Pomen uporabe arhitekturnih načrtovalskih vzorcev pri razvoju mobilnih aplikacij

Luka Pavlič, Luka Četina
Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Koroška cesta 46, 2000 Maribor
luka.pavlic@um.si, luka.cetina1@um.si

Izvleček

Izbira ustrezne arhitekture je pomemben in nujen korak razvoja mobilnih aplikacij za operacijski sistem Android. V članku predstavljamo sistematično primerjavo najbolj priljubljenih arhitekturnih načrtovalskih vzorcev in njihovo vlogo pri načrtovanju mobilnih aplikacij. Pri raziskovanju smo se omejili na osem arhitekturnih vzorcev iz kataloga Jetpack. Ugotoviti smo želeli, v kakšni meri ti vzorci vplivajo na potek in končni izid razvoja mobilnih aplikacij. Z uporabo vsakega izmed obravnavanih vzorcev smo razvili mobilno aplikacijo za operacijski sistem Android. Zanimalo nas je, kako se aplikacije, razvite z uporabo arhitekturnih vzorcev, primerjajo z njihovimi alternativami. Razvili smo osem mobilnih aplikacij, ki izkoriščajo prednosti obravnavanih načrtovalskih vzorcev. V namen primerjave smo razvili osem dodatnih mobilnih aplikacij, pri katerih nismo uporabili ustreznih arhitekturnih vzorcev, temveč smo aplikacijo razvili brez njih. Tako razvite aplikacije smo primerjali po več kriterijih, med katerimi so bili poglavitni zahtevnost razvoja, čas razvoja ter vrednosti programskih metrik. Na podlagi tako pridobljenih podatkov smo argumentirali smiselnost uporabe vzorcev v podanih kontekstih. V članku tako pokažemo, da je razvoj mobilnih aplikacij z uporabo arhitekturnih vzorcev kataloga Jetpack ne samo manj zahteven, ampak je kljub večjemu številu komponent, obseg izvorne kode manjši. Ugotovili smo, da sta, presenetljivo, čas razvoja in notranja kakovost aplikacij kljub uporabi načrtovalskih vzorcev primerljiva z alternativnimi pristopi.

Ključne besede: Android, mobilne aplikacije, arhitekturni vzorci, načrtovanje mobilnih aplikacij, Android Jetpack

Abstract

Choosing an appropriate architecture is a crucial and necessary step in mobile application development. In this paper, we present a systematic comparison of the most popular architectural patterns and discuss their importance for mobile app development. In our research, we focused on eight architectural patterns from the Jetpack library. We wanted to investigate their impact on the course and outcomes of mobile app development. This is why we have developed an Android application using each of the eight patterns. In order to assess their role, we developed eight more mobile applications. They differ from the previous ones only in that we did not use the Jetpack architectural patterns, but instead developed the applications ad hoc. We compared the applications using several criteria, including time, effort and code quality metrics. Based on this, we argued the rationale for using the patterns in the given contexts. In this paper, we demonstrate that the mobile app development using Jetpack architectural patterns is not only less demanding, but also requires fewer lines of code despite the higher component number. We show how, despite the use of architectural patterns, product development time and internal quality of applications was comparable to those developed with alternative approaches.

Keywords: Android, architectural patterns, mobile application design, Android Jetpack

1 UVOD

Za preživetje v visoko tekmovalnem trgu aplikacij za operacijski sistem Android je ključno, da razvijalci v kratkem času dostavijo kakovostne mobilne aplikacije. Razvoj aplikacij za operacijski sistem Android v povprečju traja 20-30 % dlje in je za tretjino dražji kot razvoj aplikacij za iOS. To je med drugim posledica tega, da je mobilne aplikacije potrebno testirati na

več napravah (Ardas Group Inc., 2017). Ustrezna arhitektura je zaradi tega pri razvoju mobilnih aplikacij za operacijski sistem Android še toliko bolj pomembna. Le-ta namreč omogoča lažje vzdrževanje in testiranje, kar zmanjša skupen čas, potreben za razvoj in vzdrževanje. Kot vidimo na sliki 1, bo uporaba ustrezne arhitekture sicer potek projekta na začetku upočasnila, vendar bodo prednosti postale vidne že v nekaj tednih. Takrat bomo lahko nove funkcionalnosti dodajali bistveno hitreje, kot če ustrezne arhitekture ne bi imeli (Fowler, Software Architecture Guide, 2019). Izbira dobre zasnove je pristop, ki lahko razvijalcem že na kratek rok pomaga pri doseganju višje kakovosti mobilnih aplikacij.

Kaj sploh je dobra arhitektura mobilne aplikacije za operacijski sistem Android, je vprašanje, ki so ga skušali nasloviti že mnogi avtorji (npr. Aymen, et al., 2019, Prabowo, et al., 2018, Verdecchia, et al., 2019). Kot ugotavljajo avtorji, mnenja pogosto kroji navdušenje za trenutno popularne tehnologije in ne objektivni dokazi. Ustrezna arhitektura je odvisna od vrste in namena mobilne aplikacije. Zato ni enotnega odgovora na vprašanje »Kaj je dobra arhitektura ?«. Kljub temu pa obstajajo splošni vzorci, dobre prakse, ki se pojavijo v večini arhitektur. Potrebo po dobri arhitekturi in lažji implementaciji teh vzorcev so prepoznali tudi pri podjetju Google in izdali katalog z imenom Android Jetpack (Android Developers, 2020). Katalog se deli na 4 kategorije: podporne knjižnice (ang. *foundation)*, arhitekturne komponente, vedenjske komponente (ang. *behaviour*) in komponente za gradnjo uporabniškega vmesnika (ang. *user interface*). V tej raziskavi smo se osredotočili zgolj na arhitekturni del, ki vsebuje osem arhitekturnih vzorcev.

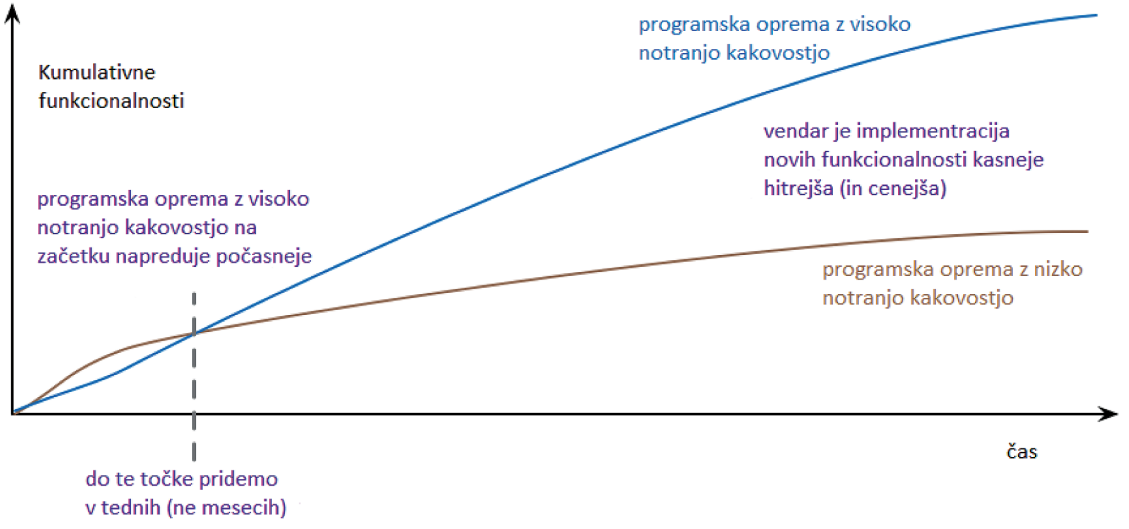
Raziskovalni vprašanji, na kateri smo odgovarjali, sta:

1. Katere so prednosti uporabe arhitekturnih vzorcev Jetpack pri razvoju mobilnih aplikacij?
   a. krajši čas razvoja,
   b. manjša zahtevnost razvoja,
   c. velikost izvorne kode izdelka,
   d. izboljšana notranja kakovost.
2. Ali, kdaj in v kolikšni meri je arhitekturne vzorce kataloga Jetpack smiselno uporabiti?

V nadaljevanju članka bodo v 2. poglavju predstavljena sorodna dela in arhitekturni vzorci podrobneje. 3. poglavje povzema metode raziskovanja. Podrobneje bomo predstavili raziskovalna vprašanja, postopek raziskovanja in način izvedbe meritev. 4. poglavje vsebuje predstavitev rezultatov, ki jih bomo v 5. poglavju podrobneje analizirali. V zadnjem poglavju na kratko povzemamo bistvo in rezultate članka.

2 SORODNA DELA

Konec 70. let se je ob nastanku grafičnih uporabniških vmesnikov pojavil arhitekturni načrtovalski vzorec MVC (ang. *Model-View-Controller*). Le-ta programsko logiko razdeli na model (ang. *model*), po-



Slika 1: **Čas implementacije funkcionalnosti pri programski opremi s kakovostno in nekakovostno zasnovo (Fowler, Software Architecture Guide, 2019)**

gled (ang. *view*) in krmilnik (ang. *controller*). Njegova glavna naloga je ločitev predstavitve podatkov od poslovne logike (Reenskaug, 2003). Iz te ideje se je razvil vzorec MVP (ang. *Model-View-Presenter*), ki je idejo ločitve logike od predstavitve podatkov popeljal še dlje in uvedel bolj pasivni pogled, ki ne vsebuje predstavitvene logike (Potel, 1996). Vzorec MVVM (ang. *Model-View-ViewModel*) je še ena variacija vzorca MVC, ki se ukvarja s problematiko ločitve poslovne logike in predstavitve podatkov. Osnova zanj je ideja Martina Fowlerja (Fowler, Presentation model, 2004), s pomočjo katere so inženirji pri Microsoftu ustvarili konkretno arhitekturo (Gossman, 2005). Razvijalci mobilnih aplikacij se pri načrtovanju arhitekture pogosto odločajo tudi za t.i. pristop *Clean Architecture*, ki ni le vzorec, ampak celostna načrtovalska filozofija. Njen glavni cilj je ločiti komponente tako, da odvisnosti prihajajo le od zunanjih proti notranjim slojem (Martin, 2012).

Avtorji (Verdecchia, et al., 2019) so v svojem delu raziskali načine, na katere razvijalci mobilnih aplikacij za operacijski sistem Android načrtujejo arhitekturo aplikacij. Zanimalo jih je predvsem, katerih vzorcev in dobrih praks se razvijalci najbolj pogosto poslužujejo in kakšen vpliv imajo le-te na kakovost. V sklopu dela so opravili vodene razgovore z razvijalci ter preučili obstoječo literaturo. Pri rezultatih so predstavili najbolj uporabljene arhitekturne vzorce in knjižnice za snovanje arhitekture mobilnih aplikacij. Sestavili so tudi seznam zahtev kakovosti, ki se razvijalcem zdijo najpomembnejše pri razvoju mobilnih aplikacij. Ugotovili so, da razvijalci najpogosteje uporabljajo arhitekturni vzorec MVP, najpogosteje uporabljeni katalogi pa so RxJava, Dagger in Jetpack. Rezultati so pokazali tudi, da je izmed množice atributov kakovosti razvijalcem najbolj pomembna ravno vzdrževalnost, sledita ji možnost testiranja in učinkovitost delovanja mobilne aplikacije.

Avtor (Lou, 2016) je v svojem delu preverjal, ali sta vzorca MVVM in MVP res boljša od tradicionalnega pristopa. Primerjavo vzorcev je pripravil na osnovi treh kriterijev: možnost testiranja, možnost spreminjanja in učinkovitost delovanja. Ugotovil je, da sta vzorca MVVM in MVP od MVC boljša po vseh kriterijih, med seboj pa sta si preveč podobna, da bi lahko enega označil za boljšega. Kot poglavitno razliko med vzorcema je izpostavil to, da nudi MVP boljšo možnost spreminjanja, MVVM pa boljšo možnost testiranja.

Avtorji (Aymen, et al., 2019) so se v svojem delu osredotočili na MVC arhitekturne vzorce mobilnih aplikacij za operacijski sistem Android. Želeli so ugotoviti, katere arhitekturne vzorce je priporočljivo uporabljati, ter kakšni so trendi na področju načrtovanja arhitekture mobilnih aplikacij za operacijski sistem Android. Zanimalo jih je tudi, kako pogosto mobilne aplikacije uporabljajo arhitekturne vzorce na osnovi vzorca MVC in katere vrste mobilnih aplikacij se takšnih vzorcev poslužujejo najbolj pogosto. V svojem delu so šli korak dlje od prejšnjega avtorja (Lou, 2016), saj so želeli avtomatizirati prepoznavo arhitekture, ki jo mobilna aplikacija uporablja. S pristopom RI-MAZ, ki z uporabo hevristik identificira dominanten arhitekturni vzorec mobilne aplikacije, so preučili 5480 aplikacij, dostopnih na tržnici Google Play. Ugotovili so, da je najbolj dominanten vzorec MVC, redkeje je uporabljen vzorec MVP, vzorec MVVM pa je v vzorcu aplikacij skoraj neuporabljen. Rezultati so pokazali tudi, da veliko število aplikacij (predvsem manjših) ne sledi nobenemu arhitekturnem vzorcu.

Avtorji (Prabowo, et al., 2018) so se v svojem delu prav tako osredotočili na arhitekturne vzorce, ki so osnovani na vzorcu MVC. Zanimalo jih je, kako se mobilne aplikacije, ki uporabljajo arhitekturne vzorce, primerjajo z aplikacijami, ki sledijo anti-vzorcem (slabim praksam) in posledično jasne arhitekture nimajo. Pri primerjavi mobilnih aplikacij so se osredotočili na modularnost in vzdrževalnost. Analizirali in primerjali so dve mobilni aplikaciji, od katerih ena uporablja arhitekturne vzorce, druga pa jasne arhitekture ni imela. Ugotovili so, da uporaba vzorca MVP znatno poveča modularnost mobilne aplikacije, vzdrževalnost pa je bila pri obeh mobilnih aplikacijah primerljiva.

Izbira na vzorcih temelječe arhitekture je torej pomemben korak načrtovanja mobilnih aplikacij, saj bo ta izbira vplivala na potek celotnega projekta. Uporaba ustrezne arhitekture sicer potek razvoja na začetku upočasni. Prednosti bodo postale vidne že v nekaj tednih, ko bomo lahko nove funkcionalnosti dodajali bistveno hitreje, kot če ustrezne arhitekture ne bi imeli (Fowler, Software Architecture Guide, 2019).

2.1 Arhitektura mobilnih aplikacij na primeru platforme Android

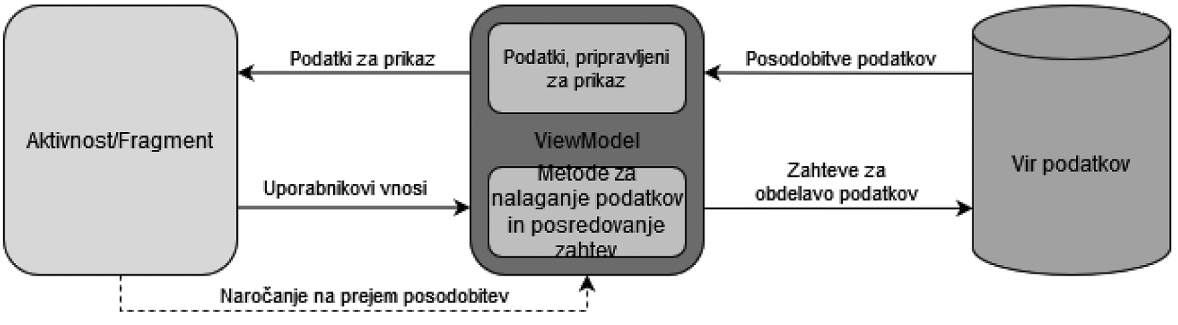
Do leta 2018 je večina mobilnih aplikacij za platformo Android uporabljala t.i. podporne knjižnice (ang. support libraries), ki so naslavljale izziv združljivosti

med napravami in različicami sistema Android. Pomembne so predvsem zaradi zagotavljanja t.i. združljivosti za naprej (ang. *forward compatibility*), kar pomeni, da bodo mobilne aplikacije, razvite za nove različice sistema Android, delovale tudi na starejših. Podporne knjižnice so se od nastanka zelo spreminjale in postajale nepregledne, zato so se pri Googlu odločili, da začnejo znova. Na podlagi prepoznanih dobrih praks so objavili katalog Jetpack, s katerim so želeli podati smernice, priporočena orodja in delno implementacijo v obliki knjižnice. Le-ta razvijalcem pomaga pri grajenju kakovostnih mobilnih aplikacij za operacijski sistem Android (Moore, 2018). Katalog se deli na 4 kategorije: podporne knjižnice (ang. *foundation)*, arhitekturne komponente, vedenjske komponente (ang. *behaviour*) in komponente za grajenje uporabniškega vmesnika (ang. *user interface*). Kljub kratkemu obstoju je katalog Jetpack hitro postal popularen, saj so ga avtorji (Verdecchia, et al., 2019) v svoji raziskavi uvrstili na 3. mesto (takoj za RxJava in Dagger) po uporabljanosti pri snovanju arhitekture mobilnih aplikacij za operacijski sistem Android. Ker se v tem članku ukvarjamo z ustrezno arhitekturo mobilnih aplikacij, se bomo osredotočili le na arhitekturni del kataloga. Ta vsebuje 8 vzorcev, ki razvijalcem pomagajo pri snovanju ustrezne arhitekture ter upravljanju in prikazovanju podatkov. To so:

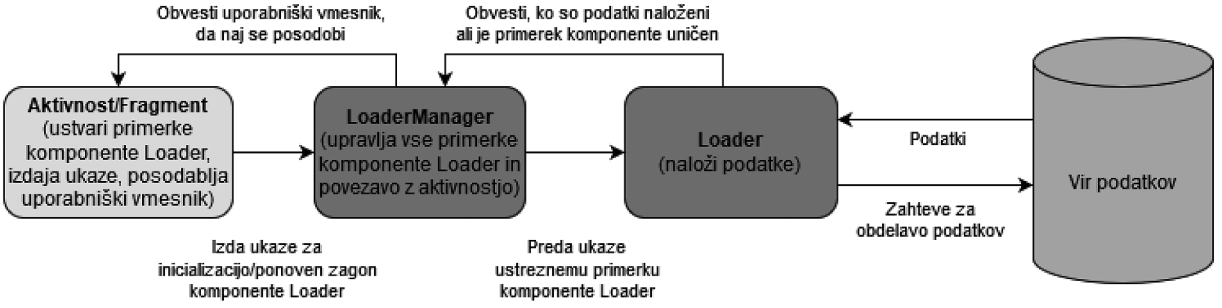
- vzorec 1: DataBinding,
- vzorec 2: Lifecycles,
- vzorec 3: LiveData,
- vzorec 4: Navigation,
- vzorec 5: Paging,
- vzorec 6: Room,
- vzorec 7: ViewModel in
- vzorec 8: WorkManager.

Za tipičnega predstavnika arhitekturnih vzorcev kataloga Jetpack lahko označimo vzorec 7 (ViewModel), ki naslavlja problematiko ločitve logike za obdelavo in prikaz podatkov (slika 2). Vzorec kataloga Jetpack je tesno povezan z arhitekturnim načrtovalskim vzorcem MVVM, saj omogoča implementacijo dela ViewModel omenjenega vzorca.

Za razumevanje implementacije vzorca ViewModel je ključno poznavanje življenjskega cikla krmilnikov uporabniškega vmesnika mobilnih aplikacij za operacijski sistem Android (aktivnosti in fragmenti). Življenjske cikle teh komponent upravlja operacijski sistem, ki lahko zaradi uporabnikovega vnosa, pomanjkanja sistemskih virov ali drugih okoliščin kadarkoli uniči ali posodobi krmilnik. S tem izbriše vse podatke, ki jih je le-ta vseboval. Za uporabnika ob neustrezni obravnavi aplikacije to pomeni nevšečnosti. Ko bo ob rotaciji zaslona aktivnost ponovno naložila začetno stanje, bodo vsi podatki, ki niso trajno shranjeni, izgubljeni. To poslabša uporabniško izkušnjo in vodi do večje porabe virov. Dodaten izziv predstavlja proženje dolgo trajajočih asinhronih klicev. Razvijalec naj bi te klice upravljal in poskrbel, da se počistijo, ko krmilnik ni več aktiven. To upravljanje zahteva ponovno veliko sistemskih virov, klici pa se pogosto ponavljajo, saj se prožijo vsakič, ko je krmilnik ponovno naložen. Vzorec ViewModel našteto problematiko reši tako, da logiko za pridobivanje podatkov loči od logike za upravljanje uporabniškega vmesnika. Slednjo umakne v razred tipa ViewModel. Objekti tega tipa se zato med spremembami konfiguracije samodejno ohranijo. Tako so podatki, ki jih vsebujejo, takoj na voljo naslednji aktivnosti ali fragmentu. S tem preprečimo izgubo podatkov, hkrati pa porabimo manj sistemskih virov, saj podatkov ni potrebno ponovno nalagati..



Slika 2: **Nalaganje podatkov z uporabo arhitekturnega vzorca ViewModel (Android Developers, 2020)**



Slika 3: **Nalaganje podatkov s pomočjo arhitekturnega vzorca Loader (Android Developers, 2020)**

Za nalaganje in posodabljanje podatkov se je tradicionalno uporabljal tudi pristop, prikazan na sliki 3. Ta definira razreda Loader, ki skrbi za pridobivanje podatkov in posredovanje podatkov ter LoadManager, ki upravlja vse primerke razreda Loader. Ko primerek razreda Loader podatke naloži, o tem obvesti razred objekt LoadManager, ta pa aktivnosti oz. fragmentu sporoči, da naj posodobi prikaze na uporabniškem vmesniku. Uporaba tega pristopa zahteva vključitev asinhronega obnašanja izvorne kode za upravljanje primerkov razreda Loader v aktivnost oz. fragment. To posledično pomeni večjo sklopljenost razredov in med drugim oteži izvajanje testov. Vzorec ViewModel je tak pristop v veliki meri zamenjal, saj omogoča enake funkcionalnosti, vendar to stori na bolj preprost in razvijalcu prijazen način. Njegova prednost je tudi v tem, da logiko za upravljanje podatkov bolje loči od uporabniškega vmesnika.

Arhitekturni vzorec ViewModel implementiramo tako, da ustvarimo razred, ki implementira vmesnik ViewModel (poleg konceptualnega opisa vzorcev je del kataloga JetPack tudi njihova implementacija). Razred vsebuje objekte, ki hranijo stanje. Te objekte nato posredujemo krmilniku uporabniškega vmesnika in jih ob interakciji uporabnika tudi posodobimo. ViewModel pridobiva podatke iz poljubnega vira in jih obdela do te mere, da jih bo krmilnik prikazal brez dodatne obdelave. Vsebuje metode za pridobivanje in vračanje podatkov ter metode za vsa dejanja, ki jih uporabnik lahko proži. ViewModel se pogosta uporablja tudi v kombinaciji z vzorcem LiveData, ki omogoča, da krmilnik uporabniškega vmesnika ob spremembah v razredu ViewModel samodejno prejme posodobljene podatke.

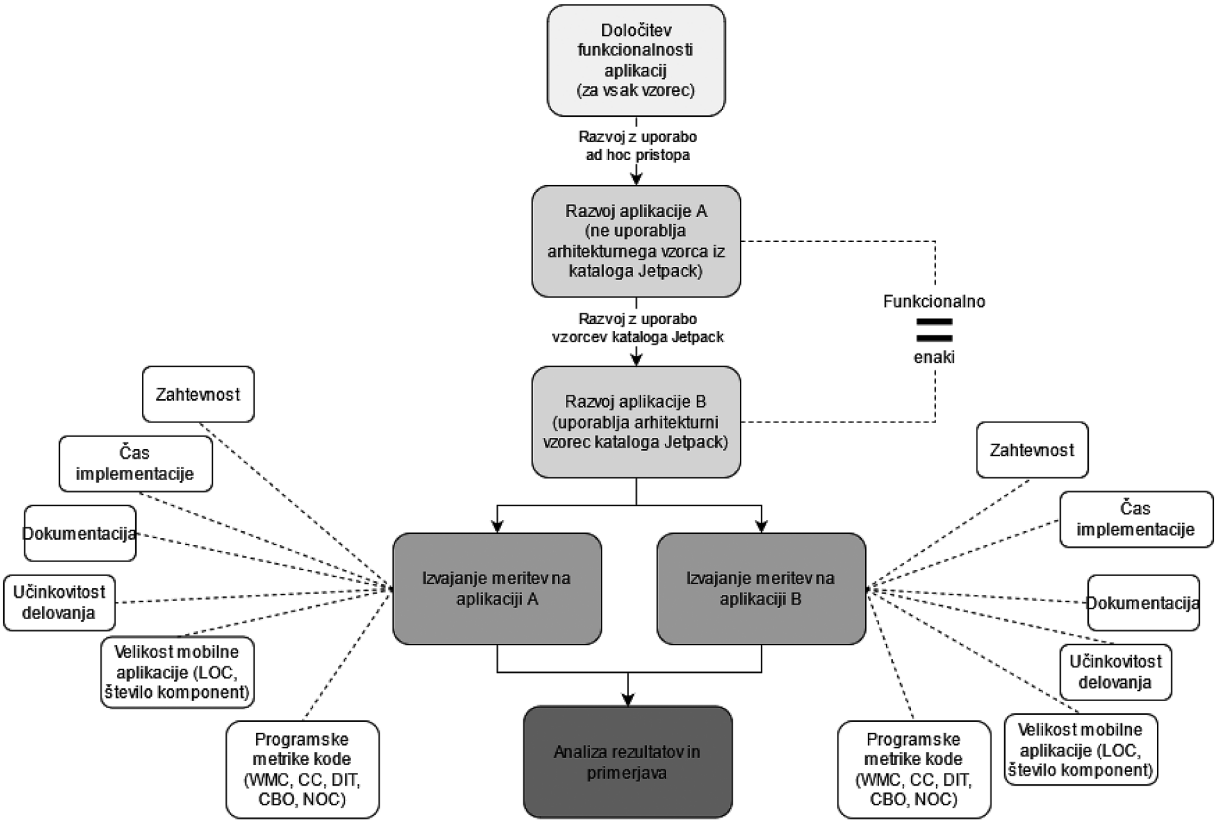
Primer razreda v programskem jeziku Kotlin, ki implementira vzorec ViewModel in skrbi za shrambo, pridobivanje ter posredovanje seznama uporabnikov:

```
class UporabnikiViewModel : ViewModel() {
    private val uporabniki: MutableLiveData<List<Uporabnik>> by lazy {
        MutableLiveData().also {
            naloziUporabnike ()
        }
    }
    fun vrniUporabnike(): LiveData<List<Uporabnik>> {
        return uporabniki
    }
    private fun naloziUporabnike() {
       //asinhrono pridobivanje seznama uporabnikov
    }
}
```

Vzorec ViewModel kataloga Jetpack omogoča relativno preprosto implementacijo ločitve podatkov od prikaza in ohranjanja stanja. Uporabimo pa ga lahko celo za prenos podatkov med različnimi aktivnostmi oz. fragmenti. Poleg lažjega prikaza podatkov razvijalce vzpodbuja tudi k vpeljavi arhitekturnega vzorca MVVM v mobilno aplikacijo.

3 METODE RAZISKOVANJA

Arhitekturne komponente kataloga Jetpack smo preizkusili tako, da smo za vsak vzorec ustvarili dve mobilni aplikaciji, ki vsebujeta enake funkcionalnosti. Arhitekturne vzorce smo uporabili le pri eni.



Slika 4: **Potek raziskovanja**

Raziskovalna metoda je prikazana na sliki 4:

1. Določitev funkcionalnosti
   Vsak arhitekturni vzorec smo preučili in določili funkcionalnosti, s katerimi bi lahko izčrpno preizkusili delovanje izbranega vzorca.
2. Na podlagi funkcionalnosti smo razvili mobilno aplikacijo A. Pri razvoju nismo uporabili arhitekturnih vzorcev, ampak je ta potekal ad hoc. V praksi je to pomenilo, da smo uporabljali že obstoječe knjižnice, funkcionalnosti, ki jih nudi že operacijski sistem Android sam. V nekaterih primerih smo komponente, potrebne za implementacijo željenih funkcionalnosti, ustvarili sami.
3. Razvili smo mobilno aplikacijo B, ki je funkcionalno enaka aplikaciji A, le da smo pri razvoju uporabili arhitekturne vzorce kataloga Jetpack. Da bi zagotovili, da je edina sprememba med aplikacijama A in B uporaba ustreznega vzorca, smo mobilno aplikacijo B razvili na osnovi implementacije A z uvedbo ustreznih sprememb.
4. Na osnovi izvorne kode obeh mobilnih aplikacijah smo zbrali podatke. Izvedli smo anketo (zahtevnost implementacije, čas implementacije, kako dobra je podpora oz. dokumentacija) in izvedli meritve. Z njimi smo ugotavljali učinkovitost delovanja, velikost rešitve (s pomočjo metrike LOC, števila komponent, metrike WMC), njeno kompleksnost (metrika CC) ter ostale strukturne značilnosti (metrike DIT, CBO in NOC).
5. Analizirali smo rezultate in pripravili primerjavo nastalih mobilnih aplikacij.

Izvajane meritve so bile sledeče:

- Zahtevnost (subjektivna ocena razvijalca: zelo nizka (1) – zelo visoka (7)),
- Čas implementacije (subjektivna ocena razvijalca,

temelječa na dejanskem merjenju časa: zelo kratek (1) – zelo dolg (6)),
- Podpora/dokumentacija (subjektivna ocena razvijalca: zelo slaba (1) – odlična (6)),
- Učinkovitost delovanja (merjenje porabe virov in odzivnosti mobilne aplikacije, agregirano v razrede: zelo slaba (1) – odlična (6)),
- Velikost mobilne aplikacije (LOC, število potrebnih komponent):
  - Število komponent – število med seboj različnih delov programske, kode, ki so nujni za implementacijo izbranega vzorca. V to štejemo razrede, statične objekte jezika Kotlin, notranje razrede, poslušalce ipd.,
- Metrike notranje kakovosti izvorne kode (CC, WMC, NOC, DIT, CBO).

Pri *zahtevnosti* smo upoštevali število potrebnih komponent, čas, težavnost in količino novega znanja, ki ga za uporabo komponente razvijalec potrebuje. Ocenili smo jo z opisno oceno na lestvici, ki se giblje od vrednosti »zelo nizka« do vrednosti »zelo visoka«.

Pri *času* smo merili čas, ki je bil potreben za implementacijo. Tudi tu smo se zaradi boljše preglednosti odločili za predstavitev podatkov v obliki opisne ocene na lestvici, ki se giblje od vrednosti »zelo kratek« do vrednosti »zelo dolg«.

Pri *podpori/dokumentaciji* smo se posvetili količini relevantne dokumentacije in njeni kakovosti. Ocenili smo jo z opisno oceno na lestvici, ki se giblje od vrednosti zelo slaba do vrednosti odlična.

Pri *učinkovitosti* delovanja smo merili porabo virov in gladkost izrisovanja uporabniškega vmesnika. Ocenili smo jo z opisno oceno na lestvici, ki se giblje od vrednosti »zelo slaba« do vrednosti »odlična«.

Metrike kakovostni izvorne kode smo izvajali s pomočjo vtičnikov CodeMR (CodeMR, 2020) in Metrics Reloaded (Leijdekkers, n.d.) ter orodja SonarCloud (SonarCloud, 2020).

4 REZULTATI

Meritve vseh implementacij smo združili in jih predstavili v tabeli 1 ter grafikonu (slika 5).

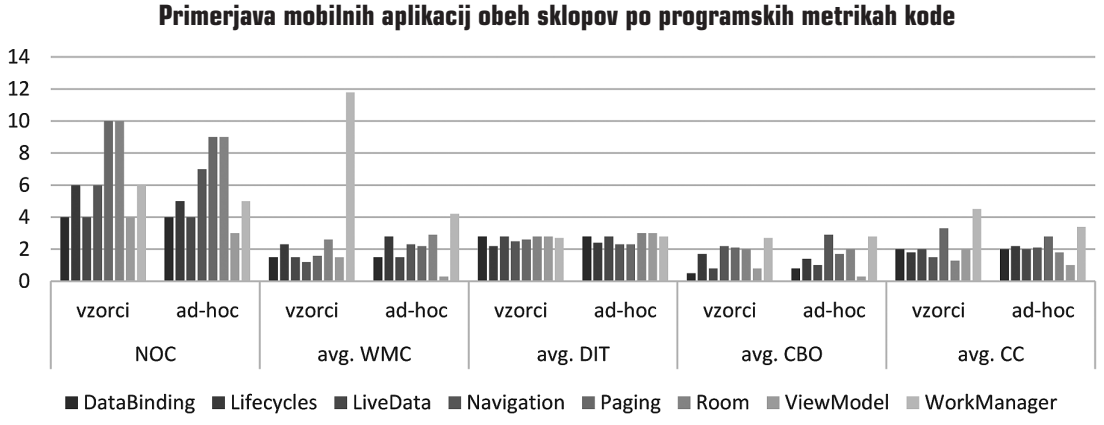
Mobilnim aplikacijam, ki so uporabljale vzorce iz kataloga Jetpack smo podali zelo podobne ocene pri kriterijih dokumentacija, čas in učinkovitost delovanja. Vsem smo pri dokumentaciji in učinkovitosti delovanja podelili najvišjo oceno, pri času implementacije pa najvišje ocene nismo podelili le mobilnima aplikacijama z vzorcema Paging in Room. Večje odstopanje smo opazili pri mobilni aplikaciji, ki uporablja vzorec WorkManager, saj vsebuje približno 500 (za faktor 3,5) vrstic kode več kot njena alternativa. Pri mobilnih aplikacijah, ki ne uporabljajo arhitekturnih vzorcev kataloga Jetpack so rezultati bolj raznoliki, vendar nobena posamezna implementacija ne izstopa.

Tudi pri programskih metrikah kakovosti izvorne kode in bilo večjih razlik. Izstopa le vrednost metrike WMC za implementacijo, ki uporablja vzorec Work Manager (slika 5).

Tabela 1: **Meritve značilnosti implementacij z in brez uporabe arhitekturnih vzorcev kataloga Jetpack**

vzorec	dokum.		čas		zahtevnost		št. komp.		učinkovitost delovanja		LOC	
Uporablja vzorce?	da	ne	da	ne	da	ne	da	ne	da	ne	da	ne
DataBinding	6	6	2	2	2	2	3	3	6	2	30	32
Lifecycles	6	5	2	1	2	1	2	1	6	6	102	88
LiveData	6	4	2	4	1	4	2	2	6	5	32	34
Navigation	6	4	2	3	1	3	1	2	6	6	76	109
Paging	6	2	5	2	5	4	4	1	6	2	232	234
Room	6	4	4	5	4	5	5	4	6	6	152	153
ViewModel	6	6	2	1	2	2	2	1	6	6	32	21
WorkManager	6	3	2	4	4	5	4	4	6	5	723	209

Legenda. dokum. = dokumentacija; čas = čas implementacije; zahtevnost = zahtevnost implementacije; št. komp. = število potrebnih komponent; LOC = število vrstic kode;



Slika 5: **Primerjava mobilnih aplikacij obeh sklopov po programskih metrikah kode**

V tabeli 2 so zbrani rezultati vseh implementacij. Podane so skupne ocene za implementacije z uporabo vzorcev kataloga Jetpack in implementacije brez. Skupno oceno za vsako vrsto implementacij smo pridobili tako, da smo izračunali povprečje vrednosti implementacij posameznega sklopa. Med analizo rezultatov smo ugotovili, da število vrstic kode pri vzorcu WorkManager izstopa in močno vpliva na skupen rezultat. Zaradi tega smo se odločili, da podamo tudi rezultate, ki ne vključujejo vzorca WorkManager.

5 RAZPRAVA

Z rezultati smo pokazali, da so se mobilne aplikacije, razvite z uporabo arhitekturnih vzorcev kataloga Jetpack in tiste razvite ad hoc, med seboj najbolj razlikovale po zahtevnosti implementacije, kakovosti dokumentacije in obsegu izvorne kode. Pri učinkovitosti delovanja, času implementacije in programskih metrikah kode so bile zaznane razlike zelo majhne. Najbolj je izstopala implementacija aplikacije, ki uporablja vzorec WorkManager, saj je imela za faktor 3,5 več vrstic izvorne kode kot implementacija aplikacije, razvite ad hoc. To odstopanje je lahko posledica tega, da so pri podjetju Google ukinili vse ostale knjižnice, ki omogočajo izvajanje nalog v ozadju in jih združili v vzorcu WorkManager. Zato je bilo ad hoc težavno ustvariti aplikacijo, ki je funkcionalno popolnoma enaka tisti z vzorcem WorkManager. Povprečen čas implementacije mobilne aplikacije z uporabo arhitekturnih vzorcev kataloga Jetpack je primerljiv, v nekaterih primerih pa malo daljši od časa implementacije aplikacije, razvite ad hoc. To je v skladu z avtorjem (Fowler, Software Architecture Guide, 2019), ki pravi da uporaba kakovostne arhitekture razvoj na začetku upočasni. Pri podjetju Google obljubljajo, da uporaba arhitekturnih vzorcev kataloga Jetpack zmanjša zahtevnost, pohitri razvoj in izboljša kakovost. Na podlagi naših rezultatov trditev o krajšem času razvoja in višji kakovosti mobilnih aplikacij nismo mogli z gotovostjo potrditi ali ovreči. Dopuščamo možnost, da se večje razlike pojavijo šele pri bolj kompleksnih implementacijah in na dolgi rok. V nadaljevanju podajamo odgovore na začetku zastavljenih raziskovalnih vprašanj.

1. **Katere so prednosti uporabe arhitekturnih vzorcev Jetpack pri razvoju mobilnih aplikacij?**
   a. **Krajši čas razvoja?**
      Pri Googlu obljubljajo, da bo uporaba kataloga Jetpack zmanjšala zahtevnost in pohitrila razvoj, vendar rezultati našega testiranja kažejo, da temu ni vedno tako. Čas implementacije je drugačen pri vsakem vzorcu. Povprečna časa implementacij z in brez uporabe vzorcev kataloga Jetpack sta podobna. Povprečen čas implementacij z uporabo kataloga je celo daljši. Kljub temu smo ugotovili, da vzorci kataloga Jetpack razvijalcu omogočajo lažje dodajanje novih funkcionalnosti in zmanjšajo čas za vzdrževanje in nadaljnji razvoj mobilnih aplikacij. **Uporaba vzorcev kataloga Jetpack ni vedno hitrejša, vendar pa lahko na dolgi rok zmanjša trud in čas za vzdrževanje in razvoj mobilne aplikacije. To je še posebej pomembno pri velikih projektih.**
   b. **Manjša zahtevnost razvoja?**
      Pri pregledu rezultatov smo ugotovili, da je vzorcem iz kataloga Jetpack skupna dobra

Tabela 2: **Primerjava implementacij, ki uporabljajo katalog Jetpack s tistimi, ki ga ne**

	Z uporabo kataloga Jetpack	**Brez uporabe kataloga Jetpack**
Zahtevnost	Nizka – srednje nizka	Srednje nizka – srednja
Čas	Srednje kratek. Za nekaj odstotkov daljši na začetku.	Srednje kratek. Čas je na začetku krajši vendar je vzdrževanje bolj dolgotrajno.
Podpora/ Dokumentacija	Odlična. Pri Googlu nudijo obsežno dokumentacijo in vodiče za vse vzorce.	Dobra. Dokumentacija je bila pogosto pomanjkljiva ali zastarela, podpora za knjižnice pa opuščena.
Povprečno število potrebnih komponent	2,9. Vzorci kataloga Jetpack v povprečju potrebujejo dobre pol komponente več. Implementacija le enega vzorca (Navigation) je potrebovala manj delov, kot implementacija brez njegove uporabe.	2,3. Implementacije brez uporabe kataloga Jetpack so dosledno (razen Navigation) potrebovale manj komponent.
Učinkovitost delovanja	Odlična. Vse implementacije vzorcev imajo majhno porabo virov, mnoge pa vsebujejo dodatne funkcionalnosti, ki aktivno pomagajo preprečiti napake.	Zelo dobra. Zaznali smo le majhna odstopanja od implementacij z uporabo vzorcev kataloga Jetpack, velika razlika pri vzorcu Paging.
Število vrstic kode (LOC)	Rezultati vseh vzorcev: Povprečno: 172 Mediana: 89 Skupno: 1379 Rezultati brez vzorca WorkManager: Povprečno: 94 Mediana: 76 Skupno: 656	Rezultati vseh vzorcev: Povprečno: 110 Mediana: 99 Skupno: 880 Rezultati brez vzorca WorkManager: Povprečno: 96 Mediana: 88 Skupno: 671
Programske metrike kode	Rezultati vseh vzorcev: Povp. WMC: 3 Povp. število razredov: 6,3 Povp. DIT: 2,7 Povp. CBO: 1,6 Povp. CC: 2,3 Rezultati brez vzorca WorkManager: Povp. WMC: 1,7 Povp. število razredov: 6,3 Povp. DIT: 2,6 Povp. CBO: 1,4 Povp. CC: 2	Rezultati vseh vzorcev: Povp. WMC: 2,2 Povp. število razredov: 5,8 Povp. DIT: 2,7 Povp. CBO: 1,6 Povp. CC: 2,2 Rezultati brez vzorca WorkManager: Povp. WMC: 1,9 Število razredov: 35 Povp. DIT: 2,7 Povp. CBO: 1,4 Povp. CC: 2

podpora in predvsem preprostost uporabe. Katalog, ki so ga ustvarili pri Google-u, nudi komponente, ki jih lahko razvijalci hitro in preprosto vključijo v svoj izdelek. Da bi dosegli večjo robustnost in preprostost uporabe, večina vzorcev zanemari določene napredne funkcionalnosti, ki jih nudijo tretje knjižnice. To je še posebej opazno, ko primerjamo vzorec LiveData in knjižnico RxJava. Slednja nudi več funkcionalnosti, vendar je zaradi preprostosti uporabe za večino projektov vzorec LiveData vseeno bolj privlačna izbira. Dobra primera tega sta tudi vzorca Navigation in Room. Le-ta nista uvedla novih funkcionalnosti, ampak sta obstoječe le ovila in razvijalcu poenostavila njihovo uporabo. Ugotovili smo, da so se pri Googlu držali obljub glede nižje zahtevnosti implementacije vzorcev kataloga Jetpack, saj je bila ta v povprečju za petino manjša od implementacij brez njihove uporabe. **Preprostost uporabe je tako eden izmed glavnih adutov kataloga Jetpack. S tem se knjižnica približa tudi začetnikom, ki z razvojem mobilnih aplikacij za operacijski sistem Android še nimajo veliko izkušenj.**

c. Velikost produkta?

Pri analizi števila vrstic kode smo ugotovili, da so bile od njihovih alternativ krajše implemen-

tacije kar petih vzorcev: DataBinding, LiveData, Navigation, Paging, Room. Štiri od teh so bile od alternativ krajše le za 2 vrstici kode ali manj, implementacija vzorca Navigation pa je bila od alternative krajša kar za 40 %. Še posebej je izstopala implementacija vzorca Workmanager, ki je bila od alternative daljša za faktor 3,5. Menimo, da je to zaradi pomanjkanja dobrih alternativ za vzorec WorkManager, zaradi česar je bilo brez njene uporabe zelo težavno ustvariti popolnoma enakovredno implementacijo. **Kljub temu odstopanju smo ugotovili, da v splošnem implementacija z uporabo kataloga Jetpack zahteva manj vrstic kode, v primerjavi z implementacijo brez njegove uporabe.** Čeprav je razlika zelo majhna, lahko rečemo, da so se pri podjetju Google obljube o manj t.i. obrtniške kode držali. Kljub manjšemu številu vrstic kode pa so implementacije, ki so uporabljale vzorce kataloga Jetpack, v povprečju potrebovale 26 % več komponent in skoraj 6 % več razredov kot njihove alternative.

d. Izboljšana notranja kakovost?

Aplikacije, razvite z uporabo vzorcev kataloga Jetpack, v povprečju potrebujejo več komponent in razredov kot aplikacije razvite ad hoc. Vendar za njihovo implementacijo porabijo manjše število vrstic kode in približno enako časa. Poleg tega je njihova implementacija tudi manj zahtevna, kar nakazuje na dobro strukturiranost vzorcev. Mobilne aplikacije, ki so uporabljale vzorce kataloga Jetpack, so imele po naših meritvah tudi večjo učinkovitost delovanja. Kljub temu smo po analizi izmerjenih vrednosti metrik kakovosti programske kode ugotovili, da z izjemo metrik WMC in NOC med aplikacijami, ki uporabljajo vzorce iz kataloga Jetpack in tistimi ki jih ne, ni večjih razlik. Aplikacije z vzorci kataloga Jetpack v povprečju vsebujejo več razredov, ciklomatična kompleksnost, globina delovanja in sklopljenost pa so primerljive z vrednosti alternativnih implementacij. Povprečna vrednost metrike WMC za aplikacije, ki uporabljajo arhitekturne vzorce kataloga Jetpack je za 36 % večja od povprečne vrednosti mobilnih aplikacij, ki teh vzorcev niso uporabljale. Ko smo izračunali še povprečne vrednosti brez aplikacij vzorca WorkManager, se je trend obrnil in so bile vrednosti mobilnih aplikacij, ki uporabljajo vzorce kataloga Jetpack, za 11 % manjše od vrednosti njihovih alternativ. Nižji sta bili tudi vrednosti metrik CC in DIT, vendar sta se vrednosti spremenili za manj kot 5 %, zato jim ne moremo pripisati velikega pomena. **Vzorci kataloga Jetpack se osredotočajo na učinkovitost delovanja ter nizko kompleksnost, vendar po naših rezultatih z gotovostjo ne moremo trditi, da uporaba arhitekturnih vzorcev vodi do višje notranje kakovosti mobilnih aplikacij.**

2. Ali, kdaj in v kolikšni meri je arhitekturne vzorce kataloga Jetpack smiselno uporabiti?

Kot so pokazali rezultati, arhitekturne vzorce kataloga Jetpack odlikujeta predvsem nizka zahtevnost implementacije in dobra dokumentacija. To vzorce približa manjšim projektom in manj izkušenim razvijalcem. **Zaradi modularnosti in nezahtevne vključitve v projekt je vzorce smiselno uporabiti tudi pri obstoječih projektih, ki so bili razviti z drugimi pristopi.** Vzorci so primerni tudi za uporabo v večjih projektih, vendar pod pogojem, da razvijalcem zagotavljajo vse funkcionalnosti, ki jih ti potrebujejo. Arhitekturni vzorci kataloga Jetpack za dosego nizke zahtevnosti pogosto žrtvujejo dodatne funkcionalnost in prilagodljivost. Če potrebujejo razvijalci več nadzora in prilagodljivosti pri implementaciji funkcionalnosti, potem arhitekturni vzorci kataloga Jetpack pogosto niso prava izbira. Dobra primera tega sta vzorca LiveData in Navigation. Vzorec navigation omogoča hitro vključitev v projekt in v veliki meri olajša pomikanje med fragmenti. Če želi razvijalec sam upravljati dejanje ob pritisku gumba »nazaj«, mora uporabiti drug pristop, saj vzorec Navigation tega ne omogoča. Podobno smo opazili pri vzorcu LiveData. Ta je zelo preprost za uporabo, vendar se je v kompleksnejših primerih iz vidika ponujenih funkcionalnosti bolje izkazala knjižnica RxJava.

Vzorci kataloga Jetpack so zasnovani tako, da jih je preprosto vključiti v nov ali že obstoječ projekt. Vzorci niso vezani na specifično različico operacijskega sistema Android in so združljivi za nazaj. To pomeni, da bodo funkcionalnosti, implementirane s pomočjo vzorcev kataloga Jetpack, delovale tudi na starejših različicah sistema Android.

Za uporabo enega vzorca ni potrebno vključiti celotnega kataloga vzorcev, ampak se lahko uporabljajo tudi povsem individualno. Kljub temu se nekateri vzorci najbolj izkažejo takrat, ko so uporabljeni skupaj. Najboljši primer tega so vzorci Room, ViewModel in LiveData. Skupaj poskrbijo za hranjenje in prenos podatkov od denimo podatkovne baze vse do uporabniškega vmesnika. Bili so zasnovani za skupno uporabo, zato jih je povezati lažje, kot če bi katerega izmed njih nadomestili z zunanjim vzorcem. Nekateri arhitekturni vzorci kataloga Jetpack so celo integrirani v Android Studio (Room, Navigation), kar še dodatno zmanjša zahtevnost implementacije in vzorce približa razvijalcem.

6 ZAKLJUČEK

V članku smo demonstrirali pomen in vpliv izbire ustrezne arhitekture na potek razvoja mobilnih aplikacij za operacijski sistem Android. Pri raziskovanju smo se osredotočili na osem arhitekturnih vzorcev kataloga Jetpack. Pokazali smo, da je razvoj mobilnih aplikacij z njihovo uporabo ne samo manj zahteven, temveč je kljub večjemu številu komponent število obseg izvorne kode manjši, čas razvoja in kakovost izdelka pa sta primerljiva z alternativnimi pristopi. Ugotovili smo, da so glavne prednosti arhitekturnih vzorcev kataloga Jetpack nizka zahtevnost implementacije, dobra dokumentacija in dobra strukturiranost. Izkazalo se je, da je vzorce smiselno uporabiti tako pri preprostih kot tudi bolj kompleksnih projektih. Preprostost (in posledično manjša prilagodljivost) arhitekturnih vzorcev kataloga Jetpack lahko predstavlja tudi dodatne izzive. Če razvijalci že vnaprej vedo, da potrebujejo veliko nadzora nad delovanjem vseh delov mobilne aplikacije, potem arhitekturni vzorci kataloga Jetpack morda niso prava izbira.

Naš članek izpostavlja pomen arhitekturnih vzorcev pri razvoju mobilnih aplikacij za operacijski sistem Android ter prednosti in pomanjkljivosti uporabe arhitekturnih vzorcev kataloga Jetpack. Razvijalcem pomaga tudi, da se na podlagi zahtev projekta odločijo, če je uporaba vzorcev kataloga Jetpack smiselna ali ne.

ZAHVALA

Ta raziskava je nastala ob podpori raziskovalnega programa št. P2-0057, katerega je sofinancirala Javna agencija za raziskovalno dejavnost Republike Slovenije iz državnega proračuna.

LITERATURA

[1] Android Developers. (2020). Android Jetpack. Pridobljeno iz Android Developers: https://developer.android.com/jetpack

[2] Android Developers. (30. 10 2020). ViewModel. Pridobljeno iz Android Developers: https://developer.android.com/topic/libraries/architecture/viewmodel

[3] Ardas Group Inc. (15. Junij 2017). How long will it take to create your mobile application. Pridobljeno iz Ardas: https://ardas-it.com/how-long-will-it-take-to-create-your-mobile-application

[4] Aymen, D., Ghizlane, E., Naouel, M., Sègla, K. (2019). An exploratory study of MVC-based architectural patterns in Android apps. SAC '19: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, 1711–1720.

[5] CodeMR. (2020). CodeMR. Pridobljeno iz CodeMR: https://www.codemr.co.uk/

[6] Fowler, M. (19. 7 2004). Presentation model. Pridobljeno iz martinfowler.com: https://martinfowler.com/eaaDev/PresentationModel.html

[7] Fowler, M. (18. 7 2006). GUI Architecture. Pridobljeno 26. 5 2020 iz https://martinfowler.com/eaaDev/uiArchs.html

[8] Fowler, M. (1. 8 2019). Software Architecture Guide. (Martin Fowler) Pridobljeno 28. 5 2020 iz https://martinfowler.com/architecture/

[9] Gossman, J. (8. 10 2005). Introduction to Model/View/ViewModel pattern for building. (Microsoft) Pridobljeno 27. 5 2020 iz https://docs.microsoft.com/en-gb/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps

[10] Leijdekkers, B. (brez datuma). Metrics Reloaded. (Github) Pridobljeno 2. 8 2020 iz https://github.com/BasLeijdekkers/MetricsReloaded

[11] Lou, T. (2016). A Comparison of Android Native App Architecture – MVC, MVP and MVVM. Eindhoven.

[12] Martin, R. C. (13. 8 2012). The Clean Architecture. Pridobljeno iz Clean Coder Blog: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[13] Moore, K. (4. 7 2018). Introduction to Android Jetpack. (Ray Wenderlich) Pridobljeno 29. 5 2020 iz https://www.raywenderlich.com/5376-introduction-to-android-jetpack#toc-anchor-001

[14] Potel, M. (1996). MVP: Model-View-Presenter. Taglient Inc, 5-9.

[15] Prabowo, G., Suryotrisongko, H., Tjahyanto, A.. (2018). A Tale of Two Development Approach: Empirical Study on The Maintainability and Modularity of Android Mobile Application with Anti-Pattern and Model-View-Presenter Design Pattern. 2018 International Conference on Electrical Engineering and Informatics (ICELTICs).

[16] Reenskaug, T. (2003). MVC XEROX PARC 1978-79. Pridobljeno iz Trygve M. H. Reenskaug: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

[17] SonarCloud. (2020). SonarCloud. Pridobljeno iz SonarCloud: https://sonarcloud.io/

[18] Tatarka, E. (2014). holdr. Pridobljeno iz https://github.com/evant/holdr

[19] Verdecchia, R., Malavolta, I., Lago, P. (2019). Guidelines for Architecting Android Apps: A Mixed-Method Empirical Study. 2019 IEEE International Conference on Software Architecture (ICSA). doi:10.1109/ICSA.2019.00023

[20] Wharton, J. (brez datuma). Butterknife. (Jake Wharton) Pridobljeno iz http://jakewharton.github.io/butterknife/

▪

Luka Pavlič je docent na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Doktoriral je leta 2009 iz tematike ponovne uporabe s pomočjo vzorcev. Njegovo raziskovalno delo obsega tehnične in organizacijske aspekte razvoja informacijskih rešitev, njihovo kakovost ter IT arhitekture. Je avtor oz. soavtor večjega števila izvirnih znanstvenih člankov, ki so objavljeni v najuglednejših revijah.

▪

Luka Četina je tehniški sodelavec na Fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Trenutno je podiplomski študent na študijskem programu Informatika in tehnologije komuniciranja.