Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Cross-platformová aplikace pro ovládání PLC inteligentního domu

Bc. Jakub Smolík

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Jakub Smolík

Informatika

Název práce

Cross-platformová aplikace pro ovládání PLC inteligentního domu

Název anglicky

Cross-platform application for controlling a smart home PLC

Cíle práce

Cílem práce je navrhnout a implementovat prototyp cross-platformové aplikace pro ovládání PLC inteligentního domu přes jeho aplikační rozhraní. Prototyp bude náhradou stávajících různých webových klientů, které se musí vytvářet pro každé jednotlivé PLC. Cross-platformová aplikace bude vylepšením směrem do kompatibility na různých mobilních platformách a také univerzality oproti stávajícím pouze webovým řešením závislých na jednotlivých PLC.

Metodika

Práce se bude skládat ze dvou částí, z teoretické a praktické. Teoretická část se bude zakládat na analýze a rešerši odborných zdrojů použitých v praktické části práce. V praktické části bude dokumentovám návrh a vývoj funkčního prototypu cross-platformové aplikace pro web, iOS a Android. Bude dodržován standard UML. Na základě syntézy teoretických a praktických poznatků budou zpracovány závěry diplomové práce.

Doporučený rozsah práce

60-80 stran

Klíčová slova

Flutter; cross-platformový vývoj aplikace; webová aplikace; mobilní aplikace; iOS; Android; iOS; Dart; ovládání inteligentního domu

---

Doporučené zdroje informací

Dart documentation [online]. Dostupné z: https://dart.dev/guides
Flutter documentation [online]. Dostupné z: https://docs.flutter.dev
FORD, Neal, et al. (2021) Software Architecture – The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures. First Edition, O'Reilly.
HARTSON, H. Rex, and PARDHA S. Pyla (2019) The UX Book: Agile UX Design for a Quality User Experience. Second edition, Morgan Kaufmann.
Nástroj WebMaker [online]. Dostupné z: https://www.tecomat.cz/modules/DownloadManager/download.php?alias=txv00328_01_mosaic_webmaker_cz

---

Předběžný termín obhajoby

2022/23 LS – PEF

Vedoucí práce

doc. Ing. Vojtěch Merunka, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 7. 3. 2023

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 13. 3. 2023

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 31. 03. 2023

Čestné prohlášení

Prohlašuji, že svou diplomovou práci Cross-platformová aplikace pro ovládání PLC inteligentního domu jsem vypracoval(a) samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor(ka) uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31.03.2023 _____

Poděkování

Rád bych touto cestou poděkoval panu doc. Ing. Vojtěchu Merunkovi, Ph.D.
za ochotnou výpomoc s prací a konzultace.

Cross-platformová aplikace pro ovládání PLC inteligentního domu

Abstrakt

Diplomová práce se zabývá návrhem a implementací prototypu cross-platformové aplikace pro ovládání PLC inteligentního domu přes jeho aplikační rozhraní. Prototyp je náhradou stávajícího řešení, tedy vícero webových klientských aplikací. Stávající řešení zahrnuje pro každé PLC a jeho web server samostatnou klientskou aplikaci. Prototyp není plným nahrazením s plnou funkcionalitou klientské aplikace pro PLC, umožňuje ovládat pouze vybraná zařízení přístupná aplikačním rozhraním PLC. Cross-platformová aplikace je vylepšením směrem do kompatibility na mobilních platformách iOS a Android a na různých velikostech obrazovky pro grafické rozhraní aplikace webové.

Klíčová slova: Flutter, cross-platformový vývoj aplikace, webová aplikace, mobilní aplikace, iOS, Android, Dart, ovládání inteligentního domu

Cross-platform application for controlling a smart home PLC

Abstract

The thesis deals with the design and implementation of a prototype cross-platform application for controlling a smart home PLC via its application interface. The prototype is a replacement of the existing solution, i.e. multiple web client applications. The existing solution includes a separate client application for each PLC and its web server. The prototype is not a full replacement with full functionality of the client application for the PLC, it allows to control only selected devices provided by the PLC application interface. The cross-platform app is an improvement towards compatibility on iOS and Android mobile platforms and different screen sizes for the app GUI.

Keywords: Flutter, cross-platform app development, web application, mobile application, iOS, Android, Dart, smart home control

Obsah

Seznam obrázků

Seznam tabulek

Seznam kódů

Seznam použitých zkratek

PLC	Programovatelný logický automat
IEC	Mezinárodní elektrotechnická komise
XML	Obecný značkovací jazyk
RGB	Barevný model; červená, zelená, modrá
IP	Internetový protokol
MAC adresa	Jednoznačný identifikátor síťového zařízení
XSLT	Transformace sloužící k převodům zdrojových dat ve formátu *XML* do libovolného jiného požadovaného formátu
CSS	Kaskádové styly první úrovně
CSS2	Kaskádové styly druhé úrovně
DOM	Objektový model dokumentu
SDK	Sada vývojových nástrojů
open source	Software s otevřeným zdrojovým kódem
framework	Aplikační rámec

JIT	Kompilace Just-in-time, také jako dynamický překlad
AOT	Kompilace Ahead-of-time, také jako kompilace v předstihu
VM	Virtuální stroj
garbage collector	Automatická správa paměti
embedder	Jednoúčelový počítač zabudován do ovládaného zařízení
engine	Základní součást komplexního softwarového systému
API	Aplikační programové rozhraní
Skia	2D grafická knihovna s otevřeným zdrojovým kódem
widget	Komponenta v aplikačním rámci Flutter
webová aplikace	Aplikační program, který je uložen na vzdáleném serveru a doručován přes internet prostřednictvím rozhraní prohlížeče
HTTP	Internetový protokol určený pro komunikaci s WWW servery
MVC	Softwarová architektura Model-view-controller
ABI	Application binary interface, nízkoúrovňové rozhraní
UIKit	Apple knihovna komponent pro interakci uživatele
Cocoa	Sada objektově orientovaných *frameworků*, které zajišťují běhové prostředí (runtime environment) pro aplikace macOS a iOS
Metal	Nízkoúrovňové, hardwarově akcelerované grafické *API* pro Apple platformy
OpenGL	Průmyslový standard specifikující multiplatformní rozhraní pro tvorbu aplikací počítačové grafiky
HTML	Hypertext Markup Language, značkovací jazyk
Canvas	Prostor pro vykreslování
SVG	Značkovací jazyk a formát souboru popisující dvojrozměrnou vektorovou grafiku pomocí *XML*
WebAssembly	Webový standard definující binární formát a odpovídající pseudo-jazyk symbolických adres pro přenositelný strojový kód spustitelný na webových stránkách
MIT licence	Druh svobodné licence
BSD-3 – Clause	Druh svobodné licence

pull-to-refresh	Gesto na obrazovce spočívající v přetažení obrazovky prstem nebo ukazovacím zařízením a následném uvolnění jako signál pro obnovení obsahu
URL	Uniform Resource Locator, řetězec znaků, který slouží k přesné specifikaci umístění zdrojů informací na internetu
stub soubor	Část kódu používaná k zastupování některých dalších programovacích funkcí
ASCII	American Standard Code for Information Interchange; standard kódování znaků pro elektronickou komunikaci
Digitální domorodec	Digitální domorodec je člověk narozen po roce 1980 včetně generace Z. Celý život využívá digitální technologie jako internet, mobilní telefony, videohry, televizi, MP3 přehrávače apod. Má k nim odmalička přístup a ví, jak je používat. Narodil se do období globalizace. (Prensky, 2001)
backend software	Část počítačové aplikace nebo kódu programu, ke kterému nemá uživatel přístup
router	Kód obstarávající navigování uživatele v aplikaci
CPU	Centrální procesorová jednotka
GPU	Grafický procesor
WebGL	*JavaScript API* pro nativní zobrazování interaktivní 3D grafiky

1 Úvod

Inteligentní domy (smart home) mají nainstalovanou řídící jednotku *PLC*. Tato jednotka obhospodařuje veškeré senzory a ovládací prvky, kterými je dům osazen. *PLC* jsou programována, aby funkcionalita senzorů a ovládacích prvků byla zarámována a poskytovala automatizaci a vzdálený přístup k periferiím pro uživatele domu.

Funkcionalita *PLC* je uživateli zpřístupněna skrze klientskou aplikaci, její grafické rozhraní. Uživatel aplikaci využívá jak z webové platformy, tak z platformy mobilní. Vyžaduje tedy, aby aplikace byla kompatibilní jak v internetovém prohlížeči s výstupní obrazovkou různé velikosti, tak v mobilním zařízení. Podle celosvětových statistik podílu mobilních operačních systémů na trhu (StatCounter, 2023) tvoří 72% Android, vyvinut společností Google, 27% iOS, vyvinut společností Apple Inc. a 1% zbylé – Samsung, KaiOS, Windows. Dominantními a neopomenutelnými operačními systémy pro mobilní zařazení, které by aplikace měla podporovat, jsou tedy Android a iOS.

Z pohledu vývoje klientské aplikace by aplikace měla být univerzální a fungující na jednom prostředí pro každou platformu. Případná údržba, nebo vývoj bude tak méně nákladný a vůbec možný. Univerzálnost aplikace mezi více *PLC*, tedy inteligentními domy, bude poskytovat možnost verzování, tudíž přehled nad jednotlivými historickými úpravami klientské aplikace.

Diplomová práce se bude zabývat návrhem a implementací prototypu cross-platformové aplikace pro ovládání *PLC* inteligentního domu skrze jeho aplikační rozhraní. Prototyp bude náhradou stávajících webových klientů hoštěných na jednotlivých *PLC*. Pro vývoj cross-platformové aplikace bude použito *SDK* Flutter v programovacím jazyce Dart.

V první části práce, části teoretické, bude čtenář seznámen s technologií řešení stávajícího, technologií pro vývoj cross-platformové aplikace s *SDK* Flutter, se sdílenými doplňky pro *SDK* Flutter, s objektově orientovaným programováním, se standardem UML, se sémantickým verzováním a s navrhováním uživatelského rozhraní.

V druhé části, části praktické, bude dokumentována analýza stávajícího řešení, návrh aplikační architektury nového řešení, návrh diagramu tříd ve standardu UML pro nové řešení, návrh uživatelského rozhraní nového řešení a použití sdílených balíčků v *SDK* Flutter pro implementaci prototypu cross-platformové klientské aplikace komunikující s *PLC* inteligentního domu přes jeho aplikační rozhraní.

2 Cíl práce a metodika

2.1 Cíl

Cílem práce je navrhnout a implementovat prototyp cross-platformové aplikace pro ovládání PLC inteligentního domu přes jeho aplikační rozhraní. Prototyp bude náhradou stávajících různých webových klientů, které se musí vytvářet pro každé jednotlivé PLC. Cross-platformová aplikace bude vylepšením směrem do kompatibility na různých mobilních platformách a také univerzality oproti stávajícím pouze webovým řešením závislých na jednotlivých PLC.

2.2 Metodika

Práce se bude skládat ze dvou částí, z teoretické a praktické. Teoretická část se bude zakládat na analýze a rešerši odborných zdrojů použitých v praktické části práce. V praktické části bude dokumentován návrh a vývoj funkčního prototypu cross-platformové aplikace pro web, iOS a Android. Bude dodržován standard UML. Na základě syntézy teoretických a praktických poznatků budou zpracovány závěry diplomové práce.

3 Teoretická východiska

3.1 Mosaic

Mosaic je programové vybavení, které se používá k tvorbě a ladění programů pro *PLC* TECOMAT® a TECOREG®, které jsou vyráběny společností Teco a.s. Kolín. Tento program je k dispozici již od roku 2000. Jeho vývoj probíhá v souladu s mezinárodní normou *IEC* EN-61131-3, která stanoví strukturu programů a programovací jazyky pro *PLC*.

Program Mosaic je dodáván jako kompletní balík nástrojů, který zahrnuje všechny nejrůznější funkce. Pokud není po instalaci k dispozici hardwarový klíč, Mosaic funguje v režimu *Lite*, zahrnující plnou simulaci a je vhodný pro výuku a testování všech funkcí a možností. Verze *Lite* umožňuje programovat i nejmenší *PLC* z řady TECOMAT®, a to bez omezení funkčnosti níže popsaných nástrojů. Pro větší typy *PLC* je nutný hardwarový klíč, který umožňuje deklarovat více vstupních/výstupních modulů.

Mosaic může být nainstalován na libovolný počet počítačů. Nové verze programu jsou vydávány několikrát ročně a obvykle zahrnují nové funkce a možnosti pro programování nových typů *PLC* vyráběných společností Teco a.s. S důrazem na zpětnou kompatibilitu je možné používat projekty vytvořené v předchozích verzích programu v nových verzích. Zvýšení verzí, včetně nových funkcí a nástrojů, jsou zdarma.

Nejnovější verze programu Mosaic je ke stažení na webu společnosti Teco a.s. v české, anglické, ruské a německé verzi. Jazykovou verzi lze kdykoliv přepnout v menu *Nástroje/Výběr jazyka*. (Teco a.s., 2010)

3.1.1 Programování *PLC* podle normy *IEC* 61 131-3

V kapitole bylo čerpáno z dokumentace o programování dle normy IEC 61 131-3 v prostředí Mosaic (Teco a.s., 2007). Mezinárodní norma *IEC* 61 131 pro programovatelné řídící systémy se skládá z pěti základních částí, které obsahují souhrn požadavků na moderní řídící systémy. Tato norma je nezávislá na konkrétní organizaci či firmě a má širokou mezinárodní podporu. Jednotlivé části normy se zaměřují na technické i programovací vybavení těchto systémů. Jedná se o důležitý nástroj pro vývojáře a výrobce těchto systémů, kteří chtějí zajistit, aby jejich produkty byly v souladu s mezinárodními standardy a normami.

Norma *IEC* 61 131-3 definuje programovací jazyky pro průmyslovou automatizaci a tvoří tak první vážný krok ke standardizaci programování v tomto odvětví. Norma byla vytvořena sedmi mezinárodními společnostmi, které využily svou desetiletou zkušenost na

poli průmyslové automatizace. Její obsah zahrnuje asi 200 stran textu a 60 tabulek a byla vytvořena pracovní skupinou SC65B WG7 mezinárodní standardizační organizace *IEC*. Výsledkem práce je specifikace syntaxe a sémantiky unifikovaného souboru programovacích jazyků, včetně obecného softwarového modelu a strukturujícího jazyka. Tato norma byla přijata jako směrnice většinou výrobců *PLC*.

3.1.2 Nástroj WebMaker

Veškerý zdrojový kód pro výsledný uživatelský program lze vytvořit v textové podobě. Pro zjednodušení práce a minimalizaci chyb nabízí prostředí Mosaic různé nástroje, které některé činnosti usnadňují a následně automaticky generují výsledný zdrojový kód. Některé z těchto nástrojů pracují obousměrně, což znamená, že lze kód zapisovat jak v textové, tak v grafické podobě. Příkladem takového nástroje je *IEC* manažer. Ostatní nástroje pracují pouze jednosměrně a generují tak zdrojový kód automaticky. Tyto soubory jsou označeny ikonou v seznamu souborů pro překlad v projektu a nelze je editovat v textové podobě, protože mají nastaven atribut *read only* a vždy se obnovují podle nastavení daného nástroje. (Teco a.s., 2010)

WebMaker slouží k vytváření *XML* stránek pro webový server, který je podporován v centrálních jednotkách (*PLC*) a základních modulech. Tento nástroj je také vhodný pro zobrazování a nastavování proměnných přímo v Mosaic. Kromě toho může být použit pro jednoduchou vizualizaci a ladění algoritmu v simulaci v Mosaic. WebMaker se spouští kliknutím na ikonu a implicitně se zobrazuje v hlavním panelu. (Teco a.s., 2010)

Okno nástroje se skládá ze tří částí: horní nástrojová lišta; levý panel stránek a skupin; editační plocha.

3.1.2.1 *Horní lišta*

Horní nástrojová lišta obsahuje:

- „Tužka/Brouk – Přepíná režim editace (tužka) a režim ladění (brouk). V režimu editace je možné přidávat a měnit skupiny, stránky, objekty a jejich vlastnosti.
- Symbol šipka <XML> – Zkompilovat webové stránky – Generování *XML* kódu pro *PLC*. Vygenerovaný *XML* kód se ukládá v projektu do podadresáře *SendRoot*. Tento adresář je automaticky synchronizován s *PLC* při vyslání kódu programu (*PLC* musí podporovat souborový systém, jinak k synchronizaci nedojde).“ (Teco a.s., 2013)

- „List s lupou – Náhled – Vygenerovaný kód se otevře z disku ve webovém prohlížeči. Tyto stránky neobsahují reálná data z *PLC* a slouží pouze pro kontrolu vzhledu a odkazů.
- Disketa – Uložit – Uloží rozpracovaný stav bez generování *XML* kódu. Rozpracovaný stav se automaticky ukládá při generování *XML* kódu a při zavírání nástroje.
- Šipka zpět – Vrací jednu změnu provedenou v editační ploše.
- Šipka vpřed – Vrací jednu změnu vrácenou *šipkou zpět*.
- Kurzorová šipka – Vybrat – Zapíná a indikuje režim výběru. V tomto režimu je možné vybírat prvky na editační ploše. Také je možné tažením měnit jejich pozici nebo u vybraných prvků i velikost.
- Písmeno A – Statický text – Zapíná režim vložení statického textu. Tento text může sloužit i jako odkaz.
- Zadávací pole – Zadávací pole – Zapíná režim vložení zadávacího pole. Zadávací pole slouží k zobrazení a/nebo editaci proměnných *PLC*. Zeditované hodnoty jsou odeslány do *PLC* po stisknutí globálního odesílacího tlačítka.
- Zadávací pole s tlačítkem – Zadávací pole s vlastním tlačítkem pro odeslání – Zapíná režim vložení zadávacího pole s vlastním tlačítkem pro odeslání. Zadávací pole slouží k zobrazení a/nebo editaci proměnných *PLC*. Zeditované hodnoty jsou z webových stránek odeslány do *PLC* po stisknutí odesílacího tlačítka u zadávacího pole, globální odesílací tlačítko na tyto pole nemá vliv.
- Tlačítko *OK* – Odesílací tlačítko pro pole bez vlastního tlačítka – Zapíná režim vložení globálního odesílacího tlačítka. Toto tlačítko po stisknutí odešle všechny zadávací pole na stránce bez vlastních tlačítek pro odeslání. Pokud toto tlačítko není na webové stránce přítomné, není možné hodnoty ze zadávacích polí do *PLC* odeslat!
- Obdélník – Obdélník – Zapíná režim vložení jednobarevného objektu obdélníkového tvaru sloužící pro rozčlenění plochy s ovládacími prvky. Pomocí podmíněné viditelnosti lze objekt využít i pro signalizaci.
- Částečně vyplněný obdélník – Sloupec ovládaný proměnnou – Zapíná režim vložení objektu obdélníkového tvaru sloužící pro vyjádření hodnoty šířkou nebo výškou sloupce. Na panelech grafických panelech ID-18/28 lze zobrazit sloupec stínovaný gradientem. Ve webovém prohlížeči je takový objekt zobrazen jako obdélník.“ (Teco a.s., 2013)

- „Obdélník s proměnnou barvou – Obdélník s barvou ovládanou proměnnou – Zapíná režim vložení objektu obdélníkového tvaru pro zobrazení barvy definované proměnnou se složkami *RGB*.
- Dvojice obrázků – Dvoustavový obrázek – Zapíná režim vložení dvoustavového obrázku. Dvoustavový obrázek slouží k ovládáním a zobrazování proměnných datového typu BOOL (pravda/nepravda). Při nenulové hodnotě je zobrazen jeden obrázek, při nulové druhý. Při kliknutí na obrázek je hodnota proměnné negována. Pro jiné typy proměnných je po nulové hodnotě nastavena hodnota 1 a po nenulové hodnotě 0.
- Tlačítko *SET* – Prvek pro nastavení hodnoty proměnné – zapíná režim vložení prvku pro nastavení hodnoty proměnné. Prvek slouží k jednorázovému nastavení přednastavené hodnoty přidružené proměnné. Prvek má přiřazené dva obrázky, první se zobrazuje v klidovém stavu, druhý je zobrazen při nastavení hodnoty tak, aby poskytoval zpětnou vazbu od stisknutí prvku.
- Filmový pás – Vícestavový obrázek – Zapíná režim vložení vícestavového obrázku. Vícestavový obrázek slouží na zobrazení stavu grafickým symbolem. Celočíselná řídící proměnná může nabývat hodnot od 0 (zobrazí první obrázek) do hodnoty o jedna menší než je počet obrázků (zobrazí poslední obrázek). Pokud má proměnná hodnotu mimo tyto meze, není zobrazen žádný z obrázků.
- Obrázek – Statický obrázek – Zapíná režim vložení statického obrázku. Prvek slouží k zobrazení statického obrázku. Tento obrázek může sloužit i jako odkaz.
- Fotoaparát – Obraz z *IP* kamery – Zapíná režim vložení obrazu z *IP* kamery. Prvek slouží k periodickému načítání obrazu produkovaného *IP* kamerou. Tento prvek může být na každé stránce pouze jeden. Prvek není aktivní v simulaci v prostředí Mosaic.
- Složka se šipkou – Pole pro odesílání souborů – Zapíná režim vložení pole pro odesílání souborů. Toto pole umožňuje odeslat přes webovou stránku souboru do *PLC*. Grafická podoba prvku se může lišit dle použitého prohlížeče.
- Obrázek se šipkou – Obrázek řízený řetězcem - Zapíná režim vložení objektu pro zobrazení obrázku, jehož umístění je specifikováno proměnnou datového typu STRING (řetězec znaků). Takto lze zobrazovat jak obrázky uložené lokálně na paměťové kartě, tak i obrázky z jiných webových serverů. „ (Teco a.s., 2013)

- „Skupina objektů – Seznam objektů – Otevírá okno se seznamem objektů na stránce, kde lze objekty vybírat a modifikovat.
- Objekty nad sebou – Vrstvy – Otevírá́ okno se seznamem vrstev. Každý́ objekt může být přirazen do jedné́ ze šestnácti vrstev. Tato vrstva pak může být v tomto dialogu zmrazena proti úpravám nebo skryta.
- Složka s obrázky – Správce obrázků – Otevírá okno pro přidávání obrázků do projektu. Tyto obrázky můžou být následně použity na stránkách.
- Složka se zátržítky – Společná́ nastavení – Otevírá okno s globálními nastavením pro celý projekt.
- Zvonek – Nastavení alarmů – Otevírá́ okno pro nastavení stránek, které se mají zobrazit v případě, že řídící proměnná́ je nenulová.
- Hlava s klíčem – Nastavení přístupu – Otevírá okno pro zadání hesel a *MAC* adres pro přístup bez přihlašování.
- Vlajky – Nastavení jazyků – Otevírá okno pro definici jazykových mutací. Přidružená nabídka umožňuje export a import textů.
- Písmeno *i* – Informace o verzi nástroje.“ (Teco a.s., 2013)

3.1.2.2 *Levý panel stránek a skupin*

Levý panel stránek a skupin obsahuje:

- „Složka s listem a plus – Přidat skupinu – Přidá další skupinu za aktuální vybraný uzel. Skupina slouží k logickému rozdělení stránek.
- List s plus – Přidat stránku – Přidá další stránku za aktuální vybraný uzel.
- List s mínus – Vymazat – Vymaže skupinu nebo stránku.
- Modrá šipka nahoru – Posunout nahoru – Posune vybranou skupinu nebo stránku před stránku nebo skupinu, která ji předchází.
- Modrá šipka dolu – Posunout dolu – Posune vybranou skupinu nebo stránku za stránku nebo skupinu, která ji následuje.“ (Teco a.s., 2013)

3.1.2.3 *Editační plocha*

V editační ploše se zobrazuje stránka, která je vybrána v levém panelu. Pokud je vybrána skupina, zobrazuje se na editační ploše první stránka ze skupiny. Na začátku je editační plocha prázdná, zobrazuje se pouze obrys stránky s ohledem na cílové rozlišení určené v

obecných nastaveních a rastr 8x8 bodů, ke kterému se přichytávají objekty. Na plochu lze vkládat objekty z lišty nástrojů. (Teco a.s., 2013)

3.1.2.4 *Správce obrázků*

Nástroj *Správce obrázků* slouží ke správě obrázků v projektu. Jeho nástrojová lišta umožňuje uživateli přidat obrázek z domovského adresáře nebo z uživatelského adresáře, přičemž si nástroj pamatuje poslední použitou cestu. Dále umožňuje odebrat z projektu vybraný obrázek. Pod nástrojovou lištou je zobrazen seznam všech obrázků dostupných v projektu a vpravo se nachází náhled právě vybraného obrázku. Pod plochou náhledu může uživatel editovat dlouhé jméno obrázku bez omezení a také jméno, pod kterým bude obrázek uložen v systému. Informace o výšce a šířce jsou zobrazeny vedle náhledu obrázku. Tlačítko *Vypustit nepoužité obrázky* nabízí všechny obrázky, které nejsou použity v objektech projektu, a umožňuje jejich odstranění. Změny provedené v dialogu se potvrzují tlačítkem *OK*. (Teco a.s., 2013)

3.1.2.5 *Společná nastavení*

Dialog s názvem *Společná nastavení* umožňuje určit velikost plochy, kterou bude zobrazovací zařízení používané prohlížečem poskytovat (tato velikost určuje rovněž velikost editační plochy). Uživatel si může ručně nastavit výšku a šířku, nebo nechat software odhadnout tyto rozměry na základě rozlišení daného zařízení. Pokud uživatel zvolí cílové zařízení ID-18/28, budou kromě velikosti stránky nastaveny i další volby specifické pro toto zařízení. (Teco a.s., 2013)

3.1.2.6 *Nastavení přístupu*

Na kartě *Nastavení hesel* je možné nastavit až deset kombinací uživatelského jména a hesla, které budou vyžadovány pro přístup k webserveru prostřednictvím webového prohlížeče. Každá kombinace může mít přiřazenu úroveň přístupu (vyšší číslo znamená vyšší práva) a výchozí stránku, která se zobrazí po přihlášení, pokud uživatel nezadá jinou adresu. Pokud se úrovně nastaví na *-1*, pak je kombinace uživatelského jména a hesla neplatná.

Výchozí nastavení je takové, že všechny jména a hesla jsou nastavena jako neplatná, takže je nutné alespoň jednu kombinaci povolit, aby se uživatel mohl k webserveru přihlásit. Je doporučeno, aby se nepoužívala kombinace uživatelského jména a hesla se stejnými čísly (0 až 9), protože neposkytují dostatečnou úroveň zabezpečení.

Pokud uživatel nevyplní jméno a heslo, může to být platná kombinace, což znamená, že se lze přihlásit bez zadání jména a hesla.

Na kartě *Nastavení přístupu* je možné zadat až deset *MAC* adres zařízení a jejich úroveň přístupu. Po zadání těchto adres nebude vyžadováno přihlášení. Každé adrese lze také přiřadit výchozí stránku.

Úroveň přístupu vyjadřuje práva přihlášeného uživatele. Uživatel může zobrazit a editovat všechny objekty, které jsou na stejné nebo nižší úrovni než jeho vlastní. Objekty na vyšší úrovni může uživatel pouze zobrazit, na stránky vyšší úrovně nemůže uživatel přistoupit a ani je nevidí v menu. Skupiny vyšší úrovně také nejsou v menu viditelné. (Teco a.s., 2013)

3.1.2.7 *Nastavení multijazyčnosti*

Nastavení jazyků umožňuje spravovat více jazykových verzí v rámci jednoho projektu. Pokud tuto funkci zapneme, je možné přiřadit až patnáct popisů ke každému objektu. Při procesu překladu se následně vygeneruje verze popisu, která je označena přepínačem *Aktivní jazyk*. (Teco a.s., 2013)

3.1.2.8 *Minimální požadavky na prohlížeč*

„Pro správné zobrazení webových stránek je nutné, aby prohlížeč splňoval následující požadavky:

- podpora *XSLT*
- podpora kaskádových stylů *CSS2*
- podpora jazyka JavaScript pro stránky s periodicky obnovovanými proměnnými (zejména objekt *XMLHttpRequest* a přístup ke stránce pomocí *DOM*.“ (Teco a.s., 2013)

3.1.3 Knihovna CanvasObjectsLib

Knihovna CanvasObjectsLib nabízí podporu pro manipulaci s grafickými objekty, které jsou dostupné prostřednictvím webového serveru v *PLC*. Tato knihovna je standardně součástí programovacího prostředí Mosaic od verze v2016.1 a pokud chceme využít její funkce v aplikačním programu *PLC*, musíme ji do projektu nejprve přidat. Kromě knihovny CanvasObjectsLib se do projektu automaticky přidají také knihovny StdLib, ColorLib,

CrcLib, CanvasLib a ToStringLib, protože knihovna CanvasObjectsLib využívá některé funkce z těchto knihoven.

Je třeba poznamenat, že knihovna CanvasObjectsLib není podporovaná v systému TC-650 a u systému TC700 nelze použít s procesorovými moduly CP-7002, CP-7003 a CP-7005. Avšak funkce z knihovny CanvasObjectsLib jsou podporovány v centrálních jednotkách řady K a L (TC700 CP-7004 a CP-7007, všechny varianty systému Foxtrot) od FW verze v9.6. (Teco a.s., 2016)

3.1.3.1 *Principy práce knihovny*

Knihovna CanvasObjectsLib nabízí funkční bloky pro tvorbu a zobrazení různých grafických objektů v rámci web stránky vytvořené v nástroji WebMaker v prostředí Mosaic. Mezi podporované objekty patří například čárové a sloupcové grafy, posuvníky a další. Každý funkční blok v této knihovně generuje sadu příkazů pro nakreslení příslušného objektu, které jsou ukládány do zásobníku (bufferu). Tento zásobník se pak použije jako řídící struktura pro prvek *Kreslící plátno* (canvas) vložený do web stránky.

Když je tedy například potřeba zobrazit čárový graf zobrazující teploty během dne, je potřeba vytvořit instanci funkčního bloku *fbLineGraph1*, který na základě pole s teplotami *temp_samples* vygeneruje potřebné příkazy pro nakreslení grafu, a uloží je do svého výstupu *canvasData*. Poté se v nástroji WebMaker vloží do web stránky prvek *Kreslící plátno* a nastaví se jeho řídící struktura na výstup funkčního bloku *fbLineGraph1*. (Teco a.s., 2016)

3.1.4 **Knihovna CanvasLib**

V knihovně jsou funkce a bloky, které umožňují programu v *PLC* vytvářet grafiku, která se zobrazí na webových stránkách. Tato grafika může být dynamická, což umožňuje kreslit grafy, indikátory hodnoty a další prvky, které nelze kreslit standardními objekty WebMakeru. Při kreslení jsou povelům a parametrům ukládány do speciálního bufferu, který je po dokončení přenesen do zobrazovacího zařízení. Výsledný obrázek je kreslen z těchto povelů až v zobrazovacím zařízení.

Používání kreslicích funkcí může prodloužit dobu cyklu *PLC*, takže je doporučeno volat je pouze tehdy, když je potřeba změnit obrázek. Cyklické kreslení stejné grafiky zbytečně zatěžuje *PLC* a přenosový kanál do zobrazovacího zařízení. Pokud přestaneme volat grafické funkce, grafický buffer si pamatuje poslední vykreslený obrázek a přenáší ho podle potřeby do zobrazovacích zařízení.

Pro použití funkcí z knihovny CanvasLib v aplikaci *PLC* je třeba nejprve tuto knihovnu přidat do projektu. Knihovna je součástí prostředí Mosaic od verze v2014.5 a lze ji použít na všech centrálních jednotkách řady Foxtrot, CP-7004 a CP-7007 od verze 8.8. (Teco a.s., 2015)

3.1.4.1 ***Datové typy***

Základním datovým typem pro grafické operace v knihovně CanvasLib je *TCanvasData*. Po dokončení kreslení se tento buffer automaticky přenese do zobrazovacího zařízení. Velikost grafického bufferu se zvyšuje s rostoucí složitostí zobrazované grafiky. Výchozí velikost bufferu v *TCanvasData* je 8195 bytů, ale pro lepší využití paměti *PLC* se doporučuje používat menší buffery.

Příliš velký buffer zbytečně zabírá místo a zpomaluje odezvu webové stránky. Celková velikost bufferů všech prvků na jedné webové stránce je omezena na 8192 bytů.

Kromě bufferu obsahuje struktura *TCanvasData* další položky pro interní potřeby knihovny CanvasLib, se kterými se uživatel nemusí zabývat. Výjimku tvoří položka *touch* (dotek), kam se ukládají souřadnice a barva bodu, na který uživatel klikl. Tuto funkci lze využít pro tvorbu různých ovládacích prvků, které reagují na kliknutí nebo výběr barvy, pokud předem nakreslíme obrázek s barevnou paletou. Souřadnice uložené v položce *touch* jsou automaticky přepočítány s ohledem na nastavení počátku souřadnicového systému a virtuální rozměr kreslicí plochy pomocí funkcí *GC_SetOrigin* a *GC_SetVirtSize* na konci kreslení. (Teco a.s., 2015)

3.1.4.2 ***Konstanty***

V knihovně CanvasLib se používají konstanty pro nastavení vlastností a vzhledu kreslených prvků. Konstanty *GCOLOR_...* určují základní barvy a přidáním konstanty *GCOLOR_TRANSPARENT* lze vytvořit průhlednou barvu.

Konstanty *GTEXT_STYLE_...* ovlivňují způsob, jakým se text vykresluje. V tabulkách jsou uvedeny významy jednotlivých konstant.

Konstanty lze kombinovat pomocí sčítání, například kombinace *GTEXT_STYLE_BOLD + GTEXT_STYLE_ITALIC + GTEXT_STYLE_FILLBG* vypíše text tučně kurzívou s vyplněným podkladem. (Teco a.s., 2015)

3.2 Cross-platformová aplikace

Při vývoji mobilní/webové aplikace je pravděpodobně cílem oslovit co nejvíce uživatelů tím, že se stejná aplikace poskytuje na různých platformách. K tomu existují dva hlavní přístupy. Vývoj aplikace pro každou požadovanou platformu, nebo vytvoření aplikace jednou a její spuštění na více platformách (s jedním zdrojovým kódem). První alternativa může znamenat větší časovou náročnost při vývoji, kdy je potřeba se naučit a osvojit si vývojová prostředí pro různé platformy. Tento problém vede ke konceptu řešení pro vývoj mobilních aplikací napříč platformami. Cílem cross-platformového řešení je vyvinout aplikaci jednou a spustit ji na více platformách (cross-platform). Odtud pochází rčení „napiš jednou, spusť kdekoli".

Cross-platformová řešení často fungují tak, že je vytvořena abstrakční (virtuální) vrstva nad platformou. Abstrakční vrstva obvykle využívá nativní knihovny platformy, například knihovny uživatelského rozhraní. Snaží se zachovat konzistentní vzhled pro zvolenou platformu. Kód aplikace cross-platformových řešení je často napsán v interpretovaném jazyce, například v jazyce JavaScript. Kód musí komunikovat se základními nativními knihovnami platformy, aby bylo možné zobrazit uživatelské rozhraní. (Haider, 2021)

3.3 Flutter

Flutter je cross-platformové *SDK* pro uživatelské rozhraní, které je navrženo tak, aby umožňovalo opakované použití kódu napříč operačními systémy, jako jsou iOS, Android a web, a zároveň umožňovalo aplikacím přímé propojení se službami platformy. Cílem je umožnit vývojářům vytvářet vysoce výkonné aplikace, které budou na různých platformách působit přirozeně a budou zohledňovat rozdíly, pokud existují, a zároveň sdílet co nejvíce kódu.

Během vývoje se o běh aplikace Flutter stará virtuální stroj, který nabízí stavové načítání změn *Hot reload* bez nutnosti úplné rekompilace. Při vydání jsou aplikace Flutter kompilovány přímo do strojového kódu, ať už s instrukcemi Intel x64 nebo ARM, nebo do JavaScriptu, pokud jsou zaměřeny na web. *SDK* je *open source* s povolenou licencí *BSD* a má prosperující ekosystém doplňků třetích stran, které doplňují základní funkce knihoven.

Flutter je používán jazykem Dart, který je optimalizován pro rychlé aplikace na libovolné platformě. (Google LLC, 2023)

3.3.1 **Jazyk Dart**

Dart je klientsky optimalizovaný jazyk pro vývoj rychlých aplikací na libovolné platformě. Jeho cílem je nabídnout nejproduktivnější programovací jazyk pro vývoj na více platformách ve spojení s flexibilní platformou pro spouštění aplikačních rámců.

Jazyk je definován svou technickou obálkou – volbami učiněnými během vývoje, které utvářejí možnosti a silné stránky jazyka. Dart je navržen pro technický balíček, který je vhodný pro klientský vývoj a upřednostňuje jak vývoj (sub-sekundové stavové načítání za chodu), tak vysoce kvalitní produkční zkušenosti v široké škále cílů kompilace (web, mobilní zařízení a počítače). (Google LLC, 2023)

3.3.1.1 ***Vlastnosti jazyku***

Jazyk Dart je typově bezpečný – používá statickou typovou kontrolu, která zajišťuje, že hodnota proměnné vždy odpovídá jejímu statickému typu. Někdy se tomu říká *zdravé* typování. Ačkoli jsou typy povinné, typové anotace jsou kvůli typové inferenci nepovinné. Systém typování jazyku Dart je také flexibilní a umožňuje použití dynamického typu v kombinaci s kontrolou za běhu, což může být užitečné při experimentování nebo pro kód, který musí být obzvláště dynamický.

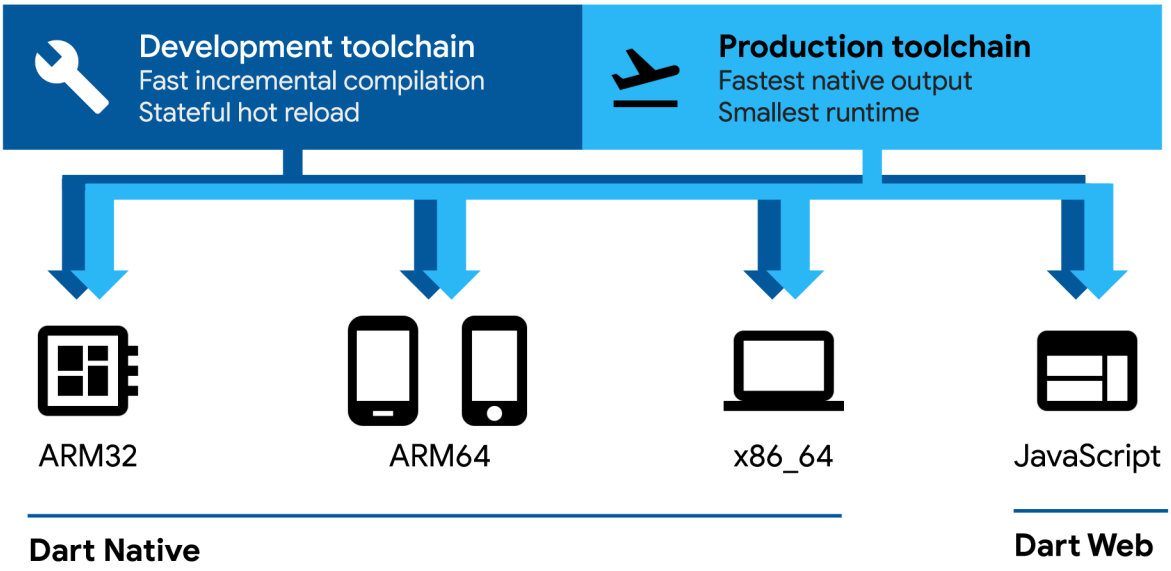
Dart nabízí solidní bezpečnost nulových hodnot, což znamená, že hodnoty nemohou být nulové, pokud neřeknete, že mohou být. Díky bezpečnosti nulových hodnot Dart může chránit vývojáře před výjimkami z nulových hodnot za běhu pomocí statické analýzy kódu. Na rozdíl od mnoha jiných nulově bezpěčných jazyků, pokud Dart určí, že proměnná není nulovatelná, je tato proměnná vždy nenulovatelná. Pokud zkontrolujete svůj běžící kód v ladiči, uvidíte, že nenulovatelnost je za běhu zachována. (Google LLC, 2023)

3.3.1.2 ***Platformy***

Technologie překladače Dart umožňuje spouštět kód různými způsoby:

Nativní platforma: Pro aplikace určené pro mobilní a stolní zařízení obsahuje Dart jak virtuální počítač Dart s kompilací *JIT*, tak kompilátor *AOT* pro tvorbu strojového kódu.

Webová platforma: Pro aplikace určené pro web lze Dart kompilovat pro vývojové nebo produkční účely. Jeho webový kompilátor překládá Dart do jazyka JavaScript. (Google LLC, 2023)



Obrázek 1 - Platformy Dart (Google LLC, 2023)

3.3.1.3 *Nástroj Dart Native*

Během vývoje je pro iteraci klíčový rychlý vývojový cyklus. Dart *VM* nabízí *JIT* kompilátor s inkrementální rekompilací (umožňující *hot reload*), *live metrics collections* (podporující DevTools – nástroje pro vývojáře) a bohatou podporu ladění.

Když jsou aplikace připraveny k nasazení do produkce – ať už jsou publikovány do obchodu s aplikacemi, nebo nasazovány na produkční server – kompilátor Dart *AOT* je může zkompilovat do nativního strojového kódu ARM nebo x64. Aplikace zkompilovaná pomocí *AOT* se spouští s konzistentní a krátkou dobou spouštění.

Kód zkompilovaný pomocí *AOT* běží uvnitř běhového prostředí (runtime environment) Dart, které vynucuje pěvně typový jazyk Dart a spravuje paměť pomocí rychlé alokace objektů a *garbage collectoru*. (Google LLC, 2023)

3.3.1.4 *Nástroj Dart Web*

Dart Web umožňuje spouštět kód Dart na webových platformách poháněných jazykem JavaScript. Pomocí nástroje Dart Web je kód Dart zkompilován do kódu JavaScript, který je následně spustitelný v internetovém prohlížeči.

Dart Web obsahuje dva režimy kompilace:

- inkrementální vývojový kompilátor umožňující rychlý vývojový cyklus,

- optimalizační produkční kompilátor, který kompiluje kód Dart do rychlého, kompaktního a nasaditelného jazyka JavaScript. Tato efektivita vychází z technik, jako je eliminace mrtvého kódu. (Google LLC, 2023)

3.3.1.5 *Běhové prostředí (runtime environment) Dart*

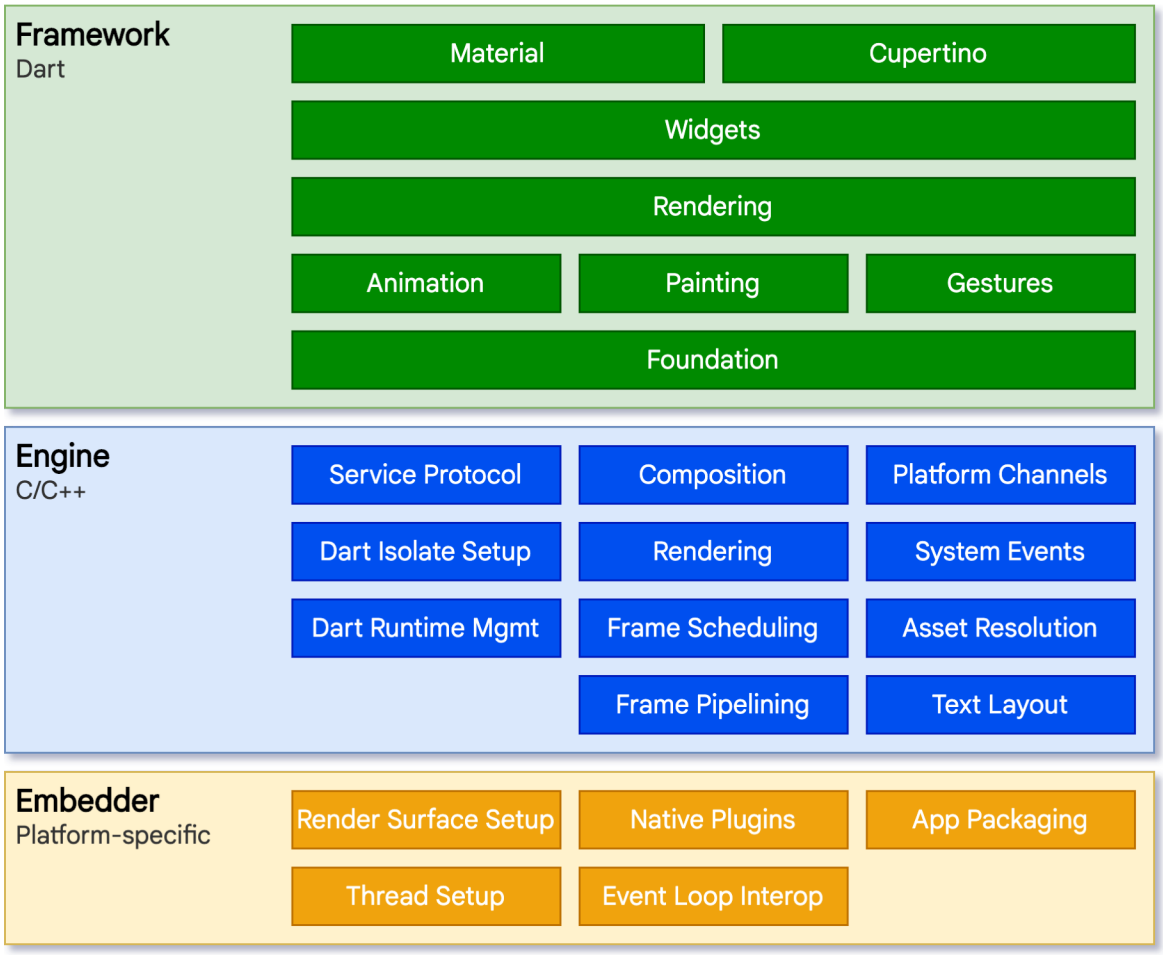
Bez ohledu na to, jaká platforma je použita, nebo jak je kód kompilován, je vyžadováno spuštění kódu běhového prostředí Dart. Toto prostředí je zodpovědné za následující úkoly:

- správa paměti: Dart používá model spravované paměti, kde je nevyužitá paměť získávána zpět pomocí *garbage collectoru*.
- Vynucování typového systému Dart: přestože většina typových kontrol v jazyce Dart je statická (v době kompilace), některé typové kontroly jsou dynamické (za běhu). Například běhové prostředí Dart vynucuje dynamické kontroly pomocí operátorů typových kontrol a přetypovávání.
- Správa izolátů: Běhové prostředí Dart řídí hlavní izolát (kde kód běží) a všechny další izoláty, které aplikace vytvoří.

Na nativních platformách je běhové prostředí Dart automaticky zahrnuto uvnitř samostatných spustitelných souborů a je součástí virtuálního počítače Dart poskytovaného příkazem *dart run*. (Google LLC, 2023)

3.3.2 **Architektura *SDK* Flutter**

Flutter je navržen jako rozšiřitelný, vrstvený systém. Existuje jako řada nezávislých knihoven, z nichž každá závisí na základní vrstvě. Žádná vrstva nemá privilegovaný přístup k vrstvě nižší a každá část úrovně *frameworku* je navržena tak, aby byla volitelná a nahraditelná. (Google LLC, 2023)



Obrázek 2 - Vrstvy SDK Flutter (Google LLC, 2023)

Pro základní operační systém jsou aplikace vyvinuté s *SDK* Flutter zabaleny stejně jako jakékoli jiné nativní aplikace. Vkládací modul specifický pro danou platformu poskytuje vstupní bod – koordinuje se základním operačním systémem přístup ke službám, jako jsou vykreslovací plochy, přístupnost a vstup, a spravuje smyčku událostí zpráv. *Embedder* je napsán v jazyce, který je vhodný pro danou platformu: v současné době je to Java a C++ pro Android, Objective-C/Objective-C++ pro iOS a macOS a C++ pro Windows a Linux. Pomocí *embedderu* lze Dart kód z *SDK* Flutter integrovat do stávající aplikace jako modul nebo může být kód celým obsahem aplikace. *SDK* Flutter obsahuje řadu *embedderů* pro běžné cílové platformy, existují ale i další *embeddery*.

Jádrem aplikace vyvinuté s *SDK* Flutter je Flutter *engine*, který je většinou napsán v jazyce C++ a podporuje primitiva nezbytná pro podporu všech aplikací vyvinuté s *SDK* Flutter. Tento *engine* je zodpovědný za rasterizaci složených scén, kdykoli je třeba vykreslit nový snímek. Poskytuje nízko úrovňovou implementaci základního rozhraní *API SDK* Flutter, včetně grafiky (prostřednictvím *Skia*), rozvržení textu, souborových a síťových

vstupů a výstupů, podpory přístupnosti, architektury zásuvných modulů a běhového a kompilačního řetězce nástrojů jazyka Dart.

Tento *engine* je vystaven *SDK* Flutter prostřednictvím knihovny *dart:ui*, která obaluje základní kód v jazyce C++ třídami Dart. Tato knihovna vystavuje primitiva nejnižší úrovně, jako jsou třídy pro řízení vstupních, grafických a textových vykreslovacích subsystémů.

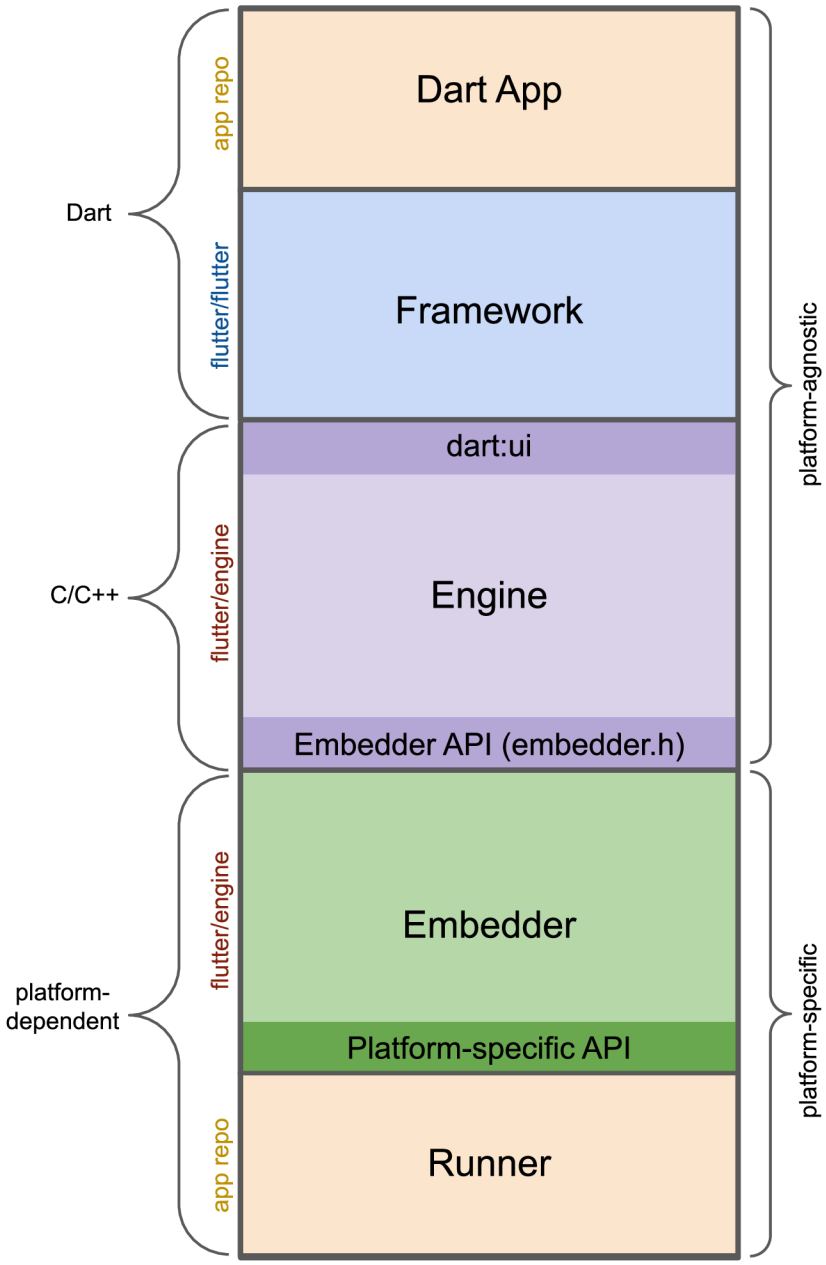
Vývojáři obvykle komunikují s *SDK* Flutter, které obsahuje bohatou sadu platformových, rozvrhovacích a základních knihoven, které se skládají z řady vrstev. Postupujeme-li zdola nahoru, máme k dispozici:

- základní třídy a služby, jako jsou animace, malování a gesta, které nabízejí běžně používané abstrakce nad základním základem.
- Vrstvu vykreslování, která poskytuje abstrakci pro práci s rozvržením grafického rozhraní. Pomocí této vrstvy můžete vytvořit strom vykreslitelných objektů. S těmito objekty můžete dynamicky manipulovat, přičemž strom automaticky aktualizuje rozvržení tak, aby se změny propsaly.
- Vrstvu *widgetů*, která je abstrakcí pro kompozici. Každý vykreslovací objekt ve vykreslovací vrstvě má odpovídající třídu ve vrstvě *widgetů*. Kromě toho vrstva *widgetů* umožňuje definovat kombinace tříd, které lze opakovaně použít. Na této vrstvě se zavádí model reaktivního programování.
- Knihovnu Material a Cupertino, které nabízejí ucelené sady ovládacích prvků, které využívají kompoziční primitiva vrstvy *widgetů* k implementaci návrhových jazyků Material nebo iOS. (Google LLC, 2023)

SDK Flutter je relativně malé; mnoho funkcí vyšší úrovně, které mohou vývojáři používat, je implementováno jako v podobě sdílených doplňků, včetně zásuvných modulů pro jednotlivé platformy, jako je kamera nebo webový náhled, a také funkcí pro jednotlivé platformy, jako jsou *HTTP* komunikace a animace. Některé z těchto doplňků pocházejí z širšího ekosystému a zahrnují služby, jako jsou platby v aplikacích, ověřování Apple. (Google LLC, 2023)

3.3.3 Rozbor aplikace vyvinuté *SDK* Flutter

Následující schéma poskytuje přehled částí, ze kterých se skládá běžná aplikace vyvinutá *SDK* Flutter vytvořená nástrojem *flutter create*. Ukazuje, kde se v tomto zásobníku nachází Flutter Engine, zvýrazňuje hranice *API* a identifikuje úložiště jednotlivých částí.



Obrázek 3 - Schéma Flutter aplikace (Google LLC, 2023)

3.3.3.1 ***Dart App***

- Skládá *widgety* do požadovaného uživatelského rozhraní.
- Implementuje obchodní (business) logiku.
- Vlastníkem je vývojář aplikace. (Google LLC, 2023)

3.3.3.2 ***Framework***

- Poskytuje *API* vyšší úrovně pro vytváření kvalitních aplikací (například *widgety*, detekce gest, přístupnost, zadávání textu).

- Skládá strom *widgetů* aplikace do scény. (Google LLC, 2023)

3.3.3.3 *Engine*

- Odpovídá za rastrování složených scén.
- Zajišťuje nízko úrovňovou implementaci základních rozhraní *API* aplikace Flutter (například grafika, rozvržení textu, běhové prostředí Dart).
- Vystavuje své funkce *frameworku* pomocí rozhraní *dart:ui API*.
- Integruje se s konkrétní platformou pomocí *API* rozhraní Embedder Engine. (Google LLC, 2023)

3.3.3.4 *Embedder*

- Koordinuje se základním operačním systémem pro přístup ke službám, jako jsou vykreslovací plochy, přístupnost a vstup.
- Spravuje smyčku událostí aplikace.
- Vystavuje rozhraní *API* specifické pro danou platformu pro integraci nástroje *embedder* do aplikace. (Google LLC, 2023)

3.3.3.5 *Runner*

- Skládá části vystavené rozhraním *API embedderu* specifickým pro danou platformu do doplňku aplikace spustitelného na cílové platformě.
- Je to součást šablony aplikace vygenerované nástrojem *flutter create*, kterou vlastní vývojář aplikace. (Google LLC, 2023)

3.3.4 Reaktivní uživatelské rozhraní

Flutter je na první pohled reaktivní, pseudodeklarativní, *framework* uživatelského rozhraní, ve kterém vývojář poskytne mapování z celkového stavu aplikace na stav rozhraní a *framework* převezme úkol aktualizace rozhraní za běhu při změně stavu. Tento model je inspirován *frameworkem* React, který s ním poprvé přišel, a který přehodnocuje množství tradičních principů návrhu.

Ve většině tradičních *frameworků* uživatelského rozhraní je počáteční stav uživatelského rozhraní popsán jednou a poté je samostatně aktualizován kódem za běhu v reakci na události. Jedním z problémů tohoto přístupu je, že s rostoucí složitostí aplikace si

vývojář musí být vědom toho, jak se změny stavu kaskádovitě promítají do celého uživatelského rozhraní.

Jedním z řešení je přístup podobný architektuře *MVC*, kdy se změny dat přenášejí do modelu prostřednictvím kontroléru a model pak prostřednictvím kontroléru přenáší nový stav do zobrazení. I to je však problematické, protože vytváření a aktualizace prvků uživatelského rozhraní jsou dva oddělené kroky, které se mohou snadno vymknout synchronizaci.

Flutter spolu s dalšími reaktivními *frameworky* k tomuto problému přistupuje alternativně, a to tak, že explicitně odděluje uživatelské rozhraní od jeho základního stavu. Pomocí rozhraní *API* ve stylu Reactu vývojář vytváří pouze popis uživatelského rozhraní a *framework* se postará o to, aby pomocí této jediné konfigurace vytvořil a/nebo aktualizoval uživatelské rozhraní podle potřeby.

Ve Flutteru jsou *widgety*, podobně jako komponenty v Reactu, reprezentovány neměnnými třídami, které se používají ke konfiguraci stromu objektů. Tyto *widgety* se používají ke správě samostatného stromu objektů pro rozvržení, který se pak používá ke správě samostatného stromu objektů pro kompozici. Flutter je ve své podstatě řada mechanismů pro efektivní procházení upravených částí stromů, převod stromů objektů na stromy objektů nižší úrovně a šíření změn napříč těmito stromy. (Google LLC, 2023)

3.3.5 Flutter *Widgety*

Flutter klade důraz na *widgety* jako na kompoziční jednotku. *Widgety* jsou stavebními kameny uživatelského rozhraní aplikace Flutter a každý *widget* je neměnnou deklarací části uživatelského rozhraní.

Widgety tvoří hierarchii založenou na kompozici. Každý *widget* se vnoří do svého rodiče a může přijímat kontext od rodiče. Tato struktura se přenáší až ke kořenovému *widgetu* (kontejneru, který hostí aplikaci Flutter, typicky *MaterialApp* nebo *CupertinoApp* dle použité knihovny).

Aplikace aktualizují své uživatelské rozhraní v reakci na události (například interakci uživatele) tak, že řeknou *frameworku*, aby nahradil *widget* v hierarchii jiným *widgetem*. *Framework* pak porovná nové a staré *widgety* a efektivně aktualizuje uživatelské rozhraní.

Flutter má vlastní implementace každého ovládacího prvku uživatelského rozhraní, místo aby se spoléhal na ty, které poskytuje systém: například existuje čistá implementace Dart ovládací komponenty *Switch* jak pro iOS, tak pro ekvivalent pro Android.

Tento přístup přináší několik výhod:

- umožňuje neomezenou rozšiřitelnost. Vývojář, který chce vytvořit variantu ovládací komponenty *Switch*, ji může vytvořit libovolným způsobem a není omezen na body rozšíření poskytované operačním systémem.
- Vyhýbá se výraznému omezení výkonu tím, že umožňuje aplikaci vyvinutou s *SDK* Flutter složit celou scénu najednou, aniž by bylo nutné přecházet mezi kódem aplikace Flutter a kódem platformy.
- Odděluje chování aplikace od jakýchkoli závislostí na operačním systému. Aplikace vypadá a působí stejně na všech verzích operačního systému, i kdyby operační systém změnil implementaci svých ovládacích prvků. (Google LLC, 2023)

3.3.6 **Vykreslovací model Flutteru**

Tradiční aplikace pro Android fungují tak, že při kreslení je nejprve volán kód *frameworku* Android v jazyce Java. Systémové knihovny Androidu poskytují komponenty zodpovědné za samotné kreslení do objektu *Canvas*, který pak Android může vykreslit pomocí *Skia*, grafického jádra napsaného v jazyce C/C++, které volá *CPU* nebo *GPU* k dokončení kreslení v zařízení.

Cross-platformové *frameworky* obvykle fungují tak, že vytvářejí abstrakční vrstvu nad základními nativními knihovnami uživatelského rozhraní systémů Android a iOS a chtějí vyhladit nesrovnalosti reprezentace jednotlivých platforem. Kód aplikace je často napsán v interpretovaném jazyce, jako je JavaScript, který musí pro zobrazení uživatelského rozhraní zase komunikovat se systémovými knihovnami Androidu založenými na jazyce Java nebo iOS založenými na jazyce Objective-C. To vše zvyšuje režii, která může být značná, zejména tam, kde dochází k velké interakci mezi uživatelským rozhraním a logikou aplikace.

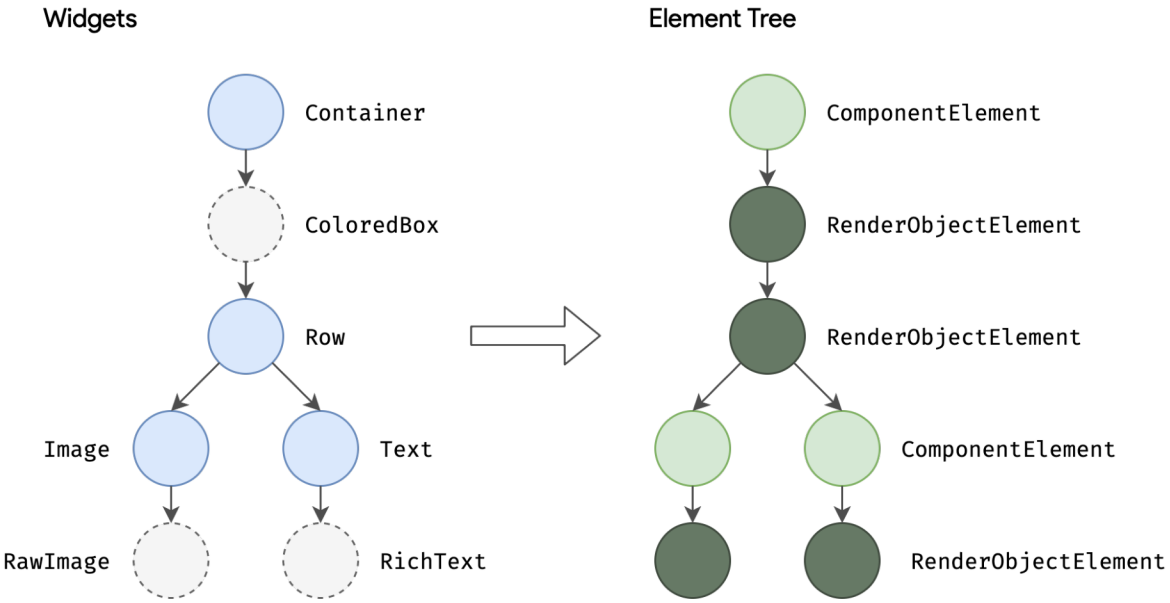
Flutter naopak tyto abstrakce minimalizuje a obchází systémové knihovny *widgetů* uživatelského rozhraní ve prospěch vlastní sady *widgetů*. Kód v jazyce Dart, který vykresluje vizuální prvky aplikace Flutter, je zkompilován do nativního kódu, který pro vykreslování používá *Skia*. Flutter také vkládá vlastní kopii *Skia* jako součást enginu, což vývojáři umožňuje aktualizovat aplikaci, aby zůstala aktualizovaná s nejnovějšími vylepšeními výkonu, i když telefon nebyl aktualizován na novou verzi systému Android. Totéž platí pro Flutter na jiných nativních platformách, jako je iOS, Windows nebo macOS. (Google LLC, 2023)

3.3.6.1 *Vykreslování fragmentu*

Když Flutter potřebuje vykreslit fragment, zavolá metodu *build*, která vrátí podstrom *widgetů*, který vykreslí uživatelské rozhraní na základě aktuálního stavu aplikace. Během tohoto procesu může metoda *build* podle potřeby zavádět nové *widgety* na základě svého stavu.

Ve fázi sestavování Flutter převádí *widgety* (*Widgets*) vyjádřené v kódu do odpovídajícího stromu prvků (*Element Tree*), přičemž pro každý *widget* je určen jeden prvek. Každý prvek představuje konkrétní instanci *widgetu* v daném místě hierarchie stromu. Existují dva základní typy prvků:

- *ComponentElement*, hostitel ostatních prvků,
- *RenderObjectElement*, prvek, který se účastní fáze rozvržení nebo malování. (Google LLC, 2023)



Obrázek 4 - Převod widgetu na strom prvků (Google LLC, 2023)

Prvky *RenderObjectElements* jsou prostředníkem mezi jejich obdobou *widgetu* a základním objektem *RenderObject*.

Na prvek libovolného *widgetu* se lze odkazovat prostřednictvím jeho *BuildContextu*, což je odkaz k umístění *widgetu* ve stromu. Jedná se o kontext ve volání funkce.

Protože *widgety* jsou neměnné, včetně vztahu rodič/potomek mezi uzly, jakákoli změna ve stromu *widgetů* způsobí vrácení nové sady objektů *widgetů*. To však neznamená, že je nutné znovu vytvořit základní reprezentaci. Strom prvků je perzistentní mezi stavy, a proto hraje rozhodující výkonnostní roli. Aplikaci Flutter se umožňuje chovat, jako by

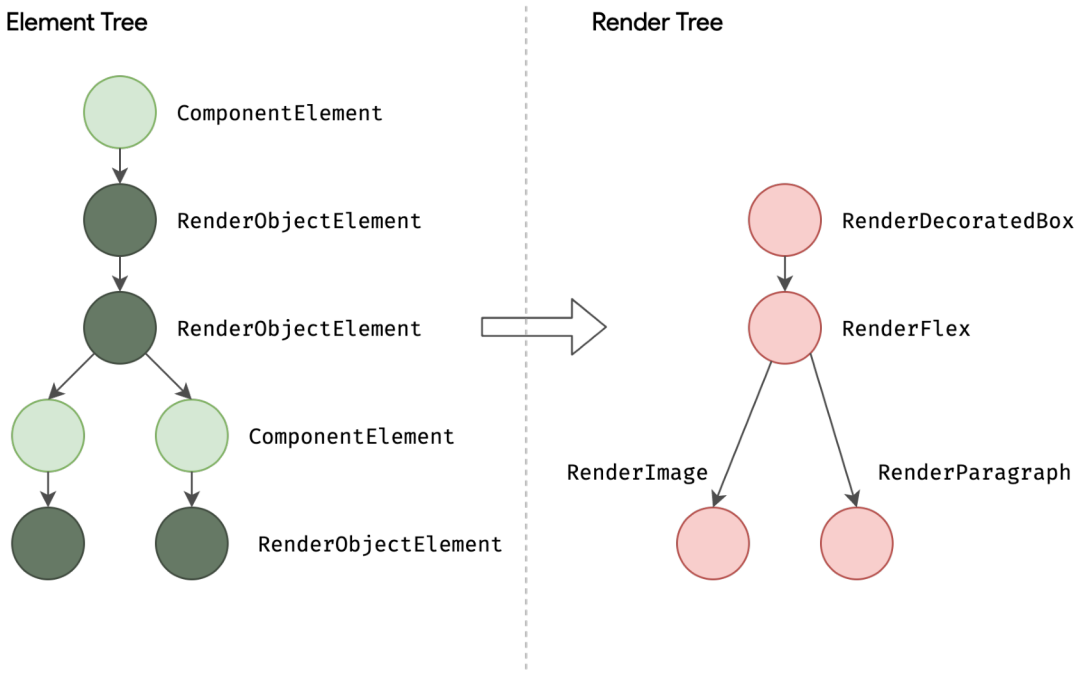
hierarchie *widgetů* měla být plně jednorázová, a zároveň umožňuje ukládat do mezipaměti její základní reprezentaci. Tím, že *SDK* Flutter prochází pouze *widgety*, ve kterých nastala změna, může znovu sestavit pouze ty části stromu prvků, které vyžadují rekonfiguraci. (Google LLC, 2023)

3.3.6.2 ***Rozvržení widgetů***

Důležitou součástí každého *frameworku* uživatelského rozhraní je proto schopnost efektivně rozvrhnout hierarchii *widgetů* a určit velikost a pozici jednotlivých prvků ještě před jejich vykreslením na obrazovku.

Základní třídou pro každý uzel ve vykreslovacím stromu *Render Tree* je *RenderObject*, který definuje abstraktní model pro rozvržení a vykreslování. Ten je nesmírně obecný: nezavazuje se k pevnému počtu rozměrů, a dokonce ani ke kartézskému souřadnému systému. Každý objekt *RenderObject* zná svého rodiče, ale o svých potomcích ví jen málo kromě toho, jak na ně přistoupit, zná jejich omezení. To poskytuje objektu *RenderObject* dostatečnou abstrakci, aby byl schopen zvládnout nejrůznější případy použití.

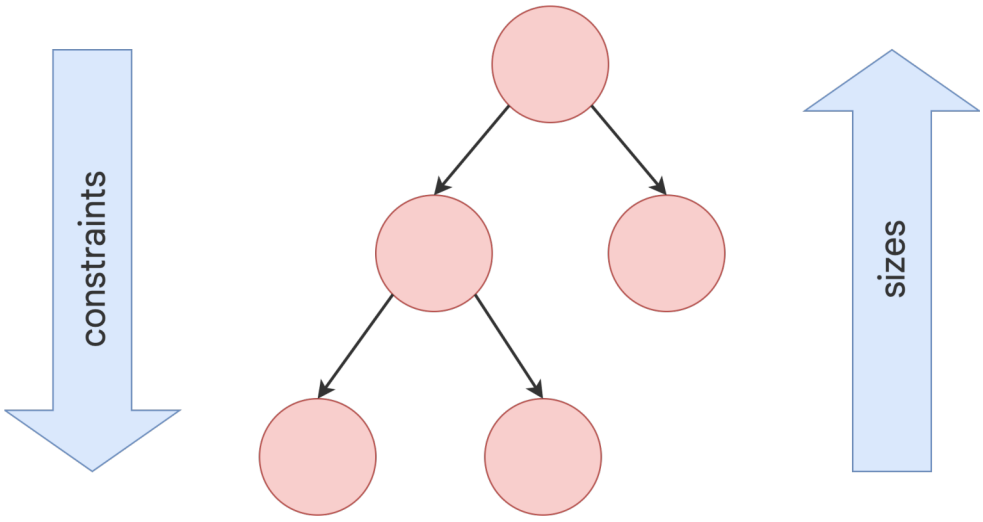
Během fáze sestavení Flutter vytvoří nebo aktualizuje objekt, který dědí od *RenderObject*, pro každý *RenderObjectElement* ve stromu prvků. *RenderObjects* jsou primitiva: *RenderParagraph* vykresluje text, *RenderImage* vykresluje obrázek a *RenderTransform* aplikuje transformaci před vykreslením svého potomka.



Obrázek 5 - Vykreslení stromu prvků (Google LLC, 2023)

Většina Flutter *widgetů* je vykreslována objektem, který dědí z podtřídy *RenderBox*, ta představuje objekt *RenderObject* pevné velikosti ve 2D kartézském prostoru. *RenderBox* poskytuje základ modelu omezení boxu, který stanovuje minimální a maximální šířku a výšku každého vykreslovaného *widgetu*.

Pro provedení rozvržení Flutter prochází strom vykreslování v hloubkovém procházení a předává omezení velikosti *constraints* z rodiče na potomka. Při určování své velikosti musí potomek respektovat omezení, která mu zadal jeho rodič. Potomci reagují předáním velikosti *sizes* nahoru svému rodičovskému objektu v rámci omezení, která rodič stanovil. (Google LLC, 2023)



Obrázek 6 - Rozvržení widgetů rodič – potomek (Google LLC, 2023)

Na konci tohoto jediného projití stromem má každý objekt definovanou velikost v rámci omezení svého rodiče a je připraven k vybarvení voláním metody *paint*.

Model je velmi výkonný jako způsob rozvržení objektů v čase:

- Rodiče mohou diktovat velikost potomka nastavením maximálního a minimálního omezení na stejnou hodnotu. Například nejvýše postavený vykreslovací objekt v mobilní aplikaci omezuje svého potomka na velikost obrazovky. Potomci si mohou vybrat, jak tento prostor využijí. Mohou například jen vycentrovat to, co chtějí vykreslit v rámci nadiktovaných omezení.
- Rodič může potomkovi diktovat šířku a poskytnout mu flexibilitu ohledně výšky (nebo diktovat výšku, ale nabídnout flexibilitu ohledně šířky). Příkladem z reálného světa je průtokový text, který se může muset vejít do horizontálního omezení, ale vertikálně se může měnit v závislosti na množství textu.

Tento model funguje i v případě, že potomek potřebuje vědět, kolik místa má k dispozici, aby se mohl rozhodnout, jak bude vykreslovat svůj obsah. Pomocí *widgetu LayoutBuilder* může podřízený objekt prozkoumat předaná omezení a na jejich základě určit, jak je využije. (Google LLC, 2023)

3.3.7 Použití *SDK* Flutter na mobilní platformě

Uživatelská rozhraní *SDK* Flutter nejsou překládána do ekvivalentních *widgetů* operačního systému, ale jsou vytvářena, rozvržena, skládána a vykreslována samotným *SDK* Flutter. Mechanismus pro získání textury a podíl na životním cyklu aplikace podkladového operačního systému se v závislosti na platformě odlišuje. Mechanismus je pro platformu agnostický a představuje stabilní rozhraní *ABI*, které poskytuje platformě určující, jak nastavit a používat *SDK* Flutter.

Platform *embedder* je nativní aplikace operačního systému, která hostí veškerý obsah Flutteru a funguje jako spojovací článek mezi hostitelským operačním systémem a Flutterem. Při spuštění aplikace Flutter *embedder* poskytne vstupní bod, inicializuje *engine* Flutter, získá vlákna pro uživatelské rozhraní a rastrování a vytvoří texturu, do které může Flutter zapisovat. *Embedder* je také zodpovědný za životní cyklus aplikace, včetně vstupních rozhraních (jako je myš, klávesnice, dotyk), nastavení velikosti okna, správy vláken a správy platformy. Flutter obsahuje platformové *embeddery* pro Android, iOS, ale také Windows, macOS a Linux. Nabízí možnost vytvořit vlastní platformový *embedder*, pro specifická zařízení.

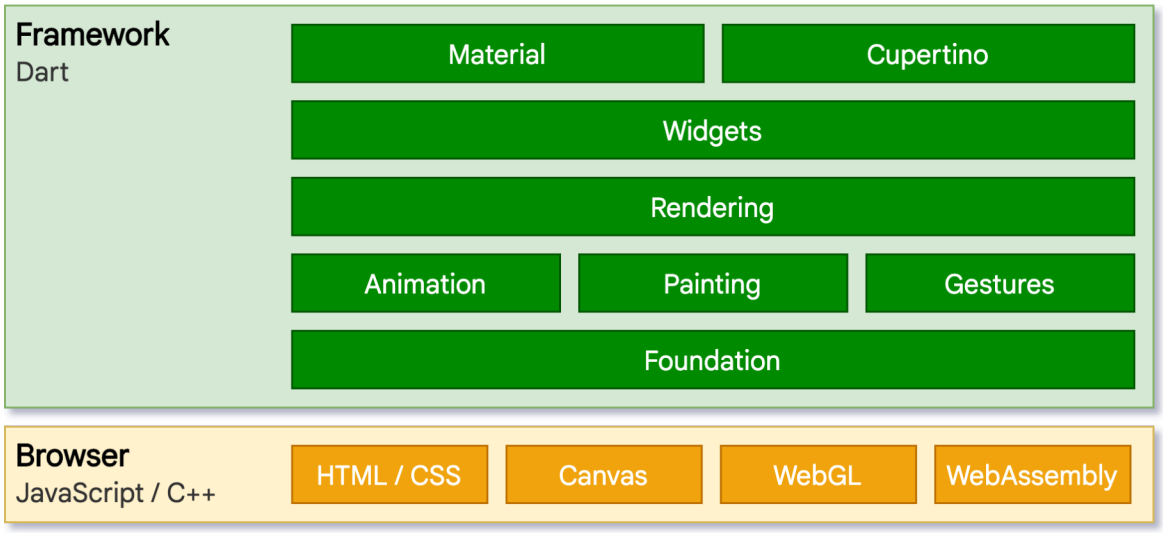
Každá platforma má vlastní sadu rozhraní *API* a omezení:

- v systémech iOS se Flutter načítá do *embedderu* jako *UIViewController*, respektive *NSViewController* (nativní komponenta iOS aplikace). Platformový *embedder* vytvoří *FlutterEngine*, který slouží jako hostitel pro Dart *VM* a Flutter runtime, a *FlutterViewController*, který se připojí k *FlutterEngine*, aby předával vstupní události *UIKit* nebo *Cocoa* do Flutteru a zobrazoval komponenty vykreslené *FlutterEngine* za pomocí *Metal* nebo *OpenGL*.
- V systému Android je Flutter ve výchozím nastavení načten do *embedderu* jako aktivita. Zobrazení je řízeno modulem *FlutterView*, který vykresluje obsah Flutteru buď jako základní zobrazovací komponentu *view*, nebo jako texturu. (Google LLC, 2023)

3.3.8 Použití *SDK* Flutter pro web

Dart se kompiluje do jazyka JavaScript již po celou dobu existence tohoto jazyka (Dart), přičemž řetězec nástrojů je optimalizován pro vývojové i produkční účely. Mnoho důležitých aplikací se dnes kompiluje z jazyka Dart do jazyka JavaScript a běží v produkčním provozu, včetně nástrojů pro inzerenty Google Ads. Protože je *SDK* Flutter napsáno v jazyce Dart, je jeho kompilace do jazyka JavaScript poměrně jednoduchá.

Engine Flutteru, napsaný v jazyce C++, je navržen tak, aby spolupracoval spíše se základním operačním systémem než s webovým prohlížečem. Proto je zapotřebí jiný přístup, na webu poskytuje Flutter reimplementaci enginu nad standardními rozhraními *API* prohlížeče. V současné době existují dvě možnosti vykreslování obsahu Flutter na webu: *HTML* a *WebGL*. V režimu *HTML* Flutter používá *HTML*, *CSS*, *Canvas* a *SVG*. K vykreslování do *WebGL* používá Flutter verzi *Skia* zkompilovanou do *WebAssembly* s názvem CanvasKit. Zatímco režim *HTML* nabízí nejlepší charakteristiky velikosti kódu, CanvasKit poskytuje nejrychlejší cestu ke grafickému zásobníku prohlížeče a nabízí poněkud vyšší grafickou shodu s nativními mobilními aplikacemi. (Google LLC, 2023)



Obrázek 7 - Flutter schéma architektury pro web (Google LLC, 2023)

Nejvýznamnějším rozdílem oproti ostatním platformám, na kterých Flutter běží, je to, že Flutter nemusí poskytovat běhové prostředí Dart. Místo toho je *framework* Flutter (spolu s veškerým napsaným kódem) zkompilován do jazyka JavaScript. Stojí také za zmínku, že Dart má ve všech svých režimech (*JIT* versus *AOT*; nativní versus webová kompilace) velmi málo sémantických rozdílů v jazyce a většina vývojářů nikdy nenapíše řádek kódu, který by na takový rozdíl narazil.

Během vývoje používá Flutter web kompilátor *dartdevc*, který podporuje inkrementální kompilaci, a proto umožňuje *hot restart* (současně bez *hot reload*). Naopak, když jste připraveni vytvořit produkční aplikaci pro web, je použit *dart2js*, vysoce optimalizovaný produkční Dart kompilátor jazyka JavaScript, který zabalí jádro a *framework* Flutter spolu s aplikací do minifikovaného zdrojového souboru, který lze nasadit na libovolný webový server. Kód lze nabídnout v jediném souboru nebo rozdělit do více souborů pomocí odložených importů. (Google LLC, 2023)

3.4 Flutter sdílené doplňky

Jednou z nejběžnějších technik sdílení kódu je použití sdílené knihovny/doplňku. Sdílená knihovna je externí artefakt obsahující zdrojový kód, který je používán více službami a který je obvykle v době kompilace sjednocen se službou. Ačkoli se technika sdílené knihovny zdá být jednoduchá a přímočará, má svůj podíl na složitostech a kompromisech, z nichž v neposlední řadě je to granularita a verzování sdílené knihovny.

Existují dvě protichůdné síly, které tvoří kompromisy u sdílených knihoven. Jsou to správa závislostí a řízení změn. (Ford, 2021)

Výhody	Nevýhody
Možnost verzování změn	Závislosti mohou být obtížně zvládnutelné
Sdílený kód je založen na kompilaci, která snižuje počet chyb při jeho běhu	Duplikace kódu v heterogenních kódových sadách
Dobrá agilita při změnách sdíleného kódu	Synchronizace verzí může být obtížná

Tabulka 1 - Kompromisy sdílených knihoven/doplňků (Ford, 2021)

SDK Flutter podporuje používání sdílených doplňků, které do Flutteru a ekosystému Dart vyvíjí vývojáři třetích stran. To umožňuje automatizování opakujících se úkonů a sdílení zpřístupnění specifických vlastností platforem.

Doplněk je přinejmenším adresář obsahující soubor *pubspec.yaml*. Kromě toho může doplněk obsahovat závislosti (uvedené v souboru *pubspec.yaml*), knihovny Dart, aplikace, zdroje, testy, obrázky, písma a příklady použití. Na webu *pub.dev* je uvedeno mnoho doplňků – vyvinutých inženýry společnosti Google a členy komunity Flutter a Dart. (Google LLC, 2023)

3.4.1 **Doplněk flutter_riverpod**

Knihovna pro správu stavu aplikace, která:

- zachycuje programové chyby v době kompilace, nikoliv za běhu programu,
- odstraňuje vnořování pro naslouchající/kombinující objekty,
- zajišťuje, že kód je testovatelný,
- *MIT licence*. (Rousselet, 2022)

Pokud je provider (předchůdce a nyní součást Riverpod) zjednodušením *InheritedWidget* (třída pro *widgety*, které efektivně šíří informace ve stromu *widgetů* (Google LLC, 2023)), pak je Riverpod reimplementací *InheritedWidget* od začátku.

V principu je velmi podobný providerovi, ale má také zásadní rozdíly, které jsou pokusem o odstranění běžných problémů, s nimiž se provider potýká.

Riverpod má několik cílů:

- být schopen bezpečně vytvářet, naslouchat a likvidovat stavy, aniž by se musel obávat ztráty stavu při obnově *widgetu*.
- Zviditelnění našich objektů ve vývojářských nástrojích *SDK* Flutter ve výchozím nastavení.
- Testovatelnost a skladatelnost.
- Zlepšení čitelnosti *InheritedWidget*, když jich máme více (což by přirozeně vedlo k hluboce vnořenému stromu *widgetů*).
- Zlepšit škálovatelnost aplikací díky jednosměrnému toku dat.

Odtud jde Riverpod ještě o několik kroků dál:

- Čtení objektů je nyní bezpečné při kompilaci.
- Vzor provider je flexibilnější, což umožňuje podporovat běžně požadované funkce, jako např:
    - možnost mít více providerů stejného typu.
    - Likvidace stavu providera, když se již nepoužívá.
    - Mít aktuální stavy.
    - Vytvoření soukromého stavu providera.
- Zjednodušuje složité grafy objektů – jednodušší závislost na asynchronním stavu.

Těchto vlastností je dosaženo tím, že již není používán *InheritedWidget*. Místo toho Riverpod implementuje vlastní mechanismus, který ale funguje podobným způsobem. (Rousselet, 2022)

3.4.1.1 *Provider*

Providery jsou nejdůležitější součástí doplňku Riverpod. Provider je objekt, který zapouzdřuje část stavu a umožňuje naslouchání tomuto stavu.

Zabalení části stavu do provideru:

- Umožňuje snadný přístup k tomuto stavu na více místech současně. Providery jsou úplnou náhradou za návrhové vzory jako *Singleton*, *Service Locator*, *Dependency Injection* nebo *InheritedWidget*.
- Zjednodušuje kombinování tohoto stavu s jinými.
- Umožňuje optimalizaci výkonu. Ať už jde o filtrování přestaveb *widgetů* nebo o ukládání výpočtů stavu do mezipaměti. Providery zajišťují, že se znovu vypočítá pouze to, co je změnou stavu ovlivněno.
- Zvyšuje testovatelnost aplikace. Díky providerům není potřeba složitých úkonů *setUp/tearDown*. Navíc lze libovolného providera přepsat tak, aby se během testu choval jinak, což umožňuje snadno testovat specifické chování.
- Umožňuje snadnou integraci s pokročilými funkcemi, jako je protokolování nebo *pull-to-refresh*. (Rousselet, 2023)

3.4.2 **Doplněk go_router**

Deklarativní směrovací (*routovací*) doplňek pro Flutter, který využívá rozhraní Router *API* a poskytuje pohodlné rozhraní *API* založené na *URL* pro navigaci mezi různými obrazovkami. Je možné definovat vzory *URL*, navigovat pomocí *URL*, zpracovávat odkazy (*deep links*) a řadu dalších scénářů souvisejících s navigací. Je vydán pod licencí *BSD-3 – Clause*.

go_router má řadu funkcí, které usnadňují navigaci:

- parsování cesty a parametrů dotazu pomocí syntaxe šablony (například "user/:id").
- Zobrazení více obrazovek pro adresu (dílčí trasy)
- Podpora přesměrování – uživatele můžete přesměrovat na jinou adresu *URL* na základě stavu aplikace, například přihlášení, pokud uživatel není ověřen.
- Podpora více navigátorů prostřednictvím *ShellRoute* – je možné zobrazit vnitřní navigátor, který zobrazí vlastní stránky na základě odpovídající trasy.
- Podpora aplikací knihoven Material i Cupertino.
- Zpětná kompatibilita s rozhraním Navigator *API*

Ve výchozím nastavení je go_router dodáván s výchozími chybovými obrazovkami pro knihovny Material i Cupertino a také s výchozí chybovou obrazovkou v případě, že není použita žádná. Výchozí chybovou obrazovku je možné nahradit vlastní. (flutter.dev, 2022)

3.4.3 **Doplněk intl**

Poskytuje možnosti internacionalizace a lokalizace, včetně překladu zpráv, množného čísla a rodů, formátování a rozboru data/čísla a obousměrného textu.

Všechny různé typy lokálních dat vyžadují asynchronní inicializační krok, který zajistí dostupnost dat. Tím se zmenší velikost aplikace, protože se načtou pouze ta data, která jsou skutečně potřebná.

Každá různá oblast internacionalizace (zprávy, data, čísla) vyžaduje samostatný inicializační proces. Tímto způsobem, pokud aplikace potřebuje pouze formátovat data, se nemusí věnovat čas ani místo načítání zpráv, čísel nebo jiných věcí, které nejsou potřebné.

U zpráv je také potřeba importovat existující soubor do doby, dokud nebude spuštěn krok generování kódu. To lze obejít vytvořením *stub souboru messages_all.dart*, spuštěním prázdného kroku překladu nebo zakomentováním importu - dokud nejsou překlady k dispozici. Doplněk je vydán pod licencí *BSD-3 – Clause*. (dar.dev, 2022)

3.4.4 **Doplněk shared_preferences**

Obaluje trvalé úložiště dat základních datových typů specifické pro danou platformu. Data mohou být persistována do úložiště asynchronně a není zaručeno, že po návratu z funkce bude zápis zdařilý. Proto se tento doplněk nesmí používat pro ukládání kritických dat.

Podporované datové typy jsou *int*, *double*, *bool*, *String* a *List<String>*.

Lokality pro ukládání dat na platformách jsou:

- *SharedPreferences* pro Android,
- *NSUserDefaults* pro iOS a macOS,
- adresář *XDG_DATA_HOME* pro Linux,
- *LocalStorage* pro web,
- adresář *roaming AppData* pro Windows.

Doplněk je vydán pod licencí *BSD-3 – Clause*. (flutter.dev, 2022)

3.4.5 **Doplněk universal_platform**

Doplněk vracející nám informaci, na které platformě je aplikace právě spuštěna. Rozpozná platformy:

- iOS,
- Android,
- web,
- macOS,
- Windows,
- Linux,
- Fuschia.

Doplněk je vydán pod licencí MIT. (gskinner.com, 2022)

3.4.6 **Doplněk web_socket_channel**

Obal *StreamChannel* (rozhraní obousměrného komunikačního kanálu) pro komunikační protokol WebSocket. Poskytuje cross-platformové *API WebSocketChannel*, která komunikuje přes základní *StreamChannel* rozhraní.

Nejdůležitější úloha třídy *WebSocketChannel* spočívá ve sloužení jako rozhraní pro proudové kanály WebSocket na všech platformách. Kromě základního rozhraní *StreamChannel* přidává metodu pro získání protokolu, který vrací vyjednaný protokol pro soket, a také metodu pro získání *closeCode* a *closeReason* poskytující informace proč byl soket uzavřen.

WebSocketChannel funguje také jako cross-platformová implementace protokolu WebSocket. Konstruktor *WebSocketChannel.connect* se připojí k naslouchajícímu serveru pomocí příslušné implementace pro danou platformu. Konstruktor *WebSocketChannel()* přebírá základní *StreamChannel*, přes který komunikuje pomocí protokolu WebSocket. Poskytuje také statickou metodu *signKey()*, která usnadňuje implementaci počátečního WebSocket handshake (tools.dart.dev, 2023) - když klient naváže spojení WebSocket se serverem, musí server provést určité kroky k přijetí spojení a odeslání počátečního serverového handshake. (Melnikov, 2011)

Doplněk je vydán pod licencí *BSD-3 – Clause*. (tools.dart.dev, 2023)

3.5 Objektově Orientované Programování

V racionálním myšlení má každý pojem své jméno a obsah. OOP (Objektově Orientované Programování) na toto použití navazuje a představuje pojmy jako třídy, konkrétně jako dvojice jejich jmen (identifikátorů) a obsahů – hodnot. Ačkoli první pokus o OOP byl požadován za seznam *životních pravidel*, deklaraci tříd, vlastností (tzv. atributů) a procedur

(později metod), první dvě složky byly pozdějšími autory téměř zapomenuty. To, co se dnes považuje za OOP, jsou třídy obsahující pouze komponenty atributy a metody.

Podobně jako čistý pojem v racionálním myšlení nemůže třída sama o sobě nic vytvořit. Třída je definována pro použití jako určitý vzor, podle kterého se generují instance. Pro instanci třídy se také používá výraz být členem (*member*) třídy. Každá instance třídy má sadu vlastních atributů, které jsou přítomny v definici třídy, a je schopna provádět jakoukoli metodu, která je pro třídu deklarována. Instance jsou jednotky objektů, které nesou nějaké vlastnosti (odpovídající atributům) a které mohou provádět nějaké akce (odpovídající metodám). Z hlediska atributů představují instance datové struktury a kousek počítačové paměti, který je nese, je určitým obrazem toho, co aristotelská filozofie nazývá *materia prima* a co umožňuje uvažovat o dvou instancích jako o různých objektech i v případě, že jejich atributy mají shodné hodnoty. OOP tedy připouští formalizaci pojmů, aniž by bylo omezeno zákonem rozšiřitelnosti. Metody jsou popsány ve formě algoritmů. Mohou mít parametry a volat metody. Pro takové volání se běžně používá termín *message* a v zásadě má tvar *A-M-P*, který vyjadřuje, že instance *A* (tzv. selektor) je požadována k provedení metody *M* s parametrem *P*. Takové volání je obrazem konstrukce přirozeného jazyka, jako je *subjekt-predikát-objekt* nebo *subjekt-predikát-doplněk* nebo například *podstatné jméno-předložka-podstatné jméno* nebo *podstatné jméno-konjunkce-podstatné jméno*. Pro konstrukce používané v tradiční matematice může sdělení *A–M–P* představovat výraz, kde *M* je operátor provedený nad *A* a *P*. Množina tříd tedy může představovat obraz speciálního jazyka a/nebo formální teorie. (Kindler, et al., 2011)

Nedílnými součástmi OOP jsou specializace a virtualita. Specializace (nazývaná také podtřída) je vztah mezi dvěma třídami *C* a *D* (konkrétně: *D je podtřída C* nebo *D je specializace C*), který říká, že atributy a metody zavedené pro *C* platí i pro *D* a že na instance *D* lze pohlížet také jako na instance *C*. Specializace mimo jiné umožňuje zjednodušit formulaci tříd. Třídu lze specializovat na jednu nebo více podtříd. Specializaci lze iterovat (např. definovat *D* jako podtřídu *C*, *E* jako podtřídu *D* atd.) (Kindler, et al., 2011)

Ve třídě *C* může být metoda *M* zavedena jako virtuální, tj. jako smysluplná, ale bez specifického obsahu. Předpokládá se, že obsah bude zaveden v podtřídách třídy *C*. Umožňuje například používat zprávy typu *A–M–P*, když někdo píše stránky definic vztahujících se k *C*, takže během běhu programu budou interpretovány (a případně měněny) podle příslušnosti právě působícího prvku A. Možnost virtuality mimo jiné umožňuje

vyjadřovat obecné vazby mezi množinami akcí, které mají stejná jména, ale různé obsahy pro různé typy právě používaných objektů. (Kindler, et al., 2011)

3.6 Unified Modeling Language

Unified Modeling Language (UML) je grafický jazyk pro komunikaci specifikací návrhu softwaru. Komunita zabývající se vývojem objektově orientovaného softwaru vytvořila jazyk UML, aby vyhověl speciálním potřebám popisu návrhu objektově orientovaného softwaru. UML se stal standardem pro návrh digitálních systémů obecně. Existuje řada různých typů diagramů UML sloužících různým účelům. Typy diagramů tříd a diagramů aktivit jsou zvláště užitečné pro diskusi o otázkách návrhu databází. Diagramy tříd UML zachycují strukturální aspekty, které se vyskytují ve schématech databází. Diagramy aktivit v jazyce UML usnadňují diskusi o dynamických procesech spojených s návrhem databáze. Tato kapitola je přehledem syntaxe a sémantiky konstrukcí diagramů tříd a diagramů aktivit UML. Stejné koncepty jsou užitečné při plánování, dokumentování, diskusi a implementaci databází. Používáme UML 2.0. Diagramy tříd UML a modely entit a vztahů (ER) si jsou podobné jak formou, tak sémantikou. Původní tvůrci UML poukazují na vliv ER modelů na vznik diagramů tříd. Vliv UML zase ovlivnil databázovou komunitu. Diagramy tříd se nyní často objevují v databázové literatuře k popisu databázových schémat. Diagramy aktivit UML jsou svým účelem podobné vývojovým diagramům. Procesy jsou rozděleny na jednotlivé činnosti spolu se specifikacemi řídicích toků. (Teorey, et al., 2011)

3.6.1 Diagram tříd

Třída je deskriptor pro množinu objektů, které sdílejí některé atributy a/nebo operace. Třídy objektů si představujeme v každodenním životě. Například automobil má atributy, jako je identifikační číslo vozidla (VIN) a počet ujetých kilometrů. Auto má také operace, jako je zrychlení a brzdění. Tyto atributy a operace mají všechna auta. Jednotlivá auta se liší v detailech. Daný vůz má danou hodnotu VIN a počet ujetých kilometrů. Jednotlivé automobily jsou objekty, které jsou instancemi třídy Auto.

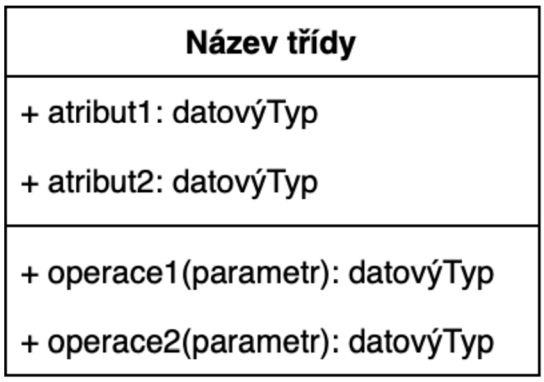
Třídy a objekty jsou přirozeným způsobem konceptualizace světa kolem nás. Koncepty tříd a objektů jsou také paradigmaty, která tvoří základ objektově orientovaného programování. Rozvoj objektově orientovaného programování vedl k potřebě jazyka pro popis objektově orientovaného návrhu, čímž vznikl jazyk UML.

Diagramy tříd v jazyce UML a ER diagramy spolu úzce korespondují. Třídy jsou obdobou entit. Databázová schémata lze promítat do diagramů pomocí jazyka UML. Databázovou tabulku je možné koncipovat jako třídu. Sloupce v tabulce jsou atributy a řádky jsou objekty této třídy. Například můžeme mít tabulku s názvem Auto se sloupci s názvy "vin" a "kilometry". Každý řádek tabulky by obsahoval hodnoty těchto sloupců a představoval by jednotlivé auto.

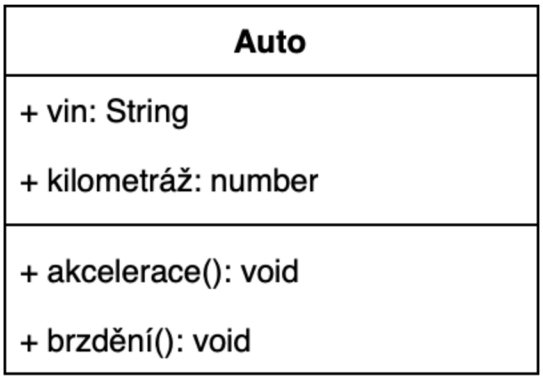
Hlavním rozdílem mezi třídami a entitami je absence operací v entitách. Všimněte si, že termín operace je zde použit ve smyslu jazyka UML. Uložené procedury, funkce, spouštěče a omezení jsou formy pojmenovaného chování, které lze v databázích definovat; nejsou však spojeny s chováním jednotlivých řádků. Termín operace v UML označuje metody vlastní třídám objektů. Tato chování nejsou uložena v definici řádků v databázi. Třídy mohou být v jazyce UML zobrazeny s atributy a bez operací, což je typické použití pro databázová schémata. (Teorey, et al., 2011)

3.6.2 Základní notace diagramu tříd

Ikonou UML pro třídu je obdélník. Je-li třída zobrazena s atributy a operacemi, je obdélník rozdělen na tři vodorovné přihrádky. Horní přihrádka obsahuje název třídy, vycentrovaný tučným písmem, začínajícím velkým písmenem. Názvy tříd jsou obvykle podstatná jména. Prostřední přihrádka obsahuje názvy atributů, zarovnané vlevo běžným písmem, začínající malým písmenem. Spodní přihrádka obsahuje názvy operací, zarovnané vlevo, začínající malým písmenem, zakončené závorkami. Závorky mohou obsahovat argumenty operace. (Teorey, et al., 2011)



Obrázek 8 – UML definice zápisu třídy (vlastní)



Obrázek 9 – UML příklad zápisu třídy (vlastní)

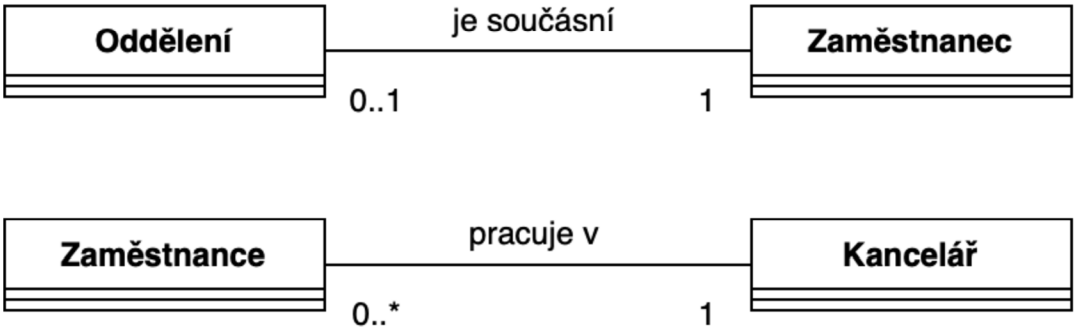
Zápis třídy má některé varianty, které odrážejí význam. Třídy lze zapsat bez oddílu atributů a/nebo bez oddílu operací. Operace jsou u softwaru důležité. Pokud se chce tvůrce softwaru zaměřit na operace, může být třída uvedena pouze s názvem třídy a oddílem operací. Zobrazení operací a skrytí atributů je velmi častá syntaxe používaná návrháři softwaru. Na druhou stranu návrháři databází se operacemi třídy obvykle nezabývají. Atributy jsou nesmírně důležité. Potřebám databázového návrháře lze vyhovět tak, že se třída zapíše pouze se zobrazením názvu třídy a oddílem s atributy. Skrývání operací a zobrazování atributů je pro návrháře softwaru neobvyklá syntaxe, ale pro návrh databáze běžná. A konečně, ve vysokoúrovňových diagramech je často žádoucí znázornit vztahy tříd, aniž by se zaplétaly do detailů atributů a operací. Pokud je žádoucí jednoduchost, lze třídy zapsat pouze s přihrádkou s názvem třídy. (Teorey, et al., 2011)

Mezi třídami mohou existovat různé typy vztahů. Jedním typem vztahu jsou asociace. Nejobecnější forma asociace je nakreslena pomocí čáry spojující dvě třídy.

Existuje několik typů asociací, například agregace a kompozice jsou velmi běžné. UML má pro tyto asociace určené symboly. Agregace označuje asociace typu *je část*, kde části mají nezávislou existenci. Například *Auto* může být součástí třídy *Sdílené Auto*. *Auto* existuje také samostatně, nezávisle na třídě *Sdílené Auto*. Dalším rozlišovacím znakem agregace je, že část může být sdílena objektu mezi více objekty. Například *Auto* může patřit více než jedné třídě *Sdílené Auto*. Agregační asociace je označena dutým kosočtvercem připojeným ke třídě, naznačuje, že *Sdílené Auto* agreguje *Auto*. Kompozice je další asociace *je součástí*, kde jsou části striktně vlastněny, nikoli sdíleny. Například třída *Rám* je součástí jednoho *Auta*. Zápisem pro kompozici je asociace ozdobená plným černým kosočtvercem připojeným ke třídě, která vlastní části. Dalším běžným vztahem je zobecnění. Například třída *Sedan* je typem třídy *Auto*. Třída *Auto* je obecnější než třída *Sedan*. Zobecnění je

naznačeno plnou čarou ozdobenou dutým hrotem šipky směřujícím k obecnější třídě. (Teorey, et al., 2011)

3.6.3 **Násobnost**



Obrázek 10 - UML násobnost příklad (vlastní)

Na koncích vztahu může být také uveden počet objektů zapojených do vztahu, označovaný jako násobnost (multiplicity). Hvězdička označuje, že na daném konci vztahu se asociace účastní mnoho objektů.

Binární asociace je vztah mezi dvěma třídami. Například jedna divize má mnoho oddělení. Plný kosočtverec je znakem asociace, která označuje složení. Všechny třídy účastnící se asociace budou spojeny s dutým kosočtvercem.

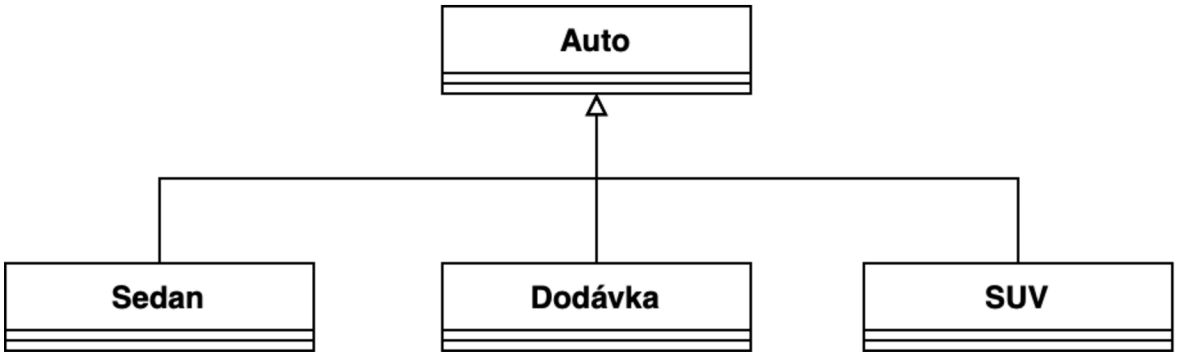
Každý konec ternární asociace je označen hvězdičkou, což znamená mnoho. Význam každé násobnosti je izolován od ostatních násobností. Je-li dána třída a v asociaci je přesně jeden objekt z každé jiné třídy, je násobnost počet asociovaných objektů pro danou třídu. Jeden *Zaměstnanec* pracující na jednom zadání projektu používá mnoho *Dovedností*. Jeden *Zaměstnanec* používá jednu *Dovednost* na mnoha *Projektových úkolech*. Jedna *Dovednost* používaná na jednom *Projektu* je plněna mnoha *Zaměstnanci*.

Diagram s asociací *one-to-many* znamená, že ku příkladu každé *Oddělení* má mnoho *Zaměstnanců* a každý *Zaměstnanec* patří přesně k jednomu *Oddělení*.

Many-to-many na příklad znamená, že každý *Zaměstnanec* je spojen s mnoha *Projekty* a každý *Projekt* je spojen s mnoha *Zaměstnanci*. Pokud má asociace atributy, jsou tyto zapsány ve třídě, která je k asociaci připojena přerušovanou čarou. Asociace a třída společně tvoří třídu asociace. Násobnost může být rozsah celých čísel, zapsaný s minimální a maximální hodnotou oddělenou dvěma tečkami. Hvězdička sama o sobě má stejný význam jako nekonečně mnoho *0..**. Také pokud jsou minimální a maximální hodnota stejné číslo,

pak lze násobnost zapsat jako jediné číslo. Například *1..1* znamená totéž co *1*. Nepovinnou existenci lze zadat pomocí nuly *0*. (Teorey, et al., 2011)

3.6.4 **Dědičnost**



Obrázek 11 - UML dědičnost příklad (vlastní)

Dalším typem vztahu je zobecnění (dědičnost). Nadtřída je zobecněním podtřídy. Specializace je opačný vztah než zobecnění. Podtřída je specializací nadtřídy. Vztah zobecnění se v UML zapisuje dutou šipkou směřující od podtřídy ke zobecněné nadtřídě. (Teorey, et al., 2011)

3.7 Návrhový vzor singleton

Návrhový vzor singleton omezuje instanci třídy na jedinou instanci. Koná se tak za účelem zajištění koordinovaného přístupu k určitému objektu v rámci celého softwarového systému. Prostřednictvím tohoto návrhového vzoru zajišťuje třída singleton, že je instancována pouze jednou, a může poskytovat snadný přístup k jediné instanci. (Soni, 2019)

3.7.1 **Použití návrhového vzoru singleton**

Singletony mohou být někdy považovány za alternativu ke globálním proměnným nebo statickým třídám. Oproti globálním proměnným mají singletony následující výhody:

- instance singletonu nezabírají místo v globálním jmenném prostoru,
- singletony mohou být líně (lazy) inicializovány.

Především díky tomu, že singleton obsahuje instancovaný objekt, zatímco statické třídy nikoli, mají singletony oproti statickým třídám následující výhody:

- singletony mohou implementovat rozhraní,
- singletony lze předávat jako parametry,
- singletony mohou mít různé instance (například pro účely testování),

- se singletony lze pracovat polymorfně, takže může existovat více implementací. (Soni, 2019)

3.8 Sémantické verzování 2.0.0

Když má software mnoho závislostí, může být těžké vydávat nové verze, protože pokud jsou závislosti specifikovány příliš striktně, můžete být vázán jednou verzí a nemůžete přejít na další bez vydání nové verze všech balíčků a knihoven. Naopak, pokud jsou závislosti specifikovány příliš volně, software může být kompatibilní s více verzemi, než je potřeba, a to může vést k problémům v rámci bezpečnosti a vývoje.

Řešením tohoto problému je použití pravidel pro přiřazování a navyšování verzí, která jsou založena na praxi vývoje softwaru. Nejdůležitější je definovat způsob, jakým se bude komunikovat s aplikací pomocí *API*. Toto *API* může být popsáno v dokumentaci nebo přímo v kódu, ale musí být srozumitelné a snadno čitelné. Je důležité mít kompletní dokumentaci a oznamovat rozdíly mezi verzemi. Verze jsou zapsány v formátu *MAJOR.MINOR.PATCH*, kde navýšení čísla *PATCH* znamená opravu chyby, které neovlivnily *API*, navýšení čísla *MINOR* znamená, že jsou přidány zpětně kompatibilní změny v *API* a navýšení čísla *MAJOR* verze znamená, že jsou provedeny změny v *API*, které nejsou zpětně kompatibilní.

Tento systém je známý jako Sémantické verzování a pomáhá udržovat závislosti v pořádku a umožňuje bezpečný vývoj. (Preston-Werner, 2023)

3.8.1 Specifikace Sémantického verzování (SemVer)

1. „Software používající Sémantické verzování, MUSÍ mít nadefinované *API*, buďto přímo ve zdrojovém kódu, anebo v externí dokumentaci. V obou případech to musí být hlavně přesné a komplexní.
2. Číslo verzí MUSÍ být ve formátu *X, Y, Z*. Jedná se o celá nezáporná čísla, přičemž *X* se nesmí rovnat hodnotě nula. Může se rovnat nule jen v případě, kdy se jedná o počáteční vývoj. *X* je číslo *MAJOR* verze, *Y* je číslem *MINOR* verze a *Z* je číslem *PATCH* verze, přičemž každé číslo má svoji hodnotu a navyšují se zvlášť a standardně, např. *1.9.0 => 1.10.0 => 1.11.0*.
3. Jakmile se vydá očíslovaná verze programu, nesmí se měnit a každá další úprava nebo oprava je vydána pod novou verzí.“ (Preston-Werner, 2023)

4. „*MAJOR* verze s hodnotou *0* (*0.y.z.*) je určena pro počáteční vývoj. Cokoliv se může změnit a *API* v tomto formátu by nemělo být považováno za stabilní.
5. Verze *1.0.0* definuje veřejně vydané *API*. Způsob, jakým se dále navyšuje číslo verze je ovlivněné tímto *API* a jeho změnami.
6. Číslo *PATCH* (*Z*) MUSÍ být navýšené jenom pokud byly implementované zpětně kompatibilní opravy chyb. Oprava chyb je definována jako interní změna opravující nežádoucí chování programu.
7. Číslo *MINOR* (*Y*) MUSÍ být zvýšené, když byla do *API* přidána nová, zpětně kompatibilní funkcionalita nebo pokud byla jakákoliv funkcionalita odebrána (jako zastaralá) i pokud neovlivňuje samotný *API* kód. Může zahrnout i změnu *PATCH* verze. Číslo *PATCH* verze se musí vynulovat vždy, když se změní *MINOR* verze.
8. Číslo *MAJOR* (*X*) musí být zvýšené, když byly přidané změny, které způsobily zpětnou nekompatibilitu. Může zahrnout i změny v rámci *MINOR* a *PATCH* verze. Číslo *MINOR* i *PATCH* se musí vynulovat vždy, když se změní *MAJOR* verze.
9. Předběžné verze (angl. pre-release) mohou být označeny přidáním pomlčky a sérií identifikátorů oddělených tečkou hned za číslo *PATCH* verze. Identifikátory musí obsahovat pouze *ASCII* alfanumerické znaky a pomlčku *[0-9A-Za-z-]*, nesmí být prázdné a číselné identifikátory nesmí obsahovat úvodní nulu. Předběžné verze mají nižší prioritu jako související normální verze. Předběžná verze je nestabilní a nemusí splňovat požadavky a závislosti jako normální verze. Např. *1.0.0-alpha*, *1.0.0-alpha.1*, *1.0.0-0.3.7*, *1.0.0-x.7.z.92*.
10. Metadata mohou být označena ve verzi přidáním znaku plus (+) a sérií identifikátorů oddělených tečkou hned za číslo *PATCH*, anebo pomocí předběžné verze. Identifikátory musí obsahovat pouze *ASCII* alfanumerické znaky a pomlčku *[0-9A-Za-z-]*, nesmí být prázdné a číselné identifikátory nesmí obsahovat úvodní nulu. Metadata by neměla hrát roli při volbě priority verze. Např. verze *1.0.0-alpha+001*, *1.0.0+20130313144700*, *1.0.0-beta+exp.sha.5114f85* mají všechny stejnou prioritu.
11. Priorita se vztahuje na to, jak se verze vzájemně porovnávají. Priorita musí být určována rozdělením verze na *MAJOR*, *MINOR*, *PATCH* a identifikátory předběžných verzí – přesně v tomto pořadí (s metadaty se nepočítá). Priorita je daná prvním rozdílem při porovnání zleva doprava, přičemž čísla *MAJOR*, *MINOR* a *PATCH* jsou vždy porovnávána jako čísla. Např. *1.0.0* < *2.0.0* < *2.1.0* < *2.1.1*. Pokud jsou čísla *MAJOR*, *MINOR* a *PATCH* stejná“, (Preston-Werner, 2023)

„předběžná verze má menší prioritu než normální. Např. *1.0.0-alpha* < *1.0.0*. Priorita pro dvě předběžné verze, které se shodují v číslech *MAJOR*, *MINOR* a *PATCH* musí být počítána zleva doprava od tečky oddělených identifikátorů a to do doby, dokud se nenajde rozdíl a to následujícím způsobem:

I. Identifikátory obsahující pouze číslice, jsou porovnané číselně a identifikátory s písmeny nebo pomlčkami jsou porovnávané lexikálně, zařazené podle *ASCII*.
II. Číselné identifikátory mají vždy nižší prioritu jak nečíslené.
III. Jsou-li všechny předchozí identifikátory v předběžné verzi stejné, tak větší množství identifikátorů značí vyšší prioritu než s menším počtem. Např. *1.0.0-alpha* < *1.0.0-alpha.1* < *1.0.0-alpha.beta* < *1.0.0-beta* < *1.0.0-beta.2* < *1.0.0-beta.11* < *1.0.0-rc.1* < *1.0.0*.“ (Preston-Werner, 2023)

3.9 User Experience design

Písmena *UX* jsou populární zkratkou, která znamená *user experience*. Tato dvě písmena označují celou praxi, veškerou práci, která se v tomto oboru dělá, a výsledný uživatelský zážitek, který z této práce vzejde.

V září 2010 se na Schloss Dagstuhl sešla mezinárodní skupina (seminář Demarcating User eXperience), aby vypreparovala podstatu uživatelského zážitku a pomohla definovat její hranice. Ve své následné zprávě poukazují na to, že multidisciplinární charakter UX vedl k mnoha definicím z různých perspektiv, včetně UX jako teorie, UX jako fenomén, UX jako obor studia a UX jako praxe. (Hartson, et al., 2019)

3.9.1 Vzestup uživatelského zážitku

První velké sálové počítače z před-konzumní éry se používaly k provozování velkých podnikových softwarových systémů a uživatelé byli vyškoleni k používání systému pro konkrétní a určité účely.

Interakce probíhala prostřednictvím děrných štítků, papírových pásek a papírových výtisky, takže při vývoji systému se ve skutečnosti nehledělo na použitelnost nebo UX.

Pak se díky osobnímu počítači dostala výpočetní technika na stůl podnikových uživatelů a díky spotřebitelskému hnutí se dostala do domácností. Zákaznický servis a podpora jako první zjistili, že rozšíření trhu pro konzumní společnost bez dostatečného porozumění tomu, jak se produkt používá, má zásadní dopad na náklady na podporu.

Chytrá zařízení a internet daly výpočetní techniku do rukou každého člověka a umožnily podnikům přímou komunikaci se spotřebiteli. Paradigma se posunulo od toho, že uživatelé potřebují školení k používání systému, k požadavku, aby systém odpovídal očekáváním uživatelů, a tak byla cesta k použitelnosti, HCI (Human-computer interaction) a UX nevyhnutelná.

Digitální domorodci nyní považují výpočetní techniku za něco, co existuje – je pro ně transparentní, že za produktem stojí design. Jednoduše očekávají, že to bude fungovat. (Hartson, et al., 2019)

3.9.2 Charakteristika UX designu

Uživatelský zážitek je samozřejmě druh zkušenosti a "zkušenost je velmi dynamický, komplexní a subjektivní fenomén", který do značné míry závisí na kontextu související činnosti.

Uživatelský zážitek je souhrn účinků, které uživatel pociťuje před, během a po interakci s produktem nebo systémem v jeho ekosystému.

Úkolem UX návrhářů je navrhnout tuto interakci tak, aby uživatelská zkušenost byla produktivní, naplňující, uspokojující, a ve finále radostná.

Klíčové charakteristiky uživatelského zážitku, které se odrážejí ve výše uvedené definici, jsou:

- výsledek interakce, ať už přímé, nebo nepřímé,
- zaměření na souhrn účinků,
- vcítění se do uživatele,
- zahrnutí možnosti ekosystému. (Hartson, et al., 2019)

3.9.3 Důležitost uživatelského zážitku

Význam UX je stále více uznáván a design UX se dostává do popředí zájmu. Jak řekl jeden z vedoucích viceprezidentů společnosti IBM: "Již neexistuje žádný skutečný rozdíl mezi obchodní strategií a návrhem uživatelského prostředí". S tím souhlasí i Knemeyer (německý letecký inženýr, letec a vedoucí technického vývoje na říšském ministerstvu letectví nacistického Německa během druhé světové války) když říkal, že "uživatelský zážitek (UX) se stal kritickým hlediskem pro společnosti v každém odvětví a všech tvarů a velikostí".

Jedním ze způsobů, jak zdůraznit význam dobrého návrhu UX, jsou příklady vysokých nákladů na špatný návrh UX. Jako příklad lze uvést špatný UX design v budovách

a obytných prostorech, který může znamenat dlouho trvající náklady. "Lidé, kteří navrhují a staví budovy a parky, se příliš často nestarají o to, zda budou správně fungovat nebo kolik bude stát jejich provoz. Jakmile je projekt dokončen, mohou se přesunout k další zakázce. Veřejnost však musí žít se špatně postavenými, špatně navrženými budovami a prostory. Daňoví poplatníci často musí zaplatit účet za jejich opětovné uvedení do pořádku."!

Špatný návrh UI (User Interaction design)/UX stojí obrovské množství peněz a hlavně životů. Rozptylování pozornosti v důsledku špatného návrhu UX pro ovládání automobilů může vést k dopravním nehodám, zraněním, a dokonce i k úmrtí.

Stejná opatrnost platí i pro návrh UX pro ovládání letadel a lodí na moři. Například příčina havárie letu EgyptAir 990 v roce 1999 byla vyhodnocena jako vliv špatného UX designu ovládacích prvků v pilotní kabině. Srážka lodi USS McCain byla údajně důsledkem špatného návrhu UX navigační konzole.

V lékařské oblasti je potřeba dobrého návrhu UX snad ještě přesvědčivější s ohledem na účinky bezpečnosti v každodenním provozu. Jak uvádí společnost Nielsen: "Terénní studie identifikovala 22 způsobů, jak mohou automatizované nemocniční systémy vést k tomu, že pacientům budou vydány nesprávné léky. Většina z těchto nedostatků jsou klasické problémy použitelnosti, které jsou známy již desítky let." (Hartson, et al., 2019)

3.9.4 **Komponenty UX**

UX je složeno z následujících komponent:

- použitelnost
- užitečnost,
- emocionální dopad,
- smysluplnost. (Hartson, et al., 2019)

3.9.4.1 ***Použitelnost***

Kdysi dávno se obor interakce člověka s počítačem, akademická disciplína pro UX, zabýval v podstatě jen použitelností, která zahrnuje (ISO 9241-11, 1997):

- snadnost používání,
- výkonnost a produktivita uživatele,
- efektivitu,
- předcházení chybám,
- učitelnost,

- uchovatelnost (snadné zapamatování).

I v současnosti je použitelnost stále velmi důležitou součástí UX. Vzhledem k tomu, že se tento obor zaměřil na okázalejší části uživatelského prostředí, někdy se zapomíná na základní složku, použitelnost. Například v dnešní době populární tzv. *flat design* styl vypadá a působí vizuálně atraktivně, ale postrádá důležitou dostupnost, která odhaluje, které prvky na obrazovce jsou interakční a které ne. Bez dobré použitelnosti se jen zřídkakdy začne uvažovat o dalších složkách uživatelského zážitku. (Hartson, et al., 2019)

3.9.4.2 *Užitečnost*

Druhou složkou je užitečnost, možná zapomenuté nevlastní dítě uživatelského zážitku. Užitečnost se týká výkonu a funkčnosti *backend softwaru*, který vám dává možnost pracovat (nebo se bavit). Je to skutečný hlavní důvod produktu nebo systému.

Hassenzahl a Roto (2007) charakterizují použitelnost a užitečnost jako službu uživateli k dosažení jeho cílů. Těmi mohou být kontrola e-mailu nebo odeslání komentáře na Facebook. (Hartson, et al., 2019)

3.9.4.3 *Emocionální dopad*

Třetí složkou je emocionální dopad, afektivní část uživatelského zážitku.

Jak již tento termín napovídá, emocionální dopad zahrnuje emocionální pocity uživatelů z interakce včetně jejich spokojenosti.

Ačkoli o emocích v uživatelském prožitku existovaly již dřívější akademické práce, Norman (2002) byl jedním z prvních, kdo toto téma v širším měřítku uvedl na pravou míru a spojil ho se svým tématem každodenních věcí. Nyní existují konference věnované speciálně tomuto tématu, včetně konference Design & Emotion, která se koná každé dva roky a jejímž cílem je podpořit mezioborový přístup k designu a emocím.

Ačkoli technicky vzato je veškerá uživatelská zkušenost emocionální, protože ji uživatel celou vnitřně prožívá, existují některé faktory uživatelské zkušenosti, které jsou čistě emocionální, faktory, které jsou při používání technologie (ať už špičkové, nebo nízké) pociťovány osobně; faktory, které uživatele posouvají za hranice prosté spokojenosti k zábavě, potěšení a sebevyjádření. To může mít někdy silné emocionální důsledky. (Hartson, et al., 2019)

3.9.4.4 *Smysluplnost*

Zatímco použitelnost, a často i emocionální dopad, je obvykle jednorázová, smysluplnost je o tom, jak se produkt nebo artefakt stává smysluplným v životě uživatele. Smysluplnost vychází z osobního vztahu produktu a jeho lidským uživatelem v delším časovém horizontu. Ztělesňuje ji pocit souručenství, který mnozí chovají ke svým chytrým telefonům, a to až do té míry, že někteří uživatelé se cítí fyzicky nepříjemně, pokud se od svého telefonu odloučí.

Smysluplnost úzce souvisí s akademičtějším pojmem fenomenologie. (Hartson, et al., 2019)

3.9.5 **Informační architektura**

Informační architektura je obor, který se zaměřuje na organizaci informací v digitálních produktech. Například při tvorbě aplikací a webových stránek rozvrhují designéři jednotlivé obrazovky tak, aby uživatel snadno našel potřebné informace. Vytvářejí také průchod uživatelů mezi obrazovkami bez větší námahy. Architekti UX určují správnou organizaci a pohyb uživatele v softwaru.

Obsah je důvodem, proč lidé navštěvují webové stránky. Je důležité vytvářet obsah, který uživatelé považují za hodnotný, ale stejně důležité je zajistit, aby byl obsah snadno dostupný.

Čas je nejcennějším zdrojem, který lidé mají. Žijeme ve světě, kde lidé očekávají, že najdou řešení svých problémů s co nejmenším úsilím. Pokud je vyhledávání informací příliš složité nebo pomalé, hrozí, že je lidé jednoduše opustí. A když lidé aplikaci nebo web opustí, je obtížnější je přivést zpět. Právě zde hraje klíčovou roli návrh informační architektury.

Pro pochopení rozdílu mezi informační architekturou a UX designem, je důležité uvědomění, co je to UX design. Uživatelský zážitek je způsob, jakým člověk přemýšlí a cítí se při používání produktu, systému nebo služby. UX zahrnuje užitečnost, použitelnost a potěšení z používání systému – mnohem více než jen strukturu obsahu.

Zároveň je téměř nemožné vytvořit dobrý uživatelský zážitek bez pevného základu informační architektury. (Babich, 2020)

3.9.6 **Prototyp typu wireframe**

Wireframe je druh prototypu, čárkovaná reprezentace návrhu UX, zejména návrhu interakce a obrazovky. Wireframe se skládá z čar a obrysů (odtud název wireframe) krabic a dalších tvarů, které znázorňují vznikající návrhy interakce. Jsou to:

- schémata a náčrty, které definují obsah webové stránky nebo obrazovky a navigační tok.
- Schémata sloužící k ilustraci konceptů na vysoké úrovni, přibližného vizuálního rozvržení a chování.
- Schémata, někdy používaná k zobrazení vzhledu a dojmu z návrhu interakce.
- Schémata pro zobrazení přechodů obrazovky nebo jiných stavů během používání, které znázorňují předpokládané toky („flow“) úloh z hlediska akcí uživatele pro objekty uživatelského rozhraní. (Hartson, et al., 2019)

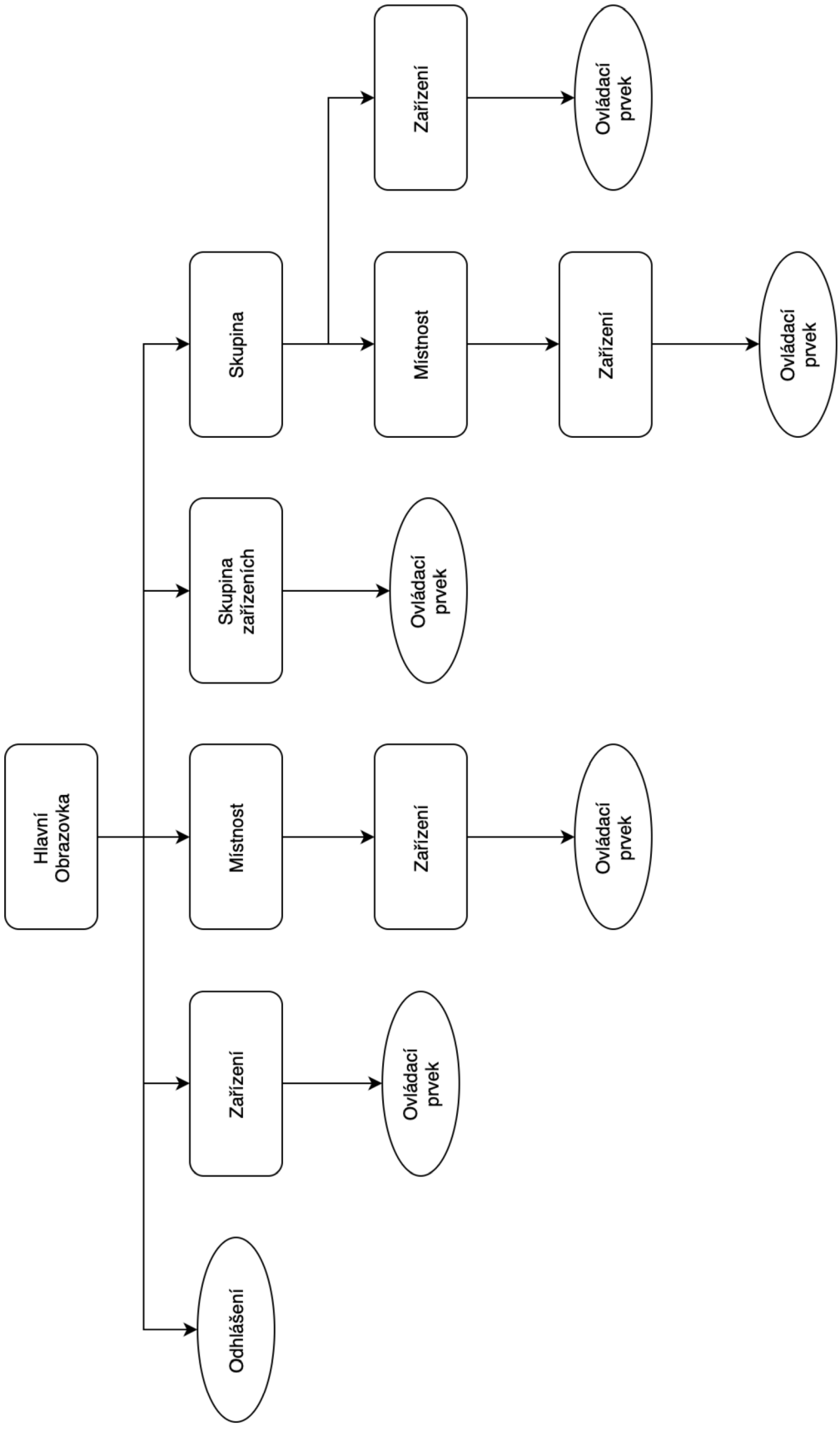
4 Vlastní práce

4.1 Analýza stávajícího řešení

Stávající *webová aplikace* dovoluje uživateli skrze její grafické rozhraní ovládat zařízení daného inteligentního domu, přesněji jeho *PLC*.

Jak je zmíněno v kapitole *Nástroj WebMaker*, každé *PLC* má vlastní web server, na kterém je pro *PLC* konkrétní *webová aplikace* nasazena. Uživatel se tedy připojuje přímo na dané *PLC* skrze veřejnou *IP* adresu, nebo za pomoci *IP* tunelu, který výrobce *PLC* nabízí. Jednotlivé aplikace pro jednotlivé inteligentní domy nejsou jednotné, jedná se vždy o samostatný software, pro každý inteligentní dům musí být vytvořena aplikace zvlášť.

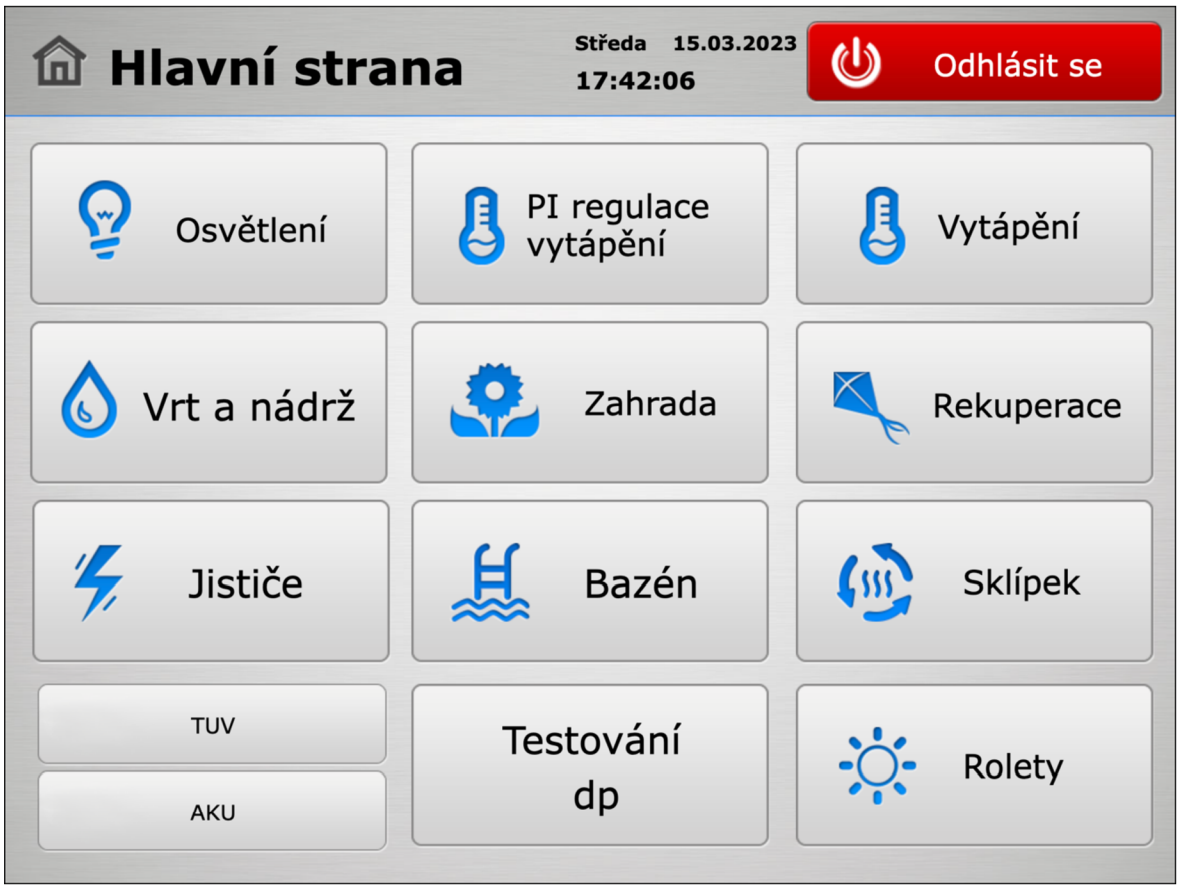
4.1.1 Informační architektura aplikace konkrétního *PLC*



Obrázek 12 – Diagram informační architektury stávajícího řešení (vlastní)

Z analýzy informační architektury stávajícího řešení vzniká diagram *Obrázek 12 – Diagram informační architektury stávajícího řešení*, kde prvky v elipse označují komponenty pro interakci uživatele s aplikací, prvky v obdélníku označují skupiny komponent a dalších podskupin.

4.1.1.1 *Hlavní obrazovka*



Obrázek 13 - Obrazovka typu "Hlavní obrazovka" (vlastní)

Po přihlášení je uživatel směrován na obrazovku s názvem *Hlavní strana*. Z této obrazovky uživatel může provést úkony jako: odhlásit se; navigovat na obrazovky kategorií:

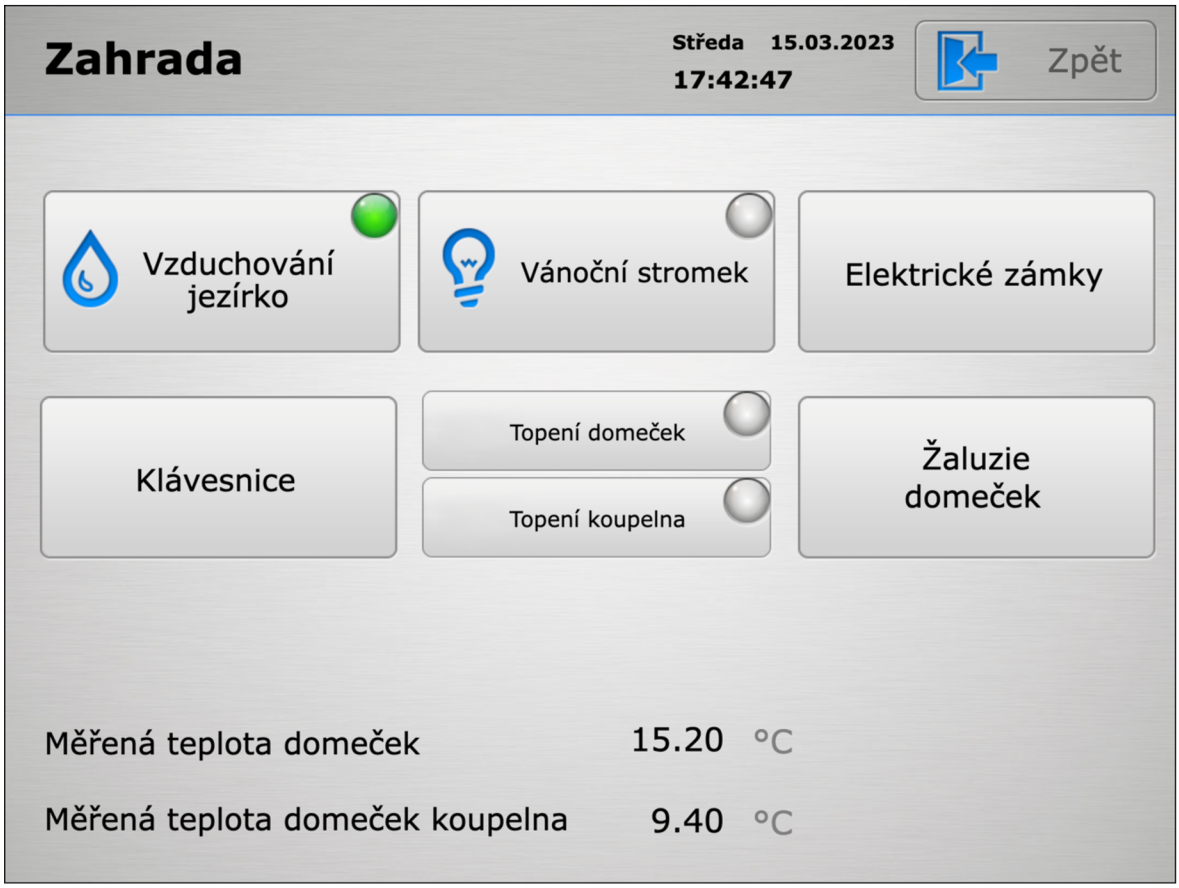
- Zařízení – *Vrt a nádrž*, *PI regulace*, *Rekuperace*, *Bazén*, *TUV*, *AKU*,
- Místnost – *Zahrada*, *Sklípek*,
- Skupina *zařízeních* – *Vytápění*. *Jističe*, *Rolety*,
- Skupina – *Osvětlení*.

4.1.1.2 *Zařízení*

Obrázek 14 - Obrazovka typu "Zařízení" (vlastní)

Na obrazovce typu *Zařízení* uživatel přímo interaguje s komponentami *PLC* a může tak zařízení ovládat, obrazovka nese informační komponenty, které předávají uživateli informace o stavu zařízení.

4.1.1.3 *Místnost*



Obrázek 15 - Obrazovka typu "Místnost" (vlastní)

Na obrazovce typu *Místnost* se uživatel dostává do seznamu skupin *Zařízení*. Obrazovka tohoto typu není standardizována, mohou se na ní zobrazovat ovládací a informační prvky.

4.1.1.4 *Skupina zařízeních*

Obrázek 16 - Obrazovka typu "Skupina zařízeních" (vlastní)

Na obrazovce typu *Skupina zařízeních* se uživatel dostává k množině komponent typu *Zařízení*, skrze které může přímo interagovat s *PLC*. Uživatel má možnost se prokliknout do detailnějšího přehledu zařízení pro detailnější informace a ovládání.

4.1.1.5 *Skupina*

Obrázek 17 - Obrazovka typu "Skupina" (vlastní)

Na obrazovce typu *Skupina* uživatel dostává možnost pro navigování se do podskupin typu *Zařízení* a *Místnost*.

4.1.2 **Aplikace na různých platformách**

Jak bylo již zmíněno, jedná se o webovou aplikace, ke které lze přistoupit skrze kterékoliv zařízení s webovým prohlížečem splňující minimální požadavky viz *Minimální požadavky na prohlížeč*.

Aplikace tedy lze spustit na desktopových operačních systémech (např. macOS, Linux, Windows) a mobilních operačních systémech (např. iOS, Android). Těmito operačními systémy zahrnuje tedy jak platformu desktopovou, tak mobilní. Na desktopu je aplikace responzivní viz *Obrázek 18 – Stávající řešení aplikace v prohlížeči, desktop* a na mobilním zařízení viz *Obrázek 19 – Stávající řešení aplikace v prohlížeči, mobil*.

Obrázek 18 – Stávající řešení aplikace v prohlížeči, desktop (vlastní)

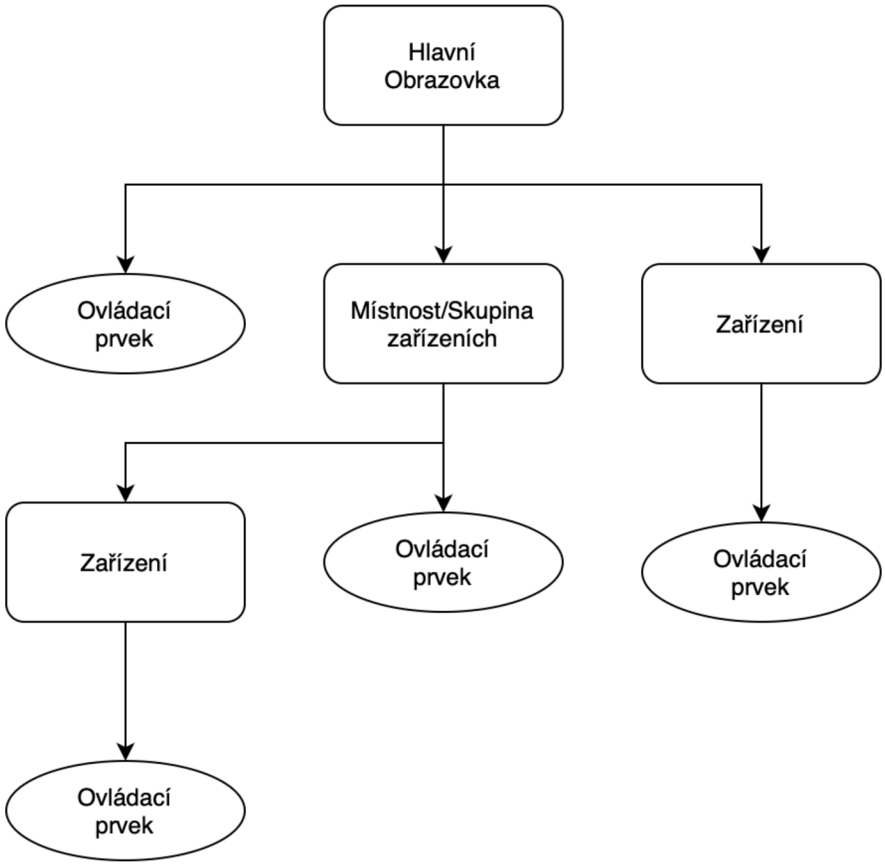
Obrázek 19 – Stávající řešení aplikace v prohlížeči, mobil (vlastní)

4.2 Návrh prototypu nového řešení

Nové řešení klientské aplikace pro ovládání *PLC* inteligentního domu by mělo být řešením obecným – využívání aplikace více uživateli různých inteligentních domů; řešením cross-platformovým – aplikace je pro uživatele přívětivá na mobilních zařízeních a v okně internetového prohlížeče různých rozměrů; řešením jednoduchým a přehledným.

Bude se jednat o omezený prototyp se statickou konfigurací uloženou v aplikaci Prototyp bude komunikovat s *PLC* skrze komunikační protokol WebSocket. Prototyp bude zobrazovat informace a umožňovat ovládat definovaná zařízení konkrétního *PLC* inteligentního domu, které mu *API PLC* poskytne. Komponenty jsou světlo (sledování a nastavení stavu vypnuto/zapnuto) a ovládání rolet (sledování a nastavení procent stažení).

4.2.1 Informační architektura



Obrázek 20 – Diagram aplikační architektury nového řešení (vlastní)

Nové řešení aplikace má být obecná klientská aplikace ovládající jednotlivá zařízení všech *PLC*. Zařízení jsou tedy obecná a všechna na stejné úrovni.

K zařízením a jejich ovládání se dostaneme skrze obrazovku po spuštění aplikace, na *Obrázek 20 – Diagram aplikační architektury nového řešení* jako *Hlavní obrazovka*. Na této

obrazovce můžeme zařízení omezeně ovládat a získávat omezené informace; navigovat na detail *Zařízení*; navigovat do *Místnost/Skupina zařízeních*.

Obrazovka *Místnost/Skupina zařízeních* obsahuje množinu zařízeních buď dle místnosti, do které jsou přiřazeny, nebo dle jejich typu. Jedná se o vedlejší obrazovku. Z této obrazovky lze jednotlivá zařízení omezeně ovládat a získávat omezené informace; navigovat na jejich detail *Zařízení*.

Obrazovka *Zařízení* je detailem jednotlivého zařízení, které má specifické funkce a zobrazuje specifické informace pro svůj typ.

4.2.2 **Wireframe návrh**

Obrázek 21 - Návrh hlavní obrazovky nového řešení (vlastní)

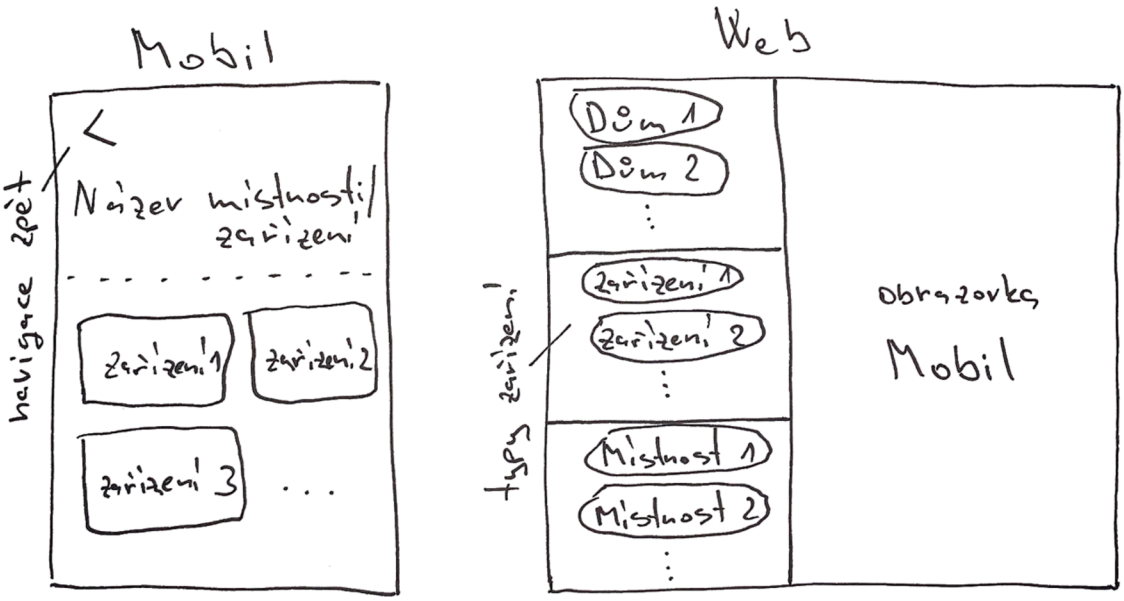
Obrázek 21 - Návrh hlavní obrazovky nového řešení rozvrhuje hlavní obrazovku do několika částí, *Mobil* zobrazení shora postupně:

- Název domu – v této části je název právě zvoleného domu, inteligentního domu, kterému je aplikace přizpůsobena pro ovládání. Vpravo od názvu se nachází ikonka pro změnu/výběr domu.
- Horizontálně rolovatelný list pro zařízení– v tomto listu se nachází ikony typů zařízení, které jsou v domě použity. Po stisku/kliku na ikonu je uživatel navigován na vedlejší obrazovku obsahující komponenty všech zařízení v domě daného typu.
- Název místnosti – název místnosti pro místnost obsahující ovladatelná zařízení. Po stisku/kliku na místnost je uživatel navigován na vedlejší obrazovku obsahující komponenty všech zařízení pro danou místnost.

- Horizontálně rolovatelný list pro místnost– v tomto listu se nachází komponenty všech zařízení pro danou místnost nesoucí název části výše.

Části *Název místnosti* a *Horizontálně rolovatelný list pro místnost* se opakují pro každou místnost inteligentního domu.

Web zobrazení hlavní obrazovky se skládá z menu zarovnaného vlevo s určitou šířkou a zbytek prostoru vyplňuje responzivní zobrazení identické se zobrazením *Mobil*. Zmíněné menu je rozděleno tří částí, skrze které uživatel přepíná ovládaný inteligentní dům a je navigován na vedlejší obrazovku obsahující list komponent zařízení dle typu zařízení nebo místnosti ke které jsou zařízení přiřazena.

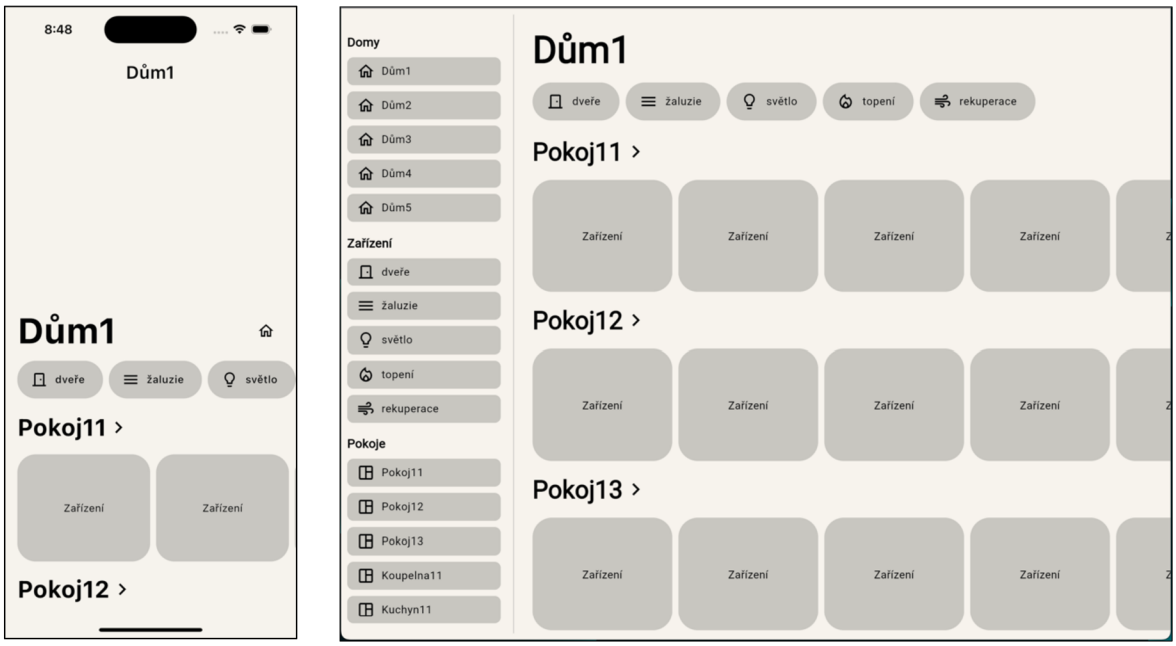


Obrázek 22 - Návrh vedlejší obrazovky nového řešení (vlastní)

Obrázek 22 - Návrh vedlejší obrazovky nového řešení rozvrhuje vedlejší obrazovku, obrazovku obsahující list komponent zařízení pro dané zobrazení. Skrze komponenty lze omezeně ovládat zařízení, poskytují omezené informace o zařízení, lze se skrze ně stiskem/klikem dostat na detail zařízení pro podrobnější ovládání a informace.

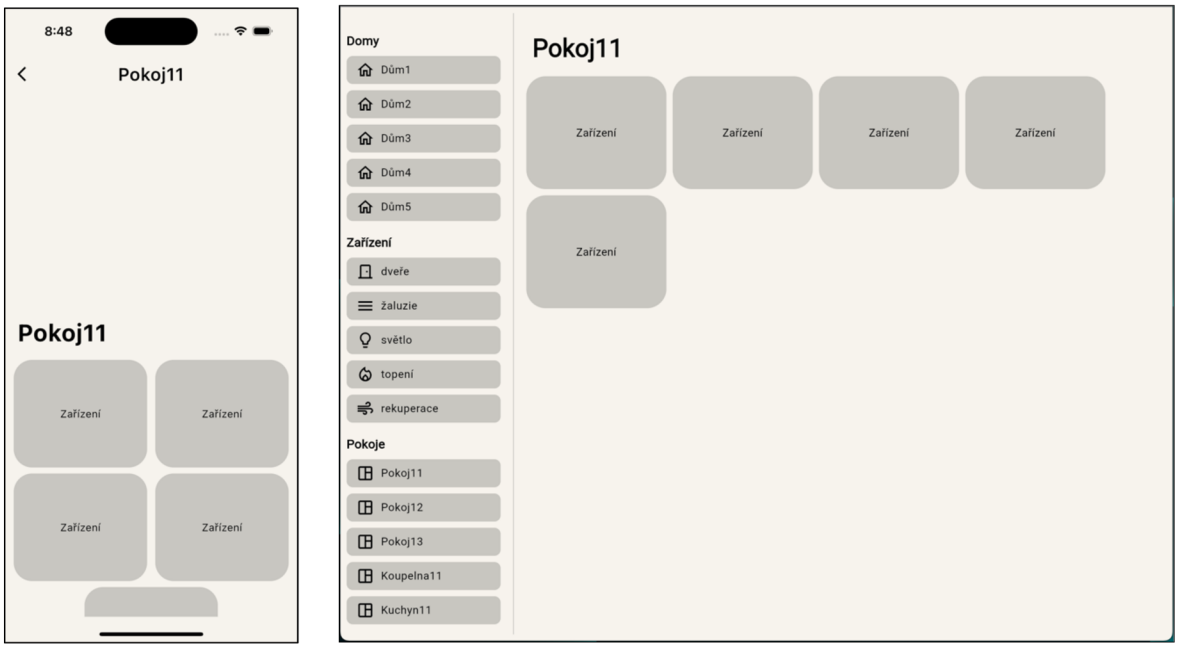
V zobrazení *Web* navíc obsahuje menu zarovnaného vlevo s určitou šířkou, chovající se stejně jako na hlavní obrazovce. Zbytek prostoru je vyplněn zobrazením *Mobil* stejně jako na hlavní obrazovce.

4.2.3 Grafický návrh



Obrázek 23 - Grafický návrh hlavní obrazovky nového řešení (vlastní)

Obrázek 23 - Grafický návrh hlavní obrazovky nového řešení je grafickou implementací *Obrázek 21 - Návrh hlavní obrazovky nového řešení.*



Obrázek 24 - Grafický návrh vedlejší obrazovky nového řešení (vlastní)

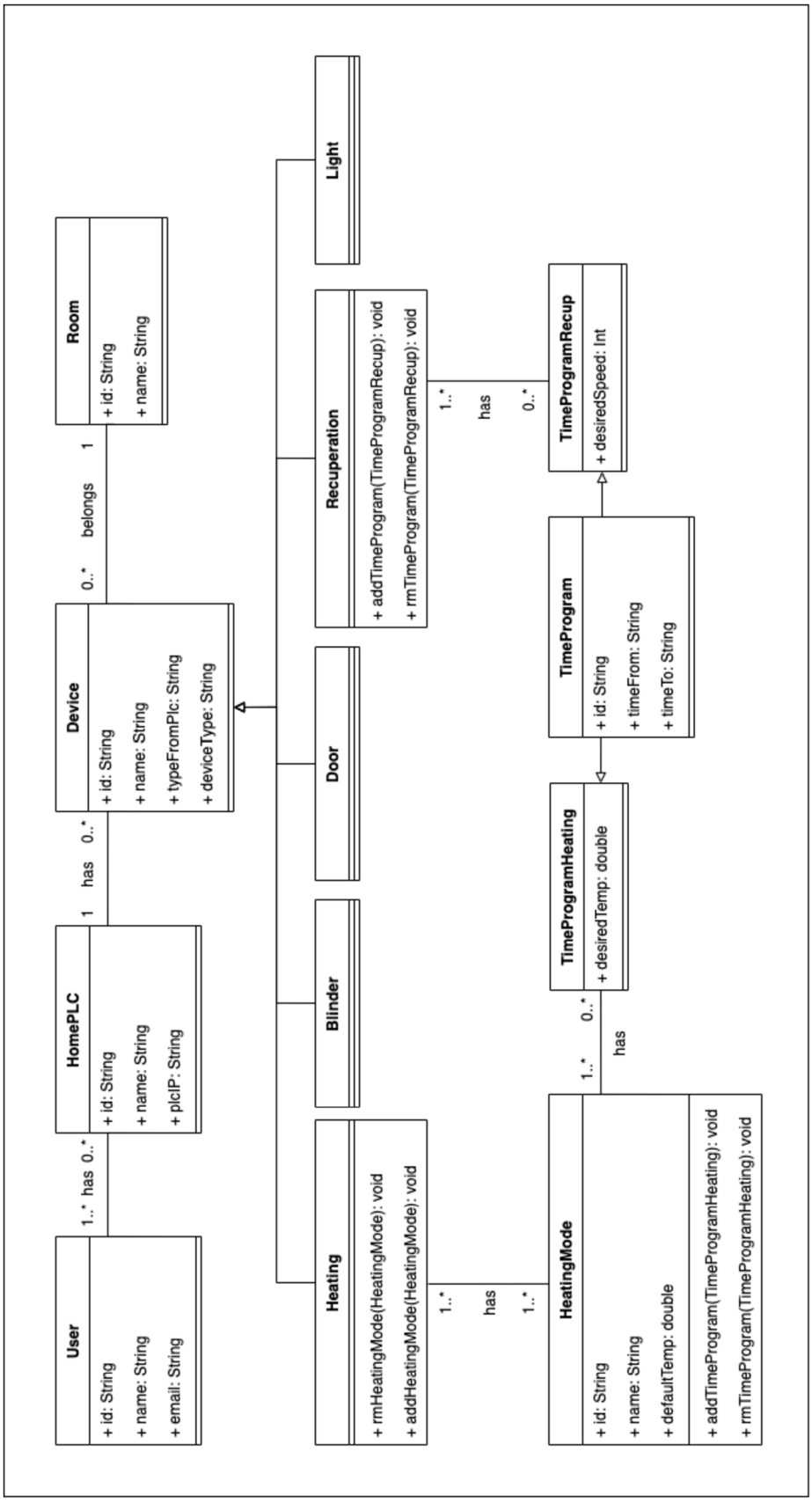
Obrázek 24 - Grafický návrh vedlejší obrazovky nového řešení je grafickou implementací *Obrázek 22 - Návrh vedlejší obrazovky nového řešení.*

V grafických návrzích byl brán ohled na doporučovaný vzhled a rozložení komponent zobrazení dle příruček *Apple Human Interface Guidelines* (Apple Inc., 2023) a *Material Design Guidelines* (Google LLC, 2023).

Barevné rozvržení je děleno na dvě schéma:

1. schéma pro světlý režim zařízení, hexadecimální kódy barev:
   - pozadí – #F6F3ED,
   - pozadí komponent – #C8C6C1,
   - text – #000000.
2. Schéma pro tmavý režim zařízení, hexadecimální kódy barev:
   - pozadí – #000000,
   - pozadí komponent – #282827,
   - text – #F6F3ED.

4.2.4 Diagram tříd UML



Obrázek 25 - Diagram tříd konfigurace nového řešení (vlastní)

Obrázek 25 - Diagram tříd konfigurace nového řešení popisuje a graficky ztvárňuje třídy, jejich atributy a metody konfigurace aplikace. Díky této konfiguraci aplikace pracuje s uživatelem, k němu přidanými inteligentními domy. Každý intelignetní dům (jeho *PLC*) osahuje zařízení různých typů. Těmto zařízením připadají místnosti.

Inteligentní dům *HomePLC* má atribut *plcIP* datového typu řetězec znaků, nesoucí *IP* adresu *PLC*. Na tuto *IP* adresu klientská aplikace otevírá komunikační kanál a dostává informace/ovládá jednotlivá zařízení. Každé zařízení je identifikováno unikátním identifikátorem.

4.2.5 Použití *SDK* Flutter

Nové řešení bylo vytvořeno za pomoci *SDK* Flutter ve verzi 3.7.3 v programovacím jazyce Dart ve verzi 2.19.2. Učiněno tomu bylo z důvodu vyhovujících požadavků na technologii pro řešení viz kapitola *Flutter*.

4.3 Vývoj prototypu aplikace s *SDK* Flutter

Cross-platformová aplikace byla vyvíjena ve vývojovém prostředí *VSCode*, které je pro vývoj aplikace s *SDK Flutter* doporučováno. (Google LLC, 2023)

Komponenty aplikace jsou upravené komponenty knihovny *Material*, jejíž komponenty jsou pro navržený design vhodné. (Google LLC, 2023)

4.3.1 Správa stavu aplikace

Pro správu stavu aplikací vyvíjených s *SDK Flutter* je dokumentací doporučován doplněk vývojáře třetí strany *Provider* (Google LLC, 2023). Sám vývojář ale dle historie verzování a komunikace v komunitě od doplňku upouští a věnuje se jeho novému doplňku pro správu stavu aplikace, doplňku *Riverpod*. Ten byl ve verzi 2.1.3 v aplikaci použit.

4.3.1.1 *Nastavení doplňku Riverpod*

```
1 void main() {
2   _setupFimber();
3   setPathUrlStrategy();
4   runApp(
5     const ProviderScope(
6       child: MyApp(),
7     ),
8   );
9 }
```

Kód 1 - Nastavení doplňku Riverpod (vlastní)

V kódu *Kód 1 - Nastavení doplňku Riverpod* v inicializační funkci aplikace *runApp* (řádek 4) jejímu parametru namísto *widgetu* aplikace *MyApp* byl vložen *widget* doplňku, *widget ProviderScope* (řádek 5). Jeho potomkem se stal již zmíněný *widget MyApp* (řádek 6).

Tímto bylo umožněno doplňku *Riverpod* spravovat stav aplikace definované v *MyApp*.

4.3.1.2 *Definice vlastního providera*

Ukázka definice providera, konkrétně *favoritesProvider*, providera, který je využíván k obhospodařování identifikátorů oblíbených zařízení.

```
1 final favoritesProvider =
2     StateNotifierProvider<FavoritesNotifier, FavoritesState>(
3   (ref) => FavoritesNotifier(),
4 );
```

Kód 2 - Definice vlastního providera (vlastní)

V kódu *Kód 2 - Definice vlastního providera* byla nadefinována globální konstanta typu *StateNotifierProvider*, která obsahuje typ *Notifier* a typ *State*. Konstanta je používána při práci s daty, stavem nebo funkcionalitou providera.

```
1 class FavoritesState with _$FavoritesState {
2   factory FavoritesState({
3     @Default([]) List<String> deviceIDs,
4     @Default(true) bool isLoading,
5   }) = _FavoritesState;
6 }
```

Kód 3 - Stav vlastního providera (vlastní)

V kódu *Kód 3 - Stav vlastního providera* byla definována třída, která je stavem providera. Stav má dva atributy, list identifikátorů oblíbených zařízeních (řádek 3) a binární proměnnou

označující činnost providera (řádek 4). List je definován jako prázdný a proměnná označující činnost (načítání) je definována jako *pravda*, provider je tedy činný.

```
1  class FavoritesNotifier extends StateNotifier<FavoritesState> {
2    FavoritesNotifier() : super(FavoritesState()) {
3      _getFavoriteDeviceIDs();
4    }
5
6    Future<void> _getFavoriteDeviceIDs() async {
7      final prefs = await SharedPreferences.getInstance();
8      final deviceIDs = prefs.getStringList(Consts.favoriteDevicesKey);
9      state = state.copyWith(deviceIDs: deviceIDs ?? [],
10                            isLoading: false);
11   }
12
13   Future<void> toogleDevice({required String deviceId}) async {
14   ...
15   }
16 }
```

Kód 4 - Funkcionalita vlastního providera (vlastní)

V kódu *Kód 4 - Funkcionalita vlastního providera* byla definována třída typu *StateNotifier*, která obsahuje funkcionalitu providera. V této třídě se nachází logika obhospodařující stav providera (řádky 9, 10).

4.3.1.3 *Použití vlastního providera*

```
1 class Widget extends ConsumerWidget {
2   @override
3   Widget build(BuildContext context, WidgetRef ref) {
4     return AnotherWidget();
5   }
6 }
```

Kód 5 - Definice třídy widgetu používající provider (vlastní)

V kódu *Kód 5 - Definice třídy widgetu* používající provider je definice třídy, které je umožněno providera používat a aktualizovat se dle měnícího se stavu providera. Třída musí dědit z třídy doplňku *Riverpod*, z třídy *ConsumerWidget* (řádek 1). *Widget*, který třída vykresluje pak poskytuje referenci, ze které lze k providerům přistoupit (řádek 3).

```
1 @override
2 Widget build(BuildContext context, WidgetRef ref) {
3   final favDeviceIDs = ref.watch(favoritesProvider).deviceIDs;
4   ref.read(favoritesProvider.notifier)
5       .toogleDevice(deviceId: device.id);
6   return ChildWidget();
7 }
```

Kód 6 - Použití reference ve widgetu (vlastní)

V kódu *Kód 6 - Použití reference ve widgetu* je ukázáno, jak se reference v aplikaci využívá. Metodou *watch* naslouchám změnám stavu určité proměnné providera (řádek 3). Při změně tohoto stavu se se *widget*, který stav využívá, překreslí.

Metodou *read* čteme providera a přistupujeme tak k metodám měnící stav providera (řádek 4). Můžeme volat metodu providera, která v logickém algoritmu změní stav jeho proměnných (řádek 5).

4.3.2 Definice hlavního *widgetu* aplikace

```
1 MaterialApp.router(
2   title: 'App name',
3   localizationsDelegates: AppLocalizations.localizationsDelegates,
4   supportedLocales: AppLocalizations.supportedLocales,
5   theme: CustomAppTheme.getLightTheme(visualDensity: density),
6   darkTheme: CustomAppTheme.getDarkTheme(visualDensity: density),
7   themeMode: ThemeMode.system,
8   routerConfig: _router,
9 )
```

Kód 7 - Definice hlavního widgetu aplikace (valstní)

V kódu *Kód 7 - Definice hlavního widgetu aplikace* byla definován *widget* knihovny *Material*, *MaterialApp*. Aplikace využívá navigaci na obrazovky přes *router* (řádek 1).

V kódu jsou definovány parametry aplikace, jako jsou titulek (řádek 2), lokalizace (řádek 3) a podporované jazyky pro vícejazyčnost (řádek 4), téma aplikace pro světlý (řádek 5) a tmavý (řádek 6) režim s automatickým přepínáním dle systému (řádek 7) a konfigurace *routeru* (řádek 8).

4.3.2.1 *Navigování v aplikaci mezi obrazovkami*

```
1  final GoRouter _router = GoRouter(
2    routes: <RouteBase>[
3      GoRoute(
4        path: '/',
5        builder: (BuildContext context, GoRouterState state) {
6          return const MainScreen();
7        },
8        routes: <RouteBase>[
9        ...
10       ],
11     ),
12   ],
13   errorBuilder: (context, state) => const ErrorScreen(),
14 );
```

Kód 8 - Definice konfigurace routeru (vlastní)

V kódu *Kód 8 - Definice konfigurace routeru* je uvedena definice konfigurace využitého *routeru* v aplikaci. Jako *router* byl použit doplněk *GoRouter* ve verzi 6.0.2. *GoRouter* má x *GoRoute* cest *routes* (řádek 2), které mají y pod cest *routes* (řádek 8).

V aplikaci byla definována pouze jedna cesta bez parametru s odkazem na hlavní obrazovku *MainScreen* (řádek 6). Pokud chce uživatel přistoupit na cestu jež není definována, bude přesměrován na chybovou obrazovku *ErrorScreen*.

```
1  <RouteBase>[
2    GoRoute(
3      path: 'detail',
4      builder: (BuildContext context, GoRouterState state) {
5        final content = state.extra as DetailScreenContent;
6        return DetailScreen(
7          content: content,
8        );
9       },
10     redirect: (context, state) {
11       if (state.extra == null) return '/';
12       return null;
13     },
14   ),
15 ]
```

Kód 9 - Definice pod cesty routeru (vlastní)

V kódu *Kód 9 - Definice pod cesty routeru* je definována pod cesta detail (řádek 3). Tato cesta směřuje uživatele na obrazovku *DetailScreen*. Pod cesta má parametr, jejíž kontent se předává obrazovce. Pokud uživatel chce přistoupit na pod cestu bez parametru, je přesměrován na začátek cesty, tedy obrazovku *MainScreen*.

4.3.2.2 *Lokalizace a podporované jazyky*

Aplikace byla vytvořena pro podporu pouze jazyka čeština s možností rozšíření pro další. Pro lokalizaci byla použita lokalizace *l10n*, která je obsažena *SDK Flutter*.

```
1 arb-dir: lib/l10n
2 template-arb-file: intl_cs.arb
3 output-localization-file: app_localizations.dart
```

Kód 10 - Konfigurace l10n (vlastní)

V kódu *Kód 10 - Konfigurace l10n* je uveden obsah souboru *l10n.yaml*, tedy konfigurace lokalizace *l10n*. Je konfigurována složka (řádek 1), soubor s překlady pro vícejazyčnost (řádek 2) a výstup lokalizace (řádek 3).

```
1 {
2   "blinder_type_title": "žaluzie",
3   "@blinder_type_title": {},
4   "door_type_title": "dveře",
5   "@door_type_title": {},
6   ...
7 }
```

Kód 11 - Příklad definice překladů (vlastní)

V kódu *Kód 11 - Příklad definice překladů* je uveden příklad definice překladu pro jazyk čeština. Soubor s českým překladem je v aplikaci pojmenován *intl_cs.arb*.

Každá položka pro překlad má svůj klíč a hodnotu. Hodnota obsahuje překlad pro dan jazyk.

```
1 title = context.l10n.blinder_type_title;
```

Kód 12 – Použití lokalizace l10n (vlastní)

V kódu *Kód 12 – Použití lokalizace* l10n je uveden příklad použití hodnoty klíče z překladu. Příklad je uveden v programovacím jazyce *Dart*, hodnota se tedy volá v aplikaci pro statické texty aplikace. Pokud bude aplikace přepnuta do jiného jazyka, který bude v aplikaci definován a přeložen, pro daný klíč bude vrácena hodnota v daném jazyku.

4.3.2.3 *Možné typy zařízení*

```
1  class DeviceType {
2    DeviceType._();
3
4    static final bool _isIOS = UniversalPlatform.isIOS;
5    static final bool _isAndroid = UniversalPlatform.isAndroid;
6    static bool isWeb = UniversalPlatform.isWeb;
7    static bool get isMobile => _isAndroid || _isIOS;
8
9    static bool isSmallScreen(BuildContext context) {
10     if (MediaQuery.of(context).size.width <
11         Consts.screenBreakpointWidht) {
12       return true;
13     }
14     return false;
15   }
16 }
```

Kód 13 - Definice třídy typu zařízení (vlastní)

Kód 13 - Definice třídy typu zařízení definuje třídu, ze které je získávána informace, na jakém typu zařízení je aplikace spuštěna. V kódu aplikace se pracuje s informací, zda aplikace je spuštěna na webovém rozhraní (řádek 6), na mobilním telefonu s operačním systémem iOS nebo Android (řádek 7), nebo zda je šíře obrazovky menší, než určitá velikost (řádek 9).

4.3.2.4 *Téma aplikace*

```
1  static ThemeData getLightTheme(
2      {required VisualDensity visualDensity}) {
3    final colorScheme = _getLightThemeColorScheme();
4
5    return ThemeData(
6      brightness: Brightness.light,
7      visualDensity: visualDensity,
8      useMaterial3: true,
9      colorScheme: colorScheme,
10     scaffoldBackgroundColor: colorScheme.background,
11     appBarTheme: _getAppBarTheme(colorScheme),
12     chipTheme: _getChipThemeData(colorScheme),
13     iconTheme: _getIconThemeData(colorScheme),
14     textTheme: _getTextTheme(colorScheme),
15     filledButtonTheme: _getFilledButtonThemeData(colorScheme),
16   );
17 }
```

Kód 14 - Definice světlého tématu aplikace (vlastní)

Kód 14 - Definice světlého tématu aplikace je funkcí jejímž výstupem je téma aplikace. Jedná se o funkci pro světlé téma.

Jako parametr funkce dostává *visualDensity*, což je hodnota nastavená dle typu zařízení/velikosti obrazovky. Pro mobilní zařízení je hodnota jiná než pro web.

V objektu *ThemeData* byly nastaveny parametry tématu pro vrácení z funkce. Nastavuje se světlý/tmavý režim (řádek 6), barevné schéma (řádek 9), pozadí *widgetu Scaffold* (řádek 10) a témata jednotlivých *widgetů* knihovny *Material* (řádky 11-15).

Tmavé schéma aplikace je definováno funkcí *getDarkTheme*.

```
1  static ColorScheme _getLightThemeColorScheme() {
2    return const ColorScheme(
3      brightness: Brightness.light,
4      primary: ConstColors.light,
5      onPrimary: ConstColors.dark,
6      secondary: ConstColors.lightSec,
7      onSecondary: ConstColors.dark,
8      error: ConstColors.red,
9      onError: ConstColors.red,
10     background: ConstColors.light,
11     onBackground: ConstColors.dark,
12     surface: ConstColors.light,
13     onSurface: ConstColors.dark,
14   );
15 }
```

Kód 15 - Definice světlého barevného schématu (vlastní)

Kód 15 - Definice světlého barevného schématu je funkce vracející objekt barevného schématu pro světlý režim. Ve funkci se nastavují parametry schématu, jako jsou světlost a jednotlivé barvy. Barvy jsou definovány ve třídě s konstantami pro aplikaci.

Pro tmavý režim existuje funkce *_getDarkThemeColorScheme*, jejíž hodnoty pro parametry objektu barevného schéma jsou odlišné – barvy pro tmavé schéma.

```
1 static ChipThemeData _getChipThemeData(
2     ColorScheme colorScheme) {
3   return ChipThemeData(
4     shadowColor: Colors.transparent,
5     backgroundColor: colorScheme.secondary,
6     iconTheme: _getIconThemeData(colorScheme),
7     side: BorderSide.none,
8   );
9 }
```

Kód 16 - Téma widgetu knihovny Material (vlastní)

Kód 16 - Téma widgetu knihovny Material je ukázkou definování téma *widgetu* knihovny *Material* pro celou aplikaci. Tímto kódem byla nastavena výchozí hodnota *widgetu Chip*.

4.3.3 Rozvržení obrazovek

```
1 Row(
2   crossAxisAlignment: CrossAxisAlignment.start,
3   children: [
4     if (!isSmallScreen) const LeftMenuWidget(),
5     Expanded(child: bodyWidget),
6   ],
7 )
```

Kód 17 - Rozvržení obrazovek (vlastní)

Každá obrazovka se primárně skládá ze dvou *widgetů* v horizontální rovině. Jimi jsou levé menu a obsah obrazovky. Je-li šíře celkového zobrazení menší než určitá hodnota z konstant, levé menu není zobrazováno.

4.3.3.1 *Hlavní obrazovka aplikace*

```
1  final userConfig = ref.watch(userConfigProvider).userConfig;
2  final cfgIsLoading = ref.watch(userConfigProvider).isLoading;
3
4  final currentHome = ref.watch(currentHomeProvider).currentHome;
5  final homeIsLoading = ref.watch(currentHomeProvider).isLoading;
6
7  final favIsLoading = ref.watch(favoritesProvider).isLoading;
8
9  unawaited(
10   ref.read(userConfigProvider.notifier)
11       .parseDummyConfig(ctx: context, widgetRef: ref),
12 );
```

Kód 18 - Využití providerů na hlavní obrazovce (vlastní)

Kód 18 - Využití providerů na hlavní obrazovce ukazuje, jak je na hlavní obrazovce využito providerů. Obrazovka naslouchá na změnu stavu uživatelské konfigurace (řádek 1), stavu načítání uživatelské konfigurace (řádek 2), stavu ovládaného inteligentního domu (řádek 4), stavu načítání ovládaného inteligentního domu (řádek 5), stavu načítání identifikátorů oblíbených zařízení (řádek 7). Obrazovka při své inicializaci zpracuje a uživatelskou konfiguraci z lokálního souboru (řádky 9-12).

Pokud jeden ze stavů načítání má hodnotu *pravda*, je zobrazen *widget* načítání pro informování uživatele a práci aplikace na pozadí. Pokud žádný ze stavů načítání nemá hodnotu *pravda*, ale *lež* a zároveň ovládaný inteligentní dům není nulové hodnoty, jsou načteny *widgety* obrazovky.

4.3.3.2 *Vedlejší obrazovka aplikace*

```
1  class DetailScreenContent {
2    const DetailScreenContent(
3      {required this.title, required this.devices});
4    final String title;
5    final List<DeviceModel> devices;
6  }
7
8  class DetailScreen extends ConsumerWidget {
9    const DetailScreen({required this.content, super.key});
10
11   final DetailScreenContent content;
12   ...
13 }
```

Kód 19 - Atributy vedlejší obrazovky (vlastní)

Vedlejší obrazovka jako parametr dostává zařízení, jejichž ovládací a informační *widgety* zobrazuje. Atributy vedlejší obrazovky jsou v *Kód 19 - Atributy vedlejší obrazovky*.

4.3.4 Komunikace s *PLC* inteligentního domu

PLC, s nímž aplikace komunikuje a jehož zařízení klientská aplikace ovládá, komunikuje komunikačním protokolem WebSocket. Jeho *API* je tedy pro tento komunikační protokol vystaveno a v aplikaci byl implementován doplněk *Doplněk web_socket_channel* ve verzi 2.3.0.

4.3.4.1 *Pomocná třída WebSocketHeplper*

```
1  class WebSocketHelper {
2    factory WebSocketHelper() {
3      return _webSocketHelper;
4    }
5
6    WebSocketHelper._internal();
7    static final WebSocketHelper _webSocketHelper =
8                      WebSocketHelper._internal();
9
10   WebSocketChannel? _channel;
11
12   WebSocketChannel connectWebSocket(String? address) {}
13
14   void wsUpdateDevice(String deviceId, Map<String, dynamic> props) {}
15
16   void closeWebSocket() {}
17 }
```

Kód 20 - Definice pomocné třídy WebSocketHelper (vlastní)

Kód 20 - Definice pomocné třídy WebSocketHelper ukazuje, jak se s *PLC* přes WebSocket komunikuje. Jedná se o třídu návrhového typu singleton (řádky 2-8). Třída má pouze jeden atribut *_channel*, který lokální a který uchovává instanci otevřeného komunikačního kanálu s *PLC*. Třída má metody k připojení se k *PLC connectWebSocket* (řádek 12), komunikování zpráv na *PLC wsUpdateDevice* (řádek 14) a uzavření komunikačního kanálu s *PLC closeWebSocket* (řádek 16).

```
1 WebSocketChannel connectWebSocket(String? address) {
2     final wsUrl = Uri.parse('ws://$address');
3     _channel = WebSocketChannel.connect(wsUrl, protocols: ['devs']);
4     _channel?.sink.add('{"intent": "list"}');
5     return _channel!;
6   }
```

Kód 21 - Definice metody připojení PLC (vlastní)

Metodou v *Kód 21 - Definice metody připojení PLC* se aplikace připojuje k *PLC* komunikačním protokolem WebSocket. Metoda má vstupní parametr *address*, *IP* adresu *PLC*. V metodě se vytváří *URL* adresa (řádek 2) a kanál otevřené komunikace s *PLC* (řádek 3). Lokální proměnná třídy je nastavena právě tímto kanálem. Následně komunikován požadavek stavů všech zařízení na *PLC* (řádek 4) a instance otevřeného kanálu je metodou vrácena.

```
1 void wsUpdateDevice(String deviceId, Map<String, dynamic> props) {
2     final message = jsonEncode(
3       PlcModelPost(payload: [Payload(id: deviceId, props: props)]),
4     );
5     _channel?.sink.add(message);
6   }
```

Kód 22 - Definice metody komunikace do PLC (vlastní)

Metodou *Kód 22 - Definice metody komunikace do PLC* aplikace komunikuje změnu stavu zařízení do *PLC*. Parametry má *deviceId*, řetězec znaků identifikující konkrétní zařízení *PLC*, a *props*, mapu klíčů a jejich hodnot. Tato mapa je pro každý typ zařízení specifická, proto ji nelze definovat jako třídu (model) s určitými atributy. V metodě je sestavena zpráva, řetězec znaků (řádek 2), která je následně komunikována WebSocket kanálem do *PLC* (řádek 5).

```
1 void closeWebSocket() {
2     _channel?.sink.close();
3   }
```

Kód 23 - Definice metody uzavření WebSocket kanálu (vlastní)

Metodou *Kód 23 - Definice metody uzavření WebSocket kanálu* aplikace uzavírá WebSocket kanál na *PLC* (řádek 2).

4.3.4.2 *Řízení stavu zařízeních komunikovaných z PLC*

```
1  final webSocketProvider =
2      StreamProvider.autoDispose<Map<String?, Map<String, dynamic>>>(
3          (ref) async* {
4    final currentHome = ref.watch(currentHomeProvider).currentHome;
5    final homeIsLoading = ref.watch(currentHomeProvider).isLoading;
6    final devicesToProcess = <String?, Map<String, dynamic>>{};
7
8    if (homeIsLoading) return;
9    final channel =
10       WebSocketHelper().connectWebSocket(currentHome?.plcIP);
11
12   ref.onDispose(() => WebSocketHelper().closeWebSocket());
13
14   await for (final value in channel.stream) {
15     final listMap =
16             jsonDecode(value.toString()) as Map<String, dynamic>;
17     final list = PlcModelList.fromJson(listMap);
18
19     list.list?.forEach((element) {
20       devicesToProcess[element.id] = element.props;
21     });
22
23     list.diff?.forEach((element) {
24       devicesToProcess[element.id] = element.props;
25     });
26
27     yield devicesToProcess;
28   }
29 });
```

Kód 24 - Definice provideru pro komunikace s PLC (vlastní)

Kód *Kód 24 - Definice provideru pro komunikace s PLC* je definicí provideru, který obhospodařuje komunikaci s *PLC* komunikačním protokolem WebSocket.

Provider reaguje na změny provideru *currentHomeProvider*, přesněji na, v aplikaci, aktuálně zvolený inteligentní dům (řádek 4).

Při volbě inteligentního domu se pro daný dům otevírá komunikační kanál WebSocketu, *channel* (řádek 9). Při likvidaci provideru je kanál uzavřen (řádek 12).

Při obdržení nové zprávy kanálem od *PLC* probíhá její zpracování (řádky 14-28). Ve zpracování jsou hodnoty zprávy dekódovány do objektu (řádek 15) a následně je jimi naplněn příslušný model (řádek 18). Tím se aktualizují hodnoty všech zařízeních a mění stav provideru, na který reagují jednotlivé *widgety* zařízeních (řádek 27).

4.3.4.3 *Příkladné použití provideru webSocketProvider*

```
1  @override
2    Widget build(BuildContext context, WidgetRef ref) {
3      final message = ref.watch(webSocketProvider);
4      PlcModelLight? light;
5
6      return message.when(
7        loading: () => const PlatformActivityIndicator(),
8        error: (err, stack) => Text('Error: $err'),
9        data: (devices) {
10       ...
11       },
12     );
13   }
```

Kód 25 - Použití provideru pro komunikaci s PLC (vlastní)

V *Kód 25 - Použití provideru pro komunikaci s PLC* je uvedeno příkladné použití provideru *webSocketProvider* ve *widgetu* zařízení typu světlo. Je vytvořeno naslouchání pro zprávu (řádek 3), na kterou je následně reagováno (řádek 6). Zpráva má více stavů:

- načítání – je vrácen *widget* načítání (řádek 7),
- chyba – je zobrazen *widget* textu s popisem chyby (řádek 8),
- úspěšná data – zpráva je zpracována a je zobrazen *widget* dle typu zařízení (řádek 9).

4.3.5 Verzování aplikace

```
1 ...
2 version: 1.0.2+3
3 ...
```

Kód 26 - Verze aplikace v pubspec.yaml souboru (vlastní)

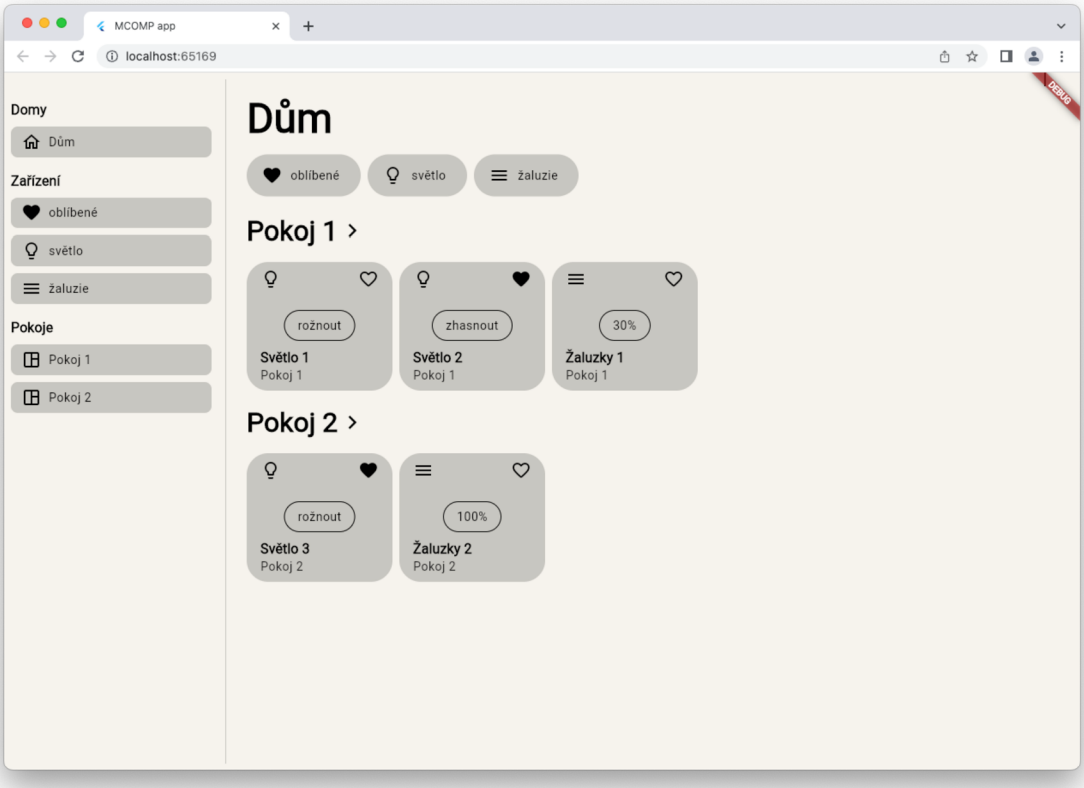
Flutter aplikace je verzována v jejím konfiguračním souboru *pubspec.yaml*. V souboru se nachází parametr *version*, který je ve formátu *w.x.y+z*, kde *w* je označení MAJOR verze, *x* MINOR, *y* PATCH a *z* číslo určující o jaký build v pořadí se jedná. Postup verzování následoval poznatky z kapitoly *Sémantické verzování 2.0.0*.

5 Výsledky a diskuse

V teoretické části diplomové práce byly analyzovány odborné zdroje. Poznatky z nich byly uplatněny v části praktické.

Návrhu nového řešení klientské aplikace předcházela analýza řešení stávajícího. Informační architektura nového řešení je jednotná a jednoduší pro snazší ovládání aplikace uživatelem.

Nové řešení aplikace je cross-platformové, aplikace má tedy pouze jednu kódovou základnu v jazyce Dart a za pomoci *SDK* Flutter je sestavena pro platformy Android, iOS a web. Na webu je responzivní dle nejrůznější velikosti výstupního zobrazovacího displeje. Původní řešení responzivní zcela není, pro porovnání viz kapitola *Aplikace na různých platformách.*



Obrázek 26 - Nové řešení aplikace v prohlížeči, desktop (vlastní)

Pro mobilní aplikaci byla implementována vlastnost, kdy obsah, se kterým uživatel může interagovat je vždy dostupný do poloviny obrazovky s možností vyrolování výše. Tato vlastnost usnadňuje ovládání aplikace jednou rukou u mobilních zařízení s větší obrazovkou.

Obrázek 27 - Nové řešení aplikace v prohlížeči, mobil (vlastní)

Aplikace komunikuje s *PLC* skrze komunikační protokol WebSocket a *widgety* pro zařízení v aplikaci se automaticky aktualizují po změně jejich stavu. Změna stavu může nastat jak z klientské aplikace, tak z komunikačního kanálu s *PLC*. V prototypu aplikace byly implementovány *widgety* pro zařízení typu světlo a typu žaluzie. Levá horní ikonka widgetu značí jeho typ, ta pravá, zda je přiřazeno mezi oblíbené (plná výplň), či nikoliv. Přiřazení a odebrání mezi oblíbená zařízení lze měnit stiskem/klikem na ikonku.

Obrázek 28 - Widget zařízení typu světlo (vlastní)

Stiskem tlačítka ve středu *widgetu* „rožnout“ měníme stav zařízení. Zařízení světlo je nyní ve vypnutém stavu.

Obrázek 29 - Widget zařízení typu žaluzie (vlastní)

Stiskem tlačítka ve středu *widgetu* „30%“ měníme stav zařízení, hodnotu stažení žaluzií. Zařízení typu žaluzie je ve stavu, kdy jsou žaluzie staženy na 30%.

Ovládací prvky ve widgetech pro zařízení jsou v prototypu aplikace pouze pro testování funkcionality. Je doporučeno interagovat a stav zařízení komunikovat směrem k uživateli za použití ikon a grafického vyobrazení.

Aplikace se nastavuje za pomoci konfiguračního souboru, který je pro prototyp její součástí. Konfigurační soubor nese informace o uživateli a inteligentních domech, které má přiřazeny. Dům má své informace včetně *IP* adresy, na kterou je navázána komunikace aplikace. Dům má zařízení s identifikátory, které zařízení odlišují a rozpoznávají na straně *PLC*. Budoucí možné vylepšení je umístění konfiguračních souborů zvlášť na server a namísto lokálního uchovávání v aplikaci ho stahovat – aplikace tak dosáhne maximální univerzálnosti.

Aplikace komunikuje nešifrovanou WebSocket komunikací, což v rámci prototypu není problém. Pro budoucí využití a produkční nasazení je nutné komunikaci zabezpečit, ideálně standardem, který *PLC* poskytuje – pokud takový existuje.

Nové řešení aplikace je cross-platformové, má pouze jednu kódovou základnu, a to umožňuje přehledně evidovat jednotlivé verze aplikace. Verzování u předchozího řešení nebylo možné. Každé *PLC* mělo na svém web serveru vlastní klientskou aplikaci.

6 Závěr

Hlavním cílem diplomové práce byl návrh a implementace cross-platformové klientské aplikace pro vzdálené ovládání *PLC* inteligentního domu. Dle návrhu vyvinutá aplikace s *PLC* komunikuje skrze aplikační rozhraní *PLC* a komunikační protokol WebSocket. Prototyp vyvinuté aplikace je náhradou aplikace stávající, tedy náhradou různých a různě programovaných webových klientů hoštěných vždy na web serveru *PLC*, se kterým aplikace komunikuje. Nové řešení aplikace je cross-platformové, je tedy vylepšením směrem do kompatibility na různých mobilních platformách a na obrazovkách různých rozměrů. Nové řešení je nezávislé na jednotlivých *PLC* inteligentních domů, konfiguruje se dle vlastního konfiguračního souboru a je nasazeno na prostředích dle platformy.

V teoretické části diplomové práce byly analyzovány odborné zdroje týkající se vývoje a funkčnosti řešení stávajícího, technologie pro vývoj cross-platformové aplikace *SDK* Flutter, sdílených doplňků pro *SDK* Flutter, objektově orientovaného programování, standardu UML, sémantickému verzování a návrhu uživatelského rozhraní.

Z teoretické části bylo čerpáno v části praktické, kde byla dokumentována analýza stávajícího řešení, návrh aplikační architektury nového řešení, návrh diagramu tříd ve standardu UML pro nové řešení, návrh uživatelského rozhraní nového řešení a použití sdílených balíčků v *SDK* Flutter pro implementaci prototypu cross-platformové klientské aplikace, komunikující s *PLC* inteligentního domu přes jeho aplikační rozhraní.

Při návrhu byl dodržován standard UML. Tato aplikace je funkční na mobilních platformách (Android, iOS) a na web serveru (aplikace dostupná z webu).

7 Seznam použitých zdrojů

Teco a.s. 2010. ZAČÍNÁME V PROSTŘEDÍ MOSAIC. *Teco a.s.* [Online] duben 2010. https://www.tecomat.cz/modules/DownloadManager/download.php?alias=txv00320_01_mosaic_progstart_cz.
—. 2007. Programování PLC podle normy IEC 61 131-3 v prostředí Mosaic. *Teco a.s.* [Online] listopad 2007. https://www.tecomat.cz/modules/DownloadManager/download.php?alias=txv00321_01_mosaic_progiec_cz.
—. 2016. Knihovna CanvasObjectsLib. *Teco a.s.* [Online] duben 2016. https://www.tecomat.cz/modules/DownloadManager/download.php?alias=txv00397_01_mosaic_canvasobjectslib.
—. 2013. Nástroj WebMaker. *Teco a.s.* [Online] březen 2013. https://www.tecomat.cz/modules/DownloadManager/download.php?alias=txv00328_01_mosaic_webmaker_cz.
—. 2015. Knihovna CanvasLib. *Teco a.s.* [Online] duben 2015. https://www.tecomat.cz/modules/DownloadManager/download.php?alias=txv00389_01_mosaic_canvaslib_cz.
Google LLC. 2023. Flutter architectural overview. *Flutter documentation.* [Online] 2023. https://docs.flutter.dev/resources/architectural-overview.
—. 2023. Dart overview. *Dart documentation.* [Online] 2023. https://dart.dev/overview.
—. 2023. Using packages. *Flutter documentation.* [Online] 2023. https://docs.flutter.dev/development/packages-and-plugins/using-packages.
Rousselet, Remi. 2022. flutter_riverpod package. *pub.dev.* [Online] srpen 2022. https://pub.dev/documentation/flutter_riverpod/latest/.
Google LLC. 2023. InheritedWidget class. *Flutter API reference documentation.* [Online] 2023. https://api.flutter.dev/flutter/widgets/InheritedWidget-class.html.
Rousselet, Remi. 2023. Providers. *Riverpod.* [Online] 2023. https://riverpod.dev/docs/concepts/providers.
gskinner.com. 2022. universal_platform package documentation. *universal_platform package.* [Online] 2022. https://pub.dev/documentation/universal_platform/latest/.
dar.dev. 2022. intl package documentation. *intl package.* [Online] 2022. https://pub.dev/documentation/intl/latest/.
flutter.dev. 2022. go_router package documentation. *go_router package.* [Online] 2022. https://pub.dev/documentation/go_router/latest/index.html.
—. 2022. shared_preferences package documentation. *shared_preferences package.* [Online] 2022. https://pub.dev/documentation/shared_preferences/latest/.
Melnikov, A. 2011. The WebSocket Protocol. [Online] prosinec 2011. https://www.rfc-editor.org/rfc/rfc6455#section-4.2.2.
tools.dart.dev. 2023. web_socket_channel package documentation. *web_socket_channel package.* [Online] leden 2023. https://pub.dev/documentation/web_socket_channel/latest/.
Haider, Adibbin. 2021. *Evaluation of cross-platform technology Flutter from the user's perspective.* Stockholm : KTH ROYAL INSTITUTE OF TECHNOLOGY, 2021.
Kindler, Eugene and Krivy, Ivan. 2011. *Object-oriented simulation of systems with sophisticated control.* s.l. : Informa UK Limited, 2011. 0308-1079.
Teorey, Toby, et al. 2011. *Database Modeling and Design.* s.l. : Elsevier Science & Technology, 2011. 978-0-12-382020-4; 0-12-382020-0.
Preston-Werner, Tom. 2023. Sémantické verzování 2.0.0. *Sémantické verzování 2.0.0.* [Online] 2023. https://semver.org/lang/cs/.

Hartson, H. Rex and Pyla, Pardha S. 2019. *The UX book : agile UX design for a quality user experience.* s.l. : Cambridge, MA : Morgan Kaufmann, Elsevier, 2019. 978-0-12-805342-3.
Apple Inc. 2023. Human Interface Guidelines. *Apple Developer.* [Online] 2023. https://developer.apple.com/design/human-interface-guidelines/guidelines/overview.
Google LLC. 2023. Guidelines. *Material Design.* [Online] 2023. https://m2.material.io/design/guidelines-overview.
—. 2023. Set up an editor. *Flutter Documentation.* [Online] 2023. https://docs.flutter.dev/get-started/editor.
—. 2023. Material Components widgets. *Flutter Documentation.* [Online] 2023. https://docs.flutter.dev/development/ui/widgets/material.
—. 2023. Simple app state management. *Flutter Documentation.* [Online] 2023. https://docs.flutter.dev/development/data-and-backend/state-mgmt/simple.
Babich, Nick. 2020. The Beginner's Guide to Information Architecture in UX. *Xd Ideas.* [Online] listopad 24, 2020. https://xd.adobe.com/ideas/process/information-architecture/information-ux-architect/.
Soni, Devin. 2019. What Is a Singleton? *Medium.* [Online] 31. červenec 2019. https://betterprogramming.pub/what-is-a-singleton-2dc38ca08e92.
Ford, Neal. 2021. *Software architecture : the hard parts : modern trade-off analyses for distributed architectures.* Beijing; Boston; Farnham; Sebastopol; Tokyo : O'Reilly, 2021. 978-1-492-08689-5.
Prensky, Marc. 2001. *Digital Natives, Digital Immigrants* . místo neznámé : On the Horizon, 2001. 1074-8121.
StatCounter. 2023. Mobile Operating System Market Share Worldwide. *StatCounter.* [Online] 2023. https://gs.statcounter.com/os-market-share/mobile/worldwide.
Schedule and notes are the KEY!

8 Přílohy

Zdrojový kód prototypu cross-platformové aplikace pro ovládání PLC inteligentního domu skrze jeho aplikační rozhraní je dostupný v přiloženém .zip souboru a na přiloženém CD.