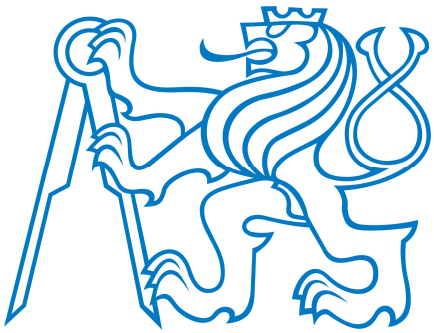
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta strojní – Ústav přístrojové a řídicí techniky



BAKALÁŘSKÁ PRÁCE

ANALÝZA VÝUKOVÉ APLIKACE VE FRAMEWORKU FLUTTER

ANALYSIS OF EDUCATIONAL APPLICATION IN FRAMEWORK FLUTTER

Pavel Šorf

2022

Prohlašuji, že jsem tuto práci vypracoval samostatně s použitím literárních zdrojů a informací, které cituji a uvádím v seznamu použité literatury a zdrojů.

Datum: . . . .

Podpis

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Šorf** Jméno: **Pavel** Osobní číslo: **491526**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav přístrojové a řídící techniky**

Studijní program: **Teoretický základ strojního inženýrství**

Studijní obor: **bez oboru**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Analýza výukové aplikace ve frameworku Flutter

Název bakalářské práce anglicky:

Analysis of educational application in framework Flutter

Pokyny pro vypracování:

1) Rešerše možností frameworku Flutter pro tvorbu multiplatformních aplikací
2) Analýza zdrojového kódu vzdělávací aplikace a návrh doporučení pro přechod na nejnovější verzi Flutteru
3) Studie generátorů matematických úloh a návrh jejich dopracování

Seznam doporučené literatury:

[1] BIESSEK, Alessandro a ProQuest Ebook Central (online SLUŽBA). Flutter for beginners: an introductory guide to building cross-platform mobile applications with Flutter and Dart 2: an introductory guide to building cross-platform mobile applications with Flutter and Dart 2. Birmingham;Mumbai: Packt, 2019. Dostupné také z: https://go.exlibris.link/BgQbwRjF
[2] PECINOVSKÝ, Rudolf. OOP: naučte se myslet a programovat objektově. Brno: Computer Press, 2010. ISBN 978-80-251-2126-9.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Matouš Cejnek, Ph.D. U12110.3

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **29.04.2022** Termín odevzdání bakalářské práce: **09.06.2022**

Platnost zadání bakalářské práce: _____

Ing. Matouš Cejnek, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	doc. Ing. Miroslav Španiel, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____
Datum převzetí zadání Podpis studenta

Úkoly

1. Rešerše možností frameworku Flutter pro tvorbu multiplatformních aplikací
2. Analýza zdrojového kódu vzdělávací aplikace a návrh doporučení pro přechod na nejnovější verzi Flutteru
3. Studie generátorů matematických úloh a návrh jejich dopracování

Tasks

1. Search for possibilities of framework Flutter for creating multiplatform applications
2. Analysis of educational application source code and suggestions for transition to the latest version of Flutter
3. Studies of mathematical task generator and suggestions for finishing them

Abstract

Tato práce se zabývá problematikou tvorby mobilních multiplatformních aplikací ve frameworku Flutter. Nejdříve se zabývá obecně návrhem aplikace ještě před započetím samotného programování. Následuje část věnující se problematice objektově orientovaného programování. Dále se zmiňuje o licencování softwaru a jeho testování. Nakonec se v teoretické části dostáváme ke struktuře aplikace ve frameworku Flutter a poslední zmínka je o správě verzí. V praktické části se pak práce zabývá doporučeními povýšení verze a analýzou generátorů úloh.

Abstract

This thesis describes the development of mobile multiplatform applications in framework Flutter. At the very beginning it describes proposal of structure of mobile application. Than there is a part about object oriented programming. Next the thesis describes software licences and testing applications. And to the end of theoretical part it describes the structure of application in framework Flutter and version management. The practical part of this thesis offers recommendation for the version update and analyses the task generator.

1 Klíčová slova

Flutter, Dart, multiplatformní aplikace, OOP, iOS, Android, Edukids.cz

2 Keywords

Flutter, Dart, multiplatform mobile applications, OOP, iOS, Android, Edukids.cz

Obsah

3 Úvod

3.1 Mobilní aplikace

Mobilní aplikace jsou softwarové aplikace vyvinuté přímo pro mobilní zařízení, jakými jsou chytré telefony nebo tablety. Obrovsky rozšířily možnosti, jakými dnes můžeme používat naše mobilní zařízení. Může jít o nakupování, hraní her nebo často také vzdělávání. Dnes nejvíce používanými operačními systémy, pro které se aplikace vyvíjejí jsou Android a Apple iOS. [10]

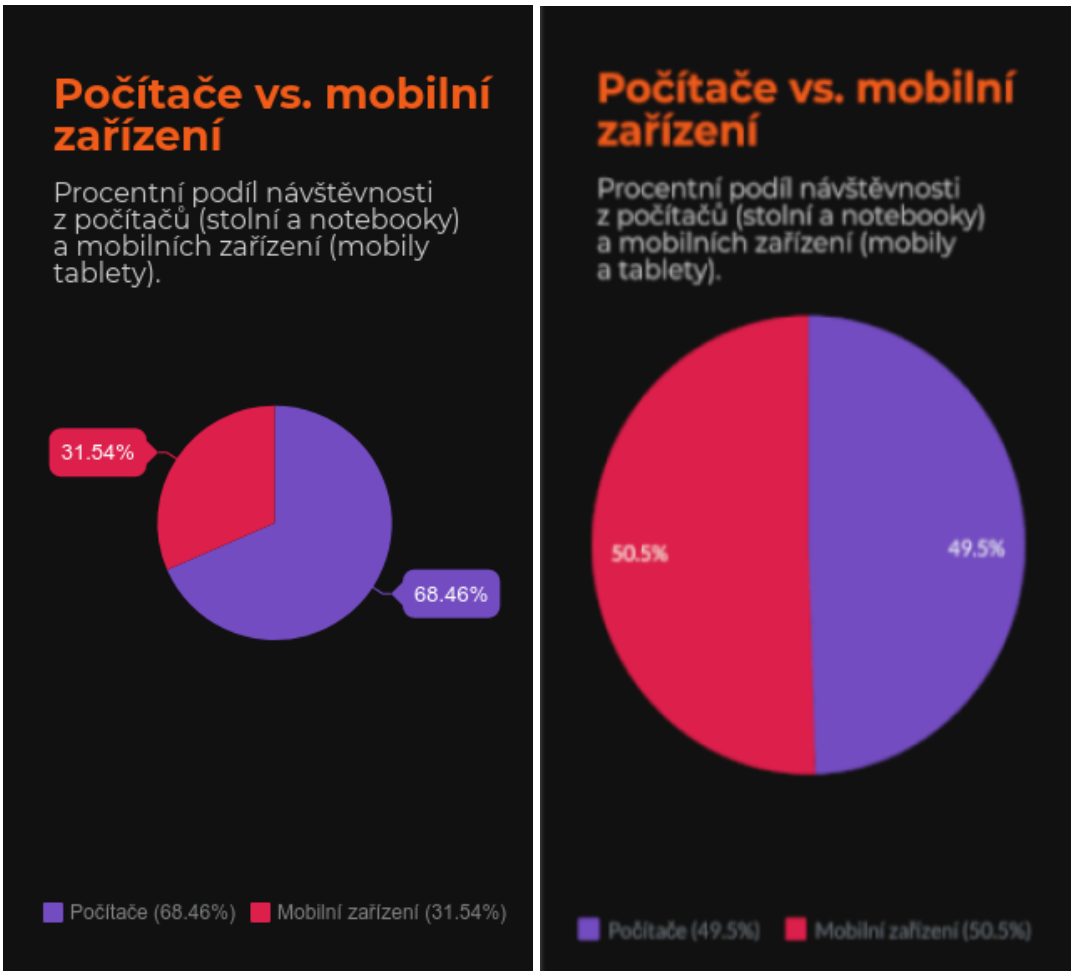
Důvodem, který může motivovat například firmy a společnosti pro vývoj své vlastní aplikace může být stále se zvyšující návštěvnost webových stránek z mobilních zařízení. Dnes již většina chytrých zařízení má téměř neomezený přístup k internetu a potenciální zákazník tak může hledat co potřebuje kdykoliv a kdekoliv. Pokud tedy uživatel může hledat informace přímo z aplikace nainstalované v telefonu je pro něj přístup jednak rychlejší a samotné procházení snadnější a přirozenější než přístup přímo na webovou stránku přes rozhraní prohlížeče. Tato skutečnost významně ovlivňuje ochotu zákazníků investovat do daného produktu nebo služby. [18]

Na obr. 1 níže vidíme porovnání návštěvnosti webových stránek z různých zařízení. Konkrétně jde o mobilní zařízení jako tablety nebo telefony a stolní počítače. Na levém grafu jsou uvedeny údaje z roku 2017. Zde ještě výrazněji převládá návštěvnost ze stolních počítačů. Ale už v roce 2020 (pravý graf) je znatelný nárůst návštěvníků z mobilních zařízení. A to takovým způsobem, že dokonce převládá. Stejný trend zaznamenávají i e-shopy. Lidé dnes nakupují převážně na mobilních zařízeních. Tomu samozřejmě napomáhá velké rozšíření různých nákupních aplikací jako alternativ ke standartním webovým e-shopům. [21]

3.2 Iniciativa Edukids.cz

V dnešní digitální době je internet plný nástrah a lákadel, které ještě před několika lety nebyly známé a mnoho generací s nimi vůbec nepřišlo do styku. Děti, které v tomto světě vyrůstají jsou tím vším však obklopeny od malička. Bohužel ještě ve svém věku nedokážou rozpoznat, co je a co není dobré pro jejich správný rozvoj. Navíc často už od útlého věku mají neomezeně k dispozici některé z chytrých zařízeních jako mobilní telefon nebo tablet. Na něm pak mají přístup ke sledování pohádek, videí a hraní her. Na druhou stranu ani rodiče nemohou všechen svůj čas věnovat dětem pro jejich správný rozvoj. [1]

Tým Edukids.cz se snaží nalézt ideální balanc mezi časem stráveným zábavou a časem věnovaným určitému rozvoji dítěte. Hlavní myšlenka je taková, že činnosti jako hraní her nebo sledování videí je podmíněno splněním určitých úkolů, které dětem může zadat rodič nebo učitel. Dítě tak získává větší jistotu v probraném učivu a úspěchy ho motivují k dalšímu



Obr. 1: Porovnání návštěvnosti webových stránek z počítačů a mobilních zařízení. [21]

studiu. Za jejich splnění dostane dítě určitou dotaci času, který může věnovat zábavě. [1]

Základem celého prostředí jsou dvě aplikace, které spolu úzce komunikují. Jedná se o *dětskou* a *dospěláckou* část. Dětská část aplikace individuálně provází dítě výukou. To se mimo jiné učí také hospodařit se svým časem, jelikož čas strávený zábavou je podmíněn splněním povinností. Dítě má svou část nainstalovanou na všech svých zařízeních. Naproti tomu v dospělácké části má rodič nebo učitel možnost spravovat rozhraní dítěte a sledovat jeho postup. [1]

4 Teoretická část

4.1 Návrh aplikace

Pro úspěšný vývoj mobilní aplikace je dobré si ještě před započetím psaní jakéhokoliv kódu svou práci dobře naplánovat. Lze si tím ušetřit mnoho potenciálních problémů a ve výsledku tedy i času a peněz. [2]

V první řadě jde vůbec o definici zadání. Je klíčové si se zadavatelem práce dobře vyjasnit jeho požadavky na funkčnost a vzhled. Dotyčná osoba může mít různé znalosti softwaru a často má i nereálné požadavky, proto je dobré si vše včas vyjasnit a tuto část nezanedbat. S tím úzce souvisí i cena za vývoj takové aplikace. Cenové představy zákazníka a realita se mohou často značně rozcházet. Je pak zbytečné, aby aplikace obsahovala mnoho přebytečných funkcí, které zákazník nevyužije a vývoj je pak dražší, než je nutné. V zásadě je tedy nutné přihlížet k požadavkům zákazníka, ale také skutečným potřebám uživatelů aplikace. [2]

Může nastat i situace, že si zadavatel myslí, že aplikaci potřebuje a jeho zákazníci ji budou využívat. Po důkladnějším průzkumu trhu pak zjišťuje, že aplikace své využití zkrátka nemá. Chybou je špatný odhad konkrétní cílové skupiny. Práce na aplikaci pak přichází vniveč, jelikož není používána. [2]

Pokud je již definována cílová skupinu, je třeba se v dalším kroku zamyslet nad smyslem aplikace. Aby měl celý vývoj po obě strany smysl, aplikace musí generovat zisk. [2]

Pro definování základní skupiny uživatelů a funkcí se vývoj začíná s tzv. MVP (minimum viable product-produkt s nejmenší možnou funkcionalitou). Jedná se o produkt, který pokrývá minimální požadavky zákazníka a je s ním již možné vyzkoušet, jestli přinese aplikace očekávané benefity. Samotný zadavatel, ale i uživatelé vidí fyzický produkt a mohou k němu vznášet námitky. Tím zároveň vzniká nový prostor pro zlepšení. [2]

Celý vývoj tak můžeme rozdělit do několika po sobě jdoucích kroků:

- analýza trhu
- pochopení potřeb uživatelů
- práce designera na návrhu grafického rozhraní
- validace zadání
- programování
- validace software produktu
- testování
- umístění do obchodu s aplikacemi

– následná správa (rozvoj, opravy chyb a další)

Neustálá validace neboli ověřování během vývoje je klíčovým faktorem celého procesu. Kromě správného směřování produktu se postupně rozšiřuje i uživatelská základna, což hraje důležitou roli po vypuštění aplikace na trh. Navíc uživatelé již prostředí a funkce po uvedení na trh znají, takže užívání aplikace je pro ně mnohem snazší. Nesmíme ani zapomínat, že finální produkt se prvotnímu návrhu příliš nepodobá. Celý proces se v čase mění a přizpůsobuje se aktuální situaci a požadavkům. Během vývoje se totiž často některé prvky ukážou jako důležitější než jiné a některé zase jako zbytečné. Závěrem tedy lze říci, že vývoj aplikace není dílo, které se na začátku navrhne a poté pouze naprogramuje. Jde o proces tvorby, který se neustále mění a zkouší. Díky tomu se i cena finálního produktu může značně lišit od původního odhadu. [2]

Pokud práce neprobíhá na malém projektu je vývoj prakticky vždy týmovou prací. Takový tým se nejčastěji skládá ze čtyř vývojářů, dále dvou lidí na serverové řešení, a nakonec projektového manažera a v ideálním případě i testera. Vyšší počet vývojářů je způsoben nutností vývoje pro dvě platformy-iOS a Android. Pro každou platformu tedy dva vývojáři, aby byla možná vzájemná zastupitelnost a vývoj se nezastavil. V případě, že se bavíme o vývoji multiplatformních aplikací se počet vývojářů snižuje na polovinu. Protože odpadá nutnost psaní kódu pro obě platformy a je vytvořen jeden společný kód. Testeři jsou nezbytnou součástí týmu, jelikož nikdy není dobré dodávat zákazníkovi software s chybami. Nakonec součástí týmu bývá i tzv. product owner. Jde o zástupce ze strany klienta, který přijímá rozhodnutí a odpovědnost. [2]

Za vyvrcholení celého procesu lze považovat uvedení na trh. Po úspěšném vydání funkční aplikace ve virtuálním obchodě však práce zdaleka nekončí. Dále je nutná pravidelná údržba s odstraňováním drobných chyb. Během používání se také přidávají nové funkce nebo se naopak odebírají ty málo používané. A to vše na základě zpětné vazby uživatelů. V dnešní době se všechny technologie ubírají stále kupředu a nejinak je tomu i u operačních systémů. I ty se neustále proměňují a rozšiřují o nové funkce. Pro správný chod softwaru je nutné ho neustále udržovat aktuální a konkurenceschopný. [2]

4.2 Objektově orientované programování

Na úvod je důležité zmínit některé základní rysy objektově orientovaného programování, protože jazyk Dart, o kterém budu mluvit později a je používán jako základní jazyk pro tvorbu aplikací ve Flutteru, je objektově orientovaný jazyk.

Obecně lze říct, že všechny programy fungující na našich počítačích nebo mobilních zařízeních simulují reálné děje ve virtuálním světě. Jako dobrý příklad poslouží textový editor. Ten simuluje dokument psaný na papíře se všemi náležitostmi jako znaky, odstavce atd. Současně můžeme říci, že v reálném světě jsou různé děje výsledkem různých interakcí

mezi skutečnými objekty. Takto si lze zároveň dobře ilustrovat i fungování objektově orientovaného programu. I v programu figurují jakési objekty a jejich vzájemné interakce jsou reprezentovány zprávami mezi objekty.[3]

Tímto se zároveň OOP liší od strukturovaného jazyka. U něj jsou všechny situace řešeny definováním funkcí a parametrů, které program vyhodnotí a na jejich základě provede požadovanou akci. Jde tedy o posloupnost příkazů, jak se má daná situace vyřešit. [3]

Definice objektů, které používáme v OOP může být lehce zavádějící. Jako lidé vnímáme objekty často jako hmotné věci (židle, telefon atd.). V objektově orientovaném jazyce je však za objekt považováno téměř vše, co může nabývat nějakých vlastností. Potom nás nepřekvapí, že objektem jsou i vlastnosti, události nebo stavy. Tedy v podstatě je za objekt považováno vše, co označujeme podstatným jménem. [3]

4.2.1 Atributy

Samotné objekty mohou ve své datové struktuře obsahovat další objekty, které ovlivňují jeho stav a definují jeho vlastnosti. Potom tyto objekty nazýváme právě *atributy*. Konkrétněji se zmíním o atributech tříd a jejích instancí. Zatímco atributy třídy budou vždy stejné pro všechny její instance. Tedy pokud změním některý z atributů dané třídy, ovlivním zároveň tento atribut i u všech instancí dané třídy. Na druhou stranu změna atributu jedné z instancí se projeví pouze u této jediné instance. [3]

4.2.2 Metody

Již bylo zmíněno, že objekty mezi sebou interagují a tyto interakce jsou vyjádřeny zprávami mezi objekty. V praxi tento děj funguje tak, že oslovený objekt na zaslanou zprávu nějak reaguje a tato reakce je označována jako metoda. Celou akci pak označujeme spíše jako volání metody než zasílání zprávy. Objekt může reagovat na zaslanou zprávu různě. Záleží od koho byla zaslána a v jakém stavu se v danou chvíli objekt nacházel.[3]

4.2.3 Třídy

Objekty se jak v reálném, tak ve virtuálním světě často opakují. Pro usnadnění práce programátorům byly v objektově orientovaných jazycích zavedeny tzv. třídy. Třídy definují, jak budou vytvářeny objekty. Ty pak označujeme za instance dané třídy. Třída dále definuje i atributy daného objektu. Avšak je důležité si uvědomit, že atributy mohou nabývat různých hodnot, a tedy třídy nebudou vždy shodné. Jinými slovy všechny objekty v daném programu rozdělíme do určitých skupin-tříd podle nějakých společných vlastností. [3]

Pro lepší přehlednost se k označování jednotlivých entit používají tzv. *identifikátory*, tedy naše pojmenování. Při programování již nyní v moderních OOP nepracujeme přímo s objekty, ale pouze s odkazy na ně. [3]

4.2.4 Rozhraní, implementace a zapouzdření

U moderních programovacích jazyků jsou běžnou praxí pro zachování dobré funkčnosti programu a zvýšení efektivity vývoje dvě vlastnosti-rozhraní a implementace. [3]

Rozhraní definuje, které informace o sobě daná entita (třída, metoda, atribut) zveřejní. S těmito informacemi pak mohou spolupracující programy pracovat a zároveň je musí respektovat. Rozhraní nám tedy specifikuje, co daná třída umí a jak s ní máme komunikovat. Jinými slovy, na jaké zprávy dokáže reagovat. Informace obsažené v rozhraní lze dále dělit do dvou skupin. [4]

Nejdříve zmíním pojem *signatura*. Jde o soubor informací, které jsou kontrolovatelné překladačem neboli kompilátorem. Ten zpracovává zdrojový kód programu do formy čitelné pro stroj, respektive ho překládá do příkazů pro procesor. Ve výsledku je tedy nedodržení signatury vyhodnoceno jako syntaktická chyba a námi napsaný program se nespustí. Z formální stránky se do signatury zahrnují bližší informace o dané entitě. U dat jako jsou proměnné nebo konstanty jde o název a typ. U signatury metod zahrnujeme ještě typy parametrů, synchronizovanost a další.[4]

Za druhou kategorii rozhraní považujeme *kontrakt*. Ten obsahuje informace, které už nejsou kontrolovatelné kompilátorem. Pro příklad může jít o omezení hodnot, kterých může daný parametr nabývat. Jejich nedodržení je často zjištěno až při běhu samotného programu. I přesto, že kontrakt nelze deklarovat přímo v kódu, lze jeho splnění ověřit. K tomu slouží kontrolní příkazy, které kontrolují vstupní a výstupní hodnoty. Protože tyto testy značně prodlužují čas spuštění, spouští se pouze při odlaďování programu. Při ostrém uvedení do provozu již program pracuje bez nich. [4]

Hlavní roli při dodržení kontraktu tak hraje sám programátor. Pro jeho snazší dodržení slouží dokumentační komentáře, kde autor dané entity uvádí, které náležitosti je třeba dodržet. [4]

Implementace nám na druhou stranu říká, jakým způsobem je dosaženo správné funkčnosti dané entity. Tedy plnění funkcí uvedených v rozhraní. Pro usnadnění budoucí práce a možnost aplikovat změny je užitečné, aby okolní program neměl o implementaci dané entity žádné informace. Zamezíme tím závislosti programu na dané implementaci. [4]

Třetí důležitou vlastností OOP je *zapouzdření*. Tento pojem definuje dvě vlastnosti. První z nich říká, že kódy, které s těmito daty pracují jsou definovány v rámci jedné třídy. Druhá zakazuje manipulovat s těmito daty jinak, než dovolují metody vlastníka těchto dat. Tím se vracíme zpět k rozhraní. Tedy přístup k datům je možný jen způsobem v něm popsaným. Samotný programovací jazyk je pak posuzován podle toho, jak moc podporuje ideální zapouzdření. [4]

4.2.5 Dědění

Tato konstrukce nabízí efektivní a snadné řešení, jak zkrátit délku programu a tím i čas strávený psaním kódu. Avšak jde o koncept narušující zásadu zapouzdření, proto by ho mělo být využíváno pouze tam, kde je jiné řešení výrazně složitější. [4]

Často se stává, že narazíme na skupinu instancí, které sdílí nějakou speciální vlastnost. Můžeme pak definovat podtyp, který charakterizuje danou skupinu objektů. Tento podtyp nazýváme *potomkem* původního typu. Z toho poté plyne, že původní nadtyp je považován za *rodiče*. Prvky mohou být někdy také označovány jako *základní* a *odvozený* typ. V zásadě potomek zdědí všechny vlastnosti a schopnosti svého rodiče, ale díky jeho specializaci u jeho instancí můžeme definovat další vlastnosti a schopnosti. [4]

4.2.6 Programování proti rozhraní

Pro správnou tvorbu a zároveň funkčnost programů je nutné při tvorbě kódu dodržet několik základních zásad. Kromě zřetelných zásad jako je přehlednost kódu, vyvarování se duplicitám nebo přidávání komentářů uvedu jednu méně zřejmou zásadu, kterou je *programování proti rozhraní*.[4]

Základní myšlenkou této zásady je, aby proměnné nebyly deklarovány jako instance daných tříd, ale jako instance nějakého datového typu. Jinými slovy si použitím některého z datových typů zadefinuji proměnné a tímto datovým typem mohu dále odkazovat na potřebné třídy. Výhodou je, že při případných budoucích úpravách změním pouze třídu, ale nemusím měnit definici proměnných. [4]

4.3 Návrhové vzory

Historicky bylo současné moderní programování velmi ovlivněno nástupem tzv. návrhových vzorů. Návrhové vzory přináší jakési standardizované řešení pro opakující se situace. Jako dobrá analogie mohou posloužit matematické vzorce, které se pro úspěšné vyřešení úlohy musíme naučit. V tomto případě však nedosazujeme za neznámé čísla, ale prvky OOP, tedy třídy a objekty. [4]

Kromě docílení vyšší efektivity a spolehlivosti programů jsou díky tomuto postupu zachovány i základní znaky OOP. Vývojář dále již nemusí vymýšlet nové řešení opakujícího se problému, ale použije některé ze standardizovaných a praxí ověřených řešení. Tím se předejde i potenciálním chybám. Části programu vytvořené pomocí návrhových programů je možné použít i v dalších programech. A v neposlední řadě usnadňují i komunikaci mezi vývojáři, kteří popíšou danou problematiku jedním názvem a není nutné řešení dlouze a složitě vysvětlovat. [4]

Používané vzory se nejčastěji publikují v katalozích. Za nejznámější a nejuznávanější

je považován katalog nebo spíše kniha GoF. Ta obsahuje 23 základních a nejčastěji používaných vzorů. Název je odvozen od přezdívky, kterou si dala skupina čtyř tvůrců-Gang of four. [4]

4.4 Programování řízené testy

Cílem každého programátora je samozřejmě funkční kód bez chyb. Toho však není vždy snadné dosáhnout. Ale můžeme se tomuto případu alespoň limitně blížit a technika *programování řízeného testy (TDD)* nám může výrazně pomoci takového kódu dosáhnout.[5]

Nejdříve se podíváme na obecný postup při programování řízeném testy:

1. Červený ukazatel
2. Zelený ukazatel
3. Refaktorování
4. Opakování celého procesu

Jedná o způsob vývoje, který je řízen automatickými testy. V první fázi si definujeme funkcionalitu programu, respektive jeho části. Následuje napsání samotného automatizovaného testu. Podstatou je tvorba testů ještě dříve než samotného kódu. Z toho jasně vyplývá, že po spuštění samotného testu bez jakéhokoliv kódu by mělo dojít k chybě, tedy nesplnění testu. Tím ověříme, že test není splněn už jen ze své podstaty. Testy nejčastěji kontrolují nejmenší jednotky kódu, kterými jsou třídy, metody nebo funkce. Zároveň si sám vývojář při psaní testu ověří své dostatečné porozumění řešenému problému a zamezí se tím odchýlení od původní zamýšlené funkcionality programu. [5]

Druhým krokem je vytvoření takového kódu, který náš test již splní. Toho můžeme dosáhnout i méně elegantními kroky, primární je v této fázi splnění testu. Na celkovou efektivitu kódu se zaměřují až další kroky. Pokud se nám i toto podaří pokračujeme celkovým zobecněním kódu a uplatněním pro celou škálu případů, které mohou nastat. [5]

V poslední fázi už pouze provedeme kontrolu a odstraníme duplicity v kódu. Tomuto procesu se říká *refaktorování*. Samozřejmě čím bude naše úloha složitější, tím budeme i my muset postupovat po menších částech kódu. Pro tento krok existují i automatizované nástroje, které tyto úkony provedou za nás. [5]

Celý tento proces se za stálého testování opakuje. Přitom postupujeme po malých krocích směrem k cíli. Pokud některý z testů neprojde, tak se vracíme o několik kroků zpět a hledáme chybu. [5]

Jedná se o velmi předvídatelný způsob vývoje, se kterým vždy víme, kdy jsme s kódem hotovi. Při použití prvního řešení, které nás napadne nebudeme mít možnost zvážit jiné a lepší. Pro použití některého z možných řešení se rozhodujeme na základě výsledků našich

automatizovaných testů. [5]

Další výhodou je také to, že po splnění testu již víme, že máme část programu funkční a eliminujeme tím obavy programátorů z vnesení neúmyslných chyb, které by ve výsledku znemožnily fungování a jejich nalezení by bylo obtížné. Tyto chyby totiž automatické testy okamžitě odhalí. Zároveň v případě změny požadavků na funkcionalitu nebo zavádění nových funkcí stačí změnit stávající nebo vytvořit nové testy a poté podle toho přepsat i kód. [5]

Díky TDD píšeme nový kód pouze v případě selhání automatizovaného testu. Zároveň tím eliminujeme duplicity v kódu. A program se celkově chová mnohem předvídatelněji. [5]

Jako kontrola správné funkčnosti našich testů se také používá úmyslné vložení chyby. Náš dříve úspěšný test by nyní měl selhat. Dalším vhodným prvkem je tzv. párové programování, kdy naši práci kontroluje zkušený kolega. [5]

Ve výsledku bychom měli dojít k tomu, že testy budou pokrývat celý program. Na druhou stranu tyto testy nejsou navrženy na testování výkonu, zátěže nebo použitelnosti. Pro tyto účely bude nutné použití jiných testů. Také je nelze považovat testy, na jejichž základě budou vydávány akceptační protokoly nutné pro vypuštění programu. V neposlední řadě se v testech, stejně jako v kódu, mohou vyskytovat chyby. [5]

Jednou ze zásad je také izolovanost testů. Tedy pokud se pokazí jeden z testů nemělo by to mít vliv na funkčnost ostatních testů. Snáze tak odhalíme, kde se nachází chyba.[5]

Stejně jako pro tvorbu programů i v technice TDD existují vzory, které nám napoví, jak navrhnout testy pro opakující se problémy. V první řadě jde o vzory pro červený ukazatel (ukazatel selhání testu). Tyto vzory nám řeknou, kde a kdy testy psát. Jaké testy psát nám zase napoví *testovací vzory*. Nakonec vzory zeleného ukazatele nám napoví, jak se dostat od červeného ukazatele až k testu, který projde. [5]

Na první pohled nemusí TDD vypadat příliš užitečně, protože samotné testy často svou délkou převyšují samotný kód. Psaní testů samozřejmě zabírá více času. A v počátcích může vést k celkovému snížení produktivity. Musíme však brát v potaz celkový čas, který by nám zabrala standardní tvorba kódu i s jeho konečným čištěním a kontrolou. Ve výsledku se tak s TDD pravděpodobně dostaneme k rychlejšímu, ale hlavně efektivnějšímu vývoji našeho programu. [5]

4.5 Testování aplikace

Průběžné testování navržené aplikace je základem pro její správnou funkčnost a produktivní vývoj. O metodice vývoje založené přímo na testování se zmiňuji výše. Přímo Flutter nabízí několik možností pro implementaci těchto testů, které se příliš neliší od testování jiných softwarových aplikací. Kromě testování jednotlivých prvků aplikace jako jsou třídy můžeme

ve Flutteru testovat přímo i samotné widgety. [6]

4.5.1 Testování widgetů

Testování widgetů pracuje v zásadě totožně jako zkoušení jiných prvků. Pouze je zaměřen na widgety. U widgetů testujeme jejich vzájemné interakce a zda ve výsledku vypadají, tak jak jsme očekávali. Pro usnadnění práce vývojářům přidal Flutter balíček pod názvem *flutter test*, který nabízí nástroje pro psaní testů přímo pro widgety. [6]

4.6 Softwarové licence

Licence se v oblasti informatiky používají pro právní ochranu díla, které je zákonem chráněno. Zároveň v závislosti na typu licence dále upravuje jakým způsobem může být s dílem nakládáno. Tvorbu samotných licencí mají na starosti softwaroví právníci, protože jde o velmi složitou problematiku. Následnou volbu konkrétní licence už určuje sám autor. Pokud se na tvorbě díla podílí více účastníků, nejčastěji se vzdají svých autorských práv ve prospěch třetí strany, která potom stanoví licenci. [7]

Licence se dále dělí podle různých požadavků na koncového uživatele případně svou dostupností. Mohou být zdarma nebo je za jejich použití vyžadován poplatek. [7]

4.6.1 Licence MIT

Jde o softwarovou licenci, která vznikla na Massachusettském technologickém institutu (MIT). Tato licence dává uživateli možnost volně používat a šířit daný kód. Zároveň ho může dle svých potřeb měnit. Jedinou podmínkou použití licence je vydávání díla společně se jménem autora a kopií licence. To z ní dělá nejjednodušší open-source licenci. Jde o licenci nejčastěji používanou na univerzitní půdě. Lze ji použít i pro komerční účely. [8]

4.6.2 Licence Creative Commons (CC)

Licence Creative Commons nabízejí autorskoprávní nástroj pro umožnění ostatním užívat autorovo dílo. Díky tomu mohou další uživatelé používat, šířit a pozměnit zdrojový software a to vše při dodržení autorského práva. Zároveň se autorovi dostane patřičného uznání. [9]

Licence typu CC se dále dělí na několik podtypů. Volba už záleží na dalších konkrétních specifických požadavcích autora. Například zda si přejí povolit komerční použití nebo dovolí vytvoření odvozených děl. Potom však další autoři musí uvádět poskytovatele a zachovat autorskoprávní výhrady. [9]

Každá licence CC se skládá ze tří tzv. vrstev. První je napsána v řeči právníků a jde o tzv. Plné znění licenčních podmínek. Jako další se zde objevuje část srozumitelnější a pochopitelná i pro běžnou veřejnost-Zkrácené znění. Poslední je strojově čitelná verze

licenčních podmínek, která umožňuje webu rozpoznat, která díla jsou chráněna licencí CC. [9]

4.6.3 BSD licence

Licence typu BSD nese název po prvním softwaru, pro který byla možnost šíření použita. Šlo o operační systém Berkeley Software Distribution (BSD). Je považována za jednu z nejsvobodnějších a nejdostupnějších licencí použitelnou i pro komerční účely. Využívá se pro nabízení obsahu, který podléhá některým licencím, kdy je požadováno jen uvedení autora a licenčních informací. [14]

Licence lze použít jako tzv. 2, 3 nebo 4 bodovou. Pro komerční účely stačí uvádět informace o autorovi a zřeknout se zodpovědnosti. Vše ale musí být v souladu s licenčními podmínkami. Výhodou také je, že v některých proprietárních programech není nutné uvádět zdrojový kód. [14]

4.6.4 GPL licence

GPL je zkratka pro General Public Licence-v překladu obecná veřejná licence. Jde o licenci pro svobodný software. Je příkladem tzv. copyleftové licence. Tedy licence, která vyžaduje, aby další odvozená díla byla vydávána pod toutéž licencí. Nabyvatel licence získává s licencí právo upravovat, kopírovat a dále šířit dílo. [15]

4.7 Nástroje pro tvorbu mobilních aplikací

Při vývoji nové mobilní aplikace je nutné brát velký zřetel na to, že mobilní zařízení se svým výkonem nemohou rovnat výkonu běžných počítačů. Předně jde o omezenou paměť, kapacitu baterie nebo procesor či grafické čipy, které už kvůli své velikosti nemají takový výkon. V neposlední řadě je nutné uvažovat také odvod tepla z pracujících součástek. Z těchto zmíněných důvodů je potřeba pro dobrou funkčnost aplikace její dokonalá optimalizace. Nechceme, aby se aplikace zasekávala nebo se vůbec nespustila. S tímto neduhem často trpěly aplikace pro operační systém Android, ale i zde se už tomuto problému daří předcházet. [16]

4.7.1 Nativní vývojové prostředí

Nativní vývojové prostředí je první možností pro vytváření aplikace. To je vytvořeno přímo tvůrcem operačního systému a nabízí prvky specifické přímo pro daný operační systém. Společnost Apple pro tyto účely nabízí systém Xcode, kde probíhá vývoj aplikaci pro iOS. A společnost Google, která stojí za zrodem systému Android, nabízí pro tyto účely Android Studio. Velkou výhodou je okamžité uplatnění nově vydaných aktualizací a jistota, že operační systém bude vývojovou platformu v budoucnu nadále podporovat. Jako nevýhoda

se jeví fakt, že pokud tvůrce potřebuje funkční aplikaci na více platformách je nucen vše tvořit pro každou platformu zvlášť, protože každá používá jiný kódovací jazyk a navzájem se nepodporují. Tato skutečnost prodlužuje čas vývoje a tím oddaluje uvedení na trh. Zároveň se tím zvyšuje potřebná velikost pracovního týmu, s čímž jde ruku v ruce i zvýšení celkových nákladů na vývoj. [11]

4.7.2 Webové aplikace

Webové aplikace jsou alternativou k prvnímu zmíněnému řešení. Jedná se o jednodušší způsob tvorby mobilní aplikace, kdy není nutné tvořit dvakrát zdrojový kód a aplikace bude fungovat na všech operačních systémech, protože pro své spuštění využívá webový prohlížeč. Toto výrazně snižuje čas potřebný pro vývoj aplikace, protože není třeba tvořit dva zdrojové kódy. Nevýhodou je však ztráta některých nástrojů nativního prostředí, které pak musíme pro webové aplikace dotvořit nebo úplně vynechat. Aplikace na webu mají také o něco delší odezvu. Navíc pro implementaci aktualizací je třeba ručně přepsat zdrojový kód. Na druhou stranu aktualizaci není třeba nahrávat do aplikace přes některý z obchodů, ale vše se vyřeší přímo na webu. Webové aplikace také nemohou využít všech vlastností našeho chytrého zařízení jako jsou vibrace, notifikace a další. [11]

4.7.3 Multiplatformní software

Multiplatformní software spojuje výhody obou předcházejících variant. Zdrojový kód je funkční pro obě platformy a není ho nutné tvořit dvakrát. Přitom lze takto dosáhnou nativního prostředí, které je nerozeznatelné od toho vytvořeného pomocí nativního softwaru. Takovými nástroji jsou například software Xamarin společnosti Microsoft nebo React Native společnosti Facebook. Výhodou je okamžitá implementace aktualizací, které vydají společnosti Apple nebo Google. [11]

Multiplatformní aplikaci je také možné vystavit v obchodě s aplikacemi bez rozdílu oproti těm nativním. Tím se zároveň vyřeší i otázka případného zpoplatnění. Autor nastaví v obchodě požadovanou cenu za aplikaci a další transakce již řeší příslušný obchod (Google Play případně App Store). Toto se u webových aplikací řeší jen těžko, protože se k nim přes prohlížeč dokáže dostat téměř kdokoliv. V neposlední řadě je nutné zmínit, že multiplatformní aplikace umožňuje také přístup k hardwaru zařízení jako je GPS senzor nebo fotoaparát. [17]

Multiplatformní software skýtá také určité nevýhody, které se liší podle vydavatele. Díky velkým rozdílům napříč operačními systémy je někdy těžké dosáhnout perfektního nativního vzhledu na všech platformách. [13]

Dalším úskalím může být i hardware, a to především u zařízení s OS Android. Produkty se dodávají na trh s různými velikostmi displejů a rozdílnými výkonovými parametry. Připravit

potom aplikaci perfektně pracující na všech typech zařízení je tedy téměř nemožné a můžeme se setkat s různými nedostatky jak grafickými, tak v plynulosti zobrazování. [13]

4.8 Framework Flutter

Pro začátek se ohlédneme lehce do historie a stručně se podíváme na vývoj Flutteru. Úplně prvotně byl Flutter představen již v roce 2015 na Dart Developer Summit Ericem Seidelem. Prvotním impulsem pro vytvoření takového nástroje byla snaha společnosti Google představit po letech experimentování něco nového a lepšího pro vývoj aplikací pro mobilní telefony. V roce 2016 již byl software prezentován jako Flutter a o rok později, v květnu, přichází první alfa verze. V roce 2018 již máme na trhu stabilní verzi. [6]

Samotní vývojáři společnosti Google popisují Flutter jako přenosný nástroj pro tvorbu aplikací s nativním uživatelským rozhraním a jednotným zdrojovým kódem pro více platforem. [18]

První verze Flutteru se zaměřovala především na tvorbu aplikací pro OS Android a iOS. V nové verzi 2.0 nově umožňuje tvorbu aplikací i pro operační systémy jako je Linux, Windows, MacOS, Google Fuchsia a dokonce i pro web. Největší podporu má ale u již zmíněných mobilních aplikací.[13] Ač by se mohlo zdát, že jako nový hráč na trhu pro něj bude těžké si zde vydobýt své místo, není tomu tak. Velké podpory se mu dostává právě od Googlu. Díky tomu pro Flutter nebylo tak obtížné se stát populární. [6]

Jedná se také o multiplatformní software, ale od předešlých dvou zmíněných v kapitole o multiplatformním software se trochu liší. Pro tvorbu aplikací vůbec nevyužívá nativní komponenty, ale má své vlastní rozhraní a design, které pro vzhled aplikací používá. Ze své sady komponentů (widgetů) potom vykreslí na čisté plátno (canvas) funkční aplikaci. Zároveň obsahuje komponenty specifické vzhledem pro Android i iOS a aplikace díky tomu vypadá jako nativní. Velkou část funkcí vytvoříme použitím společných prvků. Pokud ale dojde k situaci, kdy je potřeba v zařízeních s rozdílnými OS jejich nativních prvků, je nutné do zdrojového kódu přidat rozhodovací funkci, která každému OS přiřadí správný prvek. [13]

Díky kreslení grafických prvků na canvas je software velmi rychlý. A to je jedna z jeho velkých předností. Tato skutečnost také nahrává velké plynulosti animací, pro které má Flutter velkou podporu a snaží se jejich tvorbu učinit pro vývojáře co nejjednodušší. Aplikace vytvořené ve Flutteru běžně podporují obnovovací frekvenci 60 nebo 120 Fps. [13]

A vysoký výkon je to, na čem si vývojáři Flutteru velmi zakládají. Napomáhá tomu i tzv. AOT (ahead-of-time) kompilátor, který překládá Dart do nativního kódu daného OS a běží tedy stejně jako nativní aplikace. [6] Vlastní grafické prvky přináší ale jednu komplikaci. Vždy, když vyjde nová aktualizace pro jeden z OS i Flutter musí přijít s novou aktualizací. Nespornou výhodou všech multiplatformních softwarů je, že aplikaci stačí odladit pro jednu

z platforem a dále máme jistotu, že stejný sled akcí nastane i na ostatních OS. [13]

Základní prvky frameworku Flutter

- Jazyk Dart
- Flutter modul
- Základní knihovna
- Widgety

4.8.1 Programovací jazyk Dart

Dart je programovacím jazykem Flutteru. Jde o již dříve používaný jazyk, který se v mnohém inspiroval například jazykem Java, JavaScript nebo Swift. Proto nebylo nutné vyvíjet úplně nové koncepty. Pouze byly zdokonaleny ty již známé a použité pro potřeby Dartu. Navíc pro trochu znalého programátora není těžké se ho naučit. Ale i úplný začátečník by s ním neměl dlouze bojovat.[6]

Byl také vyvinut společností Google, ale neměl mnoho praktických využití a nebyl vývojáři příliš používán. To se s nástupem Flutteru změnilo a nyní je naopak pro mnoho svých funkcí chválen. Je také považován za typově bezpečný jazyk, díky čemuž je zamezeno fatálním chybám v kódu.[6]

Jeho nespornou výhodou, usnadňující práci vývojářům, je modul JIT (just-in-time) umožňující okamžitou kompilaci. K tomu nabízí dvě funkce. Tzv. hot reload, který přenese změny v kódu přímo do běžící aplikace bez nutnosti úplného restartu aplikace. Změny jsou okamžitě viditelné na souběžně zapnutém simulátoru. Tato funkce velmi urychluje vývoj.[6]

Dále nabízí funkci tzv. Stateful hot reload, tedy okamžitého obnovení stavu aplikace. Aplikace se tedy obnoví do původního stavu, jaký měla hned po spuštění.[6]

Velmi nápomocná je i funkce vestavěného analyzátoru, který zvýrazní potenciální problémy v kódu a navrhne řešení ještě před samotnou kompilací. [6]

4.8.2 Knihovny, balíčky a pub

Jedním z důležitých softwarových nástrojů jazyka Dart je *pub*. Sloužící pro správu knihoven a softwarových balíků. [6]

Knihovny slouží jako nástroj pro štěpení kódu kvůli lepší orientaci. Zároveň se opět vracíme k pravidlu zapouzdřenosti kódu. Tímto krokem definujeme, co je a není vidět pro ostatní knihovny. Rozdělení kódu a vytvoření knihoven nám navíc dává možnost jednotlivé části použít na více místech bez nutnosti jejich opětovného vytváření. Případně je můžeme sdílet s ostatními vývojáři.[6]

Novou knihovnu vytvoříme jednoduše vytvořením složky s koncovkou *.dart*. A pro její použití v jiné složce jí jen importujeme.[6]

Velmi podobným způsobem fungují *balíčky (packages)* třetích stran. Ve své podstatě nejde o nic jiného než o již vytvořený kód, který plní určitou specifickou funkci. Tím činí práci vývojářů mnohem snazší a rychlejší. Tyto balíčky dále ještě rozlišujeme podle jejich zdrojového kódu. Některé jsou napsány přímo jazykem Dart, těm pak říkáme *packages*. V druhém případě je zdrojový kód napsán jazykem jiným (Java, Kotlin...) jeho funkcionalita je však stejná. Potom jim říkáme *pluginy (zásuvný modul)*. [6]

Jak jsem již zmínil v našem projektu můžeme v zásadě využít dvou typů knihoven. První typ je ten, který jsme si sami vytvořili pro rozštěpení kódu do několika složek a lepší orientaci v kódu. Důležité je zmínit, že tyto složky se nachází ve stejné zdrojové složce celého našeho projektu. Druhou možností je použití knihoven, které již někdo vytvořil za nás. Ty ovšem v našem počítači ještě nemáme. Pro tyto účely slouží nástroj *pub*. Ten spravuje všechny balíčky třetích stran. K tomu má k dispozici složku *pubspec.yaml*, kde jsou všechny balíčky blíže specifikovány. *Yaml* je často používaný syntax právě pro specifikaci složek. [6]

Jedním z velmi užitečných balíčků je balíček *test*. Ten nabízí soubor jednotkových testů, které kontrolují malé jednotky kódu. Můžeme tak efektivněji dosáhnout funkčního kódu. Jedná se o trochu jiný přístup než při TDD, protože zde testujeme kód až po jeho vytvoření. Nepíšeme tedy kód až na základě výsledku testů. [6]

4.8.3 Flutter modul

Tento modul má na starosti vykreslování grafiky na nízké úrovni, k čemuž využívá grafické knihovny Skia, které jsou také od společnosti Google. Zajišťuje také implementaci základní knihovny společně s animací a grafikou. Jeho základním jazykem je C++, ale většina vývojářů se s ním fyzicky nepotká a veškerá komunikace se odehrává ve frameworku Flutter. [6]

4.8.4 Základní knihovna

Základní knihovna je napsána v jazyce Dart a obsahuje základní funkce a třídy pro tvorbu aplikace. [12]

4.8.5 Widgety

Widgety jsou jakési stavební kostky celé aplikace a díky nim získává Flutter své velké výhody ve variabilitě aplikací. Každý prvek implementovaný do aplikace je vytvořen pomocí widgetů. Ty se postupně skládají dohromady až vzniknou složitější widgety a nejvyšší z nich potom tvoří celou aplikaci. Jde o jakési základní nástroje pro tvorbu aplikace. Vše je potom

vykresleno na plátno (canvas) obrazovky v podobě uživatelského rozhraní, se kterým již interaguje samotný uživatel aplikace.[6]

Samozřejmě mezi widgety existuje určitá hierarchie. Již při prvotním návrhu vzhledu aplikace se nám začne formovat v určitém uspořádání. To může být do sloupců, řádků nebo naprosto jinak. To vše záleží na vývojáři. A i tohoto rozložení dosáhneme pomocí widgetů. Pro příklad uvedu widget *collumn*, který seřadí ostatní vložené widgety (své potomky) do sloupce. [6]

Každý widget v sobě musí obsahovat metodu. Nejčastěji jde o metodu *build*. Ta říká, jak se má daný prvek zobrazit v souvislosti ostatních widgetů. [6]

Zde se trochu vrátím k základům OOP zmíněných výše. I zde je důležité, a především přehlednější uchovávat části našeho kódu separátně. Tedy každá část kódu, která má na starost jistou funkci v aplikaci je oddělena ve vlastní složce. To slouží vývojářům pro větší přehlednost ve složitějších projektech. [3]

State

Je důležité zmínit, že widgety nejsou součástí samotné aplikace. Pouze popisují, jak se mají dané prvky vytvořit na plátně. Jsou tedy imunní vůči vnějším zásahům. Tím se dostáváme ke "stavům"aplikace. Pokud chceme, aby s námi interagovala a provedla v sobě změny, které chceme, je potřeba její stav definovat. [6]

Primárně rozlišujeme dva druhy widgetů. Existuje ještě třetí typ, ale ten je vůči těmto dvěma spíše okrajový a zmiňovat se o něm tedy nebudu. První je *StatelessWidget*. Jde o widget, který nemá, respektive nemění svůj stav. Přebírá ho od svého rodiče. Je tedy imunní vůči zásahům uživatelům aplikace. Naproti tomu druhý *StatefulWidget* svůj stav měnit dokáže a díky tomu si můžeme například odškrtávat položky v nákupním seznamu. To by se *StatelessWidgetem* nebylo možné. [6]

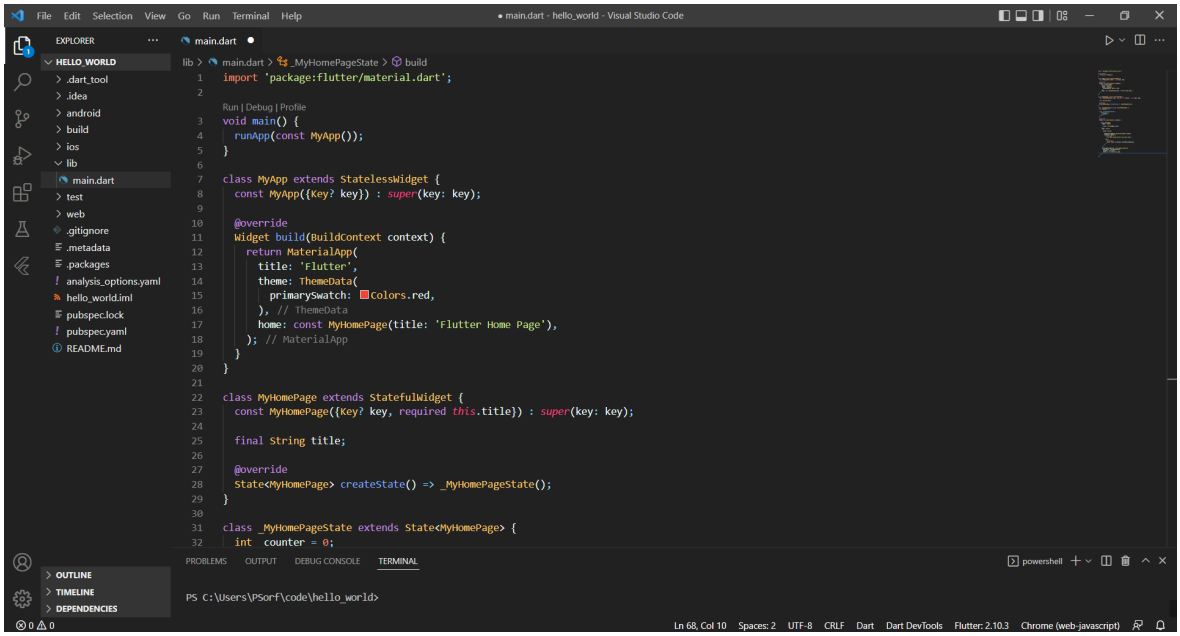
Příklady použití v praxi

Jakožto tvůrce s Flutterem nejvíce do budoucna počítá Google. Byl ale také použit například pro tvorbu Google asistenta. Jako programovací nástroj si ho vybrala i společnost Toyota pro tvorbu svého infotainmentu. Už to, že s ním pracuje tak obrovská společnost jako Google nebo Toyota mu slibuje velkou budoucnost. [12] [13]

4.9 Struktura aplikace ve Flutteru

Pro tvorbu aplikace je v první fázi nutné mít nainstalované některé z vývojových prostředí, kterými mohou být Android Studio případně Visual Studio Code. Tato prostředí dělají samotný kód přehlednější pomocí barevného zvýraznění a usnadňují samotnou tvorbu nabídkou, respektive dokončováním příkazů. Pokud jde o Flutter, tak ten je do vývojového prostředí nainstalován jako plug-in. [18]

Struktura zdrojového kódu je ve Flutteru pevně dána. Pro vytvoření projektu lze použít nejjednodušší příkaz *flutter create*. Následně se vytvoří základní projekt s jednoduchou příkladovou aplikací a komentáři k ní. [18] Příkaz lze dále modifikovat a vytvořený nový projekt tím pozměnit, ale pro naše účely se tím nebudeme zabývat. [6]



Obr. 2: Struktura aplikace ve Flutteru

Na obr. 2 můžeme vidět strukturu zdrojového kódu jednoduché ukázkové aplikace v prostředí SDK Visual Studio Code. V dolní části je vidět terminál, pomocí kterého můžeme s prostředím komunikovat.

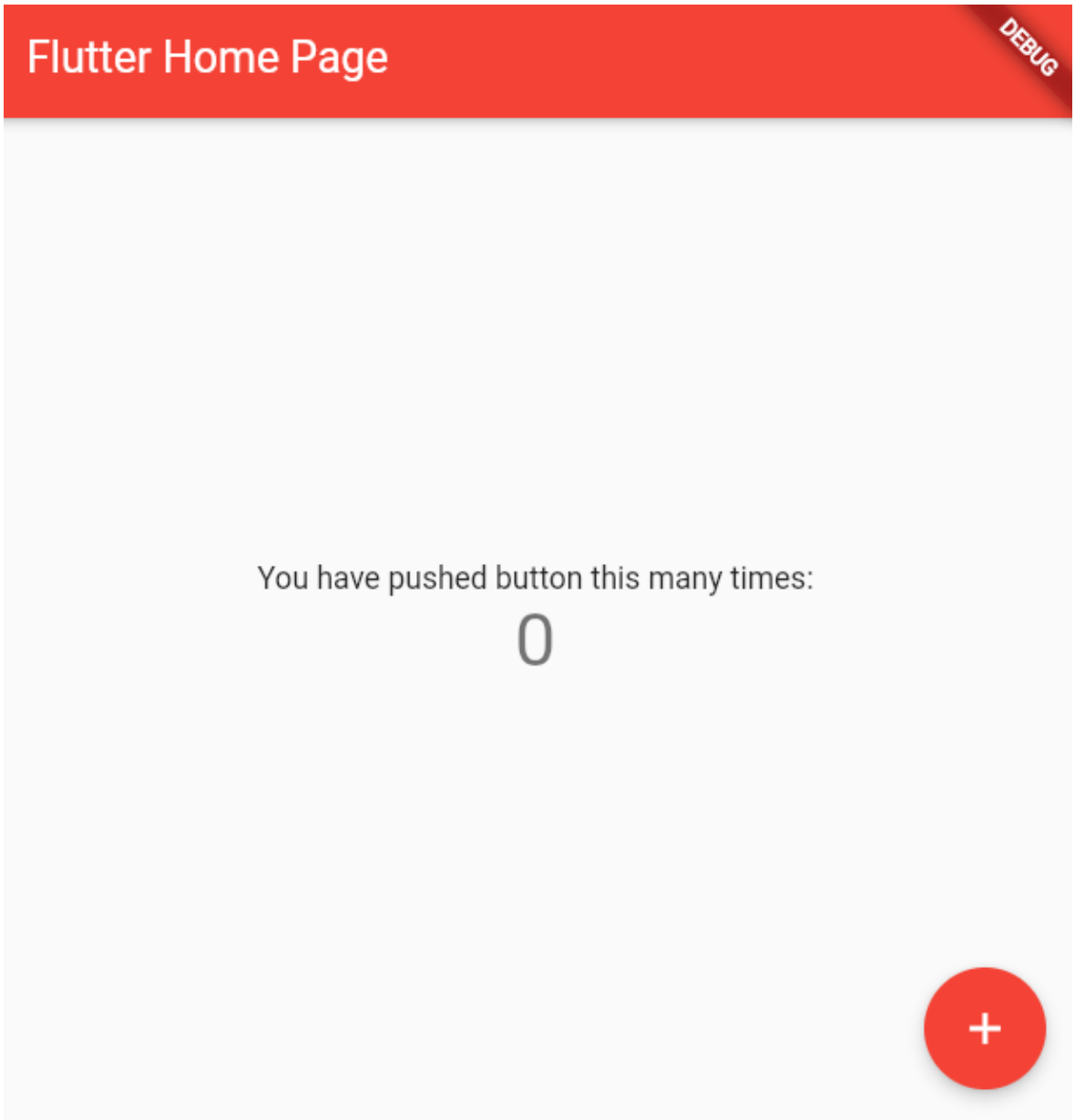
Aplikace v sobě po vygenerování obsahuje několik složek, které vidíme na obr.2 po levé straně. Z nichž bych zmínil ty, které se v projektu musí vždy objevit. Prvně jde o složky *ios* a *android*, které v sobě zahrnují funkce pro překlad vytvořené aplikace do nativního kódu obou platforem. Další a pravděpodobně nejdůležitější je složka *lib*, která v sobě zároveň zahrnuje spouštěcí soubor *main.dart*. Zároveň se ve složce *lib* vyskytuje všechen námi napsaný kód včetně paralelních knihoven. Dále zde máme složky *pubspec.yaml* spravující právě knihovny. Nakonec ještě složka *README.md* se základním popisem celého projektu a ještě složku *tests*. Zde uchováváme všechny testy, které se vážou na náš projekt. [6]

A nyní již přejdeme do samotného kódu a spuštění aplikace. Primárně musíme k našemu vývojovému prostředí mít připojené zařízení, na kterém spuštěnou aplikaci uvidíme. To může být buď fyzické zařízení, ale mnohem častěji spíše jeho simulátor přímo v našem PC. Informaci o něm zobrazuje naše vývojové prostředí vpravo dole, přesně jako je vidět na obr. 2. [6]

Jako první vždy nahráváme knihovny, na kterých bude naše aplikace závislá. V příkladu na obrázku jde o knihovnu *material.dart*. Tím si rozšíříme pole prvků, které můžeme použít.

Důležité je zmínit, že knihovny je nutné nahrát do každého jednotlivého souboru zvlášť v závislosti na tom, se kterými knihovnami bude daná složka pracovat. Počátkem každé aplikace vytvořené ve Flutteru je příkaz *main()*. Ten je klíčový vůbec pro spuštění aplikace a je první, po kterém bude kompilátor (překladač) pátrat.[6]

Aplikaci spustíme zadáním příkazu *flutter run* do terminálu. Tím spustíme debugger. Jde o nástroj při odstraňování chyb ve zdrojovém kódu. Poznáme to mimo jiné červenou značkou s nápisem *debug* v pravém horním rohu běžící aplikace, přesně jak je vidět na obr.3. Navíc tím zpřístupníme funkci *hot reload*, o které jsem se již zmiňoval výše. [6]



Obr. 3: Spuštěná ukázková aplikace debug módu

4.10 Správa verzí

Nejen po dokončení zdrojového kódu aplikace, ale i během procesu tvorby, často potřebujeme nástroj, který nám umožní spravovat a ukládat naše pokroky. [20]

Kromě prostého uložení našeho výsledného kódu, ale někdy potřebujeme s kódem experimentovat a vyzkoušet nové nápady. Není však pravidlem, že se tyto změny osvědčí a budeme je chtít skutečně implementovat do naší aplikace. Potom se dostáváme do situace, kdy se potřebuje vrátit o několik kroků zpět do bodu, ze kterého jsme vyšli. Samozřejmě se nabízí varianta prostého kopírování kódu do textového editoru, ale tento postup je velmi zdlouhavý, neproduktivní a náchylný k chybovosti. Navíc bychom se s velkou pravděpodobností ve změti kódů brzy ztratili a bylo by pro nás velmi obtížné sledovat, které změny byly v jednotlivých verzích aplikovány. [20]

Dalším specifikem vývoje jakéhokoliv softwaru je častá spolupráce mezi vývojáři, kteří mezi sebou potřebují kód nějakým způsobem sdílet. I zde se nabízí varianta sdílení například emailem, ale ani toto řešení by nebylo příliš šikovné. [20]

Všechny tyto výše zmíněné příklady nám dávají záminku k použití nástroje, který je pro tyto účely navržen a navíc nabízí mnoho přidaných funkcí pro ještě snadnější práci. Takový software se nazývá *správce verzí*. [20]

Mezi programátory, ale i například vědci je nejpopulárnější a nejvíce používaný nástroj Git a GitHub. GitHub funguje jako cloudové úložiště, kde mohou tvůrci sdílet svoji práci s ostatními. Tato práce je uložená prostřednictvím tzv. *commintů* v Git, který běží lokálně na našem počítači. Tyto *commity* jsou pak ukládány právě na GitHub. [20]

Proces ukládání změn pomocí nástroje *Git* není příliš složitý. V zásadě k tomu stačí dva příkazy - *git add* a *git commit*. V první fázi musíme ve složce s naší prací samotný Git vůbec spustit. To provedeme příkazem *git init*. Tím se vytvoří neviditelné skladiště (repository) našich verzí s koncovkou *.git*. Potom už můžeme začít tvořit.[20]

Po každé provedené změně, kterou budeme chtít uložit napíšeme do terminálu příkaz *git add* a název souboru, který chceme uložit. Tím se naše verze s provedenými změnami uloží do jakési mezistanice s názvem *staging area*. Zde můžeme průběžně ukládat změny z různých souborů. Důvod je ten, že některé změny často zahrnují přepsaní více souborů najednou, ale my tuto změnu chceme zabalit do jedné. Nakonec následuje příkaz *git commit* se stručným komentářem provedené změny. Je zvyklostí, že jsou tyto zprávy psány anglicky. V případě open-source softwaru je pak kód lépe čitelný pro širší okruh vývojářů. [20]

Daná verze je potom permanentně uložena nezávisle na následně provedených změnách a je možné se k ní vrátit. Každá takto uložená verze má své označení-kombinace číslic a písmen-podle kterého je možné jí později najít. Kromě označení se ve výpisu verzí píše také kdo změnu provedl a komentář, který k ní dopsal. [20]

Jednou z dalších užitečných funkcí Gitu je tzv. branchování. Ve zkratce jde o možnost vytvoření více paralelních větví naší práce v repository. Může jít například o dva různé vzhledy webové stránky, které vytvořím souběžně a potom se rozhodnu, kterým směrem budu pokračovat. Primární větev, na které pracujeme hned od začátku nese označení *master*. My se pak můžeme kdykoliv odchýlit jiným směrem a tuto svou větev si můžeme i pojmenovat. Samozřejmě pak tuto větev můžeme později prohlásit i za hlavní (master) pokud v ní provedené změny chceme uplatnit. [20]

Pro zobrazení kompletního výpisu z historie uložených verzí se používá příkaz *git log*. Ten nám vypíše přehledně do seznamu naše uložené commity s označením, zprávou o provedených změnách a datem uložení. [20]

Jako poslední užitečný příkaz uvedu *git status*. Po jeho zadání do konzole nám Git vypíše seznam složek v našem adresáři, na kterých byly provedeny nějaké změny, které doposud nebyly uloženy. Společně s tím také vypíše složky čekající ve staging area na commit a přesunutí do repository. [20]

I přes všestranné využití správce verzí ho není vhodné aplikovat na všechny typy souborů. Obecně platí, že je nejužitečnější pro soubory, které mají textový základ. Git totiž poprvé uloží celý dokument a poté již v každém commitu ukládá pouze provedené změny, které zabírají svou velikostí minimum úložiště. To neplatí pro soubory s jiným než textovým základem-například obrázky. Git potom při každém commitu ukládá celý soubor a velikost našeho repository exponenciálně roste. [20]

Kromě typu ukládaného souboru je také dobré zvážit, jaké informace ukládáme. To především v případě, že budeme naši práci veřejně sdílet online. Může se stát, že text obsahuje citlivé informace jako hesla. Ta jistě veřejně sdílet nechceme. Pro separaci těchto informací existuje složka *.gitignore*. Sem zahrneme data, která si nepřejeme sledovat a Git je nebude commitovat. [20]

Pro účely online sdílení se používá platforma GitHub. Zde si založíme účet a svoji práci můžeme sdílet. Případně můžeme používat i práce jiných uživatelů, kteří ji zde sdílejí. S tím velmi úzce souvisí nutnost použití některé z licencí, o kterých jsem se již zmiňoval.[20]

Naše nahraná práce je od začátku veřejně viditelná. V případě práce na projektu, který nechceme učinit viditelným pro ostatní dokud nebude dokončen je možnost ho zpřístupnit pouze skupině vybraných uživatelů případně ho zneviditelnit úplně. [20]

Další výhodou ukládání práce do vzdáleného úložiště je záloha dat v případě náhlé ztráty nebo poškození našeho počítače. K naší práci pak můžeme pohodlně přistupovat z jakéhokoliv jiného počítače s připojením na internet. [20]

V případě komplikací při vývoji sem můžeme také nahrát svůj kód a nechat ostatní uživatele, aby nám pomohli s řešením. [20]

V dnešní době je mnoho prací nebo softwaru sdíleno pro efektivní práci právě tímto způsobem pro miliony uživatelů na celém světě. Při jeho používání se může stát, že nalezneme nějakou chybu a chceme ji opravit a usnadnit tím práci ostatním. Potom budeme postupovat tak, že si nejdříve originální repository naklonujeme do našeho počítače jako by bylo naše vlastní. Commity s provedenými změnami ale nemůžeme do původního repository přímo nahrát jako do našeho vlastního, ale pošleme vlastníkovi repository jakousi zprávu - *pull request* o tom, že chceme do jeho repository nahrát určitou změnu. Majitel naši změnu zkontroluje a povolí její přidružení do svého repository. [20]

Správa verzí však nemusí sloužit pouze programátorům. Je široce použitelná pro kohokoliv, kdo potřebuje jakýmkoliv způsobem spravovat verze své práce. Pro příklad mohu uvést spisovatele nebo vědce. [20]

4.10.1 Verze software

Nejdříve zmíním, jaký je nejčastější postup vývoje a označování průběžně vytvářených verzí. [19]

Označení verze je složeno ze tří čísel např. 7.5.1. První číslo je označováno jako hlavní, druhé jako vedlejší a poslední je číslo revize. [19]

Jako první vznikají verze začínající nulou - např. 0.95. Jde o již funkční software, ale vývojář ho ještě nechce oficiálně uvést na trh, ale již ho chce nabídnout uživatelům a doladit poslední detaily. [19]

Dále již přichází verze 1.0.0. Jde o verzi, která vyšla jako první a vše s ní začíná. Posléze přicházejí revize, ve kterých jsou napraveny menší chyby, které se vyskytly během používání a ty jsou označovány 1.0.x. [19]

Ve chvíli, kdy je už revizí příliš je nutné kód znovu důkladně projít a reevidovat. Po těchto změnách je uvedena verze 1.1.0. I zde se samozřejmě mohou objevit nedostatky a na místě třetího čísla se opět objevují čísla revizí. [19]

Pokud je program od základů přepracován dochází na změnu verze a na scénu se dostává verze 2.0.0. I s ní potom nadále přicházejí další větší či menší revize a celý koloběh se opakuje. [19]

Číselné označování však není jediný způsob, jak zachovat pořádek ve vydaných verzích. Některé společnosti jako např. Microsoft použily pro označení svého softwaru rok jeho vydání (95, 98). Stejná společnost však používá také slova. [19]

Verze alfa značí, že program je ve fázi vývoje a je určen především pro testování zkušeným programátorům. [19]

Verze beta je již dostupná i pro běžné uživatele, ale jde především o fanoušky daného softwaru, kteří chtějí, co nejdříve vidět nové funkce. Program ani zde ještě není zcela stabilní

a může působit systémové problémy. Jejich rizika však nejsou nějak velká. [19]

V testovací fázi může být ještě vydána verze Gamma. Jde o obdobu verze beta. Dále se objevují verze s označením pre. Ty mohou být pro systém uživatele velmi nebezpečné a můžou zhroutit celý systém. [19]

Jako poslední zmíním RC verzi (Release Candidate-kandidát na uvedení na trh). V tomto případě jde stále o testovanou a vyvíjenou verzi, která se stále může chovat nepředvídatelně. Je to však poslední verze před finálním uvedením na trh. [19]

5 Praktická část

5.1 Povýšení aplikace na nejnovější verzi

Vývojáři často kromě vývoje kompletně nové aplikace nebo softwaru řeší problém přechodu na nejnovější verzi. Obdobný problém se vyskytl i u aplikace matikadokapsy. Samotná aplikace vznikala v roce 2020, kdy byla ještě dostupná pouze první verze Flutteru 1.x.x. V březnu roku 2021 vyšla ovšem nové verze 2.0.0. [22]

5.1.1 Nové funkce verze 2.0.0

Z mého pohledu nejzásadnější změnou v nové verzi byla přidaná podpora pro tvorbu aplikací i na dalších platformách. Konkrétně šlo například o web, Windows, macOS nebo Linux. Samozřejmostí je také oprava mnoha chyb a přidání spousty nových widgetů. [22]

Druhou klíčovou novinkou je podpora *nullability*. Pro pochopení tohoto konceptu nejdříve vysvětlím pojem *null*. Jak už název napovídá, jde o prázdnou nebo žádnou hodnotu. A i s takovým případem se program musí umět vypořádat, protože hodnota se například nemusí správně načíst z konzole nebo ji uživatel zkrátka nezadá. Pokud by však náš program vrátil hodnotu jako 0 nebo 1 my bychom nepoznali, zda jde opravdu o zadanou hodnotu nebo o označení chyby. Toto byl důvod pro vytvoření speciální proměnné *null*. Jenom doplním, že aby daný datový typ mohl být nullovatelný, musíme ho deklarovat jako nullovatelný. Koncept si lze představit jako box, který vždy existuje, ale buď v něm najdeme přiřazenou hodnotu nebo je prázdný (null). [23]

A nyní již přejdeme k vlastnosti *nullsafety*. Jak jsem již zmiňoval výše, pokud by hodnota nepodporovala nullabilitu a nebyla jí přiřazena žádná hodnota, tak by kompilátor při překladu kódu nebyl schopen tuto operaci vykonat. **Null-safety** je tedy mechanismus, který při překladu kontroluje, zda používáme nullovatelné datové typy. Pokud pak datový typ nebude deklarován jako nullovatelný a nebude mu přiřazena žádná hodnota, kompilátor nebude schopen kód přeložit a vrátí chybu. V opačném případě pokud bude deklarován jako

nullovatený a nebude mu přiřazena žádná hodnota, tak se program přeloží, ale nevypíše žádnou hodnotu. [23]

5.1.2 Přechod na novou verzi

Nejprve se zmíním o praktických důvodech, které nás k tomuto kroku vůbec vedou. Nejprve jde o nové prvky dostupné v nové verzi. Jak jsem zmiňoval výše, Flutter 2.0.0 nabízí kromě opravy chyb i mnoho nových funkcí a widgetů. Pokud je chceme do našeho projektu implementovat nezbývá než povýšit náš projekt. Dále přechodem na novou verzi získáme podporu pro mechanismus null-safety, který předchozí verze nepodporovala. V neposlední řadě jde také o kompatibilitu s knihovnami (packages). Může se velmi snadno stát, že autor námi používané knihovny v jejím zdrojovém kódu provede nějaké změny, které nahraje v její nové verzi a starší verzi odstraní. Náš starší projekt potom nebude moci s touto knihovnou pracovat, protože jeho verze zkrátka prvky knihovny nebude podporovat. Kompilátor by pak měl problém s překladem a aplikace by se nespustila. [22].

A nyní již přejdeme k doporučenému postupu povýšení verze pro můj případ s aplikací *matikadokapsy.cz*. Celý proces lze shrnout do několika po sobě jdoucích kroků.

1. **Upgrade Flutteru**
   Pro úspěšné převedení stávajícího projektu do nové verze je samozřejmě třeba mít i Flutter aktualizovaný na nejnovější verzi. [24]

   Nejjistějším krokem je použití poslední vydané *stable* verze, tedy verze, na které už nejsou prováděny žádné velké změny a je plně funkční bez ohrožení pro program. Tento typ verze lze libovolně při přechodu na novou verzi volit a Flutter nabízí tyto tři možnosti - *master, beta a stable*. *Master* je verze nebo spíše kanál s posledními a nejaktuálnějšími změnami. Ovšem není úplně stabilní a může docházet k nečekanému "padání"programu. Dále Flutter nabízí verzi *beta*. Zde jde o výběr nejlepší *master* verze za poslední měsíc a tato verze je povýšena na *beta* verzi. A nakonec zde máme již zmiňovanou *stable* verzi, která je již tou nejlepší možnou konstrukcí, kterou vývojáři zvolili. A sami ji doporučují pro tvorbu našich projektů. Důvodem je kromě stability také priorita opravy chyb. Tedy pokud se ve Flutteru vyskytne nějaký problém, první opravenou verzí je verze *master*. [25]

   Povýšení Flutteru provedu zadáním příkazu *flutter upgrade*. Tento krok může zabrat i několik minut. Současně s kontrolou aktuální verze Flutteru kontroluje systém i aktuální verze závislých komponent nezbytných pro vývoj. V případě neaktuální verze je tedy potřeba i tyto součásti povýšit na novější. Může jít například o Android Studio nebo Visual Studio Code. Následně lze zkontrolovat aktuální verzi Flutteru příkazem *flutter –version*, který mi vypíše jak stávající verzi, tak i kanál, na kterém pracuji. Pokud by mě zajímal pouze kanál, lze využít příkaz *flutter channel*.

2. **Nahrání projektu**

   Nyní se dostávám do fáze, kdy začnu již pracovat s kódem. Jelikož nezačínám s úplně novým projektem na prázdném plátně, ale budu modifikovat již stávající funkční aplikaci, musím do svého pracovního prostředí (v mém případě VS code) nahrát zdrojový kód aplikace. Ten se nachází na sdíleném online úložišti GitHub. Jde o open-source projekt, takže není třeba zvláštního povolení pro přístup ke kódu. Pro stažení projektu do mého počítače a možnost práce na něm existují v podstatě dvě varianty. První z nich je stažení *.zip* souboru se všemi zdrojovými složkami a jejich následné extrahování. Druhá možnost je postupovat přes Git přímo v počítači. Na GitHubu zkopíruji odkaz na daný soubor a dále příkazem *clone* v konzoli *Gitu* soubor stáhnu do mého počítače. Jediným rozdílem těchto dvou postupů je, že v druhém případě se mi rovnou se stažením souboru iniciuje Git a následně provedené změny lze rovnou commitovat. V prvním případě pak iniciaci provedu příkazem *git init*, přesně jako jsem se o tom zmiňoval v odstavci o správě verzí.

3. **Povýšení dartu na null-safety**

   Ihned po otevření projektu ve vývojovém prostředí (VS code) se objeví několik chyb. To je způsobeno tím, že mám již aktualizovaný Flutter na nejnovější verzi, ale samotný kód je ještě psaný ve starší verzi a tím vzniká několik nekompatibilit.

   Jak jsem již zmiňoval výše, nová verze Flutteru podporuje mechanismus *null-safety*. Tomu je ovšem třeba přizpůsobit i samotný kód mého programu psaný jazykem *Dart*. Dělat celý tento proces ručně by zabralo mnoho drahocenného času. Naštěstí jsou změny v kódu nutné k tomuto kroku často velmi předvídatelné, takže lze použít k tomu určený nástroj *Dart migration tool*. Stačí do konzole zadat příkaz *dart migrate*. [26]

   První zadání příkazu není úspěšné, protože mám stále ještě starší verze knihoven, které je třeba pro úspěšný převod aktualizovat.

4. **Kontrola knihoven** Stávající verzi knihoven a zda podporují *null-safety* zjistím příkazem *flutter pub outdated –mode=null-safety*, tak jak je to vidět na obrázku. Knihovny s fajfkou podporují *null-safety*. [24]

5. **Povýšení knihoven** Pokud tedy vidím, že všechny používané knihovny podporují *null-safety* stačí je jen příkazem *flutter pub upgrade –nullsafety* převést na jejich nejnovější verzi. Celý tento proces proběhne automaticky. Ovšem v případě, kdy zjistím, že některá z knihoven *null-safety* nepodporuje je vhodné se obrátit na tvůrce dané knihovny, aby tento problém u své knihovny nahradil. Případným řešením může také být její úplné nahrazení jinou knihovnou. [26]

   Toto je i případ aplikace matikadokapsy. Jedna z knihoven generující klávesnici pro zadávání číselných výsledků nepodporuje *null-safety* a pro úspěšný převod jí bylo

třeba nahradit knihovnou jinou. Co se týče ostatních knihoven, tak jejich povýšení bylo úspěšné.

Po aktualizaci knihoven hlásí vývojové prostředí tři problémy. Prvním je, že neprobíhá automatické generování čísla posledního commitu. Aplikace má totiž funkci, že v levém dolním rohu toto číslo zobrazuje. Tento problém vyřešíme ručním zadáním příkazu, který toto napraví. Vydaná aplikace už toto generuje sama a není nutné nic ručně zadávat.

Druhým problémem je použití starší již nepodporované třídy *WhitelistingTextInputFormatter*. Tato třída byla již ve verzi 1.2 nahrazena třídou *FilteringTextInputFormatter*. [27]

A nakonec špatné použití konstruktoru u knihovny FirebaseAnalytics. Zde jsem problém vyřešil pouze dopsáním *.instance* při generování nové instance.

Po úspěšné aktualizaci všech knihoven by již opětovné provedení příkazu *dart migrate* mělo proběhnout bez obtíží. V terminálu vyskočí odkaz na web, kde už jen zkontroluji změny a následně je už jen potvrdím tlačítkem *Apply Migration* aplikaci provedených změn. Následně opět opravím chyby, které nám hlásí vývojové prostředí.

Nakonec po úspěšném provedení všech změn a oprav by se aplikace měla po zadaní příkazu *flutter run* zkompilovat a spustit. Funkčnost mechanismu *null-safety* potvrdí výpis v konzoli: **Running with sound null-safety**. To je znamení o úspěšném převedení aplikace na nejnovější verzi a úspěšném uplatnění mechanismu *null-safety*.

Package Name	Current	Upgradable	Resolvable	Latest
direct dependencies:				
cloud_firestore	✗0.14.4	✗0.14.4	✓1.0.1	✓1.0.1
cupertino_icons	✗1.0.0	✓1.0.2	✓1.0.2	✓1.0.2
firebase_analytics	✗6.3.0	✗6.3.0	✓7.1.1	✓7.1.1

Obr. 4: Možný výpis použitých knihoven
[24]

5.2 Generátory matematických úloh

5.2.1 Zdroje matematických úloh

Na začátek se zmíním, že struktura a obtížnost úloh není náhodně vybírána, ale vždy odpovídá požadavkům na to, co by mělo dítě v dané třídě a části roku zvládat a co by se zrovna mělo učit ve škole. Pro to se využívá určité schéma úloh vytvořené pedagogem. Ten zase vychází z rámcového vzdělávacího plánu stanoveného MŠMT.

5.2.2 Probírané učivo

Rámcově se také dotknu témat, kterých se úlohy týkají. Tedy témat, které postupně děti během prvního stupně ZŠ probírají.

První třída začíná seznámením s číslicemi od 0 do 20 a s tím souvisejících matematických operací jako sčítání a odčítání, případně porovnávání a aplikace těchto poznatků na řešení jednouchých slovních úloh. Ke znalostem by mělo též patřit rozeznání základních matematických symbolů a fundamentálních fyzikálních jednotek. [28]

S přechodem do druhé třídy se kromě opakování učiva prvního ročníku přistupuje k počtům až do stovky. Mezi operacemi se nově objevuje zaokrouhlování a témata se také dotýkají násobení a dělení. S čímž také souvisí zvládnutí malé násobilky do pěti. [28]

Ve třetí třídě již přichází přechod i k vyšší násobilce. Zároveň se zde objevují už i složitější úlohy na číselné řady probrané v minulých ročnících. Co se týče rozšíření čísel, tak se zde počítá až do tisíce. [28]

Dítě ve čtvrté třídě již zvládá složitější matematické operace s čísly do tisíce. Některé jednodušší operace dokonce i zpaměti. Rozšíření počtu je zde až do milionu. Navíc by dítě mělo zvládnout také práci s kalkulátorem a jednoduchými grafickými závislostmi. [28]

Závěrem prvního stupně, tedy v páté třídě, se žáci seznámí se zlomky a desetinnými čísly a jejich praktickou aplikací na příkladech. Sám žák již sestaví jednoduchou grafickou závislost. Navíc se zde objevuje první seznámení s úlohami s jednou neznámou v zadání. [28]

5.2.3 Přiřazení úloh

Prvním krokem před přistoupením k samotnému řešení úloh je nutné vůbec přiřadit dítěti správnou úlohu s odpovídající obtížností. K tomu slouží volba na úvodní obrazovce, kde dítě nebo rodič zvolí, ve kterém ročníku se dítě nachází, a který měsíc aktuálně probíhá. Na základě této volby poté program sestaví souřadnici konkrétního bodu. Tyto si můžeme představit tak, že ročník určuje řádek v mapě a měsíc určuje sloupec. Mapa, o které se zmiňuji je ve skutečnosti matice složená z indexů úloh. Její rozměry jsou 5x10 prvků. Tedy bude obsahovat 5 řádků, protože sestavujeme úlohy pro 1. až 5. ročník ZŠ a 10 sloupců, protože školní rok má typicky 10 měsíců. Začíná v září, které tvoří i první sloupec a pokračuje až do června, který tvoří zase poslední sloupec.

Samozřejmě se v každém měsíci probírá více než jeden typ a obtížnost úlohy. Ovšem určitým ročníkem a měsícem přiřadíme vždy jen určitou úlohu a můžeme takto vytvořit pouze omezený počet kombinací. Z toho důvodu je v matici úloh vždy jen jedna úloha z daného měsíce. Pokud poté dítě úlohu správně vyřeší bude dotázáno, zda chce pokračovat na náročnější úkol nebo si pouze chce zopakovat obdobnou úlohu pouze s jinými hodnotami. Při volbě těžší úlohy program vybere úlohu s o jedna vyšším indexem. Pokud volba padne

na stejnou úlohu, tak generátor vytvoří stejnou úlohu jen s jinými čísly.

Stejný princip platí i opačným směrem. V případě, že si dítě nebude vědět s úlohou rady může zvolit přechod na jednodušší typ. Zde se opět program posune o index níže.

Protože aplikace je stále ve fázi vývoje a úloh, které musí pokrýt celou škálu učiva, kterým si děti musí projít je velký, tak i úloh je mnoho a né všechny jsou v kódu implementovány. Z toho důvodu zde vystupuje také rozhodovací funkce, která v případě, že hledaný index úlohy není ještě k dispozici vybere úlohu o index nižší.

Ještě musím zmínit, že každá úloha má svůj specifický index, podle kterého je identifikována. Indexy jsou dvojího typu. Jeden je číselný, kde čísla jsou seřazena vzestupně dle obtížnosti. Druhý je poté složen typicky ze tří písmen. Pro vytvoření mapy zmíněné výše se používají číselné indexy.

5.2.4 Generátory

Všechny úlohy jsou tvořeny v zásadě velmi podobným způsobem. V první řadě je definována třída *Level*, která má své nezbytné parametry pro správné vytvoření úlohy. Nejdříve jde o zadání správného indexu, a to jak číselného, tak písemného. Pro příklad uvedu, že číselný index může být *2* a písemný *btx*.

V rámci třídy *Level* poté figuruje funkce *onGenerate*. Ta již tvoří konkrétní číselné hodnoty v úloze. Ty definujeme podle zadání úlohy jako náhodnou volbu z požadovaného rozmezí. Pokud tedy budu chtít vytvořit úlohu pro sčítání dvou čísel v rozsahu 0-6, tak zadefinuji dvě proměnné *x* a *y*, kterým náhodnou volbou z rozmezí přiřadím konkrétní hodnotu. Je nutné mít však na paměti, že rozmezí nemůže být libovolné, protože se s výsledkem nechci dostat přes 6. Z toho důvodu bude pro první hodnotu *x* možný interval 0-5, ale pro druhou *y* již pouze od 1 do 6-x.

Dále je třeba třídě přiřadit i správnou tzv. *masku*. Ta již určuje konečnou podobu úlohy. Tedy zda bude dítě doplňovat výsledek sčítání nebo výsledek již bude znám, ale bude třeba doplnit některý ze sčítanců. Maska může jako v příkladu na obr.5 vypadat *x+y=Z*. Kde malá písmena značí viditelná čísla, která se budou sčítat a velké písmeno značí hledanou doplňovanou hodnotu.

Nakonec se u úlohy objevuje její slovní popis a vzorový příklad, jak takový typ úlohy může vypadat. Za příklad mohu uvést *"Sčítání v číselném oboru 0-6"* a vzorem může být *"4+2=?"*.

Nakonec ještě zmíním, že pro vytvoření dalšího typu úlohy lze použít již existující úlohu, jejíž parametry byly již definovány dříve. V případě, že se nová úloha týká stejného rozmezí hodnot, ale místo zadávání výsledku budu chtít zadávat jeden ze sčítanců, převezmu všechny vlastnosti předchozí úlohy a zadefinuji pouze jinou *masku*.

```
List<Level> levelBuilder() {
  return [
    Level(
      index: 2,
      xid: "btx",
      onGenerate: () {
        int x = randomMinMax(0, 5);
        int y = randomMinMax(1, 6 - x);
        return [[x, y, x + y], [y, x, x + y]][random(1)];
      },
      masks: ["x+y=Z"],
      valueRange: [0, 6],
      description: "Sčítání v číselném oboru 0 - 6.",
      example: "4 + 2 = ?",
    ),
```

Obr. 5: Příklad generátoru úlohy pro sčítaní
[24]

6 Závěr

Tato práce se zabývá možnostmi vývoje multiplatformních aplikací s prvky specifickými pro framework Flutter. Zdůraznil bych především dvě vlastnosti, které Flutter odlišují od ostatních vývojových prostředí. Jde o používání widgetů, tedy jakýchsi stavebních kostek pro celou aplikaci a vykreslování vizuální části na plátno (canvas), čímž je docíleno velmi plynulého chodu aplikace. Dále se práce dotýká dalších stěžejních aspektů spojených s programováním jako jsou návrh aplikace, její testování nebo licence s ní spojené. Praktická část se již věnovala konkrétním úkolům aplikovaných na skutečné aplikaci zaměřené na výuku matematiky dětí prvního stupně ZŠ. Prvním úkolem bylo povýšení zdrojového kódu na nejnovější stabilní verzi. Během povýšení bylo potřeba vyřešit aktualizaci knihoven a související migraci kódu. Poslední krok se věnoval analýze generátorů, které zajišťují vytvoření zadání dané matematické úlohy. Zde jsem věnoval pozornost celému procesu od výběru úlohy dle ročníku a aktuálního měsíce až po tvorbu konkrétní úlohy dané obtížnosti a na dané téma. Finálním výstupem mé práce jsou aktualizované části kódu doplněné o připomínky pro budoucí vývojáře společně s návrhem generátorů několika dalších typů úloh, které mohou být rovnou implementovány.

Seznam použité literatury a zdrojů

[1] *Edukids - kvalitní digičas* [online]. [cit. 2022-03-23]. Dostupné z: https://www.edukids.cz/mobilni-aplikace

[2] ZIMA, Vratislav. *Jak probíhá vývoj mobilních aplikací krok po kroku?* [online]. Praha, 25. 6. 2020 [cit. 2022-04-26]. Dostupné z: https://synetech.cz/cs/blog/vyvoj-mobilnich-aplikaci-krok-po-kroku

[3] PECINOVSKÝ, Rudolf. OOP: naučte se myslet a programovat objektově. Brno: Computer Press, 2010. ISBN 978-80-251-2126-9.

[4] PECINOVSKÝ, Rudolf. *Návrhové vzory: [33 vzorových postupů pro objektové programování]*. Brno: Computer Press, 2007. ISBN 978-80-251-1582-4.

[5] BECK, Kent. *Programování řízené testy.* Praha: Grada, 2004. Moderní programování. ISBN 9788024709017.

[6] BIESSEK, Alessandro. *Flutter for beginners: an introductory guide to building cross-platform mobile applications with Flutter and Dart 2* [online]. Birmingham: Packt, [2019] [cit. 2022-03-28]. ISBN 978-178-8990-523.

[7] ŠTĚDROŇ, Bohumír. *Ochrana a licencování počítačového programu.* Praha: Wolters Kluwer Česká republika, 2010. Právní monografie (Wolters Kluwer ČR). ISBN 978-80-7357-555-7.

[8] What is MIT Licence.*Snyk.io* [online]. [cit. 2022-02-07]. Dostupné z: https://snyk.io/learn/what-is-mit-license/

[9] Creative Commons. *Creativecommons.org* [online]. Los Angeles [cit. 2022-02-08]. Dostupné z: https://creativecommons.org/licenses/?lang=cs

[10] Mobilní aplikace.*Tarifomat.cz*[online]. Praha [cit. 2022-02-08]. Dostupné z: https://tarifomat.cz/slovnik-pojmu/mobilni-tarify/mobilni-aplikace/

[11] Choosing the Best Mobile App Framework. *Codetraveler.io* [online]. [cit. 2022-02-08]. Dostupné z: https://codetraveler.io/bestmobileframework/

[12] *FAQ Flutter* [online]. [cit. 2022-03-23]. Dostupné z: https://docs.flutter.dev/resources/faq

[13] Flutter - vývoj multiplatformních aplikací nejen pro Android a iOS (Jan Rohan). Youtube [online]. 26.3.2021 [cit. 2022-02-10]. Dostupné z: https://www.youtube.com/watch?v=ef8_xLuBaeY. Kanál uživatele InstallFest

[14] *Co je to BSD licence?* [online]. [cit. 2022-02-14]. Dostupné z: https://it-slovnik.cz/pojem/bsd-licence

[15] *Licences* Licences [online]. [cit. 2022-03-23]. Dostupné z:

http://www.gnu.org/licenses/

[16] HOCHMUTH, Tomáš. *Vývoj mobilních aplikací.* 2014. PhD Thesis. AMBIS vysoká škola, as.

[17] LANG, PRÁCE BC JIŘÍ. MULTIPLATFORMNÍ MOBILNÍ APLIKACE. 2014.

[18] DAGNE, Lukas. Flutter for cross-platform App and SDK development. 2019.

[19] CHYTIL, Jiří a Zdeněk LEHOCKÝ.*Teorie programování - verze softwaru* [online]. In: . 30. 6. 2005 [cit. 2022-04-06]. Dostupné z: http://programujte.com/clanek/2005070702-teorie-programovani-verze-softwaru/

[20] BLISCHAK, John D.; DAVENPORT, Emily R.; WILSON, Greg. A quick introduction to version control with Git and GitHub. PLoS computational biology, 2016, 12.1: e1004668.

[21] INFOGRAFIKA: Návštěvnost webů z mobilních zařízení v roce 2020. In: Martindomes.cz [online]. 21. 7. 2021 [cit. 2022-05-06]. Dostupné z: https://www.martindomes.cz/infografika-navstevnost-webu-z-mobilnich-zarizeni-v-roce-2020/

[22] PACHUCY, Arkadiusz. How to Upgrade Your Mobile App to Flutter 2.0+ [online]. 2. 8. 2021 [cit. 2022-05-06].

[23] KODYTEK, Samuel. Lekce 6: - Typový systém: Null safety v Kotlin. *Itnetwork.cz* [online]. [cit. 2022-05-07]. Dostupné z: https://www.itnetwork.cz/kotlin/zaklady/typovy-system-null-safety-v-kotlin

[24] LARSEN, Felix. Update your Flutter project to Flutter 2.0. *Felixlarsen.com* [online]. 2021, 12. 3. [cit. 2022-05-07]. Dostupné z: https://www.felixlarsen.com/blog/update-your-flutter-project-to-flutter-20

[25] HICKSON, Ian. Flutter build release channels: Flutter's channels. *Github.com* [online]. 2021, 10. 1. [cit. 2022-05-07]. Dostupné z: https://github.com/flutter/flutter/wiki/Flutter-build-release-channels

[26] Migrating to null-safety. *Dart.dev* [online]. [cit. 2022-05-07]. Dostupné z: https://dart.dev/null-safety/migration-guide

[27] Deprecated API removed after v2.5. *Dart.dev* [online]. [cit. 2022-05-07]. Dostupné z: https://docs.flutter.dev/release/breaking-changes/2-5-deprecations

[28] *Učební osnovy - 1. stupeň* [online]. [cit. 2022-05-30]. Dostupné z: http://www.zskomenskeho-skutec.cz/ke_stazeni/Ucebni_osnovy_1stupen.pdf

Seznam obrázků

Seznam obrázků

Seznam použitého SW

- Texmaker
- MiKTeX (LaTeX)
- Flutter
- Visual Studio Code
- Android Studio