ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	iOS aplikace k ovládání 3D tiskáren
Student:	Josef Doležal
Vedoucí:	Ing. Miroslav Hrončok
Studijní program:	Informatika
Studijní obor:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Cílem práce je vytvořit mobilní iOS aplikaci, která bude komunikovat s OctoPrint serverem a pomocí něho ovládat 3D tiskárnu.

1. Navrhněte a implementujte iOS aplikaci, která bude pomocí OctoPrint REST API komunikovat s 3D tiskárnou. Využijte programovací jazyk Swift 3.
2. Aplikace musí implementovat všechny funkce OctoPrint API.
3. Aplikace automaticky nalezne dostupné OctoPrint servery na lokální síti.
4. Aplikace bude ukazovat video stream z OctoPrint serveru, pokud bude k dispozici.
5. Výsledná aplikace bude obsahovat optimalizované rozhraní pro zařízení iPhone a iPad.
6. Kód musí být open source, dobře strukturovaný, řádně otestovaný, vhodně okomentovaný a dokumentovaný (obojí v anglickém jazyce).

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 7. prosince 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra informačních technologií

Bakalářská práce

iOS aplikace k ovládání 3D tiskáren

Josef Doležal

Vedoucí práce: Ing. Miroslav Hrončok

15. května 2017

Poděkování

V této části bych rád poděkoval každému, kdo mi během této práce pomohl. Nejprve Miroslavovi Hrončokovi, vedoucímu mé práce, za podnětné rady, pomoc s korekturou a uvedení do problematiky 3D tisku.

Děkuji Tomášovi Sýkorovi za cenné rady v implementační části, za pomoc s automatizací testování a kompilace a za propůjčení vývojářských certifikátů pro podepisování aplikace.

Děkuji také své rodině za podporu a pevné nervy.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 15. května 2017 .....

Odkaz na tuto práci

Doležal, Josef. *iOS aplikace k ovládání 3D tiskáren.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017. Dostupný také z WWW: ⟨`https://github.com/josefdolezal/fit-bi-bap`⟩.

Abstrakt

Bakalářská práce se zabývá vytvořením mobilní aplikace pro ovládání 3D tiskárny. Cílem projektu je naprogramování aplikace, která uživatelům zjednoduší ovládání 3D tiskárny z mobilních zařízení s operačním systémem iOS. Důležitou součástí řešení je možnost přidání síťové tiskárny s minimální konfigurací. Z důvodu možnosti rozšíření funkcionalit v budoucnu jsou zdrojové kódy volně dostupné.

Klíčová slova mobilní aplikace, 3d tisk, octoprint, iOS, Swift, open source

Abstract

Bachelor thesis aims to create an application to control 3D printers. The application simplifies control of 3D printers from devices running iOS operating system. An important part of the solution is the ability add new printer with a minimum configuration steps. Application is available as open source to allow community to possibly add new functions in the future.

Keywords mobilní aplikace, 3d tisk, octoprint, iOS, Swift, open source

Obsah

Seznam obrázků

Seznam ukázek kódu

Úvod

V posledních letech můžeme být svědky velkého technologického rozmachu. Nedílnou součástí tohoto rozmachu jsou mobilní aplikace a také 3D tiskárny. V oblasti 3D tiskáren se popularita zvýšila za posledních pět let více než osmkrát (měřeno dle zájmu o téma v Google vyhledávání) [2]. Mnohem většího úspěchu dosáhly mobilní zařízení, která v září 2016 poprvé přesáhly využívání osobních počítačů [3].

I navzdory těmto faktům je v současné době využívání a ovládání 3D tiskáren doménou zejména počítačů. O tom svědčí především rozšíření mobilních aplikací. Ty jsou v současné době pouze dvě, jejich funkcionalita je navíc oproti desktopové verzi značně omezená.

Ve své bakalářské práci se zabývám návrhem mobilní aplikace pro ovládání 3D tiskáren prostřednictví rozhraní OctoPrint, její analýzou a implementací. Práci uvádím kapitolou *3D tisk*, která čtenáře stručně uvede do problému. V kapitole *Analýza* zkoumám možné technologie a paradigmata. Použité technologie následně zmiňuji v kapitole o implementaci. V části věnující se implementaci popisuji, jaké funkcionality aplikace obsahuje a způsob jakým jsem je do aplikace zakomponoval. O udržení kvality aplikace a využití automatizace v průběhu vývoje mluvím v kapitole *Testování*.

Výsledný zdrojový kód je volně šířený s možností libovolných úprav pod licencí MIT.[1] Přestože aplikace není distribuována standardní cestou obchodem App Store, je na tuto formu distribuce plně připravena. Jednotilivé verze aplikace jsou v tuto chvíli distribuovány pouze registrovaným vývojářům pomocí Fabric Beta.[2]

[1] Licence uvedena na https://github.com/3DprintFIT/octoprint-ios-client/blob/dev/LICENSE.md

[2] Komerční platforma umožňující distribuci aplikace vývojářům mimo služby Apple

Kapitola 1

3D tisk

V této kapitole stručně zmiňuji princip a technologii 3D tisku. Čtenáře uvedu také do stávajícího stavu z pohledu ovládání 3D tiskáren a využití mobilních aplikací k tomuto účelu.

1.1 Technologie 3D tisku

3D tisk je proces, při kterém lze z digitálního modelu vytvořit reálný objekt. Tyto objekty vznikají postupným nanášením jednotlivých vrstev modelu na sebe ve vertikálním směru. Jedna z metod využívajících vrstvení materiálu se nazývá Fused deposition modeling (zkráceně FDM). Princip FDM spočívá v tavení plastu (případně jiných materiálů) pomocí tiskové hlavy, která následně nanáší taveninu ve vrstvách na sebe [4]. Tato tavenina se nazývá filament.

Takto vytvořené plastové objekty mají velkou výhodu v nízkých nákladech na výrobu. Jsou tedy ideálním prostředkem k výrobě prototypů nebo produkování omezeného množství výrobků [5].

1.2 OctoPrint – Ovládání 3D tiskárny

Tisk lze obsloužit pomocí mnoha aplikačních rozhraní. Velmi oblíbeným nástrojem je OctoPrint, který nabízí ovládání pomocí webového prohlížeče. Jedná se tedy o webové rozhraní, které reprezentuje uživateli příkazy tiskárny pomocí tlačítek či jiných grafických prvků. Z prohlížeče je tak možné měnit základní vlastnosti jako např. teploty, průběh tisku či nastavení připojení k tiskárně. Bohužel je ale toto rozhraní zcela zaměřeno na použití z počítače. Ovládací prvky nejsou dostatečně veliké a manipulace s nimi přináší velmi špatnou uživatelskou zkušenost na mobilních zařízeních.

Kromě webového rozhraní nabízí OctoPrint také Application Programming Interface (API) pro aplikace třetích stran. Pro implementaci své aplikace využiji právě tohoto rozhraní.

V době psaní práce je dostupná verze API 1.3, která oproti předchozí verzi nabízí nové funkce jako např. správu uživatelů či nastavení systému. Některé z nich ale sama autorka označuje za experimentální [6]. Pro mobilní aplikaci navíc nejsou stěžejní. Z tohoto důvodu jsem se rozhodl využít starší verzi 1.2.15, která většinu funkcí označuje jako stabilní.

1.3 Tisk modelů

Modely jsou vytvářeny pomocí 3D modelovacích nástrojů. Z nich mohou být vyexportovány do formátu `STL`. Tento formát popisuje třírozměrnou povrchovou geometrii modelu. [7]

Model je následně potřeba převést do formátu `GCode`. To je jazyk, který popisuje jakým způsobem se bude tisková hlava pohybovat během tisku. [8]

1.4 Definice pojmu tisk

Přestože se obvykle slovem tisk rozumí běžný papírový tisk pomocí inkoustové nebo laserové tiskárny, ve své práci tento druh tisku vůbec neuvažuji a pojmem tisk rozumím vždy 3D tisk. Analogické předefinování si dovolím i u pojmů odvozených.

Ve své práci uvažuji jako metodu tisku pouze FDM zmíněnou v části 1.1 o technologiích tisku.

Kapitola 2

Analýza a rešerše

2.1 Programovací jazyk

V současné době lze pro vývoj mobilní aplikace využít mnoho programovacích jazyků. Nejpoužívanějšími se jeví `ReactNative` společnosti Facebook, `Xamarin` společnosti Microsoft a dále `Swift` a `Objective-C` od firmy Apple.

V souvislosti s nejnovější trendy [9] mě nejvíce oslovily nativní programovací jazyky. S ohledem na zadání práce jsem proto vybral programovací jazyk `Swift`.

2.2 Architektura aplikace

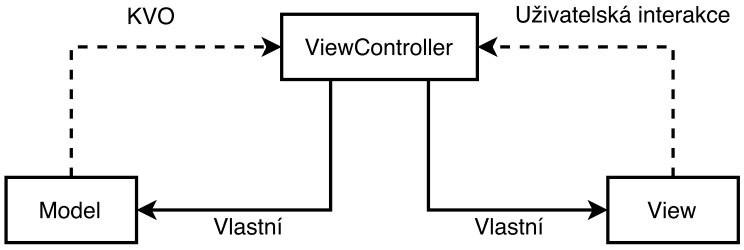
Jako doporučenou architekturu aplikací pro platformu iOS uvádí Apple Model-View-Controller (zkráceně MVC) [10]. Přestože se MVC pro vývoj aplikací jeví jako nejpopulárnější, rozhodl jsem se před začátkem implementace prozkoumat i jiné existující architektury. Z alternativních architektur jsem nakonec zvolil Model-View-ViewModel, kterou porovnám s doporučeným MVC. V závislosti na výsledku porovnání zvolím ideální architekturu pro svou aplikaci.

2.2.1 MVC: Model-View-Controller

Tato architektura rozděluje aplikaci do tří vrstev: `Model`, `View` a `Controller`. Názvy vrstev se běžně do českého jazyka nepřekládají. Jejich významu se věnuji níže.

Popis architektury

Model Reprezentuje perzistentní objekty, které aplikace využívá pro vnitřní logiku a prezentaci dat uživateli. Každý modelový objekt může být v relaci s libovolným počtem jiných modelových objektů. Tato vrstva je často reprezentována databází, příkladem mohou být databáze `Core Data`,



Obrázek 2.1: Architektura MVC

`Realm` nebo `SQLite`. Ukázka 2.1 prezentuje možnou podobu modelového objektu.

```
1  import Foundation
2
3  class Printer {
4      dynamic var url: URL
5
6      dynamic var displayName: String
7
8      init(url: URL, displayName: String) {
9          self.url = url
10         self.displayName = displayName
11     }
12 }
```

Ukázka kódu 2.1: Modelový objekt v architektuře MVC

View Jedná se o datový objekt viditelný uživatelem. `View` obsahuje logiku pro vykreslení a interakci s uživatelem. Přestože se `View` standardně používá pro zobrazení `Model` objektů nebo jejich úpravu, jsou od sebe tyto vrstvy striktně odstíněny. Na platformě iOS tuto vrstvu reprezentuje framework `UIKit` vytvořený Applem. Ukázkový kód 2.2 zobrazuje možnou implementaci `View`.

Controller Aplikační vrstva, která na základě vstupů z `View` aktualizuje a mění `Model` nebo překresluje `View` v případě, že zobrazovaná data už nejsou aktuální. Jedním z úkolů `Controlleru` je striktně zamezit přímé interakci mezi `View` a `Modelem`. Toto oddělení je zavedeno proto, aby `View` nemuselo znát konkrétní strukturu `Modelu` a aby `Model` nemusel obsahovat logiku formátování dat (cena, čas, ...) pro vykreslení. Dále se stará o navigaci mezi obrazovkami, síťování a interakci s uživatelem. Při rozdělení do obrazovek platí pravidlo, že jeden `Controller` obsluhuje jedno nebo více `View`. Ke korektnímu vykreslení `View` využívá libovoné

množství modelových objektů. O jednu obrazovku se typicky stará právě jeden `Controller`, je ale možné jich použít více. Zjednodušenou implementaci lze vidět v ukázce 2.3.

Z tohoto shrnutí vyplývá, že `Controller` je velmi blízce spjat s `View`. Toto propojení reprezentuje obrázek 2.2.

Modelový příklad použití

Pro možné porovnání architektury jsem připravil scénář stažení libovolných dat na základě požadavku uživatele. V MVC by se architektura chovala takto:

- Uživatel v aplikaci klikne na tlačítko „Stáhnout data“.
- Tuto interakci odchytí `View` a upozorní `Controller`.
- `Controller` na základě upozornění stáhne data a předá je `Modelu` k uložení.
- `Model` ukládá data a notifikuje `Controller` o změně.
- `Controller` aktualizuje `View`.
- Nastane-li během stahování chyba, `Controller` vytváří nové `View` a chybu prezentuje uživateli.

```
1  import UIKit
2
3  class PrinterView: UIView {
4
5      private let nameLabel: UILabel!
6      private let urlLabel: UILabel!
7
8      func configureView() { /* ... */ }
9
10 }
```

Ukázka kódu 2.2: View v architektuře MVC

Shrnutí

Shrneme-li vlastnosti vrstev, jejich klíčové role jsou:

- `Model` udává, jakým způsobem jsou data uložena,
- `View` se stará o správné vykreslení předformátovaných dat,

```swift
import UIKit

class PrinterDetailController: UIViewController {

    private var printerView: PrinterView!
    private var printer: Printer!

    override func viewDidLoad(animated: Bool) {
        super.viewDidLoad(animated: animated)

        if let printer = printer {
            printerView.nameLabel.text = printer.displayName
            printerView.urlLabel.text =
                printer.url.absoluteString
        } else { /* ... */ }
    }
}
```

Ukázka kódu 2.3: ViewController v architektuře MVC

- `Controller` se stará o ostatní logiku.

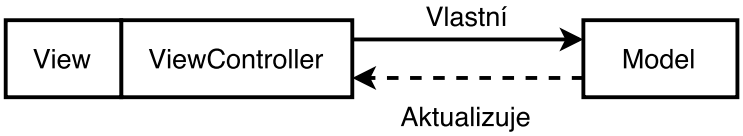
Pro zmíněné notifikace nabízí Apple řešení pomocí Delegate pattern návrhového vzoru. Tento návrhový vzor udává, že `Controller` musí naimplementovat specifické rozhraní, čímž se stane delegátem. Jako delegát se pak může zaregistrovat na notifikace obejktů, jejichž rozhraní implementoval.

Domnívám se, že MVC je v době psaní této práce nejpoužívanější architekturou a to především díky své jednoduchosti. Při tvorbě větších aplikací ale nemusí být vhodné. `Controller` se při nestandardním grafickém návrhu může stát velmi složitým, což výrazně snižuje jeho čitelnost a testovatelnost. Z tohoto důvodu se MVC občas přezdívá „Massive `View Controller`“ [11]. Kvůli přímému napojení `Controlleru` na `View` se při testování jeho chování (behavior testing) musí využít simulátoru mobilního operačního systému. Pro simulátor je navíc nutné naskriptovat uživatelův průchod aplikací, aby bylo možné testy automatizovat.

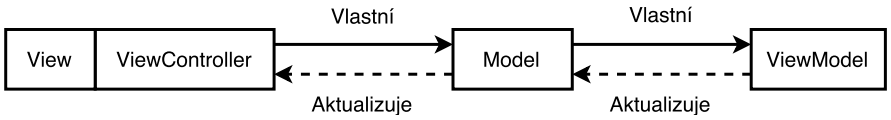
To zvyšuje časovou náročnost testování, dokonce v některých případech znemožňuje testování úplně (`Controlleru` nezle podvrhnout mock objekty). Tento problém se snaží řešit architektura Model-View-ViewModel (MVVM) od společnosti Microsoft [12].

2.2.2 MVVM: Model-View-ViewModel

Z důvodu nárustu nároků na mobilní aplikace se v posledních letech rozmáhá architektura MVVM. Tato architektura vychází ze zmíněného MVC a jejím základním úkolem je zjednodušit `Controller`. [12]



Obrázek 2.2: Controller spjatý s View



Obrázek 2.3: Architektura MVVM

Popis architektury

Za účelem zjednodušení `Controlleru` se ke stávajícím třem vrstvám přidává `ViewModel`, který se stará o přípravu dat z Modelu pro zobrazení a také o perzistenci změn.

ViewModel Objekt vlastněný `Controllerem` za pomoci kompozice. Pro `Controller` připravuje naformátované výstupy a poskytuje mu rozhraní pro vstupy. Výstupem se rozumí veškerá data, která jsou potřebná pro sestavení `View`. To může být např. datum ve specifickém formátu, cena včetně měny nebo informace o tom, kolik řádků bude obsahovat tabulka na obrazovce. Oproti MVC tedy perzistentní data nejsou viditelná `Controlleru`, ale pouze `ViewModelu`. Ten je nejdříve připraví pro zobrazení. Vstupem může být libovolná interakce uživatele: změna textu v textovém poli, stisknutí tlačítka, ale i fyzický pohyb telefonem (otočení obrazovky). Na základě vstupů spouští `ViewModel` svou vnitřní logiku a generuje výstupy. Jednoduchou implementaci `ViewModelu` bez použití vstupů lze vidět v ukázce 2.4.

Zodpovědnost `Controlleru` se zavedením `ViewModelu` dramaticky snižuje. V ideálním případě je `Controller` zodpovědný pouze za správné sestavení `View` a napojení zfromátovaných výstupů na něj. Dále pak za odchycení uživatelských interakcí a jejich propagaci do `ViewModelu`. Toto chování zachycuje obrázek 2.3.

Modelový příklad použití

Pro porovnání architektury s MVC lze opět využít scénář pro stažení dat. Pro tento scénář by se architektura MVVM chovala následovně:

- `Controller` napojuje výstupy `ViewModelu` na `View` a vytváří pravidla pro převod uživatelské interakce na vstupy `ViewModelu`.

```
class PrinterViewModel {
    var name: String {
        return printer?.displayName ?? "unknown"
    }

    var url: String {
        return printer?.url.absoluteString
            ?? "unknown"
    }

    private var printer: Printer?

    init() {
        self.printer = loadPrinter()
    }

    /* ... */
}
```

Ukázka kódu 2.4: ViewModel v architektuře MVVM

- Uživatel v aplikaci klikne na tlačítko „Stáhnout data“.
- `View` upozorňuje `Controller` na interakci uživatele, ten automaticky vytváří vstup pro `ViewModel`.
- `ViewModel` na základě vstupu stahuje data a předává je `Modelu`.
- `Model` po uložení notifikuje `ViewModel`, ten vytváří výstup pro `Controller`, který nechává překreslit `View`.
- V případě chyby vytvoří `ViewModel` chybový výstup, ten se pomocí `Controlleru` propaguje do `View`.

Shrnutí

Vrstvy mají následující klíčové vlastnosti:

- `Model` definuje jakým způsobem jsou data uložena a při změně notifikuje `ViewModel`,
- `View` vykresluje na obrazovku naformátované výstupy a upozorňuje `Controller` při interakci uživatele,
- `Controller` sestavuje hierarchii `View`, napojuje zformátované výstupy `ViewModelu` na `View` a z uživatelské interakce vytváří vstupy pro `ViewModel`,

- `ViewModel` načítá data `Modelu`. Na základě vstupů z `Controlleru` nebo změny `Modelu` generuje výstupy pro `Controller`.

Oproti MVC je na tomto příkladu vidět snížení zodpovědnosti `Controlleru`. Tato zodpovědnost se přesunula do `ViewModelu` (viz ukázka 2.5). Na první pohled nemusí být tato změna opodstatněná, protože logika aplikace nezmizela, jen se přesunula. Právě to ale umožnilo (nebo minimálně zjednodušilo) způsob, jakým lze logiku testovat. `ViewModel` generuje výstupy na základě vstupů, v testech tedy lze uživatelskou interakci podvrhnout a testovat pouze výstupy (není potřeba vytvářet `View` ani `Controller`). Dodatečně lze otestovat i uživatelské rozhraní. Protože logika aplikace je otestována pomocí testů `ViewModelu`, uživatelské rozhraní už stačí otestovat např. shodou `View` s referenčním obrázkem.

```
1  import UIKit
2
3  class PrinterViewController: UIViewController {
4      var printerView: printerView!
5
6      private let viewModel = PrinterViewModel()
7
8      override func viewDidLoad(animated: Bool) {
9          super.viewDidLoad(animated: animated)
10
11         printerView.nameLabel.text = viewModel.name
12         printerView.urlLabel.text = viewModel.url
13     }
14 }
```

Ukázka kódu 2.5: Controller v architektuře MVVM

Při pohledu na notifikace je vidět, že přibyl typ, který nebyl v MVC potřeba. Jedná se o notifikace směrem z `ViewModelu` ke `Controlleru` (`ViewModel` nemá referenci na `Controller`, nemůže ho notifikovat přímo). Některé výstupy `ViewModelu` je tedy potřeba sledovat v čase a na jejich změny reagovat. Toto lze vyřešit pomocí Key-Value Observing (KVO), které nabízí Apple mezi standardními knihovnami. KVO umožňuje objektu zaregistrovat se na notifikace o změně stavu nějakého libovolného jiného objektu. V případě `Controlleru` by se registroval na změny stavu výstupů `ViewModelu`. Kdykoliv by se výstup změnil, `Controller` by dostal notifikaci. Tento přístup ale není běžný pro použití s jazykem Swift [13]. Tento postup navíc neřeší synchronizaci vláken, z tohoto důvodu by mohlo docházet k nekonzistenci dat či neočekávanému chování. Místo KVO se nyní standardně používají reaktivní rozšíření, které popisuji v následujících kapitolách [13].

Přestože mnou implementovaná aplikace není v ohledu na uživatelské scénáře nijak složitá, obsahuje mnoho obrazovek. Obrazovky jsou vysoce interaktivní a více se k jejich implementaci hodí reaktivní přístup.

2.3 Synchronizace vláken aplikace

Mobilní aplikace se svou povahou značně liší od běžných webových nebo desktopových aplikací. Oproti zmíněným jsou často mnohem interaktivnější, tedy ovládané nejen běžnými vstupy, ale i vlastnostmi zařízení (GPS poloha, orientace zařízení, dostupnost periferií). Aby tyto vstupy nekazily uživatelský zážitek, jsou v systému implementovány asynchronně.

Jako příklad lze vzít získávání polohy z družic. To provádí telefon na vlákně v pozadí. Tím je zaručené, že hlavní vlákno aplikace (které se mimo jiné stará o vykreslování) není blokované a s aplikací lze bez problému intereagovat. Jakmile jsou dostupné informace o poloze, aplikace zažádá systém o prostředky na hlavním vlákně a získanou polohu prezentuje uživateli.

Tento přístup se jeví v Software Development Kit (SDK) poskytovaným Applem velmi obvyklým a využívá ho mnoho standardních knihoven. Ne vždy je ale běžné, aby výsledek byl prezentovaný na hlavním vlákně. Takový princip lze využít u síťových požadavků.

Síťový požadavek se vykonává na vlákně v pozadí a na stejném vlákně je zpracována i odpověď ze vzdáleného serveru. Je-li potřeba odpověď zpracovat na hlavním vlákně, musí vývojář explicitně čas na hlavním vlákně vyžádat pomocí Grand Central Dispatch či vlákna synchronizovat jiným způsobem. Od verze systému iOS 9 je navíc nemožné udělat síťový požadavek synchronně (blokovat vlákno, které požadavek vytvořilo až do konce zpracování) [14].

Otázkou tedy zůstává, jak správně synchronizovat všechny úkoly, které potřebují zpracovat data na jednom společném vlákně.

Při analýze tohoto problému jsem se zaměřil na tři možná řešení. První dvě jsou implementovány přímo v dodávaném SDK, třetí možnost je knihovna od společnosti GitHub.

2.3.1 Grand Central Dispatch

Grand Central Dispatch (zkráceně GCD) je technologie vyvinutá společností Apple přinášející, optimalizovanou podporu pro aplikace fungující na vícejádrových procesorech. GCD je implementována nad standardními systémovými vlákny, vývojáři ale nabízí mnohem jednoduší rozhranní.

Pro zjednodušení je využit princip front (z angl. queue), které jsou reprenzentovány třídou `DispatchQueue`. `DispatchQueue` je implementována jako `thread-safe`, je tedy možné k ní přistupovat ve stejný okamžik z několik vláken najednou [15]. Do této fronty je možné vkládat jednotky práce, GCD se postará o to, aby byly spuštěny ve správném pořadí. V závislosti na konfiguraci pak umožňuje jednotlivé úkoly spouštět synchronně nebo asynchronně.

V základu je dostupná hlavní (`main`) fronta, která je synchronní a umožňuje vykonat práci na hlavním vlákně. Pro synchronizaci vláken stačí požadované jednotky práce přidávat do správných front.

Je-li potřeba vytvořit vlastní frontu (z důvodu uvolnění času na hlavním vlákně), stačí specifikovat název fronty a její prioritu. Vytvoření nové fronty běžící v pozadí je vidět v ukázce 2.6.

```
1 import Foundation
2
3 func createBackgroundQueue() -> DispatchQueue {
4     return DispatchQueue(
5         label: "My Queue",
6         qos: DispatchQoS.background
7     )
8 }
```

Ukázka kódu 2.6: GCD: Vytvoření vlastní fronty

Quality of Service

Pro možnost rozlišení priorit front využívá Apple `Quality of Service` (zkráceně QoS). Díky QoS je možné určit, jakou prioritu bude mít danná fronta při rozdělování vláken. V současné době existují čtyři QoS priority [16]:

User-interactive Práce, se kterou uživatel přímo interaguje. Tato fronta má nejvyšší prioritu, nejčastěji se v ní provádí překreslování uživatelského rozhraní či animace. Jednotlivé úkoly z fronty jsou vykonané ihned.

User-initiated Práce, kterou zadal uživatel a vyžaduje okamžitý výsledek. Nejčastěji se stará o akce, které nastanou po interakci s některým z ovládacích prvků (např. tlačítko).

Utility Práce, která vyžaduje více času pro své dokončení. Typicky se jedná o síťové požadavky nebo načítání dat.

Background Práce s nízkou prioritou, kterou si nevyžádal uživatel. Využívá se zejména pro dávkové mazání souborů, synchronizaci nebo indexování databáze.

Při vytváření front je důležité správně volit prioritu. Budou-li všechny fronty využívat jednu prioritu, může se stát, že systémová vlákna nebudou správně využita. To by vedlo ke snížení výkonu aplikace a špatné uživatelské zkušenosti.

Přestože GCD je velmi flexibilní abstrakce nad standardními vlákny, jedná se stále o nízkoúrovňové API. Mezi jednotkami práce nelze vyvářet závislosti či je řetězit. Tuto možnost ale nabízí `Foundation` framework pomocí `OperationQueue`.

2.3.2 OperationQueue a Operation

`OperationQueue` a `Operation` je abstrakce nad metodou GCD zmíněnou výše. Toto API je od verze systému iOS 4 implementováno právě pomocí GCD, nabízí ale nové možnosti přístupu k prováděným jednotkám práce. Hlavním rozdílem oproti GCD je vyloučení pojmu `vlákno`. Jednotky práce nejsou nadále reprezentovány pomocí `first class` funkcí, ale nově pomocí instancí podtříd `Operation`, o jejichž správu se stará `OperationQueue`. [17]

Operation

Pro reprezentaci jednotek práce se standardní funkce v GCD nejevily jako dostatečné. S představením `OperationQueue`, je ale nově možné nad opakovanými jednotkami práce zakomponovat abstrakci pomocí tříd. Toho lze docílit pomocí vytváření vlastních tříd dědících od `Operation`.

`Operation` představuje samostatnou jednotku práce (operaci). Jedná se o abstraktní třídu zajišťující `thread-safe` přístup ke správě stavu, priority a závislostí.

Jako úkol lze chápat například jednotlivé síťové požadavdy, zpracování vstupů a uživatele a jejich perzistence nebo komplexní výpočty. Úkolem je možné označit ale i libovolnou strukturovanou jednotku práce, u které je potřeba udržovat stav nebo zpracovat její datový výstup.

Oproti GCD má objektový přístup výhodu nejen v přehlednější strukturování, ale i v možnosti vytváření závislostí a správě stavu [18].

Závislosti Definují, jaké další operace musí být vykonány před tím, než bude daná operace spuštěna. Přidat lze libovolný počet dalších operací, tím lze dosáhnout komplexního procesu za pomoci skládání menších bloků. Jako příklad lze uvést síťovou komunikaci, kde každý požadavek závisí na ověření dostupnosti internetu (první operace) a na patřičném ověření uživatele (druhá operace). Ke každému požadavku následně stačí přiřadit tyto dvě operace jako závislost, tím je zaručené že požadavek se odešle, jen když je dostupné připojení k internetu a zároveň je uživatel ověřený.

Správa stavu Umožňuje operaci pozastavit, spustit znovu nebo zrušit. Oproti GCD lze například do ovládání operací zapojit uživatele. V GCD naopak úkol, který se začal zpracovávat, už nebylo možné zastavit.

OperationQueue

Obdobně jako u GCD, je i zde potřeba určit, kde a kdy se bude práce vykonávat. Pro tento účel slouží fronta `OperationQueue`. Tato fronta je implementována jako prioritní. Tedy v momentě, kdy je vložena operace s vyšší prioritou, bude vykonána dříve než stávající operace s nízkou prioritou (pokud to její závislosti dovolí). Z tohoto důvodu není zaručeno, kdy se jednotlivé operace spustí.

Základní vytvoření operace a vložení do fronty je vidět na ukázce 2.7.

```swift
import Foundation

/// Vytvoření společné fronty pro zpracování operací
let queue = OperationQueue()

/// Operace, která má být spuštěna jako druhá
let secondOperation: Operation = BlockOperation {
    print("second.")
}

/// Operace, která má být spuštěna jako první
let firstOperation: Operation = BlockOperation {
    print("First come before", separator: " ", terminator: " ")
}

/// Vyžádá vykonání operace firstOperation před svým spuštěním
secondOperation.addDependency(firstOperation)

/// Vložení operací do fronty
queue.addOperation(secondOperation)
queue.addOperation(firstOperation)

/// Vytiskne "First comes before second.",
/// tedy nezáleží na pořadí vložení do fronty
```

Ukázka kódu 2.7: OperationQueue: Vytvoření fronty s operacemi

Prioritizace úkolů

Z důvodu potřeby prioritizace některých úloh je možné jednotlivým operacím nastavit prioritu. V takovém případě se vždy jako první vykonají operace s vyšší prioritou, následně se přistupuje k těm s nižší.

Quality of Service

Jednou z nejsilněších stránek `OperationQueue` je abstrakce front nad vlákny (tedy celého GCD). V nastavení fronty lze zvolit stejně jako u GCD výchozí QoS. Všechny operace pak budou využívat právě tuto prioritu. Oproti GCD má však `OperationQueue` výhodu, že nepracuje pouze s jednou frontou.

To dává vývojáři možnost definovat QoS pro každou operaci zvlášť, a to bez nutnosti vytvářet více front. V závislosti na této vlastnosti je dále možné definovat, kolik operací může být *maximálně* spuštěno v jeden okamžik.

2.3.3 Reaktivní programování

Reaktivní programování získává poslední dobou velký zájem vývojářů [19]. Jedná se o programování založené na reakci na uživatelské vstupy. Nejčastěji je tento princip spojován s programováním uživatelského rozhraní, protože je implementováno asynchronně a neblokuje tedy hlavní vlákno aplikace.

Myšlenka reaktivního programování je nepřistupovat k datům jako stálým hodnotám, ale jako k proudu (streamu) hodnot v konkrétní čas. Implementací existuje více, např. Promise [20], Callback [21] či funkcionálně reaktivní programování. V analýze jsem zabýval pouze funkcionálně reaktivním programováním, konkrétně imlementací v knihovně `ReactiveCocoa`.

Funkcionálně reaktivní programování – FRP

Funkcionálně reaktivní programování (zkráceně FRP) označuje stream hodnot jako signál. Z teorie funkcionálního programování se k signálům přistupuje jako ke konstantám. Jednou vytvořený signál nelze měnit ani ho ovlivnit vedlejšími účinky. Je-li potřeba signál změnit, je nutné vytvořit nový signál a ten používat místo původního.

Signál může reprezentovat libovolnou událost v aplikaci: stisknutí tlačítka, síťový požadavek, změny hardware. Hodnota je v signálu dostupná pouze ve chvíli, kdy je signálem poslána a doručena objektům, které se zaregistrovali o její příjem. Těmto objektům se v terminologi `ReactiveCocoa` říká `Observer`. Signály tedy nepodporují přímý přístup k hodnotám.

Přestože se absence přímého přístupu může zdát jako velké negativum, v praxi to není žádný problém. Místo standardního přístupu k hodnotám se využívá kombinování, řetězení a přetváření signálů. V aplikaci pak není nutné k datům přistupovat na vyžádání. Jakmile jsou totiž nová data dostupná, zpropagují se na patřičná místa podle předem namodelovaného procesu. Vytvoření signálu je vidět v ukázce 2.8.

Hot a Cold signály

Z ukázky 2.8 je vidět, že hodoty prochází signálem okamžitě. Zde nastává problém ve chvíli, kdy se `Observer` o přijímání hodnot zaregistruje až po té,

```swift
// Vytvoření signálu a Observeru pro vkládání hodnot
let (signal, sink) = Signal<Int, NoError>.pipe()

// Vytvoření nového signálu posílajícího změněné hodnoty,
// přídání Observeru pro přijímání hodnot
signal.map({ $0 * 10 }).observeValues { number in
    print(number)
}

// Vložení hodnot do signálu
sink.send(value: 1) // Vytiskne 10
sink.send(value: 2) // Vytiskne 20
```

Ukázka kódu 2.8: Vytvoření signálu v ReactiveCocoa

co nějaké hodnoty už signálem proběhly. V `ReactiveCocoa` jsou proto dva typy signálů.

`Hot` signál reprezentovaný třídou `Signal` odesílá hodnoty okamžitě a neudržuje jejich historii. Pokud se `Observer` zaregistruje pozdě, hodnotu nikdy neobdrží. U síťového požadavku by se mohlo stát, že přijde odpověď od serveru, ale nebude zpracována. Tento typ se tedy využívá v částech programu, kde absence některých hodnot nezpůsobí nekonzistenci.

Druhým typem signálu je `Cold` signál reprezentovaný třídou `SignalProducer`. Tato třída odesílá hodnoty až ve chvíli, kdy se o ně nějaký `Observer` přihlásí. Tomuto `Observeru` navíc budou odeslány postupně všechny hodnoty, které byly do tohoto signálu vloženy dříve, jak je vidět v ukázce 2.9. To je ideální přístup pro manipulaci s daty či síťové požadavky.

```swift
// Vytvoření Cold signálu
let producer = SignalProducer<Int, NoError>() { sink, _ in
    sink.send(value: 1)
    sink.send(value: 2)
}

// Tento Observer dostane hodnoty 1 i 2
producer.startWithValues { /* ... */}

// Tento Observer dostane také hodnoty 1 i 2,
// přestože už signálem prošly dříve
producer.startWithValues { /* ... */}
```

Ukázka kódu 2.9: Vytvoření Cold signálu

Operátory nad signály

Protože signály jsou konstantní a jejich hodnotu nelze editovat, nabízí `ReactiveCocoa` operátory nad signály. Pomocí těchto operátorů je možné vytvořit nový signál stejného typu, jehož hodnota vychází z původní hodnoty libovolnou úpravou.

Tyto operátory přejímají názvy z teorie funkcionálního programování. Dostupné jsou tedy operátory `map`, `filter`, `reduce` a další.

Pomocí rozšíření tříd dostupných v jazyku Swift je také možné programovat vlastní operátory.

Synchronizace vláken

Pomocí signálů lze jednoduše zařídit synchronizaci vláken. K tomu jsou dostupné knihovní funkce `flatMap`, `combineLatest`, `merge` a další. Signály mohou vzniknout na různých vláknech, knihovna se ale stará o to, aby k signálům bylo přistupováno vždy právě z jednoho vlákna. Každý `Observer` si také může zvolit, na jakém vlákně mu budou hodnoty předávány.

Více informací ke knihovně `ReactiveCocoa` a jejímu využití v reaktivním programování je dostupné na její GitHub stránce [22].

2.4 Síťování

Nedílnou součástí aplikace je také síťování, tedy komunikace s rozhraním OctoPrint pomocí REST API. Síťování je v mém řešení stěžejní, protože aplikace komunikuje s tiskárnou vždy v reálném čase. Při analyzování síťové vrstvy jsem dospěl ke třem základním požadavkům:

1. jednotlivé enpointy musí být dobře strukturované,
2. implementace musí být generická, aby bylo možné dosáhnout maximální znovupoužitelnosti,
3. Jednotlivé požadavky musí jít snadno řetězit bez ztráty informací či ignorování chyb.

Mým požadavkům se nejvíce přiblížila tři možná řešení, která následně podrobně popíši. Řešení jsem analyzoval od těch nejvíce nízkoúrovňových, které pracují pouze se standardními knihovnami, až po implementace za využití externích knihoven.

2.4.1 URLSession

Jako první možnost pro analýzu jsem vybral základní knihovnu `URLSession`, která je dostupná ve frameworku `Foundation`. V tuto chvíli se jedná o API, které je nejvíce nízkoúrovňové, i tak ale nabízí vysokou flexibilitu.

Spolu s `URLSession` se pojí i mnoho dalších tříd, které jsou vzájmně závislé.

URLSessionConfiguration

Při vytváření nové instance `URLSession` je potřeba poskytnout konfiguraci. Tato konfigurace popisuje základní nastavení pro politiky cachování požadavků, využívané protokoly, nastavení cookies a správu přihlašovacích údajů.

Pro jednoduché aplikace je možné využít předpřipravenou konfiguraci, ta ale přistupuje ke všem požadavkům stejně, což je nevhodné pro využití při komunikaci s více vzdálenými body. Pro svou implementaci bych tedy musel využít dynamické konfigurování v závislosti na aktuálně využívané tiskárně. V ukázce 2.10 je vidět možná implementace dynamické konfigurace.

```
1 import Foundation
2
3 let configuration = URLSessionConfiguration()
4
5 /// Přidání API klíče pro konkrétní tiskárnu
6 configuration.httpAdditionalHeaders = ["X-Api-Key": "123-ABC"]
7 configuration.httpCookieAcceptPolicy = .always
8
9 let printer = URLSession(configuration: configuration)
```

Ukázka kódu 2.10: Dynamické nastavení URLSession

URLSessionTask

Pro samotnou reprezentaci požadavků se využívají podtřídy abstraktní třídy `URLSessionTask`. Těmito podtřídami jsou `URLSessionDataTask`, `URLSessionUploadTask` a `URLSessionDownloadTask`. Ty zprostředkovávají získávání dat (JSON, XML, ...) a stahování a nahrávání souborů. Pro vytvoření těchto tříd nabízí `URLSession` předpřipravené metody.

Samotné vytvoření požadavku pro stažení JSON objektu je vidět v ukázce 2.11.

```
1 let endpoint = URL(string: "http://10.0.2.27:3200/api/version")
2
3 printer.dataTask(with: url) { (data, response, error) in
4     /* zpracování odpovědi */
5 }.resume()
```

Ukázka kódu 2.11: Vytvoření požadavku pomocí URLSession

Shrnutí

Přestože `URLSession` nabízí velmi komplexní řešení síťových požadavků, byla by implementace velmi náročná. V současné době nabízí na řetězení požadavků pouze `completion blok`, který ale není pro řetězení mnoha požadavků vhodný (časté opakování kódu, zanořování metod).

Strukturování koncových bodů v základu také není řešené, bylo by tedy potřeba vytvořit novou abstraktní vrstvu. Z tohoto důvodu jsem se rozhodl přistoupit ke zanalyzování knihoven třetích stran.

2.4.2 Alamofire

Jako první knihovnu třetích stran jsem se rozhodl zanalyzovat populární /8`Alamofire` [23]. Jedná se o síťovací knihovnu, naprogramovanou s využitím `URLSession`. `Alamofire` nabízí oproti `URLSession` moderní API využívající sílu jazyka Swift, formátování požadavků (JSON, URL, ...) a také validaci odpovědí serveru.

`Alamofire` navíc nabízí velmi jednoduchý a intuitivní způsob strukturování koncových bodů pomocí `Routeru`. Další velkou výhodou je vysoká znovupoužitelnost. Pomocí vlastních pravidel pro serializování odpovědí serveru je možné vytvořit generické požadavky, jejichž odpovědi nebudou prostý text ale už platný Swift objekt. Takto serializované objekty lze následně uložit do databáze či používat bez zanesení chyby za běhu aplikace (díky typové kontrole během kompilace).

Pro samotné požadavky využívají globální metody, které sdílí společné nastavení sezení (z anglického *session*).

Router

Pro rozsáhlejší projekty poskytuje `Alamofire` velmi elegantní způsob strukturování koncových bodů. V terminilogii `Alamofire` se objektu, který je struktuje říká `Router`. Zpravidla se jedná o výčtový typ (`enum`). Koncové body jsou v něm reprezentovány pomocí výčtů.

Tyto výčty lze pak předat do `Alamofire` a tím vytvořit požadavek na server. Ukázková implementace `Router` objektu je vidět v ukázce 2.12.

SessionManager

Přestože standardně používané metody jsou běžně naprosto dostačující, pro mé řešení jsou nevhodné z důvodu sdílení společného nastavení i pro různé vzdálené body (tiskárny).

Z tohoto důvodu je v `Alamofire` dostupná třída `SessionManager`, jejíž instance mohou využívat různé nastavení. `SessionManager` dále nabízí obdobné API jako globální funkce pro vytváření požadavků.

```swift
import Alamofire

enum Router: URLRequestConvertible {
    case .authorize(apiKey: String)
    case .printFile(URL)
    case .movePrintHead(direction: Direction)

    func asURLRequest() throws -> URLRequest { /* .. */ }
}
```

Ukázka kódu 2.12: Ukázková implementace `Router` objektu

Příklad vytvoření požadavku s nakonfigurovaným `SessionManager` objektem je uveden v ukázce 2.13.

```swift
import Alamofire

let fileURL = ...
let configuration = ...
let manager = SessionManager(configuration: configuration)

manager.request(Router.printFile(fileURL))
    .validate().response { response in
        /* zpracování odpovědi*/
    }
```

Ukázka kódu 2.13: Vytvoření požadavku pomocí Alamofire

Shrnutí

`Alamofire` je velmi rozsáhlou knihovnou pro síťové požadavky. Již v základu obsahuje abstraktní vrstvu nad `URLSession`, kterou bych ve své implementaci musel také napsat. Pomocí `Router` objektu lze velmi přehledně a čitelně zapouzdřit jednotlivé koncové body. Vlastní serializace odpovědí serveru pak umožňuje vysokou znovupoužitelnost bez opakování kódu.

Pro zpracování je nicméně opět dostupný pouze `completion blok`. Řetězení požadavků zůstává prakticky stejné jako u `URLSession`.

2.4.3 Moya

Jako poslední možné řešení síťových požadavků jsem zvážil open-source knihovnu `Moya` [24].

`Moya` je abstraktní vrstva nad výše zmíněným `Alamofire`. Nabízí tedy stejné možnosti, klade si ale za cíl uživatele odstínit od správy URL adres, parametrů požadavku a dalšího nastavení.

Druhou výhodou je zapouzdřování koncových bodů do tkzv. `Provider` objektů. Na rozdíl od `Router` objektu zapouzdřuje `Provider` pouze ty se stejnými vlastnostmi (např. stejná úroveň řízení přístupu). Během kompilace tak lze ověřit, že požadavky, které vyžadují oprávnění vytváří stejný `Provider`.

Velmi užitečný je i zcela jiný přístup k vytváření požadavků. `Moya` vytváří požadavek ve třech fázích. Nejprve nechá uživatele vytvořit `Target`, ten se následně mapuje na `Endpoint`, z kterého teprve `Provider` vytvoří samotný požadavek.

Target

Prvním krokem vytváření požadavku je `Target`. `Target` reprezentuje akci, která se má vykonat na vybraném vzdáleném API. Typicky je implementovaný jako výčet a poskytuje vysokoúrovňové API pro využití ve zbytku aplikace. Implementace `Target` objektu je vidět v ukázce 2.14.

```
1  import Moya
2
3  enum Printer: TargetType {
4      case printFile(URL)
5      case disconnectPrinter
6
7      var baseURL = /* ... */
8      var path = /* ... */
9      var method = /* ... */
10     var parameters = /* ... */
11     var parametersEncoding = /* ... */
12     var sampleData = /* ... */
13     var task = /* ... */
14 }
```

Ukázka kódu 2.14: Moya – Implementace Target objektu

Endpoint

`Endpoint` je interní struktura reprezentující koncový bod, která je využita k sestavení požadavku. Hlavním cílem této struktury je možnost modifikovat způsob, jakým budou požadavky vytvořeny. Pomocí `Endpoint` objektu lze např. přidávat autorizační token či některé požadavky rušit, zpozdit atp.

Provider

Při použití `Moya` se veškeré požadavky odesílají skrz `Provider`. `Provider`ovider je generický objekt, který operuje na objektem `Target`. Tím se v době kompilace dosáhne kontroly kompatibility `Provider` s `Target` objektem.

`Provider` nabízí rozhraní pro modifikaci vytvoření `Endpointu` i pro modifikaci požadavku. Vytvoření požadavku je vidět v ukázce 2.15.

```
import Moya

let provider = ReactiveSwiftMoyaProvider<Printer>()

provider.request(.disconnectPrinter)
    .filterSuccessfullStatusCodes()
    .startWithResult { result in
        /* zpracování odpovědi */
    }
```

Ukázka kódu 2.15: Moya – vytvoření požadavku

Reaktivní rozšíření

`Moya` v základu nabízí rozšíření pro použití s reaktivními knihovnami. Díky tomu lze velmi dobře integrovat s architekturou MVVM, konkrétně do vrstvy `ViewModel`. Usnadňuje tím také možnost řetězení požadavků pomocí `flatMap` operátoru, jak je vidět v ukázce 2.16.

```
import Moya

provider.request(.authorize)
    .flatMap(.latest) {
        return provider.request(.movePrintHead(direction: .x))
    }
    .flatMap(.latest) {
        return provider.request(.extrude)
    }
    .startWithResult { /* ... */}
```

Ukázka kódu 2.16: Moya – Řetězení požadavků

Testování

Jednou z největších výhod `Moya` je možnost testování. V `Target` objektu je možné definovat, jaká data má požadavek vrátit v době, kdy se aplikace testuje. Tím `Moya` zaručí, že testy nebudou selhávat v případě, kdy nebude dostupná síť.

Pro případ testování chování aplikace v momentě, kdy je síť nedostupná, nebo kdy požadavky selžou, je možné opět využít úpravy `Endpoint` objektu (tedy i takovou situaci lze nasimulovat).

Shrnutí

`Moya` je knihovna pro vytvoření abstrakce nad síťovou vrstvou. Tato nová vrstva se tak stává jediným spojením aplikace s okolní sítí.

Obsahuje totožné funkce jako knihovna `Alamofire`, kterou ve své implementaci využívá. Oproti `Alamofire` navíc nabízí striktní oddělení `Endpoint` a `Target` objektů.

Tím umožňuje požadavky automaticky mockovat při testování.

Díky reaktivním rozšířením lze navíc velmi dobře integrovat s architekturou MVVM.

2.5 Datová vrstva

Pro uložení nastavení tiskáren je důležité, aby aplikace mohla ukládat data na disk. Většina ostatních dat není potřeba ukládat, protože je velmi pravděpodobné, že při dalším spuštění aplikace už by data nebyla aktuální. I přesto jsem se rozhodl veškerá data, která aplikace získá z tiskárny uložit lokálně. Lokální uložiště následně bude sloužit jako *source of truth*, tedy kdykoliv vznikne v aplikaci nekonzistence, data se obnoví z lokálního uložiště.

Pro lokální uložiště jsem vybral dvě technologie, které porovnám a následně jednu implementuji. První technologií jsou `Core Data`, technologie vyvinutá společností Apple běžně používaná na platformách iOS a macOS. Druhou technologií je Realm Mobile Database, open-source knihovna třetí strany.

2.5.1 Core Data

`Core Data` je framework představený společností Apple v roce 2005 [25]. Jedná se o technologii starající se o modelovou vrstvu aplikace. Kromě reprezentování stavu aplikace umožňují `Core Data` také data perzistovat na disk. Nejedná se ale pouze o Object Relational Mapping (ORM) nadstavbu. `Core Data` spravuje data i ve chvíli, kdy jsou v paměti.

Během analyzování tohoto frameworku jsem dospěl k závěru, že implementace by byla velmi složitá. Z tohoto důvodu jsem neprováděl hlubší analýzu.

2.5.2 Realm Mobile Database

Jako druhou možnost jsem zvolil technologii `Realm`. Přestože verze 1.0 této knihovny byla vydána teprve v květnu 2016, má k dnešnímu dni má dohromady více než dvacet tisíc hvězdiček na službě Github [26].

`Realm` je modelová technologie vytvořená pro využití v mobilních zařízeních. Oproti běžným databázím `Realm` nevyužívá ORM a nekopíruje data z databáze. V `Realm` aplikace pracuje s reálnými daty uloženými na disku, z tohoto důvodu jakmile jsou data jednou aktualizována, změna se automaticky propisuje všude, kde jsou data využívána (reaktivní přístup) [27].

Modelové objekty

`Realm` je založen na modelových objektech. Ty se definují v jazyku, ve kterém je aplikace programována (v mém případě Swift). Databázový soubor se následně na disku vytvoří z těchto modelových objektů. Přístup je tedy opačný než u ORM, kde se v aplikaci vytváří objekty podle toho, jak jsou uložené v databázi. Jedinou podmínkou při vytváření objektu je nutnost dědit od třídy `Object` a využít správné datové typy (ne všechny je možné serializovat na disk). V ukázce 2.17 je třída `Printer`, kterou lze použít pro reprezentaci objektu tiskárny.

```
1 import RealmSwift
2
3 class Printer: Object {
4     dynamic var url = ""
5
6     dynamic var accessToken = ""
7
8     dynamic var name = ""
9 }
```

Ukázka kódu 2.17: Modelový objekt v technologii Realm

Relace

`Realm` implementačně sice není relační databází, nabízí ale podobné API pro relace mezi objekty. Objekty mohou být v relacích `M:N`, `1:M` nebo `1:1`. Na základě těchto vlastností je následně možné vytvářet databázové dotazy.

Transakce

Z důvodu zajištění konzistence dat v aplikaci je nutné modelové objekty upravovat v transakcích [28]. `Realm` neodděluje standardní CRUD operace, místo toho je zavedena `write` transakce, v níž je možné CRUD operace využít. Transakce implementuji v ukázce 2.18.

Notifikace

Velkou výhodou této technologie jsou notifikace. `Realm` umožňuje notifikovat o změnách objektů v databázi. Na základě těchto notifikací pak lze např. překreslit obrazovku či změnit stav aplikace.

Notifikace lze zaregistrovat i na změny kolekcí. Zobrazuje-li např. aplikace seznam tiskáren, lze se zaregistrovat na jejich změny. Je-li pak do kolekce přidána položka nebo je stávající položka změněna, lze nechat překreslit seznam.

Tomuto principu se říká *Reaktivní programování* a lze ho velmi dobře integrovat do architektury MVVM.

```swift
import RealmSwift

let realm = try! Realm()

try! realm.write {
    let printer = Printer()

    realm.add(printer)
}
```

Ukázka kódu 2.18: Transakce v technologii Realm

2.6 Grafické prvky

Pro zvýšení použitelnosti a přehlednosti jsem se rozhodl do aplikace zakomponovat grafické prvky (ikony tlačítek, zobrazovaná varování, ...). Protože cílím aplikaci nejen na iPhone, ale i na iPad, musí veškerá zakomponovaná grafika být ve vysokém rozlišení. Jednotlivé ikony a obrázky lze do Xcode nahrát pomocí `Assets catalog`, tedy galerie obrázků. Jako alternativní možnost jsem se rozhodl zanalyzovat knihovnu `Iconic`, která umožňuje využít grafických fontů.

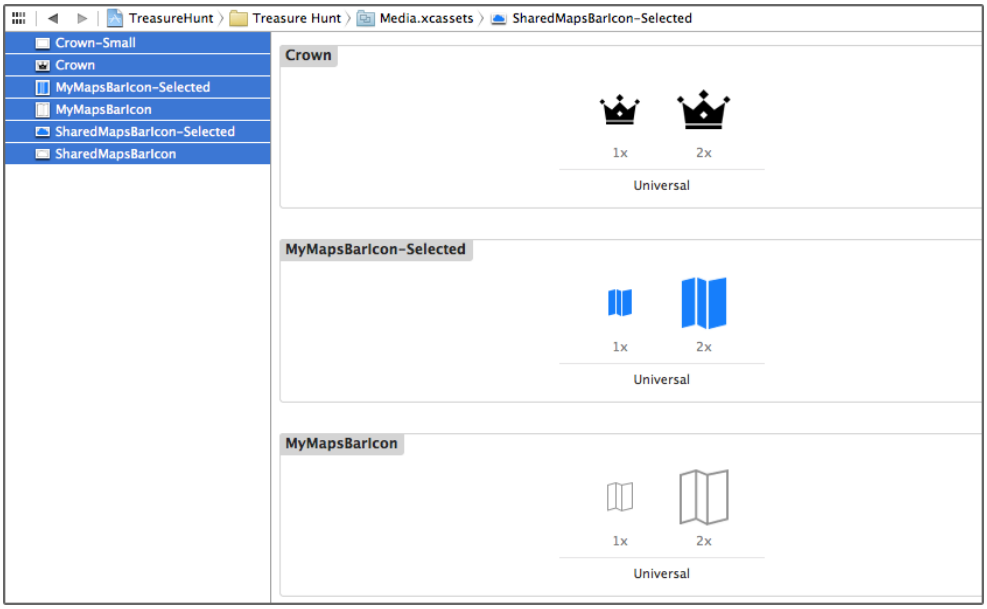
2.6.1 Assets catalog

Assets catalog je galerie obrázků přímo integrovaná do vývojového prostředí Xcode a grafického frameworku `UIKit` [29]. Obrázky se do galerie přidávají v různých velikostech (pro různou hustotu pixelů na displayi), typicky v rozlišení 1:1, 2:1 a 3:1. Správa různých velikostí je vidět na obrázku 2.4. V aplikaci se následně přistupuje k obrázku podle jeho jména, systém sám rozhodne jakou veliost využít aby na displayi vypadala nejlépe.

S roustoucím počtem obrázků se ale dramaticky snižuje přehlednost galerie a naopak roste velikost aplikace.

2.6.2 Iconic

Alternativní metodou k Assets catalog jsou grafické fonty. Ty mají výhodu v tom, že jsou obsažené v jednom souboru a jsou vektorové. Pokud tedy přijde zařízení s ještě větší hustotou pixelů než existují dnes, nebude potřeba obrázky znovu exportovat, vše bude fungovat automaticky. V současné době jediným frameworkem převádějícím fontové znaky na obrázky je `Iconic`. `Iconic` rozparsuje zadaný font a vytvoří pro něj výčtový typ v jazyku Swift. Pomocí výčtového typu lze pak k obrázkům přistupovat pomocí jejich jména, kde navíc platnost jména ověřuje kompilátor. Protože se fonty typicky generují na začátku tvoření aplikace, pouští se `Iconic` typicky jen jednou a jeho výstup včetně fontu se následně přidává do projektu.



Obrázek 2.4: Assets catalog v Xcode IDE [1]

Pro obdobné použití jako s Assets catalog nabízí `Iconic` rozšíření frameworku `UIKit`. Tyto rozšíření umožňují využívat fonty nejen ve zdrojovém kódu, ale i v `Interface builder` rozhraní. Využití v kódu je naznačené v ukázce 2.19.

```
let icon = FontAwesomeIcon.homeIcon.image(
    ofSize: CGSize(width: 44, height: 44),
    color: .black
)
```

Ukázka kódu 2.19: Využití grafického fontu s frameworkem Iconic

Oproti Assets catalog ale postrádá možnost cachování [30]. Bude-li tedy na obrazovce velké množství stejných obrázků nebo budou-li se často překreslovat, mohl by vzniknout problém s výkonem aplikace.

2.7 Správa závislostí

Pro zakomponování knihoven třetích stran jsem se rozhodl využít správce závislostí. Pomocí správce závislostí je možné jednoduše udržovat knihovny aktuální či s projektem svázat konkrétní verze knihoven. Při analyzování nástrojů na správu závislostí jsem se zabýval dvěma nejpoužívanějšími [31]. Prvním nástrojem jsou `CocoaPods` [32], open-source nástroj řízený komunitou. Druhým

nástrojem je `Carthage` [33], vytvořený společností GitHub a nyní vedený jako open source.

2.7.1 CocoaPods

`CocoaPods` je nástroj s rozhraním pro příkazovou řádku, který umožňuje spravovat závislosti pro Xcode projekty. Základem jsou takzvané `Pods`, tedy balíčky závislostí. Tyto balíčky může uživatel integrovat do svého projektu pomocí souboru `Podfile`, do kterého vypíše seznam seznam závislostí. `CocoaPods` následně závislosti stáhne a vytvoří pracovní `workspace`, do kterého závislosti nalinkuje. Pro následné programování pak uživatel využívá nový `workspace` místo původního souboru projektu.

Nevýhoda tohoto přístupu je, že jsou staženy přímo zdrojové kódy závislostí a ty jsou připojeny k projektu. Z tohoto důvodu se při překladu kódu může stát, že se kompilují soubory nejen aplikace, ale i všech závislostí. Tím se zvyšuje čas potřebný ke zkompilování aplikace. To má za následek snížení rychlosti vývoje.

2.7.2 Carthage

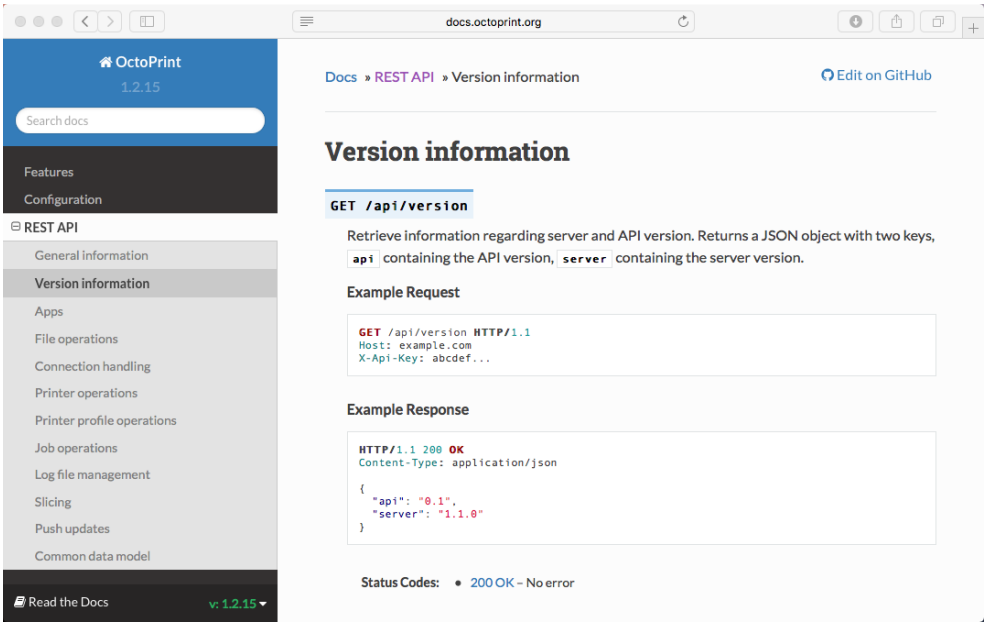
Druhým analyzovaným nástrojem je `Carthage`. Obdobně jako `CocoaPods` nabízí rozhraní příkazové řádky. Hlavním rozdílem je způsob integrace. `Carthage` žádným způsobem není integrován do projektu jako takového. Jedinou funkcí tohoto správce závislostí je stáhnout a zkompilovat zdrojové kódy požadovaných závislostí. Uživatel následně musí sám knihovny do projektu naimportovat.

Pro urychlení kompilace mohou správci knihoven předkompilovat knihovny do binární formy, ta se následně pouze stáhne. Předkompilované knihovny ale musí být napsané ve stejné verzi Swiftu jako používá aplikace sama, v opačném případě nepůjde knihovny slinkovat, ani zkompilovat aplikaci. Takový případ není v lokálním vývoji velkou komplikací, může ale narušit funkcionalitu integračního procesu na vzdáleném serveru. I to je ale situace, která se vyskytne pouze v době vydání nové verze nástroje Xcode. To bývá zpravidla jen několikrát ročně.

2.8 OctoPrint API

V implementační části práce se zabývám vytvořením mobilní aplikace pro OctoPrint. Pro pro ovládání tiskárny využiji REST API, které OctoPrint v základu nabízí. V dalších sekcích se budu podrobně zabývat analýzou jednotlivých funkcionalit implementovaných v mém řešení.

Z důvodu bezpečnosti musí být každý požadavek na tiskárnu autorizovaný. To OctoPrint zajišťuje pomocí ověřovacích tokenů. Token je náhodně vygenerovaná posloupnost znaků jednoznačně spjatá s účtem uživatele. Pokud



Obrázek 2.5: Oficiální dokumentace OctoPrint

v požadavku je uveden platný token, předpokládá se, že požadavek vytvořil uživatel, jehož účet je s token svázaný.

V analýze jsem vycházel z oficiální dokumentace aplikace OctoPrint [34]. Ukázka popisu funkcionality z dokumentace je vidět v obrázku 2.5.

2.8.1 Verze API a ověřování

První analyzovanou funkcionalitou je verze API. Dostupná je na `api/version`. V případě platného přístupového tokenu vrátí textovou reprezentaci verze API, v opačném případě odpovídající stavový kód.

Ve své implementaci jsem se rozhodl uživatele nezatěžovat využívanými verzemi. Jedná se ale o jedinou funkci, která nemá žádné vedlejší efekty. Zároveň neexistuje žádný jiný koncový bod umožňující samostatně ověřit platnost tokenu.

Vzhledem k výše uvedenému je tato funkcionalita vhodná pro simulaci přihlašování k tiskárně.

2.8.2 Správa souborového systému

Jednou ze skupin funkcionalit REST API je správa souborového systému. Pomocí API lze nejen obsah systému číst, ale také s ním manipulovat. Jednotlivé funkce jsou zanalyzovány níže.

Seznam souborů

OctoPrint umožňuje vypsat seznam uložených souborů. Funkcionalita je dostupná na `api/files`.

V současnou chvíli mohou na soborový systém být uloženy pouze tisknutelné soubory (s příponou .gcode) nebo modelové soubory (s příponou .stl).

Uloženy mohou být přímo na zařízení kde je OctoPrint nainstalovaný (označované jako lokální) nebo na paměťovou kartu. Výše zmíněná funkce stáhne seznam všech souborů bez ohledu na jejich lokace. Příznak lokace je v odpovědi uveden u každého souboru zvlášť.

Lze ale stáhnout soubory z obou lokalit samostatně. Pro stažení seznamu lokálních souborů existuje koncový bod `api/files/local`. Seznam ouborů uložených na paměťové kartě je možné stáhnout z `api/files/sdcard`.

O souborech jsou dostupná jejich metadata (název, velikost, datum změny). GCode může navíc obsahovat analýzu tisku (předpokládaná doba tisku, spotřeba filamentu a statistiky tisknutí).

Struktura odpovědí je pro všechny tři požadavky stejná, liší se pouze obsah seznamu v závislosti na vybrané lokalitě.

Nahrávání souborů

Pomocí API je možné nejen číst informace o existujících souborech, ale také nahrávat nové. Pro nahrávání slouží koncové body `api/files/local` a `api/files/sdcard`. Podle zvoleného koncového bodu se soubor uloží buď přímo na zařízení nebo na paměťovou kartu.

Soubor se odesílá v těle požadavku kde je zároveň uveden jeho název. V případě vzniku konfliktu názvů bude stávající soubor nahrazen novým.

Systém iOS neobsahuje sdílený souborový systém, soubory lze nahrávat buď pouze z aplikace nebo lze využít cloudových služeb [35].

Informace o konkrétním souboru

Aby bylo možné aktualizovat informace o jednotlivých souborech, umožňuje OctoPrint využít koncový bod `api/files/[location]/[filename]`. Takový požadavek zajistí stažení aktuálních informací o souboru v lokaci `[location]` s názvem `[filename]`.

Strukturou se odpověď nelyší od položky souboru v seznamu souborů. Požadavek lze ale využít v momentě, kdy se z klientské aplikace se souborem v tiskárně manipuluje pro udržení konzistence.

Manipulace se soubory – Tisk

OctoPrint umožňuje pro manipulaci se soubory více konfigurací. Pro využití v mobilní aplikaci se ale hodí pouze konfigurace pro tisk. Ta je dostupná na `api/files/[location]/[filename]`.

V těle požadavku je nutné uvést typ příkazu a explicitně vyžádat tisknutí. Více o konfiguracích se lze dočíst v dokumentaci [36].

Smazání souboru

Další funkcionalitou, je odstranění souboru. Lze k ní přistupovat skrz `api/files/[location]/[filename]`. Požadavek je nutné odeslat metodou `DELETE`.

2.8.3 Připojení tiskárny

Protože OctoPrint běží na samostatném zařízení a hardware tiskárny jen ovládá, je nutné ho k tiskárně explicitně připojit. Připojení probíhá pomocí výběru dostupného portu. Seznam dostupných portů poskytuje sám OctoPrint.

Nastavení připojení

Pomocí koncového bodu `api/connection` lze zjistit akuální nastavení připojení k tiskárně. Přestože tato informace není stěžejní pro mobilní aplikaci, odpověď serveru na tento požadavek obsahuje seznam portů.

Seznam portů lze využít pro výzvu uživatele k připojení tiskárny. To může nastat v momentě kdy uživatel chce s tiskárnou pracovat v době kdy žádná tiskárna připojena není.

Příkaz připojení tiskárny

Podobně jako v sekci o manipulaci souborů, i zde je mnoho konfigurací. Pro mobilní aplikaci ale nedávají z důvodu komplikovaného nastavení velký smysl.

Zajímavou konfigurací je ale možnost připojit tiskárnu. Tento koncový bod se nalézá na `api/connection` a jako metodu je nutné využít `POST`.

V těle požadavku musí být explicitně zadaný příkaz `connect` a také port pro připojení. Hodnota portu musí být jedna z uvedených v seznamu portů. Seznam portů je možné získat z nastavení připojení.

2.8.4 Ovládání tiskárny

Ovládání tiskárny je docíleno pomocí příkazů jednotlivým částem tiskárny. Ve verzi API 1.2.15 jsou rozpoznávány následující příkazy následujícím částem:

Tisková hlava Umožňuje pohybovat tiskovou hlavou tiskárny po všech třech osách.

Tryska Umožňuje nastavení teplot trysky.

Podložka Umožňuje nastavení teplot podložky.

Paměťová karta Dovoluje manipulovat s připojením paměťové karty v momentě, kdy je dostupná.

Aktuální stav tiskárny

Prvním analyzovaným koncovým bodem této skupiny je aktuální stav tiskárny. Ten je dostupný na `api/printer`. Odpověď obsahuje informace o teplotách podložky a trysky. Dále lze v odpovědi nalézt informace o celkové připravenosti tiskárny.

Vhodný může být pro obrazovku s přehledem stavu tiskárny.

Ovládání trysky

Ovládání trysky je nastavitelný koncový bod přístupný na `api/printer/tool`. Každý požadavek na tento koncový bod musí být odeslaný metodou `POST`.

Z důvodu zjednodušení uživatelského rozhraní jsem se rozhodl analyzovat nastavení pro tavení filamentu. Toto nastavení vyžaduje explicitně zadat příkaz pro tavení a množství filamentu, které má být nataveno.

Tato funkce je ideální pro zkombinování s ovládáním tiskové hlavy.

Aktuální stav trysky

Pro kontrolu správného průběhu tisku je důležité sledovat teplotu trysky. OctoPrint nabízí z tohoto důvodu koncový bod `api/printer/tool`. V odpovědi se nachází informace o akuální teplotě.

Ovládání podložky

Podobně jako u trysky je možné konfigurovat teplotu i u podložky. Koncový bod se nachází na `api/printer/bed`, požadavek je nutné vytvořit metodou `POST`.

Aktuální stav podložky

OctoPrint nabízí také informace o aktuální teplotě podložky. Infomace jsou dostupné na `api/printer/bed`.

Tyto informace je možné zobrazit na přehledu stavu tiskárny.

Ovládání paměťové karty

Aby bylo možné manuálně spravovat stav paměťové karty, nabízí OctoPrint koncový bod `api/printer/sd` Pomocí této funkce lze paměťovou kartu připojit, odpojit nebo vynutit načtení dat.

Jednotlivé konfigurace je nutné v požadavku explicitně uvést. Každý požadavek musí být metodou `POST`.

Aktuální stav paměťové karty

Pro možnost informovat uživatele o akutálním stavu paměťové karty je dostupný koncový bod `api/printer/sd`. Odpověď vrací booleovskou hodnotu indikující je-li karta připravena či ne.

2.8.5 Tiskové profily

OctoPrint umožňuje spravovat tiskové profily, pomocí nichž lze definovat fyzické vlastnosti tiskárny. Mezi tyto vlastnosti patří informace o podložce, maximální rychlost tiskové hlavy či obsah tiskového profilu.

Tiskové profily nejsou v mé implementaci stěžejní funkcí, rozhodl jsem se proto i analýzu nepokrývat podrobně. API poskytuje standardní Create, Read, Update, Delete (CRUD) rozhraní pro správu profilů. Je tedy možné profily vytvářet, číst, aktualizovat a mazat. Rozhraní jsou dostupná na `api/printerprofiles`. Jednotlivým požadavkům pak odpovídají HTTP metody podle CRUD standardu.

Více informací je možné získat z oficiální dokumentace [37], i ta je ale na toto téma velmi skromná.

2.8.6 Aktuální tisk

Jednou z nejzajímavějších funkcí OctoPrint pro mobilní aplikaci je aktuální tisk. Uživatel může jednoduše zkontrolovat, že tisk probíhá v pořádku, případně jak dlouho bude tisk ještě trvat.

Informace o aktuálním tisku

Informace o aktuálním tisku jsou dostupné pomocí koncového bodu `api/job`. Odpověď od serveru obsahuje metadata tisknutého souboru, odhad zbývajícího času a spotřebu materiálu. Dále lze získat také informace o průběhu, tedy kolik procent je hotovo a jak dlouho se soubor tiskne.

Ovládání aktuálního tisku

Ovládání tisku obsahuje mnoho konfigurací. Pro mobilní aplikaci jsem zvolil možnost konfigurace pro zrušení tisku. Pro zrušení aktuálního tisku lze využít `api/job` s metodou `POST`. Do těla požadavku je nutné explicitně vložit příkaz pro zrušení tisku. Více informací o konfiguracích je dostupné v dokumentaci [38].

Vlastní příkaz tiskárně

Protože OctoPrint nepokrývá veškeré funkce tiskárny samostatnými konocovými body, je možné pro odeslat vlastní příkaz tiskárně v jazyce GCode.

Tento koncový bod je dostupný na `api/printer/command`. Vlastní příkaz se vkládá do těla požadavku. Požadavek je nutné vytvořit metodou `POST`.

Tato funkcionalita je vhodná pro vytvoření obrazovky s vlastní terminálovým emulátorem.

2.8.7 Správa log souborů

Užitečnou funkcí OctoPrint je zobrazování logů. OctoPrint seskupuje logy ze svého používání spolu s logy svých služeb (např. ze *slicovacích nástrojů*).

Seznam logů

Jednou z poskytovaných funkcí je seznam logů. V seznamu jsou obsaženy metadata souboru a jeho umístění na souborovém systému. Seznam je dostupný z přístupového bodu `api/logs`.

OctoPrint nenabízí samostatný endpokoncový bod pro detail logu, pro zobrazení obsahu je tedy nutné soubor stáhnout a teprve následně zobrazit.

Smazání logového souboru

Z důvodu zvětšujícího se počtu souborů nabízí OctoPrint možnost manuálně mazat logy. Mazání je možné pomocí funkce `api/logs/[filename]` metodou `DELETE`.

2.8.8 Správa slicing profilů

Obdobně jako u tiskových profilů nejsou slicing profily stěžejní funkcí. Vzhledem k jejich složité konfiguraci by jejich tvoření z mobilní aplikace mohlo způsobit špatný uživatelský zážitek. Z tohoto důvodu jsem neprováděl podrobnou analýzu této funkcionality.

OctoPrint nabízí koncový `api/slicing`, na kterém jsou dostupné CRUD operace. Manipulace je tedy možná pomocí dalších funkcí z nichž každá operuje s odpovídající HTTP metodou.

Podrobné informace k jednotlivým funkcím jsou dostupné v oficiální dokumentaci [39].

2.9 Funkční a nefunkční požadavky

Pro úspěšné dokončení implementace jsem analyzoval také funkční a nefunkční požadavky na aplikaci. Funkčními požadavky se rozumí funkcionalita, kterou výsledná aplikace disponuje. Nefunkční požadavky jsou nároky na aplikaci, které se netýkají přímo jejích funkcionalit.

2.9.1 Funkční požadavky

- Aplikace umí mezi jednotlivými spuštěními uchovat nastavení tiskáren.
- Aplikace uživateli nabízí seznam uložených tiskáren, ale i seznam síťových tiskáren.
- V aplikaci je dostupný video stream aktuálního tisku tiskárny.
- Ke každé tiskárně je možné zobrazit její detail a přehled.
- Z aplikace lze do tiskárny nahrát nový soubor a nechat ho vytisknout.
- Aplikace umožňuje uživateli zadat libovolný příkaz tiskárně.
- Aplikace nabízí rozhraní pro ovládání tiskové hlavy.

2.9.2 Nefunkční požadavky

- Aplikace je naprogramována v jazyku Swift 3.
- Podporovaná verze OctoPrint API je 1.2.15.
- Podporovaný systém je iOS 10.3 a vyšší.

2.10 Doménový model

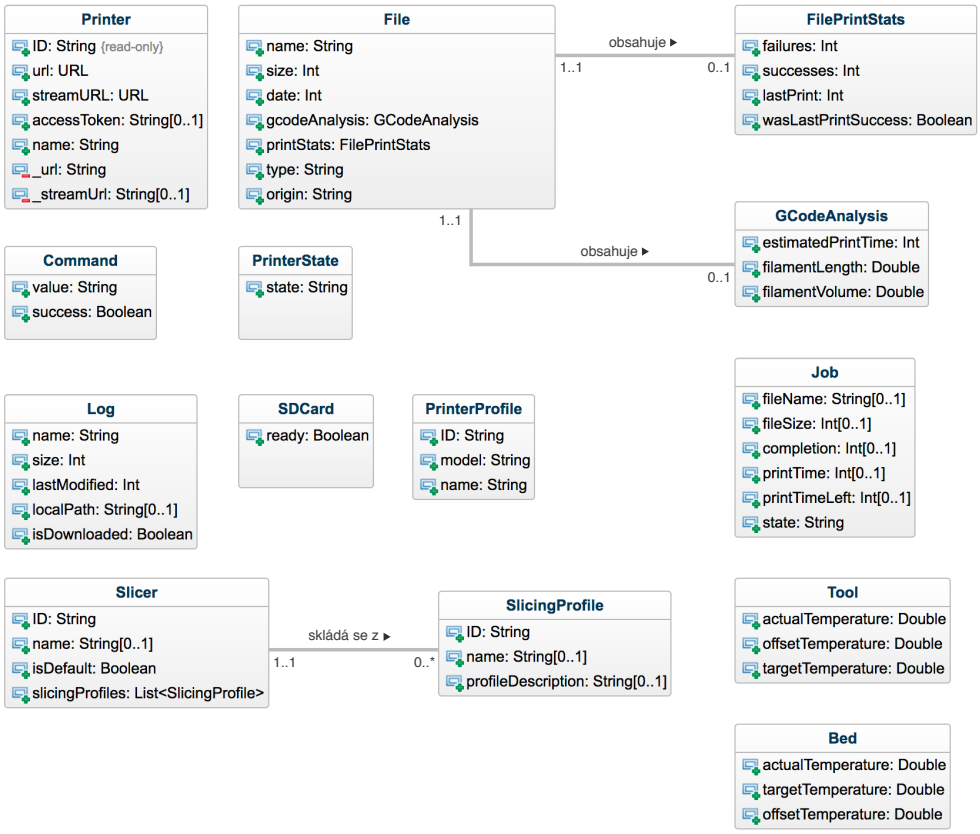
Během analýzy koncových bodů OctoPrint jsem sestavoval doménový model. Tento model pokrývá funkcionalitu koncových bodů, uvažuje také četnosti parametrů a vztahy entit.

Jak je vidět z obrázku 2.6, relace se vyskytují jen mezi některými entitami. To je dáno tím, že aplikace v jednu chvíli pracuje vždy s právě jednou instancí tiskárny.

Objekty týkající se tiskárny se tedy vždy vytváří až při výběru jedné konkrétní. Po ukončení práce s tiskárnou se opět mažou.

Tento přístup je vhodný, protože informace o tiskárně se pravděpodobně mezi jednotlivými spuštěními aplikace změní. Uchování starších dat sice působí při používání lépe, protože aplikace nabízí data ihned po otevření, nemusí být ale aktuální, což by vedlo ke zmatení uživatele.

Jediná data, která se uchovávají mezi jednotlivými spuštěními je seznam tiskáren.



Obrázek 2.6: Doménový model aplikace

Kapitola 3

Implementace

V této kapitole se budu podrobně věnovat použitým technologiím a implementaci jednotlivých funkcionalit. V první části shrnu jaké technologie analyzované v kapitole 2 jsem zvolil. V dalších částech se věnuji konkrétním funkcionalitám navrženým podle analýzy API v části 2.8.

3.1 Návrh vzhledu

Po úplném zanalyzování problému je přistoupil k návrhu aplikace. Uživatelské rozhraní jsem se snažil tvořit jednoduché a intuitivní.

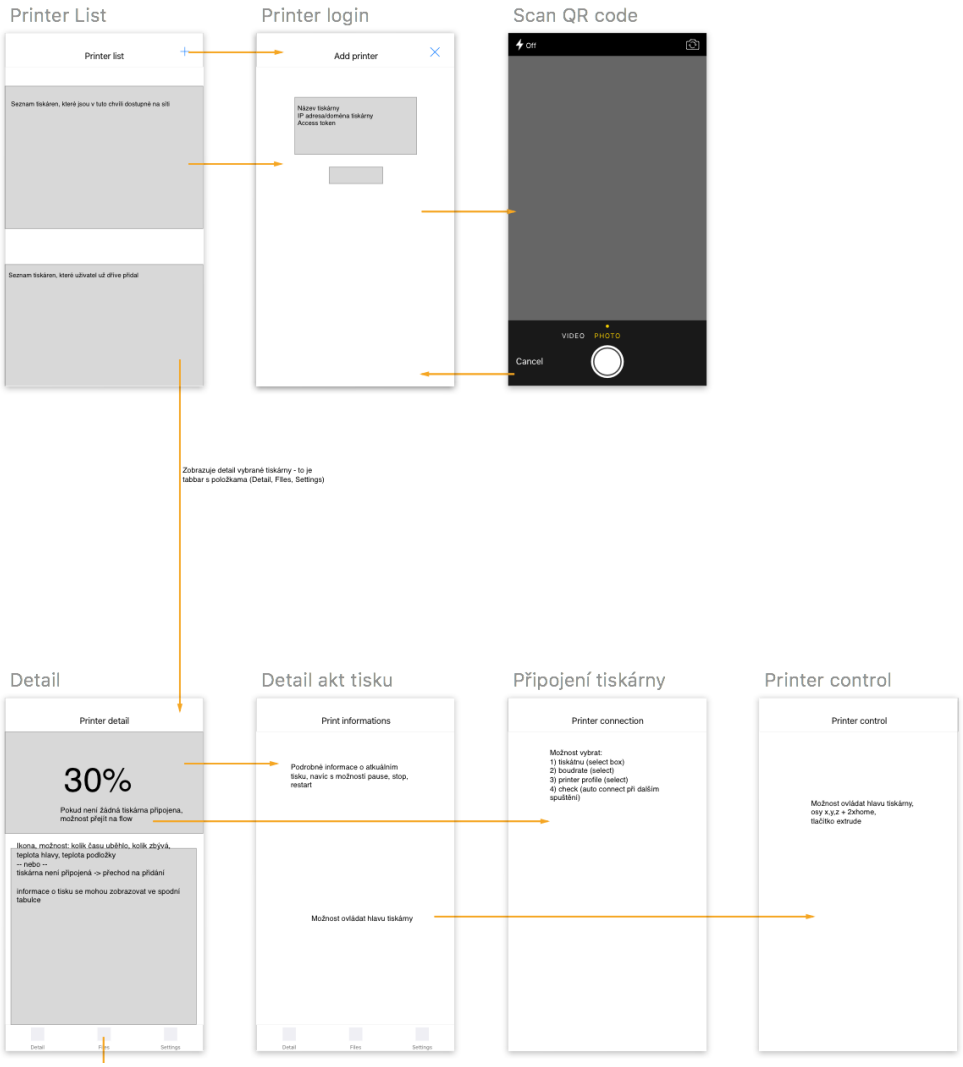
Návrh jsem vytvářel z kostry aplikace. Nejdříve jsem ze seznamu funkcionalit vybral takové, které se budou nacházet na jedné obrazovce. Následně jsem vytvořil mapu obrazovek. Část mapy lze vidět na obrázku 3.1.

V mapě obrazovek jsem také znázornil průchod aplikací a u některých funkcionalit vytvořil wireframe či výčet funkcí. V návrhu jsem se dále nepouštěl do kompletních grafických zpracování. Vzhledem obrazovek a jejich rozložením jsem se více zabýval až při konkrétním návrhu.

Obecným požadavkem byl univerzální vzhled. iOS umožňuje vytvářet univerzální aplikace, které fungují s jedním vzhledem na iPhone i iPad. Design se programuje relativně, tím je zajištěno, že funguje i na větších zařízeních. [40] Tímto způsobem jsem optimalizoval aplikaci pro obě zařízení.

3.2 Použité technologie

V této části se postupně zabývám jednotlivými body analýzy a rozebírám jaké řešení z těch, která jsem analyzoval, jsem zvolil pro implementaci. Jedním z hlavních kritérií při konečném rozhodování byla slučitelnost jednotlivých částí. Vzhledem k rozsahu implementace jsem si musel být jistý, že jednotlivé části budou fungovat správně v celé aplikaci. Rozhodujícím faktorem tedy nebyla popularita jednotlivých řešení, ale spíše jejich flexibilita.



Obrázek 3.1: Mapa obrazovek

3.2.1 Architektura aplikace

Nejdůležitějším rozhodnutím bylo správné vybrání architektury. Špatný výběr architektury by mohl mít dopad na celkovou funkčnost aplikace či znemožnit implementaci některých z vyžadovaných funkcí.

Po podrobném analyzování obou zmíněných architektur jsem se rozhodl využít MVVM. Přestože MVVM se v současné době jeví méně rozšířené, nabízí spoustu vlastností, které MVC nemá.

Silným argumentem je testování. V architektuře MVVM lze jednotlivé části otestovat samostatně. K otestování logiky aplikace navíc není potřeba vizuální vrstva aplikace.

Neméně podstatným argumentem je čitelnost kódu. Vrstva `Controller` obsahuje mnohem méně kódu a lze se v něm snáze orientovat. `ViewModel` naopak neobsahuje žádné prvky uživatelského rozhraní a znázorňuje tak pouze způsob, jakým se daná část aplikace chová. Jednotlivé vrstvy jsou od sebe striktně odděleny a dodržují princip jedné odpovědnosti [41].

Implementace ViewModelu

V části 2.2.2 kde jsem analyzoval architekturu MVVM jsem naznačil strategii implementace. `ViewModel` jsem se rozhodl implementovat jako logiku `Controlleru`, která jako rozhraní nabízí pouze vstupy či výstupy.

Vstupem označuji uživatelskou interakci, na základě které je potřeba vyžádat nová data nebo zobrazit jinou obrazovku. Výstupem jsou zobrazovaná data, která jsou `ViewModelem` předformátovaná, aby je `Controller` mohl přímo zobrazit pomocí `View` vrstvy. Jiným výstupem pak může být informace o stavu dat či o jejich počtu (počet položek v seznamu).

Každá obrazovka vlastní svůj `ViewModel`. Pro ten jsou dostupné tři protokoly: `ViewModelInputs`, `ViewModelOutputs` a `ViewModelType`. Tyto prokoly jsou konkrétně pojmenovány podle obrazovky, stejně tak `ViewModel` samotný a také `Controller`. Definice rozhraní je vidět v ukázce 3.1.

První protokol popisuje množinu vstupů, druhý množinu výstupů a třetí popisuje rozhraní `ViewModelu`. Tím jsem dosáhl možnosti zaměňovat `ViewModel` obrazovky či sdílet jeden `ViewModel` mezi více obrazovkami. Návrhem tohoto rozhraní jsem inspiroval v open-source aplikaci Kickstarter, dostupné na portálu GitHub [42].

Rozhraní konktrétního `ViewModelu` pro obrazovku se seznamem tiskáren je vidět v diagramu tříd na obrázku 3.2.

Implementace Controlleru

`Controller` při inicializaci vyžaduje instanci `ViewModelu`. Veškerou logiku obrazovky pak na `ViewModel` během svého životního cyklu deleguje.

Uživatelské vstupy jsou implementovány pomocí funkcí. `Controller` sleduje uživatelskou interakci a volá patřičné funkce `ViewModelu`. Výstupy z

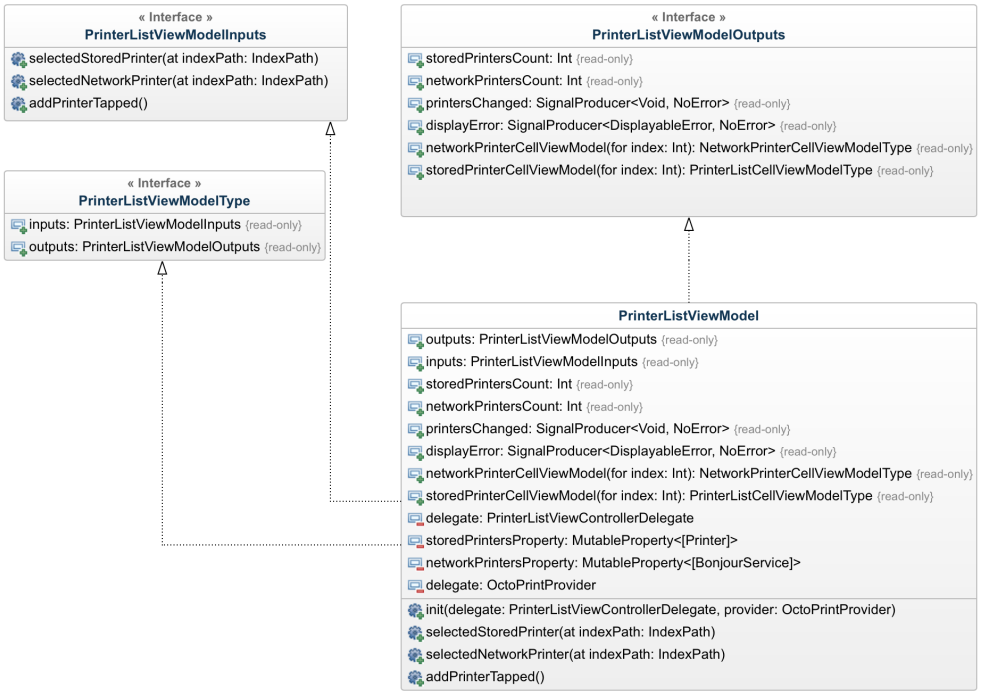
```swift
/// Rozhraní vstupů
protocol PrinterDetailViewModelIntputs { /* ... */}

/// Rozhraní výstupů
protocol PrinterDetailViewModelOutputs { /* ... */}

/// Rozhraní ViewModelu pro Controller
protocol PrinterDetailViewModelType {
    var inputs: PrinterDetailViewModelIntputs { get }
    var outputs: PrinterDetailViewModelOutputs { get }
}

/// Implementace ViewModelu
class PrinterDetailViewModel: PrinterDetailViewModelIntputs,
PrinterDetailViewModelOutputs, PrinterDetailViewModelType {
    /* ... */
}
```

Ukázka kódu 3.1: Rozhraní objektu ViewModel



Obrázek 3.2: Diagram tříd pro ViewModel seznamu tiskáren

`ViewModelu` jsou pomocí `UI bindings` napojeny na `View` v metodě `bindViewModel`, která je volána těsně po načtení `View` vrstvy do paměti.

`Controller` vyžaduje v konstruktoru instanci implementující protokol vstupů a výstupů, nikoli konkrétní třídu.

Coordinator

MVVM standardně nepoužívá objekt spravující navigaci v aplikaci. Implementoval jsem proto další objekt nazvaný `Coordinator`. Jedná se o objekt, který se stará o průběh jedné konkrétní části aplikace (flow). Jakmile uživatel vyžaduje přechod na obrazovku mimo rozsah stávajícího `Coordinatoru`, vytvoří se nový `Coordinator` starající se o novou obrazovku a její životní cyklus. `Coordinator` má také na starosti vytvoření `ViewModelu` a předání jeho instance `Controlleru`.

Scénář obrazovky se spustí metodou `start`. Po ukončení (uživatel chce zavřít obrazovku) volá `Coordinator` metodu `completed` kde se uvolní naalokované zdroje.

S `Coordinatorem` spolupracuje `ViewModel` metodou Delegate pattern.

3.2.2 Synchronizace vláken

Pro synchronizaci vláken jsem se rozhodl použít knihovny `ReactiveCocoa` a `ReactiveSwift`, tedy reaktivní přístup. Díky těmto knihovnám lze jednoduše tvořit závislosti asynchronních operací.

S využitím knihovny `ReactiveCocoa` od verze 5.0 lze navíc využít tkzv. `UI bindings`. Ty zaručují, že hodnoty signálů jsou vždy zpracovány na hlavním vlákně a to i v momentě, kdy jsou odeslány z vlákna v pozadí. `ReactiveSwift` pak zaručuje konzistenci dat mezi vlákny. Dohromady tak tyto knihovny zamezují vzniku `race condition` za běhu aplikace. Více o těchto knihovnách lze zjistit z oficiální dokumentace [43] a [22].

Neméně podstatným faktorem byla i velmi dobrá integrace v architektuře MVVM. Pomocí operátorů nad signály lze formátovat vlastnosti modelových objektů (přidání jednotek, standardizace čísel, zástupné texty). Takto naformátované signály zaručí, že kdykoliv se aktualizuje modelový objekt, jeho vlastnosti budou správně naformátovány. `ViewController` tak vždy dostane data ve správném formátu bez ohledu na to, jakým způsobem byly změny na `Modelu` aplikovány.

3.2.3 Síťová vrstva

Síťovou vrstvu jsem se zpočátku rozhodl implementovat knihovnou `Alamofire`. Důvodem byla snazší implementace a integrace do aplikace. Protože nemá `Alamofire` striktně danou strukturu koncových bodů, bylo jednoduché implementovat dynamickou URL tiskárny.

Chybějící podpora reaktivního programování ale velmi brzy negativně zasáhla běh aplikace. K synchronizaci vláken jsem nemohl využít `ReactiveSwift` a aplikace byla zatížena množstvím chyb vznikajících při vykonávání síťových požadavků a následné serializaci dat.

Z tohoto důvodu jsem se rozhodl využít knihovny `Moya`, která nabízí reaktivní rozšíření a je kompatibilní s `ReactiveSwift`. Jedinou výjimkou je komunikace přes stálé připojení (sockety). To je více rozebráno v části 3.5.3.

Veškeré síťové požadavky nakonec využívají knihovnu `Moya`, s jejíž implementací chyby s vlákny vymizely.

Implementace Provider objektu

Při implementaci jsem se neobešel se standardními objekty, které jsou běžné potřebné. `Provider` objekt předimplementovaný v knihovně předpokládá, že adresa vzdáleného serveru je známá již během kompilace a neumožňuje ji zadat za běhu programu. Z tohoto důvodu jsem také nemohl využít standardní definici koncových bodů pomocí `TargetType` protokolu, který URL vyžaduje staticky.

Rozhodl jsem vytvořit podtřídu `DynamicProvider`, která v konstruktoru přijme URL serveru. Místo využití protokolu `TargetType` jsem vytvořil nový protokol `TargetPart`, který má stejné rozhraní, jen nevyžaduje URL serveru.

Má implementace `Provider` objektu využívá tento nový protokol pro vytvoření požadavku. Abych mohl využít třídu, od které můj objekt `DynamicProvider` dědí, musel jsem ještě vytvořit strukturu `DynamicTarget`, která implementuje rozhraní `TargetType` a využaduje tedy URL serveru. Tu jí ale může poskytnout v konstruktutoru `DynamicProvider` spolu s `TargetPart` koncovým bodem.

Tím jsem dosáhl stejného rozhraní pro požadavky jako má nadtřída, jediným rozdílem je možnost dodat URL až při běhu aplikace.

Síťové požadavky mimo OctoPrint

Pro umožnění zobrazení video streamu jsem opět potřeboval dynamickou URL, ale tentokrát pro každý požadavek (uživatel zadává pouze URL, nejsou žádné koncové body). Tato implementace nebyla složitá, využil jsem standardních tříd knihovny `Moya`. Vytvořil jsem nový objekt spravující koncové body a v něm definoval pouze jeden bod `get`. Ten pro zkonstruování využaduje koncovou URL a chová se tedy velmi podobně jako kdyby URL byla známá už v čase kompilace.

Při využití je ale nutné pro každý požadavek explicitně URL zadat. To je krok, který v `DynamicProvider` není nutný.

3.2.4 Datová vrstva

Jak vyplývá z části 2.5 kde popisuji datovou vrstvu, rozhodl jsem se využít knihovnu `Realm`. Díky jejím reaktivním rozšířením se skvěle hodí nejen k architektuře, ale i k síťové vrstvě.

Při implementaci jsem dospěl k závěru, že standardní cestou využití `Realm` opakuji spoustu kódu. Vytvořil jsem proto rozšíření pro `ReactiveCocoa` a její `Cold` signál.

Pro `SignalProducer`, který jako hodnotu nese `Realm` objekt jsem přidal dvě metody. První z nich, `fetch(collectionOf:)` umožňuje z databáze načíst kolekci. Druhou metodou je `fetch(classType:, forPrimaryKey:)`. Ta vybere z databáze prvek podle primárního klíče. Využití metod je vidět v ukázce 3.2.

```
1  /// Databázové připojení
2  let realm = try! Realm()
3  let producer = SignalProducer<Realm, RealmError>(value: realm)
4
5  /// Kolekce pruků z databáze
6  producer.fetch(collectionOf: File.self)
7      .startWithValues { files in
8          /* ... */
9      }
10
11 /// Jeden konkrétní prvek podle primárního klíče
12 producer.fetch(File.self, forPrimaryKey: "file.gcode")
13     .startWithValues { file in
14         /* ... */
15     }
```

Ukázka kódu 3.2: Reaktivní rozšíření pro Realm

Díky těmto rozšířením jsem odstranil spoustu duplicitního kódu, navíc jsem sjednotil rozhraní využívaných knihoven. Mohl jsem díky tomu synchronizovat vlákna síťových volání a datové vrstvy pomocí `ReactiveCocoa`.

3.2.5 Grafické prvky

Pro ilustrace a ikony v aplikaci jsem se rozhodl využít framework `Iconic`. Oproti zmíněnému Assets catalog v mém řešení výkon aplikace nezhoršuje, naopak poskytuje velmi snadným způsobem desítky ikon v jediném fontovém souboru.

Jako grafický font jsem vybral FontAwesome verze 4.7 [44]. Grafický font jsem využil pro ikony záložek na obrazovce detailu tiskárny ale také pro ikony, které nejsou běžně v systému dostupné.

Protože pomocí grafického fontu není možné vytvořit ikonu aplikace, využil jsem také Assets catalog. Využit je ale právě pro ikonu aplikace, ostatní potřebné ilustrace poskytuje `Iconic`. K sestavení ikony aplikace jsem také využil FontAwesome grafický font, výsledný obrázek jsem exportoval do formátu `png`.

`Iconic` v současné době není podporován správcem závislostí, který jsem si zvolil. Z tohoto důvodu jsem potřebné soubory pomocí frameworku vygeneroval zvlášť a následně je do projektu vložil.

Současně jsem také nabídl autorovi frameworku pomoc s podporou pro jiné cesty distribuce. Během implementace této práce se nám ale nepodařilo možnosti distribuce rozšířit. Více informací o tomto tématu je dostupné na stránce projektu [45].

3.2.6 Správa závislostí

Při výběru správce závislostí jsem své nároky směřoval především na ovlivnění kompilace aplikace. V konečném důsledku pracují oba zanalyzované nástroje podobně, mají ale rozdílný přístup k linkování knihoven.

Přestože nastavení `CocoaPods` je mnohem snazší, z mého pozorování vyplynulo, že negativně ovlivňuje čas kompilace. To je způsobeno častou kompilací jednotlivých knihoven a to i v případě, že jejich zdrojový kód nebyl upraven. Dále jsem vypozoroval, že prostředí Xcode je s `CocoaPods` méně stabilní, přestává fungovat našeptávání a zvýrazňování syntaxe.

Z těchto důvodů jsem zvolil `Carthage`. Počáteční nastavení bylo nepatrně složitější z důvodu ručního integrování knihoven do projektu. Integrují se ale předem zkompilované knihovny, čas jednotlivých buildů aplikace se tedy nezvyšuje. Linkování zkompilovaných knihoven má pravděpodobně za následek lepší integraci s Xcode. Za dobu psaní práce se stalo jen výjimečně, že by se Xcode nečekaně ukončil. Zlepšilo se také zvýrazňování syntaxe. To fungovalo během psaní celé práce bez problému.

Výměnou za zlepšené fungování při lokálním vývoji byly problémy při kompilaci na `CI` serveru. Nekompatibilita zkompilovaných knihoven s verzí Swift jazyka použitým při vývoji aplikace způsobila, že více než dvě třetiny kompilací na serveru selhalo. Přehled jednotlivých buildů je vidět na [46].

3.3 Seznam dostupných tiskáren

Aplikaci jsem navrhl tak, aby klíčové funkcionality byly dostupné s co nejkratším průchodem aplikace. Jako úvodní obrazovku jsem zvolil seznam tiskáren. V případě, že již uživatel aplikaci dříve používal, zobrazí se v seznamu na prvních místech tiskárny, které si dříve uložil. Na dalších místech jsou pak tiskárny dostupné na stejné síti, které aplikace automaticky nalezla.

Pokud aplikace požadovanou tiskárnu nenalezla, je ze seznamu tiskáren možné přejít na obrazovku pro manuální přidání tiskárny jejím výběrem.

3.3.1 Implementace seznamu

Abych dosáhl uživatelského rozhraní kompatibilního se zařízeními iPhone i iPad, rozhodl jsem se seznam implementovat pomocí `CollectionView`. Z analýzy vyplynulo, že velmi podobné seznamy budou dostupné na většině obrazovek. Rozhodl jsem se tedy zavést novou třídu pokrývající společnou logiku a výchozí nastavení nazvanou `BaseCollectionView`.

Tato základní třída se stará o barevné sjednocení obrazovek, zobrazování chyb a nastavení `View` pro prázdné obrazovky. Jednotlivé obrazovky pak využívají třídy dědící z `BaseCollectionView`.

Sekce seznamu

Seznam tiskáren je rozdělen na dvě základní sekce. Pomocí rozdělení do samostatných sekcí jsem dosáhl vytvoření nezávislých řádkových indexů pro každou sekci. `ViewModel` pro `Controller` poskytuje počet řádků pro každou ze sekcí samostatně. Při vytváření seznamu stačí tedy ověřit, pro jakou sekci se záznam v seznamu vytváří a následně `ViewModel` požádat o data s konkrétním řádkovým indexem číslovaným od nuly.

V případě jedné sekce by bylo nutné vzájmně porovnávat aktuální číslo řádku s počtem uložených tiskáren a tiskáren nalezených. To požaduje komplexní výpočet a dává prostor mmoha chybám.

Obrazovka s rozdělením do sekcí je vidět na obrázku 3.3.

Uložené tiskárny

První sekce zobrazuje tiskárny načtené z lokální databáze. Při sestavování seznamu si `CollectionView` pomocí Delegate pattern nejdříve vyžádá počet prvků pro tuto sekci. `Controller` tedy využije výstup `ViewModelu` a vrátí hodnotu jeho proměnné nazvané `storedPrintersCount`. Implementace této metody je vidět v ukázce 3.3.

```
override func collectionView(_ collectionView: UICollectionView,
                             numberOfItemsInSection section: Int
) -> Int {
    switch section {
    // Sekce uložených tiskáren
    case 0: return viewModel.outputs.storedPrintersCount
    // Sekce automaticky nalezených tiskáren
    default: return viewModel.outputs.networkPrintersCount
    }
}
```

Ukázka kódu 3.3: Konfigurace počtu tiskáren v seznamu

Obrázek 3.3: Ukázka aplikace: Seznam tiskáren

V momentě kdy `CollectionView` ví kolik prvků bude zobrazovat, začne vyžadovat jednotlivé položky seznamu. K tomu je opět využit Delegate pattern. Pro správné dodržení MVVM architektury nesmí `Controller` využívat modelové objekty. Pro každou buňku je tedy potřeba vytvořit `ViewModel`, který ji bude obsluhovat. `Controller` si pro řádkový index vyžádá od svého `ViewModelu` nový `ViewModel` buňky a ten jí předá. Buňka se následně sama z `ViewModelu` nakonfiguruje. Vyžádání `ViewModelu` buňky zachycuje ukázka 3.4.

Síťové tiskárny

Tato část názorně ukazuje sílu MVVM architektury. Přesto, že logika stojící za vyhledáním síťových tiskáren není triviální, `ViewController` zůstává velmi krátký a dobře čitelný.

Z pohledu `Controlleru` totiž logika není podstatná, důležité jsou výstupy z ní. `ViewModel` pro seznam tiskáren jsem proto navrh tak, aby poskytoval

```
override func collectionView(_ collectionView: UICollectionView,
                             cellForItemAt indexPath: IndexPath
) -> UICollectionViewCell {
    // Vyžádání objektu buňky pro CollectionView
    let cell = collectionView.dequeueReusableCell(
        withReuseIdentifier: "identifier",
        for: indexPath
    ) as! PrinterListCollectionViewCell

    // Vyžádání nového ViewModelu pro buňku
    cell.viewModel =
        viewModel.outputs.storedPrinterCellViewModel(
            for: indexPath.row
        )

    return cell
}
```

Ukázka kódu 3.4: Vytváření buňky v seznamu

obdobné rozhraní jako poskytuje pro uložené tiskárny.

Pro zobrazení druhé sekce seznamu stačí `CollectionView` předat informaci o počtu síťových tiskáren a buňkám předat odpovídající `ViewModely`. Toho se docílí ve stejných metodách jako pro uložené tiskárny. Počet síťových tiskáren je uložen v proměnné `networkPrintersCount` a `ViewModely` buněk je možné získat metodou `networkPrinterCellViewModel`.

3.3.2 Automatické nalezení tiskáren

Pro zlepšení uživatelského zážitku jsem se snažil minimalizovat nutnost konfigurace aplikace. Jako uživatelsky nejméně přívětivé se jeví přidávání tiskárny. Uživatel musí zjistit IP nebo URL adresu OctoPrintu, jeho port a přístupový token.

Z tohoto důvodu je v aplikaci dostupné automatické hledání tiskáren. V případě, že je zařízení s nainstalovanou aplikací připojeno ke stejné síti jako tiskárna, je možné ji nechat nalézt automaticky.

Síťové tiskárny jsou v aplikaci jako druhá část seznamu tiskáren. K jejich načtení a aktualizaci nemusí uživatel provádět žádnou akci, stačí aplikaci zapnout.

Aplikace pomocí technologie `Bonjour` prozkoumá lokální síť a uživateli zobrazí dostupná zařízení. Rozhodne-li se uživatel přidat síťovou tiskárnu, jejím výběrem otevře obrazovku pro přihlášení. Na obrazovce jsou předvyplněné údaje, které aplikace o tiskárně zjistila. Konkrétně se jedná o její síťový název a adresu s portem. Pro dokončení přidání stačí pouze zadat přístupový token.

Technologie Bonjour

Bonjour je balík síťových technologií od firmy Apple. Pomocí Multicast Domain Name Service (mDNS), které zajišťuje převod hostname zařízení na IP adresu, umí Bonjour na lokální síti nalézt počítače, tiskárny a jiná zařízení. Na systémech iOS a macOS jsou tyto nástroje předinstalovány, na jiné platformy je potřeba je doinstalovat. Více informací k Bonjour je dostupné na stránkách Apple [47].

Reaktivní rozšíření

Standardní knihovna nabízí pro práci s Bonjour třídy `NetService` a `NetServiceBrowser`, které využívají Delegate pattern [48]. Tento přístup mi nevyhovoval, protože vnesl mnoho kódu do `ViewModel` objektu a zhoršil tím jeho přehlednost a čitelnost. Kód byl navíc přímo vázaný na logiku jedné obrazovky což znemožňovalo jeho znovuvyužitelnost.

Rozhodl jsem vytvořit novou třídu `Bonjour`, která místo Delegate pattern nabízí reaktivní přístup. Třída je implementována jako `Singleton` [49] poskytující pouze metodu `searchServices`.

Tato metoda spustí vyhledávání síťových zařízení (které je asynchronní) a vrátí signál, na kterém budou jako hodnoty posílány `BonjourService`, tedy objekty reprezentující tiskárny. Uvnitř třídy nejprve proběhne nalezení všech dostupných hostname (názvů síťových zařízení). Ve chvíli kdy systém ohlásí, že prohledal celou síť, spustí se překlad hostname na IP adresu.

Systém po jednom projde nalezená zařízení a pokusí se zjistit jeho IP adresu. Pokud je překlad úspěšný, zařízení se přidá do pole nalezených tiskáren a celé pole se odešle signálem.

Díky reaktivnímu přístupu tak `Controller` dostane informaci o novém prvku, který je nutné zobrazit a překreslí seznam.

3.3.3 Automatické aktualizace

Abych zajistil přijemnou uživatelskou zkušenost z aplikace, rozhodl jsem se seznam tiskáren implementovat jako `push-based`. Tato strategie udává, že při změně dat jejich vlastník (např. `ViewModel`) upozorní své odběratele (`Controller`). Každý odběratel se sám rozhodne, jakým způsobem informaci využít. V mé konkrétní implementaci lze vidět rozdíl od `pull-based` strategie v automatických aktualizacích.

Strategie `pull-based` spočívá v tom, že data jsou dostupná na požádání. To znamená, že změní-li se data (např. bude-li nalezena nová síťová tiskárna), odběratel si je explictině musí vyžádat. Zde vzniká otázka, jak často se dotazovat, zda změna proběhala či nikoli.

Odpověď se odvíjí konkrétního využití. Některá data není potřeba po načtení aktualizovat vůbec, jiná naopak prvidelně s krátkými intervaly.

Abych zajistil konzistenci napříč celou aplikací, rozhodl jsem se využít strategii `push-based` směrem z `ViewModelu` do `Controlleru`. `Controller` sleduje signál změn a pokaždé když se na signálu objeví hodnota (obvykle typu Void), překreslí odpovídající `View`.

Za pomoci `UI bindings` frameworku `ReactiveCocoa` jsem napojil signál změn na metodu překreslení seznamu. `Controller` tedy nemusí o data explicitně žádat. Kdykoliv se totiž změní vyšle `ViewModel` signál a tabulka se překreslí. Pro překreslení se využívá stejný postup jako při iniciálním načtení, nehrozí proto problém s čátečnou nekonzistencí. Výsledkem je, že uživatel vidí na obrazovce vždy aktuální data. Ukázka 3.5 demonstruje překreslení seznamu pomocí `UI bindings`.

```
1 func bindViewModel() {
2     let outs = viewModel.outputs
3     collectionView.reactive.reloadData <~ outs.printersChanged
4 }
```

Ukázka kódu 3.5: Překreslení CollectionView pomocí UI bindings

3.3.4 Přidání tiskárny

V momentě kdy nebude požadovaná tiskárna automaticky nalezena a zároveň ji uživatel dříve nepřidal, je možné tiskárnu přidat ručně. Přidání tiskárny se odehrává na nové obrazovce. Na tu se uživatel dostane stisknutím tlačítka „+“ na obrazovce se seznamem tiskáren.

Pro vytvoření obrazovky je potřeba nový `Controller` a k němu odpovídající `ViewModel`. Za zprostředkování nové obrazovky zodpovídá `Coordinator` zmíněný v sekci 3.2.1.

`Controller` seznamu tiskáren vytvoří vstup pro svůj `ViewModel`, ten na základě vstupu notifikuje `Coordinator` o uživatelské interakci. `Coordinator` má předem definováno jakým způsobem na interakce reagovat. Průběh je vidět v ukázce 3.6.

Na základě notifikace `Coordinator` vytvoří potřebné objekty k sestavení obrazovky a prezentuje ji uživateli. Až do uzavření obrazovky je seznam tiskáren neaktivní.

Pokud uživatel tiskárnu úspěšně přidá, seznam je automaticky aktualizovaný díky `push-based` strategii.

3.3.5 Detail tiskárny

Pro interakci s tiskárnou je nutné otevřít její detail. Ten uživatel otevře vybráním uložené tiskárny ze seznamu.

```
// Funkce napojená na stisknutí tlačítka v Controlleru
func addPrinterButtonTapped() {
    // Předání vstupu ViewModelu
    viewModel.inputs.addPrinterButtonTapped()
}

/* ... */

// Funkce ViewModelu reagující na vstup
private func addPrinterButtonTapped() {
    // Notifikace Coordinatoru na uživatelskou interakci
    delegate?.addPrinterButtonTapped()
}
```

Ukázka kódu 3.6: Průběh zobrazení nové obrazovky

Průběh je velmi podobný výběru tiskárny. Opět je nutné zpropagovat uživatelskou iterakci z `View` do `Coordinatoru` pomocí `Controlleru` a `ViewModelu`.

3.4 Přidání nové tiskárny

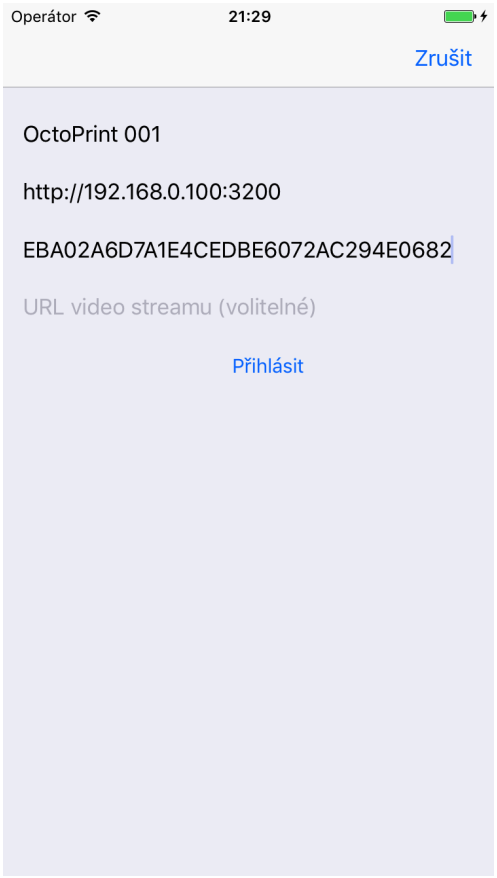
Obrazovka přidání tiskárny slouží pro případ, kdy uživatel chce ovládat novou tiskárnu a nenalezl ji v seznamu síťových tiskáren. Pro úspěšné přidání tiskárny je nutné zadat její název, URL či IP adresu a přístupový token. Volitelně je možné také přidat url adresu, na které se vyskytuje video stream z web kamery natáčející průběh tisku. Ukázka obrazovky pro přidání aplikace je na obrázku 3.4.

3.4.1 Validace formuláře

Jako u každé aplikace, která přijímá uživatelské vstupy i v mé implementaci je nutné ověřit platnost zadaných údajů. Jelikož se jedná o logiku, je validace umístěna ve `ViewModelu`. Text zadaný uživatelem se při každé změně odešle `ViewModelu`, který všechny vstupy zvaliduje. Na základě platnosti kombinace vstupů vyšle `ViewModel` booleanovskou hodnotu `Controlleru`. `Controller` pomocí `UI bindings` sváže tuto hodnotu s blokací tlačítka „Přihlásit“.

Je-li formulář platný, tlačítko je povolené. V opačném případě je zablokované a nelze stisknutím spustit jeho interakci.

Hodnoty formuláře jsou validované pomocí operací nad signály. Signály vstupů jsou nejprve zkombinovány a následně `map` operátorem převedeny na booleanovskou hodnotu. Použití validace nad signály je vidět v ukázce 3.7.



Obrázek 3.4: Ukázka aplikace: Přidání tiskárny

```
// Zkombinování nejnovějších hodnot formuláře
let formFields = Signal.combineLatest(/* ... */)

// Zvaliduje hodnoty kdykoliv se libolná
// z nich změní
let isFormValid = formFields.map(validateFields)

// Funkce validující hodnoty formuláře
func validateFields(_ name: String, _ url: String,
                    token: String, stream: String?
) -> Bool {
    /* Validační logika */
}
```

Ukázka kódu 3.7: Validace formuláře pomocí signálů

3.4.2 Uložení tiskárny

Ve chvíli kdy je formulář ověřený, může uživatel připojení uložit do lokální databáze. Uložení provede pomocí tlačítka „Připojit“. `ViewModel` upozorněný na interakci uživatele načte aktuální hodnoty formuláře a vytvoří požadavek na tiskárnu.

Je-li odpověd od tiskárny pozitivní (status kód odpovědi je v rozmezí 200 až 299), pak se hodnoty uloží do lokální databáze. `Coordinator` je následně upozorněn na ukončení scénáře a obrazovku zavře. Díky reaktivnímu přístupu strategii se rovnou tiskárna zobrazí v seznamu mezi dostupnými k použití. `ViewModel` ani `ViewController` seznamu nemusí být o výsledku přidání tiskárny nijak notifikovány, o propagaci této informace se postará datová vrstva aplikace.

V případě selhání požadavku je vytvořena chybová hláška. Díky reaktivním rozšířením, které jsem vytvořil pro `ViewController` třídu chybu zobrazit pomocí `UI bindings`.

Ukázka 3.8 zjednodušeně implementuje vytvoření požadavku a následné zpracování odpovědi od tiskárny.

```
1  // Načtení hodnot v momentě stisknutei tlačítka
2  formFields.sample(on: loginButtonPressed)
3      .map({ values -> (Printer, OctoPrintProvider)
4        /* Vytvoření Moya Provider objektu a objektu tiskárny */
5      })
6      .flatMap(.latest) { printer, provider in
7          // Vytvoření požadavku na přihlášení
8          return provider.request(.version)
9          // Validace status kódu odpovědi
10         .filterSuccessfulStatusCodes()
11     }
12     // Blok zpracování odpovědi
13     .startWithResult { result in
14         switch result {
15         case .success: /* Uložení tiskárny */
16         case let .failure(error): /* Zpracování chyby */
17         }
18     }
```

Ukázka kódu 3.8: Vytvoření požadavku pro přihlášení k tiskárně

3.5 Detail a přehled tiskárny

V momentě, kdy má uživatel přidanou tiskárnu s platným přístupovým tokenem, může se jejím výběrem ze seznamu dostupných tiskáren dostat na detail.

Obrazovka detailu se skládá ze tří hlavních záložek, z nichž každá pak v rámci svých funkcionalit prezentuje další obrazovky.

První částí, která se uživateli zobrazí je přehled tiskárny. Přehled je klíčovou funkcí aplikace. Umožňuje totiž velmi rychle získat základní údaje o probíhajícím tisku.

3.5.1 Připojení tiskárny

OctoPrint se může nacházet ve stavu, kdy je připravený k použití, ale není připojen k tiskárně. V tuto chvíli nelze získat informace o aktuálním tisku ale ani o tiskárně samotné.

Z tohoto důvodu jsem implementoval „prázdnou obrazovku“, která uživatele upozorňuje na odpojenou tiskárnu a vyzývá ho k jejímu připojení. Fakt, že data nejsou dostupná se propaguje z `ViewModelu`. Ten vysílá signál nazvaný `contentIsAvailable` booleanovského typu. Obrazovka s výzvou k připojení tiskárny je dostupná právě když hodnota signálu je `false`.

Pokud se uživatel rozhodne tiskárnu připojit, může k tomu využít tlačítko „Připojit“. To zařídí otevření nové obrazovky. Na nové obrazovce může uživatel ze seznamu dostupných portů vybrat, k jaké tiskárně se má OctoPrint připojit. Běžně tento seznam bude obsahovat právě jednu položku, protože každou tiskárnu obsluhuje právě jedna instance OctoPrint.

Tuto obrazovku zachycuje obrázek 3.5.

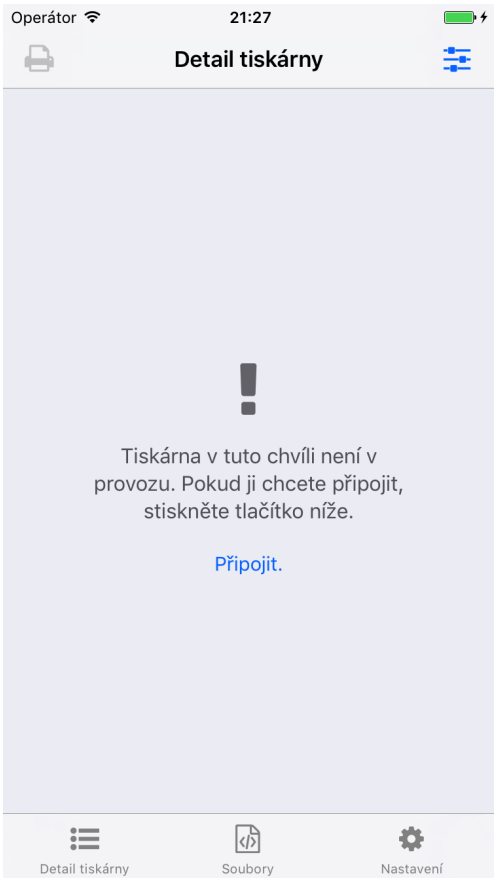
3.5.2 Aktuální tisk

V momentě kdy je tiskárna v pořádku připojena k OctoPrint rozhraní, zobrazuje aplikace informace o jejím stavu. Součástí těchto informací jsou teploty podložky a trysky, aktuální tisk a video stream. Obrazovka je zachycena na obrázku 3.6.

Protože všechny tyto informace nejsou dostupné z jediného koncového bodu, aplikace požadavky řetězí. Postupně jsou volány požadavky pro aktuální stav tiskárny, aktuální tisk, informace o trysce a informace o podložce. Implementace řetězení je vidět v ukázce 3.9.

Řetězení pomocí reaktivního přístupu přináší možnost chyby řešit pomocí `railway` strategie. Tedy řetězení probíhá po „úspěšné koleji“ až do výskytu první chyby. Chyba slouží jako výhybka z této koleje a dále pokračuje po „chybné koleji“. Tento přístup umožňuje nezabývat se v jednotlivých požadavcích chybami. Jakmile se v jednom požadavku vyskytne chyba, další požadavky se nevykonají. Chybu tedy stačí ošetřit na jednom místě. Více informací o této strategii je dostupné na [50].

Video stream aktuální tisku tisku je implementovaný pomocí pravidelného stahování obrázku tisknutí. Aby aplikace nezatěžovala zbytečně síťové rozhraní, obrázek stahuje nejdříve pět vteřin po poslední aktualizaci.



Obrázek 3.5: Ukázka aplikace: Detail nepřipojené tiskárny

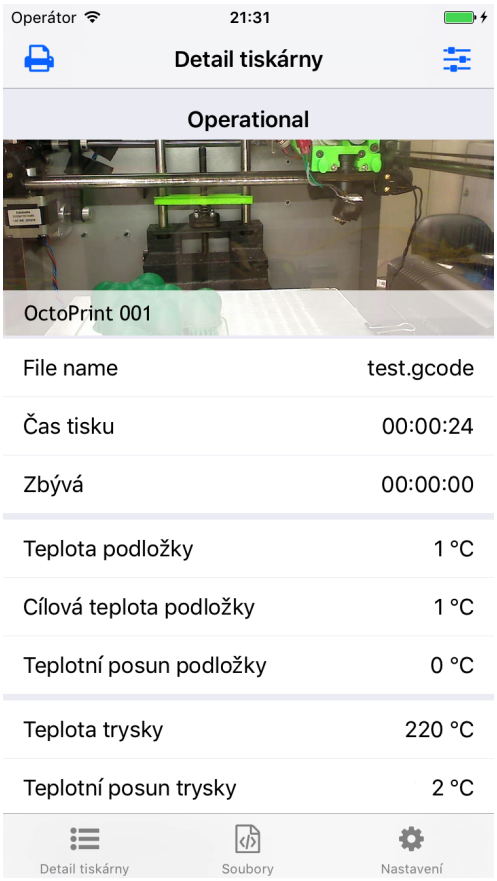
3.5.3 Využití síťových socketů

V sekci 3.5.2 se zmiňuji o zobrazení informací o aktuálním tisku. Ty se stahují při příchodu na obrazovku a následně se zobrazují uživateli. Tisk je ale kontinuální činnost a informace o něm je potřeba aktualizovat. Interval aktualizování dat by měl být co nejkratší, aby informace co nejpřesněji vystihovaly skutečnost.

Pomocí běžných síťových požadavků by ale mohlo dojít k větší zátěži sítě a rychlejšímu vybíjení mobilního zařízení. OctoPrint proto poskytuje rozhraní pro komunikaci skrz sockety.

Toto rozhraní umožňuje vytvořit jedno spojení s OctoPrint a pomocí něj dostávat průběžně nové informace. Na straně OctoPrint je rozhraní implementováno knihovnou `SockJS`.

Komunikace funguje tak, že klientské zařízení otevře nové spojení na adrese `sockjs/[id1]/[id2]/xhr_streaming`. Parametry `id1` a `id2` mohou být voleny náhodně.



Obrázek 3.6: Ukázka aplikace: Aktuální tisk

Po navázání komunikace OctoPrint průběžně odesílá informace klientskému zařízení. V těchto infomarcích se také vyskytuje JSON obejkt s aktuálním stavem tiskárny.

Ve své implementaci jsem chtěl využít knihovny `Moya`, kterou využívám ke všem ostatním síťovým požadavkům. V současné době ale nepodporuje tento typ komunikace. Musel jsem tedy využít zmíněné knihovny `Alamofire`, kterou `Moya` využívá ve své implementaci.

Reaktivní rozšíření

Abych dosáhl reaktivního přístupu k tomuto typu komunikace, vytvořil jsem pro `Alamofire` obalovou třídu `WebSocket`. Jejím jediným účelem je vytvořit síťové spojení s libovolným síťovým zařízením, které tento typ komunikace podporuje.

Pro vytvoření spojení stačí využít metodu `connect(url:_, method:_)`, která jako výsledek vrátí `Cold` signál. Tento signál je obecného typu `Data`

```
provider.request(.currentPrinterState).filterSuccess().mapJSON()
    .mapTo(object: PrinterState.self)
    .flatMap(.latest) { state in
        /* Dokončení prvního požadavku a vytvoření nového */
        self.stateProperty.value = state
        return self.provider.request(.currentJob).
    }
    .filterSuccess().mapJSON().mapTo(object: Job.self)
    .flatMap(.latest) { job in
        /* Dokončení druhého požadavku a vytvoření nového */
        self.jobProperty.value = job
        return self.provider.request(.currentToolState)
    }
    .filterSuccess().mapJSON()
    .mapDictionary(collectionOf: Tool.self)
    .flatMap(.latest) { tools in
        /* Dokončení čtvrtého požadavku a vytvoření nového */
        self.toolProperty.value = tools.first
        return self.provider.request(.currentBedState)
    }
    .filterSuccess().mapJSON().mapTo(object: Bed.self)
    .startWithResult { result in
        /* Zpracování odpovědi */
    }
```

Ukázka kódu 3.9: Řetězení požadavků v Moya

reprezentující poslední informace odeslané OctoPrintem.

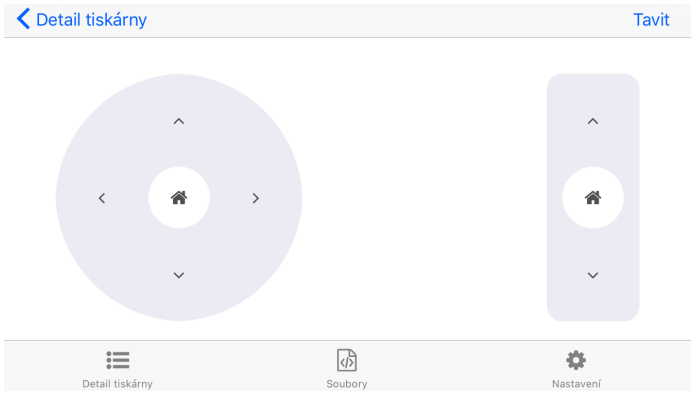
Pro tuto vrstvu jsem vytvořil další obalovou třídu nazvanou `OctoPrintPushEvents`, která se stará o zpracování generických dat. Jako veřejné rozhraní poskytuje konstruktor vyžadující URL tiskárny. Druhým veřejným rozhraním je metoda `temperatures()`, která se stará o zpracování dat z OctoPrint. Metoda vrací `Cold` signál s teplotami podložky a trysky.

Tato metoda ve své implementaci vytváří spojení pomocí třídy `WebSocket` a zpracová data z vráceného signálu. Tyto data se nejdříve převedou na text, očistí se od informací o komunikaci a rozdělí se na řádky. V každém řádku by se nyní měl vyskytovat platný JSON objekt.

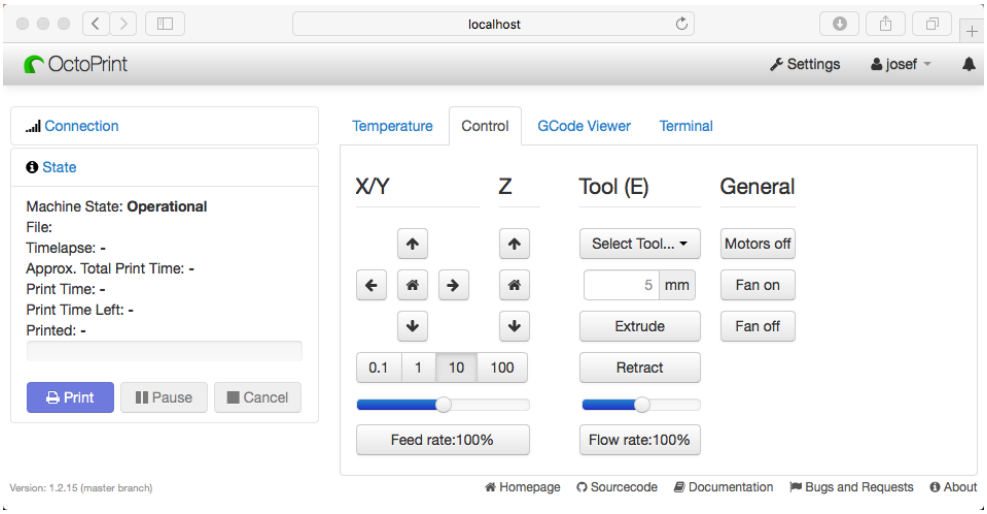
Tento objekt je následně zpracován na objekty `Bed` a `Tool`, které reprezentují teploty podložky a trysky.

Propojení s ViewModelem

K implementaci ve `ViewModelu` stačí pouze vytvořit instanci `OctoPrintPushEvents` s adresou tiskárny a zpracovat data signálu.



Obrázek 3.7: Ukázka aplikace: Ovládání tiskové hlavy



Obrázek 3.8: Ovládání tiskové hlavy v prostředí OctoPrint

Zpracování dat se provádí stejně jako u běžných požadavků knihovny `Moya`, nebylo tedy potřeba přidávat žádnou další logiku.

3.6 Ovládání tiskové hlavy

Při implementaci ovládání jsem se inspiroval rozhraním OctoPrint, které je znázorněno na obrázku 3.8. Pohyb hlavy je možný po všech osách. Aby bylo uživatelské rozhraní intuitivní, sjednotil jsem osy x a y do ovladače pro jednu ruku a osu z do ovladače pro druhou. To lze vidět na obrázku 3.7.

Rozhraní je tvořeno pomocí tlačítek, kde každá osa má jedno tlačítko pro pohyb v kladném a jedno pro pohyb v záporném směru. Dále každý z ovla-

dačů obsahuje tlačítko „Domů“, které zavede tiskovou hlavu na souřadnici 0 v patřičných osách.

3.6.1 Pohyb hlavy

Jelikož jsou prvky tvořeny tlačítky, je pohyb implementovaný pomocí požadavků pro každou interakci zvlášť. `Controller` pro každý stisk vytvoří vstup `ViewModelu`, ten následně vytváří požadavky na tiskárnu.

Požadavek pro posun hlavy je implementovaný genericky parametrizovaný množinou os a směrem. Aby UI nebylo zbytečně komplikované, rozhodl jsem uživateli neumožnit explicitně zvolit o jakou vzdálenost se má hlava posunout. Každý požadavek tedy posune hlavu po dané ose o 10 mm vpřed nebo vzad.

Každý uživatelský vstup je namapovaný na právě jeden vstup `ViewModelu`. Tím jsem dosáhl jednoduchého rozhraní `ViewModelu` a odstranil potřebu využití složitých řídících konstrukcí (switch konstrukce či řetězené podmínky). Implementace je znázorněna v ukázce 3.10.

```
// Odchycení uživatelské interakce v Controllertu
view.moveLeftButton.reactive.controlEvents(.touchUpInside)
    .observeValues { [weak self] _ in
        self?.viewModel.inputs.moveHeadLeft()
    }

/* ... */

// Převedení uživatelského vstupu na konkrétní požadavek
// ve ViewModelu
func moveHeadLeft() {
    movePrintHead(axis: .x, direction: .decrease)
}

// Generický požadavek ve ViewModelu
func movePrintHead(axis: JogAxis, direction: JogDirection) {
    printHeadRequest(.jogPrintHead(axis, direction))
}
```

Ukázka kódu 3.10: Implementace pohybu tiskové hlavy

3.6.2 Ovládání trysky

Z důvodu možnosti čištění trysky jsem přidal na tuto obrazovku tlačítko pro manuální tavení filamentu. Obdobně jako u pohybu tiskové hlavy ani zde jsem neumožnil uživateli zvolit množství. Při uživatelské interakci (stisku tlačítka „Tavit“) aplikace vždy nechá roztavit 5 mm.

Implementace tohoto požadavku se velmi blízce podobá požadavkům na pohyb tiskové hlavy.

3.7 Správa souborového systému

Druhou obrazovkou detailu tiskárny je správa souborů. Uživatel vidí seznam souborů a může je třídit.

Při příchodu na obrazovku se vyšle požadavek na OctoPrint pro stažení souborů. `ViewModel` opět dbá pouze na data uložená v lokální databázi. Jakmile tedy přijde odpověď na požadavek, data se uloží do databáze a informace se pomocí signálu propaguje do `Controlleru`.

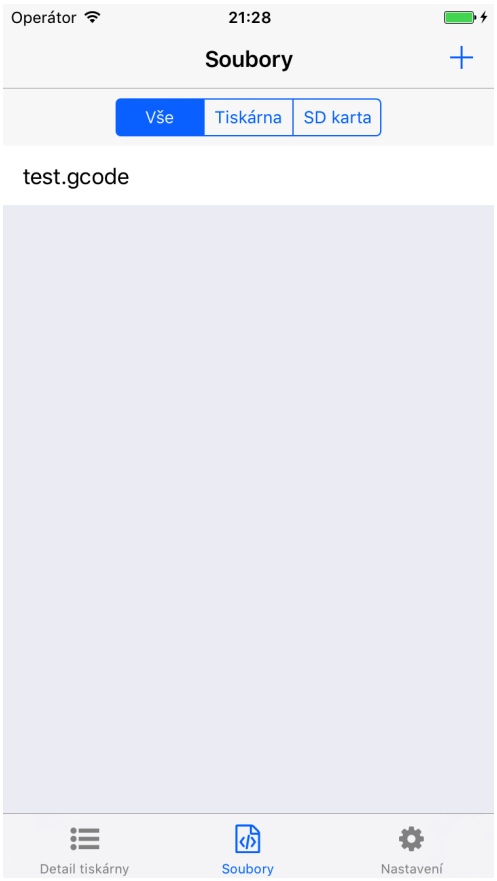
3.7.1 Filtrování

Ve `ViewModelu` jsem připravil tři filtry, ty představují „vše“, „soubory na tiskárně“ a „soubory na paměťové kartě“. Implemetace filtrů je vidět v ukázce 3.11. Výchozí filtr zobrazuje soubory dostupné jak v paměti tiskárny tak na paměťové kartě. Jakmile se rozhodne uživatel změnit filtr, vytvoří svou interakcí vstup do `ViewModelu` kde se jen změní aktivní filtr a zpět se upozorní `Controller`. Obrazovka seznamu souborů je vidět na obrázku 3.9.

```
1  struct Filters {
2      typealias Filter = NSPredicate
3
4      // Zobrazí veškeré soubory
5      static let all = NSPredicate(value: true)
6
7      // Zobrazí pouze soubory v tiskárně
8      static let local = NSPredicate(format: "_origin = %@",
9                                              Origin.local)
10
11     // Zobrazí pouze soubory na peměťové kartě
12     static let sdcard = NSPredicate(format: "_origin = %@",
13                                          Origin.sdcard)
14 }
```

Ukázka kódu 3.11: Filtry souborů

Tímto jsem zajistil, že nevznikne nekonzistence při aktualizaci souborů. Tedy vždy jsou zobrazeny právě ty soubory, které uživatel zvolil. Žádné nechybí ani nepřebývají ani v době, kdy se seznam znovu stáhne z OctoPrint.



Obrázek 3.9: Ukázka aplikace: Seznam souborů

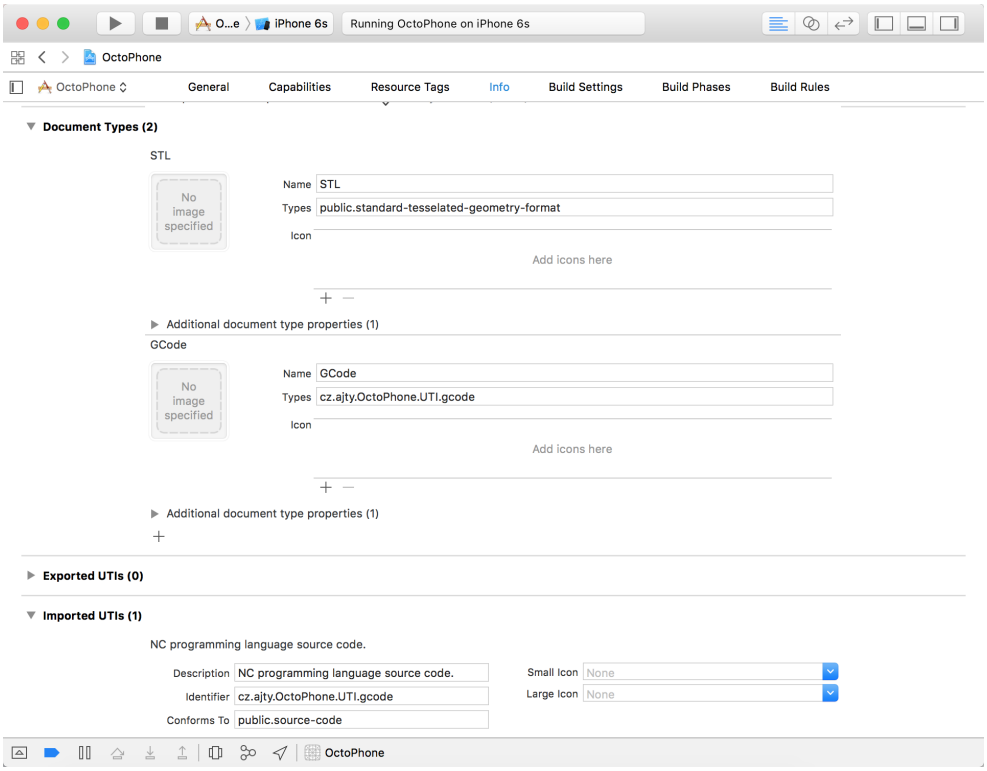
3.7.2 Nahrávání souborů

Nahrávání souborů k tisku je z mého pohledu důležitou funkcí této aplikace. Systém iOS je ale bohužel uzavřený a nedovoluje uživateli přistupovat k souborovému systému [51]. Umožňuje ale otevírat soubory z jiných aplikací. Tyto aplikace musí funkci poskytování souborů explicitně povolit.

Při implementaci této funkce stačilo v nastavení projektu zvolit jaký typ souborů aplikace umí otevřít. Systém se následně postará o zobrazení dialogu uživateli a provede ho výběrem souboru. Jakmile uživatel soubor vybere, systém ho předá aplikaci.

Po výběru se předá umístění souboru do `ViewModel` vrstvy, která pomocí knihovny `Moya` soubor do tiskárny nahraje.

Standardně má uživatel jako jediné dostupné uložiště iCloud implementovaný přímo v systému, používá-li ale některé z cloudových uložišť může soubor nahrát přímo z nich.



Obrázek 3.10: Nastavení projektu pro otevírání souborů

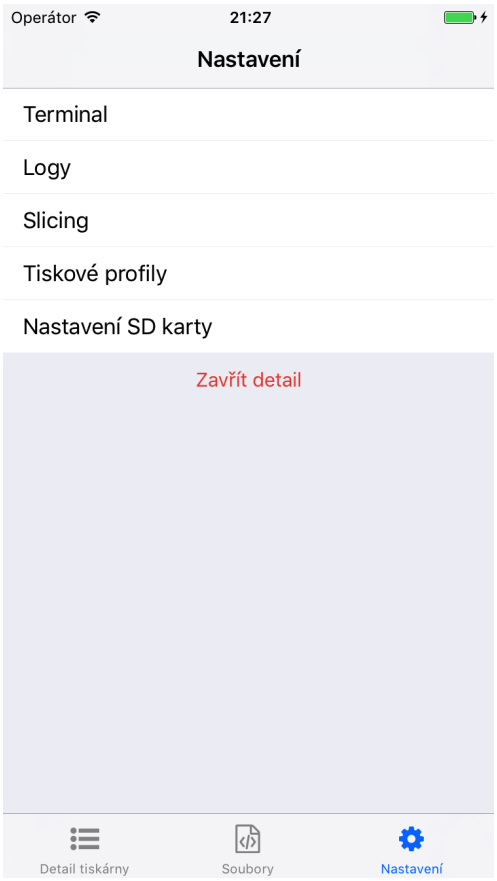
3.8 Terminálový emulátor

Abych uživateli umožnil vytvořit libovolný příkaz tiskárně, vytvořil jsem obrazovku připomínající terminálové rozhraní. Obrazovka se skládá z historie příkazů a vstupu pro nový příkaz.

3.8.1 Vytvoření příkazu

Jakmile uživatel napíše příkaz a stiskne tlačítko „Odeslat“, vytvoří `ViewModel` na základě vstupu novou položku v lokální databázi. Příkaz je v databázi vytvořen s příznakem „Zpracováván“. Na základě vložení `ViewModel` upozorní `Controller`, že v historii příkazů proběhla změna a `Controller` překreslí seznam.

V rámci dodržení architektury MVVM jsem pro každou položku seznamu vytvořil další `ViewModel`. `ViewModel` položky seznamu se stará o to, aby byl příkaz proveden. Ve chvíli vytvoření položky v seznamu `ViewModel` prozkoumá objekt příkazu. Má-li příznak „Zpracováván“, vytvoří požadavek na tiskárnu o provedení příkazu. Na základě odpovědi je nastaven nový příznak. V případě



Obrázek 3.11: Ukázka aplikace: Nastavení

úspěchu „Zpracován“, v opačném případě „Chyba“.

`ViewModel` o výsledku informuje položku seznamu (`View`). Pokud je výsledkem chyba, `View` se červeně podbarví aby uživateli naznačilo, že příkaz selhal.

3.9 Nastavení

Jako třetí obrazovku detailu tiskárny jsem zvolil nastavení. Tato obrazovka obsahuje funkcionality, u kterých předpokládám, že budou využívány s menší intenzitou. Funkce nastavení jsou rozděleny do více obrazovek, těm se samostatně věnuji v jednotlivých bodech níže.

Pod seznam funkcionalit jsem také umístil tlačítko pro zavření detailu tiskárny. Po stisknutí tlačítka se obrazovka zavře a zobrazí se opět seznam tiskáren.

Seznam funkcionalit je vidět na obrázku 3.11.

3.9.1 Správa log souborů

Obrazovku pro správu log souborů jsem implementoval opět pomocí tabulky. Při příchodu na obrazovku stáhne `ViewModel` seznam logů a uloží je do lokální databáze. Z důvodů urychlení stahování a minimalizace přenesených dat se nejprve stahuje pouze seznam souborů, obsah nikoli.

Pro stažení souboru musí uživatel otevřít novou obrazovku s detailem logu. V tuto chvíli se spustí stahování logového souboru, který je po dokončení stažení zobrazen uživateli.

Z této obrazovky je také možné soubor smazat.

3.9.2 Správa slicing profilů

Správa slicing profilů je rozdělena do dvou částí. Na první obrazovce se nachází seznam slicerů. Z tohoto seznamu může uživatel výběrem sliceru přejít na seznam slicovacích profilů. Seznam profilů odpovídá vybránému sliceru.

Dále je možné přejít na detail profilu, kde ho uživatel může upravit či smazat. Aplikace také umožňuje vytvořit profil nový.

3.9.3 Tiskové profily

Tiskové profily jsou implementovány formou seznamu. Po výběru konkrétního tiskového profilu je zobrazena nová obrazovka s detailem. Z detailu profilu je možné profil upravit nebo smazat.

Pro zjednodušení uživatelského rozhraní jsem uživatele při vytváření nového profilu omezil pouze na základní informace. Rozšířené nastavení profilu je nutné vytvořit ve webovém rozhraní OctoPrint.

3.9.4 Správa paměťové karty

Aplikace umožňuje kontrolovat stav paměťové karty. Na samostatné obrazovce lze zjisit, zda je karta momentálně připojena či nikoli.

Obrazovka dále obsahuje tři tlačítka. Jedno pro připojení paměťové karty, druhé pro aktualizaci načtených dat z karty a třetí pro odpojení. Tlačítka se zobrazují podle aktuálního stavu karty. Tedy např. tlačítko „Odpojit“ není zobrazeno ve chvíli, kdy karta není připojena.

Kapitola 4

Testování

V kapitole o testování se podrobněji zabývám způsoby a postupy naznačenými v kapitolách 2.2.2 a 2.3.3 o architektuře MVVM a reaktivní programování.

Testováním software se rozumí postupy a procesy, pomocí kterých lze měřit, zda testovaný software (či jeho části) splňuje požadované nároky či nikoliv. Opakovaným aplikováním těchto postupů lze v softwaru nalézt chyby, nedostatky nebo chybějící vlastnosti oproti dodané specifikaci. Výsledky testování následně vypovídají o kvalitě softwaru a o míře splnění specifikace. [52]

Jako techniku testování jsem zvolil testy chování (z anglického `behavior tests`). Ty ověřují, zda aplikace na sadu vstupů vrací odpovídající výstupy.

4.1 Testy chování

Chování modulu aplikace lze otestovat tak, že se zkoumá plnění jeho závazků. Testované objekty mají definované rozhraní a závisloti, tím mají také deklarovány i závazky. Závazky určují, jakým způsobem by měl objekt působit na zbylé části aplikace a jaké parametry a funkce má. Z parametrů a funkcí lze odvodit jakým způsobem se má objekt chovat.

4.1.1 Testování komponent

Pomocí těchto testů lze ověřovat funkčnost libovolné komponenty aplikace. Složitost a rozsah těchto testů se následně odvíjí od počtu závazků komponenty. Testování probíhá tak, že vlastnosti objektu jsou pomocí rozhraní měněny. Přitom se sleduje, jestli se objekt na základě změn chová podle očekávání. [53]

4.1.2 DSL

Testy popisuje Domain-Specific language (DSL). To je jazyk který pomocí kombinace klíčových slov a textového popisu definuje jak se komponenta má

v určitou chvíli chovat. Více o DSL lze zjistit na [54].

V běžném testovacím rozhraní prostředí Xcode tento jazyk nenabízí. Rozhodl jsem pro to využí dvou knihoven. Pomocí lnihovny `Quick` jsem mohl DSL v projektu využít. Očekávané hodnoty jsem ověřoval knihovnou `Nimble`.

4.1.3 Testování ViewModelu

Tento princip využívám ve své implementaci pro testování `Modelu` a `ViewModelu`. Díky dostatečnému rozsahu testů logiky aplikace lze usuzovat, že v produkčním nasazení se vyskytne jen nepatrné množství chyb. `View` vrstvu následně není samostatně nutné testovat, protože z testů `ViewModelu` je téměř jisté, že data jsou správně připravena k zobrazení.

U `ViewModelu` jsem podle zvoleného scénáře navrhl DSL. Tímto jazykem jsem definoval jak očekávám, že s objektem bude pracovat. Pomocí `Nimble` knihovny jsem dále ověřoval, jestli na vytvořené vstupy vytváří `ViewModel` očekávané výstupy jak je vidět v ukázce 4.1.

Během implementace jsem vytvořil bez mála sto testů, díky kterým jsem odchytil velké množství chyb. Tyto chyby jsem často do aplikace zanesl v momentě, kdy jsem k již hotové obrazovce implementoval novou funkcionalitu.

4.1.4 Vývoj řízený testováním chování

S pojmem testování chování je velmi blízce spjat *vývoj řízený testováním chování* (z anglického Behavior Driven Development, zkráceně BDD). V tomto přístupu k vývoji se před konkrétní implementací nejdříve nadefinuje, jakým způsobem se mají testované komponenty chovat. Následně se sestavují testy, které vyžadují implementování vyžadovaného chování. Tím má objekt definované, jaké rozhraní musí implementovat a jaké závislosti bude vyžadovat.

Průběh vývoje probíhá cyklicky. Nejdříve se nadefinují testy některé funkcionality, následně se implementuje jen tolik kódu, kolik je nutné aby testy byly úspěšné. Tento postup se opakuje dokud nejsou implementovány veškeré funkce vyžadované po objektu. Více informací o tomto přístupu je dostupných na [53].

Tento přístup jsem využil na část `ViewModelů`. Vzhledem k rozsahu implementace jsem ale většinu testů napsal až po kompletní implementaci `ViewModelu`.

4.2 Průběžná integrace

Průběžná integrace (z anglického *Continous integration*, zkráceně `CI`) je praktika vývoje software, při které členové týmu integrují svou práci mnohdy až několikrát denně. Každá integrace je automaticky ověřena kompilací na buildovacím serveru a spuštěním testů. Tato technika si klade za cíl odhalid integrační chyby aplikace co nejdříve je to možné. [55] To má za následek snížení

```swift
import Nimble
import Quick

class PrinterListViewModelTests: QuickSpec {
    var onAddPrinterButtonTapped: (() -> Void)?

    override func spec() {
        // Objekt, který bude testován
        var subject: PrinterListViewModel!
        // Vytvoření dat před každým testem
        beforeEach { }
        // Odstranění dat po konci testu
        afterEach { }

        // DSL testu
        it("notify delegate only when add button is tapped") {
            // Otestování hodnoty pomocí Nimble
            expect(addPrinterTapped) == false
            subject.inputs.addPrinterButtonTapped()
            expect(addPrinterTapped) == true
        }

        it("update printers count on change") { }
        it("supply correct network provider") { }
        it("provides correct view model for cell") { }
    }
}
```

Ukázka kódu 4.1: Použití DSL pro definici testu

časové náročnosti implemetace nových funkcí bez rizika narušení funkcionalit původní aplikace.

Nové funckionality se běžně skládají z nového kódu, upraveného produkčního kódu a upravených testů. V momentě kdy vývojář označí funkcionalitu za kompletní, odešle své změny na vzdálený buildovací server, kde se aplikace zkompiluje a otestuje. Integrace je následně provedena jen v případě, že celý proces proběhl bez chyby. Jak je zjevné, při této technice je zásadní vysoké pokrytí aplikace testy.

Průběžné integrování je často velmi blízce vázáno na správu zdrojových kódů. Některé systémy jako GitHub či GitLab je dokonce možné nastavit tak, aby by změny byly přidány do hlavní větve až ve chvíli, kdy build projde bez komplikací. [56]

Pro průběžnou integraci je na trhu dostupných mnoho řešení. Rozhodoval jsem se mezi použitím komerčních řešení Travis CI a Circle CI. První zmíněná

možnost se jeví v open-source komunitě velmi populární a dalo by se říct, že je pro `CI` téměř synonymem. Podle nezávislého průzkumu z roku 2016 využívá Travis CI téměř pětinásobek uživatelů než ostatních poskytovatelů dohromady. [57]

Druhé řešení jsem chtěl vyzkoušet kvůli rostoucí popularitě. [58] Bohužel je kompilace na operačním systému macOS pro open-source projekty možná až po individuálním schválení. [59] Z tohoto důvodu jsem nakonec zvolil první řešení, kde je kompilace na systému macOS pro studenty zdarma.

4.2.1 Statická analýza kódu

Spolu s roustoucím týmem a rostoucím zdrojovým kódem se často zavádí směrnice programování. Tyto směrnice udávají jakým způsobem je kód formátovaný. Při dodržování těchto směrnic se kód stává konzistentním a dobře čitelným, to má za následek zrychlení vývoje a zvyšuje udržitelnost kódu. Aby nebylo nutné analyzovat kód ručně, existují nástroje které analýzu automatizují.

Pro zajištění konzistence jsem použil nástroj `SwiftLint` od společnosti `Realm`. `SwiftLint` je nástroj obsahující sadu pravidel pro statickou analýzu kódu. Mimo jiné kontroluje správné zarovnání kódu, délky řádků či názvy proměnných. V době psaní práce obsahuje tento nástroj dohromady téměř osmdesát pravidel. Ve své implemetaci jsem se rozhodl vynechat dvě pravidla, které jsem nepovažoval za stěžejní a jejich dodržování by z mého pohledu vedlo ke snížení čitelnosti. Kromě těchto dvou pravidel jsem využil i možnost vypnout pravidla pro určité části kódu. Nejčastěji se jednalo o pravidla na kontrolu délky metod.

Tuto analýzu zajišťuje buildovací server před začátkem kompilace. Pokud nejsou v kódu závažné porušení směrnice, pokračuje se ke kompilaci. V opačném případě nastane chyba a je nutné kód opravit a integraci pustit znovu.

4.3 Průběžné doručování

Průběžné doručování (z anglického *Continuous delivery*, zkráceně `CD`) je způsob, kterým lze nasadit nové funkcionality, změny konfigurace či opravy chyb do produkčního prostředí. Cílem této praktiky je automatizovaně aplikovat opakované postupy potřebné k vydání aplikace. Tím lze minimalizovat množství chyb, které by vývojář jinak mohl při vydávání způsobit. Vývojáři stačí v tomto případě pouze doručování spustit, server zařídí aby aplikace byla správně nasazena.

Kromě minimalize výskytu chyb tato technika snižuje čas vývoje. Díky automatizovanému vydávání mají vývojáři více času na samotný vývoj. Tím lze zaručit vyšší kvalitu kódu, nižší náklady na vývoj a také častější aktualizace software. [60]

Přestože pro svou práci nemám reálné produkční prostředí, rozhodl jsem se CD použít pro distribuci testovací verze aplikace. Díky tomu jsem mohl jednotlivé verze vydávat v průběhu vývoje velmi rychle a ověřit funkčnost aplikace na skutečných zařízeních. Vydání verze jsem nastavil na každé nahrání zdrojových kódů do repozitáře. Po statické analýze a kompilaci byla aplikace nahrána do prostředí Fabric, odkud si ji mohli registrovaní testeři stáhnout.

4.4 Lokální testování

Protože aplikace komunikuje výhradně s tiskárnou, musí být tiskárna dostupná i během vývoje. Abych nemusel tiskárnu pořizovat, využil jsem možnost vytvořit virtuální tiskárnu.

S využitím technologie Docker jsem vytvořil kontejner s nainstalovanou aplikací OctoPrint. Pomocí konfiguračního souboru jsem povolil vytvoření virtuální tiskárny.

OctoPrint jsem takto mohl mít spuštěný lokálně na počítači. Po připojení OctoPrint k virtuálnímu portu tiskárny jsem mohl vyvíjet aplikaci i bez nutnosti vlastnit tiskárnu.

V ukázce 4.2 demonstruji spuštění kontejneru s virtuální tiskárnou z příkazové řádky svého počítače.

```
1 docker run -p"3200:5000" josefdolezal/virtuprint-docker
```

Ukázka kódu 4.2: Spuštění virtuální tiskárny pomocí Docker

Závěr

Cílem práce bylo vypracování mobilní aplikace pro platformu iOS umožňující nastavovat 3D tiskárny a tisknout z nich.

Výsledná aplikace umožňuje zobrazit přehled aktuálního tisku, správu souborů a nastavení 3D tiskárny. Pomocí aplikace je také možné sledovat video stream z webové kamery připojené k tiskárně. Uživatel také může vybrat soubor ve svém zařízení, nahrát ho do tiskárny a následně vytisknout.

V aplikaci lze komunikovat s libovolnou tiskárnou dostupnou na místní síti, která je aplikací automaticky nalezena. Uživatel může velmi jednoduše takto tiskárnu přidat a obsloužit celý tisk.

Během implementace jsem se snažil většinu pozornosti věnovat architektuře aplikace. Mnoho částí jsem proto v průběhu implementace několikrát přepsal.

Velkou výzvou bylo implementovat aplikaci reaktivně. S reaktivním programování jsem doposud neměl žádné zkušenosti. Nyní jsem ale rád, že jsem se takto rozhodl. Využití reaktivního přístupu mi ušetřilo mnoho práce a velmi pravděpodobně i mnoho chyb, které bych jinak standardní cestou do aplikace zanesl.

Jsem přesvědčený, že díky využití zvolené architektury jsem také zvýšil testovatelnost aplikace. Přesto, že jsem testy aplikace nikdy dříve nepsal, nebylo složité je s toutou architekturou vytvořit.

Výhled do budoucna

V budoucnu je možné do vývoje aplikace přizvat programátory pohybující se pravidelně v prostředí 3D tisku a rozšířit ji tak v komunitě. Z tohoto důvodu jsou zdrojové kódy práce volně dostupné jako open source. Aplikaci je díky tomu možné rozšířit o nové funkcionality v případě aktualizace API OctoPrint.

V současné době ale neplánuji aplikaci distrubuovat běžným uživatelům obchodem App Store.

Literatura

[1] Hollemans, M.: How to Update Your App for iOS 7. *Ray Wenderlich [online]*, říjen 2013, [cit. 2017-5-4]. Dostupné z: https://www.raywenderlich.com/49316/how-to-update-your-app-for-ios-7

[2] Trendy výrazu 3D print. *Google Trends [online]*, duben 2017, [cit. 2017-4-11]. Dostupné z: https://trends.google.cz/trends/explore?q=3d%20print

[3] Titcomb, J.: Mobile web usage overtakes desktop for first time. *The Telegraph [online]*, listopad 2016, [cit. 2017-4-11]. Dostupné z: http://www.telegraph.co.uk/technology/2016/11/01/mobile-web-usage-overtakes-desktop-for-first-time/

[4] Fused Deposition Modeling. 3D tisk. *3D tisk*, [cit. 2016-12-11]. Dostupné z: http://www.3d-tisk.cz/fused-deposition-modeling/

[5] Rapid Prototyping: An Overview. *eFunda [online]*, červen 2016, [cit. 2017-4-11]. Dostupné z: http://www.efunda.com/processes/rapid_prototyping/intro.cfm

[6] Slicing. *OctoPrint [online]*, září 2013, [cit. 2017-5-11]. Dostupné z: http://docs.octoprint.org/en/master/api/slicing.html

[7] STL. *3D-tisk.cz [online]*, 2015, [cit. 2017-5-12]. Dostupné z: http://www.3d-tisk.cz/stl/

[8] G-code - RepRapWiki. *RepRapWiki [online]*, květen 2017, [cit. 2017-5-12]. Dostupné z: http://reprap.org/wiki/G-code

[9] Carbonnelle, P.: PYPL PopularitY of Programming Language. *PYPL [online]*, květen 2016, [cit. 2017-5-12]. Dostupné z: http://pypl.github.io/PYPL.html

[10] Onorato, M.: MVC Recommendations for Cocoa Touch. *GameChanger [online]*, září 2013, [cit. 2017-5-12]. Dostupné z: http://tech.gc.com/mvc-recommendations-for-cocoa-touch/

[11] Khanlou, S.: Massive View Controller. *Sorush Khanlou [online]*, prosinec 2015, [cit. 2017-5-12]. Dostupné z: http://khanlou.com/2015/12/massive-view-controller/

[12] Rasic, S.: From MVC to MVVM in Swift. *Srdan Rasic [online]*, prosinec 2014, [cit. 2017-5-12]. Dostupné z: http://rasic.info/from-mvc-to-mvvm-in-swift/

[13] Eberhardt, C.: EXPLORING KVO ALTERNATIVES WITH SWIFT. *Scott Logic [online]*, únor 2015, [cit. 2017-5-12]. Dostupné z: http://blog.scottlogic.com/2015/02/11/swift-kvo-alternatives.html

[14] What's new in iOS. *Apple Guides and Sample Code [online]*, březen 2017, [cit. 2017-4-14]. Dostupné z: https://developer.apple.com/library/content/releasenotes/General/WhatsNewIniOS/Articles/iOS9.html

[15] Concurrency Programming Guide. *Apple Guides and Sample Code [online]*, prosinec 2012, [cit. 2017-4-14]. Dostupné z: https://developer.apple.com/library/content/releasenotes/General/WhatsNewIniOS/Articles/iOS9.html

[16] Prioritize Work with Quality of Service Classes. *Apple Guides and Sample Code [online]*, září 2016, [cit. 2017-4-14]. Dostupné z: https://developer.apple.com/library/content/releasenotes/General/WhatsNewIniOS/Articles/iOS9.html

[17] Jacobs, B.: Choosing Between NSOperation and Grand Central Dispatch. *Cocoacasts [online]*, červenec 2016, [cit. 2017-4-17]. Dostupné z: https://cocoacasts.com/choosing-between-nsoperation-and-grand-central-dispatch/

[18] Operation Queues. *Apple Guides and Sample Code [online]*, prosinec 2012, [cit. 2017-4-17]. Dostupné z: https://developer.apple.com/library/content/documentation/General/Conceptual/ConcurrencyProgrammingGuide/OperationObjects/OperationObjects.html

[19] Lilja, P.: An Introduction to Reactive programming. *ONE AGENCY [online]*, leden 2017, [cit. 2017-5-11]. Dostupné z: https://www.oneagency.se/reactive-programming-with-rxjava/

[20] Laks, S.: Concurrency, part 3: Promises. *SLaks.Blog [online]*, leden 2015, [cit. 2017-5-11]. Dostupné z: http://blog.slaks.net/2015-01-05/introducing-promises/

[21] Teixeira, P.: The Callback Pattern. *YLD Software engineering blog [online]*, říjen 2015, [cit. 2017-5-11]. Dostupné z: https://blog.yld.io/2015/10/19/the-callback-pattern/

[22] ReactiveSwift: Streams of values over time. *GitHub [online]*, duben 2017, [cit. 2017-5-7]. Dostupné z: https://github.com/ReactiveCocoa/ReactiveCocoa

[23] Alamofire: Elegant HTTP Networking in Swift. *GitHub [online]*, květen 2017, [cit. 2017-5-3]. Dostupné z: https://github.com/Alamofire/Alamofire

[24] Moya: Network abstraction layer written in Swift. *GitHub [online]*, květen 2017, [cit. 2017-5-3]. Dostupné z: https://github.com/Moya/Moya

[25] Eggert, D.: Core Data Overview. *objc.io [online]*, září 2013, [cit. 2017-5-11]. Dostupné z: https://www.objc.io/issues/4-core-data/core-data-overview/

[26] Realm. *GitHub [online]*, květen 2017, [cit. 2017-5-3]. Dostupné z: https://github.com/realm

[27] Realm Mobile Platform Overview. *Realm [online]*, květen 2017, [cit. 2017-5-3]. Dostupné z: https://realm.io/docs/get-started/overview/

[28] Swift 2.7.0. *Realm [online]*, květen 2017, [cit. 2017-5-3]. Dostupné z: https://realm.io/docs/swift/latest/#writes

[29] Assets Catalog Format Reference: Format Overview. *Apple Guides and Sample Code [online]*, září 2016, [cit. 2017-5-4]. Dostupné z: https://developer.apple.com/library/content/documentation/Xcode/Reference/xcode_ref-Asset_Catalog_Format/

[30] Remero, I.: Image cache support. *Github [online]*, červenec 2016, [cit. 2017-5-4]. Dostupné z: https://github.com/dzenbot/Iconic/issues/65

[31] Jagtap, S.: Swift Dependency Management. *XCBlog [online]*, leden 2017, [cit. 2017-5-4]. Dostupné z: http://shashikantjagtap.net/swift-dependency-management-ios/

[32] CocoaPods.org. *CocoaPods [online]*, leden 2017, [cit. 2017-5-4]. Dostupné z: https://cocoapods.org/about

[33] Carthage: A simple, decentralized dependency manager for Cocoa. *GitHub [online]*, květen 2017, [cit. 2017-5-4]. Dostupné z: https://github.com/Carthage/Carthage

[34] Welcome to OctoPrint's documentation! *OctoPrint [online]*, červenec 2015, [cit. 2017-5-6]. Dostupné z: http://docs.octoprint.org/en/1.2.15/api/general.html

[35] File System Basics. *Apple Guides and Sample Code [online]*, září 2016, [cit. 2017-5-6]. Dostupné z: https://developer.apple.com/library/content/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html

[36] File Operations. *OctoPrint [online]*, červenec 2016, [cit. 2017-5-6]. Dostupné z: http://docs.octoprint.org/en/1.2.15/api/fileops.html#issue-a-file-command

[37] Printer profile operations. *OctoPrint [online]*, březen 2015, [cit. 2017-5-7]. Dostupné z: http://docs.octoprint.org/en/1.2.15/api/printerprofiles.html

[38] Job operations. *OctoPrint [online]*, červen 2016, [cit. 2017-5-6]. Dostupné z: http://docs.octoprint.org/en/1.2.15/api/job.html#issue-a-job-command

[39] Slicing. *OctoPrint [online]*, červenec 2016, [cit. 2017-5-7]. Dostupné z: http://docs.octoprint.org/en/1.2.15/api/slicing.html

[40] Designing for iOS 10. *DesignCode [online]*, 2017, [cit. 2017-5-12]. Dostupné z: https://designcode.io/iosdesign-guidelines

[41] Fayzrakhmanov, A.: Single Responsibility Principle: A Recipe for Great Code. *Toptal [online]*, březen 2016, [cit. 2017-5-7]. Dostupné z: https://www.toptal.com/software/single-responsibility-principle

[42] Kickstarter for iOS. *GitHub [online]*, květen 2017, [cit. 2017-5-11]. Dostupné z: https://github.com/kickstarter/ios-oss

[43] ReactiveCocoa: Streams of values over time. *GitHub [online]*, květen 2017, [cit. 2017-5-7]. Dostupné z: https://github.com/ReactiveCocoa/ReactiveSwift

[44] FontAwesome, the iconi font and CSS toolkit. *FontAwesome [online]*, duben 2017, [cit. 2017-5-7]. Dostupné z: http://fontawesome.io

[45] Doležal, J.: Iconic, Add support for brew (Issue #65). *GitHub [online]*, březen 2017, [cit. 2017-5-7]. Dostupné z: https://github.com/dzenbot/Iconic/issues/65

[46] Travis CI - octoprint-ios-client builds. *Travis CI [online]*, květen 2017, [cit. 2017-5-8]. Dostupné z: https://travis-ci.com/3DprintFIT/octoprint-ios-client/builds

[47] Bonjour for Developers. *Apple [online]*, březen 2010, [cit. 2017-5-10]. Dostupné z: https://developer.apple.com/bonjour/

[48] API Reference | RefereNetServiceBrowser. *Apple Developer Documentation [online]*, 2017, [cit. 2017-5-10]. Dostupné z: https://developer.apple.com/reference/foundation/netservicebrowser

[49] Singletons in Swift. *that thing in swift [online]*, březen 2016, [cit. 2017-5-10]. Dostupné z: https://github.com/nickoneill/thatthinginswift/blob/master/content/singletons.md

[50] Railway Oriented Programming. *F# for fun and profit [online]*, květen 2013, [cit. 2017-5-10]. Dostupné z: https://fsharpforfunandprofit.com/rop/

[51] Wang, J.: How to Access iPhone or iPad File System. *iMobile [online]*, červen 2015, [cit. 2017-5-10]. Dostupné z: https://www.imobie.com/support/access-iphone-ipad-file-system.htm

[52] Software testing tutorial. *Tutorials point [online]*, červenec 2015, [cit. 2017-3-28]. Dostupné z: https://www.tutorialspoint.com/software_testing/

[53] Dudek, P.: Behavior-Driven Development. *objc [online]*, srpen 2014, [cit. 2017-3-29]. Dostupné z: https://www.objc.io/issues/15-testing/behavior-driven-development/

[54] Kainulainen, P.: Replace Assertions with a DSL. *Petri Kainulainen [online]*, červen 2014, [cit. 2017-5-11]. Dostupné z: https://www.petrikainulainen.net/programming/testing/writing-clean-tests-replace-assertions-with-a-domain-specific-language/

[55] Cano, P.: 7 reasons why you should be using Continuous Integration. *GitLab [online]*, únor 2015, [cit. 2017-5-12]. Dostupné z: https://about.gitlab.com/2015/02/03/7-reasons-why-you-should-be-using-ci/

[56] Building Pull Requests. *Travis CI [online]*, listopad 2016, [cit. 2017-4-13]. Dostupné z: https://docs.travis-ci.com/user/pull-requests/

[57] Hilton, M.: Open-source CI Usage. *Oregon State University [online]*, 2016, [cit. 2017-4-14]. Dostupné z: http://cope.eecs.oregonstate.edu/CISurvey/

[58] Gorbatchev, A.: Continuous Integration in the Cloud: Comparing Travis, Circle and Codeship. *StrongLoop [online]*, leden 2015, [cit. 2017-4-14]. Dostupné z: `https://strongloop.com/strongblog/node-js-travis-circle-codeship-compare/`

[59] Pricing. *CircleCI [online]*, 2017, [cit. 2017-4-14]. Dostupné z: `https://circleci.com/pricing/#faq-section-os-x`

[60] What is Continuous Delivery? *Continuous Delivery [online]*, 2016, [cit. 2017-4-14]. Dostupné z: `https://continuousdelivery.com`

Příloha **A**

Seznam použitých zkratek

API Application Programming Interface. 1, 3, 4, 14, 20, 22, 25, 29–31, 35, 37, 71

BDD Behavior Driven Development. 1, 66

CRUD Create, Read, Update, Delete. 1, 25, 33

DSL Domain-Specific language. xvi, 1, 65–67

FDM Fused Deposition Modeling. 1, 3, 4

FRP Functional Reactive Programming. 1, 16

GCD Grand Central Dispatch. 1, 12, 14–16

GPS Global Positioning System. 1, 12

HTTP Hypertext Transfer Protocol. 1, 33, 34

IDE Integrated Development Environment. 1

JSON JavaScript Object Notation. 1, 19, 20, 55, 56

KVO Key-Value Observing. 1, 11

mDNS Multicast Domain Name Service. 1, 48

MVC Model-View-Controller. xv, 1, 5–9, 11, 39

MVVM Model-View-ViewModel. xiii, xv, 1, 8–11, 23–25, 39, 41, 46, 61, 65

ORM Object Relational Mapping. 1, 24, 25

REST Representational State Transfer. 1, 18, 28, 29

SDK Software Development Kit. 1, 12

SQL Structured Query Language. 1

URL Uniform Resource Locator. 1, 20, 21, 47, 50, 56

XML eXtensible Markup Language. 1, 19

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF