UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

DIPLOMOVÁ PRÁCE

2019 Jan Maděra

Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Manažer snů

Jan Maděra

Diplomová práce

2019

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2018/2019**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jan Maděra**
Osobní číslo:	**I17211**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Manažer snů**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

Diplomová práce je zaměřena do oblasti vývoje mobilních aplikací pro platformu Android. V úvodu práce bude provedena rešerše vývojových postupů, jazyků, nástrojů, technik a prostředků určených a uplatňovaných při vývoji mobilních aplikací s důrazem na platformu Android. Dále budou představeny a zhodnoceny integrační rozhraní z architektonického i aplikačního pohledu. V praktické části práce bude vytvořena mobilní aplikace určená pro rychlý záznam snů. Aplikace bude umožňovat na jedno kliknutí přidat záznam v audio formě, který bude následně strojově převeden do textové podoby a přes aplikační programový interface uložen do aplikace třetí strany (např. Seznam úkolů). Pro vývoj aplikace bude vytvořen analytický materiál popisující softwarové dílo za užití UML. Výsledná aplikace bude zveřejněna v GooglePlay.

Rozsah grafických prací:

Rozsah pracovní zprávy: **50-60 stran**

Forma zpracování diplomové práce: **tištěná**

Seznam odborné literatury:

PATNI, Sanjay, Pro RESTful APIs: design, build and integrate with REST, JSON, XML and JAX-RS, Berkeley, Apress, 2017, Books for professionals by professionals, ISBN 9781484226643
LACKO, L'uboslav, Mistrovství - Android, Brno: Computer Press, 2017, Mistrovství, ISBN 978-80-251-4875-4

Vedoucí diplomové práce: **Ing. Lukáš Čegan, Ph.D.**
Katedra informačních technologií

Datum zadání diplomové práce: **22. října 2018**

Termín odevzdání diplomové práce: **18. května 2019**

Ing. Zdeněk Němec, Ph.D.
děkan

L.S.

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 17. listopadu 2018

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 15. 5. 2019

Jan Maděra

PODĚKOVÁNÍ

Tím to bych chtěl poděkoval panu Ing. Lukáši Čeganovi Ph.D., vedoucímu mé diplomové práce, za cenné rady během vypracování této práce. V neposlední řadě bych rád poděkoval své přítelkyni, Ninja aréně a svým blízkým, za podporu, jež mi byli po celou dobu studia.

ANOTACE

Diplomová práce se zabývá návrhem a implementací mobilní aplikace pro operační systém Android, která je zaměřena na zaznamenávání snů. V úvodu teoretické části je provedena rešerše vývoje mobilních aplikací. Dále se práce zabývá softwarovou architekturou a v závěru teoretické části práce jsou představeny knihovny z kolekce Android Architecture Components.

V následující praktické části je provedena analýza a popis datové struktury. Poté jsou popsány použité technologie a jsou představeny aplikační rozhraní aplikací třetích stran, se kterými aplikace komunikuje. V závěru praktické části je uvedena ukázka vyvinuté aplikace.

KLÍČOVÁ SLOVA

Mobilní aplikace, Android, Kotlin, Softwarová architektura, Android Architecture Components

TITLE

Dream Manager

ANNOTATION

The thesis deals with the design and implementation of a mobile application for the Android operating system, which is focused on recording dreams. At the beginning of the theoretical part, there is a research of mobile applications development. Furthermore, the thesis deals with software architecture and libraries from Android Architecture Components are described at the end of the theoretical part.

In the introduction of the practical part is dedicated to analysis and description of data structure. Subsequently, the technologies used are described and application interfaces of third-party applications with which the application communicates are introduced. At the end of the practical part there is an example of developed application.

KEYWORDS

Mobile apps, Android, Kotlin, Software architecture, Android Architecture Components

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM ZDROJOVÝ KÓDŮ

SEZNAM TABULEK

SEZNAM ZKRATEK

API	Application Programming Interface
ART	Android Run Time
JS	JavaSript
DI	Dependency Injection
HAL	Hardware Abstraction Layer
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
MVC	Model View Controller
MVI	Model View Intent
MVP	Model View Presenter
MVVM	Model View ViewModel
OpenGL	Open Graphics Library
ORM	Objektově relační mapování
OS	Operační systém
REST	Representational state transfer
RN	React Native
SQL	Structured Query Language
UI	User inteface
UML	Unifed Modeling Language
UX	User experienc

ÚVOD

V dnešní době jsou mobilní zařízení součástí života většiny lidí. Jejich využití je velmi různorodé, a to hlavně díky mobilním aplikacím, kterých je v dnešní době nespočet. Mobilní aplikace dnes disponují mnoha funkcemi, lze je používat například pro správu e-mailů, prohlížení webových stránek, hraní her nebo k sebevzdělávání. Aplikace se v současnosti dají využít i pro řízení procesů v rámci celé firmy. Například pomocí mobilní aplikace lze spravovat celý sklad či provádět revize vlakových souprav.

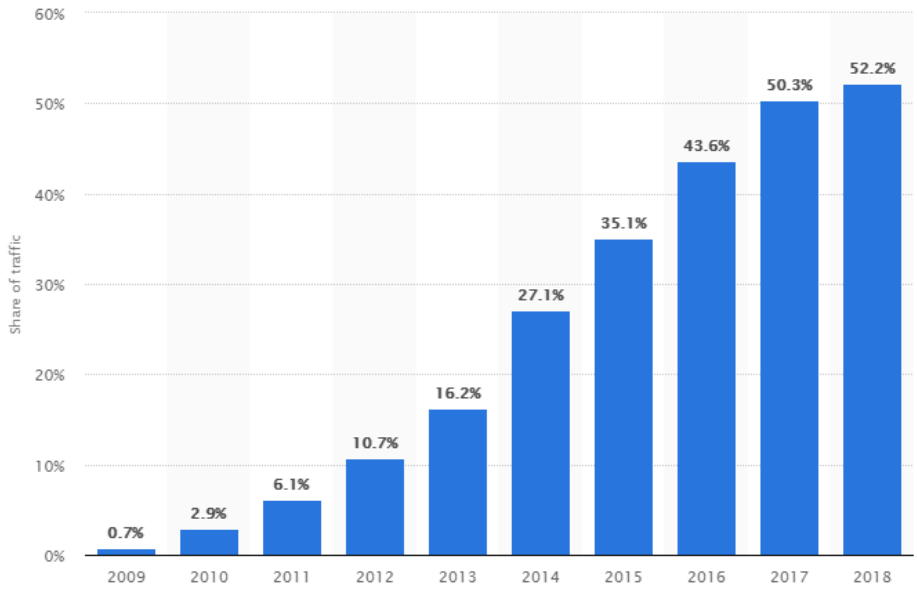
Tato diplomová práce se zabývá tvorbou mobilní aplikace pro operační systém Android, jež slouží k zaznamenávání snů, a to i pomocí zadávání hlasem. Následně je možné tyto sny sdílet s aplikacemi třetích stran, jako jsou Kalendář Google a Úkoly Google. Aplikace by měla sloužit pro analýzu snů, se kterou přišel jako první Sigmund Freud a jehož myšlenku dále rozšiřoval slavný švýcarský lékař a psychoterapeut Carl Gustav Jung, který je označován za zakladatele analytické psychologie.

V první kapitole této práce je provedena rešerše vývoje mobilních aplikací, která je zaměřena na nejpoužívanější operační systémy současnosti a jejich architektury. Dále je v této kapitole popsán multiplatformní vývoj. V následující kapitole jsou rozebrány softwarové architektury se zaměřením na operační systém Android. Jsou zde popsány nejpoužívanější architektury a jejich implementace, přičemž jedna z těchto architektur je použita pro vývoj aplikace, jež je předmětem praktické části. Poslední teoretická kapitola představuje knihovny z kolekce Android Architecture Components, které jsou dnes využívány pro vývoj moderních Android aplikací a byly též využity při vývoji aplikace *„Manažer snů“*.

Praktická část této práce se věnuje analýze, kde jsou stanoveny funkční a nefunkční požadavky a jsou zde také představeny diagramy případů užití a aktivit. Dále se tato kapitola věnuje popsání použitých technologií, které pomohli při vývoji aplikace *„Manažer snů“*. Poté jsou uvedeny aplikace třetích stran, se kterými vyvíjená aplikace komunikuje. V závěru je představena vyvinutá aplikace, která byla publikována v obchodě Google Play.

1 REŠERŠE VÝVOJE MOBILNÍCH APLIKACÍ

V současné době je používání mobilních zařízení pro většinu lidí na denním pořádku. Používají ho ku příkladu ke spravování svého kalendáře či hraní her. Důkazem, že v dnešní době jsou chytré telefony součástí života lidí po celém světě, je graf, který zobrazuje nárůst zobrazovaných internetových stránek na mobilních telefonech od roku 2009 do roku 2018.

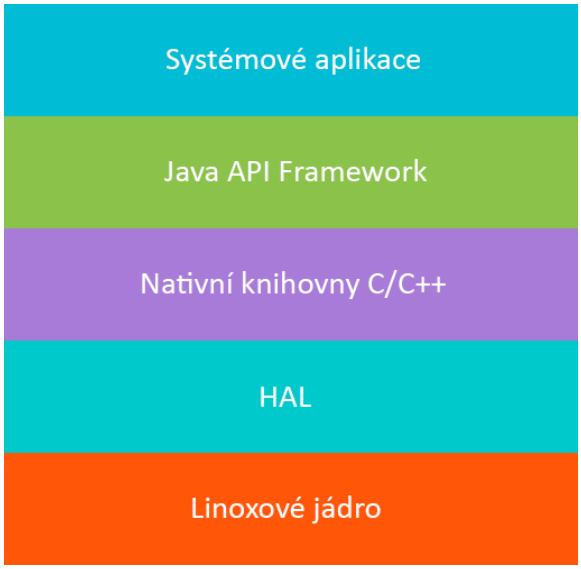


Obrázek 1 Graf zobrazení internetových stránek na mobilních telefonech [1]

Úvod této kapitoly je věnován nejpopulárnějším operačním systémům a jejich architektuře. Dále jsou uvedeny a popsány programovací techniky a jazyky jednotlivých operačních systémů. Následně budou představeny softwarové nástroje, které je možno využít při vývoji. V závěru kapitoly jsou popsány různé procesy, jež je možné využít při vytváření mobilní aplikace.

1.1 Android

Android je open source operační systém, který je založen na Linuxovém jádře. Na následujícím obrázku jsou zobrazeny jednotlivé vrstvy systému.



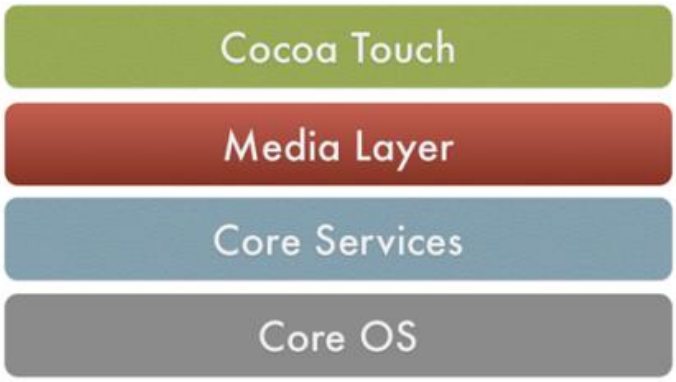
Obrázek 2 Architektura platformy Android [2]

Komponenty ležící v nižších vrstvách jako ART a HAL vyžadují nativní knihovny, jež jsou napsané v programovacích jazycích C a C++. Z obrázku lze vidět, že mezi linuxovým jádrem a nativními knihovnami leží abstraktní hardwarová vrstva HAL, jež poskytuje standardní rozhraní pro vyšší vrstvy. Od Androidu verze 5.0 (API 21) běží každá aplikace ve vlastním odděleném procesu s instancí ART. Android Runtime je napsáno tak, aby bylo schopno spouštět více virtuálních strojů s nízkými paměťovými nároky, a to je umožněno pomocí spouštění DEX souborů. Android poskytuje rozhraní Java, pomocí kterého je možno využívat vlastnosti některých nativních knihoven, například OpenGL, což je knihovna umožňující přidání různé podpory pro vykreslování a manipulování s 2D a 3D grafikou v aplikaci. [2]

Android aplikace lze programovat pomocí jazyků Java a Kotlin. Kotlin je v současné době oficiální programovací jazyk pro Android. Kotlin je jazyk vyvinutý společností JeBratins, jež stojí za populárními programy, jako je IntelliJ, DataGrip a mnoho dalších. Kotlin je objektově orientovaný a je plně kompatibilní s jazykem Java. Kotlin oproti Javě má častokrát kratší syntaxi, což mnohdy vývojářům šetří čas při vývoji, jelikož se nemusí neustále opakovat. Je možné ho používat jak pro vývoj mobilních aplikací, tak na webový vývoj, například populární Framework Spring nyní podporuje Kotlin. [3]

1.2 iOS

Architektura operačního systému iOS je vrstvená podobně jako v případě platformy Android. Jednotlivé vrstvy jsou zobrazeny na následujícím obrázku.



Obrázek 3 Architektura platformy iOS [4]

Z obrázku je vidět, že systém se skládá ze čtyř vrstev, kde spodní vrstvy poskytují základní služby a vrchní vrstvy poskytují už složitější grafická a aplikační rozhraní. Apple poskytuje většinu svých systémových rozhraní ve speciálních balíčcích, jež se nazývají frameworky. Framework si lze představit jako jakýsi adresář, jenž obsahuje sdílenou dynamickou knihovnu, zdroje jako jsou hlavičkové soubory, obrázky a pomocné aplikace, které daná knihovna potřebuje.

Každá ze čtyř vrstev má svou sadu frameworků, jež jsou používány vývojáři pro vytváření aplikací. První vrstva CoreOS obsahuje nízkoúrovňové vlastnosti a technologie, jež mnoho aplikací používá, jako např. bluetooth, akcelerátor, bezpečnostní služby, autentifikaci a mnoho dalších. Druhá vrstva Code Service Layer poskytuje rozhraní k frameworkům, jež zprostředkovávají například lokaci, pohybové senzory, kontakty. Nově lze v této vrstvě nalézt rozhraní, které poskytuje informace týkající se zdraví uživatele, nebo framework, pomocí kterého uživatel může ovládat své domácí Apple zařízení (např. Apple Home). Ve vrstvě Media Layer můžeme nalézt technologie, jako jsou zvuk, video či grafika. V nejvyšší vrstvě architektonického modelu se nachází vrstva Cocoa Touch, která zodpovídá například za zobrazování a upravování kalendáře. Implementuje podporu pro Game Center, pomocí něhož může uživatel sdílet informace týkající se her online. Dále se v ní vyskytuje MapKit, jež zprostředkovává mapu. Tato vrstva má také podporu pro multitasking či dotykové senzory, spravuje uživatelské rozhraní a další služby vyšší úrovně. [4]

Dříve se aplikace pro OS X a iOS vyvíjely v programovacím jazyce Objective-C, který vychází z jazyka C, ale na rozdíl od svého předchůdce je objektově orientovaný a přidává syntaxi pro definování tříd a metod. Navíc doplňuje jazyk například i o správu grafů. V dnešní době se pro vývoj aplikací pro OS X a iOS využívá programovací jazyk Swift. Jedná se o moderní jazyk, který si dává za úkol nahradit jazyky C, C++ a Objective-C. Zakládá si na třech hlavních pilířích, a to bezpečnost, rychlost a čitelnost. [5], [6]

1.3 Multiplatformní vývoj

V dnešním světě mobilních aplikací existuje mnoho zařízení používajících různé operační systémy. Na dnešním mobilním trhu dominují převážně dvě platformy, a to Android a iOS, které byly popsány v předešlých kapitolách. Aby vývojáři pokryli co největší část trhu, musí se snažit vyvíjet aplikace pro více než jednu platformu, což není jednoduchý úkol, jelikož musí vytvořit aplikaci pro každou platformu zvlášť, což je samozřejmě náročné jak časově, tak i finančně. Proto na trh nastoupil takzvaný multiplatformní vývoj, který má tyto problémy řešit, jelikož tak vzniká pouze jedna kódová báze.

Vývoj mobilních aplikací lze rozdělit do následujících kategorií: webové aplikace, hybridní multiplatformní aplikace, nativní aplikace, nativní multiplatformní aplikace. Multiplatformní vývoj je v podstatě proces, při kterém vzniká aplikace, která používá jednu kódovou základu, místo toho aby se pro každou platformu programovala aplikace zvlášť. Tento přístup pro vývoj má své klady a zápory, kde mezi hlavní klady patří cena, za kterou se aplikace vyvíjí, udržitelnost kódu, rozsah na trhu. Naproti tomu stojí jisté nevýhody, mezi které patří výkon, řešení různých systémových výjimek pro různé platformy a možné zpoždění při vydávání updatu. I přesto že multiplatformní vývoj šetří čas učení se různých technologií, musí vývojář mít povědomí a znalosti o technologiích, které používá právě multiplatformní vývoj. Například pokud se používá technologie React Native, musí vývojář znát technologie jako JavaSript s JSX, React.js. [7]

1.4 Technologie multiplatformního vývoje

V následujícím textu budou krátce popsány některé vybrané technologie, jež se dnes používají pro multiformní vývoj.

1.4.1 React Native

React Native je pseudo-nativní framework, který je vyvinutý a udržovaný společností Facebook. Je založen na technologii ReactJS a využívá JSX, což hybrid mezi JavaScriptem a HTML. Tento framework se stal velmi populárním zejména díky tomu, že v dnešní době řada vývojářů dokáže programovat v technologii JavaScript a obdobný počet vývojářů dokáže používat ReactJS, jež se využívá zejména na webový Front-end. RN také dodržuje design specifický pro jednotlivé platformy, takže prostředí, ve kterém se uživatel pohybuje, mu připadá přirozené, avšak toto se týká zejména výchozích komponent. To přináší různé komplikace v podobě přidávání různých oklik pro každou platformu zvlášť. Reacti Native je momentálně ve vedení, co se týče komunity a zkušeností s vydáváním aplikací na produkční trh, což nabádá některé společnosti jít touto cestou. [8]

1.4.2 Flutter

Flutter je multiplatformní vývojový framework, který je vytvořený a udržovaný společností Google, což je stejná firma, jež stojí za vývojem operačního systému Android. Technologie Flutter je poměrně nová, ale i přesto má již dnes mnoho příznivců. Používá programovací jazyk Dart, který je také vytvořen a spravován společností Google. Jedná se o jazyk, jenž se využívá pro vývoj mobilních, serverových, desktopových nebo webových aplikací. Bohužel oproti technologii React Native mu schází produkční a testovací čas, ale přes tento nedostatek mnoho společností začíná své aplikace vytvářet pomocí technologie Flutter. [8]

1.4.3 Xamarin

Xamarin je multiplatformní vývojový framework pro mobilní aplikace vlastněný společností Microsoft, jež používá technologii C#. Pomocí Xamarinu lze vyvinout aplikaci pro Android, iOS, macOS, Tizen, GTK# a Windows. Jednou z výhod je rychlost aplikací, jelikož se kód kompiluje na nativní kód, dosahují aplikace vyvinuté v Xamarinu podobné či stejné rychlosti. Další z výhod je vytváření uživatelského rozhraní, jelikož vytvořené komponenty se kompilují do komponent, jež jsou specifické pro danou platformu. I přes tyto nesporné výhody aplikace zabírají často více místa v paměti mobilního zařízení než aplikace vyvinuté v nativním kódu. Jednou z dalších nevýhod je neudržení kroku s aktualizacemi různých platforem. [7], [9]

1.5 Životní cyklus vývoje mobilních aplikací

Tato podkapitola je věnována základním postupům a principům, které jsou aplikované při vývoji softwaru se zaměřením na aplikace pro platformu Android.

První proces, jímž každý vývoj začíná, je danou aplikaci vymyslet. Při tomto procesu je důležité se především zaměřit na tyto dvě otázky: „Může aplikace vyřešit problém *X*?“ a „Jsou na trhu aplikace, které problém *X* řeší?“. Pokud lze na otázky odpovědět konstruktivně, dalším logickým krokem je, jak bude aplikace daný problém řešit, a pokud už některé aplikace na trhu existují, zda nově vyvinutá aplikace bude daný problém řešit lépe. V případě platformy Android je velmi častý scénář, že aplikace již na trhu existuje a je tedy důležité zvolit strategii, kterou se vydat. [11]

Další na řadě je UX celé aplikace. User experience je proces navrhování designu aplikace, který je jednoduchý k použití a zcela intuitivní. Dbá na to, aby uživatel získal určitý zážitek při používání produktu. [10]

Po navržení UX aplikace následuje samotný design. Design lze rozvrhnout do dvou částí, kde první část je vytvoření takzvaných drátových modelů, které nespecifikují barvu či konkrétní komponenty, ale spíš rozvržení daných komponent po obrazovce. Druhá část je samotný design, který se týká barev, obrázků, navržení určitých komponent, které budou specifické pro navrhovanou aplikaci. V případě Android mobilních aplikací vládne na trhu vizuální jazyk, který se nazývá Material Design, který udává určitý styl, jak by aplikace měly vypadat, aby orientace z pohledu uživatele byla co nejjednodušší a co nejvíce intuitivní.

Ve stejném čase, kdy je navrhován design aplikace, je možné už aplikaci vyvíjet. Zde už není moc rozdílů od klasického vývoje. Programátoři musí hledět na architekturu aplikace. V současnosti vývojáři volí například architekturu MVP či MVVM, obě tyto architektury budou popsány a porovnány v následujících kapitolách. Měli by také dodržovat standardní programátorské zásady, jako například znovu použitelnost a čitelnost kódu. Po dokončení vývoje následuje samotné testování, marketing a nasazení aplikace. V případě platformy iOS se aplikace publikují do obchodu App Store a Android aplikace do Obchodu play. Pro Android existují i různé alternativní obchody jako Galaxy Store od společnosti Samsung, ale nejsou tolik využívané. Po dokončení všech těchto kroků se aplikace udržuje, opravuje a vylepšuje, aby neskončila za svojí konkurencí. [11], [12]

2 SOFTWATOVÁ ARCHITEKTURA

Následující text se zabývá softwarovými architekturami, které se v dnešní době implementují do aplikací. V úvodu kapitoly bude představena definice z důvodu náhledu do problematiky. Dále budou popsány už konkrétní implementace se zaměřením na operační systém Android. Budou zde uvedeny i ukázky kódů z konkrétních implementací. V závěru bude uvedeno shrnutí všech architektur.

2.1 Uvedení do problematiky

V současnosti prozatím nebyla zavedena žádná oficiální definice termínu softwarová architektura, avšak vzniklo mnoho definic, které mají společné jádro a liší se například pouze sférou, na které je dané řešení aplikováno. Pro tento text, jenž se zabývá problematikou softwarové architektury, byla zvolena následující definice. *„Softwarová architektura je struktura komponent programu/systému, jejich vzájemné vazby, principy a předpisy určující jejich návrh a vývoj v průběhu času.“*

Činnosti, které jsou prováděny při návrhu, lze rozdělit do několika základních kategorií. Jedna z těchto skupin je provádění dekompozice systému do různých dekompozičních jednotek, tyto činnosti lze pojmenovat jako vlastní návrh. Za další kategorii činností lze označit reprezentaci a realizování těchto dekompozičních jednotek systému. Činnosti, jež popisují realizaci komunikace. Jsou to tedy činnosti, které se obecně týkají nároků na architekturu vyvíjeného systému, lze je označit termínem nároky na architekturu. Vlastní návrh by neměl předcházet určeným nárokům na architekturu. Stanovená architektura ovlivňuje funkčnost, vývoj, údržbu a následné požadavky na rozšíření. Z toho pohledu je jasné, že vytváření jisté architektury pomocí vlastního návrhu by nemělo být jen vedlejším efektem, ale vědomou činností. Nároky na architekturu určitým způsobem stanovují konečnou strukturu systému.

Stanovení nároků na softwarovou architekturu, kterými má systém disponovat, před implementací vlastního návrhu s sebou nese mnoho kladů. V následující části budou představeny některé z kladů, které může dobře navrhnutá architektura přinést, a mělo by z ní vyplynout, proč je softwarová architektura důležitá. [14]

Vývoj založený na komponentách (Component-based development)

Systém lze sestavit velmi rychlým a účinným způsobem pomocí implementací externě vyvinutých komponent. Tato vlastnost umožňuje vývoj, který je samostatný, nezávislý a je prováděn paralelně, právě díky nezávislosti. Stěžejním bodem zde může být skládání a integrace komponent do sebe. [14]

Včasná předpověď jakosti (Early quality prediction)

Určité vlastnosti lze stanovit při studiu architektury i bez kompletní znalosti návrhu a kódu. Jedná se o vlastnosti, kterými systém disponuje při běhu, či o vlastnosti statické. Příkladem může být výkon, který lze často určit pomocí zjištění frekvence změn a analýzou komunikace mezi komponentami. Udržitelnost lze z části vyčíst z lokalizací změn. V důsledku toho je třeba přizpůsobit nároky na vlastnosti, které určují kvalitu, tvorbě nebo výběru softwarové architektury. Je ale důležité si uvědomit, že špatný výběr architektury může vytvořit určitou překážku pro dosažení stanovených kvalitativních atributů vyvíjeného systému. [14]

Vývoj produkčních řad (Product line development)

Vyvinuté produkční řady mohou sdílet společnou architekturu, což může mít za následek znovu použití ve velkém měřítku. Za produkční řadu lze označit určitou skupinu systémů, které společně zahrnují zaměřovanou oblast. [14]

Oddělení funkcionality a propojení (Separation of functionality from interconnection)

Je důležité při vývoji myslet na oddělení funkcionalit a mechanismů vytvořených komponent. Tato vlastnost s sebou nese mnoho kladů. Například pokud bude kód psán více abstraktně, nebudou komponenty, které pracují na určité úrovni, vědět o komponentách, které se vyskytují na úrovni jiné, může se docílit určité oddělitelnosti, která umožňuje již zmíněnou znovu použitelnost. Jako další klad lze uvést případ, kdy se začne pracovat na vlastním návrhu, který se po uběhnutí určitého času ukáže jako chybný. Úprava se potom bude konat pouze v rámci chybné komponenty a nikoli v komponentách ostatních. [14]

Omezení prostoru návrhu (Constraining the design space)

Omezení prostoru návrhu neboli méně je více, znamená, že je možné při návrhu stanovit jistá omezení struktury programového návrhu a řešení určitých scénářů, které se mnohdy opakují. Tato omezení přinášejí výhody zejména v čitelnosti, znovu použitelnosti, testovatelnosti, udržovatelnosti. Tuto oblast pokrývají takzvané návrhové vzory (design patterns). [14]

2.1.1 Vývoj založený na softwarové architektuře

Při aplikování nějaké softwarové architektury v rámci vývoje a údržby systému je důležité věnovat pozornost problémům, které mohou vzniknout. Jeden z problémů může být vývoj či výběr architektury. Architekturu je většinou potřeba vyvinout a ukazuje se, že vývoj dané architektury je spíše iterativního rázu, jelikož tento vývoj obsahuje prototypování, testování a analýzu. Jako další problém lze uvést reprezentaci a sdělování architektury třetím stranám. Dále se lze ve vývoji setkat s analýzou a hodnocením architektury, jelikož je nutné ověřovat, zda je zvolená architektura správná a zda vývoji přinese více kladů než záporů. Jako poslední lze zmínit, že je potřeba zajišťovat aplikaci zvolené architektury, pokud při vývoji přestane být předepsaná architektura dodržována, veškerá snaha při jejím návrhu přichází vniveč. [14]

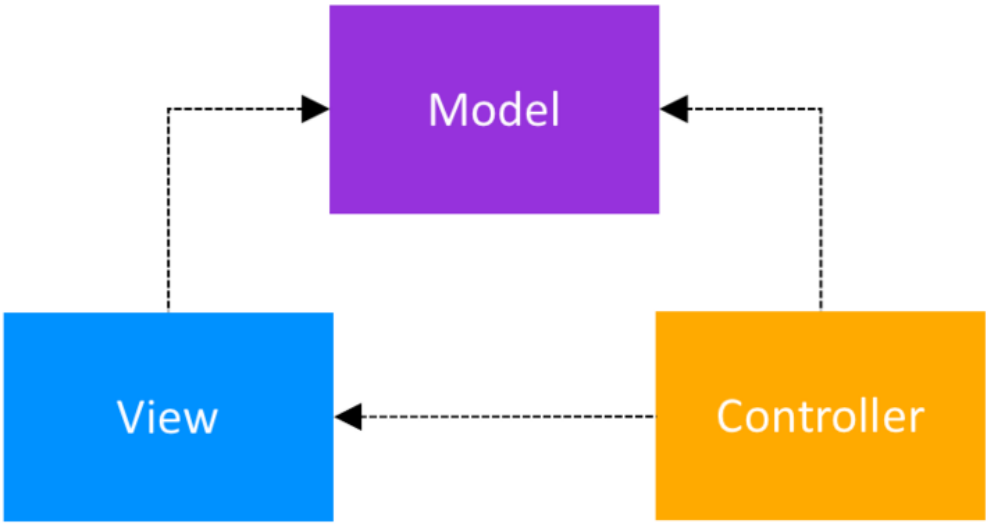
2.2 MVC

Softwarová architektura MVC byla jedna z prvních, které se začaly využívat pro vývoj na platformu Android. Byla jedním z prvních přístupů, jak by mohl být software definován a popsán na základě odpovědností jednotlivých komponent. Prvně byla představena v 70. letech 20. století norským autorem Trygve Reenskaugem. [15]

V aplikacích dnešní doby se stává mnohem častěji, že se mění logika uživatelského rozhraní rychleji než byznys logika, a tím pádem je zapotřebí mechanismu, který by tyto stavy reflektoval. Architektura Model View Controller se skládá ze tří hlavních vrstev.

- *Model* – je datová vrstva, která je zodpovědná především za správu byznys logiky aplikace a také síťových požadavků.
- *View* – jedná se o vrstvu, která představuje uživatelské rozhraní aplikace, tudíž zobrazuje modelovou vrstvu uživateli.
- *Controller* – je ve své podstatě logická vrstva, která je upozorněna ve chvíli, kdy uživatel provede nějakou akci přes UI aplikace.

Na následujícím obrázku je vidět diagram všech tří hlavních složek architektury a jejich závislosti.



Obrázek 4 Architektura MVC [16]

Z obrázku lze vyčíst, že vrstvy Controller a View přímo závisí na modelové vrstvě. Pomocí vrstvy Controller je pak možné měnit data, která se zobrazují uživateli a mění stav modelu. Mnoho aplikací však vyžadovalo oddělenou modelovou vrstvu, kterou lze samostatně testovat, proto se začaly objevovat různé implementace této architektury. Mezi nejznámější a nejpoužívanější varianty se řadí ty, jež mají vrstvu Model pasivní anebo aktivně upozorňovanou na změněný stav. [16]

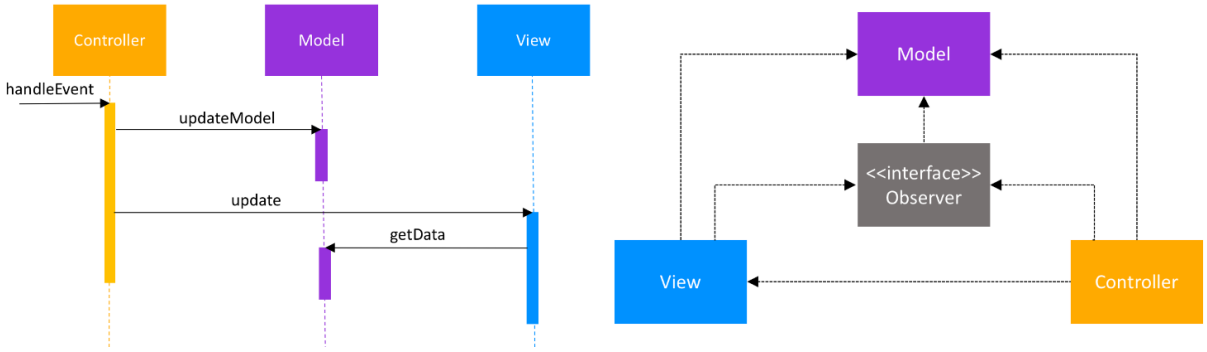
Pasivní

Ve verzi, kde vrstva Model hraje pasivní roli, je vrstva Controller jedinou třídou, jež manipuluje s vrstvou modelovou. Na základě akcí od uživatele, které jsou většinou vyvolány z vrstvy Controller, má Controller za úkol upozornit modelovou vrstvu na změnu, jež byla důsledkem akce. Po dokončení změny v modelové vrstvě Controller upozorní View vrstvu, která aktualizuje svůj stav, a tím pádem si vyžádá od modelové vrstvy její aktuální stav. Tím se zaručí integrita dat, jelikož stav View pouze odráží stav modelové vrstvy uživateli. [16]

Aktivní

V případech, kdy Controller není jedinou třídou, která je schopna spravovat modelovou vrstvu, musí vrstva Model upozornit View a ostatní třídy na změny, a to za pomocí návrhového vzoru Observer. Model tedy v tomto případě obsahuje kolekci typu Observer. Každá položka v kolekci je ve své podstatě View, které chce být o změnách informováno. Příkladem může být, že View je zde pozorovatelem a naslouchá změnám modelové vrstvy, proto pokaždé změní-li

se Model, je View upozorněno, aby aktualizovalo svůj stav, a požádá Model o aktuální data. Tyto příklady jsou zobrazeny na následujícím obrázku, kde na levé straně se nachází diagram pasivního modelu a na pravé straně modelu aktivního. [16]



Obrázek 5 MVC - aktivní a pasivní model [16]

V rámci platformy Android bylo dříve zvykem považovat aktivitu či fragment za vrstvu View a Controller, tím bylo sice zaručeno, že modelová vrstva je oddělena a je ji možno testovat, ale mít v rámci jedné třídy dvě vrstvy nedodržuje správně stanovenou architekturu MVC. Později se ukázalo, že aktivity či fragmenty by měly zastupovat roli View a Controller by měl být oddělenou třídou, jež nevyužívá žádné Android třídy.

Mezi hlavní výhody využívání MVC na platformě Android patří především oddělení zodpovědnosti. Tato vlastnost podporuje nejen testovatelnost jednotlivých tříd, především modelové vrstvy, ale podporuje ve velké míře také rozšiřitelnost aplikace. Jak už bylo zmíněno, modelová vrstva by neměla záviset na žádné Android třídě, tím pádem je na ní možno aplikovat unit testy. Pokud je MVC dodrženo ve správné míře, nepodléhá ani vrstva Controller žádné závislosti na Android třídě, využívá pouze rozhraní z vrstvy View, takže je možné ho také odděleně testovat. Na druhou stranu softwarová architektura Model View Controller s sebou přináší i řadu nevýhod. Přestože modelová vrstva a vrstva Controller nezávisí na žádné Android třídě, vrstva View vždy přímo závisí na vrstvě Controller a Model. Aby bylo možné tuto závislost minimalizovat, měl by Model poskytovat testovatelné metody pro každé View, které je uživateli zobrazeno. V případě aktivního modelu toto řešení výrazně zvyšuje počet metod a tříd, jelikož Observer bude požadován pro každý typ dat, na kterém je nutno naslouchat. Jeden z dalších problémů
se objevuje při rozhodování, která vrstva má na starost logiku UI, jelikož jak již bylo zmíněno, Controller spravuje Model a View je informováno o změně modelu a zobrazí data. Ale není přesně určeno, jak zobrazit data do View. To byl jeden z důvodů, proč vývojáři hledali alternativu k MVC. [16]

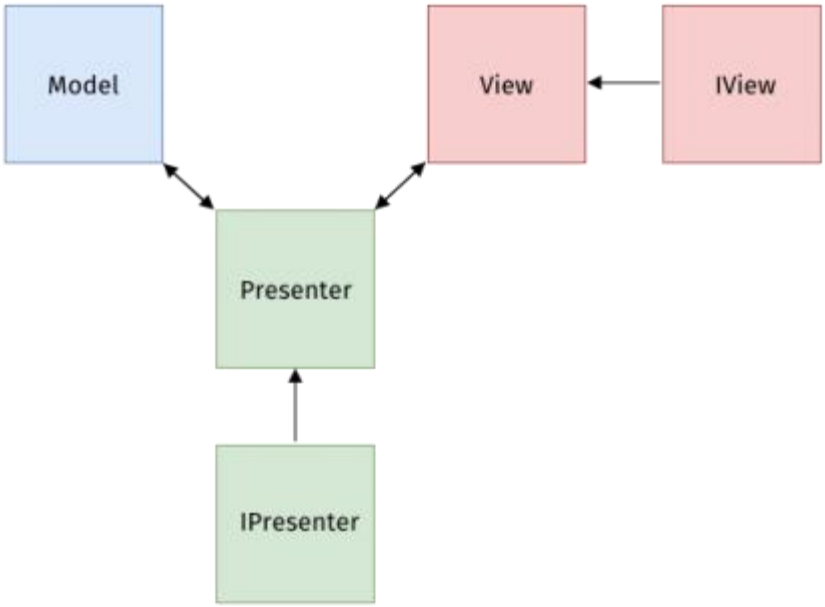
2.3 MVP

Jak už bylo zmíněno na konci předešlé kapitoly o MVC, architektura MVC má značné nevýhody, jelikož mnoho logiky skončí ve vrstvě Controller a bývá častým případem, že Controller je aktivita či fragment, a tak vzniká velmi rozsáhlá třída, která má velkou zodpovědnost za to, co bude uživateli zobrazeno, což bývá čím dál větším problémem s přibývající funkcionalitou. Na platformě Android je ovšem také problémem to, že aktivita či fragment jsou úzce svázány s UI a modelovou vrstvou, to má často za následek porušení principu oddělení zodpovědnosti. Důsledkem toho je například špatné testování jednotlivých vrstev. Úzce spojené vrstvy se špatně testují zejména v případě, pokud nějaká komponenta vyžaduje Android SDK, potřebuje testování fyzického zařízení nebo emulátor s operačním systémem Android. V důsledku toho pouhé unit testy nepostačí a je nutné přistoupit k jiným možnostem testování.

Jedna z alternativ k problémům softwarové architektury Model View Controller je architektura MVP. Model View Presenter poskytuje modularitu, testovatelnost a je obecně čistší. Kódová báze je lépe udržitelná v čase a skládá se z následujících vrstev:

- Model – stejně jako u MVC je datová vrstva, která je zodpovědná především za správu byznys logiky aplikace a také síťových požadavků.
- View – podobně jako u MVC View bude implementováno v aktivitě či fragmentu, avšak MVP mění rozsah, který View spravuje.
- Presenter – má na starost UI aktualizace na základě změn v modelové vrstvě, a také zpracovává vstupy od uživatele. Obsahuje rovněž většinu byznys logiky a nahrazuje Controller z MVC. [17]

V MVP jsou Controller a View rozdělné do dvou částí, Presenter a View, místo aby třída aktivity (fragmentu) spravovala změny z modelové třídy a změny zobrazující se na UI. Model a UI spolu komunikují pouze prostřednictvím třídy Presenter. Presenter v sobě tedy obsahuje byznys logiku a View má jen jediný účel, a to zobrazovat data uživateli. Logika, jež byla v případě MVC uvnitř vrstvy Controller, se nyní rozdělí mezi vrstvy Presenter a View, kde Presenter spravuje tok dat, který přichází z modelové vrstvy, a View data zobrazuje. Veškerý kód, který se týká SDK Androidu, zůstává ve vrstvě View a Presenter tedy může být lehce testovatelný unit testy, jelikož neobsahuje žádné komponenty, jež náleží Androidu. Tok dat mezi jednotlivými vrstvami je vidět na následujícím digramu. [17]



Obrázek 6 Architektura MVP [18]

Z diagramu je vidět, že třída Presenter či View vždy implementuje rozhraní (označení *I* na začátku názvu třídy), přes které komunikuje dále. Tato rozhraní také pomáhají k testovatelnosti těchto vrstev. Rovněž je vidět, že model je kompletně oddělný od vrstvy View, která s ním komunikuje pouze prostřednictvím Presenteru. Z diagramu také lze vyčíst, že pokud nastane nějaká změna dat, Presenter je na toto upozorněn a předává informaci dále do View. Tento scénář funguje i opačným směrem, pokud uživatel v UI provede nějakou akci, View na toto zareaguje a změnu předává do vrstvy Presenter, která předává tuto akci modelu, který na tuto změnu zareaguje. Pro jednodušší přehled této architektury budou uvedeny a vysvětleny krátké ukázky kódu. Na následující ukázce kódu je vidět rozhraní, jež může sloužit pro jednu obrazovku aplikace. [18]

```
   interface DreamsContract {

    interface View {
        fun showButtonClick(b: Boolean)
        fun setButtonColor(color: Int)
        fun navigateNextScreen()
        fun showError(error: String)
        fun showTickVisibility(value: Int)
    }

    interface Presenter {
        fun loadNextScreen()
        fun verifyEntries()
      }
}
```

Zdrojový kód 1 MVP rozhraní obrazovky [18]

Z kódu je vidět rozhraní *DreamsContract*, které uvnitř sebe má další dvě rozhraní, View a Presenter. Takto sestavené rozhraní má smysl v tom, že jeho jméno popisuje obrazovku, ke které je rozhraní vázáno. Vnitřní rozhraní View a Presenter, jak již názvy avizují, budou implementovány na jednotlivé komponenty. Z názvů metod rozhraní *DreamsContract.Presenter* je vidět, že zodpovídá za akce jako načtení obrazovky, validace a podobně. Na druhou stranu rozhraní *DreamsContract.View* zodpovídá za akce jako nastavení barvy tlačítka či zobrazení chyby při chybné validaci. Pro lepší představu, jak může vypadat již implementovaná třída Presenter, slouží následující ukázka kódu. [18]

```
   class DreamsPresenter(val view: DreamsContract.View): DreamsContract.Presenter
{
    private val model: DreamModel = DreamModel()

    override fun loadNextScreen() {
        view.navigateNextScreen()
    }

    override fun defaultSettings() {
        view.setButtonColor(R.drawable.btn_ash)
        view.showButtonClick(false)
        view.showTickVisibility(View.INVISIBLE)

    }
}
```

Zdrojový kód 2 MVP Presenter [18]

Při inicializaci třídy *DreamsPresenter* se aktivuje model, což není úplně nejlepší způsob, jak model používat, a proto bývá zvykem na tomto místě použít Dependency Injection, která slouží pro „podávání“ jednotlivých instancí. V signatuře třídy je vidět, že jako parametr zde slouží rozhraní pro komponentu View. Z kódu lze vyčíst, že je zde splněna podmínka oddělených závislostí, jelikož model nemá žádnou závislost na komponentě Presenter, a je tudíž samostatně testovatelný. Presenter v tomto případě nemá žádnou závislost na třídách, které implementují Android SDK, a tak je možné ho také testovat pomocí unit testů, což v případě MVC nebylo proveditelné. Jediné, co teď tedy chybí, je ukázka kódu, která bude představovat implementaci View. [18]

```
class DreamsActivity : AppCompatActivity(), DreamsContract.View {
    .
    .
    .
    override fun onCreate(savedInstanceState: Bundle?) {
        presenter = AppPinPresenter(this)
        presenter!!.defaultSettings()

        mButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                presenter.defaultSettings()
            }
        });
    }

    override fun showError(error: String) {
        Toast.makeText(this, "Nastala chyba" ,Toast.LENGTH_SHORT).show()
    }
}
```

Zdrojový kód 3 MVP View [18]

View implementuje již zmíněné rozhraní *DreamsContract.View* a pomocí něho různě interaguje s uživatelem. Presenter je zde inicializován hned na začátku v metodě *onCreate(Bundle)*, což není nejlepší způsob, který může vést k zbytečné znovu inicializaci, proto Presenter bývá vkládám pomocí Dependecy Injection. Jako příklad funckionality architektury může sloužit situace, když uživatel klikne na tlačítko *mButton,* které vyvolá metodu v komponentě Presenter *defaultSettings()*, který na tento popud aktualizuje View, které je mu předáno v konstruktoru. Tato implementace MVP je velmi obecná a existuje mnoho alternativ, jak s MVP pracovat. [18]

Jak již bylo avizováno, Presenter se vyskytuje uvnitř aktivity či fragmentu, které podléhají svému životnímu cyklu, což má za následek, že Presenter se může pokusit aktualizovat View, které již neexistuje. To je jedna z nevýhod v rámci MVP, jelikož musí být implementovány i metody, které reagují na tyto životní cykly. I přes tento nedostatek tato architektura s sebou nese řadu výhod, jako je například možnost znovu použití kódu komponenty Presenter, jelikož bývá zvykem definovat menší Presenter, který v sobě nese základní funkcionalitu a tu je poté možno využít na více místech. [18]

2.4 MVVM

V předešlé kapitole byla představena architektura MVP, která přes své výhody měla jednu značnou nevýhodu, a to že Presenter z části podléhal životnímu cyklu (aktivity či fragmentu), a právě tento problém řeší softwarová architektura MVVM. Model View ViewModel obecně řeší problémy testovatelnosti, jelikož odděluje uživatelské rozhraní od byznys logiky či síťového back-endu a snaží se udržovat kód UI co možná nejjednodušší a kompletně ho oddělit

od veškeré logiky. Tato softwarová architektura přišla na svět v rámci platformy Android s představením řady knihoven Android Architecture Components (těmto knihovnám je věnována kapitola Android Architecture Component) vyvíjenými společností Google. MVVM se v rámci operačního systému Android skládá ze tří hlavních částí:

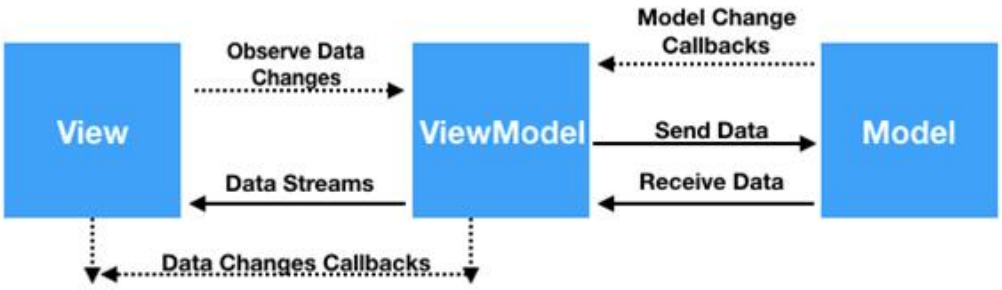
- Model – podobně jako u MVP či MVC se jedná o vrstvu, která je zodpovědná především za správu byznys logiky aplikace a také síťových požadavků.
- View – je uživatelské rozhraní aplikace, v případě Androidu se jedná o aktivitu či fragment.
- ViewModel – je komponenta, jež propojuje zbylé části aplikace. Stará se o změnu stavu View, tak i Modelu. [19]

Další důležitou součástí MVVM je reaktivní programování.

Reaktivní programování

Reaktivní programování je vcelku nové paradigma, které využívá návrhový vzor Observable pro zachytávání událostí. Pro reaktivní programování se používají například knihovny LiveData, RxJava, RxKotlin, RxSwift a další. Jeden z hlavních úkolů reaktivního programování je zjednodušit kód pro zpracování asynchronních volání. Reaktivní programování by se dalo klasifikovat jako programování založené na událostech. Všechny toky dat, které probíhají, mohou mít na sobě zaregistrovaného pozorovatele, který čeká například na jejich načtení z modelové vrstvy. Jedna z dalších výhod, které nabízí reaktivní programování, je, že každá událost probíhá ve svém vlastním vlákně, tím pádem vykonávané úkoly (pokud na sobě nejsou závislé) mohou probíhat paralelně. Jednoduchým příkladem reaktivního programování může být rovnice $x = y + z$, kde suma y a z je přiřazena do proměnné x. Ve chvíli, kdy se změní například y hodnota, bude x automaticky aktualizováno, a to může být dosaženo pozorováním změn na proměnných y a z. [20]

Na následujícím diagramu je zobrazena architektura MVVM.



Obrázek 7 Architektura MVVM [21]

Z obrázku je vidět, že hlavní roli zde hraje ViewModel, jenž slouží jako prostředník mezi vrstvou View a Model. ViewModel má tedy na starost zachytávat události z obou stran, například provede-li uživatel nějakou akci v uživatelském rozhraní (kliknutí na tlačítko) nebo pokud například přijde nějaká událost ze síťového rozhraní. To vše má na starost ViewModel, je však velmi těžké a komplikované zachytávat tyto události ze všech stran a reagovat na ně. Právě pro tyto scénáře se využívá v komponentě ViewModel reaktivní programování. Z obrázku je také patrné, že View implementuje pozorovatele na datech, který je zobrazuje. V případě, že nastane nějaká změna, ViewModel upozorňuje View a poskytuje mu aktualizovaná data. ViewModel má také uvnitř sebe referenci na Model, kterému posílá data a od něhož přijímá aktualizace. V následujících ukázkách kódu bude představena základní implementace této architektury.

```
    class DreamListViewModel(private val model: DreamModel) : ViewModel() {

     var dreams = MutableLiveData<List<DreamTagsDTO>>()

     fun getDreamsOrderByDate() {
         dreams.value = model.getAllDreamsWithTags()
     }
}
```

Zdrojový kód 4 MVVM ViewModel

Na ukázce kódu lze vidět základní implementaci komponenty ViewModel. V konstruktoru se předává reference na modelovou třídu, která komunikuje s databází. Dalším atributem třídy je proměnná *dreams*, která je obalená do komponenty *LiveData*, což umožňuje využívat reaktivní programování. Ve třídě *DreamListViewModel* se vyskytuje pouze jediná metoda, která přiřazuje data z modelové vrstvy do hodnoty komponenty *LiveData*, což upozorňuje pozorovatele, že data byla změna.

```
class DreamListFragment : BaseHomeFragment(), DreamAdapter.DreamClickCallBack
{

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

        viewModel = ViewModelProviders.of(this).get(DreamListViewModel.class)
          .
          .
        viewModel.dreams.observe(viewLifecycleOwner, Observer {
            adapter.updateItems(it)
        })
        viewModel.getDreamsOrderByDate()
    }
}
```

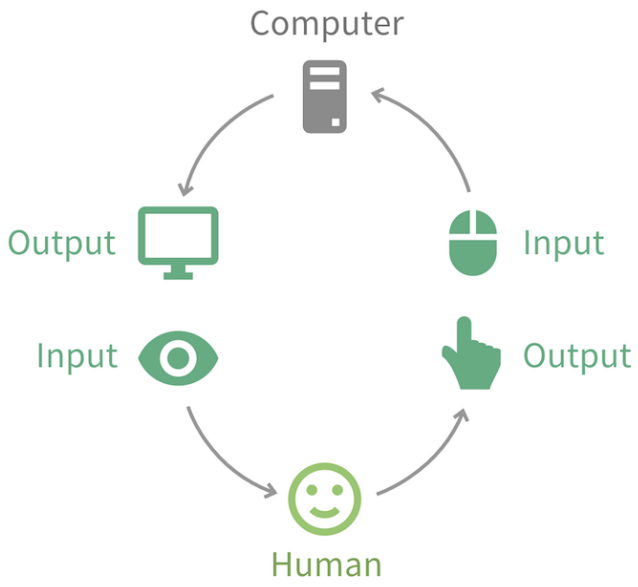
Zdrojový kód 5 MVVM View

V ukázce je zobrazen fragment, který v sobě drží instanci třídy *DreamListViewModel* z předešlé ukázky. ViewModel je inicializován uvnitř těla metody *onViewCreated(View, Budle).* Na dalších řádcích kódu je vidět, že na *LiveData* z třídy ViewModel je zaregistrovaný Observer, který čeká na změnu dat. Ve chvíli, kdy se data ve třídě ViewModel změní, je View informované a může svůj stav aktualizovat.

Z příkladu je vidět, že ViewModel není závislý na View, a proto se stává snadno přenositelným a testovatelným. To platí i o Modelu, který je kompletně oddělen od View a ViewModelu a je absolutně nezávislý na jakýchkoliv Android třídách, tím pádem je velmi snadné ho testovat unit testy. Dalším velkým přínosem komponenty ViewModel je, že není závislý na životních cyklech aktivit či fragmentů, a tak překonává jisté nedostatky z architektury Model View Presenter. [19]

2.5 MVI

Model View Intent patří mezi nejnovější softwarové architektury současnosti. Byl inspirovaný javascriptovou knihovnou Cycle.js a poprvé byl představen v rámci operačního systému Android Hannesem Dorfmannem. MVI spoléhá především na reaktivní a funkcionální programování. Model View Intent si zakládá na MVC, kde byl hlavní cíl oddělit Model od View, avšak zakládá na čisté architektuře MVC, kde každý UI element má svůj vlastní model a svůj vlastní Controller. Příkladem může být jednoduchý CheckBox, který má svůj vlastní model, v podstatě jen boolean. Jako Controller lze označit metodu, která sleduje měnící se stav CheckBoxu, v případě že uživatel vykoná funkci nad touto komponentou, je metoda spuštěna. MVI si také zakládá na tom, aby veškeré toky dat, které procházejí aplikací, byly jednosměrné a neměnné, což samozřejmě vede k čistšímu a udržitelnějšímu kódu. Otázkou je, jak lze zaručit jednosměrný kruhový tok v aplikaci. [22]



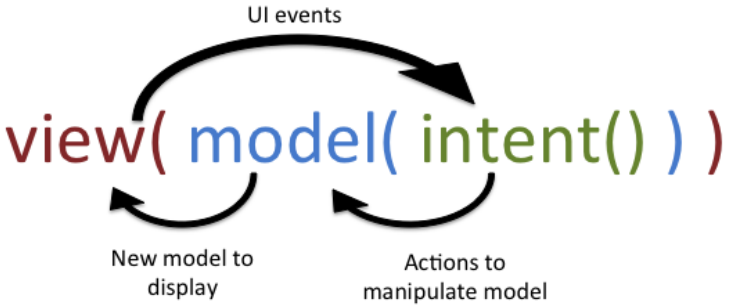
Obrázek 8 Tok dat v MVI [22]

Z obrázku je vidět demonstrace jednosměrného kruhové toku. Fyzické zařízení (mobil nebo počítač) přebere vstup, převede ho a zobrazí ho na displeji uživateli. Uživatel uvidí výstup na UI a produkuje výstup pomocí komponent na obrazovce (textové pole, tlačítko). Jedná se ve své podstatě o matematickou funkci, kterou lze zobrazit následovně.

$$view(model(intent()))$$

- *intent()* – je funkce, která přebírá vstup od uživatele, jako například klinutí na tlačítko, a převede to na „něco“, co bude předáno modelu jako parametr funkce. To může být jednoduchý textový řetězec, číslo nebo komplexnější datové struktury jako akce nebo příkazy.
- *model()* – přebírá výstup od funkce *intent()* jako vstup, který zapříčiní manipulaci s modelem. Výstup této funkce je zcela nový model, který má jiný stav. Nový stav se zakládá z důvodu, že jeden z cílů této architektury je neměnitelnost, takže místo změny stávajícího modelu se vytváří model nový.
- *view()* – přebírá model jako vstupní parametr a poté má View za úkol tento stav zobrazit, může se jednat o načítání nebo zobrazení listu položek. [22]

Po uvedení hlavních funkcí už pouze stačí zajistit kruh tohoto toku funkcí. Tento kruhový tok je vyobrazen na následujícím obrázku.



Obrázek 9 Matematická funkce MVI [22]

Z obrázku je vidět, že předem zmíněná matematická funkce je teď obohacena o události nad jednotlivými funkcemi. Kruh tedy vzniká ve chvíli, když uživatel vyvolá nějako akci na UI, vyvolá a spustí se Intent, který manipuluje s modelem, jenž následně „změní“ svůj stav a UI tento nový model zobrazí. Následující ukázky kódu zobrazují velmi stručnou a základní implementaci aplikace, která má za úkol vyhledat a zobrazit GitHub repositáře. Je v nich použita reaktivní knihovna RxJava. Na následující ukázce je zobrazeno, jak by mohl vypadat model. [22]

```
data class SearchModel(val searchTerm: String, val results: List<GithubRepo>)
```

Zdrojový kód 6 MVI Model

V ukázce je vidět, že se jedná o jednoduchou třídu se dvěma atributy, a to *searchTerm* a *results*. Hlavní třída, která má za úkol vyvolávat akce a měnit model aplikace, je zobrazena na následující ukázce kódu.

```
class SearchActivity : AppCompatActivity() {
    val editSearch: android.widget.SearchView by bindView(R.id.searchView)
        .
        .

    fun onCreate(savedInstanceState: Bundle?) {
        .
        .
        Observable<String> = RxSearchView.queryTextChangeEvents(editSearch)
                .filter { it.queryText().count() >= 3 }
                .debounce(500, TimeUnit.MILLISECONDS)

                // načítání z modelu
                .startWith("")
                .flatMap { queryString ->
                    if (queryString.isEmpty())
                        Observable.just(SearchModel("", emptyList))
                    else
                        githubBackend.search(queryString)
                                .map { response ->
                                    SearchModel(queryString, response.items)
                                }
                }
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe({ result ->
                    adapter.items = result.items
                    adapter.notifyDataSetChanged()
                }, { error -> })
                .
                .
```

Zdrojový kód 7 MVI View

Jako funkci *intent()* v tomto kódu lze označit funkci, která naslouchá změnám nad komponentou *editSearch* a podle toho, co uživatel zadá do vyhledávání, se výsledek *„namapuje“* a předá se do adapteru. V kódu je hodně funkcí z knihovny RxJava, která zde není předmětem zkoumání, ale její hlavní funkcionalita spočívá ve zpracování asynchronního volání měnícího se stavu aktivity. Implementace, která zde byla předvedena, nesplňuje všechny vlastnosti, na které bylo směřováno v ostatních architekturách. MVI lze například kombinovat s architekturou MVP, která byla probírána v předchozí kapitole. Model View Intent je architektura, která se hodí, pokud aplikace bude pracovat s různými stavy aplikace. Další výhodou je, že kód aplikace je dobře čitelný, jelikož využívá jednosměrný kruhový průchod dat, předvídatelný stav a neměnnost dat. [22]

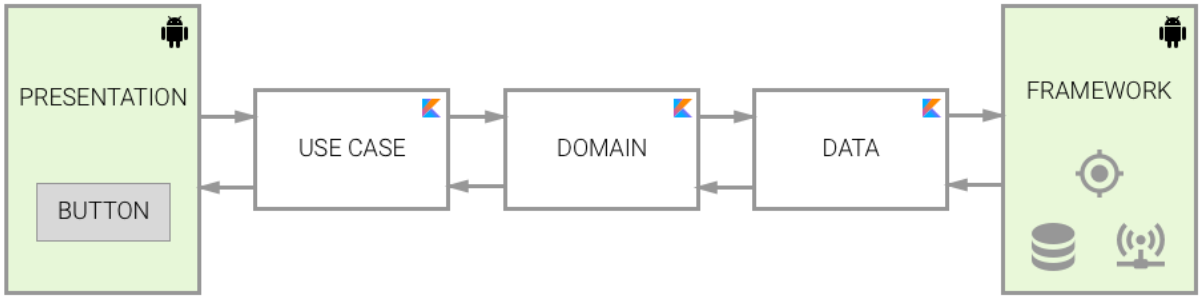
2.6 Clean architecture

Architektury obecně umožňují oddělení různým komponentám kódu organizovaným způsobem za účelem testovatelnosti a lehčího porozumění kódu. Avšak je-li architektura velmi složitá a komplexní, může mít opačný efekt, jelikož vytvoření oddělování kódu vyžaduje vytvoření různých forem hranic, modelů a transformací dat, což má za následek zvyšování učící se křivky,

což může mít vliv na rozsah projektu. Clean architecture je jedna z těchto komplexních architektur. Skládá se z následujících pěti vrstev:

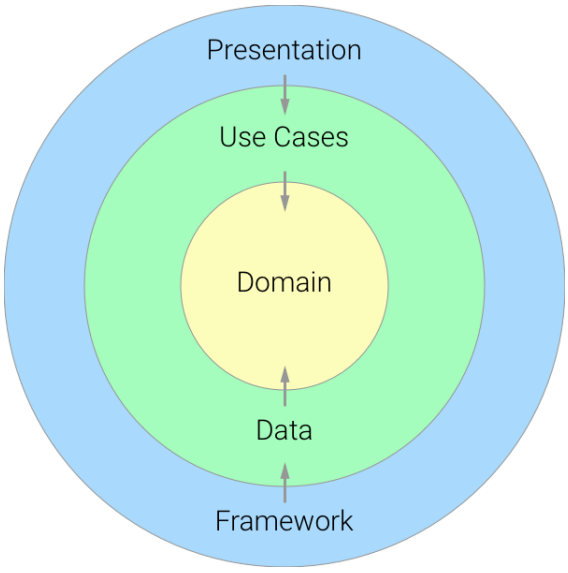
- Presentation – jedná se o vrstvu, která interaguje s uživatelským rozhraním. Někdy bývá tato vrstva rozdělena do dvou, kde jedna obsahuje pouze komponenty týkající se Android frameworku. Tato vrstva obsahuje Android UI (aktivity, fragmenty, view) a Presenter (MVP) nebo ViewModel(MVVM) podle toho, jaký architektonický software je zvolen.
- Use Case – bývá též nazývána interactors. Zde se nachází většinou akce, které může uživatel vyvolat. Může se jednat o akce, jako například uživatel stiskne tlačítko nebo implicitní akce jako přecházení na různé obrazovky aplikace.
- Domain – také nazývána byznys logika. Vrstva Domain obsahuje byznys logiku celé aplikace. Obsahuje veškeré byznys modely. Například u aplikace, která zobrazuje filmy, to mohou být právě třídy Film, Titulky a Herec. Ideálně je tato vrstva největší ze všech, jelikož Android aplikace často pouze odráží jen určitý druh dat, který může být v databázi nebo na API.
- Data – v této vrstvě se vyskytují abstraktní definice různých zdrojů dat a také informace, jak mají být tyto zdroje používány. Zde se často používá vzorec repositář, ten má za úkol při příchozím požadavku rozhodnout, který datový zdroj použít. Typický průběh u Android aplikací je, že aplikace stáhne data ze síťového zdroje a uloží si je lokálně. Tím pádem tato vrstva může zkontrolovat, jestli jsou uložena lokální data a jestli jsou aktuální, pokud tomu tak není, vyšle požadavek na API a vrátí výsledek. Získaná data však nemusí být z databáze nebo z API, jedná se také o data ze senzorů telefonu.
- Framework – ve své podstatě tato vrstva obaluje interakci s frameworkem, takže zbytek kódu může být znovu použitelný v aplikacích napříč různými platformami. V dnešní době to mohou být platformy využívající technologii Kotlin. Frameworkem je zde myšlen nejen Android Framework, ale jakákoliv externí knihovna, jež může být jednoduše v budoucnosti změněna či nahrazena jinou. Tato vrstva by měla být co nejjednodušší a měla by obsahovat co nejméně logiky. [23]

Tyto představené vrstvy jsou pouze jednou z možných implementací. Vrstvy se mohou mezi sebou různě slučovat. Vrstvy mezi sebou různě komunikují, pro demonstraci lze použít následující obrázek.



Zdrojový kód 8 MVI Architektura [23]

Na diagramu lze vidět interakce, které pracují tak, že Presentation vrstva používá vrstvu Use Case, která pak používá vrstvu Domain, která má přístup do vrstvy Data, která používá vrstvu Framework, aby měla přístup k datům. Z diagramu je vidět, že pouze dvě vrstvy (Presentation, Framekwork) využívají Android Framework, zbytek vrstev je oddělený a je samostatně testovatelný nezávisle na platformě Android. To může být maximálně výhodné, pokud vrstva, jež není závislá, jde nadále použít ve webové aplikaci. V souvislosti s Clean architecture lze nalézt různá grafická zobrazení. Jednou z dalších grafických interpretací je následující vrstvený graf.



Zdrojový kód 9 MVI graf vrstev [23]

Z tohoto grafu je vidět, že Clean architecture disponuje vrstvami, které na sobě mají různé závislosti. Vnější vrstvy vykazují vždy závislost pouze na vrstvě, která je pro ně vnitřní, ale už

nemají žádnou závislost vůči vrstvě, která je pod jejich vnitřní vrstvou. Pokud vrstvy komunikují po směru nakreslených šipek, je vše jednoduché, problém však nastává ve chvíli, chcemeli z nějakého důvodu komunikace obrátit. K vyřešení tohoto problému slouží Dependecy Inversion. [23]

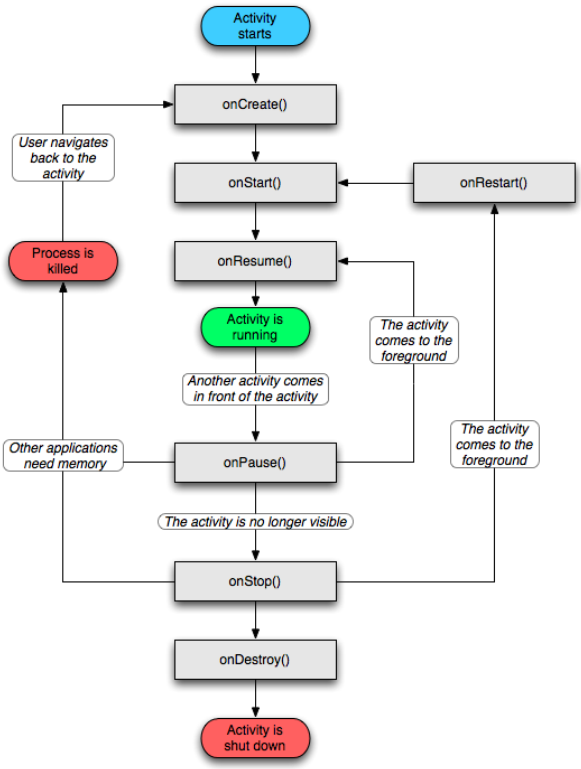
2.7 Shrnutí

Všechny představené architektury se snaží o zpřehlednění kódu, oddělení závislosti, zvýšení možnosti testovatelnosti aplikace, snaží se o znovu použitelnost kódu a celkově o zpříjemnění vyvíjení aplikace. Každý z představených návrhových vzorů s sebou nese jisté výhody a nevýhody. Je těžké a také velmi důležité rozhodnout se pro konkrétní implementaci, jelikož před vývojem je nutné, aby všichni vývojáři měli architekturu nastudovanou a nedocházelo během vývoje k porušování zásad, které jsou spjaté s konkrétní architekturou. Každá architektura má také určité výhody oproti druhé v souvislosti s typem aplikace, jež bude vyvíjena, jelikož každá z architektur disponuje jistou složitostí, kterou je nutné před zahájením vývoje pochopit.

3 ANDROID ARCHITECTURE COMPONETS

V následujícím textu jsou představeny technologie, které jsou zahrnuty do Android architecture components, jež jsou součástí Android Jetpack. V závěru kapitoly bude představen návrh architektury aplikace, kde budou tyto komponenty využity.

Android architecture components je skupina knihoven, která zjednodušuje spravování životních cyklů aktivit a fragmentů. Životní cykly jsou jedním z problémů, na které musí vývojáři myslet při vývoji, jelikož jsou častým důvodem pádu aplikace. Na následujícím obrázku je zobrazen životní cyklus aktivity. [24]

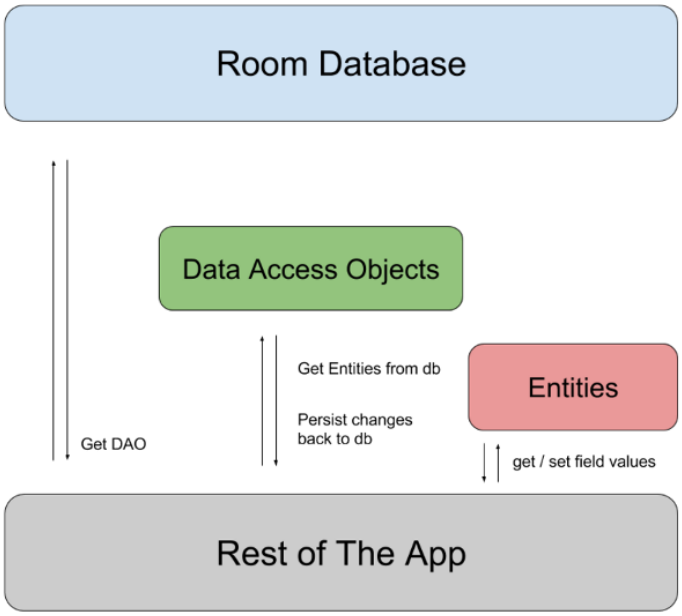


Obrázek 10 Životní cyklus aktivity [13]

Z diagramu lze vidět, že cyklus jedné aktivity je poměrně složitý a jsou různé situace, kdy se tento cyklus opakuje a celá aktivita se vytváří znova. Jedna ze situací, kdy se obnovuje celý cyklus, je přetočení displeje, což nastává poměrně často, a tyto problémy je tedy nutno ošetřit. Právě z toho důvodu byly vytvořeny kolekce knihoven Android architecture components. Mezi tyto knihovny patří především LiveData, ViewModel, Room, DataBinding. Tyto knihovny budou detailněji popsány v následujících podkapitolách. [24]

3.1 Room

Android má ve svém API zahrnutou databázi SQLite, a právě Room je knihovna, která poskytuje abstraktní vrstvu nad touto databází. V základu je zacházení s SQLite poměrně komplikované, jelikož uživatel musí definovat schéma databáze za pomocí textových řetězců, jež nejsou kontrolované, a je velká pravděpodobnost, že při psaní vznikne chyba. Dále musí definovat třídu SQLiteOpenHelper, která poskytuje instanci databáze, vytváří tabulky, plní je daty a zodpovídá za migrace. Navíc k vybírání dat z databáze si musí uživatel definovat kurzory, pomocí nichž se data prochází. Při psaní dotazu se dají využít QueryBuilder, které trochu usnadňují práci s dotazy, ale pokud chce vývojář psát složitější dotaz, nezbývá mu nic jiného než napsat dotaz pomocí jazyka SQL, pomocí textového řetězce, který není kontrolovaný kompilátorem.



Obrázek 11 Room architektura [27]

Začaly se tedy využívat různé ORM knihovny. Objektově relační mapování je technika, která se stará o konverzi dat mezi relační databází a objekty. Tato technika má určitou výhodu v tom, že programátor již nemusí psát dotazy do databáze za pomocí jazyka SQL, ale postačí mu právě znalost ORM, které tyto dotazy vytváří za něj, a to i lépe optimalizovanější. Mnoho těchto knihoven přidává k základnímu mapování i další vlastnosti, jako jsou spravování transakcí, migrace a mnoho dalších. Na druhou stranu, pokud má programátor velmi dobrou znalost SQL jazyka, dokáže dotazy lépe optimalizovat, než kdyby to za něj provedlo ORM. Díky této abstraktní vrstvě je kontrola dění o mnoho menší. Navíc každé ORM pracuje tak trochu jinak, proto je při použití důležité nastudovat jeho dokumentaci. Mezi nejpopulárnější Android ORM kni-

hovny patří Room, Realm, greenDAO anebo DBFlow. Room na rozdíl od těchto knihoven nabízí plnou integraci s ostatními Android architecture components a ověření doby kompilace. [25]

Room podobně jako ostatní knihovny ORM používá anotace k označení tříd, které pracují s Databází. Mezi nejdůležitější patří @Table, @Entity, @Dao, @Query. V následujícím textu budou předvedeny příklady těchto anotací a jejich použití. [26]

Anotace @Entity je využívána pro vytváření tabulek v databázi. Do této anotace je možné přidat různé parametry, jako je název databáze (v případě že se název neuvede, bere se dle názvu třídy), primární a cizí klíče. Následující zdrojový kód představuje tabulku člověk, která má sloupečky id, název, věk a cizí klíč z tabulky zvíře.

```
@Entity(tableName = "human", foreignKeys = [ForeignKey(entity = Animal::class,
parentColumns = ["id"], childColumns = ["id"])])
class Human {
    @PrimaryKey(autoGenerate = true) val id: Long = 0
    @ColumnInfo(name = "name") val name: String = ""
    @ColumnInfo(name = "age") val age: String = ""
    @ColumnInfo(name = "date_born") val dateBorn: Long = 0
}
```

Zdrojový kód 10 Room entity

Každá třída označená anotací @Entity musí mít alespoň jeden primární klíč. Anotace @ColumnInfo slouží k označení sloupečku tabulky a je zde možnost změnit jméno sloupečku, který se jmenuje jako název atributu. @ForeignKey označuje cizí klíč z tabulky Animal, kterému se zadávají parametry jako entity, parentColumns, childColumns. Jejich použití je vidět na zdrojovém kódu nad tímto odstavcem.

Komunikaci s databází zprostředkovávají komponenty Dao, což je označení pro Data access object. Jedná se o objekt, který poskytuje abstraktní rozhraní pro komunikaci s datovou vrstvou. V knihovně Room se pro takovéto objekty používá anotace @Dao. Na následujícím kódu je ukázáno základní rozhraní pro tabulku člověk.

```
@Dao
interface HumanDao {

    @Query("SELECT * FROM human WHERE id = :id")
    fun getHumanById(id: Long) : Human

    @Insert
    fun insertHuman(human: Human)

    @Delete
    fun deleteHumanById(id: Long)
}
```

Zdrojový kód 11 Room Dao

Na příkladu lze vidět použití anotací @Query, @Insert a @Delete. V případě metody *getHumanById(Long)* se pomocí sestaveného dotazu v těle anotace @Query vybere člověk z databáze na základě jeho id a zbylé dvě metody vyskytující se ve třídě HumanDao představují vkládání a mazání entity z databáze. Textový řetězec, který je umístěn v těle anotace @Query, je kontrolován kompilátorem, tím pádem kód nejde zkompilovat, není-li dotaz sestaven správně, například je-li uveden chybný název tabulky. Krom objektu člověka je možné pomocí @Query vrátit například i list nebo kurzor do databáze. Navíc jelikož je Room součástí Android Architecture components, mohou se návratové hodnoty dotazů obalit do knihovny LiveData, o které je psáno v následující kapitole.

Další anotací je @Database, která slouží pro vytváření samotné databáze. Tato třída musí dědit ze třídy RoomDatabase a poté se musí uvést, které tabulky v databázi existují. Dále se uvádějí třídy Dao, jež napříč aplikací používáme. K databázi je nutné přistupovat z jiného vlákna než z hlavního, pokud uživatel neuvede opak. To slouží k tomu, aby se nestávalo, že aplikace zamrzne při vykonávání dotazu do databáze. Příklad této anotace je vidět na následujícím kódu. [28]

```
@Database(entities = arrayOf(Human::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): HumanDao
}
```

Zdrojový kód 12 Room databáze

Do databáze SQLite jdou uložit převážně primitivní typy, jako jsou Integer, Double, String, Byte. Pokud je potřeba uložit nějaký objekt, příkladem může být i datum, je zapotřebí použít konvertory, které Room nabízí. Pro použití je potřeba sestrojit metodu, která bude převádět objekt na jednoduchý datový typ, který lze použít v databázi, a metodu na převod z datového typu hned na objekt. Aby byla databáze schopna použít tento konvertor, je nutné označit třídu

obsahující převodové metody anotaci @TypeConvertor a poté tuto třídu zaregistrovat při vytváření databáze. Následující kód je příklad, při kterém se převádí Date na Long a zpět. [29]

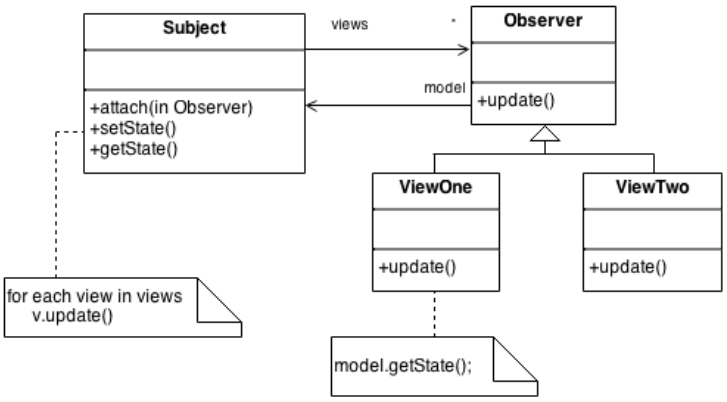
```
class DateConvertor {
    @TypeConverter
    fun fromTimestampToDate(value: Long): Date {
        return Date(it)
    }

    @TypeConverter
    fun dateToTimestamp(date: Date): Long {
        return date.time.toLong()
    }
}
```

Zdrojový kód 13 Room konvertor typů

3.2 LiveData

V případě, že aplikace zobrazuje nějaká data ať už z databáze nebo z nějaké služby třetí strany, je důležité mít data co nejaktuálnější. Proto je důležité naslouchat změnám těchto dat, aby aplikace byla schopna reagovat v okamžiku, kdy jsou data změněna. Obvykle je to velký problém, jelikož ne vždy je zdroj dat známý a míst, kde mohou být data změněna, může být několik. Je potřeba naslouchat na více komponentách zároveň a tím může mezi nimi vzniknout závislost. To ovšem komplikuje testování a debug aplikace.

Právě pro řešení těchto problémů vznikla knihovna LiveData. Pomocí nich je vždy UI aplikace v aktuálním stavu, jelikož jsou vytvořeny pomocí návrhového vzoru Observer (Pozorovatel). Observer je takový návrhový vzor, který v podstatě definuje relaci 1:N (pozorovaný : pozorovatelé), proto pokaždé když se změní stav pozorovaných dat, všichni, kteří naslouchají, jsou ve stejné chvíli upozorněni a mohou na změny reagovat. Na následujícím UML diagramu je znázorněna implementace návrhového vzoru Observer. [30]



Obrázek 12 Struktura návrhové vzoru Observer [19]

Další nesporovou výhodou je, že je lze navázat na komponenty, které podléhají životnímu cyklu, o kterých bylo psáno v úvodu Android architecture components. Pokaždé, když je komponenta zničena, vázaný Obsever je zničen s ní. V případě, že aktivita či fragment jsou v pozadí, Observer neregistruje žádné události, jelikož LiveData jsou vázána na životní cyklus aktivity či fragmentu. LiveData sama o sobě nelze změnit, jelikož je jejich hodnota value konstanta. Pokud je potřeba data měnit, používá se třída MutableLiveData, která dědí z třídy LiveData. Tato metoda přidává dvě veřejné metody setValue(T) a postValue(T). Následující kód je příklad, jak lze LiveData definovat a jak nad daty definovat Observer.

```
var isListening = MutableLiveData<Boolean>()
.
.
.
isListening.value = !(isListening.value ?: false)
.
.
.
viewModel.isListening.observe(viewLifecycleOwner, Observer { isListening ->
    changeListenButton(isListening)
})
```

Zdrojový kód 14 LiveData příklad

LiveData obvykle bývají definována uvnitř třídy ViewModel, fragment nebo aktivita drží jeho instanci a skrz něj naslouchají změnám nad daty. Na prvním řádku kódu lze vidět, jak se LiveData definují, a na následujících řádcích je vidět změna a naslouchání nad samotnými daty. Vzhledem k tomu, že LiveData jsou součástí Android architecture components, mají tedy implementovanou integraci s Room databází, která je schopna vracet data z databáze obalená právě touto komponentou, tím je zaručeno, že data zobrazující se v aplikaci budou konzistentní s daty v databázi. [31]

3.3 ViewModel

Jeden z hlavních úkolů třídy ViewModel je udržovat a spravovat data, která jsou vázána na životní cyklus, tedy data, která by se měla udržovat v konzistentním stavu nehledě na to, kterým životním cyklem daná aktivita či fragment prochází. ViewModel tedy umožňuje „přežití“ nehledě na konfigurační změny, jako je například rotace displeje. Jak už bylo představeno v úvodu této kapitoly, životní cyklus komponent může být někdy nevyzpytatelný, jelikož kontrolu životních cyklů má na starost Android framework. Ten se může někdy rozhodnout (například z důvodu uvolnění operační paměti nebo přenesení aplikace do pozadí) znovu vytvořit celou

aktivitu. V tomto případě je velmi důležité zachovat data, která se před tímto zničením v aplikaci vyskytovala. Pro tyto případy se používá metoda onSaveInstanceState(Bundle), která je volána těsně předtím, než jsou data zahozena, a tato data se objeví v balíčku, jenž je předán do metody OnCreate(Bundle). Tato metoda se ovšem hodí pouze v případech, kdy se jedná o malý počet dat, která jsou schopna jednoduše serializovat a poté deserializovat. Serializace nebo Deserializace je proces, při kterém se objekt konvertuje do proudu bytů a poté je uložen (Deserializace je proces opačný). [32], [33]

Dalším problémem je, že UI kontroléry potřebují provádět asynchronní volání a následně potřebují čas pro navrácení. Kontrolér potřebuje provádět tato volání a musí si být jist, že systém po sobě prostor vyčistí, aby nedocházelo k únikům paměti. Tento celý proces vyžaduje velkou údržbu a v případě, že se znovu vytvářejí nějaké objekty z důvodu konfiguračních změn, je to plýtvání zdrojů, jelikož se může stát, že objekt může požadovat některé další své volání.

Aktivity a fragmenty jsou primárně určeny pro zobrazování dat a odpovídání na akce vyvolané uživatelem či systémem, jako jsou například udělování různých povolení (přístup ke kameře, souborům a tak dále). Vezmou-li se všechny akce v potaz, je velmi náročné kontrolérům přenechat i načítání dat z databáze nebo internetu. Je tedy lepší tyto všechny úkony delegovat mezi různé třídy, což umožňuje i možnost testovatelnosti a je mnohem snazší a výkonnější oddělit spravování dat od UI kontrolérů. Z toho důvodu se implementuje třída ViewModel z Android architecture components. Na následujícím kódu je zobrazen příklad, jak lze ViewModel implementovat. [33]

```
class MyViewModel : ViewModel() {
    private val users: MutableLiveData<List<User>> by lazy {
        MutableLiveData().also {
            loadUsers()
        }
    }

    fun getUsers(): LiveData<List<User>> {
        return users
    }

    private fun loadUsers() {
        // místo pro asynchronní volání pro načtení uživatelů.
    }
}
```

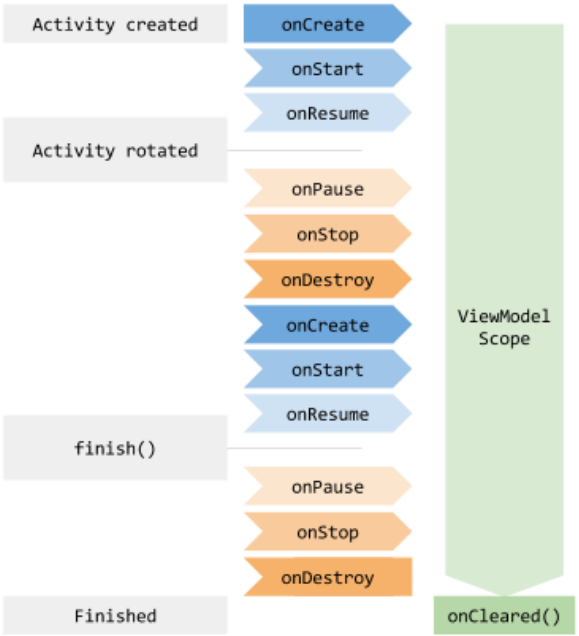
Zdrojový kód 15 ViewModel příklad

Z ukázky je vidět, že pro implementaci je nutné dědit ze třídy ViewModel. ViewModel Objekty uvnitř instance jsou automatiky svázány s jeho životním cyklem, který je schopen přežít různé konfigurační změny. Na kódu vidíme, že ViewModel zde drží instanci objektu LiveData, která obalují list uživatelů. K tomuto listu lze z aktivity přistoupit následovně.

```
class MyActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {

        val model = ViewModelProviders.of(this).get(MyViewModel::class.java)
        model.getUsers().observe(this, Observer<List<User>>{ users ->
            // aktualizace UI
        })
    }
}
```

Zdrojový kód 16 ViewModel vytvoření

Na ukázce je vidět, že pro přístup k uživatelům se volá metoda getUsers(), která vrátí LiveData, na kterých je zaregistrovaný Observer. Pokud je například aktivita znovu vytvořena, LiveData uvnitř třídy ViewModel přežijí tuto konfigurační změnu. Životní cyklus komponenty ViewModel je vidět na následujícím obrázku.

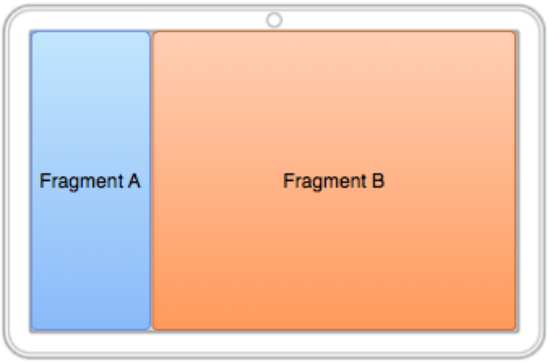


Obrázek 13 Životní cyklus komponenty ViewModel [33]

Na obrázku lze vidět, že ke zničení dojde až v případě, že aktivita je úplně zničena v metodě onDestroy() a ve třídě ViewModel se volá metoda onCleared(). Ve většině případů se instance vytváří v metodě onCreate(), ale je možné, že systém zavolá metody onCreate() vícekrát za dobu života aktivity například při rotaci, ale ViewModel přežívá do té doby, než je na aktivitě zavolána metoda finish() a aktivita je úplně zničena.

Bývá častým příkladem, že v jedné aktivitě existují dva fragmenty, které mezi sebou potřebují sdílet jedna a ta samá data nebo potřebují mezi sebou komunikovat. Konkrétním případem může

být list položek (fragment A) a detail položky (fragment B), které jsou zobrazeny v jedné aktivitě ve dvou fragmentech a fragmenty leží vedle sebe. Tento příklad je ilustrován na následujícím obrázku.



Obrázek 14 Příklad dvou fragmentu v aktivitě [34]

V případě, že uživatel změní položku v detailu, je nutné, aby list byl ihned aktualizovaný a uživatel viděl, že danou akci provedl správně. Tento příklad zní jednoduše, ale v praxi tomu tak není. Je nutné definovat rozhraní, pomocí kterých tyto dva fragmenty budou komunikovat, a aktivita musí tyto dva fragmenty tímto rozhraním spojit. Tento případ lze vcelku snadno vyřešit užitím komponenty ViewModel. Ten se vytvoří v prostoru aktivity a ViewModel sám už se postará o komunikaci mezi fragmenty. Použije-li se toto řešení, přináší to s sebou určité výhody v podobě, že aktivita a fragmenty mezi sebou nemají žádnou závislost. Tudíž je tu splněna podmínka oddělitelnosti a znovu použitelnosti. Další výhodou je, že každý z fragmentů má svůj vlastní životní cyklus a nezávisí na cyklu fragmentu druhého. Následující kód zobrazuje, jak by mohla vypadat případná implementace. [33]

```
class MasterFragment : Fragment() {

    private lateinit var model: SharedViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        model = activity?.run {
            ViewModelProviders.of(this).get(SharedViewModel::class.java)
        }
    }
}
class DetailFragment : Fragment() {

    private lateinit var model: SharedViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        model = activity?.run {
            ViewModelProviders.of(this).get(SharedViewModel::class.java)
        }
    }
}
```

Zdrojový kód 17 Sdílený ViewModel mezi fragmenty

3.4 Paging library

Poskytuje rozhraní pro postupné načítání dat dle potřeby z jakéhokoliv zdroje. Řeší hlavně problém s načítáním zbytečně velkého množství dat. Stává se, že aplikace pracuje s velkým množstvím dat, ale zobrazuje se jen určitá část v čase. Příkladem mohou být komentáře k příspěvku nebo fotky z nějaké API, která poskytuje tisíce obrázků. Pokud by aplikace nepřihlížela k těmto okolnostem, je možné, že zařízení nebude stíhat data načítat a stahovat zároveň a aplikace bude mrznout a UI aplikace nebude responzivní vůči uživateli. Knihovna Paging library poskytuje třídy, které obsahují již existující řešení. Jelikož je součástí Android architecture components, poskytuje kooperace s ostatními komponenty jako je Room. [35]

Klíčová komponenta obsažená v knihovně je PagedList, která načítá data po částech (stránkách). Pokud je potřeba více dat, než je zobrazeno, jsou přidána do současného instance PagedListu. Nastane-li situace, že data uvnitř PagedListu jsou změněna, je emitována nová instance, například do LiveData. Na následujícím krátkém příkladu je zobrazeno použití PagedListu. [36]

```
class ConcertViewModel(concertDao: ConcertDao) : ViewModel() {
    val concertList: LiveData<PagedList<Concert>> =
        concertDao.concertsByDate().toLiveData(pageSize = 50)
}
```

Zdrojový kód 18 PageList příklad

3.5 Data binding

Jedná se o knihovnu, která je součástí Android Jetpack a poskytuje rozhraní, pomocí kterého je možné svázat UI komponenty s daty. Rozložení UI, dále layout, je často definováno přímo v aktivitě a naplnění UI komponent je voláno programově. V následující ukázce je popsáno volání findViewById<T>(Int), které bere za parametr id komponenty, které je definováno v layoutu aktivity a návratový typ funkce je TextView.

```
val textView = findViewById<TextView>(R.id.sample_text)
textView.text = viewModel.userName
```

Zdrojový kód 19 Způsob nalezení View v layoutu

Na druhém řádku je vidět přiřazení uživatelského jména do textu TextView. Tento kód se opakuje pokaždé, když je potřeba přiřadit hodnotu do komponenty z layoutu. Tuto duplicitu lze vyřešit použitím knihovny Data Binding, kdy je UI komponenta svázána s určitou hodnotou objektu. Svazování komponenty se vyskytuje přímo v layoutu, takže kód už se nenachází přímo

v aktivitě, a to přispívá pro zmenšení a zpřehlednění kódu. Následující příklad zobrazuje nahrazení volání z předešlé ukázky pomocí Data Binding.

```
<TextView  android:text="@{viewmodel.userName}" />
```

Zdrojový kód 20 DataBinding příklad

Aby bylo možné přistupovat přímo k atributům aktivity, je nutné definovat atribut přímo v layoutu pomocí expresivního jazyka, který je implementovaný v knihovně Data Binding. Ten umožní propojení layoutu s proměnnými z aktivity, fragmentu nebo adaptéru. Po zápisu a sestavení aplikace knihovna automaticky vygeneruje třídy, které jsou potřebné ke svázání UI a datových objektů. Kód pro Data Binding může vypadat následovně.

Z kódu je vidět, že celý layout se obalí do tagů *layout* a uvnitř tagu *data* se nachází proměnné, které jsou využívány napříč celým layoutem. Jelikož Data Binding knihovna je poskytuje třídám metody, které naslouchají změnám dat, nemusí se programátor starat o opětovné nastavování UI komponent. [37]

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <data>
        <variable
            name="viewmodel"
            type="com.myapp.data.ViewModel" />
    </data>
    <ConstraintLayout... /> <!-- UI kořenový element layotu-->
</layout>
```

Zdrojový kód 21 Příklad vložení DataBinding do layoutu

3.6 Ukázka architektury aplikace

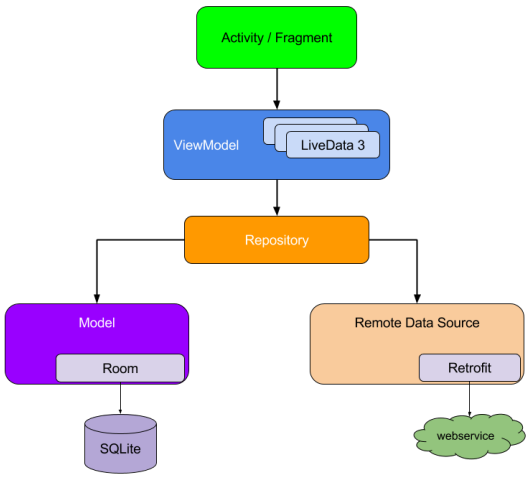
Ve většině případů mají desktopové aplikace či programy jeden spouštěcí bod. Android aplikace na rozdíl od toho mají mnohem komplikovanější strukturu. Typická aplikace na platformě Android obsahuje velké množství různých komponent, jako jsou aktivity, fragment, služby a mnoho dalších. Většina z těchto komponent je definována v souboru AndroidManifest.xml, tento soubor je používán operačním systémem, který pomocí něho rozhoduje, jakým způsobem integrovat aplikaci do uživatelských zařízení. Z toho vyplývá, že uživatel interaguje s různými aplikacemi v krátkém časovém úseku a aplikace se musí přizpůsobit různým situacím, ve kterých se mohou díky tomu vyskytnout, jelikož neexistuje právě jeden způsob, jak aplikaci spustit či obnovit. Příkladem může být, že aplikace může být spuštěna z notifikace nebo že se spustí v určitý čas jako budík. Složitějším příkladem může být, když uživatel chce sdílet fotku na nějaké své oblíbené sociální síti. Aplikace musí spustit fotoaparát v mobilu, v tu chvíli

opouští aplikaci a ocitá se v aplikaci Kamera. Aplikace Kamera může ale spustit další různé aplikace, nakonec však uživatel musí skončit v aplikaci, ze které vyšel. Když aplikace obdrží pořízenou fotku, může uživatel sdílet fotku na sociální síti. Během celého tohoto procesu může být uživatel přerušen například hovorem či SMS zprávou. V nejhorším případě se může stát, že uživatel bude mít zabrán celý proces paměti, a aplikace tedy může být systémem zabita, takže musí na všechny tyto faktory reagovat korektně a při návratu do aplikace musí být stav takový, jaký si uživatel pamatoval naposledy. Tyto scénáře nejsou nijak ojedinělé, proto vývojář musí vždy dávat pozor a měl by se vyvarovat ukládání jakýkoliv dat do komponent, které jsou závislé na stavu aplikace.

Častou chybou, jíž se vývojáři dopouštějí, je, že veškerý kód umisťují do aktivit či fragmentů a porušující princip oddělení závislostí. Tyto UI komponenty by měly obsahovat logiku, která se týká logiky UI a ničeho jiného. Dodržením tohoto principu se lze vyhnout problémům týkajícím se životních cyklů, které byly zmíněny na začátku této kapitoly. Mezi další principy, které by se měly při vývoji dodržovat, patří UI řízené datovou vrstvou. Dodržením tohoto principu nemohou vzniknout nekonzistence mezi UI a vrstvou obsahující data. Jako další výhodu lze uvést, že pokud aplikace bude zabita operačním systémem, uživatel v tu chvíli neztrácí data. [38]

3.6.1 Doporučená architektura android aplikace

V této podkapitole bude provedena ukázka možné architektury aplikace, avšak musí být bráno na vědomí, že neexistuje ideální architektura pro jakýkoliv možný scénář. Následující obrázek znázorňuje, jak by mohla architektura vypadat.



Obrázek 15 Architektura aplikace za pomocí Android architecture components [38]

Jako hlavní komponenty jsou zvoleny komponenty z Android architecture components, které byly rozebrány v předcházejících kapitolách. Z architektury na obrázku je vidět, že každá komponenta aplikace závisí pouze na jedné komponentě o úroveň níž. Například aktivita nebo fragment závisí pouze na třídě ViewModel. Ten závisí na repositáři, který může záviset na více třídách jako databáze nebo webová API.

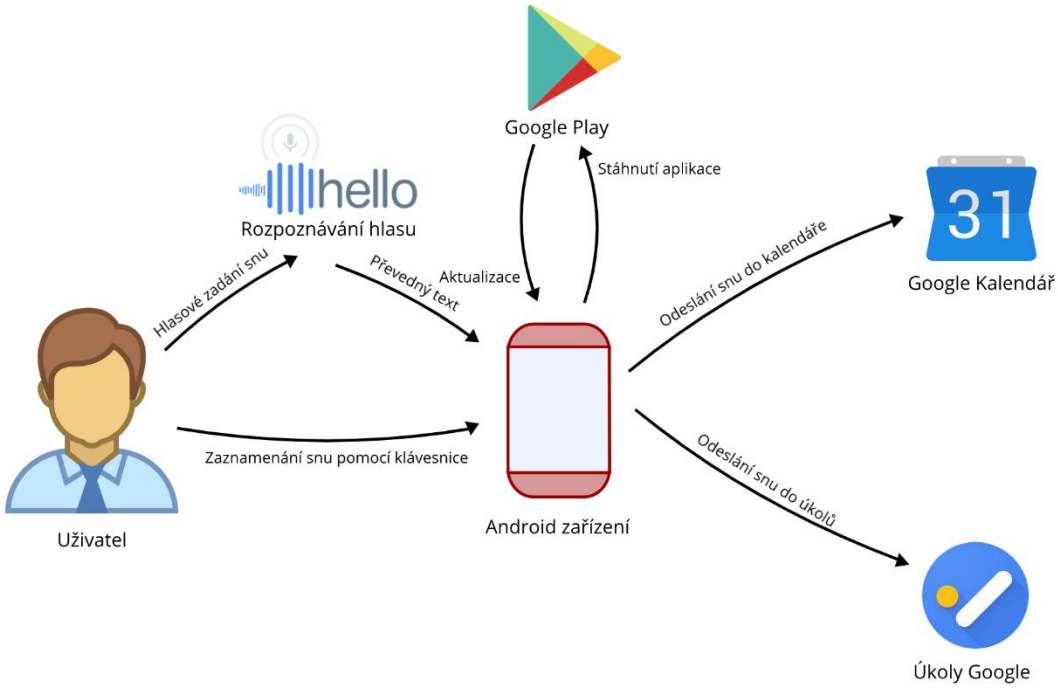
Hlavní komponenta UI je zde vytvořena pomocí aktivity nebo fragmentu, ale bývá zvykem, že aktivita je brána pouze jako obal, který obsahuje fragment či fragmenty. Ty drží jeho instanci a implementují Observer, který naslouchá změně dat. Uvnitř třídy ViewModel krom instance LiveData můžeme implementovat proměnné, které si drží stav obrazovky. Mimo to ViewModel přistupuje do třídy repositáře, který slouží jako rozhraní mezi různými zdroji dat. Na obrázku existují dva zdroje, a to datová vrstva a webové API. S databází aplikace se komunikuje pomocí již dříve zmíněného ORM Room. Room je zde prostředkem mezi databází SQLite. Repositář obsahuje instanci databáze, která si drží různé třídy Dao, které už přímo komunikují s databází. Další instancí, kterou obsahuje repositář, je instance pro webové rozhraní. Na obrázku je vidět, že je zde zvolena populární knihovna mezi android komunitou Retrofit. Retrofit je HTTP klient, který slouží jako prostředek ke komunikaci s webovou API. Na následujícím příkladu bude uvedeno, jak může celá aplikace s těmito komponenty pracovat.

Uživatel otevře aplikaci a začne se mu načítat úvodní obrazovka, která obsahuje list položek. Aktivita poskytne UI komponenty a zavolá metodu skrz ViewModel pro stažení dat. ViewModel přepne stav aplikace na načítání a aktivita na tuto změnu patřičně zareaguje. Zatímco aktivita zobrazuje stav načítání, ViewModel si vyžádá data z API po repositáři, který se rozhodne, zda má data lokálně uložena, anebo musí vyvolat požadavek API. Po navrácení dat se předají data do třídy ViewModel a tam nastaví hodnotu LiveData, na kterou reaguje aktivita a zobrazuje výsledek. Při tomto procesu může docházet k chybám, které mohou být ošetřeny změnami stavu, na které reaguje aktivita.

Na uvedeném příkladu jsou dodrženy principy oddělení závislostí, znovu použitelnosti a UI řízeného datovým modelem. Vzhledem k tomu, že každá vrstva je takto oddělená, zvyšuje se úroveň testovatelnosti aplikace. Při vývoji je nutné myslet na to, že tento postup má i své stinné stránky, Implementace celé této architektury je poměrně náročná, jelikož vývojář musí znát vlastnosti implementovaných komponent. Komponenty se musí zvlášť importovat do projektu, protože nejsou součástí jádra Android frameworku, a to má za následek, že sestavný soubor APK zabírá více místa. [38]

4 VÝVOJ APLIKACE MANAŽER SNŮ

V této kapitole bude popsána praktická část diplomové práce, která je zaměřena na vývoj mobilní aplikace pro platformu Android. Výsledná aplikace bude sloužit pro zaznamenávání snů uživatele. Uživatel bude mít možnost zadat sen pomocí jednoho kliknutí, nebo za pomoci hlasu. Nadiktovaný sen lze poté označit, zvolit typ nálady po probuzení, což slouží ke zpracování snů pro statistiky. Po vyplnění snu bude aplikace umožňovat sen odeslat do aplikací třetích stran (Google Calendar, Google Tasks). V aplikaci je také zabudovaný budík, který ulehčuje zaznamenání snu, jelikož uživatel je po jeho ukončení dotázán, jestli se mu v noci nějaký sen zdál. Jak již bylo avizováno, v rámci aplikace, je naprogramovaná obrazovka, jež poskytuje statistiky o snech uživatele. Aplikace bude také publikována na Google Play. Celý ekosystém aplikace je vidět na následujícím obrázku.



Obrázek 16 Ekosystém aplikace

Na základě rešerše uvedené v kapitole Softwarová Architektura, byla zvolena architektura MVVM. Zejména díky své dobré integraci s Android Architecture Components, jež byly představeny ve stejnojmenné kapitole. Dalším důvodem pro zvolení této architektury byla správa životního cyklu v rámci aktivity či fragmentu. Design aplikace byl navržen a nakreslen v grafickém nástroji Gravit a design aplikace dodržuje pokyny Material Design. K naprogramování aplikace byl použit jazyk Kotlin, který je v současnosti oficiální jazyk pro Android aplikace a celá aplikace byla vyvinuta ve vývojovém prostředí Android Studio.

4.1 Motiv vývoje

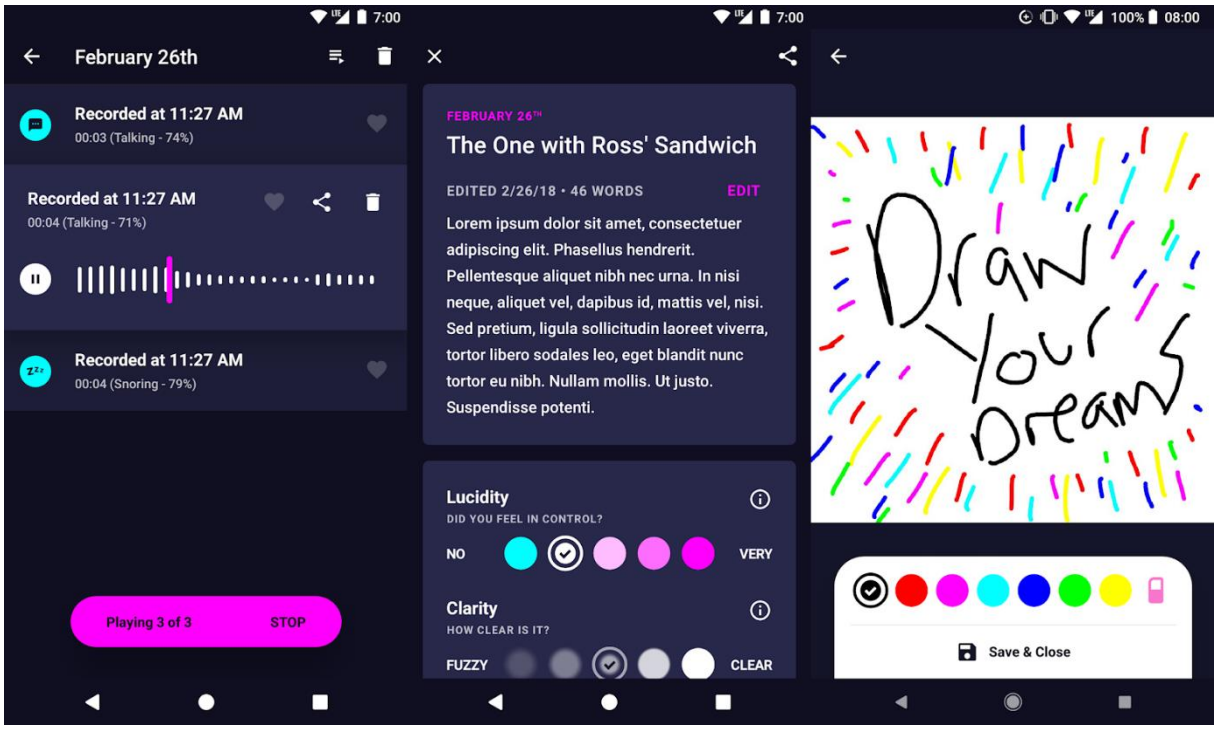
Jedním z hlavních motivů vývoje této aplikace byla analýza snů, která se využívá již řadu let v psychoterapii. Se začleňováním snů do psychoterapie začal Sigmund Freud, který věřil, že sny jsou často cestou do našeho nevědomí. Tuto myšlenku pak dále rozšiřoval Carl Gustav Jung. Analýza snů bývá aplikována na pacienty, kteří mají různé psychické potíže nebo na pacienty, jež touží po svém osobním rozvoji. Při snění naše nevědomí vyjevuje různé symboly a je právě úkolem analýzy snů tyto symboly identifkovat a pokusit se rozkrýt jejich význam. K zaznamenání snů lidé používají často papír a tužku, aplikace které slouží jako zápisník anebo dokonce aplikace, které jsou tomu přímo určeny. V následující kapitole, Analýza dostupných aplikací na trhu, bude několik těchto aplikací popsáno. [39]

4.2 Analýza dostupných aplikací na trhu

Tato podkapitola se zaobírá aplikacemi, které slouží k zaznamenání snů. První bude představena aplikace Luci, jež uživatelům pomáhá docílit stavu lucidního snění. Jako druhá bude představena aplikace Dream Catcher, jež disponuje funkcí zálohování snů.

4.2.1 Luci

Tato aplikace je dostupná pouze pro zařízení s operačním systémem Android a vyvinul jí vývojář Sam Ruston. Aplikace slouží pro uživatele, kteří si chtějí zaznamenávat sny, a přitom jim má pomáhat k dosažení lucidního snění. Aplikace obsahuje mnoho dalších funkcionalit jako například kreslení snů, nahrávání spánku, které zaznamenává mluvení pří spánku. Mimo to obsahuje vlastnost zamknutí aplikace nebo například synchronizaci mezi zařízeními. Na následujících obrázcích je tato aplikace představena na třech snímcích.

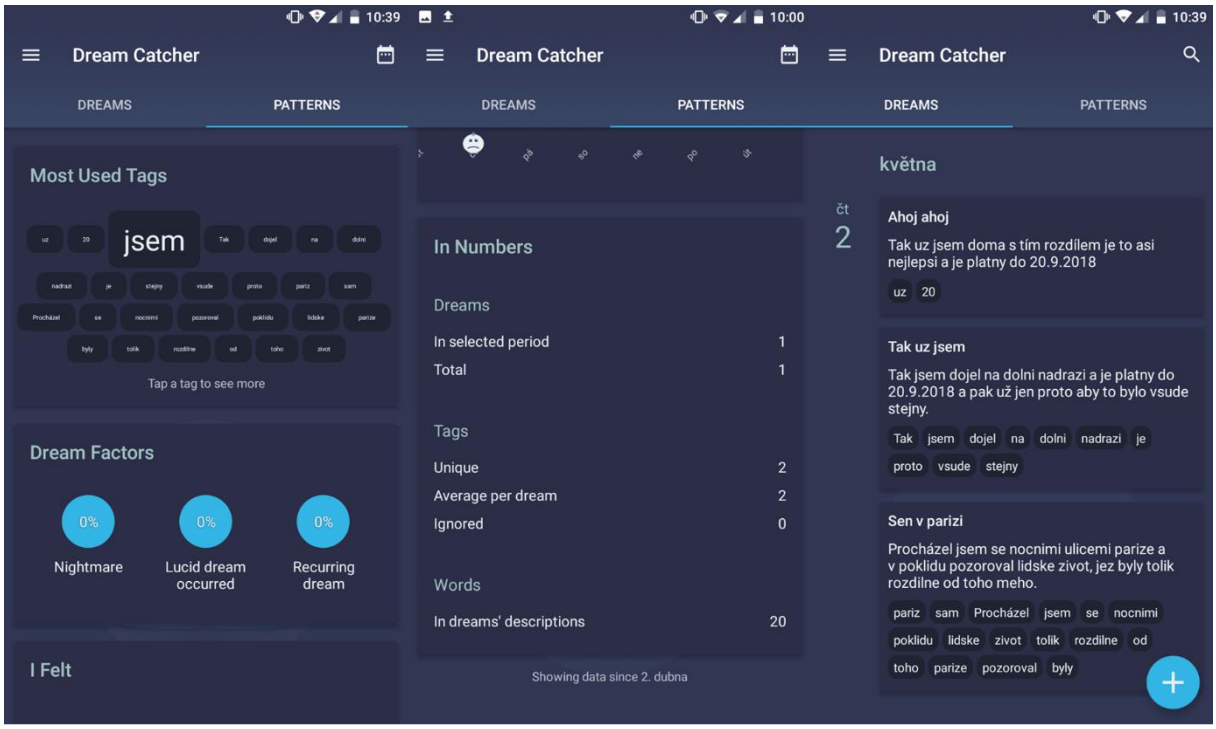


Obrázek 17 Aplikace Luci

Na snímcích je vidět, že aplikace je barevně sladěna do tmavého stylu a na první pohled splňuje vlastnosti Material Design. Na levé straně obrázku je snímek obrazovky, na kterém je vidět zaznamenaná nahrávka spánku. Uprostřed obrázku je vidět snímek s detailem zaznamenaného snu, na kterém je vidět, že sen obsahuje nadpis a svůj popis. Krom těchto dvou vlastností aplikace zaznamenává, zda uživatel měl kontrolu nad snem, jež se mu zdál a jak moc byl sen neuspořádaný. Na snímku vpravo je zobrazeno rozhraní pro kreslení snu. Aplikace se na Google Play nevyskytuje zdarma a pro základní použití je nutné za aplikaci zaplatit 54, 99 Kč. Celkové zpracování aplikace působí velmi dobře.

4.2.2 Dream Catcher

Aplikace Dream Catcher je dostupná pouze pro zařízení s operačním systémem Android 4.2 a vyšší. Aplikace se přímo zaměřuje na zaznamenávání snů v textové formě. Krom toho, nabízí také zámek aplikace, který umožňuje aplikaci zamknout na čtyřmístný pin, což slouží jako ochrana před přečtením zaznamenaných snů jinými uživateli. Zaznamenaný sen je možné poté dále rozšiřovat o různá označení, jako například jestli byl sen noční můra, či se jednalo o lucidní sen nebo jestli se sen uživateli již zdál. V aplikaci je také obrazovka, která nabízí statistiky, jež pomáhají mít určitý přehled nad svými sny. Aby uživatel nezapomínal zaznamenávat své sny, je možné si v aplikaci nastavit čas, ve který chce být uživatel notifikován. Grafické rozhraní aplikace je zobrazeno na následujícím obrázku, skládajícího se ze třech snímků obrazovek.



Obrázek 18 Aplikace Dream Catcher

Na prvním snímku z levé strany a ze snímku uprostřed jsou vidět již zmíněné statistky. Statistiky uživateli zobrazují například nejpoužívanější značky napříč všemi sny, procentuálně ukazují různé druhy snů a zobrazují statistiky snů v číslech, což je vidět na prostředním snímku obrázku. Na pravém snímku je vidět seznam zaznamenaných snů. Aplikace se v obchodu Google Play vyskytuje zcela zdarma, avšak má v sobě placené funkcionality, jako například synchronizaci všech snů do Cloudu.

4.3 Shrnutí

Obě představené aplikaci v rámci rešerše působí velmi dobře a byly také inspirací k vývoji aplikace Manažer snů, jež je předmětem praktické části této práce. Avšak žádná z těchto aplikací nenabízí způsob zadávání snů pomocí hlasu, ke kterému stačí stisknout jediné tlačítko a jsou zaměřeny spíše na pomoc dosažení stavu lucidního snění. Představené aplikace také nedisponují českým jazykem a není v nich možnost odesílat zaznamenané sny do aplikací třetích stran. A to jsou důvody, které slouží jako další motivace k vývoji aplikace Manažer snů.

4.4 Analýza

V rámci této kapitoly budou uvedeny funkční/nefunkční požadavky a UML diagramy. Za začátku každé podkapitoly, bude každá problematika vysvětlena a popsána.

4.4.1 Funkční a nefunkční požadavky

Jedná se o proces stanovující služby, jež by měl daný systém poskytovat a také určitá omezení, přes která by měl zkoumaný systém pracovat. Ve své podstatě se v tomto procesu definuje, co by měl systém dělat, ale nezkoumá, jakým způsobem by to měl daný systém zařizovat. V tomto procesu se hledají dva různé druhy požadavků, a to funkční a nefunkční. Kde funkční požadavky se snaží popsat určitou službu daného systém a na druhou stranu nefunkční požadavky stanovují omezení, jež jsou kladné na proces vývoje či na systém samotný. Při zápisu požadavků se dodržují určitá pravidla zápisu. Zápis požadavku definuje následující funkce.

$$< id >< systém > \boldsymbol{bude}(musí\ umět) < funkce >$$

Ze zápisu je vidět, že každý požadavek má své *id,* díky kterému ho je možné okamžitě identifikovat. Následně je uváděn název systému, kterého se požadavek týká. Poté se uvádí sloveso bude či musí umět. Poslední je uváděna daná funkce. V následující tabulce budou uvedeny funkční a nefunkční požadavky aplikace Manažer snů, kde nejvyšší priorita je označena číslem 1. [41]

ID	Funkce	Priorita
F1	Manažer snů bude spravovat záznamy o snech.	1
F2	Manažer snů bude umožňovat označovat jednotlivé sny značkami.	2
F3	Manažer snů bude umožňovat zadávat popis snu pomocí hlasu.	1
F4	Manažer snů bude ukládat, načítat a zobrazovat sny.	1
F5	Manažer snů bude umožňovat sdílet sny s aplikacemi třetích stran.	1
F6	Manažer snů bude umožňovat spravovat alarmy.	2
F7	Manažer snů bude umožňovat ukládat, načítat a zobrazovat alarmy.	2
F8	Manažer snů bude zapínat budíky v zadaný čas a den.	2
F9	Manažer snů bude umožňovat odkládání budíku.	2
F10	Manažer snů bude umožňovat spravovat čas odložení.	3
F11	Manažer snů bude zobrazovat různé statistiky.	2

Tabulka 1 Funkční požadavky

ID	Funkce	Priorita
N1	Manažer snů bude naprogramován v programovacím jazyce Kotlin.	1
N2	Manažer snů bude fungovat na zařízeních s OS Android 4.4 a vyšší.	1
N3	Manažer snů bude dodržovat pokyny Material Design.	2
N4	Manažer snů bude vyvinut v softwarové architektuře MVVM.	1

Tabulka 2 Nefunkční požadavky

4.4.2 Diagram užití

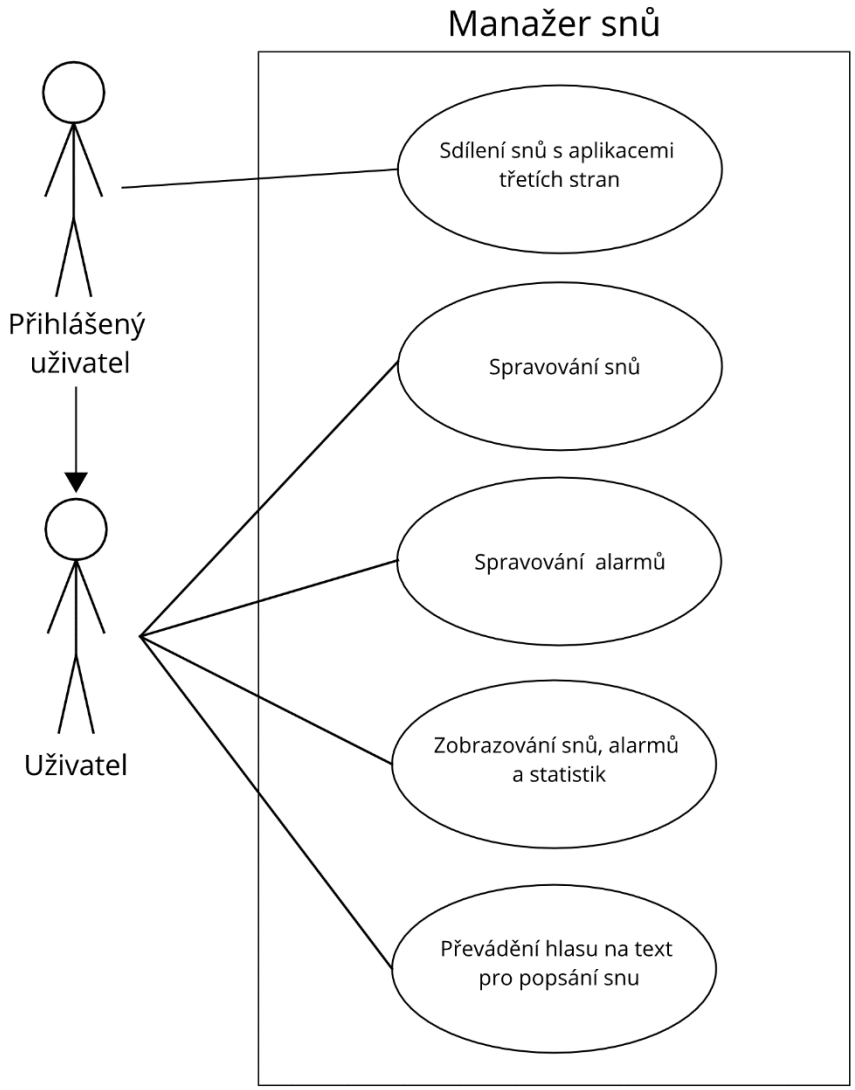
Diagram užití neboli Use Case diagram, je používán pro zachycení chování systému z hlediska uživatele, který systém používá. Definuje, jací uživatelé se systémem pracují, a které funkce na systému vykonávají. Use Case diagram znázorňuje funkční požadavky systému, tím pádem zobrazuje interakci mezi uživatelem a systémem samotným. Modelování diagramu lze provádět v následujících krocích:

- vyhledávání hranic systému,
- nacházení aktérů, jež interagují se systémem,
- nalezení případů užití,
- upřesnění případů užití,
- definování alternativní scénářů,
- opětovné provádění těchto kroků, dokud se neustálí případy užití, hranice systému a typy aktérů. [42]

V rámci diagramu užití se používají následující prvky:

- Aktér – slouží k popisu uživatele uvnitř systému. Uživatel nemusí být vždy fyzická osoba, ale může se jednat o hardwarové zařízení či další systém. Aktéry lze dále rozlišit na primární či pomocné. Ve verzi 2.0 jazyka UML jsou aktéři, jež znázorňují fyzickou osobu, označovány pomocí panáčka a pro systémy jsou vyhrazeny obdélníky.
- Případ užití – *„specifikuje část funkcionality systému, kterou využívá aktér a která plní určitý cíl.“*. Případ užití je popsán pomocí množiny scénářů, které jsou prováděny se stejným záměrem.
- Komunikační asociace (relace) – označuje se pomocí čáry, jež je zakončena šipkou a slouží k zobrazení toku informací mezi ostatními prvky
- Hranice systému – je znázorněna rámečkem okolo případů užití a je označena názvem systému. [42]

Na následujícím obrázku je zobrazen Use Case Diagram aplikace Manažer snů.



Obrázek 19 Diagram případů užití

Z diagramu případů užití je vidět, že se systémem pracují dva druhy aktérů. Jedná se o přihlášeného uživatele a uživatele bez přihlášení. Přihlášený uživatel má tu výhodu, že může své sny sdílet s aplikacemi třetích stran. V rámci diagramu užití, jež je vidět na obrázku, jsou shrnuty všechny funkční požadavky, které jsou popsané v podkapitole funkčních požadavků.

4.4.3 Use Case specifikace

Diagram užití se používá pro zobrazení funkcionalit systému a zobrazuje, kdo dané funkcionality spravuje, avšak může se stát, že samotný diagram se může jevit málo samo dokumentující. Pro tento účel se ve formě dokumentu přikládá k Use Case diagramu, takzvaná Use Case specifikace, která může mít formu tabulky nebo prostého textu. Tato specifikace popisuje jednotlivé případy užití, kde každému definuje několik následujících bodů:

- Krátký popis – pomocí jedné či dvou vět se zde daný případ užití popisuje. Popis by měl vysvětlovat, co daná funkce pro uživatele přináší a proč funkcionalitu spouští.
- Aktéři – definuje aktéry, kteří se účastní daného případu užití.
- Podmínky pro spuštění – zde se uvádějí podmínky, které je nutné splnit pro spuštění případu užití. Příkladem může být nainstalovaná aplikace nebo internetové připojení.
- Základní tok – v této části jsou v jednotlivých očíslovaných bodech popsány interakce mezi aktéry a případy užití. Zápis toku neboli scénáře je brán z pohledu uživatele.
- Alternativní toky – zde je prostor pro reagování na možné odchylky od hlavního scénáře. Většinou se jedná o chyby ze strany uživatele nebo systému, například chybné heslo.
- Podmínky dokončení – příkladem může být, že vše proběhlo v pořádku. [43]

Následná Use Case specifikace dokumentuje případ užití, při kterém přihlášený uživatel sdílí sen s aplikací třetí strany.

Případ užití	UC1 – Sdílení snů s aplikacemi třetích stran.
Krátký popis	Use case umožňuje sdílet sny do aplikací třetích stran.
Aktéři	Přihlášený uživatel
Podmínky pro spuštění	Připojení k internetu
Základní tok	1. Uživatel zobrazí sen. 2. Vybere aplikaci, do které chce sen sdílet. 3. Systém odešle sen do vybrané aplikace.
Alternativní tok	Vypadne-li internet, zobrazí se uživateli chybová hláška.
Podmínky pro dokončení	Sen bude uložen do aplikace třetí strany.

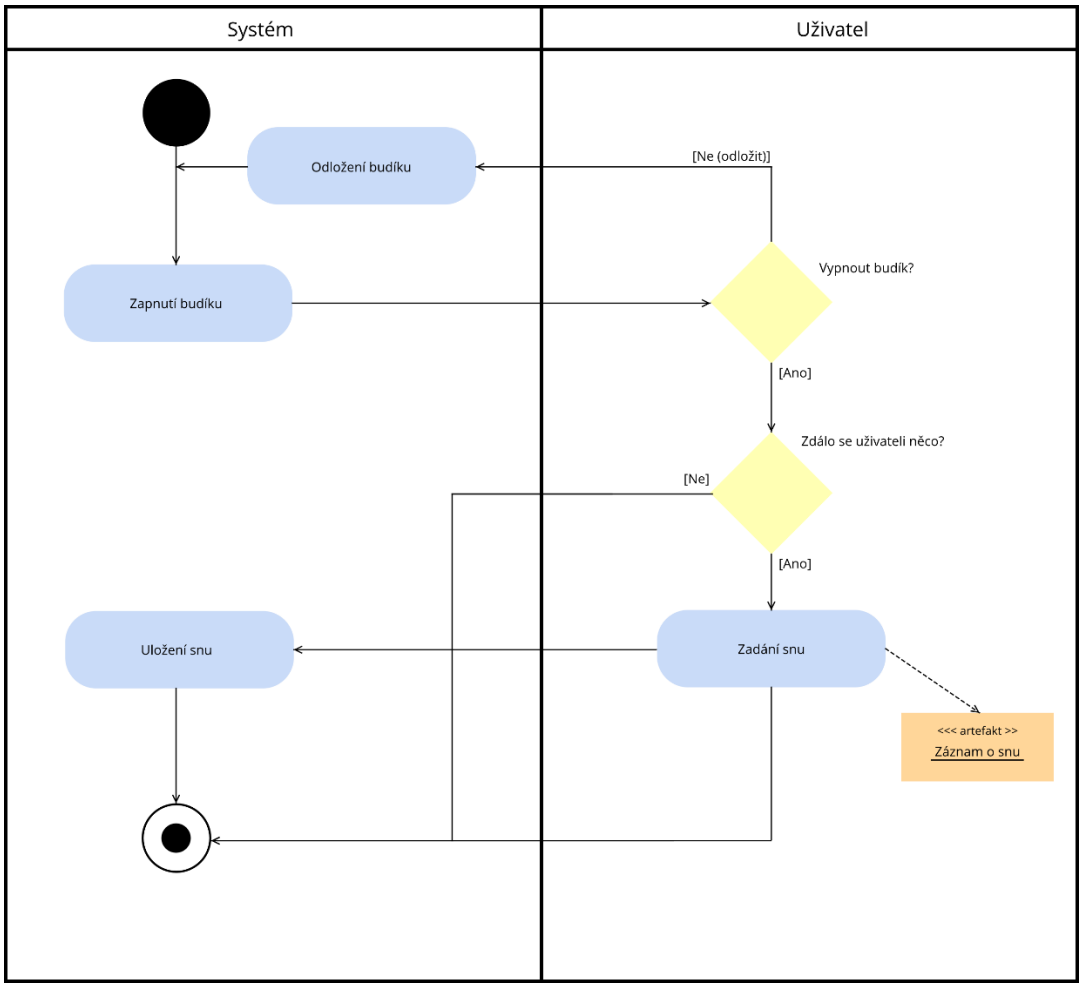
Tabulka 3 Use Case specifikace sdílení snu

4.4.4 Diagram aktivit

Dříve byl diagram aktivit brán jako speciální případ stavového diagramu, ale ve verzi UML 2.0 má tento diagram vlastní kategorii a nově je založen na sémantice Petriho sítí. Diagram aktivit je druh UML diagramu, který zobrazuje interakce, jež popisují procedurální logiku, byznys logiku nebo jednotlivé pracovní postupy. Lze také pomocí něj graficky znázornit případy užití, jako posloupnost jednotlivých akcí. V rámci diagramu aktivit se vyskytují následující prvky:

- Akční uzly – představují nedělitelné jednotky v rámci aktivity.
- Řídicí uzly – řídí tok uvnitř aktivity.
- Objektové uzly – zastupují objekty.
- Uzel rozhodnutí – má několik výstupních hran a vždy jednu hranu vstupující. Výstup je zvolen na základě splnění podmínky.
- Uzel sloučení – oproti uzlu rozhodnutí má jednu výstupní hranu a několik hran vstupujících. Používá se především ke sloučení hran, které byly rozděleny uzlem rozhodnutí.
- Uzel rozvětvení a spojení – jsou používány, pokud je nutné, aby větve diagramu probíhaly paralelně.

Celý diagram lze ještě rozdělit do takzvaných plaveckých drah (zón zodpovědností). Každá dráha může zastupovat například roli v rámci systému. Na následujícím obrázku je zobrazen diagram aktivit pro proces zadání snů z alarmu v aplikaci Manažer snů. [44]



Obrázek 20 Diagram aktivit po probuzení uživatele alarmem

Pro diagram aktivit byla zvolena aktivita, která by měla být nejčastějším případem vytvoření snu. Na obrázku diagramu aktivit je zobrazen proces, při kterém uživatel zaznamenává sen po probuzení alarmem. Celý proces začíná zazvoněním budíku, kde je následně uživatel vyzván, jestli chce budík odložit anebo ho ukončit. Po ukončení budíku je uživatel dotázán, zda měl nějaký sen. V případě, že uživatel zvolí volbu nezadávání snu, aplikace se ukončí. Pokud uživatel odpověděl na dotaz kladně, chce zaznamenat sen, tak následuje vytvoření snu a v rámci diagramu aktivit je vytvořen nový artefakt, který poté systém uloží do databáze a tím celý proces končí.

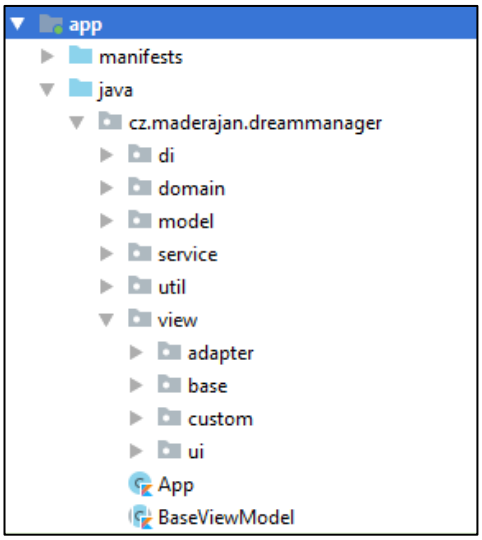
4.5 Základní struktura aplikace

V této podkapitole bude popsána základní strukturu aplikace, avšak popis nebude zacházet do detailů jednotlivých tříd, spíše bude vysvětlena obecná funkcionalita jednotlivých komponent.

Každá Android aplikace se skládá ze základních komponent, které jsou rozděleny do základních vrstev:

- manifest – obsahuje soubor *AndroidManifest.xml*. Soubor popisuje důležité informace o aplikaci, jako například id aplikace, název balíčku aplikace, ikonu aplikace, různá povolení aplikace a další. Také definuje různé komponenty v rámci celé aplikace, jako Activity, Service, BroadcastReceiver a ContentProvider. [45]
- java – obsahuje veškeré zdrojové kódy aplikace. Kódy mohou být v této sekci jak v programovacím jazyce Java, tak i Kotlin.
- res – obsahuje veškeré „nekódové“ zdroje. Jako XML layouty, textové řetězce aplikace, obrázky, animace, hodnoty pro dimenze, barvy a další.
- Gradle – pro sestavení souboru APK ze všech zdrojových souborů, používá Android Studio Gradle, což je nástroj pro sestavování, automatizování a spravování sestavovacího procesu. [46]

Následující obrázek zobrazuje balíčkovou strukturu aplikace Manažer snů.



Obrázek 21 Struktura balíčků aplikace Manažer snů

Ze struktury na obrázku je vidět, že projekt má šest hlavních balíčků, jimiž jsou:

- *di* – zde se vyskytují pouze soubory, které spravují Dependency Injection v rámci celé aplikace, za pomocí knihovny Koin, jež je popsaná v kapitole Použité technologie.
- *domain* – v tomto balíčku se vyskytují třídy, které jsou nutné pro komunikaci s aplikacemi třetích stran.
- *model* – uvnitř toho balíčku jsou pouze třídy, které slouží pro správu databáze. Jedná se většinou o třídy, jakou jsou Entity, Dao či repositář.
- *service* – v rámci balíčku service jsou třídy, které pracují na pozadí aplikace. Jedná se o třídy, jež se starají především o funkčnost alarmu.
- *util* – zde se vyskytují různá rozšíření a pomocné třídy.
- *view* – v balíčku view se vyskytují další čtyři balíčky (adapter, base, custom, ui). Hlavní je zde především balíček ui, který obsahuje všechny aktivity, fragmenty a komponenty typu ViewModel.

4.6 Datová vrstva

V úvodu této kapitoly je krátce představena databáze SQLite, která je v aplikaci používána. Následně je uvedena datová struktura aplikace.

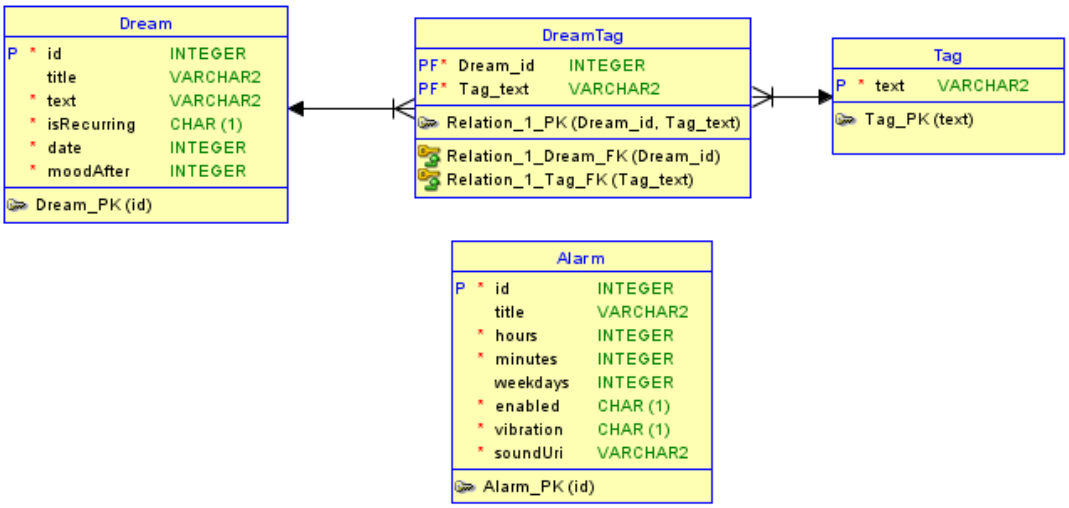
4.6.1 SQLite

SQLite je jednoduchá, malá databáze, která se ve většině případů používá jako primární databáze pro mobilní aplikace s operačním systémem Android. Databáze je reprezentovaná jako jeden soubor, většinou s příponou *.db,* který je jednoduše přenositelný a je tedy možné ho přenášet mezi zařízeními pouhým nakopírováním. Neprovádí se tedy žádná instalace služeb, konfigurování či nastavování přihlašovacích údajů. Databáze je tím pádem velmi odlehčená a rychlá, což nese i určité nevýhody, jako například neposkytování uživatelských oprávnění.

SQLite je relační netypová databáze. Z uvedených vlastností je zřejmé, že SQLite databáze neposkytuje velkou míru bezpečnosti a je vhodná spíše pro menší projekty. I přesto tuto databázi využívají často webové prohlížeče, Skype nebo i mobilní zařízení Android a iOS. V rámci aplikace Manažer snů se s databází SQLite pracuje pomocí ORM Room, které je detailně popsané v kapitole Android Achitecture Components. [47]

4.6.2 Datová struktura

Datový model aplikace je poměrně jednoduchý, jelikož obsahuje pouze čtyři tabulky. Jejich struktura a relace jsou vidět na následujícím ER diagramu.



Obrázek 22 ER diagram datové struky aplikace

Veškeré tabulky, atributy a jejich relace byli vygenerovány pomocí ORM Room. Z diagramu je vidět, že mezi tabulkou *Dream* a *Tag* je relace M:N. Alarm – tabulka sloužící pro uložení alarmu. Mezi hlavní atributy patří *hours*, *minutes* a *weekdays*. Atribut *weekdays* je v tabulce uložený jako *Integer*, se kterým se pracuje pomocí bitových operací. Účel tabulek a jejich atributů je následující:

- Dream – reprezentuje model samotného snu. Hlavní atributy této tabulky jsou *id*, *text*, *date*. Atribut *date* je reprezentovaný pomocí časového razítka.
- Tag – obsahuje pouze jediný atribut a to *text*, což je zároveň i primární klíč. Tato tabulka je vytvořena pro účel označení snů značkami. Tyto značky jsou využívány pro zachycení symbolů ve snech a jsou pomocí nich vytvořeny i statistiky.
- DreamTag – vazební tabulka mezi tabulkami *Dream* a *Tag*.

4.7 Použité nástroje a technologie

V textu této kapitoly budou uvedeny nástroje a technologie, které byly použity při tvorbě aplikace Manažer snů. U některých technologií bude ukázána i konkrétní implementace.

4.7.1 Koin

Koin je Android framework pro Dependency Injection. Jedná se o jednoduchý, odlehčený framework, který je napsán v programovacím jazyce Kotlin. V rámci světa Android se vyskytuje více frameworků, jež mají pomoci s implementací Dependency Injection, jako například Dagger 2, avšak ten je o mnoho složitější než Koin a na menších projektech ho nelze využít v plné míře.

Dependency Injection je technika, která umožňuje poskytovat závislosti, které objekt potřebuje, aniž by byly vytvořeny v rámci samotného objektu. Příkladem může být, že v rámci aplikace se vyskytuje repositář, který obsahuje instanci databáze, jež je vytvořená současně s vytvořením repositáře. V tomto případě nelze repositář efektivně testovat unit testy. Zde je více než vhodné použít Dependency Injection a instanci databáze předat rovnou v konstruktoru třídy, a tím pádem při testování, lze instanci databáze „*namockovat*“. DI je navíc doporučováno používat v rámci softwarové architektury MVVM, jelikož ViewModel se do aktivity či fragmentu pouze předá pomocí DI a nemusí se vytvoření řešit pomocí klasické inicializace v těle aktivity či fragmentu. V aplikaci Manažer snů je Dependency Injection používána právě za tímto účelem. V rámci knihovny Koin se vytváří takzvané moduly, které definují závislosti. Použití lze vidět na následující ukázce kódu. [48]

```
val appModule = module {
    single(createOnStart = true) { PrefManager(androidApplication()) }
    single(createOnStart = true) { JobPlanner() }
    single { NotificationHelper() }
    single { MediaPlayer() }

    viewModel { AlarmListViewModel(get()) }
    viewModel { AddDreamViewModel(get()) }
    viewModel { DreamListViewModel(get()) }
    viewModel { StatisticsViewModel(get()) }
}
```

Zdrojový kód 22 Koin module

Ve spodní části ukázky je vidět definice DI pro ViewModel. Na následující ukázce lze vidět už konkrétní použití v rámci fragmentu.

```
private val viewModel: AddDreamViewModel by sharedViewModel()
```

Zdrojový kód 23 Inicializace ViewModel pomocí Koin

4.7.2 Kotlin coroutines

Kotlin coroutines je nástroj v programovacím jazyku Kotlin, který složí pro asynchronní a neblokující programování. Coroutines byly přidány do Kotlinu ve verzi 1.3, avšak celý koncept

coroutines je mnohem starší a poprvé se objevil v roce 1967. V posledních letech se konceptu coroutines dostalo popularity a tento koncept začaly používat například jazyky jako Javascript, C#, Python, Ruby a Go. V rámci platformy Android řeší coroutines problémy dlouho běžících procesů a pozastavování funkcí na hlavním vlákně. V OS Android je za veškeré UI změny zodpovědné hlavní vlákno, a pokud se spustí funkce *X*, která může trvat několik sekund a spustí se z hlavního vlákna, UI aplikace nebude reagovat. Takže je nutné, aby funkce *X* použila jiné vlákno než hlavní a k tomu právě slouží Kotlin coroutines. V aplikaci Manažer snů byla technika coroutines použita především pro dotazy do databáze, jelikož použité ORM Room implicitně nedovoluje provádět dotazy na hlavním vlákně. Dále se tato technologie použila pro posílání dotazů na API aplikací třetích stran. Příklad použití Kotlin coroutines je vidět na následující ukázce kódu. [49]

```
fun getAllDreamsWithTags(): List<DreamTagsDTO> {
    var dream: List<DreamTagsDTO> = arrayListOf()

    runBlocking(ioScope) {
        dream = withContext(ioScope) {
            db.dreamDao().getAllDreamsWithTags()
        }
    }

    return dream
}
```

Zdrojový kód 24 Coroutines neblokované asynchornní volání

V ukázce je vidět příklad neblokovaného volání, které je zaručeno funkcí runBlocking, jemuž je předán *Scope*, ve kterém má pracovat.

4.7.3 Android-Job

V operačním systému Android jsou nyní tři různé API, pomocí kterých lze spustit úkol v budoucnosti. Každá z těchto API má své výhody a nevýhody:

- AlarmManager – se časem mění a je nutné dávat pozor, jaké metody a v jaké verzi operačního systému jsou dané metody používány.
- JobScheduler – je dostupný pouze na operačních systémech Android Lolipop a vyšších.
- GcmNetworkManager – je použitelný pouze v zařízeních, kde je předinstalovaná aplikace Google Play.

Je tedy velmi těžké vědět, jakou API je vhodné použít při řešení určitého problému. Také narůstá složitost kódu, jelikož určité případy musí být ošetřeny tak, aby výsledné chování vyhovovalo požadavkům. Z těchto důvodů společnost Evernote vyvinula knihovnu Android-Job, která řeší zmíněné problémy automaticky a sama se rozhoduje, kterou knihovnu použít, aby bylo dosaženo správné funkčnosti a co nejvyšší efektivity. Knihovna Android-Job v sobě obsahuje již knihovny AlarmManager, JobScheduler a GcmNetworkManger. Práce s touto knihovnou je relativně jednoduchá, jelikož je vždy nutné inicializovat JobManager, a poté stačí spustit daný úkol. Zároveň není nutné definovat žádné service, receiver nebo povolení v AndroidManifest.xml. Aplikace Manažer snů používá tuto knihovnu pro spouštění alarmů v zadaný čas a den. [50]

4.7.4 OAuth 2.0

OAuth 2.0 je poměrně nový autorizační protokol, jež je v dnešní době spíše standard pro zabezpečení komunikace REST služeb. Jeho největší předností je to, že uživatel může poskytnout klientské aplikaci přístup datům, bez toho, aby poskytnul své přístupové údaje a tím ve své podstatě povolil neomezený přístup k jeho účtu. Další výhodou je, že dovoluje vymezit kompetence jednotlivým aplikacím (scopes). Tím umožňuje podrobně sledovat využití poskytnutých privilegií.

V rámci autorizačního protokolu OAuth 2.0 lze autentizovat klientskou aplikaci dvěma způsoby. První způsob je, že klientská aplikace se autentizuje sama za sebe. Druhý způsob je přes uživatele, který tomu musí udělit oprávnění. Díky těmto vlastnostem lze přistupovat k datům uživatele, aniž by uživatel podstupoval jakékoliv rizika z hlediska porušení ochrany osobních údajů. Protokol OAuth 2.0 je v aplikaci Manažer snů využíván pro přístup k citlivým informacím z aplikací třetích stran (Google Calendar, Google Tasks). [51]

4.8 Aplikační rozhraní aplikací třetích stran

V této kapitole budou popsány aplikace třetích stran, s nimiž aplikace Manažer snů komunikuje. Konkrétně se jedná o aplikace Google Calendar, Google Task a API pro převod zvuku na text. Do aplikace Manažer snů jsou zahrnuty aplikace Google Calendar a Google Tasks, zejména z toho důvodu, že lidem se často zdají sny, jež by mohli často vést k nějakému druhu nápadu. Využilo se tedy toho, že tyto dvě aplikace disponují otevřeným rozhraním, se kterým přihlášený uživatel může komunikovat. Aby uživatel mohl komunikovat s aplikacemi od firmy

Google je nutné, aby do aplikace byl přihlášen a také, aby aplikace měla povolený přístup v Google Console. Pokud je uživatel přihlášen, tak se do aplikace Manažer snů ukládá objekt *Profile*, který v sobě obsahuje údaje o přihlášeném uživateli a token, který je se používá pro komunikaci s protokolem OAuth 2.0.

Jedním z kritérií této diplomové práce bylo, že aplikace bude schopna převádět nahraný zvuk na text. Tento druh zadávání byl do aplikace implementovaný z důvodu, aby uživatel mohl pořídit záznam snu co nejrychleji. Konkrétní popis této API a implementace bude rozebrán v závěru této kapitoly.

4.8.1 Google Calendar

Google Calendar je aplikace vyvinutá firmou Google, která slouží pro spravování kalendáře. Google Calendar se vyskytuje napříč skoro všemi platformami, například pro zařízení s OS Android či iOS, ve webovém prohlížeči nebo je dokonce možné mít klienty pro desktop. U zařízení s operačním systémem Android je Google Calendar předinstalovaná aplikace. Podmínka pro používání Google Calendar je mít založený email Google, což je nutností pro používání obchodu Google Play, tím pádem není nutné se do aplikace registrovat. Jak již bylo avizováno, Google Calendar disponuje veřejným API, které bylo využito pro komunikaci s aplikací Manažer snů.

V současnosti existují tři způsoby, jakými lze komunikovat s kalendářem v zařízeních s OS Android. Jedním ze způsobu je využití otevřené REST API, které Google Calendar nabízí. Dále lze komunikovat pomocí repositáře Calendar provider, což je ve své podstatě Content provider, jež se v Androidu používá pro správu dat v aplikacích. Ve své podstatě, při používání Calendar provider, se zapisují data přímo do tabulky, která drží data o uživatelském kalendáři. Pro tento přístup je nutné, aby uživatel udělil povolení přístupu pro zápis a čtení kalendáře. Třetí způsob komunikace s Google Calendar API je využití Calendar SDK, které obstarává HTTP komunikaci s API. Také poskytuje objekty, jako například Event, EventRemider a další, které se následně na API odesílají. Tím pádem je zajištěno, aby objekty, které budou takto poslány, budou vždy ve validním formátu. [52]

Následující ukázka kódu zobrazuje metodu, která slouží pro vytvoření objektu Event, sloužícímu jako událost v kalendáři.

```
private fun createCalendarEvent(dream: Dream) : Event{
    val eventDate = EventDateTime().apply {
        dateTime = DateTime(dream.date)
        timeZone = TimeZone.getDefault().id
    }

    val eventReminder = Event.Reminders().apply {
        useDefault = false
    }

    return Event().apply {
        summary = dream.title
        description = dream.text
        start = eventDate
        end = eventDate
        reminders = eventReminder
    }
}
```

Zdrojový kód 25 Calendar SDK - vytvoření události

Z předešlé ukázky je vidět, že pro vytvoření objektu stačí do metody *createCalendarEvent(Dream)*, předat objekt snu a na základě něho se vytvoří Event, jež slouží jako návratový typ metody. Na následujícím kódu je ukázáno použití této metody.

```
fun postCalendarEvent(calendarService: Calendar, dream: Dream,
                      exceptionHandler: CoroutineExceptionHandler) {
    GlobalScope.launch(exceptionHandler)   {
        val event = calendarService.events().
            insert(PRIMARY_CALENDAR,
            createCalendarEvent(dream))

        event.execute()
    }
}
```

Zdrojový kód 26 Calendar SDK - odeslání události na API

Z úryvku kódu je vidět, že do metoda *postCalendarEvent(Calendar, Dream, CortineExceptionHandler)*, využívá metodu z předešlé ukázky Zdrojový kód 27, kde se vytváří objekt Event, který je následně asynchronně odeslán. V případě nastane-li nějaká chyba při volání, objekt expcetionHandler tuto chybu zachytí v místě, kde byl handler definován.

4.8.2 Google Tasks

Google Task je aplikace, jež je vyvinuta firmou Google. Jedná se o aplikaci, která funguje na principu seznamu úkolů. Nabízí velmi dobrou synchronizaci s aplikacemi Gmail a Google Calendar. Aplikace Google Tasks, je stejně jako Google Calendar, dostupná napříč všemi různými platformami a je zde také potřeba mít založený email Google. Pro komunikaci s API je možné využít buď otevřené REST rozhraní, anebo je možné použít Google Tasks SDK, které bylo využito v aplikaci Manažer snů a funguje na podobné bázi jako SDK pro Google Calendar. SDK pro úkoly Google nabízí mnoho hotových objektů, a tím pádem není nutné studovat, jak daná API vypadá. SDK také zprostředkovává komunikaci mezi zařízením a serverem. Následující ukázka kódu ukazuje, jak vypadá vytvoření objektu Task. [53]

```
private fun createTask(dream: Dream): Task {
    return Task().apply {
        title = if (dream.title.isEmpty()) {
            dream.text
        } else {
            dream.title
        }

        if (!title.isEmpty()) notes = dream.text

        due = DateTime(dream.date)
    }
}
```

Zdrojový kód 27 Google Tasks API - vytvoření úkolu

Metody *createTask(Dream)* mají parametr *dream*, na jehož základě se vytváří objekt typu Task. Tato metoda je využívána v metodě *postTaskToApi*, která je vidět na následující ukázce.

```
fun postTaskToApi(tasksService: Tasks, taskTitle: String, dream: Dream,
exceptionHandler: CoroutineExceptionHandler) {
    GlobalScope.launch(exceptionHandler) {
        var id: String? = null
        val allTasks = tasksService.tasklists().list().execute()
        for (taskList in allTasks.items) {
            if (taskList.title == taskTitle) {
                id = taskList.id
                break
            }
        }

        if (id.isNullOrEmpty()) {
            val tasksList = tasksService.tasklists().insert(
                TaskList().setTitle(taskTitle)
            ).execute()

            id = tasksList.id
        }

        val task = createTask(dream)
        tasksService.tasks().insert(id, task).execute()
    }
}
```

Zdrojový kód 28 Google Tasks SDK - odeslání úkolu na API

Metoda z ukázky je o něco komplikovanější, než v případě posílání události do kalendáře, jelikož je nutné nejdříve zjistit identifikátor seznamu úkolů, kam se bude vytvořený úkol ukládat. K čemuž je použit dotaz, který si vyžádá všechny seznamy úkolů přihlášeného uživatele. Pokud se v seznamech nevyskytuje seznam úkolů s názvem „Sny“ (bráno dle jazyka nastaveného v telefonu), tak se vytvoří nový seznam a uloží se na API. Ze seznamu úkolů je důležitý atribut *id*, který se používá pro vložení nového úkolu do seznamu. Stejně jako v případě kalendáře, probíhá celá tato metoda asynchronně a v případě, že se vyskytne jakákoliv chyba, je zavolán *exceptionHandler*, jež se o chybu postará v místě své definice.

4.8.3 Převod zvuku na text

Pro převod zvuku na text byla za účelem jednoduchosti a správné integrace použita knihovna DroidSpeech, která využívá výchozí knihovnu z Android na rozpoznávání řeči. Jeden z hlavních důvodů, proč byla tato knihovna implementována je, že výchozí knihovna neposkytuje kontinuální rozhraní pro nahrávání zvuku a je nutné nahrávání znovu zapínat. Knihovna DroidSpeech tedy poskytuje rozhraní, které optimalizuje toto nahrávání a poskytuje co možná nejvíce spolehlivé nahrávání bez přestávky. Pro použití knihovny stačí založit instanci objektu DroidSpeech, kde se v parametrech konstruktoru předá pouze kontext. Poté stačí nadefinovat listener. Použití knihovny je vidět na následující ukázce kódu.

```
droidSpeech = DroidSpeech(context, null)
droidSpeech.setShowRecognitionProgressView(false)
droidSpeech.setOnDroidSpeechListener(this)
.
.
override fun onDroidSpeechLiveResult(liveSpeechResult: String?) {
    val text = "$speechResult $liveSpeechResult"
    dream_text_edit_text.setText(text)
    dream_text_edit_text.setSelection(text.length)

    Log.d("DroidSpeech", "$speechResult $liveSpeechResult")
}
```

Zdrojový kód 29 Použití knihovny DroidSpeech

Z ukázky je vidět pouze část implementace, kde se na začátku provádí inicializace objektu DroidSpeech a následně je pod touto sekcí vidět jedna z metod rozhraní *OnDSListener onDroidSpeechLiveResult(String?)*, která se využívá pro zpracování aktuálního výstupu.

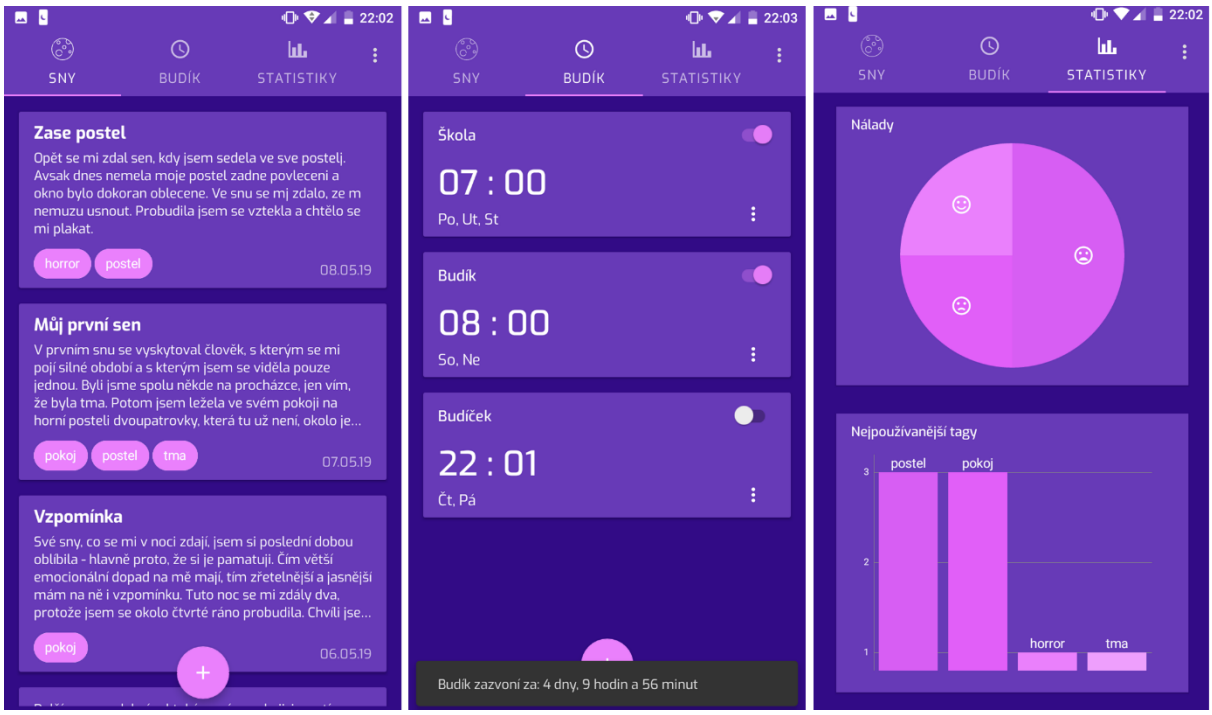
V aplikaci Manažer snů je využito nahrávání hlasu na obrazovce, kde uživatel zadává popis snu. Po stisknutí tlačítka může uživatel zaznamenávat sen, aniž by nahrávání musel znovu zapínat. Jediné, na co je uživatel vyzván je, že pokud nahrává hlas poprvé, je nutné udělit aplikaci povolení. Po zaznamenávání určité části je tato část pomocí knihovny odeslána na sever, který

rozpozná hlas a navrátí text. Tímto způsobem je zaručeno, že uživatel vidí aktuálně namluvenou zprávu. Díky Google Speech Recognition API, je možné nahrávat ve více jazycích, a také je možnost zadávat příkazy jako tečku či otazník.

4.9 Ukázka aplikace

V této podkapitole budou ukázány a popsány jednotlivé obrazovky aplikace.

Pro navržení grafického prototypu byl použit grafický nástroj Gravit. Pro aplikaci Manažer snů byly zvoleny převážně tmavé barvy a celá aplikace je laděna do odstínů růžové a fialové. Většina komponent, jež byla v aplikaci použita, dodržuje předepsané principy Material Design. Na následujícím obrázku jsou zobrazeny tři snímky uvítací obrazovky.

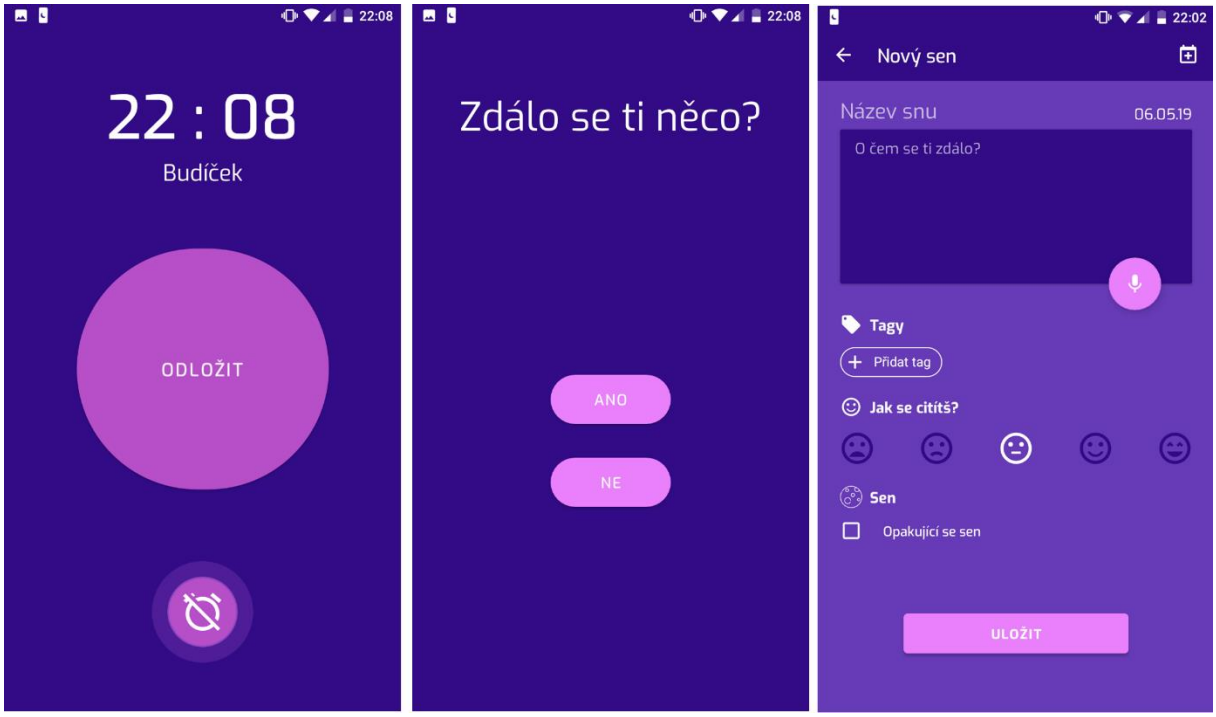


Obrázek 23 První obrazovka aplikace

První obrazovka aplikace se skládá ze tří pod-obrazovek, které jsou vidět na obrázku. Jedná se o obrazovky sny, budíky a statistiky. V obrazovce s listem snů, jež je vidět na snímku vlevo, se vyskytují sny chronologicky seřazené od nejnovějšího snu po nejstarší. U jednotlivých snů je vidět nadpis, text, který je omezen na pět řádků, datum a značky snů. Jednotlivé sny jsou umístěny v kartičkách a po kliknutí na zvolený sen, se uživatel dostává na detail. Sny lze přidávat do aplikace pomocí tlačítka s ikonou plus, vyskytující se ve spodní části obrazovky, anebo pomocí rozcestníku po zazvonění alarmu, jež bude představen dále v této kapitole. Na snímku uprostřed je vidět seznamu budíků, kde jsou jednotlivé budíky také rozmístěny do kartiček.

Každý budík v seznamu disponuje nejdůležitějšími informacemi, jako čas zazvonění či dny v týdnu, kdy je budík aktivní. Budíky lze z této obrazovky zapínat/vypínat pomocí komponenty Switch, která je umístěna v pravé horní části položky budíku. Snímek obrazovky umístěn v pravé části obrazovky ukazuje, jak vypadají statistiky v aplikaci. V současné verzi aplikace se v této obrazovce vyskytují statistiky, jež zobrazují nálady po snu napříč všemi sny, které jsou v aplikaci zaznamenané. Dále také čtyři nejpoužívanější značky a čarový graf (není vidět na snímku zobrazen), který zobrazuje výskyt snů v aktuálním měsíci.

V rámci aplikace je implementována workflow, která byla vymodelována v kapitole diagram aktivit, jež by měla uživateli poskytovat efektivní způsob pro zaznamenání snů. Obrazovky této workflow v aplikace je vidět na následujících snímcích.

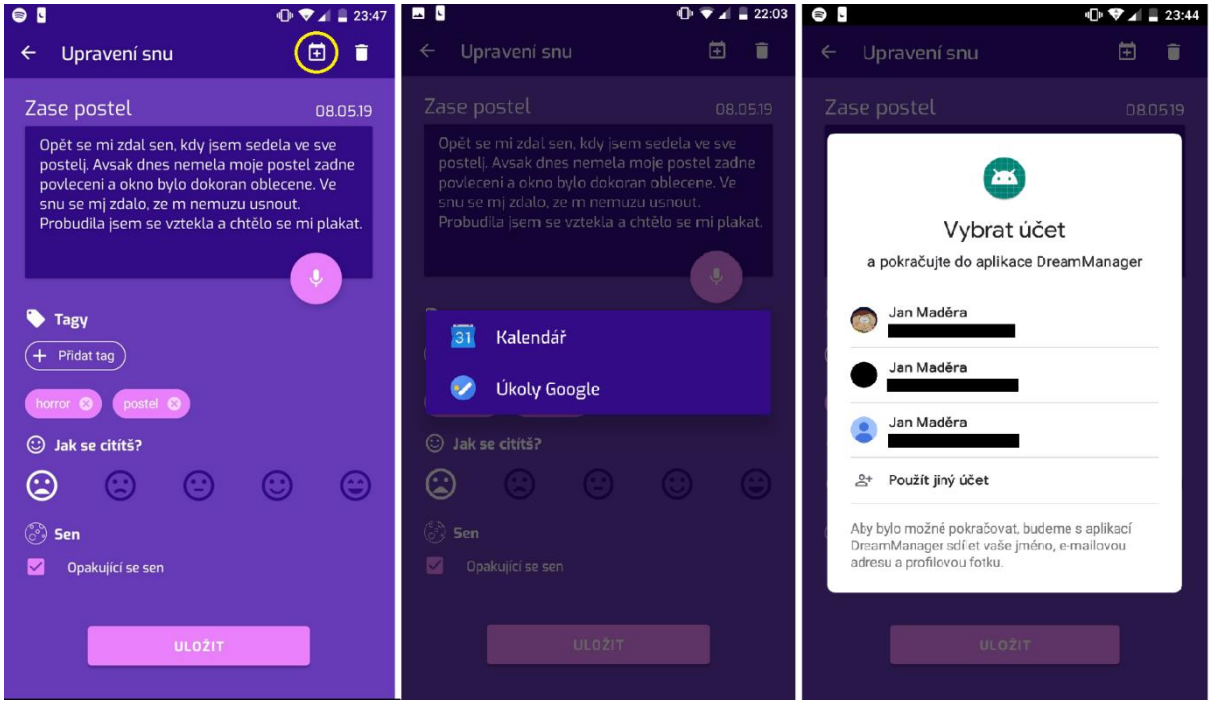


Obrázek 24 Obrazovky procesu sdílení snu

Obrazovky jsou uživateli postupně zobrazovány (čteno z levé strany). Po sepnutí alarmu je uživateli zobrazena obrazovka, na prvním snímku zleva, kde uživatel rozhoduje, zda budík odloží anebo jestli budík vypne. V případě, že je budík odložen, aplikace se vypíná a je spuštěna znovu za dobu odložení, kterou lze změnit v nastavení aplikace. Pokud uživatel budík vypne, je následně vyzván další obrazovkou. Uživatel je dotázán, zdali se mu zdál nějaký sen. Jestliže se uživateli nezdál žádný sen, aplikace se vypíná. V opačném případě je uživatel přesměrován na obrazovku, která je vidět na snímku v pravé části obrázku. Snímek v pravé části zobrazuje, jakým způsobem se v aplikaci Manažer snů vytváří záznamy o snech. Ze snímku je vidět, že záznamu o snu je možné vložit pomocí hlasu, kliknutím na tlačítko s ikonou mikrofonu.

Dále lze záznam snu označit pomocí značek, jež reprezentují symboly snu, které jsou potom použity v rámci obrazovky statistiky. Záznam je možné obohatit o náladu, titulek či záznam o tom, zda se sen opakoval. Je také možné upravit datum snu, které je vždy automaticky vyplněno podle dnu, ve kterém je sen zaznamenáván. Celý tento proces ukončí kliknutí na tlačítko uložit, jež aplikaci zavře a uživatel není nadále rušen.

Jednou z dalších vlastností aplikace je sdílení snů s aplikacemi třetích stran. Jak již bylo avizováno v předešlých kapitolách, sny lze odesílat do aplikací Google Calendar a Google Tasks. Proces sdílení je ukázán na následujícím obrázku.



Obrázek 25 Sdílení snu s aplikacemi třetích stran

Celý postup procesu sdílení je seřazen chronologicky (čteno z leva). Proces sdílení začíná ve chvíli, kdy uživatel klikne na ikonu, která je na snímku vlevo zakroužkována žlutou barvou. Po kliknutí na ikonu je uživateli zobrazen dialog, ve kterém si uživatel může vybrat, do které aplikace chce záznam o snu odeslat. V případě, že uživatel není přihlášen, zobrazí se mu obrazovka s účty, do kterých se může přihlásit. Po vybrání účtu se sen odešle do patřičné aplikace.

ZÁVĚR

Cílem této diplomové práce bylo vytvořit aplikaci pro operační systém Android, jež slouží pro zaznamenávání snů, a to i pomocí hlasu. Aplikace by měla sloužit pro uživatele, jež se například zabývají nebo pracující s analýzou snů. Pro splnění cílů byla provedena rešerše dostupných softwarových architektur a technologií, jež pomáhají tuto problematiku řešit. Tato práce byla vypracována dle přesně definovaného zadání a ve výsledku splňuje všechny požadavky, které byly na práci kladeny.

Aplikace *„Manažer snů“* je v současnosti publikována na Google Play a je ke stažení zcela zdarma. Přestože byly splněny všechny cíle, kterých bylo nutné dosáhnout v rámci zadání této diplomové práce, existuje mnoho funkcí, jež mohou tuto aplikaci vylepšit. Mezi nové funkce lze zařadit zamykání aplikace na heslo, jelikož informace vyskytující se uvnitř aplikace mohou být velmi citlivé.

Součástí aplikace jsou jednoduché statistiky, jež slouží pouze jako prostředek k zobrazení a není možné jen jakkoliv filtrovat, a tudíž výstup není dynamický a nedovoluje manipulaci s daty. Z čehož plyne, že by bylo velmi přínosné do aplikace zahrnout funkci, jež by tyto data filtrovala či v nich vyhledávala. Tato funkce by zjednodušila a zároveň zlepšila práci při analýze snů, jelikož by uživatel mohl sny filtrovat, například dle určitého období. Tato funkce vede k dalšímu vylepšení, které by spočívalo ve vytváření různých druhů reportů, které by mohly být následně sdíleny. Tento výstup by také mohl být používán jako materiál, jež by se mohl předkládat další osobě, která by se touto problematikou zaobírala.

Tato diplomová práce autorovi přinesla mnoho zkušeností týkajících se vývoje mobilních aplikací pro operační systém Android. Pomohla mu s rozšířením znalostí v oblasti softwarové architektury pro mobilní zařízení a naučila ho pracovat s knihovnami kolekce Android Architecture Components. Využití této aplikace autor vidí především v oboru psychologie při analýze snů.

POUŽITÁ LITERATURA

[1] *Percentage of all global web pages served to mobile phones from 2009 to 2018* [online]. 2019 [cit. 2019-05-01]. Dostupné z: https://www.statista.com/statistics/241462/global-mobile-phone-website-traffic-share/

[2] *Platform Architecture* [online]. [cit. 2019-04-11]. Dostupné z: https://developer.android.com/guide/platform

[3] ROUSE, Margaret. *Kotlin* [online]. [cit. 2019-04-11]. Dostupné z: https://whatis.techtarget.com/definition/Kotlin

[4] *What is the architecture of iOS* [online]. [cit. 2019-04-11]. Dostupné z: https://intellipaat.com/tutorial/ios-tutorial/ios-architecture/

[5] *About Objective-C* [online]. [cit. 2019-04-11]. Dostupné z: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html

[6] *About Swift* [online]. [cit. 2019-04-11]. Dostupné z: https://swift.org/about/

[7] *Your Guide to Cross-Platform Mobile App Development Tools* [online]. [cit. 2019-04-11]. Dostupné z: https://instabug.com/blog/cross-platform-development/

[8] GASBARRI, Gianfranco. *The truth about cross-platform mobile development* [online]. 26. 7. 2018 [cit. 2019-04-11]. Dostupné z: https://proandroiddev.com/the-truth-about-cross-platform-mobile-development-8b4f2de942e2

[9] *The Pros and Cons of Xamarin for Cross-Platform Development*[online]. 9. 10. 2018 [cit. 2019-04-11]. Dostupné z: https://hackernoon.com/the-pros-and-cons-of-xamarin-for-cross-platform-development-2a31c6610792

[10] *What is UX design? 15 user experience design experts weigh in* [online]. 27. 3. 2019 [cit. 2019-04-11]. Dostupné z: https://www.usertesting.com/blog/what-is-ux-design-15-user-experience-experts-weigh-in/

[11] SPENCER, William. *A Guide to Android Application Development Process*[online]. [cit. 2019-04-11]. Dostupné z: https://www.hokuapps.com/blogs/guide-android-application-development-process/

[12] *Introduction* [online]. [cit. 2019-04-11]. Dostupné z: https://material.io/design/introduction/#principles

[13] *Activity* [online]. [cit. 2019-04-18]. Dostupné z: https://developer.android.com/reference/android/app/Activity.html

[14] SMOLÍK, Tomáš. *Softwarová architektura: (Nezúžený "klasický" úvod)* [online]. [cit. 2019-05-01]. Dostupné z: https://cw.fel.cvut.cz/b181/_media/courses/a4m33sep/materialy/architecture_and_design/profinit_swarchitectureoverview.pdf

[15] RAWAT, Sourabh. *Implementing MVC pattern in Android with Kotlin* [online]. 4. 4. 2018 [cit. 2019-05-01]. Dostupné z: https://www.codementor.io/dragneelfps/implementing-mvc-pattern-in-android-with-kotlin-i9hi2r06c

[16] MUNTENESCU, Florina. *Android Architecture Patterns Part 1: Model-View-Controller* [online]. 6. 11. 2016 [cit. 2019-05-01]. Dostupné z: https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6

[17] KIM, Jinn. *Getting Started with MVP (Model View Presenter) on Android* [online]. 19. 12. 2018 [cit. 2019-05-01]. Dostupné z: https://www.raywenderlich.com/7026-getting-started-with-mvp-model-view-presenter-on-android

[18] ADEJUMO, Tobiloba. *ANDROID: MVP ARCHITECTURE 101* [online]. 1. 10. 2018 [cit. 2019-05-01]. Dostupné z: https://medium.com/mindorks/android-mvp-architecture-101-cf6eaa0908f3

[19] MUNTENESCU, Florina. *Android Architecture Patterns Part 3: Model-View-ViewModel* [online]. 4. 11. 2016 [cit. 2019-05-01]. Dostupné z: https://medium.com/upday-devs/android-architecture-patterns-part-3-model-view-viewmodel-e7eeee76b73b

[20] TAMADA, RAVI. *Android Introduction To Reactive Programming – RxJava, RxAndroid* [online]. [cit. 2019-05-01]. Dostupné z: https://www.androidhive.info/RxJava/android-getting-started-with-reactive-programming/

[21] CHUGH, ANUPAM. *Android MVVM Design Pattern* [online]. [cit. 2019-05-01]. Dostupné z: https://www.journaldev.com/20292/android-mvvm-design-pattern

[22] DORFMANN, Hannes. *MODEL-VIEW-INTENT ON ANDROID* [online]. [cit. 2019-05-01]. Dostupné z: http://hannesdorfmann.com/android/model-view-intent

[23] LEIVA, Antonio. *Clean architecture for Android with Kotlin: a pragmatic approach for starters* [online]. [cit. 2019-05-01]. Dostupné z: https://antonioleiva.com/clean-architecture-android/

[24] *Android Architecture Components* [online]. [cit. 2019-04-18]. Dostupné z: https://developer.android.com/topic/libraries/architecture/

[25] HOYOS, Mario. *What is an ORM and Why You Should Use it: An introduction to Object-Relational-Mappers* [online]. 24. 9. 2018 [cit. 2019-04-18]. Dostupné z: https://blog.bitsrc.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a

[26] *6 Best Android ORM Libraries to Use* [online]. 18. 8. 2018 [cit. 2019-04-18]. Dostupné z: https://www.tldevtech.com/best-android-orm-libraries-to-use/

[27] *Save data in a local database using Room* [online]. [cit. 2019-04-18]. Dostupné z: https://developer.android.com/training/data-storage/room/index.html

[28] SZKLARSKA, Paulina. *Android Architecture Components: Room—Introduction* [online]. 23. 10. 2017 [cit. 2019-04-18]. Dostupné z: https://android.jlelse.eu/android-architecture-components-room-introduction-4774dd72a1ae

[29] *TypeConverter* [online]. [cit. 2019-04-18]. Dostupné z: https://developer.android.com/reference/android/arch/persistence/room/TypeConverter

[30] *Observer Design Pattern* [online]. [cit. 2019-04-18]. Dostupné z: https://sourcemaking.com/design_patterns/observer

[31] *Architecture Components: LiveData* [online]. [cit. 2019-04-18]. Dostupné z: https://google-developer-training.github.io/android-developer-advanced-course-concepts/unit-6-working-with-architecture-components/lesson-14-architecture-components/14-1-c-architecture-components/14-1-c-architecture-components.html#livedata

[32] *Serializace a deserializace v C# .NET* [online]. [cit. 2019-04-18]. Dostupné z: https://www.itnetwork.cz/csharp/soubory/tutorial-csharp-serializace-a-deserializace

[33] *ViewModel Overview* [online]. [cit. 2019-04-18]. Dostupné z: https://developer.android.com/topic/libraries/architecture/viewmodel

[34] *Build a flexible UI* [online]. [cit. 2019-04-18]. Dostupné z: https://developer.android.com/training/basics/fragments/fragment-ui.html

[35] *Build a flexible UI* [online]. [cit. 2019-04-18]. Dostupné z: https://developer.android.com/topic/libraries/architecture/paging.html

[36] *Architecture Components: Paging library* [online]. [cit. 2019-04-18]. Dostupné z: https://google-developer-training.github.io/android-developer-advanced-course-concepts/unit-6-working-with-architecture-components/lesson-14-architecture-components/14-1-c-architecture-components/14-1-c-architecture-components.html#paginglibrary

[37] *Data Binding Library* [online]. [cit. 2019-04-18]. Dostupné z: https://developer.android.com/topic/libraries/data-binding

[38] *Guide to app architecture* [online]. [cit. 2019-04-18]. Dostupné z: https://developer.android.com/jetpack/docs/guide

[39] JUNG, Carl Gustav, Marie-Louise von FRANZ, Joseph L. HENDERSON, Jolande Székács JACOBI a Aniela JAFFÉ. *Člověk a jeho symboly*. Praha: Portál, 2017. ISBN 978-80-262-1259-1.

[40] ZENDULKA, Jaroslav. *Požadavky a jejich specifikace* [online]. [cit. 2019-05-07]. Dostupné z: http://www.fit.vutbr.cz/study/courses/PPS/public/pdf/5_2.pdf. VUT Brno.

[41] SZTURCOVÁ, Daniela. *Objektově orientované technologie: Požadavky na systém* [online]. [cit. 2019-05-07]. Dostupné z: http://gisak.vsb.cz/wikivyuka/images/9/9c/OotxPozadavky.pdf

[42] REJNKOVÁ, Petra. *Diagram případů užití*[online]. [cit. 2019-05-07]. Dostupné z: http://uml.czweb.org/pripad_uziti.htm

[43] ČÁPKA, David. *Lekce 3 - UML - Use Case Specifikace* [online]. [cit. 2019-05-09]. Dostupné z: https://www.itnetwork.cz/navrh/uml/uml-use-case-specifikace-diagram

[44] REJNKOVÁ, Petra. *Diagram aktivit* [online]. [cit. 2019-05-07]. Dostupné z: http://uml.czweb.org/diagram_aktivit.htm

[45] *App Manifest Overview* [online]. [cit. 2019-05-07]. Dostupné z: https://developer.android.com/guide/topics/manifest/manifest-intro.html

[46] *Configure your build* [online]. [cit. 2019-05-07]. Dostupné z: https://developer.android.com/studio/build

[47] MARTINEK, Michal. *Úvod do SQLite a příprava prostředí* [online]. [cit. 2019-05-07]. Dostupné z: https://www.itnetwork.cz/sqlite/sqlite-tutorial-uvod-a-priprava-prostredi

[48] ALBUQUERQUE, Levi. *Android Dependency Injection with Koin* [online]. 25. 2. 2019 [cit. 2019-05-07]. Dostupné z: https://dev.to/levimoreira/android-dependency-injection-with-koin-i34

[49] BÁRTA, Milan. *Kotlin Coroutines in Android*[online]. 16. 4. 2018 [cit. 2019-05-07]. Dostupné z: https://sourcediving.com/kotlin-coroutines-in-android-e2d5bb02c275

[50] WONDRATSCHEK, Ralf. *A unified job library for Android* [online]. 26. 10. 2015 [cit. 2019-05-07]. Dostupné z: https://evernote.com/blog/a-unified-job-library-for-android/

[51] JIRŮTKA, Jakub. *OAuth 2.0* [online]. 18. 1. 218 [cit. 2019-05-07]. Dostupné z: https://rozvoj.fit.cvut.cz/Main/oauth2

[52] *Calendar provider overview* [online]. [cit. 2019-05-07]. Dostupné z: https://developer.android.com/guide/topics/providers/calendar-provider

[53] *Google Tasks API: Java Quickstart* [online]. [cit. 2019-05-07]. Dostupné z: https://developers.google.com/tasks/quickstart/java