
                        R.layout.fragment_buttons,
                        container,
                        false
                )
    }

    override fun onViewCreated(
        view: View,
        savedInstanceState: Bundle?
        ) {  ...  }
}
```

Programski kod 22 – primjer s Hilt-om – ButtonsFragment klasa (Manuel Vivo, 2023)

„Primijetite da će instanca `LoggerLocalDataSource` biti ista kao ona koju smo koristili u `LogsFragmentu` budući da je tip ograničen na spremnik aplikacije. Međutim, instanca `AppNavigatora` biti će različita od instance u **`MainActivity`** jer je nismo dodijelili odgovarajućem spremniku `Activity`. U ovom trenutku **`ServiceLocator`** klasa više ne pruža ovisnosti pa je možemo potpuno ukloniti (obrisati) iz projekta. Jedina upotreba ostaje u klasi **`LogApplication`** gdje smo zadržali njenu instancu. Očistimo tu klasu jer nam više ne treba. Otvorite klasu **`LogApplication`** i uklonite korištenje **`ServiceLocator`**. Novi kod za klasu `Application` je: “ (Manuel Vivo, 2023)

```
1 @HiltAndroidApp
2 class LogApplication : Application()
```

Programski kod 23 – primjer s Hilt-om – očišćena klasa LogApplication (Manuel Vivo, 2023)

Još se ova klasa koristi i u klasi **`AppTest`**. Pa taj dio koda možete staviti kao komentar ili kompletno obrisati. Nakon toga pokrenete aplikaciju. Sada ta aplikacija radi s Hilt-om.

Ovime smo dobili jednostavne informacije o tome kako automatska injekcija ovisnosti radi, teorijski obradili neke njene osnove i prikazali njenu primjenu na primjeru iz android developer dokumentacija. Sada znamo na što trebamo paziti kod ručnog izvođenja injekcije ovisnosti i kako se boriti protiv nje koristeći automatiziran način preko Hilt-a.

4.5. Načini korištenja injekcije ovisnosti

Postoje tri glavna i najčešća korištenja injekcije ovisnosti, a to su preko konstruktora, preko metode i preko svojstva. Sa ovim smo se sreli u prethodnom primjeru unutar poglavlja „4.1. Koncept injekcije ovisnosti“. Sada razjašnjavamo svaku od injekcija ovisnosti.

Način injekcije konstruktora je taj da se ovisnost daje klasi koja ima svoj konstruktor preko njega. Ovo je jedan od najčešćih načina korištenja injekcije ovisnosti, tj. primjene inverzije ovisnosti. Instanciramo klasu ili objekt, nakon toga konstruktor će tražiti da proslijedimo nešto novo kreiranoj klasi ili objektu. Na taj način konstruktor osigurava da je injekcija uvijek izvršena, pošto ukoliko ništa ne proslijedimo konstruktoru, dobit ćemo grešku („Dependency Injection“, bez. Dat; Remya Mohanan, 2024; Mohammad Ramezani, 2020; Beribey, 2019; „freeCodeCamp“, 2018; Milica Dancuk, 2023; Techbalachandar, 2023; Milica Dancuk, 2023). Mohammad Ramezani (2024) u svojem članku „*The 3 Types of Dependency Injection*“ govori nam kada bi trebali koristiti injekciju preko konstruktora „Trebali biste koristiti injekciju konstruktora kada dana ovisnost ima dulji životni vijek nego jedna metoda. Proslijeđena ovisnost u konstruktor bi trebala biti korisna klasi u općenitom smislu, na način

da obuhvaća više metoda klase. Ukoliko se ovisnost koristi samo na jednom mjestu, bolje koristite injekciju metode“ (Mohammad Ramezani, 2024). Prednost ovog načina korištenja injekcije je ta da se ne moramo brinuti da li smo definirali ovisnost ili ne, pošto je obavezna. Nedostatak ovog načina je da ukoliko imamo previše stvari u konstruktoru, pa počinje biti prenatrpan podacima i na taj način otežava razumijevanje i jasnoću programskog koda („Dependency Injection“, bez. Dat; Remya Mohanan, 2024; Mohammad Ramezani, 2020; Beribey, 2019; „freeCodeCamp“, 2018; Milica Dancuk, 2023; Techbalachandar, 2023; Milica Dancuk, 2023).

Injekcija preko svojstva još je poznata po nazivu eng. setter injekcija. Pošto koristi set metodu kao prijenosnik injekcije ovisnosti. Set metoda dozvoljava injekciju u bilo koje vrijeme, pa ju to čini fleksibilnom. Ovo je jedna od najvećih prednosti i mana ovog načina injekcije. Radi tog razloga trebamo biti vrlo oprezni kada koristimo ovaj način injekcije, pošto nemamo sigurnost kao kod konstruktora. Taj manjak sigurnosti je taj da ne znamo da li su sve ovisnosti ubačene i provjerene prije nego što su korištene, pa trebamo biti oprezni i provjeriti prije uporabe da li smo sve napravili kako treba i provjerili da nam nešto ne fali („Dependency Injection“, bez. Dat; Remya Mohanan, 2024; Mohammad Ramezani, 2020; Beribey, 2019; „freeCodeCamp“, 2018; Milica Dancuk, 2023; Techbalachandar, 2023; Milica Dancuk, 2023). Mohammad Ramezani u članku „*The 3 Types of Dependency Injection*“ govori o tome zašto naša ovisnost ne bi nikada trebala biti opcionalna. „Injekcija svojstvom međutim uzrokuje Privremeno Spajanje (eng. Temporal Coupling) i kada pišete Linije poslovanja (eng. Line of Business) aplikacije, vaše ovisnosti nikada ne bi trebale biti izborne: umjesto toga trebali biste primijeniti Nulti Objekt (eng. Null Object) uzorak“ (Mohammad Ramezani, 2024). I zašto je to loš izbor „injekcija svojstva smatra se lošim u 98% scenarija jer skriva ovisnosti i ne garantira da će objekt biti umetnut kada se klasa stvori“ (Mohammad Ramezani, 2024).

Injekcija preko same metode i injekcija preko sučelja i metode su vrlo slične. Jedina razlika je ta da kod injekcije sučelja dodatno definiramo sučelje koje ima metodu. I tu definiranu metodu koristimo u našoj klasi. S druge strane injekcija preko metode odmah koristi metodu, bez prethodno kreiranog sučelja („Dependency Injection“, bez. Dat; Remya Mohanan, 2024; Mohammad Ramezani, 2020; Beribey, 2019; „freeCodeCamp“, 2018; Milica Dancuk, 2023; Techbalachandar, 2023; Milica Dancuk, 2023). Mohammad Ramezani u članku „*The 3 Types of Dependency Injection*“ govori o dva bitna slučaja korištenja injekcije metodom. „Kada implementacija ovisnosti varira i kada se ovisnost treba obnoviti nakon svakog korištenja. U oba slučaja, odluka je na pozivatelju da odluči koju implementaciju proslijediti metodi“ (Mohammad Ramezani, 2024). Prednost injekcije metodom kako je Remya Mohanan napisala u svojem članku „*What Is Dependency Injection? Meaning, Types, Control, Use, and Examples*“ kaže: „Ova metoda nudi veliku fleksibilnost za ubacivanje ovisnosti u različite točke kod životnog ciklusa objekta, potencijalno korisne za složene scenarije“ (Remya Mohanan,

2024). Isto tako je korisna ukoliko imamo klasu koja će koristiti više različitih ovisnosti. Veliki nedostatak injekcije metodom je sama kompleksnost koja nastaje u kodu. Pošto dodajemo više sučelja, nečitljivost i složenost programskog koda se povećava. Time je vrlo lako pretjerati i napraviti previše sučelja. Upravo iz tog razloga ovo je jedna od najmanje korištenijih metoda korištenja injekcije ovisnosti („Dependency Injection“, bez. Dat; Remya Mohanan, 2024; Mohammad Ramezani, 2020; Beribey, 2019; „freeCodeCamp“, 2018; Milica Dancuk, 2023; Techbalachandar, 2023; Milica Dancuk, 2023).

Sada kada smo pogledali načine korištenja ove metode injekcije ovisnosti možemo dobiti sliku koju od ove tri ćemo koristiti u praktičnom dijelu rada, a to je injekcija preko konstruktora. Ona je najviše korištena i jedna od lakših za korištenje, pošto zahtijeva od korisnika da kada pozove tu klasu, da obavezno proslijedi parametre unutar nje. Na taj način osigurali smo korištenje injekcije ovisnosti. Isto tako pitanje je da li koristiti automatsku ili ručnu injekciju? Odgovor je jednostavan. Pošto je ovo prvi puta da ću koristiti ovaj koncept u većem programu, bitno se upoznati s osnovama, a to je ručno korištenje. Kada se jednom ručno korištenje ovog koncepta savlada, onda lakše znamo koristiti automatsku injekciju ovisnosti.

Iz ovog možemo zaključiti da postoje tri glavna korištenja injekcije ovisnosti a to su preko konstruktora, metode i svojstva. Najviše korišten je preko konstruktora, ali trebamo biti oprezni da ne stavimo previše stvari unutar njega. Idući je preko svojstva pošto nam daje najveću slobodu, to je ujedno i najveća mana zbog ljudske zaboravljivosti. Metoda nam daje nešto između konstruktora i svojstva, ali je ipak najmanje korištena, pošto se odnosi samo na određenu metodu i ne obuhvaća ostatak koda.

4.6. Inverzija ovisnosti i injekcija ovisnosti

Sada kada imamo sliku o inverziji i injekciji ovisnosti, pitanje je, koje su razlike ta dva pojma? Nisu li oni gotovo pa isti? Odgovor je da nisu isti. Kako je jednostavno BuketSenturk (2024) u članku „Dependency Inversion vs Dependency Injection“ napisao: „Cilj inverzije ovisnosti je odvojiti module visoke razine od modula niske razine, promovirajući fleksibilniju bazu koda koju je moguće održavati. Dok je za injekciju ovisnosti: Cilj pomoći da injekcija postigne labav spoj između klasa, i olakša testiranje, održavanje i ponovnu upotrebu“.

Isto tako kaže „Inverzija ovisnosti je načelo dizajna koje vodi strukturu vašeg koda“ (BuketSenturk, 2024). Dok za Injekciju „Injekcija ovisnosti je specifična tehnika za implementaciju inverzije ovisnosti tako da ubacuje ovisnosti izvana umjesto da ih stvara iznutra“ (BuketSenturk, 2024).

Kao i u članku „The importance of the dependency inversion principle“ napisanog od Domenique Tilleuil-a i Guido Dechamps-a, oni kažu: „Injekcija ovisnosti je tehnika kojom osoba pridaje ovisnosti objektu. Namjera iza injekcije ovisnosti je postići odmak briga između izgradnje i upotrebe objekata. Ne navodi ništa o relativnoj važnosti između tih objekata ili ako se koristi apstrakcija. Injekcija ovisnosti je samo po sebi oblik šire tehnike inverzije kontrole (IOC). IOC sam po sebi može podržati DIP. Ali nije da primjenjujemo DIP kada koristimo DI ili IOC. Nijedan nam okvir ne može pomoći da odredimo što je visoka, a što niska razina. Niti s definiranjem odgovarajuće apstrakcije za razdvajanje to dvoje. Kada pokušavamo primijeniti DIP unutar naše baze koda, možemo se zapitati: Tko kreira nižu razinu implementacije apstrakcije ako se ona nalazi u nekom drugom modulu? Korištenje IOC spremnika, ovo je jednostavan izazov. IOC spremnik mogao bi stvoriti instancu modula niže razine i ubaciti je gdje je to potrebno. Dakle, IOC spremnik olakšava ubacivanje detalja niske razine u naše module visoke razine. Ali još uvijek moramo sami osigurati odgovarajuće apstrakcije. I dalje smo odgovorni za postavljanje apstrakcija na ispravno mjesto, pored smjera visoke razine. Dakle, je li IOC spremnik potreban kada se želi primijeniti DIP? Naravno da ne. Samo nam treba neka vrsta "glavnog" modula koji povezuje našu aplikaciju. "Glavni" može pristupiti svim potrebnim objektima i spojiti ih kako treba. Ovo je čisto tehnička stvar koju bismo mogli sami riješiti, ali to je riješen problem za koji često radije koristimo IOC. Ali kada koristimo IOC-a on ne jamči primjenu DIP-a. Na nama je da definiramo ispravne arhitektonske granice i odvajanja smjera. Dakle, DI ne implicira DIP i obrnuto. Odvojene stvari“ (Domenique Tilleuil i Guido Dechamps, 2019). Iz ovoga možemo vidjeti da ukoliko koristimo injekciju ovisnosti, ne znači da koristimo inverziju ovisnosti. Mi ju sami moramo svjesno koristiti. Ali korištenjem ovih koncepata taj nam se posao olakšava.

Najlakše moguće možemo reći razliku ova dva pojma na način da jedan odgovor može biti da je inverzija ovisnosti kao teorija, dok je injekcija kao praksa, tj. primjena te teorije. Dok drugi da je inverzija zapravo osiguravanje da su sučelje, implementacija i njihovo korištenje odvojeno, dok injekcija odlučuje na koji način će to biti povezano (preko konstruktora, metode ili svojstva). Sada smo obuhvatili sve osnove ovih dviju ovisnosti. I sa sigurnošću znamo što koja radi.

4.7. Primjena injekcije ovisnosti u web tehnologijama

Već smo demonstrirali kako koristiti inverziju ovisnosti preko injekcije ovisnosti u C# i Kotlinu. Sada kao zadnju primjenu pokazujemo korištenje injekcije ovisnosti kod web aplikacija. Ovaj primjer prikazan je preko jednostavne web stranice i koristi programski kod iz C# primjera, samo je pretvoren u jezik JavaScript. Kao što smo naveli, jezik koji koristimo je JavaScript i prikazujemo rezultat unutar HTML stranice. Isto kao i kod primjera za desktop

aplikacije unutar C#, radimo po jedan primjer s konstruktorom, svojstvom i metodom. Bitno je za napomenuti da JavaScript nema sučelja, nego koristimo klase kao zamjena.

4.7.1. Prvi primjer – JavaScript – injekcija pomoću konstruktora

U prvom koraku kreiramo glavnu HTML stranicu. Trebamo neki element preko kojeg prikazujemo našu poruku, u ovom slučaju to je `div` element. Isto tako ubacujemo skriptu na našu glavnu JavaScript datoteku kojoj dajemo ime Main.js. To sve izgleda ovako:

```
<!DOCTYPE html>
<html lang="hr">
<head>
    <meta charset="UTF-8">
    <title>Inverzija ovisnosti</title>
</head>
<body>
    <h1>Inverzija ovisnosti - JavaScript</h1>
    <div id="message"></div>
    <script type="module" src="./Main.js"></script>
</body>
</html>
```

Programski Kod 24 – Prikaz glavne stranice – injekcija preko konstruktora – HTML

Drugi korak je kreiranje sučelja `IMessageWriter`. Kao što smo rekli, sučelja ne postoje u JavaScriptu pa će `IMessageWriter` zapravo biti klasa. Unutar te klase slažemo metodu imena `Write` koja prima poruku unutar sebe. To izgleda ovako:

```
export class IMessageWriter {
    Write(message) {
        throw new Error("Method not implemented");
    }
}
```

Programski Kod 25 – Prikaz sučelja IMessageWriter – injekcija preko konstruktora – JavaScript

Koristimo `throw new Error` jer želimo reći korisniku da ovu metodu treba naslijediti i implementirati. Iduća klasa je `ConsoleMessageWriter`. Prvo uvozimo našu kreiranu `IMessageWriter` klasu tj. sučelje. Nakon toga proširujemo `ConsoleMessageWriter` klasu a to radimo s ključnom riječi `extends`. To je zapravo nasljeđivanje, naša klasa `ConsoleMessageWriter` naslijedila je klasu `IMessageWriter`. U idućem koraku kreiramo funkciju `Write` i unutar nje dohvaćamo element s identifikatorom `message`. Dohvaćeni element poruke tipa div nam pomaže u prikazu naše poruke na glavnoj stranici. I za kraj postavimo da je sadržaj tog dohvaćenog `div`-a jednak poruci. Kod klase izgleda ovako:

```
 1 import { IMessageWriter } from './IMessageWriter.js';
 2
 3 export class ConsoleMessageWriter extends IMessageWriter {
 4     Write(message) {
 5         const messageDiv = document.getElementById('message');
 6         if (messageDiv) {
 7             messageDiv.innerHTML = message;
 8         }
 9     }
10 }
```

Programski Kod 26 – Prikaz klase ConsoleMessageWriter – injekcija preko konstruktora – JavaScript

Nakon stvaranja `ConsoleMessageWriter`, stvaramo klasu **`Salutation`**. Prva stvar je da uvezemo klasu `IMessageWriter`. Nakon toga kreiramo konstruktor koji prima `writer`. Pošto JavaScript-u ne možemo direktno reći da `writer` treba biti tipa `IMessageWriter` stavljamo **`if`** funkciju koja provjeri da li je dan `writer` tipa `IMessageWriter`. I na kraju unutar konstruktora postavljamo `writer` na `writer`. Nadalje kreiramo metodu `Exclaim` koja poziva unutar sebe metodu `Write` i predaje joj tekst tipa „Hello DI!“. Na taj način osigurano je korištenje `Write` metode i ovisnost klase **`Salutation`** o sučelju `IMessageWriter`.

```
 1 import { IMessageWriter } from './IMessageWriter.js';
 2
 3 export class Salutation {
 4     constructor(writer) {
 5         if (!(writer instanceof IMessageWriter))
 6                 throw new Error(
 7                         "writer must implement IMessageWriter"
 8                 );
 9         this.writer = writer;
10     }
11
12     Exclaim() {
13         this.writer.Write('Hello DI!');
14     }
15 }
```

Programski Kod 27 – prikaz klase Salutation – injekcija preko konstruktora – JavaScript

Za kraj kreiramo naš program, a u ovom slučaju je to Main. Unutar njega uvozimo `Salutation` i `ConsoleMessageWriter` klase. Stvaramo slušni događaj koji unutar sebe ima varijablu writer koja instancira novi `ConsoleMessageWriter` i novi `Salutation` koji prima `writer`-a unutar sebe. I za kraj pozivamo `Exclaim` metodu. To izgleda ovako:

```
1 import { ConsoleMessageWriter } from './ConsoleMessageWriter.js';
2 import { Salutation } from './Salutation.js';
3
```

```
4 document.addEventListener('DOMContentLoaded', (event) => {
5     const writer = new ConsoleMessageWriter();
6     const salutation = new Salutation(writer);
7     salutation.Exclaim();
8 });
```

Programski Kod 28 – prikaz glavnog programa – injekcija preko konstruktora – JavaScript

Primjer možemo pokrenuti unutar Visual Studo Code programa. Otvaramo cijelu mapu u kojoj se nalaze maloprije kreirane datoteke. Dolazimo s mišem do html datoteke -> desni klik -> „Open with live server“. Ukoliko ta opcija nije vidljiva, potrebno je instalirati proširenje unutar Visual Studio Code-a zvano „Live Server“. Otvaramo prozor proširenja. Klik u tražilicu i tražimo „Live Server“. Odabiremo ga i klik Install.

Napomena: moguće je da će tražiti ponovo pokretanje Visual Studio Code-a.

4.7.2. Drugi primjer – JavaScript – inverzija pomoću svojstva

Kako bi prikazali kako koristiti inverziju ovisnosti pomoću svojstva, koristimo istu glavnu stranicu, istu `IMessageWriter` i `ConsoleMessageWriter` klasu. Samo mijenjamo klase Main i `Salutation`.

`Salutation` klasa je vrlo slična prethodnoj klasi Salutation kod primjera s metodom u C#-u – primjer „Programski Kod 5 – prikaz klase `Salutation` – injekcija preko svojstva – C#“. Klasa ima uvoz klase `IMessageWriter`. Kreiran je set `writer` koji prima `writer` i provjerava da li je instanceof `IMessageWriter`. Isto tako `writer` postavlja `writer` na `writer`. Set `Writer(writer)` koristi se tako da smo sigurni da naše svojstvo `Writer` možemo postaviti na `writer` kojeg mu da dodijelimo. Metoda `Exclaim` provjerava da li je `writer` postavljen i poziva metodi `Writer` sa porukom „Hello DI!“. To izgleda ovako:

```
 1 import { IMessageWriter } from './IMessageWriter.js';
 2
 3 export class Salutation {
 4     set Writer(writer) {
 5         if (!(writer instanceof IMessageWriter))
 6                 throw new Error(
 7                         "writer must implement IMessageWriter"
 8                 );
 9         this._writer = writer;
10     }
11
12     Exclaim() {
13         if (!this._writer)
14                 throw new Error("writer is not set");
15         this._writer.Write('Hello DI!');
16     }
17 }
```

Programski Kod 29 – Prikaz klase Salutation – injekcija preko svojstva – JavaScript

I za kraj naš main. Isti je koncept kao i u C# primjeru s metodom. Jedina razlika je što moramo ubaciti `ConsoleMessageWriter` i `Salutation` klase i postaviti slušača događaja. A to izgleda ovako:

```
1 import { ConsoleMessageWriter } from './ConsoleMessageWriter.js';
2 import { Salutation } from './Salutation.js';
3
4 document.addEventListener('DOMContentLoaded', (event) => {
5     const writer = new ConsoleMessageWriter();
6     const salutation = new Salutation();
7     salutation.Writer = writer;
8     salutation.Exclaim();
9 });
```

Programski Kod 30 – glavni dio implementacije – injekcija preko svojstva – JavaScript

Kao i kod prethodnog primjera, program se pokreće na isti način. Kao što možete vidjeti, programski kod je vrlo sličan kao kod C#-a.

4.7.3. Treći primjer – JavaScript – inverzija pomoću metode

Kako bi prikazali kako koristiti inverziju ovisnosti pomoću metode, koristimo istu glavnu stranicu, istu `IMessageWriter` i `ConsoleMessageWriter` klasu. Samo se mijenjaju klase Main i `Salutation`.

Klasa `Salutation` unutar sebe ima samo metodu `Exclaim` koja prima `writer`. Unutar nje obavezno provjerimo da li je writer tipa `IMessageWriter` i nakon toga pozovemo `Write` s porukom „Hello DI!“. To izgleda ovako:

```
1 import { IMessageWriter } from './IMessageWriter.js';
2
3 export class Salutation {
4     Exclaim(writer) {
5         if (!(writer instanceof IMessageWriter))
6                 throw new Error("writer must implement IMessageWriter");
7         writer.Write('Hello DI!');
8     }
9 }
```

Programski Kod 31 – prikaz klase Salutation – injekcija preko metode – JavaScript

Naš kod vrlo je sličan prethodnim JavaScript primjerima. Samo što ovoga puta kod poziva `Exclaim` metode, prosljeđujemo `writer`-a koji je tipa `ConsoleMessageWriter`. To izgleda ovako:

```
1 import { ConsoleMessageWriter } from './ConsoleMessageWriter.js';
2 import { Salutation } from './Salutation.js';
3
4 document.addEventListener('DOMContentLoaded', (event) => {
5     const writer = new ConsoleMessageWriter();
6     const salutation = new Salutation();
7     salutation.Exclaim(writer);
8 });
```

Programski Kod 32 – prikaz glavnog dijela – injekcija preko metode – JavaScript

Kao i kod prethodna dva primjera, programski kod se pokreće preko LiveServer-a.

Ovime smo pokazali da kada znamo logiku injekcije ovisnosti i inverzije ovisnosti, prebacivanje na drugu sintaksu, tj. programski jezik nije preteško.

4.7.4. Zaključak primjene injekcije ovisnosti kod različitih tehnologija

Kako bi implementirali injekciju ovisnosti, koristili smo razne tehnologije kao C#, Kotlin i JavaScript. Sam zaključak ove primjene je da je svaki kod zapravo identičan i identično se implementira ova ovisnost u raznim tehnologijama. Razlike u programskom jeziku nisu toliko značajne da previše utječu na glavnu logiku inverzije ovisnosti. Kada shvatimo logiku i ideju iza koncepta, a to je u ovom slučaju injekcija i inverzija ovisnosti, taj koncept se implementira na gotovo identičan način u raznim programskim jezicima. Ako još imamo i primjer implementacije u jednom programskom jeziku, a znamo ga i znamo neki drugi u kojem želimo raditi, taj će manevar biti još lakši. Pošto ćemo logiku shvatiti na onome na kojem je originalno napisan programski kod i prebacivanje logike će zahtijevati samo pažnju, strpljenje i naše shvaćanje koncepta.

4.8. Zaključak o injekciji ovisnosti

Ovim dijelom obuhvaćena je injekcija ovisnosti, od toga što je, kako se koristi i koje su joj prednosti. Znamo da postoji ručno i automatizirano rukovanje s injekcijom ovisnosti, opisali ih i dali primjere kako bi to izgledalo u Kotlin-u. Navedena su tri bitna načina korištenja injekcije ovisnosti i kako se koji koristi. Objašnjeno je da injekcija i inverzija ovisnosti nisu isti pojam i da se razlikuju. Dan je primjer u JavaScript-u i opisana je razlika u primjenu kod C#-a, Kotlin-a i JS-a. Isto tako nakon ovih informacija, sada možemo ući u korištenje ovih koncepata preko konstruktora na ručni način u praktičnom dijelu rada.

5. Praktični primjer inverzije ovisnosti u Android Studiu

U ovom dijelu rada fokus je na sam programski kod cijelog praktičnog dijela. Programski kod ovog rada možete pronaći na linku GitHub repository. U ovom dijelu rada fokus je na primjeni inverzije ovisnosti koristeći injekciju ovisnosti. Razvojna okolina je Android Studio i jezik Kotlin. Pobrojani i opisani funkcionalni zahtjevi kako bi znali o čemu je aplikacija. Isprobana je aplikaciju iz perspektive korisnika i date su slike kako ona radi i izgleda. Dizajn aplikacije prikazan je preko dijagrama klasa kako bi vidjeli povezanost klasa i sučelja. Prikazan je programski kod i objašnjeni svi bitni dijelovi kako bi bolje razumjeli kako funkcionira. Fokus je stavljen na dijelove koda koji u sebi imaju inverziju ovisnosti i injekciju ovisnosti..

5.1. Funkcionalni zahtjevi

Kao što smo prethodno naveli, prikazujemo funkcionalne zahtjeve. Kako bi lakše predočili funkcionalne zahtjeve, koristimo tablicu. Tablica sadrži oznake, naziv funkcionalnog zahtijeva i kratak opis.

Tablica 1. Prikaz funkcionalnih zahtjeva aplikacije

Oznaka	Naziv	Opis
F01	Prikaz sekcija i nivoa	Kada se aplikacija otvori prikazuju se sve sekcije koje u sebi sadrže nivoe. Svaki prikazani nivo korisnik može igrati. Svaki nivo prikazuje svoj naslov, broj i broj bodova koje je korisnik ostvario. Kada korisnik selektira određeni nivo, prikazuju mu se odabrani izazovi za rješavanje i učenje.
F02	Nivo	Svaki nivo ima predefinirane zadatke koje uzima iz lokalne baze. Svaki nivo u sebi ima razne zadatke koje korisnik rješava. Kada korisnik završi sve zadatke dobiva prikaz njegovih odgovora i točnih rješenja zadataka. Na temelju točno riješenih zadataka korisnik dobiva broj bodova koji se prikazuje pored nivoa na glavnom ekranu (F01).
F03	Zadaci	Kada korisnik otvori nivo prikazuje mu se zadatak po zadatak. Kada riješi zadatak otvara se drugi. Kada završi sve zadatke pokazuje se ekran sa njegovim odgovorima i točnim rješenjima. Zadaci su tipa: odaberi dali je tvrdnja točna ili netočna, odaberi

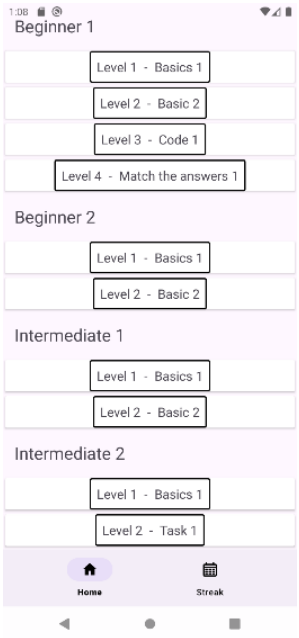
		jedan točan odgovor od više ponuđenih, odaberi više točnih odgovora od više ponuđenih, upiši tekst/programski kod koji nedostaje, spoji odgovore.
F04	Praćenje niza dana	Prikazuje koliko dana za redom smo bili u aplikaciji i riješili bilo koji nivo. Osim broja prikaza dana, prikazan je i kalendar na kojemu su označeni dani koje smo bili aktivni za redom. Ukoliko preskočimo dan, naš niz dana se resetira na nulu i ponovo počinje brojanje dana od trena kada ponovo riješimo zadatak.

Kroz ove funkcionalnosti prikazujemo inverziju i injekciju ovisnosti. Najveći fokus na kojem su ovi koncepti su na zadacima, pošto imamo više različitih zadataka, pa tako i više različitih implementacija i prikaza za svaki zadatak.

5.2. Prikaz aplikacije

Kako izgleda aplikacija? Kako funkcionira? Što sve korisnik može vidjeti i raditi? Kao navedeno u funkcionalnim zahtjevima, korisnik vidi sekcije i nivoe, rješava nivoe kroz razne tipova zadataka. Isto tako vidi sve dane za redom koje je bio aktivan i broj koliko dana za redom je riješio jedan nivo.

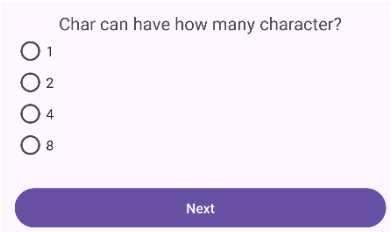
Korisnik kada otvori aplikaciju prvo vidi ekran čekanja da se podaci iz baze dohvate i prikažu na ekran. To čekanje traje par sekundi. Nakon čekanja korisnik vidi prikaz sekcija i nivoa. To izgleda ovako:



Slika 5. Prikaz ekrana ulaska u aplikaciju

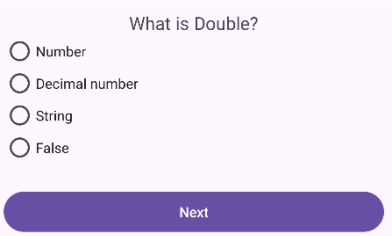
Kao što možemo vidjeti prikazane su sekcije (Beginner 1, Beginner 2, Intermediate 1, Intermediate 2,..) i nivoi u svakoj sekciji (Level 1 – Basic 1, Level 2 – Basic 2,..)

Nakon toga korisnik klikne na neki nivo i otvara mu se prikaz zadataka unutar nivoa. Kao što je navedeno u funkcionalnim zahtjevima, svaki zadatak je zasebno prikazan. To izgleda ovako:



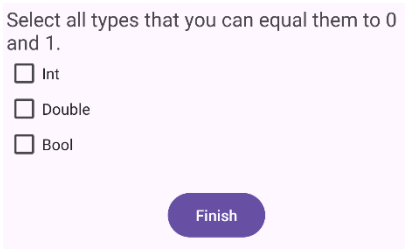
Slika 6. Prikaz zadatka 1 s više izbora i jednim točnim odgovorom unutar nivoa

„Slika 6. Prikaz zadatka 1 s više izbora i jednim točnim odgovorom unutar nivoa“ predstavlja prvi zadatak u prvom nivou. Ovo je zadatak s jednim točnim odgovorom, ali više izbora.



Slika 7. Prikaz zadatka 2 s više izbora i jednim točnim odgovorom unutar nivoa

„Slika 7. Prikaz zadatka 2 s više izbora i jednim točnim odgovorom unutar nivoa“ je drugi zadatak unutar prvog nivoa. Isto predstavlja zadatak s jednim točnim odgovorom, ali više izbora kao i „Slika 6. Prikaz zadatka 1 s više izbora i jednim točnim odgovorom unutar nivoa“.

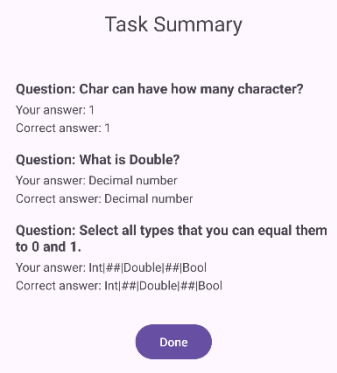


Slika 8. Prikaz zadatka s više izbora i više točnih odgovora unutar nivoa

„Slika 8. Prikaz zadatka s više izbora i više točnih odgovora unutar nivoa“ predstavlja treći zadatak unutar prvog nivoa. Ovaj zadatak je s više izbora kao i „Slika 6. Prikaz zadatka 1

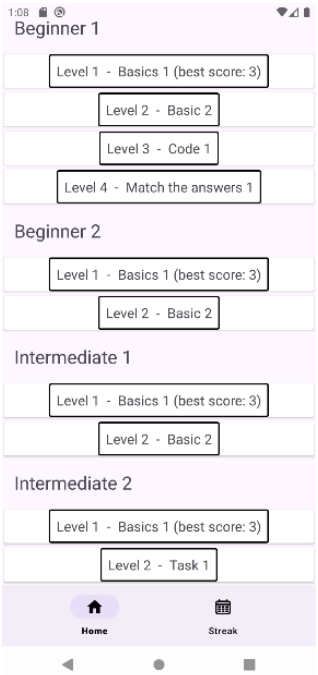
s više izbora i jednim točnim odgovorom unutar nivoa“ i „Slika 7. Prikaz zadatka 2 s više izbora i jednim točnim odgovorom unutar nivoa“, ali možemo odabrati više točnih tvrdnji, a ne samo jednu.

Nakon završetka svih zadataka unutar nivo, prikazuje se ekran koji prikazuje pitanja, naše odgovore i točne odgovore. Na taj način korisnik može provjeriti što mu je bilo točno, a što ne, tj. korisnik dobije povratnu informaciju o svojem uspjehu. Taj ekran izgleda:



Slika 9. Prikaz ekrana sažetka nakon riješenih zadataka unutar nivoa

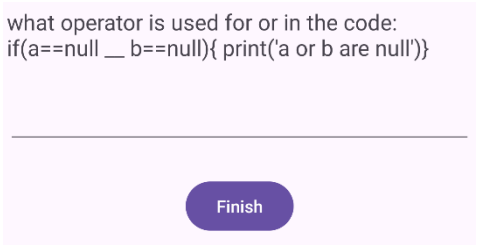
Nakon kada kliknemo „Done“ na „Slika 9. Prikaz ekrana sažetka nakon riješenih zadataka unutar nivoa“ prikazuje se glavni ekran i broj bodova koje smo ostvarili na nivou. Taj ekran izgleda:



Slika 10. Prikaz glavnog ekrana s najviše točnih stavki unutar nivoa.

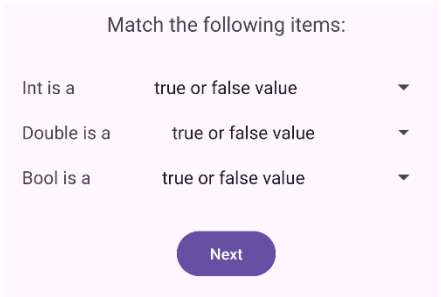
Kao što možemo primijetiti na „Slika 10. Prikaz glavnog ekrana s najviše točnih stavki unutar nivoa.“ Pored Level 1 – Basic 1 dodano je (best score: 3). Što znači da smo ostvarili 3 točna na tom nivou, tj. cijeli nivo smo točno riješili.

Prikazali smo odabir jednog s više opcija i odabir više s više opcija. Zadatak nadopuni izgleda ovako:



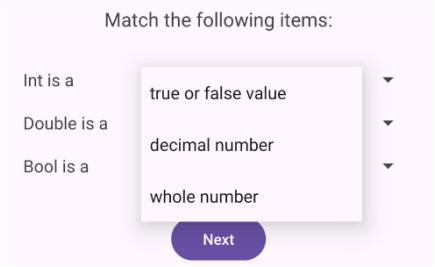
Slika 11. Prikaz zadatka nadopunjavanja

Kao što možemo vidjeti na „Slika 11. Prikaz zadatka nadopunjavanja“ prikazujemo na ekranu pitanje i možemo napisati odgovor. Zadatak gdje spajamo više opcija izgleda kao na „Slika 12. Prikaz zadatka sa spajanjem više opcija“



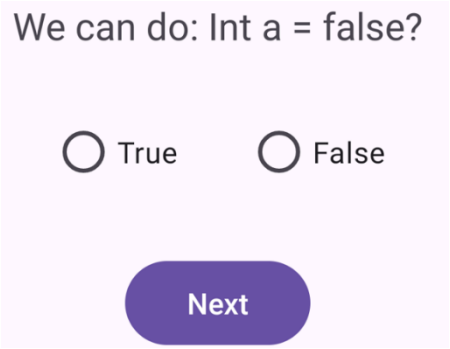
Slika 12. Prikaz zadatka sa spajanjem više opcija

Kada otvorimo ovakav zadatak „Slika 12. Prikaz zadatka sa spajanjem više opcija“ automatski se izabere prva opcija u listi. Lijeva strana je pitanje, a desna strana odgovor. Svaki odgovor na desnoj strani ima sve moguće opcije. To izgleda ovako:



Slika 13. Prikaz zadatka s spajanjem više opcija – opcije za prvo pitanje

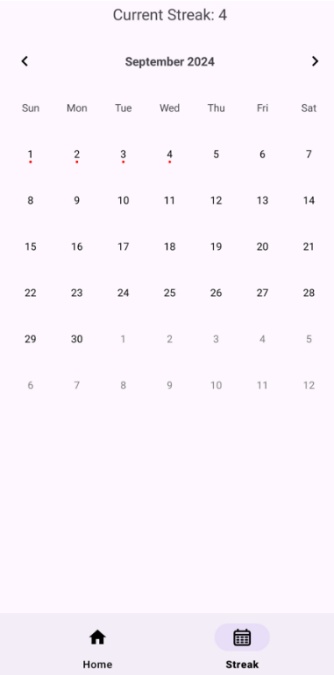
Kao što vidimo na „Slika 13. Prikaz zadatka s spajanjem više opcija – opcije za prvo pitanje“ prikazane su sve moguće opcije pod prvim odabirom, tj. pod odabirom „Int is a“. Isto tako je i za sva druga pitanja. I zadnji tip zadatka je odabir da li je tvrdnja točna ili netočna. Taj tip zadatka izgleda ovako:



Slika 14. Prikaz zadatka odabira točne ili netočne tvrdnje

Kao što vidimo „Slika 14. Prikaz zadatka odabira točne ili netočne tvrdnje“ prikazuje pitanje i određujemo da li je tvrdnja točna ili netočna.

Sada kada smo prikazali sve zadatke i kako izgledaju, još nam je ostalo vidjeti kako izgleda niz dana i kalendar. Kao što ste mogli primijetiti na glavnom ekranu u donjem navigacijskom izborniku kod „Slika 5. Prikaz ekrana ulaska u aplikaciju“ i „Slika 10. Prikaz glavnog ekrana s najviše točnih stavki unutar nivoa“ prikazan je „Streak“. Ona izgleda ovako:



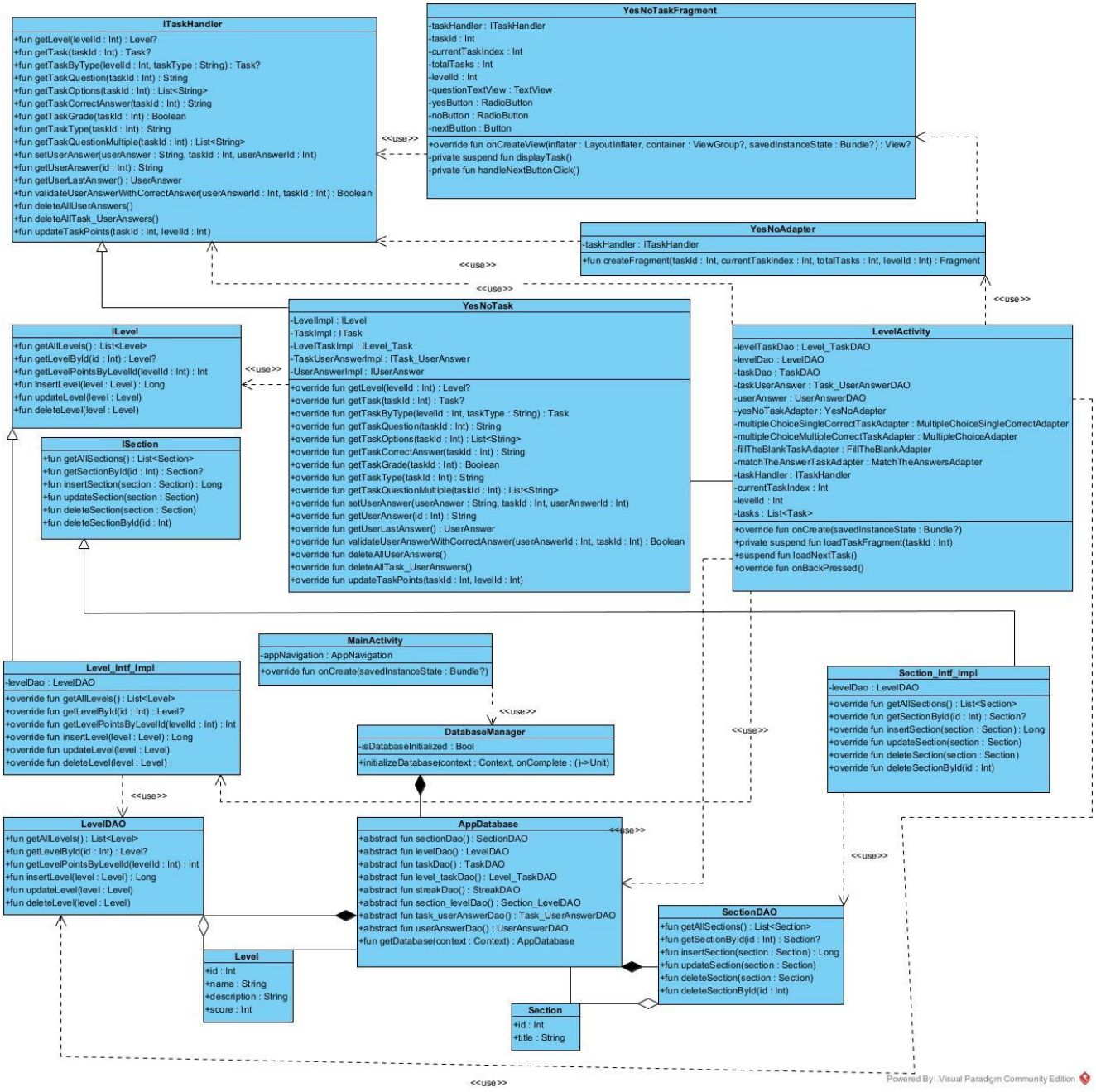
Slika 15. Prikaz kalendara i broja aktivnih dana za redom

Na „Slika 15. Prikaz kalendara i broja aktivnih dana za redom“ prikazan je broj aktivnih dana „Current Streak: 4“ u ovom slučaju to je 4. i prikaz kalendara unutar kojega su ti dani označeni (datumi od 01.09.2024 do 04.09.2024., uključujući i njih).

Sada su prikazani svi ekrani, od prikaza kada se uđe u aplikaciju, do toga kakvi su zadaci i kako izgledaju. I na kraju kako izgleda kalendar prikaza aktivnih dana. Sada prelazimo na dizajn aplikacije pa potom na programski kod i objašnjenje kako to funkcionira.

5.3. Dizajn aplikacije

Naša aplikacija izgleda preko prikaza dijagrama klasa kao na „Slika 16. Prikaz dizajna praktičnog djela rada preko dijagrama klasa“. Pokazat ćemo samo neke najbitnije dijelove preko dijagrama klase i to one dijelove koji koriste inverziju ovisnosti (i injekciju ovisnosti) kako



bi bolje predočili njene ovisnosti.

Slika 16. Prikaz dizajna praktičnog djela rada preko dijagrama klasa

Postoje entiteti (`Level` i `Section`). Entiteti su kao tablice u bazi. Nakon njih postoje sučelja DAO (`LevelDAO`, `SectionDAO`). Ta sučelja direktno su povezana s entitetima. Postoje sučelja `ILevel` i `ISection` koja nisu direktno povezana na bazu kao DAO sučelja. Implementacije `ILevel` i `ISection` nalaze se u `Level_intf_impl` i `Section_intf_impl`. Isto tako u tim implementacijama koriste se `LevelDAO` i `SectionDAO`. Sučelje `ITaskHandler` implementirano je od strane `YesNoTask` klase. Korištenje sučelja `ITaskHandler` nalazi se u `YesNoTaskFragment` klasi. Klasa `YesNoAdapter` je srednja klasa koja spaja sučelje s implementacijom (nju koristimo za injekciju ovisnosti). `YesNoAdapter` prima `ITaskHandler`. `YesNoAdapter` koristi se u klasi `LevelActivity`. `LevelActivity` je glavna klasa koja spaja implementacije zadataka (`YesNoTask`) i njihovo korištenje (`YesNoAdapter`). Kod tog poziva prosljeđuje joj se `Level_intf_impl` klasa (i ostale potrebne implementacije). Na ovaj način koristimo inverziju ovisnosti i injekciju ovisnosti preko konstruktora. Klasa `ILevel` koristi se unutar `YesNoTask` klase koja implementira `ITaskHandler`. Na taj način klasa `YesNoTask` koristi sučelje, a ne direktno implementaciju. Poziv klase `YesNoTask` kao što smo spomenuli klase je u klasi `LevelActivity`. Baza ima pristup entitetima i DAO sučeljima kako bi mogla pristupiti podacima u entitetima. Kreiranje baze radi se u `MainActivity` klasi.
S ovime smo obuhvatili najvažniji dio kako se koristi inverzija i injekcija ovisnosti u ovom projektu. I sada možemo prijeći na programski kod.

5.4. Programski kod

Sam programski kod aplikacije nije pre kompleksan. Pošto se u ovom radu fokusiramo na inverziju ovisnosti i injekciju ovisnosti, onda ćemo staviti fokus programskog koda na te dijelove i gdje se sve koriste ti koncepti. Ovi koncepti korišteni su ručno. Kao što smo objasnili u poglavlju „4.3. Čemu služi, za što i kako se koristi injekcija ovisnosti“. Koristimo ručno makar smo ustanovili da je lakše koristiti biblioteku, pošto je ovo prvi susret s ovim konceptom, bitno je shvatiti temelje. Ukoliko bi koristili automatiziranu injekciju ovisnosti ona bi za nas radila dosta stvari i ne bi se dobila šira slika korištenja ovog koncepta. Pomoći biblioteke je dobro koristiti kada dobro znamo kako funkcionira bez nje. I isto tako kada smo to isprobali na nekoj aplikaciji bez biblioteke.

Sada kada smo to razjasnili krećemo na programski kod. Kao što smo naveli, najveći fokus će biti na tome gdje se koriste principi inverzije i injekcije ovisnosti. Ali isto tako moramo obuhvatiti neke stvari izvan toga. Cijelo programsko rješenje nalazi se na linku: https://github.com/rikjalzabet/Dependancy-inversion-mobile-app.

Prva glavna klasa kada kreiramo projekt je **MainActivity**. Pošto je ona jedna od glavnih razmatrati ćemo je i objasniti što radi. Kao prvo programski kod te klase izgleda ovako:

```
1 class MainActivity : AppCompatActivity() {
2     private lateinit var appNavigation: AppNavigation
3
4     override fun onCreate(savedInstanceState: Bundle?) {
5         ...
6
7         val appNavigationImpl = AppNavigation_Intf_Impl(
8             this,
9             listOf(
10                ListSectionFragment(),
11                StreakFragment()
12            )
13        )
14        appNavigation = (applicationContext as ApplicationMain)
15                        .getServiceLocator(appNavigationImpl)
16                        .provideNavigator()
17        ...
18
19        bottomNavigationBar
20                .setOnItemSelectedListener{ menuItem ->
21            when (menuItem.itemId) {
22                R.id.nav_home -> {
23                    appNavigation.navigateToHome()
24                }
25                R.id.nav_streak -> {
26                    appNavigation.navigateToStreak()
27                }
28            }
29            true
30        }
31
32        DatabaseManager.initializeDatabase(this) {
33            appNavigation.navigateToHome()
34        }
35    }
36 }
```

Programski kod 33. Prikaz MainActivity

Imamo varijablu sučelja navigacije aplikacije. To sučelje u sebi sadrži dvije metode a to su **navigateToHome()** i **navigateToStreak()**. Nakon toga kreiramo varijablu navigacijskog izbornika i dodjeljujemo mu njegov element unutar xml datoteke. Na liniji 7 kod programskog koda „Programski kod 33. Prikaz **MainActivity**“ kreiramo instancu naše implementacije `AppNavigation` sučelja. Dodjeljujemo joj kontekst (this) i listu izbornika za navigaciju (`listOf(ListSectionFragment(),StreakFragment())`). Nakon toga spajamo našu implementaciju i korištenje sučelja s klasom **ApplicationMain**. Njoj prosljeđujemo implementaciju sučelja pomoću konstruktora i pozivamo metodu iz klase **ServiceLocator** zvanu **provideNavigator()**. Prije nego što nastavimo dalje, trebamo objasniti što ove

navedene klase rade kako bi bilo lakše shvatiti kod. Za sučelje `AppNavigation` smo već rekli, a njena implementacija izgleda ovako:

```
class AppNavigation_Intf_Impl(
    private val activity: AppCompatActivity,
    private val fragmentList: List<Fragment>
):AppNavigation {
    override fun navigateToHome() {
        val fragment=fragmentList[0]
        activityCall(fragment)
    }

    override fun navigateToStreak() {
        val fragment=fragmentList[1]
        activityCall(fragment)
    }

    private fun activityCall(fragment: Fragment){
        activity
        .supportFragmentManager
        .beginTransaction()
        .replace(
               hr.foi.final_thesis
               .coderepeat.R.id
           .activity_main_FL_main_container,
               fragment
               )
        .commit()
    }
}
```

Programski kod 34. Prikaz implementacije sučelja AppNavigation. Klasa AppNavigation_Intf_Impl

Klasa **`AppNavigation_Intf_Impl`** radi na način da prima preko konstruktora `AppCompatActivity` i listu fragmenata. Implementira **`navigateToHome`** na način da u varijablu fragment ubaci prvi element (tj. nulti) i proslijedi funkciji **`activityCall`**. To isto radi i **`navigateToStreak`** samo ona uzima drugi element (tj. prvi). Funkcija **`activityCall`** uzima fragment i prikazuje proslijeđenu aktivnost tj. stavi ju unutar FrameLayout-a. Klasa **`ApplicationMain`** ima funkciju dohvaćanja trenutnog servisa. Njen kod izgleda:

```
class ApplicationMain : Application() {
    lateinit var serviceLocator: ServiceLocator

    override fun onCreate() {
        super.onCreate()
    }
    fun getServiceLocator(
                appNavigation: AppNavigation
        ): ServiceLocator {
        serviceLocator = ServiceLocator(this, appNavigation)
        return serviceLocator
    }
```

```
13 }
```

Programski kod 35. Prikaz klase ApplicationMain

Funkcija **getServiceLocator** kod „Programski kod 35. Prikaz klase **ApplicationMain**“ prima sučelje `AppNavigation` i vraća **ServiceLocator**. Kreira novu instancu `ServiceLocator`-a i daje joj kontekst i sučelje `appNavigation`. I na kraju vraća tu novo kreiranu instancu putem `serviceLocator` varijable.

ServiceLocator ima jednu funkciju **provideNavigator** () koja vraća `appNavigation`. Ona izgleda ovako:

```
1 class ServiceLocator(
2         private val context: Context,
3         private val appNavigation: AppNavigation
4 ) {
5     fun provideNavigator(): AppNavigation {
6         return appNavigation
7     }
8 }
```

Programski kod 36. Prikaz klase ServiceLocator

Razlog korištenja klasa **ServiceLocator** i **ApplicationMain** makar možemo uspostaviti inverziju ovisnosti i injekciju ovisnosti bez njih je zbog korištenja **ServiceLocator**, a om je glavna klasa za upravljanje i davanjem servisa ovisnostima za sve velike aplikacije. Dok klasu **ApplicationMain** koristimo za spremanje i dijeljenje resursa preko cijelog programa. Ukoliko će ikada trebati koristiti neke stavke kroz cijelu aplikaciju, ovoj klasi svi imaju pristup i točno se zna njena funkcija. Pa razlog korištenja je više taj da su ove klase namijenjene za veće aplikacije i dodavanje više stavki.

Sada kada znamo što klase **ServiceLocator**, **ApplicationMain** i implementacija sučelja rade i kako izgledaju, možemo se vratiti na **MainActivity**. Kod „Programski kod 33. Prikaz **MainActivity**“ dalje imamo logiku kada kliknemo na određeni gumb u donjem navigacijskom izborniku, ta se funkcija pozove i njen xml se prikaže. I na kraju te klase imamo kreiranje baze. Ona poziva objekt `DatabaseManager`, a on izgleda ovako:

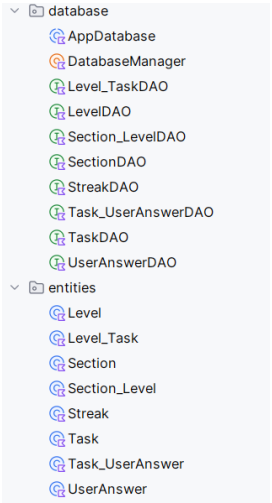
```
 1 object DatabaseManager {
 2     private var isDatabaseInitialized = false
 3     fun initializeDatabase(context: Context, onComplete: () -> Unit) {
 4         if (isDatabaseInitialized) {
 5             onComplete()
 6             return
 7         }
 8         CoroutineScope(Dispatchers.IO).launch {
 9             val db = AppDatabase.getDatabase(context)
10             val levelDao = db.levelDao()
11             val taskDao = db.taskDao()
12             val sectionDao = db.sectionDao()
```

```
13              ...
14              val levels = levelDao.getAllLevels()
15              val tasks = taskDao.getAllTasks()
16              val sections = sectionDao.getAllSections()
17
18              if (levels.isEmpty() && tasks.isEmpty()
19                                   && sections.isEmpty()) {
20                  populateData(context)
21              }
22              isDatabaseInitialized = true
23              withContext(Dispatchers.Main) {
24                  onComplete()
25              }
26          }
27      }
28  }
```

Programski kod 37. Prikaz objekta DatabaseManager

Objekt `DatabaseManager` ima varijablu za provjeru da li je već baza kreirana, u funkciji **`initializeDatabase`** provjerava s **`if`**-om ukoliko je već baza kreirana, vrati se van. Nadalje dohvaća podatke iz baze za tablice nivo, zadatak i sekciju. Ukoliko su te tablice prazne poziva se pomoćna funkcija koja stvara i sprema podatke u tablice u bazu. Nakon toga ažurira korisničko sučelje s `onComplete()`.

Ovime smo pojasnili što se događa u glavnoj klasi. Sada prebacujemo fokus na inverziju i injekciju ovisnosti. Koriste se kod baze i kod zadataka unutar nivoa. Pošto se baza kreira prva, nju pogledamo prvo. U paketu database i entities nalazi se sve potrebno za bazu. U paketu entities su tablice za baze podataka, a u paketu database su sučelja sa funkcijama koja direktno pristupaju bazi. Isto tako u paketu database nalazi se prethodno navedeni objekt `DatabaseManager` i klasa **`AppDatabase`**. To izgleda ovako:



Slika 17. Prikaz paketa database i entities

Kao primjer kako izgleda neki entitet i DAO sučelje uzet ćemo **Level** i LevelDAO. **Level** izgleda ovako:

```
@Entity(tableName = "Level")
data class Level(
    @PrimaryKey(autoGenerate = true) val id: Int=0,
    val name: String,
    val description: String,
    var score: Int = 0
)
```

Programski kod 38. Prikaz entiteta Level

Svaki entitet unutar sebe ima razne stavke. Kod levela to su `id`, `name`, `description` i `score`. `Name` se koristi kada korisnik otvori aplikaciju i prikazan mu je Level 1 – naziv. Dok `score` je isto u imenu, ukoliko je on veći od 0, biti će prikazan. Sučelje `LevelDAO` u sebi sadrži funkcije koje direktno pristupaju na bazu, a to izgleda ovako:

```
@Dao
interface LevelDAO {
    @Query("SELECT * FROM Level")
    fun getAllLevels():List<Level>
    @Query("SELECT * FROM Level WHERE id = :id")
    fun getLevelById(id: Int): Level?
    @Query("SELECT score FROM Level WHERE id = :levelId")
    fun getLevelPointsByLevelId(levelId: Int): Int
    @Insert
    fun insertLevel(level: Level): Long
    @Update
    fun updateLevel(level: Level)
    @Delete
    fun deleteLevel(level: Level)
}
```

Programski kod 39. Prikaz sučelja LevelDAO

Osigurali smo da je baza lako promjenjiva ako imamo sql upite unutar našeg sučelja na način da kreiramo jedno sučelje zvano `ILevel` koje ima iste metode, samo bez sql upita. Na taj način imamo odvojenost baze od ostatka programa u većini slučajeva. Kod implementacije `ILevel` koristimo `LevelDAO` kako bi ga proslijedili konstruktoru i tako koristili koncept inverzije i injekcije ovisnosti. Ta implementacija `ILevel` -a izgleda ovako:

```
class Level_Intf_Impl(private val levelDao: LevelDAO) : ILevel {
    override fun getAllLevels(): List<Level> {
        return levelDao.getAllLevels()
    }
    override fun getLevelById(id: Int): Level? {
        return levelDao.getLevelById(id)
    }
    override fun getLevelPointsByLevelId(levelId: Int): Int {
        return levelDao.getLevelPointsByLevelId(levelId)
    }
```

```
11      override fun insertLevel(level: Level): Long {
12          return levelDao.insertLevel(level)
13      }
14      override fun updateLevel(level: Level) {
15          levelDao.updateLevel(level)
16      }
17      override fun deleteLevel(level: Level) {
18          levelDao.deleteLevel(level)
19      }
20  }
```

Programski kod 40. Prikaz implementacije sučelja ILevel

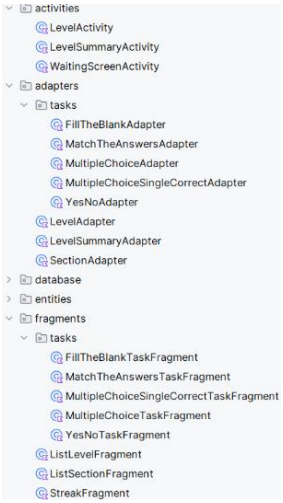
Vratimo se na paket database. Unutar njega imamo apstraktnu klasu **AppDatabase**. Ona nasljeđuje `RoomDatabase()`. RoomDatabase je biblioteka za android studio koja omogućava korištenje lokalne baze. Unutar te klase kreiramo funkcije za svaku DAO klasu koje nam daju pristup bazi. Ta klasa izgleda ovako:

```
 1 @Database(entities = [
 2         Level::class,
 3         Task::class,
 4         Streak::class,
 5         Section::class,
 6         Level_Task::class,
 7         Section_Level::class,
 8         Task_UserAnswer::class,
 9         UserAnswer::class
10     ],
11     version = 1)
12 abstract class AppDatabase : RoomDatabase() {
13     abstract fun sectionDao(): SectionDAO
14     abstract fun levelDao(): LevelDAO
15     abstract fun taskDao(): TaskDAO
16     abstract fun level_taskDao(): Level_TaskDAO
17     abstract fun streakDao(): StreakDAO
18     abstract fun section_levelDao(): Section_LevelDAO
19     abstract fun task_userAnswerDao(): Task_UserAnswerDAO
20     abstract fun userAnswerDao(): UserAnswerDAO
21
22     companion object{
23         @Volatile
24         private var INSTANCE: AppDatabase?=null
25
26         fun getDatabase(context: Context): AppDatabase{
27             val tempInstance = INSTANCE
28             if (tempInstance != null) {
29                 return tempInstance
30             }
31             synchronized(this) {
32                 val instance = Room.databaseBuilder(
33                     context.applicationContext,
34                     AppDatabase::class.java,
35                     "app_database"
36                 )
37                     .fallbackToDestructiveMigration()
```

```
38                      .build()
39                  INSTANCE = instance
40                  return instance
41              }
42          }
43      }
44 }
```

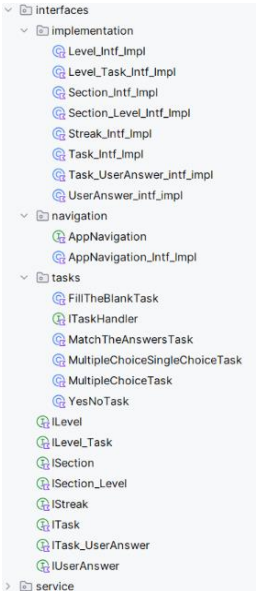
Programski kod 41. Prikaz klase AppDatabase

Ona spaja naše entitete i to joj govori da su ti entiteti zapravo tablice u bazi. `getDatabase` kreira jednu instancu baze koja osigurava da nemamo dvije ili više baza u jednom trenu. Baza se kreira u kontekstu aplikacije s imenom baze `"app_database"`. Zapravo funkcija ove klase je kreiranje, pristup i upravljanje bazom. Sada kada znamo osnovne komponente baze slijede zadaci. Sve vezano za zadatke je cijeli paket activities, adapters, fragments i kod paketa interfaces unutar njega paket tasks. Ti paketi izgledaju ovako:



Slika 18. Prikaz paketa activities, adapters i fragments

Isto paket interfaces izgleda ovako:



Slika 19. Prikaz paketa interfaces

Isto tako ostale stavke kao što je unutar paketa interfaces nalaze se sva sučelja i unutar pod paketa implementation u paketu interfaces sve implementacije sučelja i sama baza i njene stavke. Krećemo od osnova, a to su sučelja. Pošto smo većinu sučelja razgledali još nam je ostalo `ITaskHandler` sučelje. Ono sadrži sve metode koje su nam potrebne za naše zadatke. Njene implementacije su vrlo slične za skoro svaki zadatak. Pogledat ćemo implementacije **`YesNoTask`** i **`MatchTheAnswersTask`**. Obje klase, a i ostale primaju `ILevel`, `ITask`, `ILevel_Task`, `ITask_UserAnswer` i `IUserAnswer` u svoj konstruktor. Za implementacije klase vrijedi napomenuti da je **`getTaskType`** kod svih različit. Isto tako **`getTaskOptions`** i **`getTaskQuestionMultiple`** se jedino koristi kod **`MatchTheAnswersTask`**. Klasa **`YesNoTask`** izgleda ovako:

```
1 class YesNoTask(
2     private val LevelImpl: ILevel,
3     private val TaskImpl: ITask,
4     private val LevelTaskImpl: ILevel_Task,
5     private val TaskUserAnswerImpl: ITask_UserAnswer,
6     private val UserAnswerImpl: IUserAnswer
7 ) : ITaskHandler {
8
9         override fun getTaskType(taskId: Int): String {
10         return "YES_NO"
11     }
12         override fun getTaskOptions(
13                 taskId: Int): List<String> {
14         return listOf("True", "False")
15     }
16         override fun getTaskQuestion(
17                 taskId: Int): String {
18         return TaskImpl
```

```
19                .getTaskById(taskId)?.question ?: ""
20      }
21          override fun getTaskQuestionMultiple(
22                  taskId: Int): List<String> {
23          return emptyList()
24      }
25
26
27
28      override fun getLevel...
29      override fun getTask...
30          override fun getTaskGrade...
31          override fun deleteAllTask_UserAnswers()...
32          override fun deleteAllUserAnswers()...
33      override fun getTaskByType...
34
35      override fun getTaskCorrectAnswer(taskId: Int): String {
36          return TaskImpl
37                  .getTaskById(taskId)?.correctAnswer.toString()
38      }
39
40      override fun setUserAnswer(
41          userAnswer: String, taskId: Int, userAnswerId: Int) {
42          UserAnswerImpl
43                  .insertUserAnswers(UserAnswer(
44                          userAnswer=userAnswer, userMultipleAnswer = "")
45                  )
46          val getNewAddedUserAnswer=
47                  UserAnswerImpl
48                  .getLatestInsertedUserAnswer().id
49          val connectTaskUserAnswer =
50                  TaskUserAnswerImpl
51                          .insertTask_UserAnswer(
52                          Task_UserAnswer(
53                          taskId=taskId, answerId=(
54                                  getNewAddedUserAnswer.toInt())
55                          ))
56      }
57
58      override fun getUserAnswer(id: Int): String {
59          return UserAnswerImpl
60                  .getUserAnswerById(id)?.userAnswer.toString()
61      }
62
63      override fun getUserLastAnswer(): UserAnswer {
64          return UserAnswerImpl.getLatestInsertedUserAnswer()
65      }
66
67      override fun validateUserAnswerWithCorrectAnswer(
68          userAnswerId: Int, taskId: Int): Boolean {
69          val userAnswer =
70                  UserAnswerImpl.getUserAnswerById(userAnswerId)
71          val task = TaskImpl.getTaskById(taskId)
72          Log.i("YesNoTask","User answer:
73                  ${userAnswer?.userAnswer} Task answer:
74                  ${task?.correctAnswer}")
75          return userAnswer?.userAnswer?.toString()
```

```
76                  == task?.correctAnswer.toString()
77      }
78
79      override fun updateTaskPoints(taskId: Int, levelId: Int) {
80          val task = TaskImpl.getTaskById(taskId)
81          val level = LevelImpl.getLevelById(levelId)
82          if (task != null && level != null) {
83          LevelTaskImpl
84                  .updateLevel_Task(Level_Task(
85                  levelId =
86                  level.id, taskId = task.id, points = 1.0
87                  ))
88          }
89      }
90 }
```

Programski kod 42. prikaz klase YesNoTask

Dok prikaz klase **MatchTheAnswersTask** izgleda ovako:

```
 1 class MatchTheAnswersTask(
 2     private val LevelImpl: ILevel,
 3     private val TaskImpl: ITask,
 4     private val LevelTaskImpl: ILevel_Task,
 5     private val TaskUserAnswerImpl: ITask_UserAnswer,
 6     private val UserAnswerImpl: IUserAnswer
 7 ): ITaskHandler{
 8         override fun getLevel...
 9     override fun getTask...
10         override fun getTaskGrade...
11         override fun deleteAllTask_UserAnswers()...
12         override fun deleteAllUserAnswers()...
13     override fun getTaskByType...
14
15     override fun getTaskOptions(
16         taskId: Int): List<String> {
17         val task=getTask(taskId)
18         return task?.options?
19                 .split("|##|") ?: emptyList()
20     }
21
22     override fun getTaskCorrectAnswer(
23         taskId: Int): String {
24         return TaskImpl.getTaskById(taskId)
25                 ?.correctAnswer ?: ""
26     }
27
28
29     override fun getTaskType(
30         taskId: Int): String {
31         return "MATCH_THE_ANSWERS"
32     }
33
34     override fun getTaskQuestionMultiple(
35         taskId: Int): List<String> {
36         val task=getTask(taskId)
```

```
        return task?.question?.split("|##|")
                ?: emptyList()
    }

    override fun setUserAnswer(
        userAnswer: String,
        taskId: Int,
        userAnswerId: Int) {
        UserAnswerImpl.
                insertUserAnswers(UserAnswer(userAnswer =
                userAnswer, userMultipleAnswer = ""))
        val newUserAnswerId =
                UserAnswerImpl.getLatestInsertedUserAnswer().id
        TaskUserAnswerImpl.insertTask_UserAnswer(
                Task_UserAnswer(
                taskId = taskId, answerId =
                newUserAnswerId.toInt())
                )
    }

    override fun getUserAnswer(id: Int): String {
        return UserAnswerImpl
                .getUserAnswerById(id)?.userAnswer ?: ""
    }

    override fun getUserLastAnswer(): UserAnswer {
        return UserAnswerImpl
                .getLatestInsertedUserAnswer()
    }

    override fun validateUserAnswerWithCorrectAnswer(
        userAnswerId: Int, taskId: Int): Boolean {
        val userAnswer = UserAnswerImpl.
                getUserAnswerById(userAnswerId)?.userAnswer ?: ""
        val correctAnswer = TaskImpl.
                getTaskById(taskId)?.correctAnswer ?: ""
        Log.i("TaskGameInfoIntf",
                "MATCH_THE_ANSWERS - User Answer: $userAnswer")
        Log.i("TaskGameInfoIntf",
                "MATCH_THE_ANSWERS - Correct Answer: $correctAnswer")
        return userAnswer.split("|##|").sorted()
                == correctAnswer.split("|##|").sorted()
    }

    override fun getTaskQuestion(taskId: Int): String {
        return ""
    }

    override fun updateTaskPoints(taskId: Int, levelId: Int) {
        val task = TaskImpl.getTaskById(taskId)
        val level = LevelImpl.getLevelById(levelId)
        if (task != null && level != null) {
            LevelTaskImpl.updateLevel_Task(
                    Level_Task(
                    levelId = level.id, taskId = task.id,
                    points = 1.0)
                    )
```

```
94          }
95      }
96  }
```

Programski kod 43. Prikaz klase MatchTheAnswersTask

Kao što možemo vidjeti **TaskType** je drugačiji, **taskOptions** je isto tako drugačiji, jedan vraća true/false, dok drugi razdvaja string i stavlja ga u listu. **askQuestion** je kod **MatchTheAnswersTask** korišten za **getTaskQuestionMultiple** i vraća listu string zadataka. Dok **YesNoTask** vraća samo string odgovora iz baze. Kod **setUserAnswer** jedna vrijednost stavlja se na `userAnswer`, a druga na `userMultipleAnswer`. Isto tako validacija odgovora. **YesNoTask** potvrđuje na način da usporedi direktno odgovor od korisnika i zadatka, dok **MatchTheAnswersTask** još i odvaja i sortira odgovore. Kao što možemo primijetiti implementacije su vrlo slične.

Sada kada znamo sučelje i njene implementacije možemo krenuti na fragmente. Prvo ćemo obratiti pažnju na **ListLevelFragment** i **ListSectionFragment** pošto su one prva stvar koju vidimo kada otvorimo aplikaciju. Započinjemo s **ListSectionFragment**. Kod ove klase dohvaćamo elemente, kreiramo bazu i koristimo injekciju ovisnosti tako da damo implementacijama njihove DAO klase, dohvaćamo sekcije iz baze i šaljemo ih adapteru. Adapter prikazuje svaku stavku na svoju poziciju i dodjeljuje joj title iz baze kao naslov. Nakon toga **SectionAdapter** poziva **LevelAdapter** -a. **ListSectionFragment** izgleda ovako:

```
1  class ListSectionFragment : Fragment() {
2      ...
3
4      override fun onViewCreated(
5          view: View, savedInstanceState: Bundle?) {
6          ...
7          recyclerView
8                  = view.findViewById(
9                  R.id.fragment_list_section_rv_listOfSections)
10         progressBar
11                 = view.findViewById(
12                 R.id.fragment_list_section_pb_loading)
13         recyclerView.layoutManager = LinearLayoutManager(context)
14
15         val db = AppDatabase.getDatabase(requireContext())
16         levelTaskDAO = db.level_taskDao()
17         sectionDao = db.sectionDao()
18         sectionLevelDAO = db.section_levelDao()
19
20         section = Section_Intf_Impl(sectionDao)
21         sectionLevel = Section_Level_Intf_Impl(sectionLevelDAO)
22         levelTask= Level_Task_Intf_Impl(levelTaskDAO)
23
24         CoroutineScope(Dispatchers.IO).launch {
25             val sections = section.getAllSections()
26             Log.i("ListSectionFragment", "Fetched Sections: $sections")
```

```
            withContext(Dispatchers.Main) {
            val adapter =
                      SectionAdapter(
                      sections, sectionLevel, levelTask
                      ) { level ->
            val intent = Intent(context, LevelActivity::class.java)
                intent.putExtra("LEVEL_ID", level.id)
                startActivity(intent)
                }
                recyclerView.adapter = adapter
            }
        }
    }
}
```

Programski kod 44. Prikaz ListSectionFragment klase

Prikaz njegovog adaptera:

```
class SectionAdapter(
    private val sections: List<Section>,
    private val sectionLevel: ISection_Level,
    private val levelTask: ILevel_Task,
    private val onLevelClick: (Level) -> Unit
) : RecyclerView.Adapter
<SectionAdapter.SectionViewHolder>() {
    inner class SectionViewHolder(
        itemView: View) : RecyclerView.ViewHolder(
        itemView) {
        ...
         init{
         levelRecycleView.layoutManager=
                LinearLayoutManager(itemView.context)
         }
    }
    ...
    override fun onBindViewHolder(
        holder: SectionViewHolder, position: Int) {
         val section = sections[position]
         holder.sectionTitle.text = section.title

         CoroutineScope(Dispatchers.IO).launch {
             val levels = sectionLevel
                       .getLevelsForSection(section.id)
             withContext(Dispatchers.Main) {
                 val levelAdapter
                              =LevelAdapter(levels,levelTask,onLevelClick)
                 holder.levelRecycleView.adapter=levelAdapter
             }
         }
    }
    override fun getItemCount(): Int = sections.size
}
```

Programski kod 45. Prikaz SectionAdapter klase

`LevelAdapter` dohvaća sve potrebne podatke o nivou i prikazuje svaki nivo posebno. Ovo je zapravo lista u listi. Sekcija je lista koja u sebi sadrži listu nivoa. `LevelAdapter` izgleda ovako:

```
class LevelAdapter(
    private val levels: List<Level>,
    private val levelTask: ILevel_Task,
    private val onLevelClick: (Level) -> Unit
) : RecyclerView.Adapter<LevelAdapter.LevelViewHolder>() {
    inner class LevelViewHolder(itemView: View)
        : RecyclerView.ViewHolder(itemView) {
        ...
        init {
            itemView.setOnClickListener {
                val level = levels[adapterPosition]
                onLevelClick(level)
            }
            levelLayout.setOnClickListener {
                val level = levels[adapterPosition]
                onLevelClick(level)
            }
        }
    }
        ...

        override fun onBindViewHolder(
                holder: LevelViewHolder, position: Int) {
            val level=levels[position]
            val lvlNmbCounter=position+1
            holder.levelName.text=level.name
            holder.levelNumber.text="Level $lvlNmbCounter"

            if(level.score > 0){
                holder.levelPoints
                                .visibility=View.VISIBLE
                holder.levelPoints
                                .text=" (best score: ${level.score})"
            }else{
                holder.levelPoints
                                .visibility=View.GONE
            }
        }
        override fun getItemCount(): Int = levels.size
}
```

Programski kod 46. Prikaz klase LevelAdapter

Adapteri dohvaćaju potrebne elemente u xml kodu i prikazuju potrebne podatke na ekran svake stavke. `ItemCount` mu govori koliko stavki ima i koliko ih treba postaviti. Isto tako `LevelAdapter` sadrži u sebi `setOnClickListener` koji dohvaća kada korisnik pritisne na određeni nivo tako da se može pravilno učitati. Kako bi se logika zadatka prikazala koristimo klasu `LevelActivity`. Ona izgleda ovako:

```
class LevelActivity : AppCompatActivity() {
    ...

    private var currentTaskIndex: Int = 0
    private var levelId: Int = 0
    private var tasks: List<Task>? = null

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

        levelId = intent.getIntExtra("LEVEL_ID", -1)
        if (levelId == -1) {
            return
        }
        val db = AppDatabase.getDatabase(this)
        levelDao = db.levelDao()
        levelTaskDao = db.level_taskDao()
        ...

        CoroutineScope(Dispatchers.IO).launch {
            val tasks = levelTaskDao.getTasksForLevel(levelId)
            val level = levelDao.getLevelById(levelId)
        }

        yesNoTaskAdapter = YesNoAdapter(YesNoTask(
            Level_Intf_Impl(levelDao),
            Task_Intf_Impl(taskDao),
            Level_Task_Intf_Impl(levelTaskDao),
            Task_UserAnswer_intf_impl(taskUserAnswer),
            UserAnswer_intf_impl(userAnswer)
        ))
        ...

        CoroutineScope(Dispatchers.IO).launch {
            tasks = levelTaskDao.getTasksForLevel(levelId)
            val level = levelDao.getLevelById(levelId)
            withContext(Dispatchers.Main) {
                if (!tasks.isNullOrEmpty()) {
                loadTaskFragment(tasks!!
                                [currentTaskIndex].id)
                }
            }
        }
    }
    private suspend fun loadTaskFragment(taskId: Int) {
        val task = tasks?.find { it.id == taskId }
        val fragment = when (task?.type) {
        "YES_NO" -> yesNoTaskAdapter
                .createFragment(
                taskId, currentTaskIndex,
                tasks!!.size, levelId)
        ...
        else -> {
                        loadNextTask()
                        return
        }}
        supportFragmentManager.beginTransaction()
```

```
58          .replace(
59                  R.id.activity_level_RL_main_container, fragment
60                  )
61          .commit()
62      }
63
64      suspend fun loadNextTask() {
65          currentTaskIndex++
66          if (currentTaskIndex < tasks?.size ?: 0) {
67              loadTaskFragment(tasks!![currentTaskIndex].id)
68          } else {
69              withContext(Dispatchers.Main) {
70              Toast.makeText(this@LevelActivity,
71                          "All tasks completed!", Toast.LENGTH_SHORT).show()
72              }
73              finish()
74          }
75      }
76      override fun onBackPressed() {
77          AlertDialog.Builder(this)
78          .setTitle("Exit Level")
79          .setMessage(
80                  "Do you want to exit the current level?
81                  By exiting now your current progress will NOT be saved!")
82          .setPositiveButton("Yes") { dialog, which ->
83          CoroutineScope(Dispatchers.IO).launch {
84              taskHandler.deleteAllUserAnswers()
85              taskHandler.deleteAllTask_UserAnswers()}
86          super.onBackPressed()
87          }
88          .setNegativeButton("No", null)
89          .show()
90      }
91  }
```

Programski kod 47. Prikaz klase LevelActivity

`LevelActivity` klasa u sebi sadrži najviše inverzije i injekcije ovisnosti. Kreira `taskAdapter` za svaki zadatak, dodjeljuje mu njegov prikladan adapter i implementaciju. Isto tako u konstruktoru implementacije svakoj implementaciji dodjeljuje DAO klasu za potrebnu implementaciju. Još jedna bitna stvar je funkcija `loadTaskFragment`, koja provjerava koji je tip taj zadatak i prema tome proslijedi odgovarajuću xml datoteku kako bi se zadatak pravilno prikazao. Funkcija `loadNextTask` dohvaća idući zadatak koji je unutar nivoa. Isto tako implementirano je `onBackPressed` ukoliko korisnik slučajno ili namjerno klikne na back tipku na uređaju da mu prikaže pitanje jeste li sigurni da želite odustati od nivoa. Kao što smo rekli unutar `LevelActivity` nalazi se poziv adaptera za zadatke. Sada ćemo pogledati kako izgleda jedan takav adapter i njegov fragment. `YesNoAdapter` u konstruktoru ima `ITaskHandler` i funkciju `createFragment`. Funkcija `createFragment` prima `taskId`,

`levelId`, broj ukupno nivoa u levelu i trenutni nivo indeks. On vraća **`YesNoTaskFragment`**.

`YesNoTaskFragment` izgleda ovako:

```
class YesNoTaskFragment(
    private val taskHandler: ITaskHandler,
    private var taskId: Int,
    private val currentTaskIndex: Int,
    private val totalTasks: Int,
    private val levelId: Int
) : Fragment() {
        ...
        override fun onCreateView(
            inflater: LayoutInflater, container: ViewGroup?,
            savedInstanceState: Bundle?
        ): View? {
            val view = ...
            questionTextView = ...
            yesButton = ...
            noButton = ...
            nextButton = ...

            taskId = arguments?.getInt("TASK_ID") ?: -1

            if (taskId != -1) {
                CoroutineScope(Dispatchers.IO).launch { displayTask() }
            }
                        ...

            nextButton.setOnClickListener {
                CoroutineScope(Dispatchers.IO).launch {
                    handleNextButtonClick()
                }
            }
            return view
        }

        private suspend fun displayTask() {
            CoroutineScope(Dispatchers.IO).launch  {
                val taskQuestion=
                                taskHandler.getTaskQuestion(taskId)
                questionTextView.text =taskQuestion
            }
        }

        private fun handleNextButtonClick() {
            val selectedAnswer = when {
                yesButton.isChecked -> "True"
                noButton.isChecked -> "False"
                else -> {
                activity?.runOnUiThread {
                Toast
                                .makeText(context,
                                "Please select an answer",
                                Toast.LENGTH_SHORT)
                .show()
                }
```

```
54                      return
55                  }
56              }
57
58              CoroutineScope(Dispatchers.IO).launch {
59                  taskHandler.setUserAnswer(selectedAnswer, taskId, 1)
60
61                  val userAnswer =
62                                  taskHandler.getUserLastAnswer()
63                  val isCorrect =
64                                  taskHandler
65                                  .validateUserAnswerWithCorrectAnswer
66                                  (userAnswer.id, taskId)
67                  if(isCorrect){
68                      taskHandler
69                                          .updateTaskPoints(taskId, levelId)
70                  }
71                  withContext(Dispatchers.IO) {
72                      if(currentTaskIndex==totalTasks-1){
73                      (activity as LevelActivity).loadNextTask()
74
75                      LevelSummaryActivity
76                                          .taskHandler = taskHandler
77                      LevelSummaryActivity
78                                          .levelId=levelId
79                      val intent = Intent(context,
80                                          LevelSummaryActivity::class.java)
81                      startActivity(intent)
82                      }else {
83                          (activity as LevelActivity).loadNextTask()
84                      }
85                  }
86              }
87          }
88 }
```

Programski kod 48. prikaz YesNoTaskFragment -a

`YesNoTaskFragment` služi kako bi ispravno prikazali stvari na zaslon korisniku. Dohvaća pitanje i sadrži logiku koja govori kojeg smo gumba pritisnuli. Ako nismo ništa odabrali i kliknemo na idući zadatak, nećemo moći na idući zadatak dokle god nismo odabrali (ili napisali) nešto. Isto tako gleda koji će idući zadatak biti poslije njega i poziva `LevelActivity`, ukoliko je on zadnji onda poziva `LevelSummaryActivity`.

5.5. Zaključak o praktičnoj primjeni inverzije ovisnosti

S ovime smo obuhvatili najvažnije stavke i ukratko objasnili bitne dijelove programskog koda. Time smo zaključili da se sučelja implementirana i sučelja se direktno koriste na mjestu gdje trebaju. Fragmenti uglavnom koriste sučelja, a njihovi adapteri im daju implementaciju. Na taj način je injekcija ovisnosti i inverzija ovisnosti zadovoljena. Baza je odvojena od ostatka

logine na način da postoje dva identična sučelja. Jedina je razlika da jedna nema sql upite, a druga koja je direktno povezana na bazu ima. Baza je lokalna, pa tako ovisi o uređaju, a ne o vanjskom serveru. Jedino o čemu ovisi je da biblioteka Room postoji i dalje bude podržavana.

6. Zaključak

Zaključak je da inverzija ovisnosti kao arhitekturalni konstrukt u dizajnu mobilnih aplikacija je kao osnovna struktura koja se koristi za ponovo upotrebljiv programski kod. Ona se sastoji od dva dijela, klase koriste sučelje kao temelj i klase koje ovise o sučelju. Na temelju svega obrađenog u ovom radu dolazim do zaključka da su razlozi za što se inverzija ovisnosti koristi: lakoća testiranja, ponovo iskoristive komponente, održavanje koda, prilagodba, kasnija implementacija i paralelno programiranje kod rada u timu. Iz primjera i praktičnog djela rada vidljivo je da klase koriste sučelje, a ne direktnu implementaciju kako bi programski kod mogao biti ponovo korišten. Ovaj način upotrijebili smo kod praktičnog dijela rada na naše klase, ponajviše na klase vezane uz zadatke. Na taj način naše klase su ponovo iskoristive i manje zapletene. No ovime nismo saznali kako se ovo zapravo koristi u pravom svijetu. Za to je poslužila lekcija injekcija ovisnosti.

Koncept injekcije ovisnosti je zapravo korištenje teorije inverzije ovisnosti u praksi. Njene primjene korištenja su preko konstruktora, metode i svojstva. Prednosti injekcije su prednosti inverzije ovisnosti. Automatsko korištenje injekcije ovisnosti je puno jednostavnije nego ručno korištenje. Usprkos tome kada krenemo s nekim konceptom bitno je znati njene temelje, pa iz tog razloga je korištena automatska injekcija u praktičnom djelu rada. Najveći naglasak stavljen je na injekciju preko konstruktora koju smo koristili u praktičnom djelu rada unutar glavne klase za zadatke koja spaja naše implementacije s implementacijom sučelja.

Prvenstveno smo naučili da je predznanje potrebno, a što znači za razumijevanje ovih koncepata trebamo biti neko vrijeme programeri i znati OOP principe (enkapsulacija, nasljeđivanje, polimorfizam i abstrakcija). Trebamo imati i neko praktično znanje kako bi bolje shvatili i savladali ove koncepte. Na drugom mjestu trebamo biti strpljivi, voljniji učiti i razvijati se, pošto ovi koncepti znaju zakomplicirati lake stvari. Pogotovo kao što smo vidjeli primjer za ispis na ekran, programski kod je puno veći nego što bi to bilo uobičajeno. Širina predznanja je poveća pa zato ljudi teže donose odluku za ulazak u ovako nešto.

Sa svime time naši ciljevi da saznamo što su ti koncepti, čemu oni služe, zašto bi baš njih koristili te kako se koristiti inverzijom u praksi su ispunjeni. Upoznali smo se s tim idejama, napravili funkcionalnu aplikaciju i naučili nešto novo.

Sav kod praktičnog primjera možete naći na: https://github.com/rikjalzabet/Dependancy-inversion-mobile-app

Popis literature

Mark Seemann (2011) Dependency Injection in .NET. Wiley India Pvt. Limited

Brad Larson (2009). *How to explain dependency injection to a 5-year-old?* [Stack Overflow], Pristupano 27.06.2024., s https://stackoverflow.com/questions/1638919/how-to-explain-dependency-injection-to-a-5-year-old

Thorben (2024). SOLID Design Principles Explained: Dependency Inversion Principle with Code Examples [Stackify], Pristupano 14.7.2024., s https://stackify.com/dependency-inversion-principle/

BuketSenturk (2024). *Dependency Inversion vs Dependency Injection* [Medium], Pristupano 06.7.2024., s https://medium.com/@buketsenturk/dependency-inversion-vs-dependency-injection-4a2dea766e6b

Mohammad Ramezani (2020). *The 3 Types of Dependency Injection* [LinkedIn], Pristupano 20.7.2024., s https://www.linkedin.com/pulse/3-types-dependency-injection-mohammad-ramezani/

Remya Mohanan (2024). *What Is Dependency Injection? Meaning, Types, Control, Use, and Examples* [Spiceworks], Pristupano 06.07.2024., s https://www.spiceworks.com/tech/devops/articles/what-is-dependency-injection/

Android Studio (2024). Dependency injection in Android [Developer Android], Pristupano 14.7.2024., s https://developer.android.com/training/dependency-injection

Manuel Vivo (2023). *Using Hilt in your Android app* [Developer Android], Pristupano 30.7.2024., s https://developer.android.com/codelabs/android-hilt#0

Domenique Tilleuil i Guido Dechamps (2019). *THE IMPORTANCE OF THE DEPENDENCY INVERSION PRINCIPLE* [TripleD], Pristupano 14.7.2024., s https://www.tripled.io/07/05/2019/dependency-inversion-principle/

Dependency inversion principle (bez dat.). U Wikipedia. Pristupano 14.7.2024. s https://en.wikipedia.org/wiki/Dependency_inversion_principle

Samuel Olusola (2024). *Understanding the dependency inversion principle in TypeScript* [LogRocket], Pristupano 14.7.2024., s https://blog.logrocket.com/dependency-inversion-principle-typescript/

Beribey (2019). *Dependency Injection And Inversion Of Control — Part 1: Definitions* [Medium], Pristupano 14.7.2024., s https://medium.com/coderes/dependency-injection-and-inversion-of-control-part-1-definitions-5c84d281e549

Thorben (2024). *Design Patterns Explained – Dependency Injection with Code Examples* [Stackify], Pristupano 20.7.2024., s https://stackify.com/dependency-injection/

Milica Dancuk (2023). *What Is Dependency Injection?* [phoenixNAP], Pristupano 17.7.2024., s https://phoenixnap.com/kb/dependency-injection

Jorge Nicolás Nogueiras (2020). *Make your Android application rock SOLID — Dependencies* [Medium], Pristupano 14.7.2024., s https://proandroiddev.com/make-your-android-application-rock-solid-dependencies-ea0af48e5481

Peter Mortensen (2017). *What is the dependency inversion principle and why is it important?* [Stack Overflow], Pristupano 06.07.2024., s https://stackoverflow.com/questions/62539/what-is-the-dependency-inversion-principle-and-why-is-it-important

Singhankit (2023). *Dependency Inversion Principle (SOLID)* [GeeksforGeek], Pristupano 14.7.2024., s https://www.geeksforgeeks.org/dependecy-inversion-principle-solid/

Tamerlan Gudabayev (2021). *Understanding SOLID Principles: Dependency Inversion* [Dev], Pristupano 14.7.2024., s https://dev.to/tamerlang/understanding-solid-principles-dependency-inversion-1b0f

Chaewonkong (2023). *Unraveling Dependency Injection: Advantages, Disadvantages, and Why We Need It* [Medium], Pristupano 14.7.2024., s https://medium.com/@chaewonkong/unraveling-dependency-injection-advantages-disadvantages-and-why-we-need-it-b1726eef041a

freeCodeCamp (2018). *A quick intro to Dependency Injection: what it is, and when to use it* [freeCodeCamp], Pristupano 14.7.2024., s https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/

Evan Emran (2023). *Demystifying Dependency Injection in Android App Development* [codingwithevan], Pristupano 17.7.2024., s https://codingwithevan.com/android_dependency_injection/

Dependency injection (bez dat.). U Wikipedia. Pristupano 14.7.2024. s https://en.wikipedia.org/wiki/Dependency_injection

Deepakanduri (2024). *Spring Dependency Injection with Example* [geeksforgeeks], Pristupano 20.7.2024., s https://www.geeksforgeeks.org/spring-dependency-injection-with-example/

tarunsh648 (2024). *Dependency Injection vs Factory Pattern* [geeksforgeeks], Pristupano 24.7.2024., s https://www.geeksforgeeks.org/dependency-injection-vs-factory-pattern/

Techbalachandar (2023). *Dependency Injection (DI) in Android* [Medium], Pristupano 20.7.2024., s https://medium.com/@techbalachandar/dependency-injection-in-android-56c96441d942

Popis slika

Popis tablica

Popis programskog koda

Prilozi

1. Programski kod nalazi se na: https://github.com/rikjalzabet/Dependancy-inversion-mobile-app