VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2020 Bc. Jan Maloušek

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

NÁVRHOVÉ VZORY ARCHITEKTURY OS ANDROID S VYUŽITÍM JAZYKA KOTLIN
ANDROID OS SOFTWARE DESIGN PATTERNS UTILIZING THE KOTLIN LANGUAGE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Jan Maloušek**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Kryštof Zeman, Ph.D.**
SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**

Ústav telekomunikací

Student: Bc. Jan Maloušek ***ID:*** 186445
Ročník: 2 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Návrhové vzory architektury OS Android s využitím jazyka Kotlin

POKYNY PRO VYPRACOVÁNÍ:

Cílem diplomové práce bude seznámení se s návrhovými vzory využívanými v mobilních aplikacích pro OS Android za využití jazyka Kotlin a jejich následná implementace. V teoretické části bude proveden podrobný popis teorie návrhových vzorů, společně s popisem využívaných “best practices” metod při vývoji aplikací pro OS Android (Dependency Injection, reaktivní programování, testování aplikace...). Následně budou tyto poznatky aplikovány ve vzorové aplikaci, která bude sloužit k demonstraci jejich validity. Výstupem diplomové práce bude podrobný popis návrhových vzorů, a funkční aplikace demonstrující výhody návrhových vzorů pro OS Android.

DOPORUČENÁ LITERATURA:

[1] Kotlin documentation [online], 2019. [cit. 2019-09-16]. Dostupné z: https://kotlinlang.org/docs/tutorials/kotlin-android.html

[2] JEMEROV, Dmitry a Svetlana ISAKOVA, [2017]. Kotlin in action. Shelter Island, NY: Manning Publications Co. ISBN 978-161-7293-290.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 1.6.2020

Vedoucí práce: Ing. Kryštof Zeman, Ph.D.

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Cílem této diplomové práce je popis návrhových vzorů a dalších ověřených programátorských praktik využívaných při vývoji aplikací pro OS Android. V rámci teoretické části je proveden rozbor návrhových vzorů, programovacího jazyka Kotlin, dependency injection, reaktivního programování a automatického testování aplikací. Praktická část se zabývá návrhem a implementací aplikace, na které jsou demonstrovány výhody využívání návrhových vzorů a ověřených programátorských praktik popsaných v teoretické části.

KLÍČOVÁ SLOVA

Android, dependency injection, Kotlin, mobilní aplikace, návrhové vzory, reaktivní programování

ABSTRACT

The aim of this thesis is to describe design patterns and other programming best practices used in the development of Android applications. The theoretical part analyzes design patterns, Kotlin programming language, dependency injection, reactive programming and automatic testing. The practical part deals with the design and implementation of a sample Android application, which demonstrates the advantages of using design patterns and other proven programming practices described in the theoretical part.

KEYWORDS

Android, dependency injection, Kotlin, mobile application, software design patterns, reactive programming

MALOUŠEK, Jan. *Návrhové vzory architektury OS Android s využitím jazyka Kotlin*. Brno, 2020, 129 s. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. Kryštof Zeman, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Návrhové vzory architektury OS Android s využitím jazyka Kotlin“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Kryštofu Zemanovi Ph.D. za mnoho hodin, které se mnou strávil při konzultacích, za odborné vedení, trpělivost a podnětné návrhy k práci i k samotné aplikaci.
Dále bych rád poděkoval rodičům Stanislavě Malouškové a Miroslavovi Malouškovi a bratrovi Martinu Malouškovi za veškerou podporu i pomoc při propagaci aplikace. Velké díky dále patří Ing. Ondřeji Hudečkovi za pomoc s matematickým obsahem aplikace, Silvii Patzeltové za slovenskou lokalizaci aplikace a mnohým dalším, kteří mi pomohli s testováním aplikace.

Obsah

Seznam obrázků

Seznam výpisů

Úvod

Trh mobilních aplikací zažil v posledních deseti letech bouřlivý rozvoj. Aby aplikace zůstaly konkurenceschopné, je nutné vydávat časté aktualizace, které pravidelně vylepšují funkcionalitu a opravují chyby. Zdrojový kód aplikací může po čase dosahovat znatelné velikosti, kvůli čemuž mohou být další úpravy a rozšíření velmi náročné.

V současné době tak sílí tlak na kvalitní zdrojový kód dodržující ověřené programátorské praktiky, mezi které patří využívání návrhových vzorů, dodržování principu jedné zodpovědnosti, navržení vhodné architektury a další. Tyto techniky zaručí rychlejší vývoj, vyšší stabilitu aplikací a v neposlední řadě také dlouhodobou udržitelnost zdrojového kódu i při jeho značné rozsáhlosti.

Tato práce má dva hlavní cíle. Prvním cílem je v teoretické části popsat ověřené programátorské praktiky pro tvorbu kvalitního zdrojového kódu se zaměřením na architekturu aplikace a její rozčlenění na vrstvy. Druhým cílem je v rámci praktické části navrhnout a implementovat aplikaci, která demonstruje výhody využívání ověřených programátorských praktik popsaných v teoretické části práce.

První kapitola popisuje běžně využívané návrhové vzory. Zaměřuje se na tři hlavní kategorie návrhových vzorů včetně jejich často využívaných zástupců. Další kapitola se věnuje speciální skupině návrhových vzorů souvisejících s architekturou a rozčleněním aplikace na vrstvy. Třetí kapitola popisuje programovací jazyk Kotlin, který je novým doporučovaným jazykem při psaní aplikací pro OS Android [1]. V následujících dvou kapitolách jsou rozebrány techniky dependency injection a reaktivní programování, jejichž cílem je další zvýšení kvality a modulárnosti zdrojového kódu. Poslední teoretická kapitola se zaměřuje na principy automatického testování aplikací.

Sedmá kapitola této práce je zaměřena na základní popis vyvíjené aplikace. Kapitola obsahuje zdůvodnění výběru tématu dané aplikace, definici požadavků, průzkum trhu a volbu technologií, které byly při vývoji využity. Další kapitola se zaobírá návrhem uživatelského prostředí aplikace a stanovením konkrétních funkcionalit. V deváté kapitole je popsána struktura dat, která umožňuje pokrýt funkcionalitu definovanou v předchozí kapitole. Poslední, desátá kapitola, popisuje samotnou implementaci aplikace. Popsán je zde způsob, jakým je aplikace rozčleněna na vrstvy a jak spolu jednotlivé vrstvy komunikují.

V závěru této práce jsou zmíněny dosažené výsledky a možnosti rozšíření aplikace do budoucna.

1 Návrhové vzory

Návrhové vzory slouží jako znovupoužitelné šablony pro řešení určitých problémů při vývoji softwaru, přičemž nejsou nijak vázány na konkrétní programovací jazyk.

Jsou tak do určité míry podobné algoritmům, avšak s tím rozdílem, že algoritmy řeší výpočetní problémy, zatímco návrhové vzory řeší problémy spojené s konstrukcí a uspořádáním kódu. Jinými slovy algoritmy se zabývají otázkou jak daný problém vyřešit co nejefektivnějším způsobem. Návrhové vzory řeší otázku jakým způsobem navrhnout strukturu kódu tak, aby byl kód dobře čitelný, přehledný a rozšiřitelný. Výhodou využívání návrhových vzorů je tak zrychlení vývoje softwaru, vyšší čitelnost, konzistentnost a celkově vyšší kvalita zdrojového kódu [2, 3].

Návrhové vzory se dělí do třech hlavních kategorií:

1. **Tvořivé** - Vytváření a inicializace komponent.
2. **Strukturální** – Skládání komponent do větších struktur.
3. **Behaviorální** – Přiřazování zodpovědností mezi objekty a jejich vzájemná komunikace [4].

V následujících podkapitolách budou popsány zástupci návrhových vzorů z jednotlivých kategorií, které se často využívají při vývoji aplikací pro OS Android.

1.1 Tvořivé návrhové vzory

Tvořivé návrhové vzory poskytují abstrakci k inicializaci komponent, přičemž bývají zodpovědné za to, jakým způsobem jsou objekty vytvořeny nebo sestaveny, kdo tyto komponenty vytvoří a kdy budou vytvořeny. Mezi tvořivé návrhové vzory nejčastěji využívané při vývoji aplikací pro OS Android patří Builder, Factory a Singleton [4, 5].

1.1.1 Builder

Návrhový vzor Builder separuje tvorbu komplexních objektů od jejich reprezentace, díky čemuž je možné stejným konstrukčním procesem vytvořit mnoho druhů reprezentací objektů. Umožňuje tak flexibilně vytvářet objekty, aniž by jejich třídy musely obsahovat velké množství konstruktorů [4, 5].

Tvorba objektů pomocí návrhového vzoru Builder probíhá po krocích, kdy se v každém kroku stanoví jedna vlastnost tvořeného objektu. Typickým zástupcem návrhového vzoru Builder v OS Android je například tvorba Alert Dialogů [4, 5].

Příklad implementace návrhového vzoru Builder je vidět ve výpisu 1.1.

Výpis 1.1: Ukázka implementace návrhového vzoru Builder v jazyku Kotlin.

```
class RestaurantOrder private constructor(builder: RestaurantOrder.
    Builder) {
    val mainCourse: String?
    val dessert: String?

    init {
        this.mainCourse = builder.mainCourse
        this.dessert = builder.dessert
    }

    class Builder {
        //Proměnné s privátními settery
        var mainCourse: String? = null
            private set
        var dessert: String? = null
            private set
        //Nastavování vlastností objektu
        fun setMainCourse(mainCourse: String): Builder {
            this.mainCourse = mainCourse
            return this
        }
        fun setRequestDessert(dessert: String): Builder {
            this.dessert = dessert
            return this
        }
        //vytvoření objektu
        fun build() = RestaurantOrder(this)
    }
}

fun someFunction(){
//vytvoření objektu pomocí navrhového vzoru Builder
    val restaurantOrder = RestaurantOrder.Builder()
            .setDessert("Koláč")
            .setMainCourse("Řízek")
            .build()
}
```

1.1.2 Abstract Factory

Návrhový vzor Abstract Factory přesunuje zodpovědnost tvorby objektů z klienta na komponentu Factory. V rámci návrhového vzoru Factory jsou tak objekty vytvářeny voláním metod komponenty Factory, namísto voláním konstruktorů. Komponenta Factory tak může zapouzdřovat komplikovanou logiku tvorby objektů a

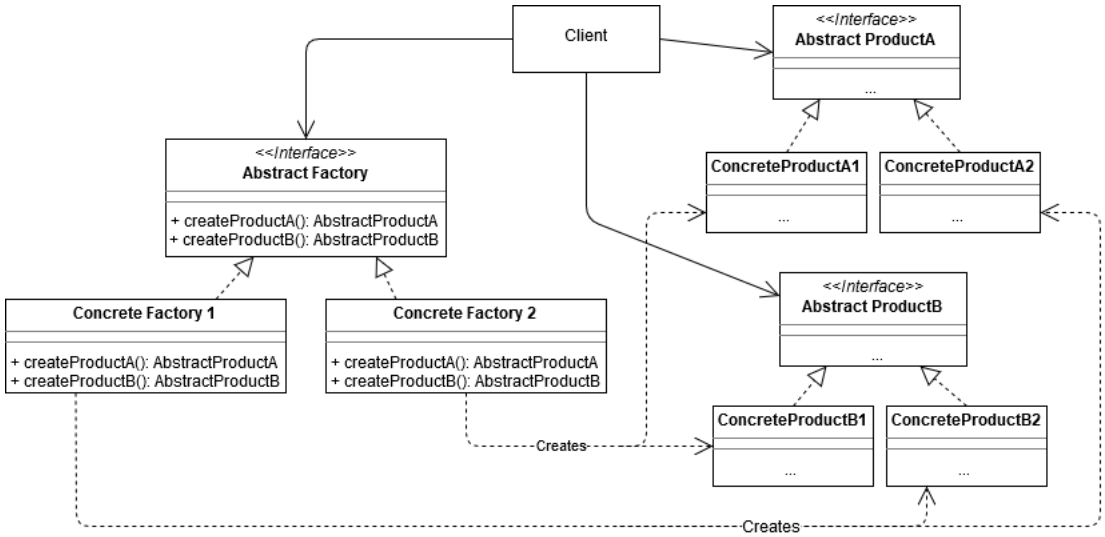
může spravovat inicializační a alokační proces, kdy nemusí nutně navracet vždy novou instanci požadovaných objektů. Typickým příkladem využití návrhového vzoru Abstract Factory je obstarávání ViewModelů v rámci sady knihoven Android Architecture Components [1, 4, 6, 7].

Komponenty návrhového vzoru Abstract Factory

Návrhový vzor Abstract Factory se skládá z těchto komponent:

- **Abstract Factory** – Interface definující operace na vytváření objektů.
- **Concrete Factory** – Komponenta, která obsahuje implementace funkcí definovaných v Abstract Factory. Právě tato komponenta je přímo zodpovědná za vytváření nových instancí objektů, implementujících interface Abstract Product. Ve většině případu se implementují jako Singletony.
- **Abstract Product** – Interface pro skupinu objektů vytvářených v rámci návrhového vzoru Abstract Factory.
- **Concrete Product** – Konkrétní třída implementující interface Abstract Product, jejíž objekty jsou vytvářeny komponentou Concrete Factory.
- **Client** – Zdrojový kód využívající k tvorbě objektů interface Concrete Factory. K vytvořeným komponentám pak přistupuje prostřednictvím interfacu Abstract Product [4].

Uspořádání komponent návrhového vzoru Abstract Factory prostřednictvím UML (Unified Modeling Language) diagramu tříd je vidět na obr. 1.1.



Obr. 1.1: UML diagram návrhového vzoru Abstract Factory [4, 7].

Výhody vytváření objektů pomocí návrhového vzoru Abstract Factory

- Návrhový vzor Abstract factory prosazuje Dependency Inversion Principle, kdy závislosti klienta jsou na interfacy a nikoliv na konkrétní implementace, což snižuje provázanost kódu. Konkrétní třídy jsou tak izolovány od zbytku kódu, díky čemuž se snadněji upravují a rozšiřují [4, 7].
- Průběh inicializace objektů lze snadno změnit výměnou/úpravou komponenty Concrete Factory [4, 7].
- Vyšší konzistentnost vytvářených objektů, neboť jsou všechny inicializovány využíváním stejné komponenty Concrete Factory [7].

1.1.3 Singleton

Cílem návrhového vzoru Singleton je vytvoření pouze jediné instance dané třídy a poskytnutí globálního přístupu k této instanci. Singletony jsou obyčejně přístupné přes statickou funkci *getInstance()*. V případě, že Singleton dosud nebyl inicializován, vytvoří tato funkce novou instanci prostřednictvím privátního konstruktoru, uloží ji do privátní statické proměnné a instanci navrátí klientovi. Pokud již Singleton inicializován byl, navrátí funkce instanci Singletonu z privátní statické proměnné [4, 8].

Příklad implementace Singletonu v jazyce Java je vidět na výpisu 1.2.

Výpis 1.2: Ukázka implementace návrhového vzoru Singleton v jazyku Java.

```
1  class Singleton
2  {
3      // statická privátní proměnná pro uložení instance Singletonu
4      private static Singleton instance = null;
5      //konstruktor s privátním přístupem
6      private Singleton(){}
7      // statická metoda na obstarání instance Singletonu
8      public static Singleton getInstance()
9      {
10         if (instance == null)
11             instance = new Singleton();
12         return instance;
13     }
14 }
15
16 void someFunction(){
17     //obstarání instance Singletonu
18     Singleton singleton = Singleton.getInstance();
19 }
```

Programovací jazyk Kotlin přímo podporuje Singletony prostřednictvím klíčového slova object. Předchozí implementace Singletonu v jazyce Java se pak v jazyce Kotlin zjednoduší, viz výpis 1.3 [9].

Výpis 1.3: Ukázka implementace návrhového vzoru Singleton v jazyku Kotlin.

```
1 //Deklarace Singletonu
2 object Singleton{}
3
4 fun someFunction(){
5     //obstarání instance Singletonu
6     val singleton = Singleton()
7 }
```

Singletony je vhodné používat všude tam, kde potřebujeme pouze jednu instanci dané třídy v celé aplikaci. Často jsou tak využívány pro přístup ke globálním službám, jako je například logování nebo databáze [4, 8, 10].

Naopak je nevhodné využívat Singletony na sdílení dat mezi komponentami. Aplikace je pak náchylná k chybám a její zdrojový kód je hůře pochopitelný, testovatelný a hůře se v něm hledají a opravují chyby. Využívání Singletonů ve velké míře zvyšuje rovněž provázanost kódu, což dále zhoršuje jeho testovatelnost a přehlednost [10, 11].

1.2 Strukturální návrhové vzory

Strukturální návrhové vzory se zabývají otázkou jak skládat komponenty programů do větších celků. Tato skupina návrhových vzorů tak aranžuje objekty známým a ověřeným způsobem, aby vykonávaly typické úlohy. Mezi strukturální návrhové vzory často využívané při vývoji aplikací pro OS Android patří Adapter a Facade [4, 5].

1.2.1 Adapter

Návrhový vzor Adapter umožňuje spolupráci dvou nekompatibilních tříd. Hlavním účelem tohoto vzoru je konverze rozhraní dané třídy na rozhraní, které očekává Client. Typické využití návrhového vzoru Adapter v aplikacích pro OS Android je konverze listu objektů pro zobrazení v *RecyclerView*. Příklad Adapteru pro *RecyclerView* můžeme vidět ve výpisu 1.4 [4, 5].

Výpis 1.4: Ukázka implementace návrhového vzoru Adapter v jazyku Kotlin.

```
1  //adapter mezi RecyclerView a třídou User
2  class UserAdapter(private val users: List<User>) : RecyclerView.
       Adapter<UserViewHolder>() {
3      override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
           : UserViewHolder {
4          val inflater = LayoutInflater.from(parent.context)
5          val view = inflater.inflate(R.layout.item_user, viewGroup,
               false)
6          return UserViewHolder(view)
7      }
8
9      override fun onBindViewHolder(vh: UserViewHolder, i: Int) {
10         vh.bind(users[i])
11     }
12
13     override fun getItemCount() = users.size
14 }
```

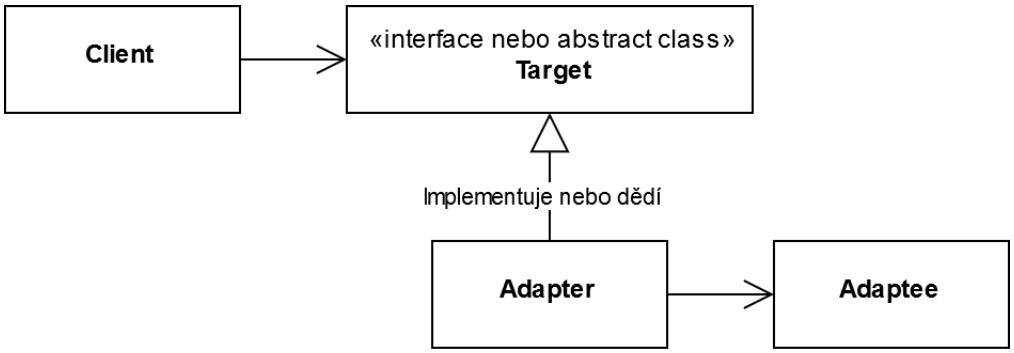
Adapter v tomto případě umožňuje spolupráci dvou nekompatibilních tříd *User* a *RecyclerView*. *RecyclerView* nemá žádnou informaci o existenci třídy *User*, jsou společně nekompatibilní, avšak díky návrhovému vzoru Adapter spolu mohou spolupracovat prostřednictvím tříd *UserAdapter* a *UserViewHolder* [5].

Komponenty návrhového vzoru Adapter

Návrhový vzor Adapter se skládá z těchto hlavních komponent:

- **Target** – Definuje interface, který využívá Client. Představitelem komponenty Target v příkladu výše jsou abstraktní třídy *RecyclerView.Adapter* a *RecyclerView.ViewHolder*, za pomocí kterých třída *RecyclerView* zobrazuje data.
- **Client** – Spolupracuje s objekty, které jsou v souladu s Target interface. V příkladu s *RecyclerView* zastupuje komponentu Client samotná třída *RecyclerView*.
- **Adaptee** – Definuje existující interface, který je nutné adaptovat. Zástupce Adaptee v příkladu výše je třída *User*, která obsahuje data k zobrazení v *RecyclerView*.
- **Adapter** – Adapter je třída, která konvertuje interface Adaptee na interface Target. Komponenta adapter je v příkladu výše zastoupena implementacemi *UserAdapter* a *UserViewHolder* [4].

Kompozice komponent v rámci návrhového vzoru Adapter je vidět na obr. 1.2.



Obr. 1.2: UML diagram návrhového vzoru Adapter [4].

1.2.2 Facade

Častým cílem správně navržené struktury zdrojového kódu je minimalizace závislostí mezi subsystémy stejně jako minimalizace vzájemné komunikace. Návrhový vzor Facade je jedním z možných prostředků, jak tohoto cíle dosáhnout [4].

Návrhový vzor Facade poskytuje jeden interface k souboru interfaců subsystémů. Poskytuje tak vyšší úroveň abstrakce pro vnější objekty, které tak nemusí znát vnitřní strukturu subsystému, díky čemuž je využívání subsystému snazší a odolnější vůči chybám [4].

Typickým příkladem využívání návrhového vzoru Facade je knihovna Retrofit, která definuje jednoduchý interface pro síťová volání viz. výpis 1.5 [4, 5].

Výpis 1.5: Ukázka návrhového vzoru Facade v jazyku Kotlin.

```
1 //deklarace interfacu pro vnější třídy. Subsystem zde představuje
2 //knihovna Retrofit + její konfigurace
3 interface UsersApi {
4     @GET("users")
5     fun listUsers(): Call<List<User>>
6 }
```

Klientský kód tak nemusí znát žádné podrobnosti systému jako například použitý JSON (JavaScript Object Notation) Parser, politiku kešování apod. V případě jakýchkoliv změn subsystému se tak změny nijak neprojeví v klientském kódu, kde by mohlo být potenciálně náročné změny provádět [5].

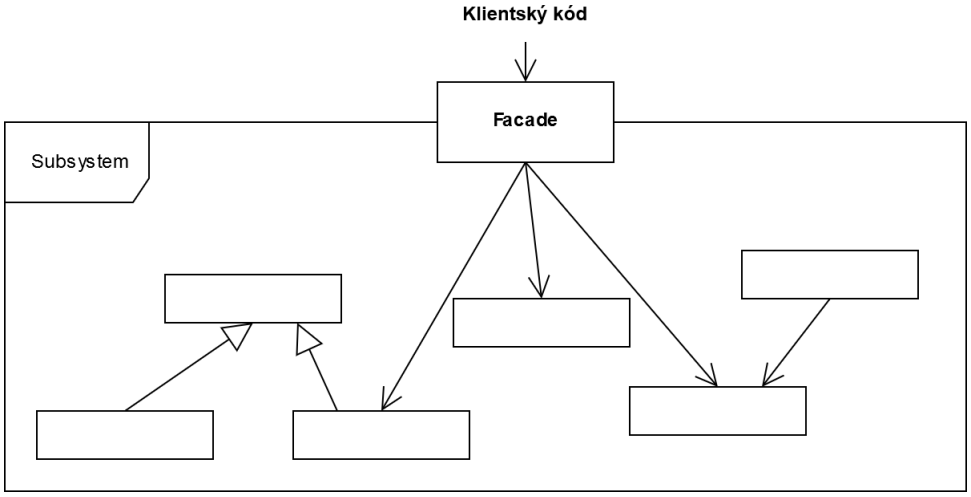
Komponenty návrhového vzoru Facade

Návrhový vzor Facade se skládá z těchto hlavních komponent:

- **Facade** – Poskytuje rozhraní k využívání subsystému. Komponenta Facade má znalosti o vnitřní struktuře subsystému a deleguje tak požadavky od klientského kódu na jednotlivé komponenty subsystému.

- **Třídy subsystému** – Tvoří funkcionalitu daného subsystému. Jejich hlavním cílem je splňovat požadavky zadané komponentou Facade. Třídy subsystému nemají žádné informace o ničem vně subsystému, a to včetně samotné komponenty Facade, na kterou nemají žádnou referenci [4].

Celkové schéma návrhového vzoru Facade je vidět na obr. 1.3.



Obr. 1.3: UML diagram návrhového vzoru Facade [4].

1.3 Behaviorální návrhové vzory

Behaviorální návrhové vzory řeší přidělení zodpovědností jednotlivým komponentám v aplikaci a zároveň charakterizují komunikaci mezi nimi. Díky behaviorálním návrhovým vzorům by měla být komunikace mezi komponenty jednoduchá, přesně definovaná a zároveň by komponenty neměly být navzájem těsně provázány. Mezi behaviorální návrhové vzory se řadí také architektonické návrhové vzory, které jsou ale popsány v samostatné kapitole 2. Velmi často používaným behaviorálním návrhovým vzorem v aplikacích pro OS Android je návrhový vzor Observer [4, 5].

1.3.1 Observer

Návrhový vzor Observer definuje vztah mezi objekty typu jeden ku mnoha tak, že pokud sledovaný objekt změní svůj stav, všechny objekty, které tento objekt sledují, jsou automaticky informovány o změnách jeho stavu. Návrhový vzor Observer se tak využívá všude tam, kde je skupina objektů závislá na stavu jiného objektu a je nutné zachovat jejich synchronizaci a současně nízkou provázanost. Návrhový vzor

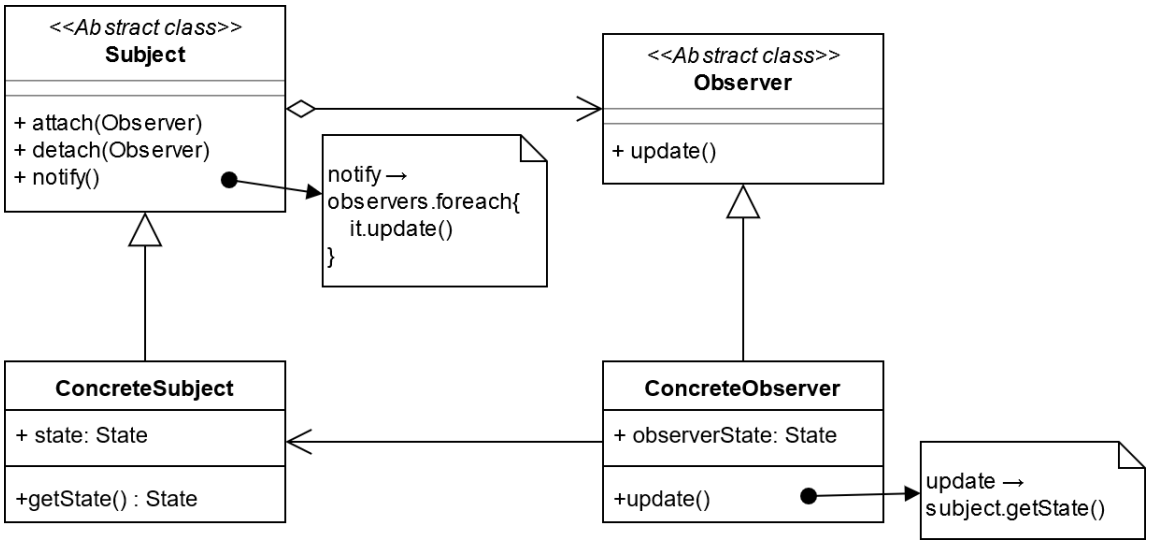
Observer v aplikacích pro OS Android implementuje například široce využívaná knihovna RxJava 2 [4].

Komponenty návrhového vzoru Observer

Hlavními komponenty návrhového vzoru Observer jsou:

- **Subject** – Má informace o všech přihlášených Observerech, poskytuje rozhraní k přihlášení a odhlášení komponent Observer.
- **Observer** – Definuje rozhraní k aktualizaci pro objekty, které byly notifikovány o změnách v Subjectu.
- **Concrete Subject** – obsahuje stav, který je sledován komponentami Concrete Observer. V případě změny stavu odesílá notifikaci Observerům prostřednictvím funkce notify() definované v Subjectu, sám ale nemá žádné informace o množství a typu Observerů přihlášených k odběru.
- **Concrete Observer** - Udržuje referenci na Concrete Subject. Obsahuje stav, který by měl být synchronizovaný se stavem v Concrete Subject. Synchronizaci udržuje implementací aktualizačního rozhraní, definovaného v komponentě Observer.

Schéma návrhového vzoru Observer je vidět na obr. 1.4.



Obr. 1.4: UML diagram návrhového vzoru Observer [4].

Výhody návrhového vzoru Observer

- Observer pattern snižuje provázanost mezi zdrojem (Concrete Subject) a příjemcem (Concrete Observer) informací. Díky tomu je tento návrhový vzor ideální pro komunikaci mezi různými vrstvami aplikace [4].

- Komponentě Subject nezáleží na tom, kolik Observeru jej observuje. K přihlášení dalších Observerů k odebírání stavu od Subjectu tak není potřeba nijak Subject upravovat [4].

2 Architektonické návrhové vzory

V případě rozsáhlejších aplikací je nutné brát v potaz nejen funkce aplikace, ale také architekturu jejího zdrojového kódu. V opačném případě se ve zdrojovém kódu začnou objevovat tzv. code smells, což je výraz popisující nevhodně napsaný a strukturovaný zdrojový kód. Takový kód nemusí nutně způsobovat chyby v programu, ale zvyšuje jejich pravděpodobnost, zhoršuje jejich opravitelnost a ztěžuje další rozvoj aplikace. Mezi významné zástupce code smells patří Spaghetti code, God classes a Blob classes [12, 13].

- **Spaghetti code** je výraz popisující komplikované a chaotické provázání jednotlivých funkcí, které má za následek nízkou přehlednost kódu. Typickými ukazateli Spaghetti code jsou třídy bez jasné struktury a dlouhé metody komunikující především prostřednictvím globálních proměnných [12, 14].
- **God class** je rozsáhlá třída, která nenásleduje princip jedné zodpovědnosti. God class tak typicky řeší větší množství úkolů namísto toho, aby je delegovala na další třídy [13].
- **Blob class** popisuje velmi rozsáhlou třídu s komplexní strukturou. Takové třídy mají mnoho zodpovědností a jsou těsně provázány s ostatními třídami [12, 13].

Správně zvolený návrhový vzor, podle něhož je architektura aplikace tvořena, napomáhá lepší přehlednosti zdrojového kódu s čímž je spojena vyšší produktivita vývojářů. Mezi další významné výhody využívání návrhových vzorů je lepší testovatelnost aplikace, snadnější opravitelnost v případě nalezení chyby, jednodušší rozšiřitelnost díky vyšší modularitě kódu a lepší výkonnost [15, 16].

Společným znakem návrhových vzorů bývá snaha o oddělení zodpovědností (SoC - Separation of Concerns) a snížení míry provázanosti kódu. V rámci principu separace odpovědností jsou jednotlivé problémy řešeny v přesně definovaných vrstvách aplikace, jejichž funkcionalita se nepřekrývá. Vrstvy spolu komunikují určitým stanoveným způsobem tak, aby byla provázanost tříd co nejmenší, což zajišťuje vyšší modularitu aplikace a snadnou úpravu či nahrazení jednotlivých modulů. Návrhové vzory rovněž podporují princip jedné zodpovědnosti, kdy každá třída má jediný přesně definovaný účel [13, 15, 16].

2.1 Architektury MV-X

V průběhu let se při vývoji aplikací pro OS Android začalo využívat několik architektonických návrhových vzorů, přičemž nejvíce využívanými zástupci jsou návrhové vzory typu MV-X, konkrétně MVP (Model-View-Presenter), MVVM (Model-View-

Viewmodel) a MVC (Model-View-Controller) [15, 16]. Návrhové vzory MV-X rozdělují architekturu aplikace na tři vrstvy.

První a zároveň nejvyšší vrstvou je grafické uživatelské rozhraní, které se ve všech architekturách shodně označuje jako View. V aplikacích pro OS Android je View reprezentováno XML (Extensible Markup Language) soubory popisujícími layout, a aktivitami, fragmenty a dialogy, které jsou na XML soubory layoutu navázány. Úkolem této vrstvy je zobrazovat data uživateli a zachytávat jeho interakci s aplikací [15, 16].

Naopak nejnižší datová vrstva aplikace, v architekturách MV-X shodně označována jako model, má za úkol obstarat data a nakládat s nimi. Právě v této vrstvě se realizuje přístup k úložištím, cloudovým službám apod. [15, 16].

V čem se architektury MV-X od sebe liší je prostřední logická vrstva a způsob komunikace mezi vrstvami.

2.1.1 MVC

Architektura MVC byla poprvé navržena už v roce 1979, přičemž se jedná o první využívanou architekturu v OS Android. Logickou vrstvu v architektuře MVC zastupuje Controller, který je vstupním bodem aplikace. Hlavními úkoly Controlleru je manipulace s View, základní logika aplikace, navigace a přenesení požadavků na data do modelu. Controller komunikuje s Modelem i View napřímo a tedy obsahuje jejich instance [17, 18].

Základní architektura aplikací připomíná právě návrhový vzor MVC, ve kterém ale chybí jasné oddělení Controlleru a View. View je představováno XML soubory a aktivitami/fragmenty, Controller je zastoupen aktivitami a fragmenty. Funkcionalita těchto dvou vrstev tak není jasně oddělená a prolíná se [17, 18].

Existuje debata zda takovou architekturu vůbec považovat za MVC. Někteří vývojáři proto používají oddělený Controller od aktivit a fragmentů, čímž lépe následují zásady architektury MVC. Takový Controller by navíc z důvodu lepší testovatelnosti neměl obsahovat žádné platformově specifické závislosti, tedy závislosti na Android třídách. Pro snížení provázanosti mezi View a Controllerem je vhodné používat interface [17, 18, 19].

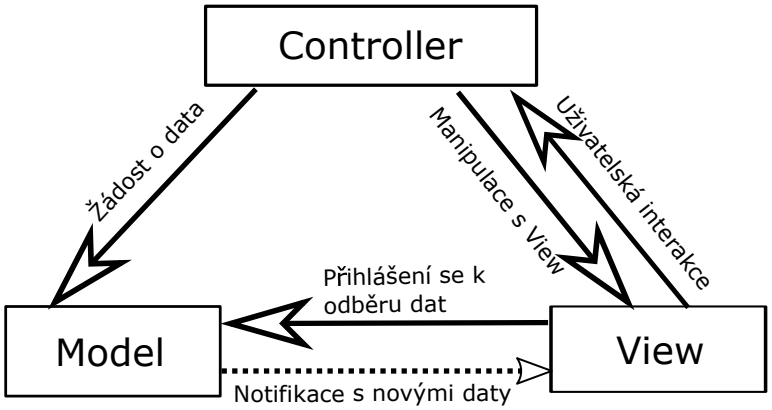
V architektuře MVC tvoří View a Controller těsně provázaný pár – ke každému View je definován právě jeden Controller. Ke Controlleru a View pak může být připojeno libovolné množství modelů. V rámci zajištění menší provázanosti by Model neměl mít referenci na Controller ani View, naopak jak View tak Controller mají referenci na Model [17, 18].

Příklad průběhu komunikace mezi vrstvami

Typická komunikace mezi komponenty v MVC probíhá následujícím způsobem:

1. View zachytí interakci od uživatele a požadavek přenese do Controlleru.
2. Controller manipuluje s Modelem v souladu s uživatelskou interakcí.
3. Model opatří data - obsahuje tedy aktuální data připravená k zobrazení.
4. View se přihlásí k odběru dat u Modelu prostřednictvím návrhového vzoru Observer.
5. Model View notifikuje při aktualizacích dat.
6. View data zachytí z modelu, transformuje je pro své potřeby a zobrazí uživateli [16, 15, 17, 18, 20].

Celkové schéma možné podoby architektury MVC můžeme vidět na obr. 2.1.



Obr. 2.1: Schéma komunikace v MVC [18].

Nevýhodou architektury MVC je, že View závisí jak na Controlleru, tak na Modelu. Jakákoliv změna ve View, tak může znamenat nutné úpravy ve více třídách, kvůli čemuž není architektura MVC tolik flexibilní jako MVP nebo MVVM. Druhým problémem je, že View získává data přímo od Modelu. View tak musí provádět logické operace na převedení dat do podoby, ve které je může zobrazit, což zvyšuje komplikovanost View a architektura MVC tak hůře následuje princip jediné zodpovědnosti. Alternativou by bylo, kdyby Model data upravil pro zobrazení ve View. V tomto případě se ale Model stává přímo závislý na podobě View, kvůli čemuž klesá jeho znovupoužitelnost [19].

2.1.2 MVP

Architektura MVP byla poprvé představena v roce 1996 v operačním systému Taligent [21]. MVP si v posledních letech vydobyl velkou oblibu u vývojářů a společně

s MVVM je nejoblíbenější architekturou využívanou pro vývoj aplikací pro OS Android [15].

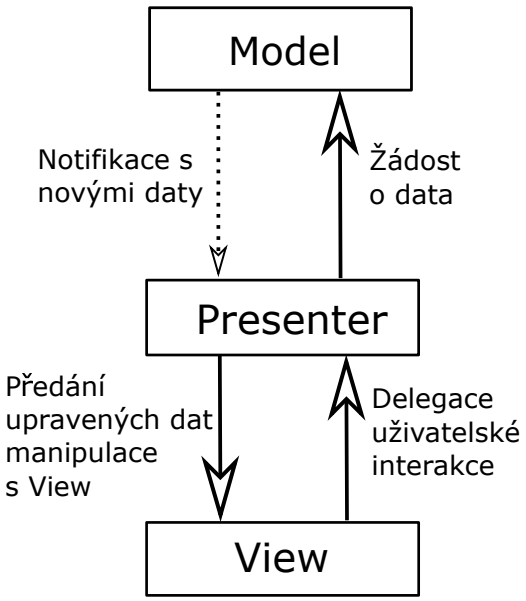
MVP vychází z architektury MVC, přičemž klade větší důraz na oddělení jednotlivých vrstev. MVP na rozdíl od MVC eliminuje přímé spojení mezi grafickou (View) a datovou vrstvou (Model). Veškerá interakce mezi těmito vrstvami probíhá prostřednictvím Presenteru, který nahrazuje Controller architektury MVC coby zástupce logické vrstvy [17].

Oproti Controlleru v architektuře MVC je Presenter více izolovaný, v aplikacích pro OS Android je Presenter vždy představován vlastní třídou, přičemž aktivity a fragmenty tak představují čistě grafickou vrstvu. Události v architektuře MVP (například kliknutí na tlačítko) jsou definované ve View a následně jsou delegovány Presenteru. Veškeré platformově specifické operace (například navigace mezi aktivitami) ale Presenter deleguje naopak na View [18].

Komunikace mezi Presenterem a View probíhá napřímo, kdy View i Presenter mají na sebe navzájem reference. Komunikace mezi Presenterem a Modelem může probíhat na přímo, kdy mají na sebe navzájem reference, případně prostřednictvím návrhového vzoru Observer, kdy Presenter požádá model o data a model data připraví do observovatelné proměnné, k jejímuž odběru je Presenter přihlášen [15, 16, 17, 18, 22].

Příklad průběhu komunikace mezi vrstvami

Celkové schéma možné podoby architektury MVP můžeme vidět na obr. 2.2.



Obr. 2.2: Schéma komunikace v MVP [16].

Běžná komunikace mezi vrstvami v architektuře MVP probíhá následujícím způsobem:

1. View zachytí interakci uživatele s aplikací.
2. Informaci o interakci předá Presenteru.
3. Presenter provede nezbytnou logiku a požádá Model o data, přičemž se přihlásí k odběru dat u observovatelného objektu Modelu.
4. Model data opatří, například z cloudové databáze a vloží je do observovatelného objektu.
5. Ve chvíli, kdy se data v observovatelném objektu změní, Presenter je notifikován, čímž tato data získá.
6. Následně je převede do podoby, kterou přímo využije View.
7. Zpracovaná data Presenter předá do View a to je zobrazí uživateli [15, 22].

Návrhový vzor MVP se dělí na dvě podkategorie: Supervising Controller a Pasive View. První varianta – Supervising Controller – počítá s tím, že jednoduché logické operace může vykonávat samo View, přičemž operace složitější deleguje Presenteru. Oproti tomu varianta Passive View počítá s tím, že veškerá logika je přenesena na Presenter, kdy View pouze zobrazuje data, aniž by obsahovalo jakoukoliv logiku. V aplikacích pro OS Android se spíše využívá varianta Passive View [17, 20].

Doporučení pro implementaci architektury MVP

Přestože neexistují přesně daná pravidla, jak architekturu MVP aplikovat, při vývoji aplikací na OS Android existují určitá doporučení a best practices, která zlepšují vlastnosti výsledného kódu [15, 17, 18].

- View by mělo obsahovat co nejméně logiky. Veškeré transformace dat by měly být provedeny v Presenteru, do View by se měla předávat data připravena rovnou k zobrazení [15].
- Presenter by neměl z důvodu lepší testovatelnosti obsahovat žádné závislosti na frameworku Android. Presenter by tak neměl přistupovat přímo například k resources, tento přístup by měl delegovat na Model či View [15].
- Modely by měly být ideálně navázány na životní cyklus aplikace, nikoliv aktivit a fragmentů. Presentery jsou svázány s životním cyklem aktivit/fragmentů, tím pádem nezachovávají svůj stav například při změně orientace obrazovky, kdy se aktivity restartují. K obnovení stavu zobrazení při těchto změnách orientace je tak vhodné využít právě Modely, které jsou napojeny na životní cyklus celé aplikace [15].
- View a Presenter by neměly k sobě navzájem přistupovat přímo, ale prostřednictvím tzv. kontraktů ve formě interfaců. Díky těmto kontraktům je zdrojový

kód aplikace srozumitelnější, neboť vztah View a Presenteru je explicitně dokumentován. Zároveň se tímto opatřením sníží provázanost těchto dvou komponent, což zlepšuje jejich modifikovatelnost [15, 17, 18].

- Životní cyklus Presenteru by měl sledovat životní cyklus přidruženého View (tj. aktivity/fragmentu), ale neměl by replikovat jeho složitost. Při restartování aktivity by Presenter neměl být zachován, nicméně neměl by obsahovat ani callbacky na změny v životním cyklu aktivit, neboť by to opět zvyšovalo jeho provázanost s View [15].
- Presentery mají tendenci stávat se tzv. God class. V zájmu následování principu jediné zodpovědnosti je vhodné, delegovat zodpovědnosti Presenteru na Modely a pomocné třídy [15].

2.1.3 MVVM

S návrhovým vzorem MVVM přišel v roce 2005 softwarový architekt společnosti Microsoft John Grossman. Jedná se o evoluci návrhové vzoru MVP. MVVM si klade za cíl další snížení provázanosti grafické a datové vrstvy. V současné době se jedná o jeden z nejvyužívanějších architektonických návrhových vzorů využívaných při vývoji mobilních aplikací pro OS Android. Použití architektury MVVM doporučuje sama společnost Google, přičemž využívání této architektury podporuje vývojem sady knihoven s názvem Architectural Components [1, 16, 18].

MVVM stejně jako MVP nemá přímé spojení mezi grafickou a datovou vrstvou, veškerá komunikace probíhá prostřednictvím logické vrstvy, která je představována ViewModelem [16, 18].

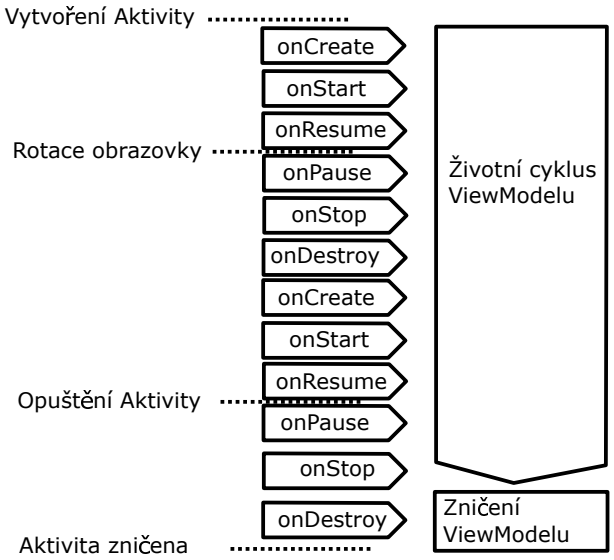
ViewModel má podobnou úlohu jako Presenter v architektuře MVP. Jeho hlavním úkolem je tak logika aplikace, delegace žádostí o data na modely, transformace dat pro potřeby View a ukládání UI-specifických dat. [1, 16, 18].

I přes vysokou podobnost existují mezi Presenterem v architektuře MVP a ViewModelem v architektuře MVVM zásadní odlišnosti. ViewModel na rozdíl od Presenteru nesmí mít na View žádnou referenci, v opačném případě by hrozily tzv. memory leaks (úniky paměti). Data jsou tak z ViewModelu do View předávána pomocí návrhového vzoru observer nebo pomocí databindingu. Zatímco u MVP tak Presenter přímo ovládal, co bude View zobrazovat, u MVVM ViewModel pouze vystaví dostupná data/události a View je samo odebírá [1, 18, 23].

V architektuře MVP tvořily View a Presenter pár jedna ku jedné, v architektuře MVVM může být k jednomu View připojeno více ViewModelů. ViewModely v aplikacích pro OS Android bývají vázány na aktivity, fragmenty nebo navigační grafy, jeden fragment tak může mít i 3 ViewModely [1, 23].

Další zásadní odlišností ViewModelu od Presenteru je životní cyklus. Přestože

obě komponenty jsou navázány na životní cyklus přidružených View, jejich chování je například při restartování aktivity odlišné. Zatímco Presentery při restartování aktivit zachovány nejsou, ViewModely tyto změny konfigurace View přežijí a lze je tedy využívat k obnovení stavu View. Životní cyklus ViewModelu je znázorněn na obr. 2.3 [1, 18].



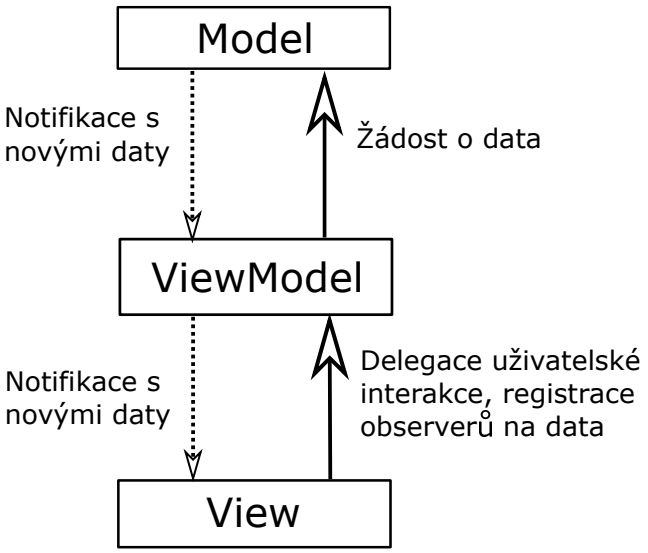
Obr. 2.3: Životní cyklus ViewModelu [1].

Příklad průběhu komunikace mezi vrstvami

Schéma a komunikační kanály architektury MVVM jsou zobrazeny na obr. 2.4.

Typická komunikace v rámci architektury MVVM v aplikacích pro OS Android probíhá následujícím způsobem:

1. View zachytí interakci od uživatele (například stisknutí tlačítka).
2. Informaci o interakci deleguje na ViewModel a zároveň se registruje k odběru souvisejících dat z observovatelného objektu ve ViewModelu.
3. ViewModel provede nezbytnou logiku a požádá Model o data, přičemž se přihlásí k odběru dat u observovatelného objektu Modelu.
4. Model data opatří, například z cloudové databáze a vloží je do observovatelného objektu.
5. Ve chvíli, kdy se data v observovatelném objektu změní, ViewModel je notifikován, čímž tato data získá.
6. ViewModel data následně transformuje do podoby vhodné pro View a uloží do observovatelného objektu.
7. Jakmile jsou data připravena, View je notifikováno s aktualizovanými daty a zobrazí je uživateli [1, 16, 22].



Obr. 2.4: Schéma komunikace v MVVM [16].

Doporučení pro implementaci architektury MVVM

Stejně jako u ostatních MV-X architektur neexistují přesně daná pravidla a různí vývojáři využívají různé modifikace návrhového vzoru MVVM. Existuje ale několik doporučení a best practices pro zlepšení kvality kódu využívající MVVM architekturu [15, 18].

- ViewModely by měly vystavovat stav, namísto jednotlivých dat nebo událostí. Například v případě, kdy potřebujeme zobrazovat položky jméno a věk ve View, data by měla být zapouzdřena do jednoho objektu, díky čemuž máme jistotu v zobrazení aktuálních a konzistentních informací [15, 23].
- ViewModely nepotřebují a ani nesmějí mít žádnou referenci na View. V opačném případě by ve chvíli restartování připojeného View (například aktivity) hrozily memory leaks. ViewModel tak vůbec neví o existenci View a pouze připravuje data, která je možné odebírat prostřednictvím návrhového vzoru Observer [15, 23].
- Veškeré logické operace, včetně těch nejjednodušších, by měly být delegovány z View na ViewModel, jediná úloha View by mělo být zobrazování dat [15, 23].
- Stejně jako Presentery v architektuře MVC i ViewModely mají tendenci stát se tzv. God class. Je tak vhodné přenášet zodpovědnosti na modely a pomocné třídy k zachování přehlednosti ViewModelu a následování principu jedné zodpovědnosti [15].
- ViewModely by měly obsahovat data o stavu, nikoliv data o zobrazení, a to jak z hlediska pojmenování, tak z hlediska typu [17].

2.2 Návrhový vzor Repository

Mnoho aplikací využívá více typu datových úložišť, například Shared Preferences pro jednoduchá data typu klíč-hodnota, Room SQL (Structured Query Language) databázi pro lokální ukládání složitějších dat a Firebase Cloud Firestore pro cloudově synchronizovaná data. Přímý přístup k těmto úložištím z logické vrstvy aplikace vede k mnoha problémům jako je duplikace kódu, vyšší riziko vzniku chyb, problémy s centralizací datové politiky (například kešování), nemožnost testování logické části aplikace v izolaci od datové vrstvy apod. [24].

Návrhový vzor Repository představuje rozšíření datové vrstvy – Modelu - architektonických návrhových vzorů MV-X. Hlavním cílem tohoto návrhového vzoru je centralizovaný přístup k datům aplikace, uložených v různých typech úložišť. Pro logickou vrstvu aplikace zajišťuje Repository abstrakci získávání dat. Jinými slovy, logická vrstva aplikace nemá žádné informace o tom, odkud a jakým způsobem byla žádaná data získána. Repository tak slouží jako prostředník mezi zdroji a konzumenty dat [1, 24, 25].

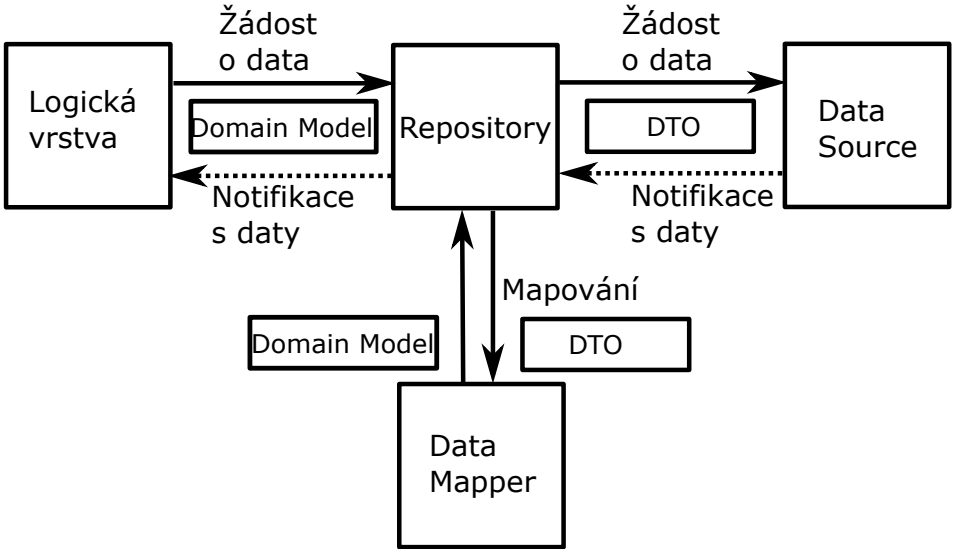
Díky vysoké abstrakci datové vrstvy se při unit testování logické vrstvy nemusí využívat konkrétní databáze, neboť může být snadno nahrazena zástupcem, který přístup k datům pouze simuluje. Silným důrazem na princip oddělení zodpovědností poskytuje návrhový vzor Repository lepší škálovatelnost a upravitelnost aplikace. Například změna databázového systému se projeví pouze v malé části kódu a nijak nezasáhne logickou ani grafickou část aplikace [1, 25].

Hlavní komponenty návrhového vzoru Repository jsou: Data Source, DTO (Data Transfer Object), Data Mapper, Domain Model a Repository [25].

- **Data Source** představuje třídu, která zajišťuje přístup ke konkrétnímu zdroji dat (Room, Shared Preferences, Cloud Firestore . . .).
- **DTO** je model dat v konkrétním úložišti, který je přímo přizpůsoben pro dané úložiště. Příkladem může být třída *RoomUserDTO*, která představuje data o uživateli uložená v SQL databázi Room, doplněná o veškeré potřebné anotace (označení primárního klíče, anotace entity apod.).
- **Domain Model** představuje model dat aplikace. Jedná se o datovou třídu, jejíž struktura je uzpůsobena aplikaci. Právě Domain Modely jsou využívány logickou a grafickou vrstvou.
- **Data Mapper** zajišťuje konverzi z Domain Modelu na DTO a naopak.
- **Repository** je třída, která zajišťuje centralizovaný přístup k jednomu typu dat, který je zastoupen Domain Modelem. Repozitory rozhoduje, kde se daná data opatří a jak se s nimi bude nakládat (uložení do lokální databáze, kešování. . .). Pro logickou vrstvu aplikace pak poskytuje jednoduché API (Application Programming Interface) pro snadné získávání dat [1, 24, 25].

2.2.1 Komunikace mezi komponenty

Schéma komponent v rámci návrhového vzoru Repository je vyobrazeno na obr. 2.5.



Obr. 2.5: Schéma návrhového vzoru Repository [25].

Typická komunikace mezi komponenty návrhového vzoru Repository může probíhat následujícím způsobem:

1. Logická vrstva požádá Repository o objekt s daným identifikátorem.
2. Repository provede vstupní logické operace, například rozhodne, ze kterého Data Source data získá a požadavek předá dále.
3. Data Source opatří data (například je stáhne z cloudového úložiště) a ve formě DTO je předá zpět Repository.
4. Repository pomocí komponenty Data Mapper transformuje DTO na Domain Model – tedy na formu, kterou žádá logická vrstva aplikace.
5. Následně může Repository nad daty provést další operace, například kešování dat, uložení do lokální databáze apod.
6. V posledním kroku Repository předá data zpět logické vrstvě, například pomocí návrhového vzoru Observer [1, 24, 25].

2.2.2 Doporučení a best practices pro návrhový vzor Repository

Stejně jako pro návrhové vzory MV-X i k návrhovému vzoru Repository existují určitá doporučení a best practices, která zaručí lepší kvalitu a robustnost výsledného kódu.

- Pro každý Data Source by měl existovat vlastní DTO. Pokud jsou například uživatelská data ukládána do databáze Room a do Cloudového úložiště Fires-

tore, měly by být vytvořeny pro data uživatele dva DTO: *RoomUserDTO* a *FirestoreUserDTO* [25].

- DTO a Domain Modely by se neměly slučovat ani v případě, že se jedná o identické třídy s identickými proměnnými. Díky tomu jsou data aplikace více oddělená od zvoleného databázového systému. V případě změny databázového systému pak stačí upravit, případně nahradit, pouze DTO, Data Source a DataMappery [25].
- Ke snížení provázaností by měla logická vrstva přistupovat k Repository přes interface, definující datové operace, jež jsou vyžadovány logickou vrstvou aplikace [25].
- Pro zajištění konzistentnosti dat by měla aplikace následovat princip SSoT(Single Source of Truth). Repozitory by tak mělo vracet data výhradně z jednoho Data Source, například lokální databáze, a ostatní Data Sources (například cloudová databáze) by měly pouze aktualizovat lokální databázi [1].
- Repository by mělo poskytovat pro logickou vrstvu co nejjednodušší API. Logická vrstva by neměla mít žádné informace o tom, jakým způsobem byla data opatřena [1].

2.3 Clean Architecture

Architektura Clean Architecture byla prezentována Robertem C. Martinem v roce 2012. Podobně jako u ostatních architektur je jejím hlavním cílem co nejlépe následovat princip separace zodpovědností. Na rozdíl od MV-X architektur ale kód nerozděluje čistě podle účelu, nýbrž podle míry abstrakce [27, 28, 29].

Vrstvy této architektury jsou znázorňovány jako soustředné kruhy. Vnitřní vrstvy představují pravidla a logiku využívanou v aplikaci. Vnější vrstvy pak představují mechanizmy a implementační detaily. Vnitřní vrstvy například mohou definovat logické operace popisující, jakým způsobem budou uživateli připisovány body. Vnější vrstvy pak mohou například definovat, v jakém databázovém systému budou uživatelské informace uloženy nebo grafické rozhraní, ve kterém budou uživatelské informace prezentovány [28, 29].

Clean Architecture nepředstavuje alternativu k návrhovým vzorům MV-X, nýbrž určitá implementační pravidla, jak návrhové vzory MV-X dále vylepšit. Clean Architecture je založená na těchto principech:

- Architektura aplikace by neměla záviset na použitém frameworku a knihovnách.
- Logika aplikace by měla být testovatelná v izolaci od datové i prezenční vrstvy.
- Uživatelské rozhraní by mělo být snadno vyměnitelné bez nutných změn ve zbytku aplikace.

- Databázové systémy by měly být snadno nahraditelné, logika aplikace by neměla být založena na použité databázové struktuře.
- Logická část aplikace by neměla vědět nic o implementačních detailech (grafické rozhraní, použité databáze . . .) [26, 28, 29].

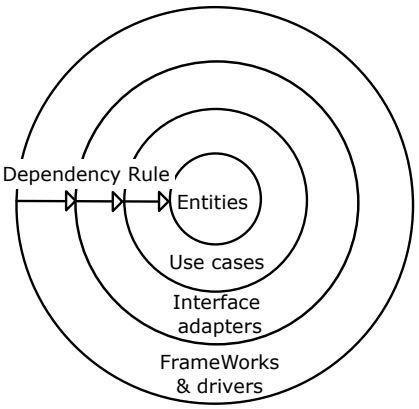
2.3.1 Dependency Rule

Hlavním pravidlem Clean Architecture je tzv. Dependency rule. Toto pravidlo říká, že závislosti zdrojového kódu mohou mířit pouze dovnitř. Jinými slovy jednotlivé vrstvy mohou obsahovat pouze závislosti z vnitřních vrstev a neměly by mít vůbec ponětí o existenci vrstev, které jsou k nim vnější. Změny ve vnějších vrstvách se pak nijak nepromítnou do vnitřních vrstev [28].

Jednotlivé vrstvy samozřejmě musí mít možnost předávat si mezi sebou obousměrně data, nicméně přímé předání dat z vnitřní vrstvy do vnější by bylo v rozporu s Dependency Rule. Komunikace mezi vrstvami tak může probíhat například pomocí návrhového vzoru Observer, kdy vnitřní vrstva pouze vystavuje observovatelné objekty, a vnější vrstva se přihlašuje k jejich odběru. Druhou možností předávání dat jsou interfacy, kdy si vnitřní vrstva definuje vstupní a výstupní interface (tzv. porty) a vnější vrstva pak pouze implementuje výstupní interface vnitřní vrstvy [27, 28, 29].

2.3.2 Struktura Clean Architecture

Clean Architecture se skládá ze čtyř základních vrstev: Entities, Use cases, Interface adapters, a Frameworks & drivers. U rozsáhlých programů mohou být jednotlivé vrstvy děleny dále na podvrstvy. Schéma Clean architecture je znázorněno na obr. 2.6 [28].



Obr. 2.6: Schéma Clean Architecture [28].

- **Entities** - Jedná se o samotné jádro aplikace obsahující základní logiku programu. Entities obsahují datové modely s logickými operacemi. V případě, že aplikace je součástí skupiny spolupracujících aplikací, vrstva Entities obsahuje veškerou společnou logiku těchto aplikací. Ze všech částí kódu mají Entity nejmenší pravděpodobnost nutnosti úprav a změn. Typickým zástupcem Entity může být objekt *User*, obsahující uživatelská data [26, 27, 28].
- **Use cases** - Tato vrstva provádí nad Entitami logické operace specifické pro konkrétní aplikaci. Objekty v rámci této vrstvy implementují v aplikaci různé scénáře využíváním Entit. Jakákoliv úprava v této vrstvě by neměla nijak ovlivnit vrstvu Entit. Obdobně by tato vrstva neměla být nijak ovlivněna změnami vnějších vrstev souvisejícími s databázemi, Frameworky, UI (User Interface) apod. Vrstva Use cases by tak měla být od těchto implementačních detailů kompletně izolována. Typickým zástupcem vrstvy Use cases může být například objekt *IncrementUsersXpUseCase*, který bude mít za úkol inkrementovat body v entitě *User* [26, 27, 28].
- **Interface adapters** - Jedná se o vrstvu, která má za úkol konvertovat data určená pro Use cases vrstvu na data určená pro UI. Typickými zástupci této vrstvy jsou Presentery (MVP) nebo ViewModely (MVVM). Kromě konverze dat pro UI zajišťuje tato vrstva rovněž konverzi dat pro datovou vrstvu aplikace. Do Interface adapters tak patří mimo jiné také Repository společně s třídami na konverzi dat (Data Mappers) [26, 27, 28].
- **Frameworks & drivers** - Jedná se o vnější vrstvu aplikace obsahující veškeré detaily (typ UI, typ použité databáze...). Do této vrstvy spadají veškeré třídy závislé na jakýchkoliv Frameworks. Patří sem mimo jiné i uživatelské rozhraní zastoupené aktivitami a fragmenty a implementační detaily databází zastoupené komponenty typu Data Source. Podobně jako tomu bylo u vnitřních vrstev, jakákoliv změna v této vrstvě by se neměla nijak promítnout do vrstev vnitřních [26, 28].

2.3.3 Implementace Clean Architecture v OS Android

Clean Architecture nebyla vytvořena specificky pro OS Android, ale jako obecná implementační doporučení pro veškerý software [28]. V následujících odstavcích tak bude popsána jedna z možných implementací Clean Architecture v OS Android ve spojení s MVVM a Repository Pattern.

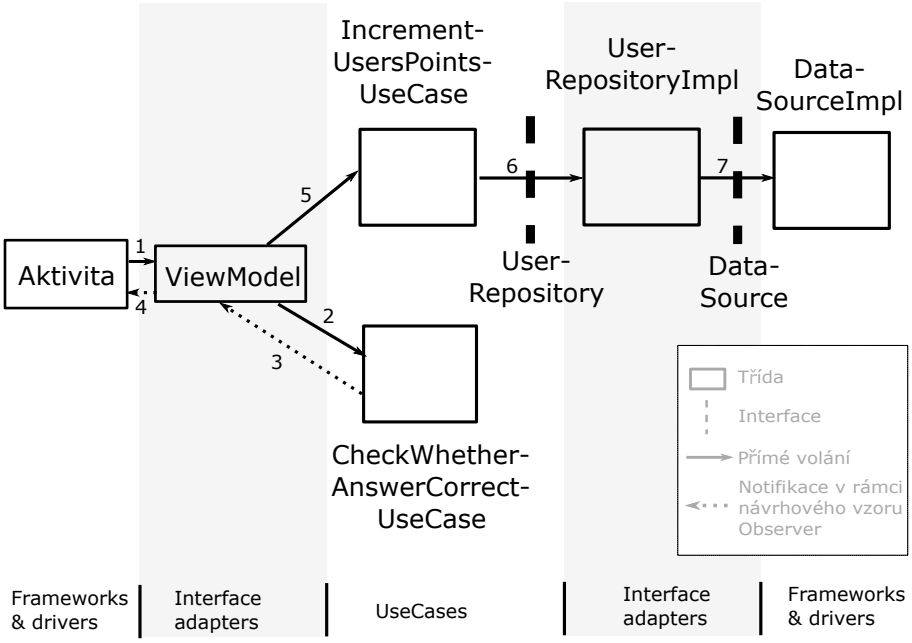
Mějme scénář, kdy uživatel odpoví na otázku v aplikaci, a měl by dostat body. Typická komunikace mezi vrstvami v rámci Clean Architecture by pak mohla probíhat následujícím způsobem:

1. Uživatel klikne na tlačítko „potvrdit odpověď“. Aktivita (vrstva Frameworks &

drivers) požadavek zachytí a spolu s označenou odpovědí požadavek deleguje na ViewModel (vrstva Interface adapters).

2. ViewModel následně zavolá objekt *CheckWhetherAnswerCorrectUseCase* (vrstva Use cases), který mu má navrátit správnost odpovědi.
3. *CheckWhetherAnswerCorrectUseCase* poté navrátí správnost odpovědi. Podle Dependency Rule ale Use case nemůže mít žádnou referenci na ViewModel, nemůže mu tak výsledek vratit přímo. Správnost odpovědi tak ViewModelu může předat například prostřednictvím návrhového vzoru Observer (případně pomocí interfaců).
4. ViewModel následně předá View pomocí observovatelného objektu požadavek na zobrazení zprávy pro uživatele, jak odpověděl.
5. V případě správné odpovědi následně ViewModel vyvolá *IncrementUsersPointsUseCase*.
6. *IncrementUsersPointsUseCase* obsahuje závislost na interface *UserRepository* (součást vrstvy UseCases), který je implementován třídou *UserRepositoryImpl* (vrstva Interface adapters). *IncrementUsersPointsUseCase* provede potřebné logické operace nad entitou *User* (vrstva Entitites) a aktualizovanou entitu *User* předá *UserRepository* k uložení.
7. Třída *UserRepositoryImpl* následně rozhodne, kam entitu *User* uloží a transformovanou Entitu předá prostřednictvím Interfacu konkrétnímu DataSource (vrstva Frameworks & drivers) k uložení [26, 27, 30].

Celá komunikace je znázorněna na obr. 2.7.



Obr. 2.7: Příklad průběhu komunikace mezi vrstvami v rámci Clean Architecture.

2.3.4 Best practices

Kromě pevně daných pravidel tvořících jádro Clean Architecture (Dependency Rule, princip separace zodpovědností) existují další doporučení a best practices které dále zvýrazňují výhody, které Clean Architecture přináší.

- V Aplikacích pro OS Android by UseCases neměly být vykonávány na hlavním vlákně [26].
- UseCases by neměly obsahovat žádná specifická data k jednomu typu databáze (DTO dané databáze). Obecně při komunikaci mezi vnější a vnitřní vrstvou by předávaná data měla být ve formě nejvíce vyhovující pro vnitřní vrstvu [28].
- V zájmu zvýšení čitelnosti kódu by jednotlivé komponenty, především Use cases, měly být pojmenovány v souladu s funkcí, kterou vykonávají [29].

2.3.5 Výhody Clean Architecture

Využívání Clean Architecture při vývoji softwaru s sebou přináší následující výhody:

- Díky vysoké míře separace zodpovědností a nízké provázanosti komponent je aplikace snadno testovatelná, opravitelná a rozšiřitelná [28, 29, 30].
- Jádro aplikace není nijak závislé na UI, logickou vrstvu tak lze přenášet mezi platformami (Android, Web apod.) [28, 30].
- Aplikace je nezávislá na zvoleném databázovém systému, v případě potřeby tak lze databázový systém snadno vyměnit za jiný [28, 29].
- Vzhledem k vysoké míře separace zodpovědností a pevně dané struktuře programu je kód aplikace přehlednější a snadněji pochopitelný [29].
- Všechny výše zmíněné body vedou k vyšší produktivitě vývojářů a s tím souvisejícím nižším nákladům na vývoj [29].

3 Kotlin

Kotlin je moderní, staticky typovaný jazyk podporující jak objektové, tak funkcionální programování. Vyvíjen je od roku 2010 společností JetBrains, přičemž v současné době je jeho další rozvoj řízen skupinou Kotlin Foundation tvořenou společnostmi JetBrains a Google [31].

Hlavním cílem při vývoji bylo vytvořit nový moderní jazyk se stručnou syntaxí, který by byl zcela interoperabilní s programovacím jazykem Java. Při vývoji se brala inspirace z mnoha současných programovacích jazyků včetně Javy, C#, Scaly a dalších s cílem z každého jazyka vybrat nejlepší charakteristiky a aplikovat je v Kotlinu. Přestože je Kotlin vyvíjen primárně pro JVM (Java Virtual Machine), je možné jej kompilovat do JavaScriptu nebo přímo do strojového kódu [31, 32].

První stabilní verze jazyka Kotlin byla vydána v roce 2016, přičemž v polovině roku 2017 se Kotlin stal vedle Javy a C++ oficiálním programovacím jazykem pro vývoj aplikací na OS Android [1, 31, 32].

3.1 Obecné vlastnosti jazyka

- **Staticky typovaný** – Typy všech proměnných jsou známy již v době kompilace. Výhodou statického typování je lepší detekce chyb při kompilaci, vyšší výkon, zvýšená čitelnost kódu a možnost rozsáhlejší podpory nástrojů programovacího prostředí (spolehlivé refaktorování, lepší možnosti automatického doplňování kódu aj.) [31].
- **Podpora funkcionálního paradigmatu** – S funkcemi (částmi chování) se pracuje jako s hodnotami. Lze je ukládat do proměnných, předávat jako parametry jiným funkcím, nebo je z jiných funkcí navracet [31].
- **Stručná syntaxe** - při vývoji jazyka Kotlin bylo dbáno na co největší redukci tzv. „boilerplate“ kódu, k čemuž napomáhá také rozsáhlá standardní knihovna, která ulehčuje běžnou práci s kolekcemi, textovými řetězci a dalšími strukturami [31].
- **Open source** - Vývoj jazyka probíhá ve veřejném repozitáři na GitHubu. Hlavním vývojářem jazyka je tým společnosti JetBrains pod vedením Andreje Breslava, ale k vývoji přispívají i vývojáři ze společnosti Google a veřejná komunita programátorů. Využití jazyka je volné a bezplatné pro jakékoliv využití pod licencí Apache v2 [33].

3.2 Srovnání jazyků Kotlin a Java

Jak již bylo řečeno, Kotlin je zcela interoperabilní s jazykem Java, což znamená, že z kódu napsaného v Javě je možné volat kód napsaný v Kotlinu a naopak. Díky tomu je možné v jednom projektu kombinovat oba jazyky. Kotlin ale navíc přináší několik dalších vlastností, které v Javě chybí. V následujících podkapitolách budou některé z těchto funkcí popsány [31].

3.2.1 Proměnné a konstruktory

Mějme v Javě definovanou datovou třídu obsahující pouze proměnné. Tyto proměnné jsou inicializovány v konstruktoru vstupními parametry a přistupuje se k nim pomocí getterů a setterů [31]. Příklad takové třídy můžeme vidět ve výpisu 3.1.

Výpis 3.1: Příklad jednoduché datové třídy definované v jazyku Java.

```
1  class User {
2      //definice proměnných
3      private String name;
4      private int age;
5
6      //konstruktor
7      public User(String name, int age) {
8
9          //inicializace proměnných
10         this.name = name;
11         this.age = age;
12     }
13
14     //gettery a settery
15     public String getName() {
16         return name;
17     }
18
19     private void setName(String name) {
20         this.name = name;
21     }
22
23     public int getAge() {
24         return age;
25     }
26
27     public void setAge(int age) {
28         this.age = age;
29     }
30 }
```

Pro vytvoření jednoduché datové třídy jsme museli napsat velké množství boilerplate kódu, který snižuje produktivitu a čitelnost kódu. Ekvivalentní třída v Kotlinu by vypadala následovně (viz výpis 3.2).

Výpis 3.2: Příklad jednoduché datové třídy definované v jazyku Kotlin.

```
1
2  /*
3      Konstruktor kombinuje definici a inicializaci
4      proměnných.
5  */
6  class User(name: String, var age: Int) {
7
8      /*
9      Vně konstruktoru definujeme proměnné
10     s asymetrickým přístupem.
11     */
12     var name: String? = null
13         private set
14
15     //tělo konstruktoru
16     init {
17         this.name = name
18     }
19 }
```

V případě, že vstupním parametrům v konstruktoru přidružíme klíčové slovo var (proměnná) nebo val (hodnota), stanou se tyto parametry plnohodnotnými proměnnými/hodnotami dané třídy. Díky tomu proměnné, které nemají asymetricky omezený přístup, nemusíme definovat mimo konstruktor, abychom je v konstruktoru pak museli následně inicializovat [31].

Zároveň není nutné pro každou proměnnou definovat gettery a settery. Defaultně je proměnná přístupná k zápisu i ke čtení, pokud chceme zakázat zápis zvenčí, nastavíme u proměnné privátní setter. Podobně můžeme nastavit privátní getter v případě, že chceme zakázat čtení z venčí [31].

Standardně nemají konstruktory v Kotlinu žádné tělo. Pokud potřebujeme provést v konstruktoru nějakou inicializaci, provádíme ji v bloku s klíčovým slovem init [31].

V případě, že by u předchozí ukázky nebyl definován asymetrický přístup k proměnné age, zdrojový kód by se zredukoval dokonce na jediný řádek (výpis 3.3).

Výpis 3.3: Příklad jednořádkového definování datové třídy v jazyce Kotlin.

```
1
2 /*
3 Konstruktor kombinuje definici a inicializaci
4 proměnných. Proměnné var v sobě automaticky
5 obsahují defaultní getter a setter.
6 */
7 class User(var name: String, var age: Int)
```

3.2.2 Null safety

Kotlin byl navržen tak, aby co nejvíce zredukoval riziko *NullPointerException*, která vzniká v případě, kdy se aplikace snaží přistupovat k objektu s hodnotou null. Typovací systém Kotlinu rozlišuje typy, které mohou nabývat hodnoty null (syntakticky vyjádřeno doplněním otazníku za typ proměnné) od těch, které hodnoty null nabývat nemohou [31] (viz výpis 3.4).

Výpis 3.4: Ukázka definice nullable a nonnullable proměnných.

```
1 //Otazník značí, že proměnná může být null
2 var nullableString: String?
3
4 //Tato proměnná nemůže být null
5 var nonNullableString: String = ""
```

Kotlin zároveň poskytuje prostředky pro nakládání s hodnotami, které mohou být null. V případě že přistupujeme k objektu, který může nabývat hodnoty null, musíme využít buď not-null assertion operator **!!** nebo safe-call operator **.?** [31, 9] viz výpis 3.5.

Výpis 3.5: Ukázka využití notnull assertion operator a safecall operátor.

```
1 //definice třídy User s nullable proměnnými
2 data class User(var name: String?, var age: Int?)
3 fun someFunction(user: User){
4     //Safe-call operator
5     user.name?.trim()  //safe-call operator
6
7     //Not-null assertion operator
8     user.age!!++
9 }
```

V případě, že využijeme safe-call operátor na proměnné *name* s hodnotou null, příkaz trim() se jednoduše neprovede, v případě využití not-null assertion operatoru na proměnné *age* s hodnotou null program spadne s NullPointerException [31].

3.2.3 Pokročilé funkce kompilátoru jazyka Kotlin

Type Inference

Přestože je Kotlin staticky typovaný jazyk, nemusíme u většiny proměnných explicitně uvádět jejich typ. Kompilátor Kotlinu totiž dokáže v mnoha případech typ odvodit z kontextu, čímž se výsledný zdrojový kód zestruční [31].

Kompilátor Kotlinu rovněž sleduje stav uvnitř výrazu (například uvnitř podmínky if), přičemž v případě, že z kontextu vyplývá že nullable hodnota nemůže nabýt uvnitř výrazu hodnoty null, chová se k ní kompilátor jako k non-nullable proměnné [31]. Příklad funkce Type Inference je vidět na ukázce 3.6.

Výpis 3.6: Ukázka funkce Type Inference.

```
1  /*
2  Kompilátor z kontextu usoudil, že
3  se jedná o proměnnou typu String
4  */
5  val greetings = "Hello!"
6
7  fun someFunction(nullableString: String?){
8      if(nullableString != null){
9          /*
10         Kompilátor z kontextu pochopil, že
11         proměnná nemůže být null a můžeme tak
12         k ní přistupovat bez safe-call operátoru
13         */
14         nullableString.lenght()
15     }
16 }
```

Inteligentní přetypování

Funkce inteligentní přetypování (Smart Cast) kombinuje kontrolu typu a přetypování do jedné operace. Pokud je ve zdrojovém kódu provedena na některé proměnné kontrola typu, kompilátor proměnnou automaticky přetypuje na daný typ. Funkce Smart Cast ale funguje pouze v případě, kdy kompilátor může zaručit, že proměnná nemohla být po kotrole typu změněna [31]. Příklad inteligentního přetypování je vidět na ukázce 3.7.

Výpis 3.7: Ukázka funkce Smart Cast.

```
1
2  fun someFunction(obj: Any){
3      if(obj is String){
4          /*
5          Kompilátor po kotrole typu provedl
6          přetypování proměnné obj na
7          typ String za nás.
8          */
9          obj.lenght()
10     }
11 }
```

3.2.4 Extension functions

Extension function jsou funkce, které mohou být volány v rámci dané třídy, přestože definovány byly vně této třídy. Jinými slovy, pomocí extension function můžeme rozšířit chování třídy, aniž bychom ji museli upravovat nebo vůbec mít přístup k jejímu zdrojovému kódu [31, 9], viz výpis 3.8.

Výpis 3.8: Ukázka extension function.

```
1
2  // Definice extension function ke třídě String
3  fun String.thirdChar(): Char?{
4      return if (this.length < 3) null
5      else this[2]
6  }
7  fun someFunction(){
8      /*
9      Extension function je volána stejně, jako jakákoliv
10     jiná metoda z třídy String
11     */
12     println("Diplomová práce ".thirdChar())
13 }
```

Na rozdíl od klasických metod nemohou extension functions přistupovat k metodám a proměnným s viditelností private nebo protected. Díky tomuto omezení tak neporušují princip zapouzdření. V místě volání jsou extension functions nerozeznatelné od plnohodnotných metod dané třídy, nicméně pro jejich využití je musíme importovat zvlášť [31].

Extension functions mohou být vytvořeny na libovolných třídách napsaných v jakémkoliv JVM jazyce (Kotlin, Java, Groovy...). Přestože se jedná o funkcialitu jazyka Kotlin, extension function je možné volat i z Javy, neboť kompilovány jsou

jako statické metody, jenž jako první argument berou přijímací objekt rozšiřované třídy [31], viz výpis 3.9.

Výpis 3.9: Ukázka volání extension function ze zdrojového kódu v jazyce Java.

```
1
2 private void someFunction(){
3     System.out.print(Utils.thirdChar("Diplomová práce"));
4 }
```

Dalším rozdílem oproti klasickým metodám je, že extension functions nelze overridovat. Tato nemožnost vyplývá z jejich statické podstaty [31].

3.2.5 Scope functions

Standardní knihovna jazyka Kotlin poskytuje celkem 5 scope functions (let, apply, with, also a run), jejichž úlohou je provedení bloku kódu v kontextu objektu. Funkce vytvářejí dočasný kontext, ve kterém můžeme k danému objektu přistupovat bez jeho jména. Hlavním benefitem scope functions je zvýšení čitelnosti a zestručnění kódu [31, 9].

Scope function můžeme dělit podle dvou kritérií:

1) Dělení podle způsobu jakým se odkazují na přijímací objekt

Funkce **apply**, **run** a **with** se odkazují na přijímací objekt jako na přijímač lambda funkce prostřednictvím klíčového slova *this*, podobně jak tomu bývá uvnitř objektů. Stejně jako je tomu u obyčejných objektů i zde je možné ve většině případů klíčové slovo *this* vynechat [9].

Funkce **let** a **also** se na přijímající objekt odkazují pomocí argumentu lambda funkce, jehož jméno můžeme buď stanovit, nebo k němu přistupovat pomocí klíčového slova it [9].

2) Dělení podle návratové hodnoty

Návratová hodnota funkcí **apply** a **also** je přímo přijímací objekt na kterém byly volány. S výhodou se tak dají využít při nastavování hodnot daného objektu [9].

Scope functions **let**, **run** a **with** navracejí hodnotu lambda funkce. Tyto funkce se využívají například v kombinaci se safe-call operátorem na vykonání bloku kódu v případě, že konkrétní hodnota není null. Z rozdělení scope funkcí vyplývá, že funkce **run** a **with** spadají do stejné kategorie, rozdíl mezi nimi je ale ten, že **with** jako jediná ze scope funkcí není definována jako extension function [9].

Ve výpisu 3.10 je demonstrováno využití scope functions let a apply.

Výpis 3.10: Ukázka využití Scope functions let a apply.

```
fun someFunction(someString: String?){
/*Blok kódu lambda funkce je proveden
pouze v případě, že proměnná someString není null
*/
someString?.let{println(it)}

/*Vytvoření objektu user a následné nastavení
jeho proměnných pomocí funkce apply
*/
val user = User().apply{
    name = "Jan"
    age = 30
    }
}
```

3.2.6 Top level funkce a top level proměnné

Programovací jazyk Java vyžaduje deklaraci veškerých funkcí a proměnných uvnitř objektů. To ale často vede ke vzniku mnoha zbytečných tříd obsahujících pouze statické metody a proměnné [31].

Kotlin tento problém řeší pomocí top-level funkcí a top-level proměnných. V Kotlinu je tak možné vytvářet proměnné a funkce vně tříd, čímž se redukuje stupeň zanoření a zvýší se stručnost a přehlednost zdrojového kódu [31].

Vzhledem k tomu, že JVM umí zpracovávat pouze kód uvnitř tříd, kompilátor Kotlinu top-level funkce a top-level proměnné zkompiluje do tříd se stejným názvem, jako je název souboru, ve kterém byly definovány. Takto je možné k těmto metodám a proměnným přistupovat z jazyka Java [31].

3.2.7 Defaultní hodnoty funkcí a jmenné argumenty

Některé třídy v Javě obsahují velké množství konstruktorů a přetížených metod (metody se stejným jménem ale rozdílnými vstupními parametry). To ale vede k opakování kódu a menší přehlednosti. Kotlin tento problém řeší zavedením defaultních hodnot a jmennými argumenty [31].

Ke každé hodnotě v konstruktoru nebo metodě tak můžeme stanovit defaultní hodnotu a tím pádem můžeme tuto hodnotu při volání funkce vynechat. Standardní syntaxí lze takto volat funkce s parametry ve stejném pořadí, jak byly definovány a vynechat je možné pouze poslední argumenty. Tento problém řeší jmenné argumenty. Díky nim, můžeme vynechat jakékoliv argumenty s defaultní hodnotou, navíc můžeme argumenty funkci předávat v libovolném pořadí [31], viz výpis 3.11.

Výpis 3.11: Ukázka využití Scope functions let a apply.

```
1
2  fun printMessage(message: String,
3                   author: String = "Unknown",
4                   place: String = "Unknown"){
5      println("Record: $message, author: $author, place: $place")
6  }
7
8  fun someFunction(){
9      printMessage("Zpráva 1")
10     printMessage("Zpráva 2", place = "Brno", author = "Jan")
11 }
```

Protože Java defaultní hodnoty ani jmenné argumenty nepodporuje, musíme při volání funkce s defaultními hodnotami z Javy zadat všechny parametry, případně můžeme pomocí anotace @Jvm-Overloads instruovat kompilátor kotlinu, aby vygeneroval přetížené metody tím stylem, že postupně od konce vynechává poslední argumenty s tím, že vynechaným proměnným nastavuje hodnoty definované v defaultním parametru [31].

4 Dependency Injection

Objekty aplikací často potřebují ke své činnosti funkcionalitu objektů jiných tříd. Pokud objekt třídy A využívá funkcionalitu objektu třídy B, říkáme, že třída A má závislost na třídě B. Existují celkem tři možnosti, jak může objekt třídy A získat objekt třídy B:

1. Vytvořit si objekt třídy B. Tato varianta zvyšuje provázanost zdrojového kódu - v případě změny konstruktoru třídy B bude potřeba upravit všechny výskyty tvorby objektu třídy B.
2. Požádat o objekt B jiný objekt, například Factory, viz kapitola 1.1.2.
3. Dostat objekt B zvenčí například prostřednictvím konstruktoru nebo setteru. Tato možnost se nazývá Dependency Injection [1, 34].

4.1 Obecný popis

Dependency Injection (DI) je soubor návrhových principů a vzorů, které umožňují vytvářet zdrojový kód s nízkou mírou provázanosti, díky čemuž jsou aplikace a programy snadněji opravitelné a rozšířitelné [35].

DI je založeno na principu Inverze Kontroly (IoC - Inversion of control), který říká, že obecný kód má řídit specifickou implementaci, nikoliv naopak. Základní myšlenkou Dependency Injection tak je, že spolupracující třídy by měly při získávání závislostí spoléhat na danou infrastrukturu (aplikaci, DI knihovnu...). Třídy by tak neměly svoje závislosti vytvářet, ani o ně explicitně žádat. Všechny potřebné závislosti by jim měly být poskytnuty z vnějšku, většinou za pomoci jedné z Dependency Injection knihoven [35, 36].

Nejdůležitějším aspektem, který Dependency Injection řeší, je kompozice tříd vytvořením objektového grafu, neboli sestavení tříd do potřebného celku (programu, aplikace, apod.). Kromě toho spravuje Dependency Injection také životní cyklus veškerých vytvořených objektů - tedy kdy mají být vytvořeny a kdy mají zaniknout. Posledním aspektem, který má Dependency Injection na starost, je modifikace vytvořených objektů před tím, než jsou předány klientskému objektu [35].

4.1.1 Typy Dependency Injection

Existuje několik typů Dependency Injection, které se dělí podle toho, jakým způsobem jsou závislosti klientské třídě předány.

Constructor Injection

Constructor Injection je forma DI, kdy jsou potřebné závislosti objektu předány hned při jeho vytvoření prostřednictvím konstruktoru. Třída, která potřebuje dané závislosti, musí mít veřejný konstruktor, přičemž by se mělo jednat o jediný dostupný konstruktor. Constructor Injection zaručuje, že daný objekt bude mít závislosti k dispozici ihned po jeho vytvoření a měl by to tak být hlavní využívaný typ DI [1, 35]. Jednoduchý příklad manuálního Constructor Injection je zobrazen ve výpisu 4.1. Třída Test si zde závislost Timer nevytváří sama, ale spoléhá na to, že jí bude dodána v konstruktoru.

Výpis 4.1: Příklad jednoduchého manuálního Constructor Injection.

```
1  class Test(private val timer: Timer){
2      fun startTest(){
3          timer.beginCountDown()
4      }
5  }
6
7  fun main() {
8      val timer: Timer = Timer()
9      val test = Test(timer)
10     test.startTest()
11 }
```

Field Injection

Field Injection je forma DI, kdy jsou závislosti objektu předány přímo do veřejných proměnných. Tato forma DI se používá tehdy, pokud není možné vytvořit potřebný konstruktor, případně pokud má třída k dispozici lokální náhradu za závislost [35].

Nevýhoda tohoto typu DI je v tom, že není explicitně zaručeno, že daný objekt bude mít v potřebný čas závislost k dispozici [35]. Jednoduchý příklad manuálního Field Injection je vidět na výpisu 4.2, kde je závislost Timer dodána třídě Test inicializací veřejné proměnné z vnějšku.

Výpis 4.2: Příklad jednoduchého manuálního Field Injection.

```
class Test(){
    var timer: Timer

    fun startTest(){
        timer.beginCountDown()
    }
}

fun main() {
    val timer: Timer = Timer()
    val test = Test()
    test.timer = timer
    test.startTest()
}
```

Method Injection

V rámci Method Injection jsou závislosti objektu předávány prostřednictvím parametru metody. Tento typ DI se využívá především v případě, kdy se typ závislosti může měnit s každým voláním metody [35]. Jednoduchý případ manuálního Method Injection je vidět na výpisu 4.3, kde je závislost Timer třídě Test dodána jako argument funkce.

Výpis 4.3: Příklad jednoduchého manuálního Method Injection.

```
class Test(){

    fun startTest(timer: Timer){
        timer.beginCountDown()
    }
}

fun main(args: Array) {
    val timer: Timer = InfiniteTimer()
    val test = Test()
    test.startTest(timer)
}
```

4.1.2 DI Kontejner

Dependency Injection se většinou neprovádí manuálně, ale pomocí knihoven (například Dagger 2, Guice, Spring...) označovaných jako DI Kontejnery. Tyto knihovny dokážou automatizovat mnoho úkonů spojených s tvorbou objektů a správou jejich

životního cyklu. Hlavním úkolem těchto knihoven je vytvoření a spravování objektového grafu, který popisuje vztahy mezi objekty. Tvorbu objektů ale často nelze plně automatizovat a je tak potřeba DI kontejnery konfigurovat. Veškeré konfigurace jsou pouze deklarativní. Jinými slovy programátor v nich pouze stanoví co potřebuje, ale nevytváří žádné logické konstrukce. [35].

Konfigurace může probíhat přímo ve zdrojovém kódu, prostřednictvím XML souborů, nebo auto-registrací. Výhodou konfigurace v kódu je, že Dependency Injection je zkontrolováno přímo za kompilace a nenastávají tak chyby za běhu programu. Další možností je konfigurace pomocí XML souborů. Ta má výhodu v tom, že lze změnit mapování objektů bez toho, aniž by bylo potřeba program opětovně kompilovat. Nevýhoda konfigurace pomocí XML je v tom, že případné chyby se budou často projevovat až za běhu programu [35].

Poslední možností je konfigurace auto-registrací, kdy knihovna rekurzivně vytváří objektové grafy pomocí obsahu konstruktorů. Pokud jsou v daném konstruktoru závislosti, knihovna rekurzivně vytvoří nejprve tyto závislosti a jakmile má vše potřebné vytvoří samotný objekt. Poslední jmenované řešení z velké části automatizuje konfiguraci DI Kontejneru, nicméně podporuje pouze Constructor Injection [35].

Obecně se doporučuje využívat v první řadě auto-registraci doplněnou o konfiguraci pomocí kódu. Konfigurace pomocí XML je vhodná pouze tehdy, pokud je potřeba měnit mapování na konkrétní závislosti bez opětovné rekompilace programu [35].

4.1.3 Výhody využívání Dependency Injection

Dependency Injection je jedna z velmi často doporučovaných praktik na zvýšení kvality zdrojového kódu. Výhody využívání Dependency Injection se podobně jako u návrhových vzorů plně projeví až u rozsáhlejších projektů [1, 35]. Mezi hlavní benefity Dependency Injection patří:

- **Nízká provázanost tříd** - díky DI lze snadno vytvářet kód s nízkou provázaností tříd. Každá třída je vyvíjena v izolaci od ostatních a vztahy mezi nimi jsou jasně stanoveny objektovým grafem, který je vytvářen DI Kontejnerem [1, 34, 35].
- **Snadná testovatelnost** - komplikovaná logika vytváření závislostí je přesunuta do DI kontejneru. DI navíc snadno umožňuje zaměnit potřebné závislosti testovacími náhradníky [1, 34, 35].
- **Snadná opravovatelnost a rozšiřitelnost** - pokud potřebujeme v průběhu vývoje nějakou třídu upravit nebo rozšířit, můžeme tak často dělat v úplné izolaci, kdy se provedené změny nijak neprojeví v jiných třídách [1, 35].

- **Snadný paralelní vývoj** - jednotlivé komponenty programů lze snadno vyvíjet paralelně. Pokud tým pracující na třídě A změní její konstruktor, žádným způsobem neovlivní třídy, které potřebují třídu A jako závislost [35].

4.2 Dagger 2

Dagger 2 je plně statický DI kontejner určený pro programovací jazyk Java. Vytvořila jej společnost Square, přičemž v současné době jej spravuje společnost Google. Dagger 2 vytváří kompletní objektový graf při kompilaci, kdy za pomoci anotací automaticky generuje Java kód zajišťující vytvoření, kontrolu, propojení a injektování daných závislostí [1, 37].

Dagger 2 podporuje všechny hlavní typy DI - tedy Constructor Injection, Field Injection a Method Injection. Konfigurace probíhá pomocí auto-registrace (pouze pro Constructor Injection) doplněné o konfiguraci zdrojovým kódem. Pro potřeby auto-registrace plně dostačuje přidat před konstruktor dané třídy anotaci @Inject. Dagger 2 následně rekurzivně vytvoří všechny potřebné závislosti a prostřednictvím konstruktoru je dané třídě předá, viz výpis 4.4 [1, 37].

Výpis 4.4: Příklad Constructor Injection pomocí Dagger 2.

```
1 class Test @Inject constructor(private val timer: Timer){
2     ...
3 }
```

V některých případech Constructor Injection používat nelze. Příkladem mohou být aktivity u aplikací pro OS Android, kde si je vytváří framework sám, přičemž vyžaduje zachování základního prázdného konstruktoru. Je tak nutné přejít k alternativě v podobě Field Injection. Field Injection nemůže využívat auto-registraci a je tak nutné provést konfiguraci pomocí Dagger Komponenty, viz kapitola 4.2.2 [1, 37]. Příklad třídy s Field Injection je zobrazen ve výpisu 4.5.

Výpis 4.5: Příklad Field Injection pomocí Dagger 2.

```
class MainActivity {

    //označení proměnné, která má být injektována
    @Inject
    lateinit var viewModelFactory: ViewModelFactory

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

        //Žádost o injektování závislostí
        ComponentUtil.appComponent.inject(this)

        ...
    }
}
```

4.2.1 Dagger Modul

Dagger Modul je třída označená anotací @Module, která má za úkol pomoci Daggeru 2 stanovit, jak vytvářet dané závislosti. V Modulech musí být definovány všechny závislosti, které Dagger 2 nedokáže sám vytvořit pomocí auto-registrace, nebo které potřebují nějakou konfiguraci. K definování toho, jak závislosti vytvořit, slouží metody označené anotací @Provide. Na výpisu 4.6 je vidět příklad deklarace vytváření závislosti typu AppDatabase [1, 37].

Výpis 4.6: Příklad Provide metody Dagger Modulu.

```
@Module
class AppModule {
    // Provide metoda stanovuje jak vytvořit objekt typu
       AppDatabase
    @Provides
    fun provideRoomDB(context: Context): AppDatabase = Room
        .databaseBuilder(context, AppDatabase::class.java, "AppDB")
        .fallbackToDestructiveMigration()
        .build()
}
```

Dalším úkolem modulů je stanovit konkrétní implementace deklarovaných interfaců. Děje se tak pomocí metod označených anotací @Binds. Například v případě, že injektovaný konstruktor obsahuje interface TestRepository, musíme v Dagger Modulu určit, jaká implementace tohoto interfacu bude injektována [1, 37]. Příklad stanovení konkrétní implementace je vidět na výpisu 4.7.

Výpis 4.7: Příklad Bind metody v Dagger Modulu.

```
@Module
abstract class RepositoryModule {
    // Provide metoda stanovuje jak vytvořit objekt typu
       AppDatabase
    @Binds
    @NonNull
    abstract fun bindTestRepository(testRepositoryImpl:
       TestRepositoryImpl): TestRepository
}
```

4.2.2 Dagger Komponenta

Dagger Komponenta je interface označený anotací @Component, který slouží k vytvoření a konfiguraci objektových grafů. Jeho hlavním úkolem je instruovat Dagger 2 jak vytvářet konkrétní závislosti (ve spolupráci s Dagger Moduly) a kam tyto závislosti injektovat (pouze pokud se nejedná o Constructor Injection).

Výpis 4.8: Příklad Dagger Component.

```

//Určení modulů, které daná komponenta využívá
@Component(modules = [AppModule::class, ViewModelModule::class,
    RepositoryModule::class])
interface AppComponent {
    //Dagger musí být instruován, že má provést DI
    //nad Aktivitou, protože využívá Field Injection
    fun inject(activity: MainActivity)
}
```

4.2.3 Dagger Rámec

Bez bližšího upřesnění vytváří Dagger 2 při každé injektaci novou instanci závislosti požadovaného typu. V některých případech je ale vytváření instancí určitých závislostí velmi drahá operace a je tak potřeba vytvořit danou závislost pouze jednou a opakovaně injektovat stejnou instanci této závislosti. Tento požadavek řeší Dagger rámec (Scope). Rámec závislosti může být deklarován v Dagger modulu přidáním anotace @Singleton k dané provide metodě. Instance takovéto závislosti bude vytvořena v celé aplikaci pouze jednou a zničena bude až ve chvíli zániku aplikace. Kromě Singleton rámce umožňuje Dagger 2 vytvářet také vlastní rámce, které budou navázané například na životní cyklus jedné aktivity. Deklarace Dagger Rámce je znázorněna ve výpisu 4.9 [1, 37].

Výpis 4.9: Příklad Dagger Modulu.

```
@Module
class AppModule {
    //Díky této anotaci bude vytvořena v celé aplikaci
    //pouze jediná instance závislosti typu AppDatabase
    @Singleton
    @Provides
    fun provideRoomDB(context: Context): AppDatabase = Room
        .databaseBuilder(context, AppDatabase::class.java, "AppDB")
        .fallbackToDestructiveMigration()
        .build()
}
```

5 Reaktivní programování

Jedním z často se objevujících problémů při vývoji aplikací pro OS Android je nutnost psaní asynchronních operací. Hlavní vlákno Android aplikací, které ovládá uživatelské rozhraní, by složitějšími operacemi nemělo být blokováno na více než 17 ms (pro 60 Hz obnovovací frekvenci). V opačném případě se uživateli jeví aplikace jako neplynulá až neresponzivní. Z tohoto důvodu je nutné časově náročné operace provádět asynchronně ve vlákně na pozadí [1].

Psaní asynchronních programů ale přináší několik problémů jako je komplikovaná správa chyb, nepřehledné callbacky, problémy se sousledností operací na pozadí a další. Jedním z prostředků, který řeší výše zmíněné problémy a usnadňuje psaní asynchronních programů je reaktivní programování [38].

Reaktivní programování je programovací paradigma založené na asynchronních datových tocích a na automatické propagaci změn [39, 40, 41]. Reaktivní paradigma se opírá o čtyři hlavní pilíře:

- **Responsivnost** - systém by měl reagovat na uživatelský vstup v co nejkratším čase, žádné operace by tak neměly blokovat vlákno ovládající uživatelské rozhraní aplikace [40, 42].
- **Odolnost** - systém by měl být odolný vůči chybám a v případě, že nějaká nastane, měl by i nadále zůstat responzivní [40, 42].
- **Pružnost** - systém by měl zůstat responzivní při různé míře zátěže. Systémy využívající reaktivní programování by tak měly být snadno škálovatelné [40, 42].
- **Zaměření na datové toky** - komponenty aplikace by spolu měly komunikovat asynchronním předáváním zpráv prostřednictvím datových toků, což snižuje jejich provázanost [40, 42].

5.1 Datový tok

Základním prvkem reaktivního programování je datový tok. Datový tok je časová sekvence datových objektů nebo událostí různého typu. Příkladem datového toku mohou být objekty přicházející ze serveru, události informující o kliknutí na tlačítko apod. Tyto datové toky je možné následně různě spojovat a transformovat [39, 40, 41].

Data emitována do datových proudů jsou konzumenty zpracovávána asynchronně. Konzument dat se tedy pouze přihlásí k odběru, ale čekáním na data není nijak blokován a může provádět další operace. Ve chvíli, kdy data konzumentovi přijdou, spustí se funkce, která je zpracuje [39].

5.2 Využívané návrhové vzory a Plánovač

Základem reaktivního programování je návrhový vzor Observer (viz kapitola 1.3.1), kdy producent dat udržuje seznam konzumentů, kteří se přihlásili k odběru dat, a automaticky je o změnách dat notifikuje. Reaktivní programování je ve skutečnosti určitou nadstavbou návrhového vzoru Observer, kdy na veškerou komunikaci mezi producentem a konzumentem dat se hledí jako na datové proudy. Kromě návrhového vzoru Observer reaktivní paradigma široce využívá návrhový vzor Iterator, který poskytuje sekvenční přístup k elementům kolekcí [39, 40].

Další důležitou součástí reaktivního paradigma jsou Plánovače, které poskytují abstrakci nad vlákny, mezi než je rozkládána zátěž programu. Díky reaktivnímu programování je tak relativně snadné psát vícevláknové aplikace, které mohou provádět výpočetně náročné operace na vláknech na pozadí, zatímco hlavní vlákno ovládající uživatelské rozhraní zůstává nezatížené a tedy responsivní [39, 40].

5.3 Výhody reaktivního programování

Reaktivní programování poskytuje oproti klasickému imperativnímu paradigmatu (program složen z posloupností příkazů, které definují jak řešit daný problém) několik důležitých výhod:

- Reaktivní programování zvyšuje abstrakci zdrojového kódu. Vývojář se tak nemusí tolik zabývat implementačními detaily a může se více zaměřit na funkcionalitu [39, 41].
- Vyšší míra abstraktnosti přináší rovněž větší stručnost a přehlednost zdrojového kódu. V případě, že je vývojář seznámen s principy reaktivního programování, je pro něj snadnější kód pochopit než v případě klasického objektově orientovaného programování [40, 41].
- Asynchronní povaha reaktivního programování zlepšuje responzivnost aplikací. Hlavní vlákno spravující uživatelské rozhraní není zatěžováno náročnými výpočetními úkoly, které jsou přesunuty na vlákna na pozadí [40, 41].
- Reaktivní programování snižuje provázanost zdrojového kódu. Třídy emitující data neví o existenci konkrétních tříd konzumující data, pouze k nim přistupují přes standardizovaný interface Observeru [40, 41].

5.4 RxJava

RxJava je knihovna uplatňující koncept reaktivního programování na JVM (Java Virtual Machine). Knihovnu RxJava vyvíjí od roku 2014 společnost ReactiveX, která podobné knihovny vydala i pro další programovací jazyky jako je C#, C++,

Swift, Python, PHP a další. Knihovna v současné době zahrnuje tři hlavní vydání označována jako RxJava 1, RxJava 2 a RxJava 3, která jsou mezi sebou navzájem nekompatibilní [43].

RxJava umožňuje vytvářet asynchronních programy řízené událostmi. Základním blokem jsou observovatelné objekty nazývané Observables, ke kterým se k odběru dat přihlašují objekty implementující interface Observer. Přihlášením Observerů k Observable objektu mezi nimi vzniká datový proud, který lze ovlivňovat velkým množstvím operátorů [43, 44]. Jednoduchý příklad vytvoření datového toku a přihlášení se k odběru je vidět na výpisu: 5.1.

Výpis 5.1: Příklad vytvoření datového toku pomocí knihovny RxJava.

```
1  //Vytvoření observable, které emituje 4 stringy.
2  val observable = Observable.just("FEKT", "FIT", "FSI", "FAST")
3      .filter { it -> it.length == 4 }
4  //Operátor filter propustí pouze čtyřpísmenné stringy.
5
6  val observer = object : Observer<String>{
7      override fun onSubscribe(d: Disposable) {}
8      override fun onComplete() {}
9      override fun onNext(t: String) {
10     //Zde se zachytí emitováné stringy, které prošly přes datový
           proud.
11     }
12     override fun onError(e: Throwable) {}
13 }
14 //Přihlášení observeru k odběru dat od observable.
15 observable.subscribe(observer)
```

5.4.1 Observables

Objekty typu Observable jsou základním stavebním prvkem reaktivního programování pomocí knihovny RxJava a jejích alternativ pro jiné jazyky. V rámci návrhového vzoru Observer jsou představovány komponentou Subject, viz kapitola 1.3.1. Hlavním úkolem objektů Observable je spravovat seznam Observerů, které jsou přihlášeni k odebírání dat, a emitování datových objektů do vytvořených datových toků [45].

Observables se dělí do dvou hlavních kategorií na takzvané Hot Observables a Cold Observables.

- **Cold Observables** - zopakují emise všech datových objektů všem Observerům po jejich přihlášení k odběru. Většina Observables v rámci knihovny RxJava je právě tohoto typu [43, 45].
- **Hot Observables** - vysílají emise všem Observerům najednou, přičemž nově přihlášeným Observerům již vyslaná data neopakuje. Pokud Observable vyšle

tři datové objekty a až poté se daný Observer přihlásí k odběru, tyto již vyslané objekty neobdrží [43, 45].

Knihovna RxJava obsahuje několik různých typů Observables, které se od sebe liší celkovým počtem emitovaných dat. Každý druh Observable pak má k sobě přidružený Observer interface [43, 45].

Observable

Observable je základním zástupcem stejnojmenné komponenty. Umožňuje emitovat $0 \ldots n$ datových objektů pomocí funkce *onNext(data : T)*. Pokud je emise datových objektů dokončena, zavolá Observable funkci *onComplete()*, prostřednictvím které je Observer notifikován o tom, že další datové objekty již nebudou emitovány a může se odhlásit z odběru. V případě, že nastane chyba, informuje o tom Observable Observery prostřednictvím funkce *onError(e : Throwable)* [43, 45]. Příklad datového toku s Observable objektem a příslušným Observerem je vidět na výpisu 5.1 výše.

Single

Single je typ Observable komponenty, která umožňuje emitovat pouze 1 datový objekt. S konzumenty dat komunikuje prostřednictvím přidruženého SingleObserveru. Úspěšné emitovaný datový objekt je Observeru předán prostřednictvím funkce *onSuccess(data : T)*, která kombinuje funkce *onNext(data : T)* a *onComplete()* z Observable. Po zavolání *onSuccess(data : T)* se tedy Observer odhlásí z odběru dat. Stejně jako tomu bylo u Observable, jsou případné chyby Observerům předávány prostřednictvím funkce *onError(e : Throwable)* [43, 45]. Na výpisu 5.2 je vidět jednoduchý příklad získání objektu uživatele z databáze pomocí Single.

Výpis 5.2: Příklad získání objektu uživatele z databáze pomocí Single.

```
1  //Získání Single z API databáze.
2  val single : Single<User> = db.getUserSingle()
3  //Vytvoření SingleObserveru.
4  val observer: SingleObserver<User> = object : SingleObserver<User>{
5      override fun onSubscribe(d: Disposable) {}
6      override fun onSuccess(user: User) {
7          //Zde je zachycen emitovaný objekt User.
8      }
9      override fun onError(e: Throwable) {
10         // Pokud nastane chyba, zde se zachytí.
11     }
12 }
13 //Přihlášení observeru k odběru.
14 single.subscribe(observer)
```

Maybe

Maybe je typ Observable komponety, která umožňuje emitovat 0 nebo 1 datový objekt. Přidružený MaybeObserver je velmi podobný SingleObserveru s tím rozdílem, že obsahuje funkce jak *onSuccess(data : T)* tak *onComplete()* [43, 45]. Příklad stažení datového objektu uživatele pomocí Maybe je vidět na výpisu 5.3.

Výpis 5.3: Příklad získání objektu uživatele z databáze pomocí Maybe.

```
1  val maybe : Maybe<User> = db.getUserMaybe(uId)
2
3  val observer: MaybeObserver<User> = object : MaybeObserver<User>{
4      override fun onSubscribe(d: Disposable) {}
5      override fun onSuccess(user: User) {
6          //Zde je zachycen emitovaný objekt User.
7      }
8      override fun onComplete() {
9          //Pokud není emitován žádný objekt, zavolá se onComplete().
10         //Např. žádný uživatel pod uId nebyl nalezen.
11     }
12     override fun onError(e: Throwable) {
13         // Pokud nastane chyba, zde se zachytí.
14     }
15 }
16 //Přihlášení observeru k odběru.
17 maybe.subscribe(observer)
```

Completable

Předchozí typy Observable slouží především na získání nějakých dat například z databáze. Naproti tomu Completable neemituje žádná data, ale pouze informaci, že požadovaná událost byla v pořádku dokončena. Využívá se tak například pro nahrávání dat do databáze. Completable komunikuje s přidruženým CompletableObserverem prostřednictvím metod *onComplete()* a *onError()*, viz výpis 5.4 [43, 45].

Výpis 5.4: Příklad uložení dat v databázi pomocí Completable.

```
val completable : Completable = db.saveString("VUT")
val observer: CompletableObserver = object : CompletableObserver{
    override fun onSubscribe(d: Disposable) {}
    override fun onComplete() {
        //string byl úspěšně uložen v db
    }
    override fun onError(e: Throwable) {
        //uložení skončilo chybou
    }
}
//přihlášením observeru se provede uložení stringu do db
completable.subscribe(observer)
```

5.4.2 Operátory

Knihovna RxJava obsahuje velké množství operátorů, které slouží k ovlivnění jednotlivých datových toků. Operátory tak mohou sloužit například k filtraci dat, transformaci dat, spojování datových toků, řešení chybových stavů a mnohé další. V rámci jednoho datového toku je možné využít i více operátorů, díky čemuž lze vyjádřit i relativně komplikované operace malým množstvím kódu [45].

Na pozadí se operátory chovají jako pár Observer-Observable. Pokud tedy přidáme operátor k nějakému Observable objektu, operátor se k němu automaticky přihlásí k odběru prostřednictvím svého skrytého Observeru a v závislosti na daném operátoru vygeneruje nový Observable, ke kterému se mohou přihlašovat k odběru další operátory nebo cílové Observery [45].

V následujících odstavcích budou popsány nejvýznamnější a často se vyskytující operátory knihovny RxJava.

Filter

Operátor Filter slouží k filtrování datových objektů v datovém proudu. Operátor tak přes sebe propustí pouze ty položky, které splní definovaný predikát. Příklad použití tohoto operátoru je možné vidět na výpisu 5.1 výše, kde jsou odfiltrovány všechny objekty typu string, které nemají délku 4 [43, 45].

Map

Operátor Map je jedním z nejdůležitějších zástupců transformačních operátoru. Jeho úkolem je transformovat každý emitovaný prvek určitého datového typu na jiný datový typ [43, 45].

Výpis 5.5: Ukázka operátoru Map.

```
1 //Emise objektů typu string jsou transformovány na číslo (Int)
2 //odpovídající délce původních stringů.
3 val observable = Observable.just("FEKT", "FIT", "FSI", "FAST")
4     .map { it-> return@map it.length }
```

ToList

Operátor ToList se řadí mezi operátory kolekcí. Operátor postupně sbírá všechny emitované datové objekty a po zavolání *onComplete()* původního Observable emituje všechny nasbírané položky najednou v podobě listu. V základní implementaci operátor emituje ArrayList, nicméně lze definovat i jiné implementace. Operátor původní Observable transformuje na Single, takže je nutné při použití tohoto operátoru použít SingleObserver. Příklad použití operátoru je vidět na výpisu 5.6. V příkladu je kromě funkce samotného operátoru ToList znázorněno rovněž řetězení operátorů, v tomto případě Just, Map a ToList [43, 45].

Výpis 5.6: Příklad řetězení operátorů Just, Map a ToList.

```
1  //Emise typu string jsou nejprve transformovány na Int a následně
2  //je z nich vytvořen List.
3  val single: Single<List<Int>>
4          = Observable.just("FEKT", "FIT", "FSI", "FAST")
5      .map { it-> return@map it.length }
6      .toList()
7
8  val observer: SingleObserver<List<Int>> = object : SingleObserver<
       List<Int>>{
9      override fun onSubscribe(d: Disposable) {}
10     override fun onSuccess(data: ArrayList<Int>) {...}
11     override fun onError(e: Throwable) {...}
12 }
13
14 single.subscribe(observer)
```

CombineLatest

Operátor CombineLatest slouží ke spojení dvou datových toků do jednoho. Operátor kombinuje emise z více Observable do jednoho datového objektu, který následně emituje. Ve chvíli, kdy jakýkoliv ze vstupních Observable emituje nový objekt, vezme jej, zkombinuje s posledními emisemi ostatních Observables a nový datový objekt emituje. Příklad využití operátoru combineLatest je vidět na výpisu 5.7, kde k vytvoření objektu Weather jsou potřeba objekty typu Wind a Temperature [43, 45].

Výpis 5.7: Příklad použití operátoru CombineLatest.

```
1  //Definování dvou zdrojů dat.
2  val windObservable = api.observeWindCondition()
3  val temperatureObservable = api.observeTemperature()
4
5  //Zkombinování dvou datových toků do jednoho.
6  val weatherObservable: Observable<Weather> = Observable.combineLatest(
7      windObservable,
8      temperatureObservable,
9      BiFunction { wind, temperature ->
10         return Weather(wind, temperature)
11     }
12 )
```

5.4.3 Vícevláknové aplikace pomocí RxJava

Knihovna RxJava umožňuje relativně snadné psaní vícevláknových aplikací. V základní podobě jsou operace v rámci datových toků prováděny na stejném vlákně, na kterém proběhlo přihlášení se k odběru dat (s výjimkou použití určitých operátorů, viz dále). Vzhledem k nutnosti nechat hlavní vlákno Android aplikací nezatížené, je ale vhodné časově náročné operace přesunout na jiná vlákna [43, 45].

Schedulers

Vytvoření nového vlákna je operace náročná na prostředky a proto je nutné vlákna po skončení konkrétního úkolu nezahazovat, ale znovu využít na další úkoly. Knihovna RxJava vytváří nad vlákny abstrakci ve formě Schedulers (konkrétní implementace komponenty Plánovač viz výše), což je ve své podstatě Thread Pool, který vytvoří určité množství vláken, mezi které následně rozděluje úkoly. Po dokončení úkolu je dané vlákno znovu využito pro další úkol, čekající ve frontě [43, 45].

Knihovna RxJava obsahuje několik základních typu Schedulers, přičemž mezi nejvýznamnější patří:

- **Schedulers.computation** - je určen především pro výpočetně náročné úlohy. Zpravidla vytváří Thread pool s pevným počtem vláken, který je roven počtu logických jader procesoru, na kterém je aplikace spuštěná. Počet vytvořených vláken je dán předpokladem, že operace prováděné v rámci tohoto Scheduleru mohou zatížit konkretní jádro procesoru na 100 % a tudíž by vytvoření většího počtu vláken pouze zbytečně spotřebovávalo systémové prostředky. Schedulers.computation využívá velké množství operátorů již v základu. Například jsou to operátory Interval, Delay, Timer a další [43, 45].

- **Schedulers.io** - je určen pro input/output operace, jako je zápis a čtení do databází, žádosti o data z webových serverů a podobně. Tyto operace jsou většinou méně náročné na procesorový čas a nepředpokládá se, že by výrazně zatížily jádro procesoru. Vzhledem k této skutečnosti je v rámci Schedulers.io vytvořen Thread pool s dynamicky se měnícím počtem vláken podle aktuální potřeby, který není limitován počtem jader procesoru [43, 45].
- **AndroidSchedulers.mainThread** - je speciální Scheduler určený pro Android aplikace, který není součástí základní knihovny RxJava, ale jejího rozšíření RxAndroid. Uživatelské rozhraní Android aplikací běží na hlavním UI vláknu. Pokud potřebujeme zobrazit výsledek operace prováděné v datovém toku na jiném vlákně, musíme tento výsledek pomocí Scheduleru AndroidSchedulers.mainThread přemístit na hlavní vlákno, v opačném případě aplikace spadne. [45].

Operátor SubscribeOn

Operátor SubscribeOn navrhne zdroji datového toku (většinou Observable), jaký využívat Scheduler. Zdroj datového toku návrh neuposlechne pouze v případě, že je již svázany s jiným Schedulerem. Tato situace nastává, pokud datový tok obsahuje operátor s definovaným základním Schedulerem, nebo v případě, že jiný operátor SubscribeOn nacházející se blíže zdroji již Scheduler definoval. Pokud není nastaven operátor ObserveOn, je daný Scheduler použit v celém datovém toku od Observable až po Observer [43, 45].

Operátor ObserveOn

Operátor ObserveOn slouží ke změně Scheduleru (a tedy vlákna) v datovém toku. Operátor zachytí emise dat na jednom vlákně a přemístí je na jiné vlákno definované vlastním Schedulerem. Na rozdíl od operátoru SubscribeOn tak ovlivňuje datový tok pouze směrem ke konzumentovi dat, a je jej možné použít v datovém toku více než jednou [43, 45].

Příklad využívání Schedulerů a operátorů SubscribeOn a ObserveOn je vidět na výpisu 5.8.

Výpis 5.8: Příklad využívání Schedulers a operátorů SubscribeOn a ObserveOn.

```
val resultObservable: Observable<Result> = db.observeResults()
    //Data jsou emitována vláknem určeným Schedulers.io.
    .subscribeOn(Schedulers.io())
    //Vše pod operátorem ObserveOn je vykonáváno na vlákně
    //určeném Schedulers.computation až po další observeOn.
    .observeOn(Schedulers.computation())
    .filter { result -> isCorrectResult(result) }
    //Konec datového proudu je přemístěn na hlavní vlákno,
    //aby emitovaná data mohla být zobrazena uživateli.
    .observeOn(AndroidSchedulers.mainThread())

val observer = object : Observer<Result>{
    override fun onComplete() {}
    override fun onSubscribe(d: Disposable) {}
    override fun onNext(t: Result) {
        /*
        Díky poslednímu operátoru ObserveOn jsou zde zachycená
        data již na hlavním vlákně a lze je zobrazit v UI.
        */
    }
    override fun onError(e: Throwable) {}
}
resultObservable.subscribe(observer)
```

6 Testování aplikací

Testování je důležitou součástí vývoje aplikací. Správně vytvořené automatické testy umožňují rychlou detekci chyb v kódu, bezpečné refaktorování a stabilní rychlost vývoje. Automatické testy lze rozdělit do dvou hlavních kategorií podle toho, na jakém zařízení jsou spouštěny [1, 51]:

- **Lokální testy** - jsou spouštěny lokálně na vývojářově počítači na JVM (Java Virtual Machine), díky čemuž jsou velmi rychlé. Jejich nevýhodou je, že nemají přístup k závislostem Android frameworku. Tyto závislosti lze do určité míry nasimulovat náhražkami, například pomocí knihovny Roboelectric. Nicméně pro testování kódu s komplexní závislostí na třídách z Android frameworku je doporučeno používat spíše Instrumentální testy [1, 51].
- **Instrumentální testy** - jsou spouštěny přímo na fyzických Android zařízeních nebo na emulátorech. Výhoda těchto testů spočívá v tom, že mají přímý přístup k závislostem Android frameworku jako je například Context. K přístupu k závislostem na Android framework se požívá Android Instrumentation API, které slouží také k ovládání životního cyklu těchto závislostí. Pro účely těchto testů je vytvořen APK soubor, který je spuštěn společně s testovanou aplikací ve společném procesu. Testovací aplikace má tak přístup ke všem částem testované aplikace. Nevýhodou instrumentálních testů je jejich menší rychlost oproti lokálním testům [1, 51].

Podle rozsahu se automatické testy dělí na čtyři hlavní kategorie:

- **Unit testy** - testování jednotlivých metod a tříd
- **Integrační testy** - testování celků a interakcí mezi nimi.
- **UI testy** - testování kompletních funkcionalit aplikace.
- **App crawler testy** - bezkonfigurační testování celé aplikace.

6.1 Unit testy

Jak už název napovídá, Unit testy slouží k otestování nejmenších samostatně testovatelných logických bloků aplikace. Jedná se o malé, velmi úzce zaměřené testy, které svým rozsahem mohou pokrývat jedinou komponentu, třídu nebo jedinou metodu. Unit testy by měly každou komponentu otestovat na všechny možné typy vstupů a validovat korektnost výstupů. Ve většině případů se řadí mezi lokální testy a měly by být optimalizovány pro paralelní vykonávání. Hlavní charakteristikou Unit testů je jejich rychlost a reprodukovatelnost. Podle doporučení společnosti Google by se soubor testů aplikace měl sestávat ze 70 % z unit testů [1, 46, 47, 51]. Příklad jednoduchého unit testu je vidět na výpisu 6.1.

Výpis 6.1: Příklad jednoduchého unit testu.

```
1  public class UnitTestExample {
2
3      private PositiveNumberValidator SUT;
4
5      @Before
6      public void setup() {
7          SUT = new PositiveNumberValidator();
8      }
9
10     @Test
11     public void positiveNumberValidator_positiveNumber_returnTrue() {
12         boolean result = SUT.isPositive(1);
13         Assert.assertThat(result, is(true));
14     }
15
16     //... třída by byla dále doplněna o test kontrolující
17     //navrácenou hodnotu při vstupu rovném 0 a negativnímu číslu.
18 }
```

Aby byly unit testy rychlé, a jejich výsledky konzistentní je nutné testovat dané části kódu v izolaci. K tomu napomáhá správně navržený zdrojový kód aplikace společně s různými knihovnami, které asistují při vytváření testovacích náhražek reálných závislostí (Roboelectric, Mockito a další) [1, 46]. Testovací náhražky mohou být několika různých druhů:

- **Fake** - Náhražka s funkční implementací dané třídy, která je často zjednodušená a uzpůsobená pro potřeby testování (například nahrazení databáze jednoduchou hashmapou) [46, 48].
- **Mock** - Náhražka, která monitoruje, které z jejich metod byly zavolány. Pomocí náhražek typu Mock tak můžeme ověřit, že byly vykonány všechny akce, které měly proběhnout [46, 48].
- **Stub** - Náhražka, která neobsahuje žádnou implementaci a její metody pouze navrací přednastavenou hodnotu [46, 48].
- **Dummy** - Náhražka, která je předávána mezi metodami, ale sama není využita [46].
- **Spy** - Náhražka, která navíc zaznamenává další data (například kolik objektů bylo vloženo do databáze) [46].

6.1.1 Testovatelnost zdrojového kódu a TDD

Ne všechen zdrojový kód je testovatelný unit testy, proto je potřeba při programování na testovatelnost myslet. Dobře testovatelný kód má následující charakteristiky:

- Využívání principu jedné zodpovědnosti - každá třída by měla řešit pouze jediný úkol [47, 49, 52].
- Využívání Dependency Injection - závislosti jsou objektu dodány namísto toho, aby si je vytvářel sám [47, 49, 52].
- Využívání principu segregace pomocí interfaců - klientské třídy jsou závislé na interfacech, nikoliv na konkrétních implementacích (snadné nahrazení závislostí testovacími náhražkami) [47, 49, 52].
- Vyvarování se globálních proměnných, statických metod a singletonů - v opačném případě dojde k problematickému paralelnímu vykonávání unit testů a k problematickému nahrazování za testovací náhražky. [47, 49, 52].
- Dodržování jednoho z architektonických návrhových vzorů (např. MVVM) - aplikace by měla být rozdělená na vrstvy s jasně danými zodpovědnostmi a s jednotnou formou komunikace [52].
- Třídy mají pouze jednoduché konstruktory - jediným úkolem konstruktorů by měla být inicializace proměnných v dané třídě [52].

Obecně se doporučuje vytvořit unit testy ještě před samotným naprogramováním dané komponenty a danou komponentu posléze vyvíjet tak, aby těmito testy prošla. Kromě ověření funkčnosti pak testy zároveň slouží jako dokumentace, kdy jednoznačně definují požadované chování. Tomuto způsobu vývoje se říká Test Driven Development (TDD) a je to jedno z moderních paradigmat pro vývoj kvalitního softwaru [47, 50].

6.2 Integrační testy

Integrační testy testují interakci mezi několika třídami s cílem ověřit, že mezi sebou spolupracují v souladu s očekáváními. Integrační testy pokrývají větší rozsah kódu aplikace než unit testy, nicméně jsou stále ještě optimalizovány na rychlost [1, 46, 47].

Podobně jako unit testy mohou být vykonávány lokálně na JVM nebo v případě nutnosti využívání komplexních Android závislostí jako instrumentální testy přímo na fyzických Android zařízeních nebo emulátorech [46].

Podle doporučení společnosti Google jsou integrační testy vhodné například na ověření funkčností fragmentů a jejich spojení s ViewModely nebo na otestování datové vrstvy aplikace. Zajímat by měly zhruba 20 % z celkového množství testů vytvořených pro aplikaci [1, 46, 51]. Jednoduchý příklad integračního testu je vidět na výpisu 6.2.

Výpis 6.2: Příklad jednoduchého integračního testu.

```
1
2  class IntegrationTestExample {
3      // inicializace reálných komponent viewmodelu
4      private val keyGenerator = KeyGenerator()
5      private val rsaEncryptor = RsaEncryptor()
6      private val textRepository = TextRepository()
7
8      lateinit var cipherViewModel: CipherViewModel
9      @Before
10     fun setup() {
11         //inicializace viewmodelu
12         cipherViewModel = CipherViewModel(keyGenerator, rsaEncryptor)
13         cipherViewModel.generateKeyPair()
14     }
15
16     //Ověření, že všechny moduly v cipherViewModel spolu správně
           fungují
17     @Test
18     fun encryptTextAssertIsSaved () {
19         cipherViewModel.EncryptAndStoreText("Some text")
20         val encryptedText = textRepository.getLastRecord()
21         val originalText = rsaEncryptor.decrypt(cipherViewModel.
               getPrivateKey(), encryptedText)
22         assertEquals("Some text", originalText)
23     }
24 }
```

6.3 UI testy

Jedná se o takzvané End-To-End testy, které ověřují celé funkcionality aplikace. Ze všech typů jsou UI testy nejrozsáhlejší a pokrývají největší části aplikace, přičemž aplikaci validují jako celek v podobě blízké produkční verzi. Na rozdíl od unit a integračních testů ale trvá jejich vykonávání velmi dlouho, řádově až minuty. UI testy jsou spouštěny výlučně na fyzických Android zařízeních nebo emulátorech, řadí se tedy mezi instrumentální testy. Společnost Google doporučuje, aby z celkového množství testů vytvořených pro aplikaci, zaujímaly UI testy 10% [1, 46, 51].

Typickým příkladem UI testu je ověření funkčnosti sledu obrazovek pro registraci a přihlášení uživatele [51]. Jednoduchý příklad UI testů je k vidění na výpisu 6.3.

Výpis 6.3: Příklad jednoduchého UI testu s využitím knihovny Espresso.

```
@RunWith(AndroidJUnit4::class)
class UiTestExample {
    //Obstará instanci testované aktivity a řídí její životní
       cyklus
    @get:Rule
    var activityRule = ActivityTestRule(MainActivity::class.java)

    @Test
    fun clickButtonAndCheckText() {
        // Nalezne tlačítko s daným ID a klikne na něj
        onView(withId(R.id.btn_display_text)).perform(click())

        // Nalezne Textview s daným ID a zkontroluje,
        // že je v něm zobrazen požadovaný text
        onView(withId(R.id.text_view))
            .check(
                matches(
                    withText(
                        activityRule.activity.getString(R.string.
                           some_text)
                    )
                )
            )
    }
}
```

6.4 App crawler testy

App crawler testy umožňují testovat aplikace bez psaní jakéhokoliv kódu. App crawler se spustí současně s aplikací, kdy ji ovládá standardními povely (kliknutí na tlačítko, scrollování...), čímž prozkoumává celý aplikační prostor. App crawler test je automaticky ukončen ve chvíli, kdy již byly provedeny všechny možné akce, případně po dosažení přednastaveného času nebo při pádu aplikace [1].

Pomocí App crawler testů je možné odhalit kritické chyby způsobující pád aplikace, chyby zobrazení grafických komponent nebo problémy s výkonovou stránkou některých komponent aplikace. Přestože App crawler testy je možné spouštět bez jakékoliv konfigurace, většinou určitou formu konfigurace umožňují (například jaký text má být vyplněn do určitého pole, jakým způsobem se mají v aplikaci přihlásit apod.). Pro App crawler testy je vhodné využít cloudových služeb typu Firebase, které mohou tyto testy spustit na velkém množství různých fyzických i emulovaných zařízeních najednou [1].

7 Vzorová aplikace

V rámci praktické části diplomové práce byla vytvořena aplikace pro interaktivní doučení základoškolské a středoškolské matematiky, na které je demonstrováno využívání návrhových vzorů a dalších ověřených programátorských praktik.

7.1 Motivace

Aby aplikace vytvářená v rámci praktické části této diplomové práce dobře ilustrovala výhody využívání návrhových vzorů, je nutné, aby byla rozsáhlejšího charakteru. V opačném případě by využívání návrhových vzorů představovalo pouze zbytečnou komplikaci zdrojového kódu. Dalším požadavkem na vytvářenou aplikaci bylo smysluplné téma, které by nejen splnilo potřebný rozsah, ale které by zároveň mohlo potenciálně zaujmout běžné uživatele chytrých telefonů. S přihlédnutím k těmto požadavkům bylo nakonec rozhodnuto o napsání aplikace zaměřené na interaktivní výuku matematiky.

Přestože matematika je jedním z nejdůležitějších předmětů, který učí studenty logicky přemýšlet, jako maturitní předmět si ji volí pouze necelá čtvrtina studentů, přičemž tento podíl dlouhodobě klesá. Tento negativní trend bohužel odpovídá dlouhodobé neoblíbenosti matematiky mezi studenty [53, 54].

Jednou z možností jak oblibu matematiky zvýšit je zavedení interaktivních pomůcek při výuce. Aplikace tvořená v rámci této diplomové práce se tak bude soustředit na vývoj mobilní platformy, která by zábavnou a interaktivní formou doplňovala výuku matematiky na základní a střední škole [55].

7.2 Základní požadavky na aplikaci

V první řadě by aplikace měla obsahovat vysvětlení matematické látky. Jednou z možností představení teorie studentům jsou videa, přičemž tuto formu volí i známá aplikace Khan Academy[1]. Problémem výukových videí je ale v jejich pasivnosti, kdy studenti brzy ztrácejí pozornost, což snižuje jejich efektivnost [56]. Další nevýhodou videí je pak jejich datová náročnost.

Aplikace vytvářená v rámci této práce tak bude matematickou teorii studentům předkládat pomocí krokovaných textových materiálů. Mezi jednotlivými kroky budou muset studenti průběžně odpovídat na otázky, což by mělo zvýšit míru jejich zapojení a s tím související míru pozornosti.

[1] https://play.google.com/store/apps/details?id=org.khanacademy.android

Druhým cílem aplikace je procvičení naučené látky. Aplikace tak bude obsahovat rozsáhlý soubor matematických úloh, na kterých budou moci studenti aplikovat naučenou teorii.

Třetím cílem aplikace je ověření znalostí studentů prostřednictvím testů. Aplikace bude obsahovat dva typy testů. Prvním typem jsou testy na konci matematických kurzů, jejichž splněním se dané kurzy dokončí. Druhým typem budou uživatelem vytvořené testy, kde si sám uživatel nastaví jejich parametry.

Dalším prostředkem, který bude aplikace využívat, je gamifikace. Podle mnohých pedagogů mají hry při vzdělávání obrovský potenciál jak zvýšit efektivitu výuky. Aplikace tak bude obsahovat herní prvky, které budou mít za cíl zvýšit míru zapojení uživatelů a poskytovat určitou formu motivace [55].

Popsané požadavky by měly zaručit nejen praktickou využitelnost, ale rovněž i značnou rozsáhlost aplikace, díky čemuž by měly být jasně patrné výhody využívání návrhových vzorů a dalších ověřených programátorských praktik.

7.3 Existující aplikace

Před začátkem vývoje aplikace je vždy vhodné provést průzkum trhu a zjistit, jaké aplikace jsou v dané kategorii k dispozici.

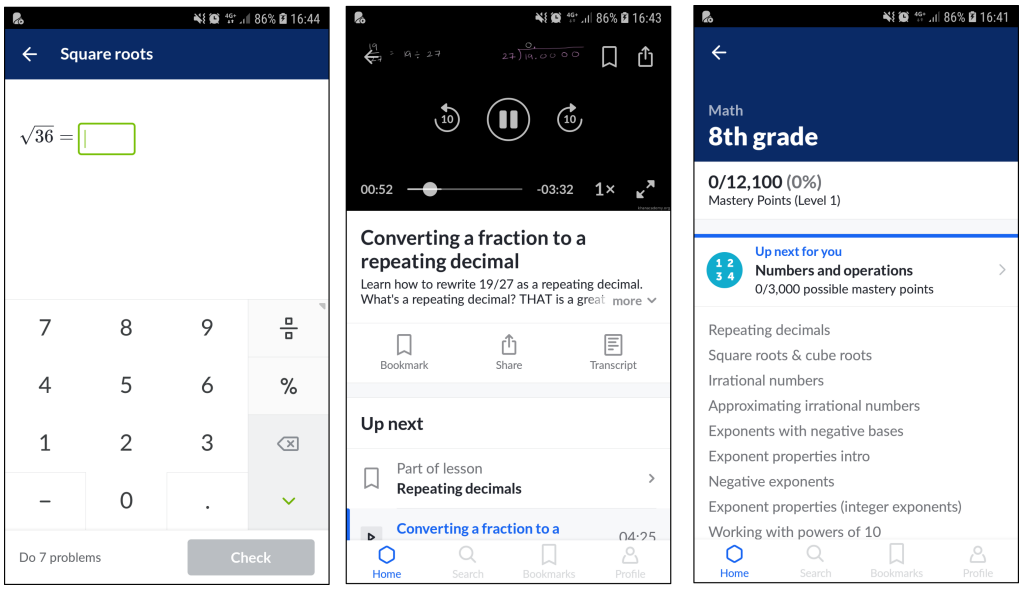
7.3.1 Khan Academy

Jedná se o jednu z nejúspěšnějších aplikací zaměřujících se na výuku s více než 10 000 000 stažení. Uživatelské rozhraní aplikace je vidět na obr 7.1.

Aplikace se nesoustředí čistě na matematiku, ale obsahuje rovněž další předměty včetně fyziky, chemie, ekonomiky a dalších. V rámci matematiky pokrývá látku od základní po vysokou školu. Teorie je vysvětlována převážně pomocí videí, a v menší míře pomocí pasivních textových souborů. Jednotlivá matematická témata jsou doplněna o krátká cvičení a kvízy. Aplikace využívá lehkou formu gamifikace v podobě bodů a odznaků, přičemž v každém tématu postupuje student úrovněmi. Aplikace je dostupná i v češtině (anglicky mluvená videa mají české titulky) a je zcela zdarma.

Na základě průzkumu aplikace byly identifikovány tyto poznatky:

- Aplikace ve zkoumané verzi 6.5.1 není příliš stabilní, často při jejím spuštění spadla.
- Většina stížností uživatelů se týkají zmíněné nestabilnosti a nefunkčnosti některých prvků.
- Obecně je i přes určité nedostatky aplikace velmi dobře přijímaná, což dokládá její hodnocení na Google Play 4,5 z 5.



Obr. 7.1: Uživatelské rozhraní aplikace Khan Academy.

- Aplikace obsahuje obrovské množství materiálů, nicméně k jednotlivým tématům je k dispozici pouze omezené množství příkladů na procvičení, což někteří uživatelé v recenzích rovněž kritizují.
- Jak již bylo řečeno, aplikace k vysvětlování využívá převážně pasivních materiálů (videa a v menším množství texty), což potenciálně vede k nižšímu zapojení uživatelů a nižší míře pozornosti.
- Jako velmi užitečná byla shledána funkcionalita "nedávné lekce", která představuje zkratku z domovské obrazovky k posledním otevřeným lekcím.

7.3.2 Matematika-testy (Eductify)

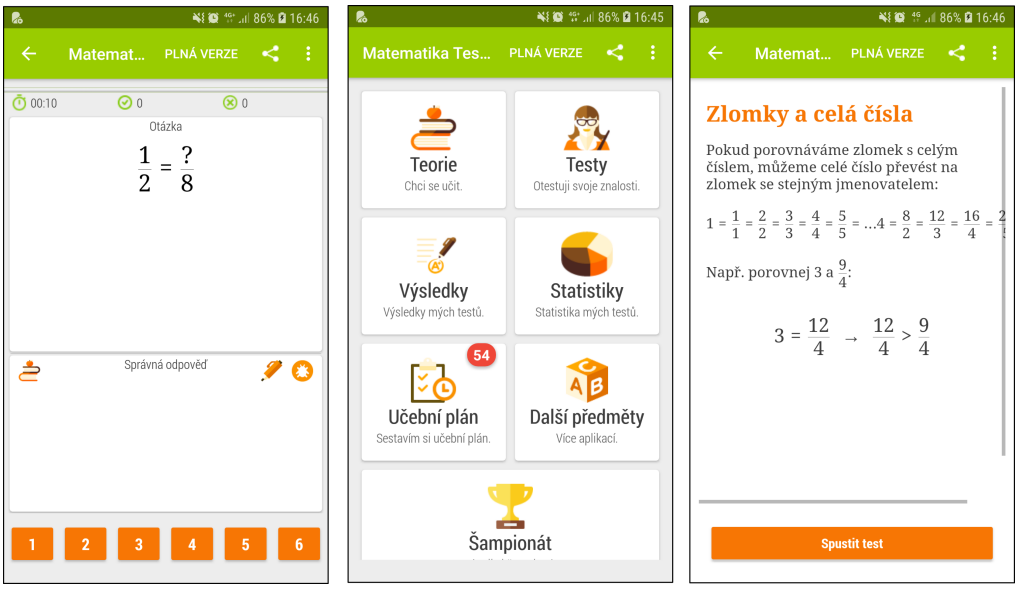
Aplikace Matematika-testy je jedním ze zástupců českých aplikací zaměřujících se na výuku matematiky. Aplikace je rovněž velmi úspěšná s více než 1 000 000 stažení v obchodě Google Play. Uživatelské rozhraní aplikace je vidět na obr. 7.2.

Obsahově aplikace pokrývá pouze základní školu, přičemž látka je vysvětlována na pasivních textových materiálech. Kromě teorie obsahuje aplikace také testy. Ty si uživatel skládá z libovolných okruhů v jednom daném tématu. Aplikace je zdarma pouze v omezené míře, pro odemknutí všech témat a příkladů je nutné zakoupit plnou verzi [2].

Na základě průzkumu aplikace byly identifikovány tyto poznatky:

- Čeští uživatelé podle recenzí oceňují především český obsah aplikace a množství příkladů na procvičení.

[2] https://play.google.com/store/apps/details?id=com.holucent.math



Obr. 7.2: Uživatelské rozhraní aplikace Matematika Testy.

- Nejčastější výtka českých uživatelů směřuje na to, že většina aplikace je dostupná pouze v placené verzi. Dle mnohých komentářů by ocenili verzi s reklamami, kde by bylo více obsahu zdarma.
- Aplikace je velmi dobře přijímána s hodnocením 4,4 z 5.
- Teorie v aplikaci je vysvětlována pouze pasivní formou v podobě textů. Kvalita těchto materiálu je velmi proměnlivá.
- Jako velmi zajímavá byla identifikována funkcionalita učebního plánu. V těchto plánech si uživatel určuje, jaká témata se chce do kdy naučit a aplikace mu sestaví cvičení, která má uživatel v jednotlivé dny projít.

7.4 Volba architektury a technologií

Důležitou součástí vývoje aplikace je rozhodnutí o využití konkrétních technologií. V následujících sekcích je popsána volba architektury aplikace, cloudové databáze a lokální databáze.

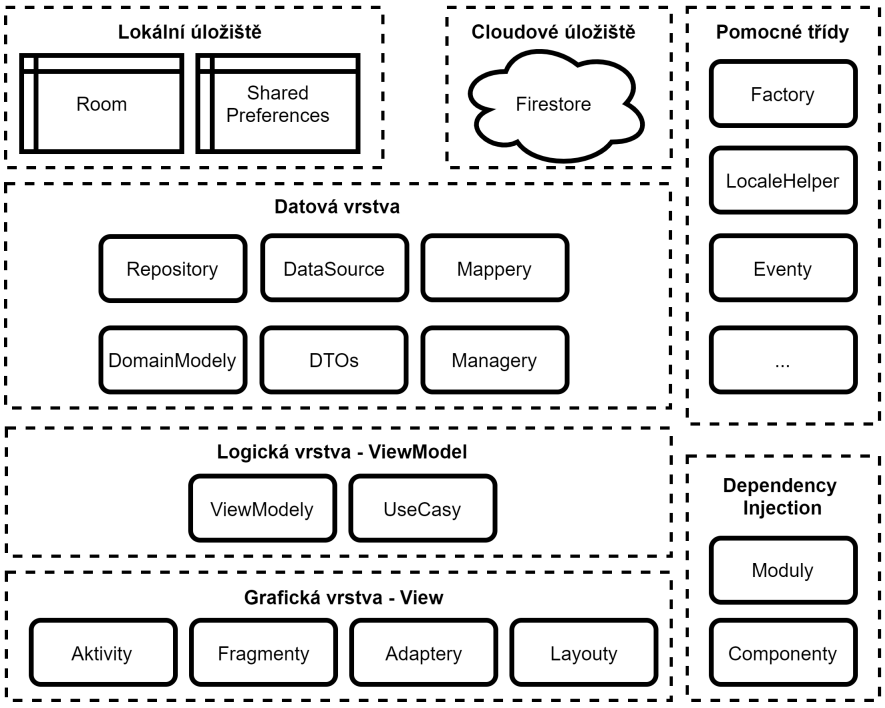
7.4.1 Architektura

Architektura nabývá na významu s velikostí aplikace. Zatímco u malých aplikací s pár obrazovkami a nepříliš rozsáhlou funkcionalitou jsou architektonické návrhové vzory spíše zbytečné zkomplikování kódu, u rozsáhlejších aplikací s desítkami obrazovek je vhodně zvolená architektura zásadní.

Aplikace tvořená v rámci této práce spadá spíše do druhé kategorie, proto u ní architektura hraje významnou roli. Aplikace bude psána podle návrhového vzoru MVVM, který rozděluje zdrojový kód podle účelu na tři vrstvy - prezenční, logickou a datovou, viz kapitola 2.1.3. Návrhový vzor MVVM byl upřednostněn před oblíbeným návrhovým vzorem MVP především kvůli větší podpoře ze strany společnosti Google, která pro návrhový vzor MVVM vydala knihovny pro ViewModel a LiveData (reaktivní stream respektující lifecycle fragmentů a aktivit) v rámci sady knihoven Architecture Components.

Aplikace obsahuje více zdrojů informací (Firebase Cloud Firestore, Room, SharedPreferences a další), které je nutné v některých případech kombinovat pro zobrazení určitých dat uživateli. Z tohoto důvodu jsou některé Modely rozšířeny v souladu s návrhovým vzorem Repository (kapitola 2.2).

Pro snadnější přidávání nových funkcionalit a lepší následování principu jedné zodpovědnosti bude zdrojový kód aplikace následovat architekturu Clean Architecture, která aplikaci dělí na čtyři vrstvy podle míry abstraktnosti, viz kapitola 2.3. Celková struktura aplikace je naznačena na obrázku 7.3.



Obr. 7.3: Struktura aplikace.

7.4.2 Cloudová databáze

Vzhledem ke své povaze může aplikace obsahovat velké množství dat, která by neúměrně zvětšovala datovou velikost aplikace. Data aplikace - matematické příklady

a teorie - bude potřeba v případě nalezení chyby rychle opravovat bez nutnosti aktualizace aplikace. Z těchto důvodů je nutné, aby tato data byla dostupná prostřednictvím cloudové databáze.

Prvním rozhodnutím při volbě databáze je, zda zvolit SQL nebo NoSQL databázi. Výhodou NoSQL databází je jednoduchost a rychlost vývoje dosažená díky absenci pevně dané struktury. Na rozdíl od SQL databází zde nejsou data ukládána do tabulek s pevně danou strukturou, ale do souborů typu klíč-hodnota. Díky tomu zde není nutné realizovat složité vztahy mezi datovými entitami, jako tomu bývá u SQL databází. Na druhou stranu ale NoSQL databáze inherentně nezaručují integritu dat, což je dané právě absencí pevné struktury. Zodpovědnost za integritu dat se zde přesouvá na mechanizmy zápisu [59]. Z důvodů jednoduchosti vývoje a absence složitých vztahů mezi daty byla zvolena databáze typu NoSQL.

Cloudová databáze může fungovat buď na vlastním serveru, nebo ji je možné zřídit jako službu na serverech třetích stran, což sebou nese nižší investiční náklady, vyšší stabilitu a jednoduchost zřízení. V současné době existuje několik firem, které nabízejí kompletní BaaS (Backend as a service), jehož součástí je i cloudová databáze. Mezi tyto společnosti patří např. Firebase a Amazon AWS. Řešení obou společností nabízí rozsáhlé služby pokrývající cloudové databáze, úložiště, analytické nástroje a další. Vzhledem k jednoduchosti a předchozím dobrým zkušenostem s využíváním řešení od Firebase byl zvolen BaaS právě od této společnosti.

Firebase nabízí dvě cloudové databáze: Firebase Realtime Database a novější Firebase Cloud Firestore. Samotná společnost Firebase doporučuje využívat druhou zmíněnou databázi, která nabízí lepší možnosti vyhledávání dat, ve většině případů nižší provozní náklady a další výhody [60]. Jako cloudová databáze tak byla zvolena Firebase Cloud Firestore.

Data v databázi Firebase Cloud Firestore jsou ukládána do dokumentů a kolekcí. Dokumenty jsou struktury podobné JSON souborům, v rámci kterých jsou data ukládána v podobě klíč-hodnota. Data ukládána v těchto dokumentech mohou být několika různých typů, například string, číslo (64-bitový float), mapa, pole, reference na další dokumenty, surová data v podobě bajtů a další. Díky této formě je možné data do Firestore ukládat přímo jako POJO (Plain Old Java Object) a při čtení je na POJO jednoduše parsovat [60].

Dokumenty jsou řazeny do kolekcí, kdy každý dokument musí být součástí jedné kolekce. Samotný dokument pak může obsahovat další kolekce. Celkově tak data tvoří určitou stromovou strukturu [60].

Přístup k datům z aplikace probíhá prostřednictvím klientské knihovny Firebase, která zajišťuje synchronizaci dat a krátkodobou lokální persistenci [60].

Velkou výhodou cloudového řešení od Firebase je integrace s dalšími službami. Firebase tak mimo jiné nabízí autentizaci uživatelů, cloudové úložiště, nejrůznější

statistiky využívání aplikace, služby testování, informace o pádech aplikace a mnohé další [60].

7.4.3 Lokální databáze

Přestože klientská knihovna Firebase vytváří lokální verzi databáze Cloud Firestore v aplikaci, je vhodné mít k dispozici lokální databázi pro data, která mají být dostupná i v offline režimu. Lokální databáze Cloud Firestore totiž není navržena na dlouhodobou persistenci dat, ale slouží spíše pro potřeby kešování [60].

Mezi lokální databáze často využívané v aplikacích pro OS Android patří Room, ObjectBox a Realm. Databáze Room od společnosti Google je jedinou SQL databází z třech zvýše zmíněných. Room je vyvíjena jako součást sady knihoven Architecture components. Výhodou databáze Room je její velmi malý vliv na velikost aplikace (řádově ve stovkách kB), velmi rozsáhlá a podrobná dokumentace a jednoduchá integrace k dalším technologiím (RxJava, Livedata...). Nevýhodou jsou pak komplikovanější vztahy mezi entitami dáno SQL podstatou databáze. Databáze ObjectBox je objektově orientovaná NoSQL databáze s podporou relací a reaktivního programování. Oproti databázi Room má větší vliv na velikost aplikace (1-1,5 MB), ale nabízí snadnější práci s daty. Výkonnostně dosahuje ze všech tří databází nejlepších výsledků. Nevýhodou databáze je menší uživatelská základna a tím pádem horší dostupnost informací a nemožnost přiřazovat objektům identifikátor jiného typu než long. Databáze Realm se vyznačuje nejsnazší prací s daty, na druhou stranu má největší vliv na velikost výsledné aplikace (3-4 MB) [62, 63].

S přihlédnutím na výhody a nevýhody jednotlivých databází byla zvolena lokální databáze Room, a to především kvůli nejnižšímu vlivu na velikost aplikace, podpoře od společnosti Google a snadné dostupnosti informací.

8 UI a UX

Návrh UI se zabývá grafickou stránkou aplikace. Pod návrh UI spadá rozložení grafických komponent z hlediska estetiky, volba barevné palety, animace a další [57].

Návrh UX (User Experience) se zabývá tím, jak bude uživatel s aplikací interagovat. Hlavním cílem správného návrhu UX je vytvoření aplikace, která se bude uživatelům snadno používat a bude intuitivní. Pod UX spadá mimo jiné i návrh rozložení grafických komponent z hlediska jednoduchosti užívání, volba navigace v aplikaci nebo určení funkcionalit, které budou na dané obrazovce k dispozici [57].

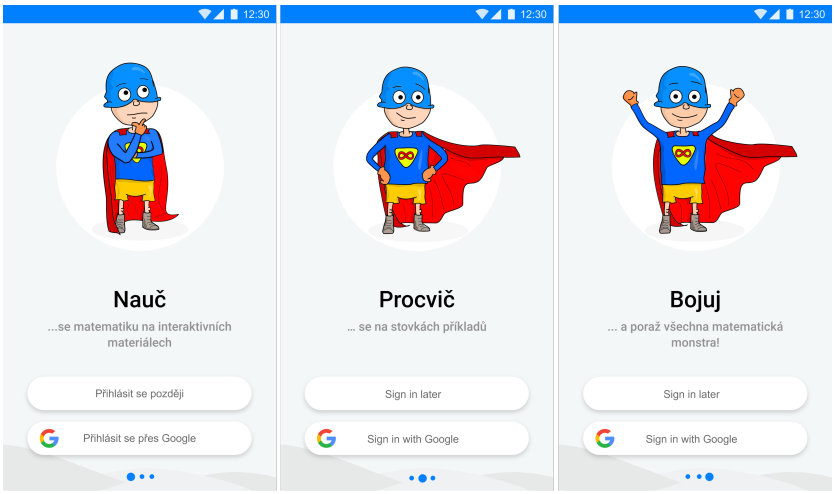
Návrh UI a UX aplikace je vytvořen v souladu s Material Design, což je soubor pokynů a doporučení od společnosti Google, sdružující doporučované praktiky z oblasti UI a UX. Veškeré grafické návrhy byly vytvořeny pomocí programu Adobe XD, který umožňuje vytvářet fotorealistické návrhy grafického prostředí mobilních a webových aplikací [58].

8.1 První spuštění a přihlášení

Dle doporučení Material Design je vhodné po prvním spuštění ukázat uživateli uvítací obrazovku, na které je vyzdviženo, co aplikace umí a jakým způsobem může uživateli pomoci s daným problémem. Při opakovaném spuštění aplikace se uvítací obrazovka již zobrazovat nesmí [58].

Dalším úkolem po prvním spuštění je přihlášení uživatele. Přihlášení uživatele je vyžadováno ze dvou důvodů: zabezpečení cloudové databáze a synchronizace pokroku uživatele v aplikaci napříč více zařízeními. Uživatel se může přihlásit přes Google účet, nebo kliknout na tlačítko přihlásit se později, čímž se pro něj vytvoří anonymní účet. Anonymní účet neumožňuje synchronizaci aplikace napříč více zařízeními, avšak uživatel má možnost anonymní účet proměnit v normální účet později v nastavení aplikace.

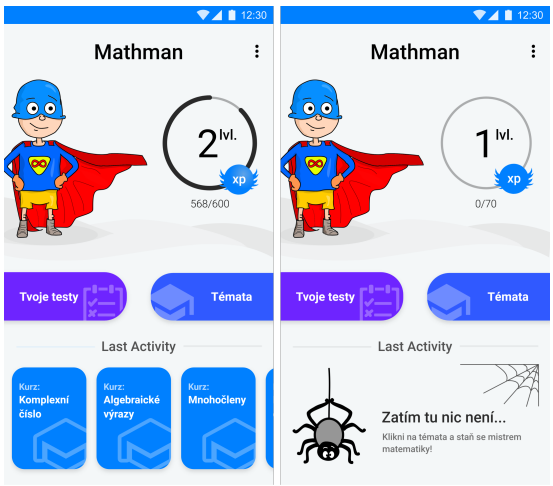
Prvotní spuštění aplikace by mělo zabrat co nejméně času a uživatel by se měl co nejrychleji dostat do hlavní části aplikace. Z tohoto důvodu je v aplikaci Mathman uvítací a přihlašovací obrazovka spojená do jedné, viz obr. 8.1.



Obr. 8.1: Grafický návrh prvního spuštění aplikace.

8.2 Domovská obrazovka

Po spuštění aplikace (mimo prvotního) se uživatel dostane na domovskou obrazovku. Domovská stránka slouží jako hlavní rozcestník aplikace, ze kterého se uživatel může dostat do menu témat, menu vlastních testů, nastavení a dalších destinací. Ve spodní části obrazovky je pak seznam posledních kurzů a testů, ke kterým se tak uživatel může rychle vrátit. Grafický návrh domovské obrazovky je vidět na obr. 8.2.

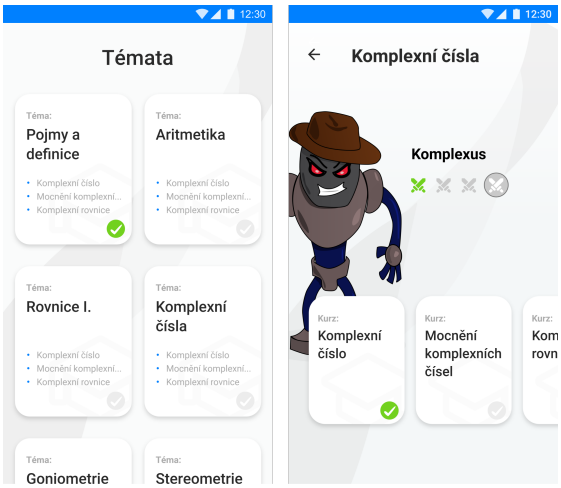


Obr. 8.2: Grafický návrh domovské obrazovky.

8.3 Členění matematického obsahu

Matematický obsah je v aplikaci členěn na témata a kurzy. Téma představuje větší matematický oddíl např. Aritmetika, Rovnice apod. Každé téma obsahuje několik kurzů, které představují již konkrétní matematické okruhy, například Lineární rovnice, Kvadratické rovnice apod.

Seznam témat je zobrazen v menu Témata dostupného z domovské obrazovky. Z menu Témata se uživatel dostává do menu konkrétního tématu, které obsahuje seznam kurzů doplněných o závěrečný souboj. Úspěšným dokončením závěrečného souboje je dané téma splněno. Grafický návrh menu členících matematický obsah je vidět na obr. 8.3.

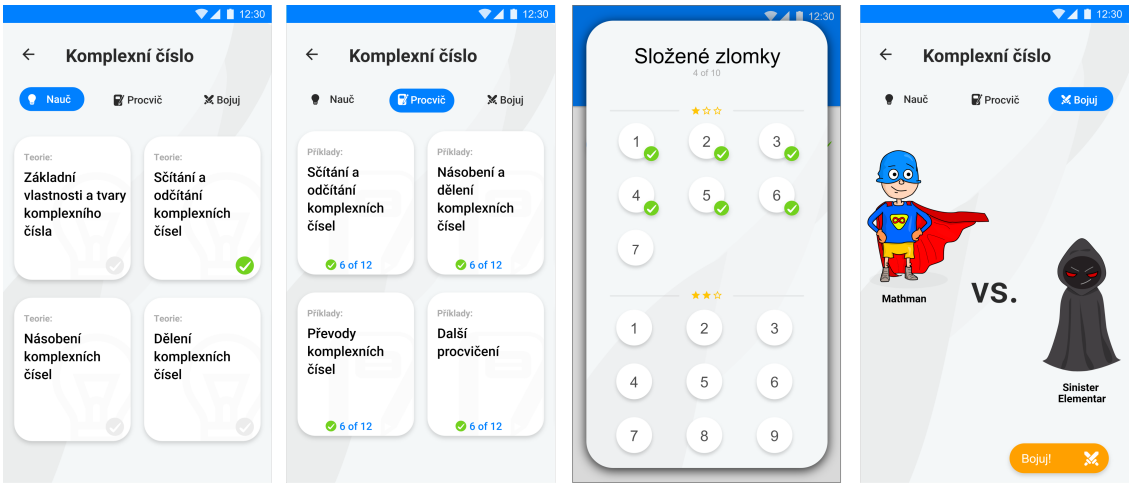


Obr. 8.3: Grafický návrh členění matematického obsahu.

8.4 Menu kurzu

Každý matematické kurz je rozdělen na tři části s názvy Nauč, Procvič a Bojuj. Část Nauč obsahuje seznam interaktivních teoretických materiálů. Menu Procvič obsahuje seznam sekcí příkladů, po jejímž rozkliknutí se objeví dialogové okno se seznamem příkladů v dané sekci rozdělených podle obtížnosti. Z menu Bojuj se spouští test ve formě souboje se zápornou postavou daného kurzu. Úspěšným zvládnutím souboje je kurz dokončen.

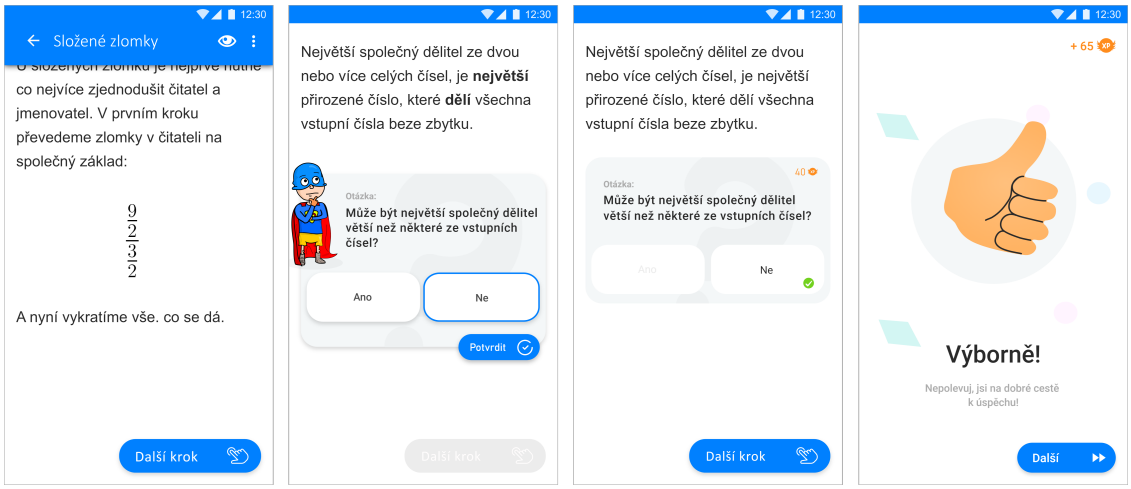
Navigace mezi částmi kurzu Nauč, Procvič a Bojuj je zajištěna pomocí horního záložkového menu. Grafický návrh menu kurzu je vidět na obr. 8.4.



Obr. 8.4: Grafický návrh menu kurzu.

8.5 Teorie

Matematická teorie je uživatelům prezentována skrze interaktivní textové materiály. Teorie je rozdělena na krátké kroky, které uživatel postupně odhaluje kliknutím na tlačítko další krok. Pro zvýšení zapojení uživatelů jsou kroky proloženy otázkami různých typů, které uživatel musí odpovědět. Za každou zodpovězenou otázku dostává uživatel body. V případě, že uživatel již někdy teorii prošel, má možnost odhalit všechny kroky naráz tlačítkem oka v horní části obrazovky. Pokud uživatel našel chybu, v horní části obrazovky má možnost chybu nahlásit. Grafický návrh interaktivních teoretických materiálu je zobrazen na obr. 8.5.

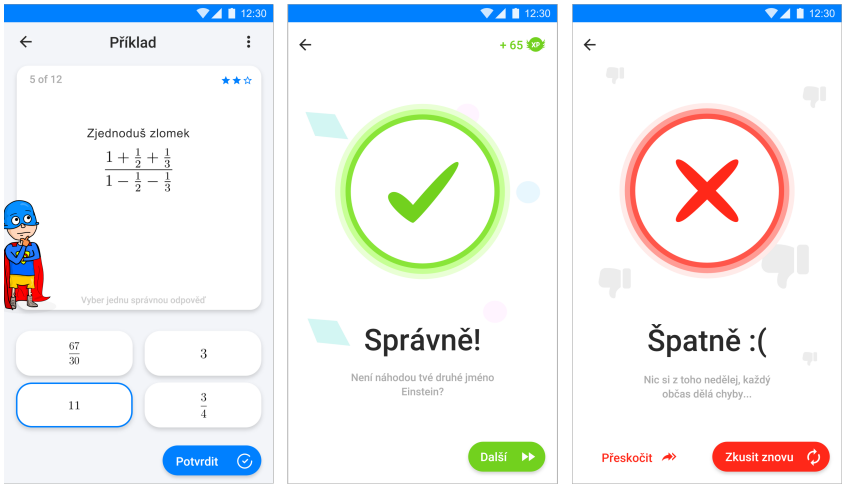


Obr. 8.5: Grafický návrh interaktivních teoretických materiálů.

8.6 Příklady

Příklady jsou dostupné ze sekce kurzu s názvem Procvič. Zadání příkladu je složeno z částí různých typů (text, latexová rovnice, obrázek atd.) zobrazených v horní části obrazovky. Ve spodní části obrazovky je prostor pro odpověď, která může být různých druhů (výběr jedné otázky ze čtyř, zadání textové odpovědi apod.). Po zodpovězení příkladu se uživateli objeví obrazovka s výsledkem, zda odpověděl správně nebo špatně. Z této obrazovky se může uživatel přesunout na další příklad v řadě, nebo v případě špatné odpovědi se může vrátit a zkusit odpovědět na příklad znovu. Za správně odpovězenou otázku je uživatel odměněn body. Stejně jako tomu bylo u teorie může i zde uživatel v horní části obrazovky nahlásit chybu.

Grafický návrh obrazovek příkladu a výsledku je zobrazen na obr. 8.6.

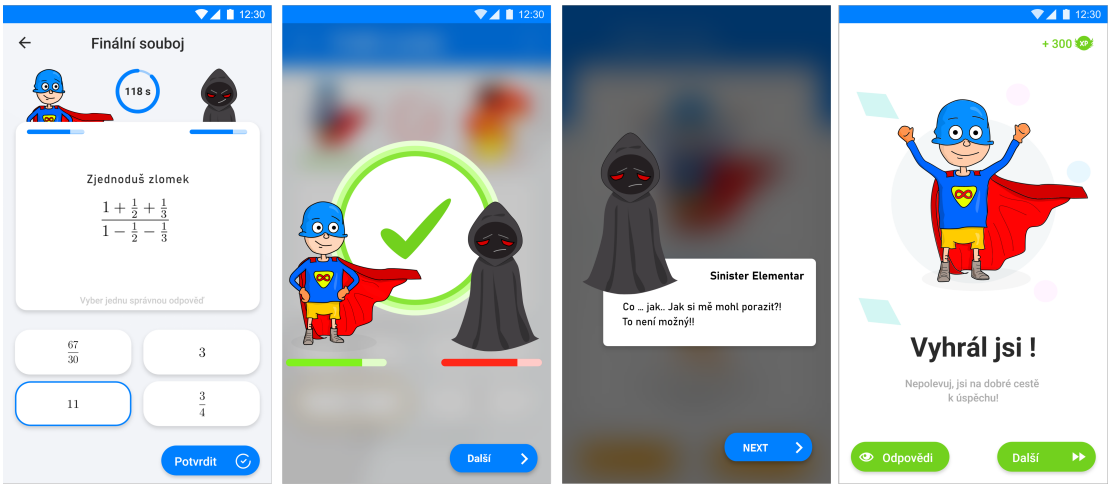


Obr. 8.6: Grafický návrh příkladu a výsledku.

8.7 Souboj

Každý kurz obsahuje finální test ve formě souboje se zápornou postavou kurzu. V rámci souboje je na každou otázku pouze omezené množství času, časomíra je zobrazena v horní části obrazovky. Prostor pro zadání a odpovědi je obdobný jako u normálních příkladů. Po každé odpovědi se zobrazí obrazovka informující o její správnosti a následně se strhne život hrdiny/záporné postavy (graficky vyznačeno pod jejich obrázky). Jakmile hrdinovi nebo záporné postavě dojde život, test je ukončen, následně je zobrazena závěrečná konverzace a výsledková obrazovka. Z výsledkové obrazovky si uživatel může projít odpovědi, nebo se vrátit zpět do menu kurzu.

Grafický návrh závěrečného souboje je zobrazen na obr. 8.7.

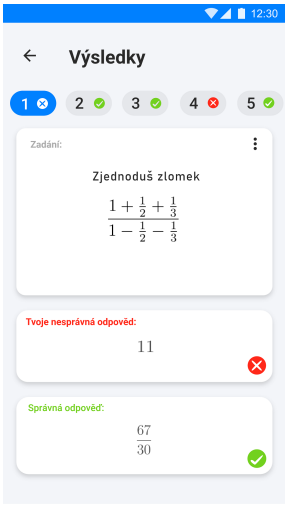


Obr. 8.7: Grafický návrh souboje a výsledku souboje.

8.8 Revize souboje/testu

V této části se uživatel může podívat na svoje odpovědi v testu nebo v souboji. Ke každému příkladu je zde zobrazeno zadání, uživatelova odpověď a případně správná odpověď. Mezi jednotlivými příklady se dá přepínat v horním záložkovém menu. U každého příkladu je možnost nahlášení chyby.

Grafický návrh revize souboje/testu je zobrazen na obr. 8.8.

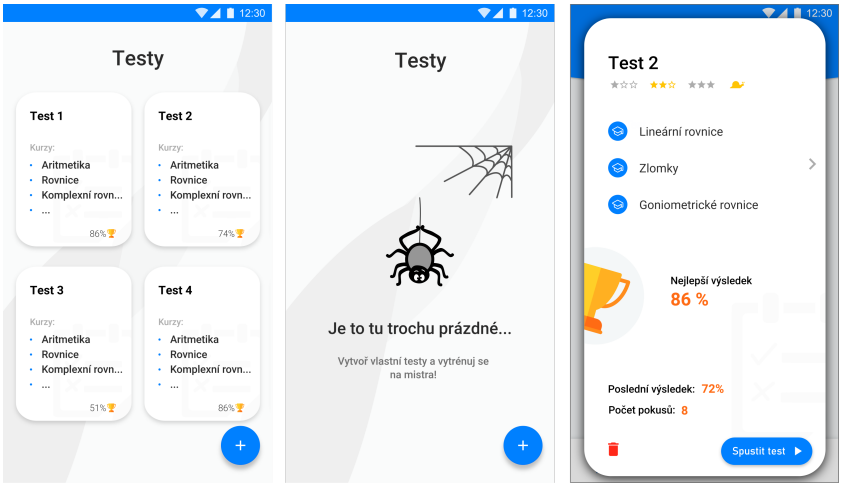


Obr. 8.8: Grafický návrh revize souboje/testu.

8.9 Menu uživatelských testů

Aplikace umožňuje rovněž vytvoření vlastních uživatelských testů. Jejich seznam je zobrazen v menu testů, do kterého se uživatel může dostat přímo z domovské obrazovky. Nové testy lze vytvářet stisknutím tlačítka plus v pravém dolním rohu.

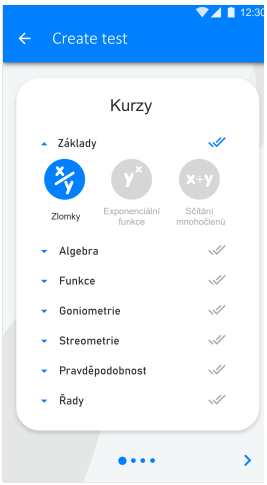
Po kliknutí na konkrétní test se objeví dialogové okno s bližšími informacemi o testu jako je seznam kurzů, náročnost, tempo, počet absolvovaných pokusů a další. Z tohoto dialogového okna lze test také spustit, případně smazat. Menu uživatelských testů je zobrazeno na obr. 8.9.



Obr. 8.9: Grafický návrh menu uživatelských testů.

8.10 Vytváření uživatelských testů

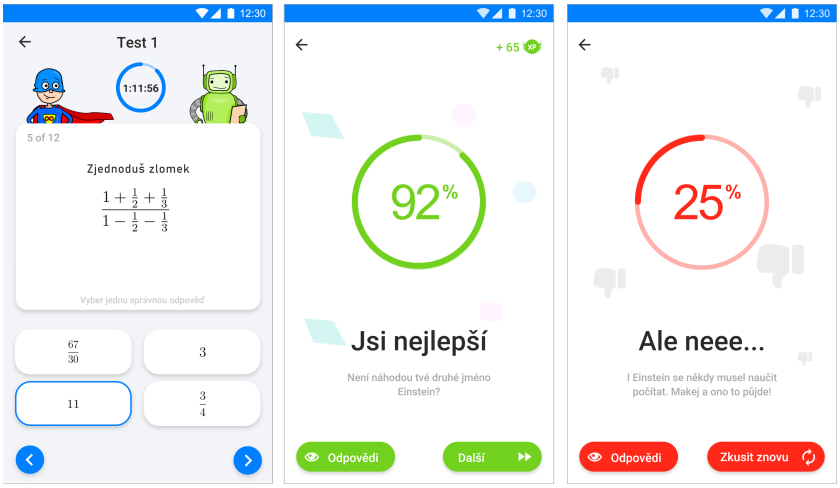
Uživatelské testy jsou v aplikaci vytvářeny prostřednictvím jednoduchého průvodce. Uživatel si tak postupně navolí z jakých kurzů se má test skládat, náročnost příkladů, tempo a název testu. Po vytvoření je test dostupný v menu uživatelských testů. Grafický návrh vytváření uživatelských testů je k vidění na obr. 8.10.



Obr. 8.10: Grafický návrh vytváření uživatelských testů.

8.11 Uživatelský test

Samotný uživatelský test se skládá z deseti náhodně vybraných příkladů dané obtížnosti z určených kurzů. Test může nebo nemusí obsahovat časomíru v závislosti na volbě uživatele. Test se v mnohém podobá souboji, nicméně na rozdíl od souboje uživatel nejprve odpovídá na všechny otázky najednou, a až následně se mu celý test vyhodnotí. Jednotlivé otázky tak může přeskočit a následně se k nim vrátit. Po dokončení testu se uživateli zobrazí výsledková obrazovka s procentuální úspěšností. Odtud se může uživatel vrátit zpět do menu uživatelských testů nebo si může prohlédnout své odpovědi, podobně jako u soubojů. Grafický návrh uživatelského testu je k vidění na obr. 8.11.



Obr. 8.11: Grafický návrh vytváření uživatelských testů.

9 Struktura dat

Cílem aplikace je zobrazovat studentům matematické materiály v podobě příkladů a interaktivních teoretických textů. Hlavní požadavky při návrhu datových struktur reprezentujících tyto materiály byla flexibilita a škálovatelnost.

Navržené datové struktury musí být flexibilní z hlediska dat, která jsou uživateli prezentovaná. Jednotlivé matematické okruhy se mezi sebou velmi liší a mohou tak vyžadovat zobrazení textů, rovnic, obrázků a dalších typů obsahu.

Struktura dat musí být navržena rovněž s ohledem na škálovatelnost. Cílem aplikace je postupně pokrýt výuku matematiky od druhého stupně základní školy až po konec střední školy. Potenciálně tak aplikace může obsahovat několik tisíc příkladů. Zároveň pak musí návrh datové struktury počítat s překlady veškerých materiálů do různých světových jazyků.

9.1 Data struktury obsahu

Matematický obsah je v aplikaci členěn na témata a kurzy. Téma představuje větší matematický oddíl např. Goniometrie, Funkce, Rovnice apod. Každé téma obsahuje několik kurzů, které představují již konkrétní matematické okruhy, například Lineární rovnice, Kvadratické rovnice apod.

Na rozdíl od ostatních dat jsou všechny datové objekty struktury v aplikaci uloženy lokálně v podobě JSON souborů a parsovány za běhu aplikace. Výhodou tohoto řešení je zrychlené načítání základních menu, nižší náklady na cloudovou databázi a především kontrola nad zobrazovaným obsahem v různých verzích aplikace.

V případě, že by se data struktury načítala z cloudové databáze podobně jako ostatní data, musel by se implementovat mechanizmus, který by bránil zobrazení obsahu, který není v dané verzi aplikace podporován (například nový typ matematické otázky).

Menší nevýhodou tohoto řešení je, že pro přidání nového matematického obsahu je nutná aktualizace aplikace.

Struktura matematického obsahu aplikace je zastoupena objekty *Structure*, *Topic*, *CoursePreview*, *Course* a *CourseExampleSection*.

9.1.1 Datový objekt Structure

Structure je jednoduchý datový objekt obsahující informaci o jazyku a seznam témat v podobě datových objektů *Topic*. Hlavním úkolem datového objektu Structure je datové vyjádření 1. úrovně menu zobrazující seznam témat.

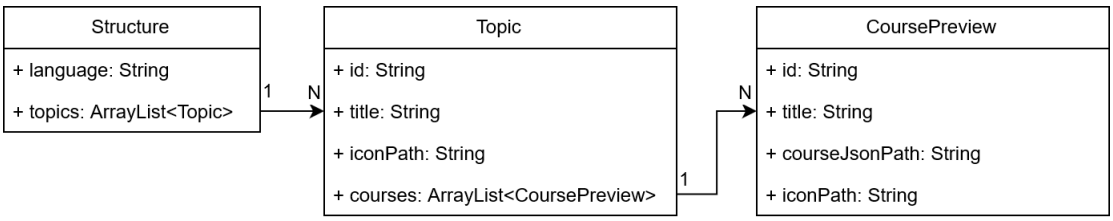
9.1.2 Datový objekt Topic

Datový objekt *Topic* představuje základní informace o daném Tématu. Obsahuje v sobě položky jako id, seznam náhledů kurzů v podobě objektů *CoursePreview*, ikonu, titulek a další pomocné informace.

9.1.3 Datový objekt Course Preview

Datový objekt *CoursePreview* obsahuje základní informace o kurzech, které je nutné znát v druhé úrovni menu, zobrazující seznam kurzů v daném tématu. Dále pak obsahuje cestu k JSON souboru popisujícího strukturu daného kurzu.

Datové objekty *Structure*, *Topic* a *CoursePreview* se parsují z jednoho JSON souboru, který je přímo součástí aplikace. Vztahy mezi těmito datovými objekty jsou zobrazeny na obr. 9.1.



Obr. 9.1: UML diagram datových objektů struktury.

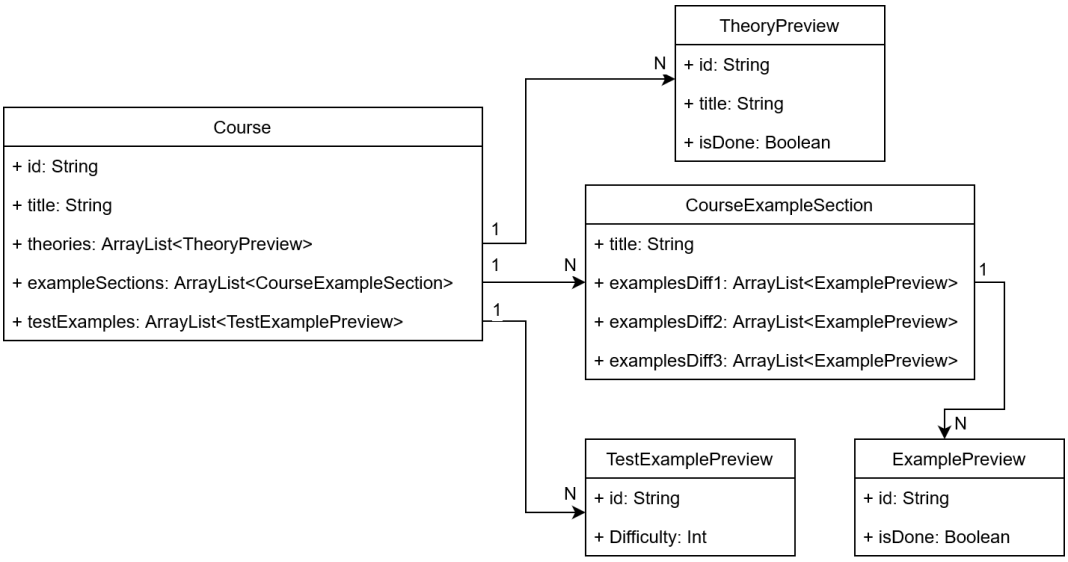
9.1.4 Datový objekt Course a CourseExampleSection

Datový objekt *Course* obsahuje veškeré informace o konkrétním kurzu včetně id, titulku a struktury matematického obsahu kurzu.

Matematický obsah kurzu se dělí na 3 části: teorie, příklady a testovací příklady. Teorie je vyjádřena jako seznam identifikátorů teoretických materiálů. Normální příklady jsou v datovém objektu *Course* sdružovány do sekcí, vyjádřených datovým objektem *CourseExampleSection*. Tento objekt obsahuje kromě názvu sekce také seznamy identifikátorů příkladů třech různých obtížností. Testovací příklady již do sekcí sdružovány nejsou a jsou tak vyjádřeny stejně jako teorie seznamem identifikátorů.

Identifikátory jsou jednoduché objekty, které sdružují id daného materiálu podle kterého se stahují z cloudové databáze a další doplňující informace (například název teorie, obtížnost příkladu apod.).

Každý kurz je vyjádřen vlastním JSON souborem, který je přímo součástí aplikace. Struktura datových objektů popisující kurz je vidět na obr 9.2.



Obr. 9.2: UML diagram datových objektů struktury kurzu.

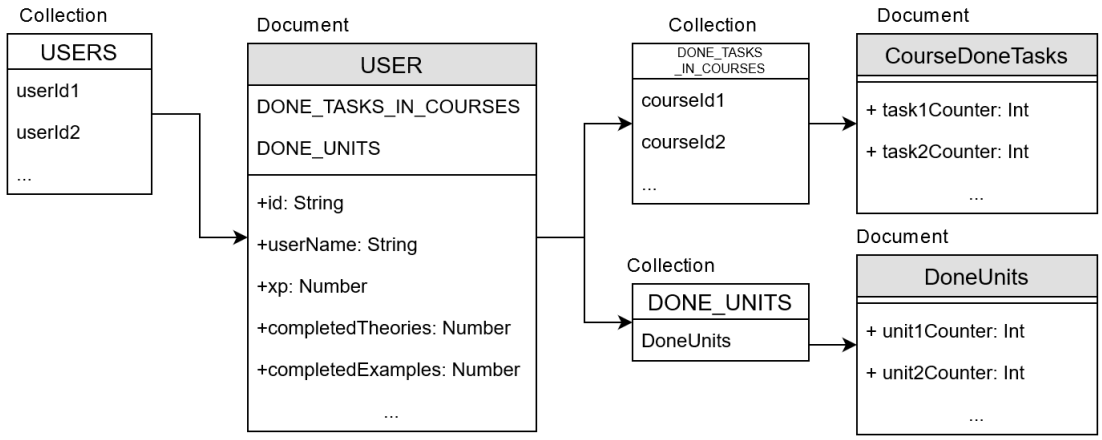
9.2 Data uživatele

Data uživatele sdružují základní informace o uživateli a o jeho pokroku v aplikaci. Uložena jsou v cloudové databázi, přičemž se jedná o jedinou sekci databáze, ke které má uživatel právo zápisu. Díky uložení uživatelských dat v cloudové databázi je možné synchronizovat pokrok uživatele napříč několika zařízeními.

Uživatelská data jsou v cloudové databázi představována dokumentem *User*, který je dostupný přes id uživatele. Dokument obsahuje základní informace včetně id, jména, počtu bodů a statistik (čas strávený v aplikaci, počet spočtených příkladů apod.).

Dokument uživatele dále obsahuje dvě kolekce: *DONE_TASK_IN_COURSES* a *DONE_UNITS*. Cílem těchto kolekcí je zaznamenání pokroku uživatele, tedy které sekce aplikace uživatel již splnil.

Struktura dat uživatele v databázi Cloud Firestore je znázorněna na obr. 9.3.



Obr. 9.3: Schéma dat uživatele v Cloud Firestore databázi.

9.3 Data teorie

Data teorie jsou uložena v cloudové databázi Cloud Firestore, přičemž uživatel k ní má pouze právo čtení. K jednotlivým dokumentům teorie se přistupuje přes jazykovou mutaci a id.

Samotný dokument teorie obsahuje id, titulek a seznam kroků.

9.3.1 Data kroku teorie

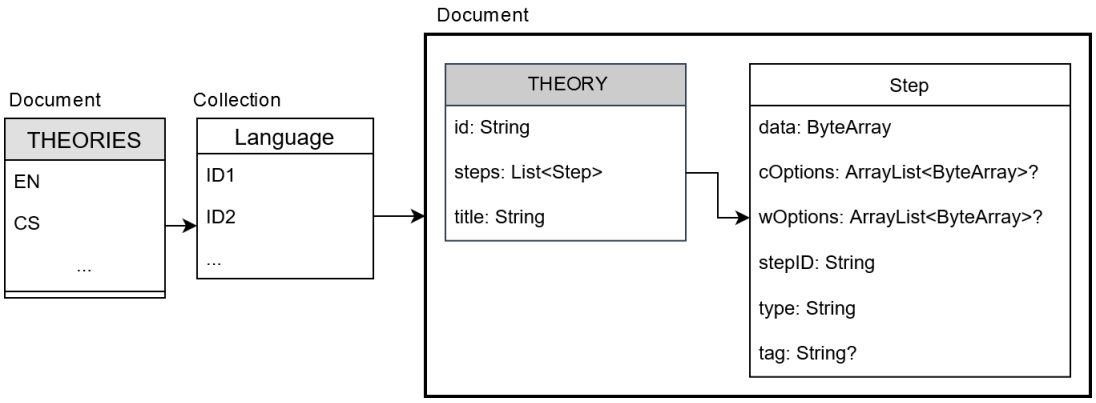
Teorie se skládá z jednotlivých kroků, které musí mít velmi flexibilní datovou podobu, která je schopna pokrýt všechny možné typy kroků (text, obrázek, různé druhy otázek, apod.). Zároveň je nutné počítat s vytvářením dalších typů kroků v budoucnu.

Proměnná s názvem *data* obsahuje hlavní obsah k zobrazení, přičemž se může jednat o text, obrázek nebo zadání otázky. Data jsou uložena přímo ve formě bajtů, které jsou následně v aplikaci castovány na daný typ (string, bitmapa atd.) v závislosti na typu kroku specifikovaném v proměnné s názvem *type*.

Proměnné *cOptions* a *wOptions* představují správné/špatné odpovědi pro případ, že daný krok je typu otázka.

Proměnná s názvem *tag* slouží převážně jako rezerva pro potřebu doplnění informací k nějakému typu kroku (například u otázky může specifikovat, zda se má uživateli otevřít textová nebo číselná klávesnice).

Strukturu dat teorie v databázi Cloud Firestore je možné vidět na obr 9.4.



Obr. 9.4: Schéma dat teorie v Cloud Firestore databázi.

9.4 Data příkladů

Rovněž data příkladů jsou uložena v cloudové databázi Cloud Firestore a podobně jako tomu bylo u dat teorie, k nim má uživatel pouze práva čtení. K příkladům se přistupuje přes jazykovou mutaci a id.

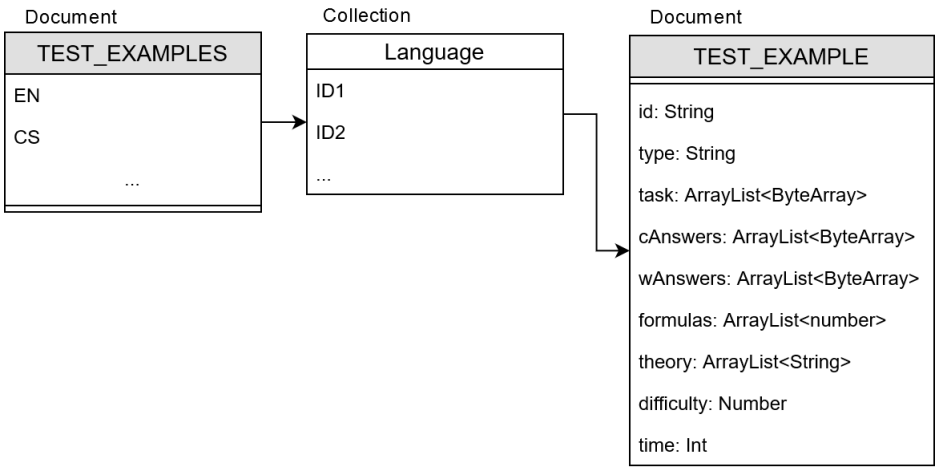
Při návrhu datové struktury příkladů bylo dbáno především na flexibilitu, neboť musí být počítáno s různými typy příkladů (různá data v zadání, různý počet a typ odpovědí), které jsou specifikovány proměnnou *type*.

Proměnná *task* obsahuje zadání v podobě seznamu částí zadání, přičemž každá část je v podobě bajtů. To umožňuje sestavovat zadání z různých typů dat, například text + obrázek.

Proměnné *cAnswers* a *wAnswers* obsahují správné, respektive špatné odpovědi na otázku. Jejich podoba seznamů polí bajtů umožňuje pro různé druhy otázek mít různý počet a druh odpovědí.

Databáze obsahuje dva typy příkladů: Normální a Testové. Po datové stránce jsou téměř totožné, pouze testové příklady obsahují navíc proměnnou *time* určující čas na vypočtení příkladu.

Podoba příkladů v databázi Cloud Firestore je vidět na obr. 9.5.



Obr. 9.5: Schéma dat testovacích příkladů v Cloud Firestore databázi.

10 Implementace

Implementace aplikace probíhala v jazyku Kotlin v oficiálním vývojovém prostředí Android studio. Minimální podporovaná verze OS Android byla zvolena 5.0 (API 21), díky čemuž aplikace podporuje (v době psaní této diplomové práce) 94.1 % aktivních Android zařízení. Minimální podporovaná verze 5.0 byla zvolena především kvůli podpoře Material Design, která s sebou přináší nativní podporu elevace a stínů u všech View v uživatelském rozhraní aplikace [1].

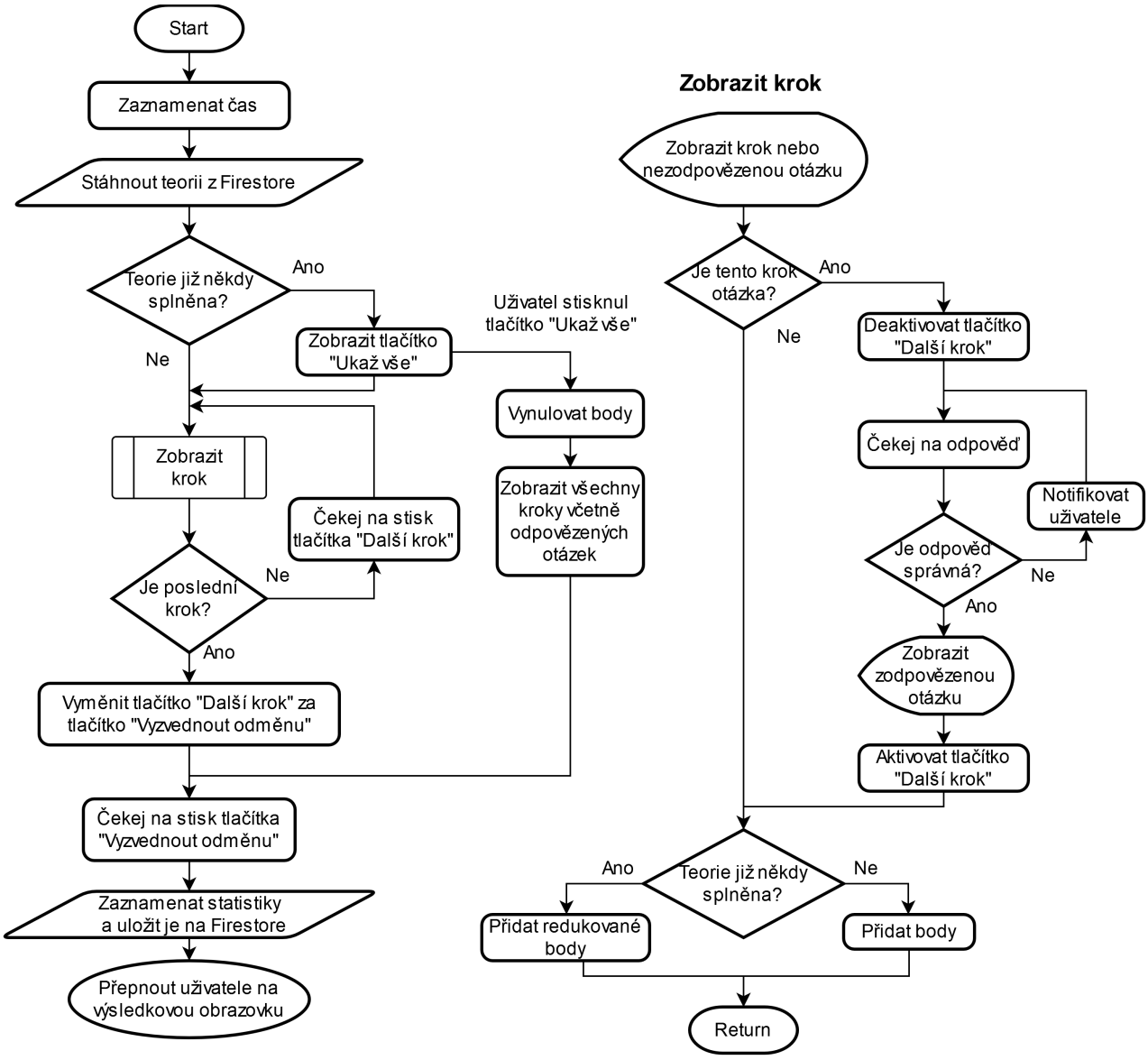
10.1 Demonstrace architektury aplikace

Vzhledem k rozsahu zdrojového kódu by bylo nepraktické popisovat implementaci všech funkcionalit aplikace. Z tohoto důvodu bude architektura aplikace demonstrována na sekci Teorie. Všechny ostatní obrazovky se pak řídí velmi podobnými pravidly.

10.1.1 Popis funkcionality teorie

Sekce teorie nejprve načte datový objekt teorie (viz kapitola 9.3) z cloudové databáze Firebase Firestore a uživateli zobrazí první krok. Zároveň se zaznamená čas pro účely statistik. Klikáním na tlačítko *"Další krok"* uživatel postupně odhaluje další kroky teorie. Některé kroky jsou typu otázka. Jakmile se otázka zobrazí, tlačítko *"další krok"* je neaktivní dokud uživatel správně na otázku neodpoví. Ve chvíli, kdy se uživatel dostane na konec teorie, tlačítko *"Další krok"* se promění v tlačítko *"Vyzvednout odměnu"*, která uživatele přepne na výsledkovou obrazovku teorie. Ještě před přepnutím na výsledkovou obrazovku se aktualizují statistiky pro účely gamifikace (čas strávený v teorii, počet dokončených teorií a splněné teorie).

Pokud uživatel již předtím teorii někdy splnil, má dvě možnosti. Buď může teorii procházet klasickým již popsaným způsobem krok po kroku (ale s redukovaným množstvím bodů, které může získat), nebo může využít funkcionality *"Ukaž vše"*, která zobrazí celou teorii najednou včetně odpovězených otázek. V takovém případě ale uživatel za teorii nedostane žádné body. Celou funkcionalitu teorie shrnuje vývojový diagram 10.1.



Obr. 10.1: Zjednodušený vývojový diagram funkcionality Teorie.

10.1.2 Architektura teorie

Jak již bylo řečeno v kapitole 7.4.1, sekce teorie (stejně jako všechny ostatní části aplikace) je vytvořena dle návrhového vzoru Model-View-ViewModel (MVVM), přičemž datová vrstva je rozčleněna dle návrhového vzoru Repository. Celková architektura navíc dodržuje principy Clean Architecture.

Všechny popsané třídy jsou k nalezení v elektronické příloze této práce.

Uspořádání zdrojového kódu

Jednotlivé třídy a interfacy aplikace jsou uspořádány do pěti hlavních balíčků - *db*, *di*, *model*, *ui* a *util*. Balíček *db* obsahuje všechny třídy spojené s lokální databází

Room (konfigurační třídu, interfacy pro přístup k datům a další). Balíček *di* sdružuje všechny komponenty spojené s konfigurací Dependency Injection pomocí knihovny Dagger 2. V balíčku *model* nalezneme celou datovou vrstvu aplikace. Balíček *ui* obsahuje logickou a grafickou vrstvu aplikace rozčleněnou podle obrazovek. Důvodem pro sdružení logické a grafické vrstvy do jednoho balíčku je to, že vrstvy spolu velmi blízce souvisí. Konkrétní třídy logické vrstvy zpravidla ovládají konkrétní třídy grafické vrstvy a proto je výhodné z hlediska efektivity práce mít třídy těchto vrstev u sebe, rozčleněné podle funkcionalit. Veškeré pomocné třídy jsou pak umístěny v balíčku *util*.

Grafická vrstva

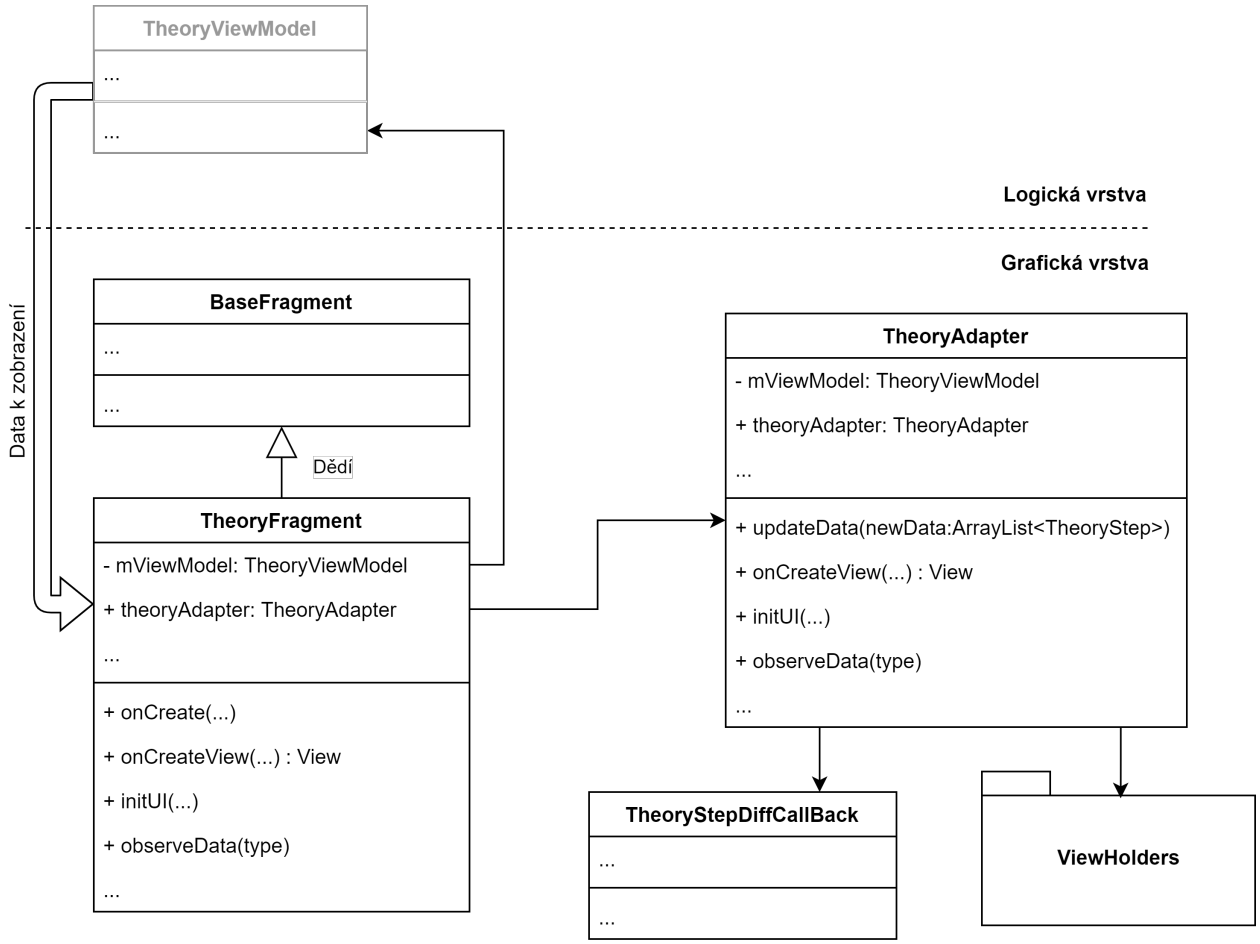
Grafická vrstva fragmentu teorie se skládá z tříd *TheoryFragment*, *TheoryAdapter*, *TheoryStepDiffCallBack* a ViewHolderů pro adaptér. Kromě těchto tříd se do grafické vrstvy řadí ještě XML soubory popisující layout daného fragmentu a všech Viewholderů. V souladu s pravidly návrhového vzoru MVVM popsanými v kapitole 2.1.3 obsahují třídy grafické vrstvy minimální množství logických operací a jejich jedinou starostí je zobrazení dat a zachycení uživatelské interakce.

Hlavní třídou grafické vrstvy sekce teorie je *TheoryFragment*. Jedná se vstupní třídu celé sekce, která komunikuje s logickou vrstvou, vytváří a ovládá uživatelské rozhraní a zobrazuje data v RecyclerView pomocí adaptéru *TheoryAdapter*. Další důležitou funkcí třídy *TheoryFragment* je navigace na další obrazovky, která probíhá s pomocí knihovny Navigation Component.

Další důležitou komponentou grafické vrstvy sekce teorie je *TheoryAdapter*. Jednotlivé kroky teorie jsou uživateli zobrazovány v RecyclerView, což je seznam s recyklovatelnými položkami. *TheoryAdapter* slouží k transformaci dat do podoby, kterou dokáže RecyclerView zobrazit.

Jednotlivé kroky jsou uvnitř RecyclerView zobrazovány prostřednictvím ViewHolderů a třídy *TheoryStepDiffCallback*. Viewholdery vytvářejí a ovládají grafické komponenty v jednotlivých zobrazených krocích. Teorie v době psaní této práce podporovala 11 typů kroků (textový krok, obrázkový krok, latexový krok a různé typy otázek), přičemž ke každému typu kroku existuje patřičný Viewholder. Třída *TheoryStepDiffCallback* slouží k vypočtení změny zobrazovaných dat tak, aby se při každé aktualizaci dat nepřekresloval celý RecyclerView, ale pouze položky, které se změnily.

Zjednodušená struktura komponent grafické vrstvy teorie je naznačena na obrázku 10.2.



Obr. 10.2: Zjednodušená struktura komponent grafické vrstvy teorie.

Logická vrstva

Logická vrstva sekce teorie se sestává z ViewModelu *TheoryViewModel*, třídy popisující stav UI s názvem *TheoryState* a několika tříd Usecasů. Hlavním úkolem logické vrstvy v souladu s architekturou MVVM je provádění veškerých logických úloh nad daty a jejich příprava pro zobrazení.

Hlavní třídou logické vrstvy sekce teorie je *TheoryViewModel*. Jedná se o jedinou třídu z logické vrstvy, se kterou přímo komunikuje grafická vrstva. Přímá komunikace ale probíhá pouze z grafické vrstvy do logické, neboť ViewModel nesmí mít žádné informace o existenci grafické vrstvy. V opačném případě by hrozily úniky paměti. Data určená pro grafickou vrstvu tak ViewModel přidruženému Fragmentu neposílá přímo, ale pouze je vystavuje k odběru prostřednictvím observovatelného kontejneru typu LiveData, který vystavuje objekty typu *TheoryState*. *TheoryState* zastřešuje veškerá data, která grafická vrstva potřebuje. Obsahuje tak nejen data samotné teorie, ale i současný status sekce (načítání, aktualizace nebo chybový stav). Jedinou výjimkou je informace o správnosti odpovědi na otázku teorie, která je pro grafickou

vrstvu vystavována zvlášť.

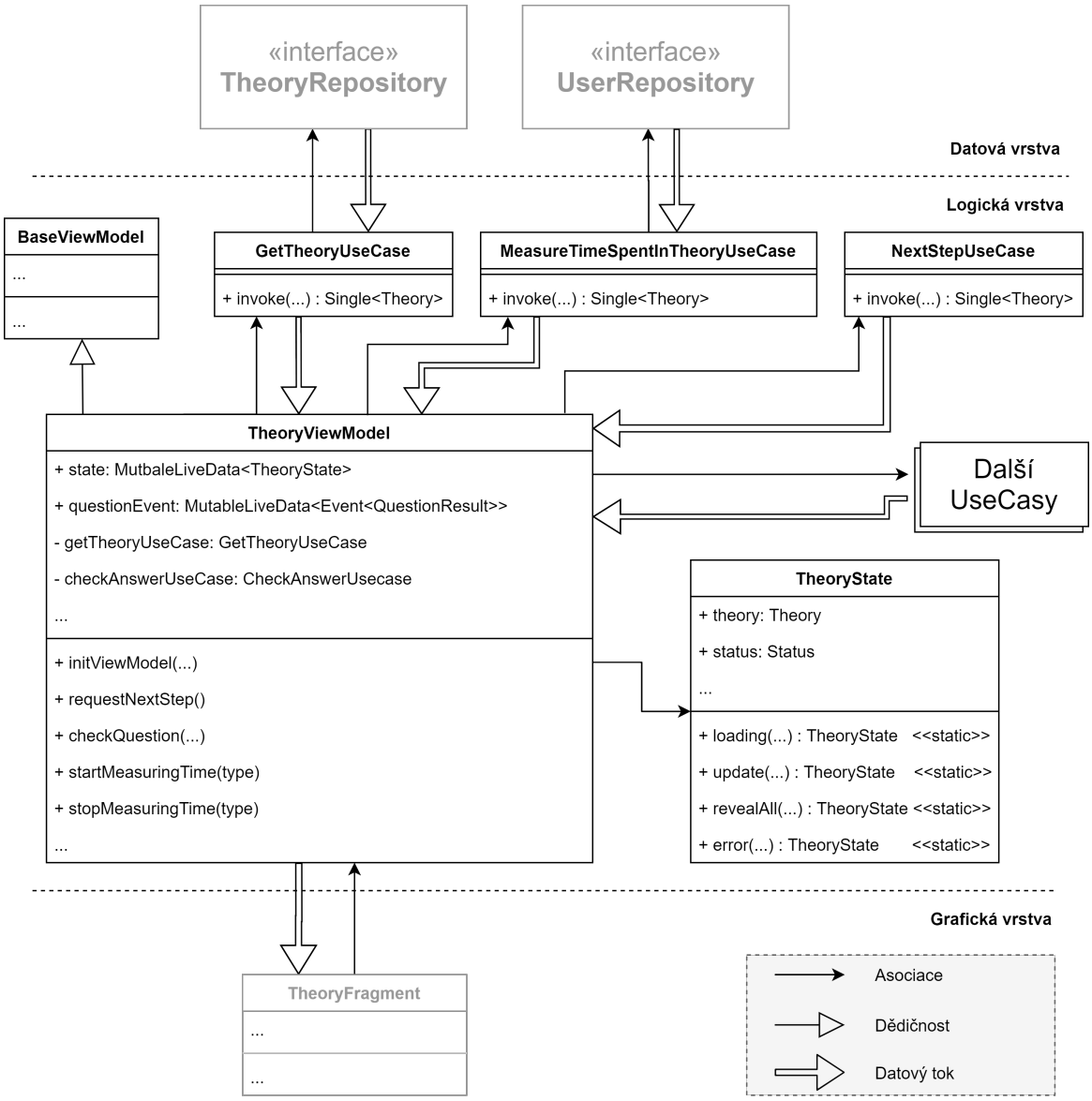
V obecném případě architektury MVVM ViewModely řeší logické operace nad daty a přímo komunikují s datovou vrstvou. Nicméně u rozsáhlejších aplikací hrozí nebezpečí, že se z ViewModelů stane příliš rozsáhlá třída s příliš mnoha zodpovědnostmi. Dle doporučení Clean Architecture je tak vhodné přesunout veškeré operace do samostatných komponent - UseCasů.

Každý UseCase by měl mít pouze jeden jediný úkol. Sekce teorie tak obsahuje UseCasy sloužící například k vyžádání dat od datové vrstvy, vyhodnocení správnosti odpovědi na otázku nebo k zaznamenání času stráveného v sekci teorie. ViewModel pak slouží pouze jako prostředník mezi grafickou vrstvou a UseCasy.

Celková komunikace pak probíhá následujícím způsobem: Grafická vrstva podá žádost k ViewModelu, ViewModel žádost deleguje na konkrétní UseCase, který mu vrátí potřebná data a tato data ViewModel poskytne grafické vrstvě k odběru. Zároveň je potřeba podotknout, že podle Dependency Rule z Clean Architecture by jednotlivé UseCasy neměly mít žádnou informaci o existenci ViewModelu, data jim tak mohou předávat pouze pomocí observovatelných objektů (datových toků) nebo prostřednictvím interfaců (portů).

Veškerá komunikace mezi ViewModelem, UseCasy a datovou vrstvou probíhá prostřednictvím reaktivních datových toků knihovny RxJava 2, díky čemuž je možné všechny logické a datové operace snadno vykonávat na vláknech na pozadí a nezatěžovat tak hlavní vlákno starající se o vykreslování uživatelského rozhraní.

Zjednodušená struktura komponent logické vrstvy sekce teorie je znázorněna na obrázku 10.3.



Obr. 10.3: Zjednodušená struktura komponent logické vrstvy teorie.

Datová vrstva

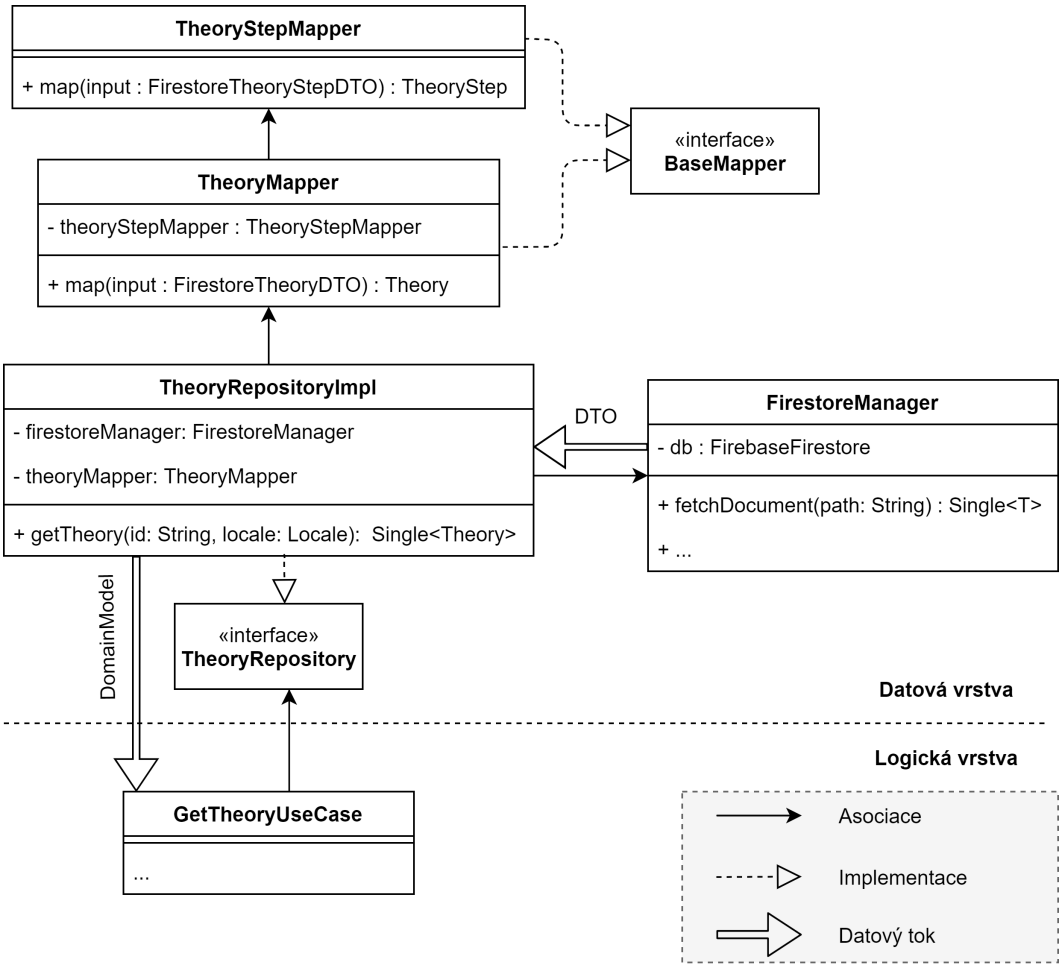
Datová vrstva sekce teorie se sestává ze dvou oddělených částí, první část slouží ke stažení teorie, druhá část slouží k přístupu k uživatelovým informacím a v tomto případě k aktualizování uživatelových statistik. Obě části fungují na stejném principu, v rámci demonstrace architektury aplikace tak bude popsána pouze část sloužící ke stažení teorie.

Datová vrstva obstarávající teorii je k nalezení v adresáři *model.theory*. Skládá se z několika důležitých komponent, které následují návrhový vzor Repository (viz kapitola 2.2). Třída *Theory*, která je k nalezení v adresáři *model.theory.domainmodel*, je DomainModel reprezentující teorii, která je používána napříč aplikací od grafické

až po datovou vrstvu. V Adresáři *dto* se nachází třída *FirestoreTheoryDTO*, která představuje komponentu DTO (Data Transfer Object), která koresponduje s podobou dat teorie v cloudové databázi. V celé aplikaci jsou DomainModely a DTO v souladu s doporučeními Clean Architecture striktně oddělovány do vlastních tříd, a to i v případě, že jsou identické. Důvodem je oddělení jádra aplikace od implementačních detailů v podobě konkrétní používané databáze. Při stahování a ukládání dat je potřeba mezi sebou DTO a DomainModely převádět. K tomu slouží třídy v adresáři *model.theory.mappers* – *TheoryMapper* a *TheoryStepMapper*.

Samotné stahování dat teorie z cloudové databáze probíhá ve třídě *FirestoreManager*, která v návrhovém vzoru Repository představuje komponentu DataSource. DataSources (v aplikace pojmenované jako *[ZdrojDat]Manager*) jsou pro datové vrstvy všech sekcí společné a proto jsou umístěny společně v adresáři *model.managers*.

Centrální komponentou datové vrstvy je komponenta Repository v podobě třídy *TheoryRepositoryImpl*, ke které je z logické vrstvy přistupováno prostřednictvím interfacu *TheoryRepository*. Repository slouží jako prostředník mezi jednotlivými zdroji dat a logickou vrstvou. Zároveň pomocí Mapperů provádí trasformaci dat z DTO na DomainModely. V souladu s pravidly Clean Architecture nemá *TheoryRepository* žádné informace o existenci logické vrstvy, přičemž data jsou ji předávána striktně reaktivními toky dat pomocí knihovny RxJava2. Všechny komponenty datové vrstvy operují výlučně na vláknech na pozadí. Struktura tříd datové vrstvy teorie je znázorněna na obrázku 10.4.



Obr. 10.4: Zjednodušená struktura komponent datové vrstvy teorie.

10.1.3 Příklad průběhu komunikace mezi vrstvami

Jak již bylo řečeno v popisu jednotlivých vrstev, aplikace ve velké míře využívá reaktivního programování v podobě datových toků. Velkou výhodou tohoto přístupu je automatická propagace změn v datech a velmi nízká provázanost jednotlivých komponent aplikace. Na příkladu stažení dat teorie bude nyní demonstrována komunikace mezi jednotlivými vrstvami aplikace.

Požadavek na data teorie je vznesen ve Fragmentu *TheoryFragment* zavoláním funkce *initViewModel()*, jejíž argumenty obsahují všechny informace nutné k nalezení daného objektu v cloudové databázi Firebase Firestore (id teorie a jazyk). Po vznesení požadavku se *TheoryFragment* ve funkci *observeData()* přihlásí k odběru dat z proměnné *state* typu LiveData uvnitř ViewModelu.

V LiveData kontejneru je nastavená inicializační hodnota *TheoryState* se statusem *loading*, kterou Fragment okamžitě po přihlášení k odběru dat obdrží a zobrazí tak animaci načítání. Požadavek na data *TheoryViewModel* deleguje na třídu

GetTheoryUseCase, který prostřednictvím funkce *invoke()* navrací datový tok typu *Single<Theory>*, ke kterému se *TheoryViewModel* přihlásí k odběru.

Požadavek na data teorie třída *GetTheoryUseCase* dále deleguje na datovou vrstvu, konkrétně na třídu *TheoryRepositoryImpl*, ke které přistupuje prostřednictvím interfacu *TheoryRepository*. Data z Repository jsou opět získávána prostřednictvím datového toku typu *Single<Theory>*, nicméně datový tok je od tohoto bodu přesunut na vlákno na pozadí pomocí operátoru *compose*.

Třída *TheoryRepository* obsahuje instanci třídy *FirestoreManager*, která má přímý přístup ke cloudové databázi, v níž se data teorie nacházejí. *FirestoreManager* obsahuje veřejnou funkci *fetchDocument()*, která jako argument přijímá cestu k danému Firestore dokumentu a navrací generický typ (v tomto případě *FirestoreTheoryDTO*) zabalený v datovém toku typu *Single<T>*. Tímto je vytvořen datový tok od třídy *FirestoreManager* až po *TheoryViewModel*, kde jsou data odebírána.

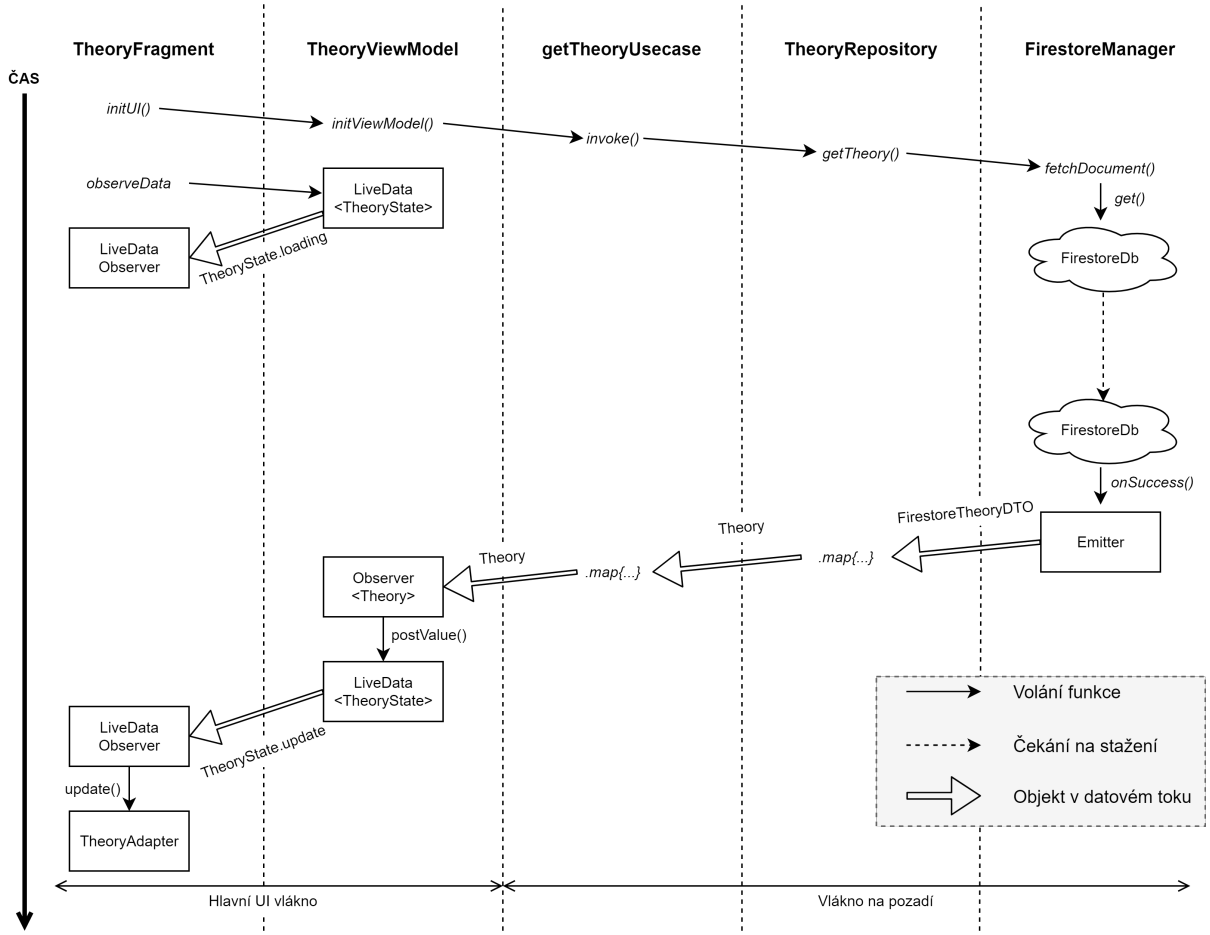
FirestoreManager nyní asynchronně data stáhne z cloudové databáze Firestore a naparsuje je do podoby datového objektu typu *FirestoreTheoryDTO*. Výsledný objekt vloží do datového proudu pomocí funkce *emitter.onSuccess()*. V případě, že se vyskytne jakákoliv chyba, do datového toku pošle chybu v podobě objektu typu *Throwable* pomocí funkce *emitter.onError()*.

Data se nyní prostřednictvím datového toku dostanou do *TheoryRepositoryImpl*, kde jsou pomocí operátoru *map* a třídy *TheoryMapper* transformována z DTO na DomainModel (z objektu typu *FirestoreTheoryDTO* na objekt typu *Theory*). Odsud se objekt *Theory* skrze datový tok dostane do třídy *GetTheoryUseCase*, kde je pomocí operátoru *map* modifikován pro potřeby zobrazení. Kromě toho je zde objekt *Theory* přesunut z vlákna na pozadí na hlavní vlákno aplikace a poslán skrze datový tok do finální destinace v podobě třídy *TheoryViewModel*. *TheoryViewModel* obdrží objekt *Theory* z datového toku prostřednictvím lambda observeru v operátoru *subscribe*. ViewModel následně vloží objekt *Theory* do datového objektu *TheoryState* a ten je poté vložen do LiveData kontejneru.

Jakmile nastane změna uvnitř LiveData kontejneru, je o této změně notifikován odběratel těchto dat - *TheoryFragment*. Ve chvíli, kdy Fragment obdrží data teorie, skryje animaci načítání a data teorie předá adaptéru *TheoryAdapter* k zobrazení v RecyclerView.

V případě, že by se stažení dat nepodařilo a *FirestoreManager* by do datového toku odeslal error objekt *Throwable*, ViewModel by do LiveData kontejneru vložil *TheoryState* se statusem *Error* a fragment by zobrazil chybovou hlášku uživateli.

Průběh celé komunikace je naznačen na obrázku 10.5.



Obr. 10.5: Komunikace mezi vrstvami v čase při stažení teorie.

10.2 Testování aplikace

Aplikace byla testována dvěma druhy testů - manuálními a App crawler testy. Manuální testování aplikace probíhalo průběžně po celou dobu vývoje. Kdykoliv byla dokončena nová funkcionalita, aplikace byla testována na několika fyzických a emulovaných zařízeních vývojáře s různou velikostí displeje a různou verzí OS Android.

Po dokončení vývoje hlavních částí byla aplikace předaná k otestování přibližně dvěma desítkám alfa-testerů. Aplikace byla mezi ně distribuována prostřednictvím alfa kanálu obchodu Google Play, který zajišťuje, že aplikaci si mohou stáhnout pouze pozvaní uživatelé. Cílem tohoto testování bylo nejen zjištění chyb v aplikaci, ale rovněž zjištění problémů s UX jako například nelogické rozložení tlačítek, příliš komplikované první spuštění apod. Zpětná vazba od alfa-testerů probíhala na osobní bázi, přičemž aplikace byla takto v 6 iteracích postupně upravena.

Poslední kolo manuálního testování probíhalo s pomocí beta-testerů, kterým byla aplikace distribuována přes otevřený beta kanál obchodu Google Play. Do beta tes-

tování se tak mohl zapojit kdokoliv i bez pozvání. Na rozdíl od normálních uživatelů není beta-testerům umožněno psát na Google Play recenze. Zpětná vazba od beta-testerů probíhala prostřednictvím sociálních sítí a e-mailové komunikace. Beta-testování odhalilo několik menších chyb, které byly iterativně opraveny (například problémy se změnou jazyka v aplikaci v případě, že uživatel má telefon nastaven do angličtiny).

Ve všech testovacích a produkčních verzích aplikace se navíc nachází modul Firebase Crashlytics, který v případě pádu aplikace zasílá do služby Firebase hlášení včetně verze aplikace, typu zařízení a logů. Díky tomuto modulu je tak možné sledovat spolehlivost aplikace v každé vydané verzi a zachytit případné problémy ještě před tím, než se negativně projeví v recenzích uživatelů.

Druhým typem testů, kterým byla aplikace podrobena jsou bez-konfigurační App crawler testy. Testy probíhaly prostřednictvím cloudové služby Firebase Robotest na dvou desítkách fyzických i emulovaných zařízeních. Výhodou těchto testů je kromě detekce míst, kde aplikace padá také měření výkonu aplikace a poukázání na místa, kde je hlavní vlákno aplikace příliš vytíženo. App-crawler testy probíhaly souběžně s manuálními testy po celou dobu vývoje aplikace.

Závěr

V teoretické části diplomové práce byl vytvořen podrobný popis často využívaných návrhových vzorů, přičemž velký důraz byl kladen především na architektonické návrhové vzory. Popsán byl rovněž i programovací jazyk Kotlin, především z hlediska jeho výhod a odlišností od programovacího jazyka Java, z něhož vychází. V teoretické části byly rovněž shrnuty další ověřené programátorské praktiky jako dependency injection, reaktivní programování a automatické testování.

V rámci praktické části diplomové práce byla vytvořena aplikace, sloužící k prokázání validity praktik popsaných v teoretické části a k demonstraci výhod využívání architektonických návrhových vzorů. Nejprve bylo zvoleno téma umožňující vytvořit dostatečně rozsáhlou aplikaci, na které by jasně vyplynuly výhody implementace architektury. Rozhodnuto tak bylo o vytvoření platformy na interaktivní doučení základoškolské a středoškolské matematiky.

Dále byly definovány základní požadavky na tuto aplikaci. Proveden byl průzkum trhu s výběrem dvou zástupců úspěšných aplikací podobného typu, u nichž byly identifikovány silné a slabé stránky. Poté byl proveden výběr vhodné architektury aplikace a databázových systémů.

V rámci návrhu grafického uživatelského rozhraní byly v programu Adobe XD vytvořeny fotorealistické návrhy všech hlavních obrazovek aplikace a blíže byla popsána a specifikována jejich funkcionalita.

Součástí návrhu aplikace je také popis datové struktury. Datový model aplikace byl navržen tak, aby umožňoval snadné přidávání nových typů matematického obsahu a okamžitou opravu příkladů při nalezení chyby bez nutnosti aktualizace aplikace. Navržený datový model rovněž počítá s jazykovými mutacemi obsahu.

V poslední kapitole byla popsána samotná implementace aplikace včetně architektury. Vzhledem k většímu rozsahu zdrojového kódu (více než 28 000 řádků) byla architektura demonstrována pouze na jedné sekci aplikace. Kromě popisu jednotlivých vrstev a komunikace mezi nimi kapitola obsahuje také popis průběhu testování.

Výsledná aplikace byla na začátku března publikována v obchodě Google Play kde si ji k 25. 5. 2020 stáhlo cca 10 000 uživatelů. Aplikace bude rozvíjena i do budoucna, v plánu je především vytvoření anglické mutace, rozšiřovaní matematického obsahu a přidávání nových funkcionalit jako například personalizovaný učební plán, nebo možnost odeslat si vypracované výpisky na email k vytištění.

Literatura

[1] Android Developers [online], 2019. USA: Google [cit. 2019-10-27]. Dostupné z: https://developer.android.com/

[2] MAHESWARI, G. a K. CHITRA, 2018. Selection of Design Pattern in each stage of Software Development. International Conference on Data Analytics & Visualization [online]. -(5), 23-29 [cit. 2019-11-17]. ISSN 2321-788X. Dostupné z: https://www.researchgate.net/profile/Jeryda_Gnanajane_Eljo/publication/335168988_International_Conference_on_Data_Analytics_Visualization/links/5d541c0ea6fdcc85f8922886/International-Conference-on-Data-Analytics-Visualization.pdf#page=35

[3] MALL, Rajib, 2018. Fundamentals Of Software Engineering. 5. ed. Delhi: PHI Learning Private Limited. ISBN 978-93-88028-03-5.

[4] GAMMA, Erich, c1995. Design patterns: elements of reusable object-oriented software. 1. Reading, Mass.: Addison-Wesley. ISBN 02-016-3361-2.

[5] HOWARD, Joe, 2017. Common Design Patterns for Android with Kotlin. In: Raywenderlich.com [online]. USA: Razeware [cit. 2019-11-17]. Dostupné z: https://www.raywenderlich.com/470-common-design-patterns-for-android-with-kotlin

[6] MEW, Kyle, 2016. Android Design Patterns and Best Practice. 1. Birmingham: Packt. ISBN 9781786465917.

[7] ELLIS, Brian, Jeffrey STYLOS a Brad MYERS, 2007. The Factory Pattern in API Design: A Usability Evaluation. 29th International Conference on Software Engineering (ICSE'07) [online]. IEEE, 2007, 29(1), 302-312 [cit. 2019-11-17]. DOI: 10.1109/ICSE.2007.85. ISBN 0-7695-2828-7. ISSN 0270-5257. Dostupné z: http://ieeexplore.ieee.org/document/4222592/

[8] RUIZ, Antonio Pachoz, 2015. Mastering Android Application Development. Birmingham: Packt. ISBN 978-1785884221.

[9] Kotlinlang [online], 2012. Russia: JetBrains [cit. 2019-11-17]. Dostupné z: https://kotlinlang.org/

[10] HABCHI, Sarra, Geoffrey HECHT, Romain ROUVOY a Naouel MOHA, 2017. Code Smells in iOS Apps: How Do They Compare to Android? 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering

and Systems (MOBILESoft) [online]. IEEE, 2017, -(-), 110-121 [cit. 2019-11-17]. DOI: 10.1109/MOBILESoft.2017.11. ISBN 978-1-5386-2669-6. Dostupné z: http://ieeexplore.ieee.org/document/7972725/

[11] BHARDWAY, Kapil, 2017. When Singleton Becomes an Anti-Pattern. In: DZone [online]. Bangalore: DZone, 12.10.2017 [cit. 2019-11-17]. Dostupné z: https://dzone.com/articles/singleton-anti-pattern

[12] PALOMBA, Fabio, Gabriele BAVOTA, Massimiliano DI PENTA, Fausto FASANO, Rocco OLIVETO a Andrea DE LUCIA, 2018. A large-scale empirical study on the lifecycle of code smell co-occurrences. Information and Software Technology [online]. 99, 1-10 [cit. 2019-10-23]. DOI: 10.1016/j.infsof.2018.02.004. ISSN 09505849. Dostupné z: https://linkinghub.elsevier.com/retrieve/pii/S0950584918300211

[13] Understanding Code Smells in Android Applications, 2016. IEEE/ACM International Conference on Mobile Software Engineering and Systems [online]. 2016, 225-236 [cit. 2019-10-23]. DOI: 10.1109/MobileSoft.2016.048. Dostupné z: https://ieeexplore.ieee.org/abstract/document/7832986/authors#authors

[14] MOHA, N., Y.-G. GUEHENEUC, L. DUCHIEN a A.-F. LE MEUR, 2010. DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Transactions on Software Engineering [online]. 36(1), 20-36 [cit. 2019-10-23]. DOI: 10.1109/TSE.2009.50. ISSN 0098-5589. Dostupné z: http://ieeexplore.ieee.org/document/5196681/

[15] SUN, Wei, Haohui CHEN a Wen YU, 2017. The Exploration and Practice of MVVM Pattern on Android Platform. Proceedings of the 2016 4th International Conference on Machinery, Materials and Information Technology Applications [online]. Paris, France: Atlantis Press, 2017(-), 1-10 [cit. 2019-10-23]. DOI: 10.2991/icmmita-16.2016.205. ISBN 978-94-6252-285-5. Dostupné z: http://www.atlantis-press.com/php/paper-details.php?id=25868250

[16] VERDECCHIA, Roberto, Ivana MALAVOLTA a Patricia LAGO, 2019. Guidelines for Architecting Android Apps: A Mixed-Method Empirical Study. 2019 IEEE International Conference on Software Architecture (ICSA) [online]. IEEE, 2019, 2019(-), 141-150 [cit. 2019-10-23]. DOI: 10.1109/ICSA.2019.00023. ISBN 978-1-7281-0528-4. Dostupné z: https://ieeexplore.ieee.org/document/8703927/

[17] LOU, Tian, 2016. A comparison of Android Native App Architecture MVC, MVP and MVVM. Espoo. Diplomová práce. Eindhoven University of Technology. Vedoucí práce Professor Heikki Saikkonen.

[18] DAOUDI, Aymen, Ghizlane ELBOUSSAIDI, Naouel MOHA a Sègla KPODJEDO, 2019. An exploratory study of MVC-based architectural patterns in Android apps. Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing - SAC '19 [online]. New York, New York, USA: ACM Press, 2019, 1711-1720 [cit. 2019-10-26]. DOI: 10.1145/3297280.3297447. ISBN 9781450359337. Dostupné z: http://dl.acm.org/citation.cfm?doid=3297280.3297447

[19] MUNTENESCU, Florina, 2016. Android Architecture Patterns Part 1: Model-View-Controller. In: Medium [online]. [cit. 2019-10-28]. Dostupné z: https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6

[20] FERNANDES, Sergio Martins a Paulo Jose MENDES, 2017. Architectural pattern for native android applications. Revista de Sistemas e Computação [online]. 2017(2), 163-178 [cit. 2019-10-26]. Dostupné z: https://revistas.unifacs.br/index.php/rsc/article/view/5082/3266

[21] POTEL, Mike, 1996. MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java. Taligent [online]. -(-), 1-14 [cit. 2019-10-26]. Dostupné z: http://www.wildcrest.com/Potel/Portfolio/mvp.pdf

[22] 2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing [online], 2015. IEEE [cit. 2019-10-26]. ISBN 978-1-5090-0154-5. Dostupné z: http://ieeexplore.ieee.org/document/7363258/

[23] Android Architecture Patterns Part 3: Model-View-ViewModel, 2016. In: Upday [online]. -: - [cit. 2019-10-29]. Dostupné z: https://upday.github.io/blog/model-view-viewmodel/

[24] The Repository Pattern, 2010. Microsoft [online]. USA: Microsoft, 04/27/2010 [cit. 2019-11-06]. Dostupné z: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649690(v=pandp.10)?redirectedfrom=MSDN

[25] BRANDI, Denis, 2019. The "Real" Repository Pattern in Android. ProAndroidDev [online]. -(-), 1 [cit. 2019-11-06]. Dostupné z: https://proandroiddev.com/the-real-repository-pattern-in-android-efba8662b754

[26] MAKARENKO, V., O. OLSHEVSKA a Yu. KORNIENKO, 2017. An architectural approach for quality improving of Android applications development

which implemented to communcation application for mechatronics robot laboratory ONAFT. ONAFT [online]. -(-), 1-6 [cit. 2019-11-09]. Dostupné z: https://journals.onaft.edu.ua/index.php/atbp/article/download/714/759

[27] MAINKAR, Praiyot, 2017. Expert Android Programming. 1. Birmingham: Packt Publishing. ISBN ISBN 978-1-78646-895-6.

[28] The Clean Architecture, 2011. The Clean Code Blog [online]. USA: Martin, 13.8.2012 [cit. 2019-11-09]. Dostupné z: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[29] IVANICS, Péter, 2017. An Introduction to Clean Software Architecture. SASMOOTA [online]. 1-4 [cit. 2019-11-09]. Dostupné z: http://pivanics.users.cs.helsinki.fi/portfolio/docs/publications/Peter_Ivanics-Clean_Software_Architecture.pdf

[30] Clean Architecture Tutorial for Android: Getting Started, 2019. Raywenderlich.com [online]. -: R. Wenderlich, 5.8.2019 [cit. 2019-11-09]. Dostupné z: https://www.raywenderlich.com/3595916-clean-architecture-tutorial-for-android-getting-started

[31] JEMEROV, Dmitry a Svetlana ISAKOVA. Kotlin in action. 2017. Shelter Island, NY: Manning Publications Co., [2017]. ISBN 978-161-7293-290.

[32] Kotlin 1.0 Released: Pragmatic Language for JVM and Android, 2016. In: Kotlin Blog [online]. Russia: JetBrains [cit. 2019-11-20]. Dostupné z: https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android/

[33] Kotlin Foundation. KotlinLang [online]. Petrohrad: JetBrains, 2018 [cit. 2019-10-16]. Dostupné z: https://kotlinlang.org/foundation/kotlin-foundation.html#lead-designer

[34] BHAVYA, Karia, 2018. A quick intro to Dependency Injection: what it is, and when to use it. Medium [online]. -(-), 1 [cit. 2020-02-04]. Dostupné z: https://medium.com/free-code-camp/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f

[35] SEEMANN, Mark, 2012. Dependency Injection in .NET. 1. Shelter Island, NY 11964: Manning. ISBN 1935182501.

[36] FOWLER, Martin, 2004. Inversion of Control Containers and the Dependency Injection pattern. In: Martinfowler.com [online]. USA: Fowler [cit. 2020-02-09]. Dostupné z: https://martinfowler.com/articles/injection.html

[37] Dagger [online], USA: Google [cit. 2020-02-10]. Dostupné z: https://dagger.dev/

[38] Reactive Programming For Android, 2017. In: AndroidPub [online]. USA: - [cit. 2020-02-22]. Dostupné z: https://android.jlelse.eu/reactive-programming-for-android-d55bdbb438b4

[39] STALTZ, André, 2014. The introduction to Reactive Programming you've been missing. Staltz Gist [online]. Helsinky: staltz [cit. 2020-02-22]. Dostupné z: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

[40] POUDEL, Subash, 2017. Introduction to Reactive Programming. In: LeapFrog Blog [online]. India: LeapFrog [cit. 2020-02-22]. Dostupné z: https://blog.lftechnology.com/introduction-to-reactive-programming-part-1-5b7c63685586

[41] SALVANESCHI, Guido, Sebastian PROKSCH, Sven AMANN, Sarah NADI a Mira MEZINI, 2017. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. IEEE Transactions on Software Engineering [online]. 43(12), 1125-1143 [cit. 2020-02-22]. DOI: 10.1109/TSE.2017.2655524. ISSN 0098-5589. Dostupné z: http://ieeexplore.ieee.org/document/7827078/

[42] BONÉR, Jonas, Dave FARLEY, Roland KUHN a Martin THOPSON, The Reactive Manifesto. The Reactive Manifesto [online]. [cit. 2020-02-22]. Dostupné z: https://www.reactivemanifesto.org/

[43] ReactiveX [online], -. USA: ReactiveX [cit. 2020-03-01]. Dostupné z: http://reactivex.io/

[44] Understanding Java: RxJava for beginners, 2016. In: https://medium.com/@factoryhr/understanding-java-rxjava-for-beginners-5eacb8de12ca [online]. Chorvatsko: Factory.hr [cit. 2020-03-01]. Dostupné z: https://medium.com

[45] NIELD, Thomas, 2017. Learning RxJava. Birmingham: Packt Publishing. ISBN 978-1-78712-042-6.

[46] Advanced Android in Kotlin 05.2: Introduction to Test Doubles and Dependency Injection, -. CodeLabs [online]. USA: Google [cit. 2020-04-03]. Dostupné z: https://codelabs.developers.google.com/codelabs/advanced-android-kotlin-training-testing-test-doubles/

[47] KOLOFIY, Sergey, 2010. Unit Tests, How to Write Testable Code and Why it Matters. Toptal [online]. USA: Toptal [cit. 2020-04-03]. Dostupné z: https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters

[48] LIPSKI, Michal, 2017. Test Doubles — Fakes, Mocks and Stubs. Medium: Pragmatists [online]. USA: Medium [cit. 2020-04-03]. Dostupné z: https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da

[49] BERNSTEIN, David, 2019. Writing Testable Code. DZone [online]. USA: DZone [cit. 2020-04-03]. Dostupné z: https://dzone.com/articles/write-testable-code

[50] NOVOSELTSEVA, Ekatarina, 2017. Benefits of Test-Driven Development. DZone [online]. USA: DZone [cit. 2020-04-03]. Dostupné z: https://dzone.com/articles/20-benefits-of-test-driven-development

[51] ROY, Aritra, 2018. Mastering the World of Android Testing (Part 1). Medium [online]. USA: Medium [cit. 2020-04-06]. Dostupné z: https://blog.aritraroy.in/understanding-and-mastering-the-world-of-android-testing-part-1-32f6a1a06d3b

[52] ROY, Aritra, 2018. Mastering the World of Android Testing (Part 2). Medium [online]. USA: Medium [cit. 2020-04-06]. Dostupné z: https://blog.aritraroy.in/mastering-the-world-of-android-testing-part-2-23293c34dbbf

[53] Cermat výsledky, In: Cermat [online]. ČR: Cermat [cit. 2019-11-27]. Dostupné z: https://vysledky.cermat.cz/data/Default.aspx

[54] Výsledky maturitní zkoušky v roce 2018 a její vývoj od roku 2011, 2019. In: Cermat [online]. ČR: Cermat, srpen 2019 [cit. 2019-11-27]. Dostupné z: https://data.cermat.cz/files/files/Zaverecne_zpravy/Zaverecna_zprava_CZVV_MZ2018.pdf

[55] DEVLIN, Keith, 2011. Mathematics Education for a New Era: Video Games as a Medium for Learning [online]. 1. Boca Raton: Taylor & Francis Group [cit. 2019-11-27]. ISBN 978-1-4398-6771-6. Dostupné z: https://content.taylorfrancis.com/books/download?dac=C2010-0-49026-7&isbn=9780429107719&format=googlePreviewPdf

[56] MULLER, Derek Alexander, 2008. Designing Effective Multimedia for Physics Education. Sydney. Disertační práce. School of Physics University of Sydney Australia.

[57] What is UI design? What is UX design? UI vs UX: What's the difference, 2019. UX Planet [online]. -(-), 1 [cit. 2019-12-07]. Dostupné z: https://uxplanet.org/what-is-ui-vs-ux-design-and-the-difference-d9113f6612de

[58] Material Design [online], 2015. USA: Google [cit. 2019-12-07]. Dostupné z: https://material.io/

[59] HAMMES, Dayne, Hiram MEDERO a Harrison MITCHEL, 2014. Comparison of NoSQL and SQL Databases in the Cloud. Aisel [online]. -(-), 1-9 [cit. 2019-11-27]. Dostupné z: https://aisel.aisnet.org/cgi/viewcontent.cgi?article=1011&context=sais2014

[60] Firebase Documentation [online], -. USA: Google [cit. 2019-11-27]. Dostupné z: https://firebase.google.com/docs

[61] ObjectBox Documentation [online], 2017. Rakousko: GreenRobot [cit. 2019-11-27]. Dostupné z: https://docs.objectbox.io/

[62] ObjectBox for Mobile, 2017. In: ObjectBox [online]. Rakousko: GreenRobot [cit. 2019-11-28]. Dostupné z: https://objectbox.io/mobile/

[63] Realm, ObjectBox or Room. Which one is for you?, 2016. In: Dev Labs [online]. -: Dev Labs [cit. 2019-11-28]. Dostupné z: https://notes.devlabs.bg/realm-objectbox-or-room-which-one-is-for-you-3a552234fd6e

Seznam symbolů, veličin a zkratek

APK	Android Application Package
API	Application Programming Interface
BaaS	Backend as a Service
DTO	Data Transfer Object
DI	Dependency Injection
IoC	Inversion of Control
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
NoSQL	Databáze, které nejsou typu SQL
SoC	Separation of Concerns
SSoT	Single Source of Truth
SQL	Structured Query Language
OS	Operační systém
POJO	Plain Old Java Object
UI	User Interface
UX	User Experience
UML	Unified Modeling Language
XML	Extensible Markup Language

Seznam příloh

A Obsah přiloženého CD

- / ..... kořenový adresář přiloženého CD
  - Aplikace ..... Aplikace apk
    - mathman.apk
  - DP ..... Text semestrální práce
    - DB_Jan_Malousek.pdf
  - UI ..... Grafický návrh aplikace
    - UI.svg
    - UI.xd
    - UI.png
  - Zdrojove_kody ..... Ukázkové zdrojové kódy aplikace
  - README.txt