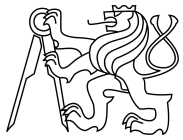
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Návrh konceptu a vývoj prototypu univerzálního uživatelského rozhraní
Student:	Bc. Rostislav Novák
Vedoucí:	Ing. Robert Pergl, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Cílem práce je navrhnout univerzální uživatelské rozhraní nezávislé na platformě a implementaci. Po rozboru a zhodnocení současných řešení v oblasti uživatelského rozhraní (UI) a univerzálního uživatelského rozhraní, vytvořte koncept univerzálního uživatelského rozhraní (UUI). Pro tento koncept vytvořte ontologii, doménově specifický jazyk (DSL) a koncept architektury klient-server. Následně demonstrujte ukázkovou implementaci prototypu na dvou vybraných platformách. Závěrem zhodnoťte možnosti UUI, jeho výhody a omezení a další možný vývoj do budoucna.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 5. ledna 2016

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

Diplomová práce

Návrh konceptu a vývoj prototypu univerzálního uživatelského rozhraní

Bc. Rostislav Novák

Vedoucí práce: Robert Pergl

10. ledna 2017

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 10. ledna 2017 .....

Odkaz na tuto práci

Novák, Rostislav. *Návrh konceptu a vývoj prototypu univerzálního uživatelského rozhraní.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Každá dnešní aplikace obsahuje uživatelské rozhraní. Také je dnes často oblíbené používat tenké klienty a data mít centrálně uložená serveru. Tato práce se zabývá možností vytvoření univerzálního uživatelského rozhraní, které by bylo platformě a i implementačně nezávislé. Dále popisuje pro vytvořené UUI architekturu klient-server, kde server poskytuje klientu uživatelské rozhraní a klient přijímá uživatelské rozhraní, které následně generuje pro danou platformu s jejími možnostmi.

Klíčová slova Reaktivní programování, funkcionální reaktivní programování, event-driven programming, uživatelské rozhraní, grafické uživatelské rozhraní, univerzální uživatelské rozhraní, UI DSL, UI XML, generování UI

Abstract

Every today's application contains the user interface. Today is also favorite use thin clients and have the data stored on a central server. This work deals with the possibility of creating universal user interface, which will be platforms and implementation independent. Also this work describes client-server

architecture, for this created UUI, where the server provides for client user interface and the client accepts user interface, which then generates to concrete user interface for its platform.

Keywords Reactive programming, funcional reactive programming, event-driven programming, user interface, graphic user interface, universal user interface, UI DSL, UI XML, generation of UI

Obsah

Seznam obrázků

Úvod

Diplomová práce se zabývá analýzou existujících řešení univerzálního uživatelského rozhraní a možností generování uživatelského rozhraní. Práce se pak zaměřuje na vytvořeni konceptu univerzální uživatelské rozhraní, který následně ověřuje implementací prototypu. Univerzální uživatelské rozhraní je zaměřeno na možnost použití na jakékoliv platformě, neboť v dnešní době existuje mnoho platforem a každá platforma obsahuje vlastní definici pro popis uživatelského rozhraní. Další trendem dnes je používání tenkých klientů, proto se zároveň s návrhem konceptu budeme zabývat vytvořením architektury klient-server s použitím UUI a reaktivního programování.

Motivace

V dnešní době každý systém obsahuje uživatelské rozhraní, přes které uživatel komunikuje se systémem. Uživatel očekává od systému rychlou zpětnou vazbu, případně informaci v jakém stavu se aplikace nachází. Dříve webové aplikace obsahovaly jeden velký formulář, který se po vyplnění odeslal na server. Dnes se již odesílají jednotlivé hodnoty komponent z formuláře. Tento trend míří i na klasické aplikace. To dalo za vznik nových přístupů, které poskytují snadnou implementaci a hlavně rychlou odpověď uživateli.

Dalším požadavkem na systém od uživatelů je, že chtějí mít uložené data na jednom místě (většinou cloudu) a ze všech zařízení mít přístup k těmto datům. Tento je často používán u webových aplikací, který disponují responzivním designem. Otázkou je, jestli je možné tento přístup použít i na klasické aplikace, kde by aplikace byla schopna částečné migrace? Pokud bychom byli schopni docílit migrace uživatelského rozhraní na jiná zařízení, tak bychom měli data uložena na jedno místě, kde by byla uložená i celá logika aplikace.

Také v dnešní době se začínají tvořit malé počítače, které nedisponují grafickým výstup nebo je tak malý, že nelze používat grafické komponenty, ale například jenom text. Další otzákou je, když bychom byli schopni migrovat

uživatelské rozhraní aplikace, bylo by možné toto rozhraní přizpůsobit danému rozhraní?

Odpovědi na tyto dvě otázky vyplynou v průběhu této diplomové práce.

Kapitola 1

Cíle a metodika

Cíle této práce lze rozdělit do čtyř celků. Prvním z nich je seznámení se s uživatelským rozhraní(UI) a softwarových technik při vývoji UI. Druhý z dílčích celků je zpracování aktuálních řešení univerzálního uživatelského rozhraní (UUI), případně grafického uživatelského rozhraní. Z těchto dvou částí by měla být zřejmá situace uživatelských rozhraní. Třetím celkem je návrh konceptu UUI, který by měl obsahovat ontologii UI, DSL a architekturu UUI. Posledním celkem je ověření konceptu pomocí implementace UUI použitím na ukázkové aplikaci.

Cílem práce naopak není výkonné a ani optimalizované řešení univerzálního uživatelského rozhraní, ale pouze koncept, který bude funkční a možný dále rozšiřovat a optimalizovat.

1.1 Metodika práce

V první části práce získáme dostatečný teoretický základ a přehled o uživatelských rozhraní a aktuálních řešení univerzálního uživatelského rozhraní. Tento základ použijeme při návrhu univerzálního uživatelského rozhraní. Metodou pro vytvoření konceptu UUI použijeme abstrahování stejných prvků z grafického uživatelského rozhraní. Abstrahovaná množina prvků musí být minimální množina, která bude stačit k vytvoření základních a i dalších komponent UI. Následně si ukážeme proč stačí vytvořit minimální koncept pro základní UI komponenty a vytvořený minimální koncept znázorníme modelem ontologie pomocí notace UML[1] a také pro něj vytvoříme DSL jazyk, který se bude používát pro popis UI ve formátu XML[2]. Dále vytvoříme architekturu klient-server pro naše UUI. Tato architektura poslouží jako základní kámen pro migraci UI na jiná zařízení. V poslední části náš koncept ověříme pomocí implementace na dvou vybraných platformách.

1.2 Struktura práce

Práce je rozdělena do čtyř kapitol. Kapitola 2 obsahuje teoretický základ pro tuto práci (definici UI, softwarové techniky při vývoji UI a techniky při vývoji GUI). Kapitola 3 obsahuje aktuální řešení univerzálního uživatelského rozhraní a technologie s tím související. V kapitole 4 si představíme náš koncept univerzálního uživatelského rozhraní. A v kapitole 5 si popíšeme možnou implementaci, která také slouží jako ověření konceptu UUI.

Kapitola 2

Teoretický základ

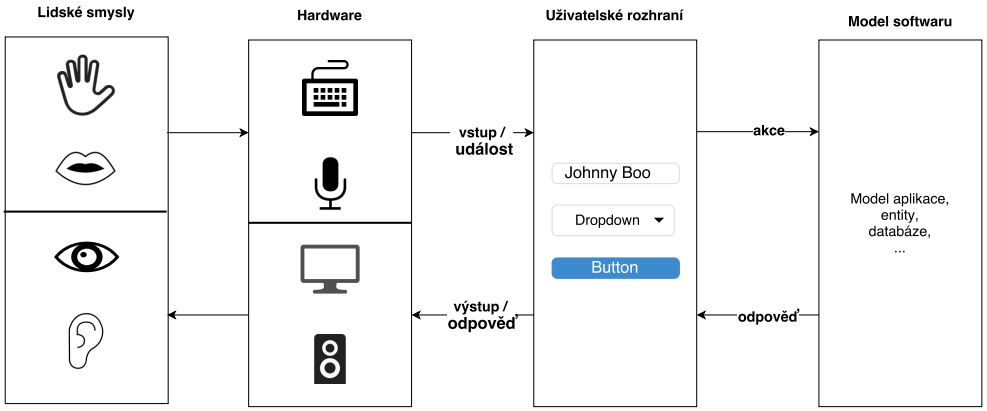
V této kapitole si popíšeme potřebnou teorii pro popis, návrh a vývoj uživatelského rozhraní. Nejdříve si zadefinujeme co je uživatelské rozhraní a stručně si popíšeme několik typů uživatelského rozhraní. Dále si popíšeme softwarové přístupy ve vývoji softwaru, které lze použít při vývoji uživatelského rozhraní. V další části se seznámíme s programovacími paradigmy v oblasti uživatelských rozhraní. Poté se seznámíme s návrhovými vzory při návrhu uživatelských rozhraní a následně se podíváme blíže na grafické uživatelské rozhraní (DSL jazyky GUI, generování GUI).

2.1 Definice uživatelského rozhraní

Uživatelské rozhraní (UI) je rozhraní mezi člověkem a zařízením. Na jedné straně je uživatel a na druhé straně zařízení. To jakým způsobem bude uživatel ovládat zařízení je dáno uživatelským rozhraním. Technicky řečeno je UI „převodníkem“ z reálného světa do světa digitálního a ze světa digitálního do světa reálného. Celé UI si můžeme rozdělit na čtyři části (viz obrázek 2.1). První částí je uživatel, druhou částí hardware, pomocí kterého se ovládá UI, třetí částí je uživatelské rozhraní a poslední částí je samotný model softwaru. Uživatelské rozhraní může být různého typu a to určuje jak uživatelské rozhraní je ovládáné uživatelem, typ použitého hardwaru a jak je UI uživateli zobrazené.

Příklady některých typů uživatelského rozhraní:

- **Rozhraní příkazové řádky (CLI)** - Komunikace se zařízením probíhá pomocí textových řádků, případně řádků příkazů. [3] CLI je často používané rozhraní u serverů, případně při použití vzdáleného připojení k jinému zařízení přes internet. Výhoda CLI oproti jiným rozhraním je možnost automatizovanosti příkazů. Proto je také preferovaným rozhraním u zkušených uživatelů. Další skupinou, pro které je CLI primární, jsou lidé s vadami způsobující horší vidění, protože CLI může být také



Obrázek 2.1: Jednotlivé samostatné části uživatelského rozhraní.

zobrazované pomocí Braillova písma. [4] Přední výhody CLI jsou: nevyžaduje skoro žádné zdroje (paměť, CPU, ...), je snadno automatizovatelný a výstup lze převést do jakéhokoliv jiného zobrazovacího formátu. Nevýhody CLI jsou: vizuálně nebohaté a neintuitivní pro začínající uživatele.

- **Grafické uživatelské rozhraní (GUI)** - Využívá grafické komponenty (widgety) k zobrazení informaci a k ovládání aplikace. Hlavní výhodou GUI je přeměna ovládání zařízení na více intuitivní a také umožňuje rychlé učení se v používaní zařízení. Například, je snazší soubor přesunou přetažením ikony souboru na složku, než si pamatovat textový příkaz. [5] GUI používá různý typ hardwaru, především: myš, klávesnici, trackball nebo touchpad. Akce v GUI jsou prováděné skrze manipulaci s grafickými komponenty GUI. [6]

- **Webově založené uživatelské rozhraní (WUI)** - UI podobné GUI s tím, že obsah zůstává ve webovém prohlížeči a změna obrazovek probíhání pomocí klikáním na webové odkazy. [7]

- **Objektově-orientované uživatelské rozhraní (OOUI)** - Rozhraní založené na objektově-orientovaných programových metaforách, které umožňují manipulovat s objekty a vlastnostmi pocházející z domény aplikace. [8] Uživatel si zvolí objekt, změní jeho vlastnosti (velikost, barvu, atd.) nebo provede akce nad objektem (přesun, kopie, atd.) Například v aplikaci na vektorovou grafiku pracujeme s čáry, kruhy, čtverci atd. Je také vidět, že OOUI má značné společné prvky s konstrukcí softwaru z doménových objektů, ale nevyplývá z toho, že OOUI objekty nutně vycházejí z objektů domény.

- **Dotekové rozhraní** - Rozhraní, které se ovládá pomocí dotekové vrstvy.
- **Zvukové uživatelské rozhraní** - Rozhraní, které přijímá jako vstup zvukové příkazy a produkuje jako výstup vygenerované zvukové příkazy. Toto rozhraní může být spuštěno pomocí klávesy, tlačítkem nebo zvukovým vstupem.

2.1.1 Design

Design nebo návrh uživatelského rozhraní je snaha o vytvoření co nejlepšího rozhraní pro komunikaci s uživatelem. Při designu dbáme na několik rád a zásad, které nám pomůžou se vyvarovat špatnému návrhu UI. To vede k tomu, že se při návrhu UI využívají i jiné disciplíny, jako například je ergonomie nebo psychologie. Hlavním cílem designu UI je vytvořit UI, které umožní uživateli jednoduše, efektivně a příjemně ovládat zařízení za účelem dosažení kyženého výsledku. [9] Zjednodušeně to znamená, chtít po uživateli minimální vstup pro danou akci a také zobrazovat minimální nechtěný výstup uživateli. Více informací o designu UI můžeme najít ve standardu ISO-9241.

2.1.1.1 Kvalita UI

Jak již bylo zmíněno v předchozím odstavci, design UI je velice důležitá část vývoje softwaru. Protože uživatel hodnotí kvalitu aplikace pouze podle UI, protože to je jediné co vidí z celé aplikace. Pokud tedy chceme vytvořit kvalitní UI, tak bychom se měli držet aspoň několika málo zásad. První zásadou je princip minimálního překvapení. Tento princip platí pro návrh všech druhů uživatelského rozhraní a říká, že člověk dokáže dávat plnou pozornost pouze na jedné věci. Další skupinou zásad jsou „Nielsenova heuristická desatera“. [10] Tato skupina shrnuje všechny různé rady a doporučení, které lze nalézt v jiných zdrojích ([11, 12]).

2.1.1.2 Nielsonovo heuristické desatero

1. Viditelnost stavu softwaru - Uživatel by měl vědět vždy co se děje s aplikací, případně bychom měli zobrazit odezvu aplikace v dostatečně rozumném čase.
2. Shodnost s reálným světem - Software by měl komunikovat s uživatelem s frázemi a slovy z reálného světa než ze světa systémově orientovaného.
3. Uživatel ovládá a má svobodu - Měla by vždy existovat možnost návratu zpět a možnost vrátit změny.
4. Konzistence a standardy - Aplikace by měla být konzistentní napříč celou aplikací.

5. Prevence chyb - Software by měl zabránit chybě a pokud chyba nastane tak by měl zobrazit uživateli potvrzení, jestli chce akci opravdu provést.

6. Rozpoznání než pamatování - Minimalizovat potřebu pamatovat si jednotlivé dialogy. Instrukce pro danou akci by měli být snadno dohledatelné.

7. Flexibilita a snadné použití - Možnost jednoduchého ovládání pro nezkušené uživatelé, tak i ovládání pro zkušené uživatele.

8. Estetika a minimalistický design - Žádné zbytečné informace by neměli být vidět.

9. Pomoc uživatelům s chybami - Chybové hlášky by měly být srozumitelné a zobrazeno konstruktivní řešení pro daný problém.

10. Pomoc a dokumentace - Software by měl obsahovat dokumentaci, která bude srozumitelná, prohledatelná a zbytečně velká.

Při designu UI nesmíme také zapomenout na zásady uživatelského rozhraní pro danou platformu, aby prožitek z používání aplikace na dané platformě byl konzistentní pro všechny aplikace. Tyto zásady obsahují jak obecné zásady pro návrh aplikace (procházení mezi obrazovkami, dialogy, layout UI, atd.), tak i konkrétní (odsazení od rámečku okna, vzdálenost mezi komponenty UI, atd.). Příkladem některých zásad můžou být zásady pro aplikace tvořené na platformu Windows [13], OS X [14], Linuxu s knihovnou Gnome [15].

2.2 Softwarové přístupy

Abychom vždy nevytvářeli UI takzvaně na „zelené louce“, je dobré dodržovat některé softwarové přístupy. Neplatí to nejen pro UI, ale pro software obecně. Tím, že tyto přístupy dodržíme lze získat několik málo výhod. Například znovupoužitelnost některých komponent, snadnější pochopení fungování systému a nebo nižší úsilí při údržbě systému.

V této podkapitole se nejdříve seznámíme s model-driven architecture, která se používá často při vývoji velkých systémů. Poté si stručně něco povíme o normalizovaných systémech a o jejich dopadu na evolvabilitu systému. A nakonec si popíšeme přístup přímo určený pro vývoj UI.

2.2.1 Model driven architecture

Myšlenkou model driven architecture (MDA) je oddělit byznys a aplikační logiku od technologické platformy. [16] To samo osobně není nová myšlenka, ale MDA přináší nové postupy a způsoby jak správně transformovat analytický a návrhový model. Hlavním cílem MDA přístupu je lepší znovupoužitelnost

a součinnost díky oddělené architektuře. Více informací o MDA lze nalézt na oficiálních stránkách společnosti OMG [17], která stojí za vytvořením MDA.

Obvyklý postup při návrhu systému s použitím MDA:

1. Specifikace celého systém nezávisle na platformě.
2. Specifice platforem, na kterých poběží systém.
3. Výběr konkretní platformy.
4. Transformace systému podle zvolené platformy.

Dále MDA obsahuje několik modelů: [16]

Computation Independent Model (CIM) Zaměřuje se na prostředí a na obecné požadavky na systém. V tomto modelu není zobrazena detailní struktura a ani konkretní implementace. CIM reflektuje byznys požadavky zákazníka. Jako CIM modely můžou být například použity: procesní modely, use case diagramy, diagramy činnosti.

Platform Independent Model (PIM) Zabývá se kompletní specifikací systému, která se ještě nemění dle platformy. Tento model musí být použitelný na všechny platformy. Především popisuje chování a strukturu systému, ale jen do míry přenositelnosti mezi platformami. Oproti CIM je doplněn o principy, pravidla, omezení, někdy i o algoritmy.

Platform Specific Model (PSM) Model již závislý na platformě systému, který doplňuje PIM model o konkrétní řešení v dané technologii a na dané platformě. Většinou je modelem model tříd nebo relační model. Jedná se o poslední krok před implementací z modelu.

2.2.2 Normalizované systémy

Údržba softwaru ve vývoji softwaru je nejdražší část životního cyklu softwaru a často také vede k zesložiťování architektury a snižuje kvalitu softwaru. [18] S každou změnou v systému systém roste a přidávají se nové vazby nebo závislosti mezi komponentami.

Teorie NS je založena na přesvědčení, že informační systémy potřebují být stabilní s ohledem na definovanou sadu očekávaných změn. Díky tomu, můžou být normalizované systémy definované jako stabilní s ohledem na definovanou sadu očekávaných změn, u kterých se vyžaduje, že budou mít za následek omezenou výši dopadu na systémová primitiva a to i v neomezeném časem života těchto systému. [18] Tato stabilita a evolvabilita může být docílena dodržením množiny teorému a použitím primitiv popsané v teorii NS.

Příklad použití NS na reálné aplikaci je popsána v publikaci [19], kde příkladem byla aplikace na rozpočty v lokální Belgické státní správě.

2.2.2.1 Dnešní problémy ve vývoji softwaru

Dnešní vývoj softwaru má několik symptomů problémů, které snižují evolvabilitu systému. Hledáním řešení na tyto problémy vznikla teorie normalizovaných systému a také tato teorie pomáhá vytvořit evolvabilní architekturu systémů. Některé tyto problémy jsou:

Omezená možnost zpětného dohledání Mapování mezi reálným světem, modulech v návrhu a zdrojovém kódu není vždy jasné ve výsledném systému.

Špatné osvojení metod vývoje softwaru Jen polovina firem používá metodiky na vývoj softwaru. [18]

Nejasné pojmy v návrhu Vágně definované pojmy v návrhu softwaru. Například nízká provázanost (low coupling) může být docílena několika způsoby.

2.2.2.2 Teorémy NS

Normalizované systémy obsahují několik teorému, které by se měli dodržet pokud vytváříme normalizovaný systém, který je stabilní vůči neočekávaným a skrytým (zvýšení provázanosti nebo závislostí mezi komponenty) změnám. [19, 18] Tyto teorémy jsou:

Separation of Concerns (SoC) Oddělení systému na různé moduly, které budou obsahovat pouze jednu funkcionalitu a budou se co nejméně překrývat.

Data Version Transparency (DvT) Data entity můžou být změněny bez dopadu na komponenty, které ji používají jako vstup nebo výstup.

Action Version Transparency (AvT) Akce entity může být změněna bez dopadu na komponenty, které ji používají.

Separation of States (SoS) Oddělení všech kroků work-flow programu, tak že se drží stav aplikace po každém kroku.

2.2.2.3 Elementy normalizovaných systémů

Aby bylo snadné dodržet teorémy NS definuje teorie NS ještě elementy, které jsou primitivy(návrhovými vzory) normalizovaných systémů a zároveň stavebními bloky flexibilních softwarových architektur. [19, 18] Tyto elementy jsou také nezávislé na technologické implementaci. Elementy NS jsou:

Data element Zapouzdřuje data s gettry, settry.

Action elements Obsahuje pouze jednu funkcionalitu. NS ještě rozlišují tento element na čtyři implementační elementy: standardní(akce spouštěna systémem), manuální(uživatelský vstup je potřeba k provedení akce), „bridge“(vytváří jiné elementy) a externí(k provedení akce je potřeba externím systém). [18]

Workflow element Popisuje sekvenci akčních elementů v izolovaném modulu.

Connector element Zajišťuje interakci s externím systémem bez volání komponent mimo stavový režim.

Trigger element Spravuje stavy a spouští akční elementy.

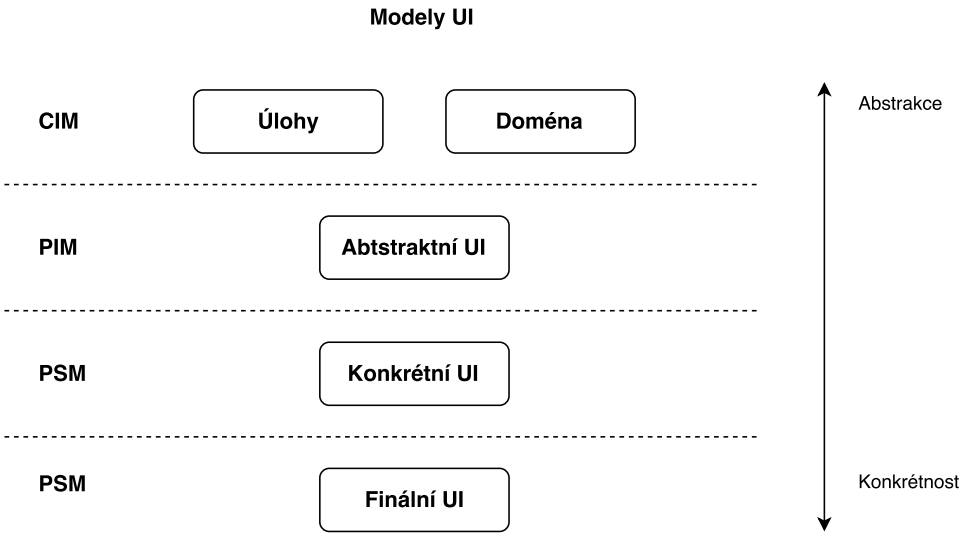
2.2.3 Model based UI development

V dnešní době se UI interaktivních systémů vyznačují různorodými prvky[20]:

- Různí koncoví uživatelé: Uživatelé dle různých preferencích, monžostí, kultury a rozdílné úrovně zkušeností.
- Různé platformy (desktopy, chytré telefony, tablety, atd.), jiné vstupy (více dotyková gesta, pohybové senzory, ...), různá interakce (grafika, mluvené slovo, gesta, ...).
- Různé programovací a značkovací jazyky (například C++, HTML, Java) a knihovny widgetů (například Swing, Qt, GTK+).
- Různé pracovní prostředí, kde se používá aplikace.

Model based user interface development (MBUID) je jeden z přístupů zaměřený na vytvoření UI pomocí modelů, kde jeden model odpovídá jedné stránce(formuláři) UI.[21] A také je zaměřen na identifikací abstraktních modelů, které nám umožní navrhovat interakci aplikace na sémantické úrovni, než začít rovnou na implementační úrovni. To nám umožní se zaměřit na aspekty UI a nebýt ovlivněn implementačními detaily. [20] Specifikace modelu v MBUID také obsahuje abstraktní popis aspektů pomocí popisného jazyka UI, to nám také poskytuje základ pro automatické generování uživatelského rozhraní.

V publikaci [21] se můžeme dočíst o použití MBUID na ukázkové implementaci MBUIDE jménem WAINE. WAINE je založen na generaci UI z databáze UI modelů, data a přizpůsobení. Výsledné vygenerované UI je ve formátu webové stránky (HTML, CSS, Javascript). V [20] můžeme vidět použití MBUID na několika aplikacích.



Obrázek 2.2: Model based UI development, adaptováno dle [20].

2.2.3.1 Postup modelování

Postup modelování uživatelské rozhraní pomocí MBUID je následující: [21, 22]

Koncepty a úlohy Specifikace hierarchie úloh pro práci s doménovými objekty. Používají se většinou doménové a úlohové modely.

Abstraktní UI Zobrazuje UI z hlediska interaktivních komponent bez implementačních a technologický detailů (platformy, programovacím jazyků nebo značkovací jazyk). Používají se dialogové modely.

Konkretní UI Popisuje konkretní interaktivní komponenty, ale stále implementačně nezávislé. Konkrétní reprezentace modelů.

Finální UI Implementace UI v programovacím jazyce nebo značkovacím jazyce.

2.2.3.2 Znovupoužitelnost v UI

Softwarová znovupoužitelnost zvyšuje produktivitu při vývoji softwaru a také zvyšuje kvalitu výsledného produktu. Znovupoužitelnost můžeme také vidět v popisu MBUID přístupu. [21] Znovupoužitelnost pomocí MBUID přístupu můžeme provést třemi způsoby:

- Znovupoužitelnost pomocí popisného jazyka UI. Tato metoda se vyznačuje vlastnostmi: kompozice, jazyk založený na komponentách a šablonách.

- Znovupoužitelnost pomocí různých transformací, které generují UI pro daný kontext(platforma, hardware, uživatele, atd.).

- Znovupoužitelnost pomocí vzorů, kde vzory jsou abstraktní meta-modely často používaných řešení, které mohou být použity k vytvoření UI specifikací.

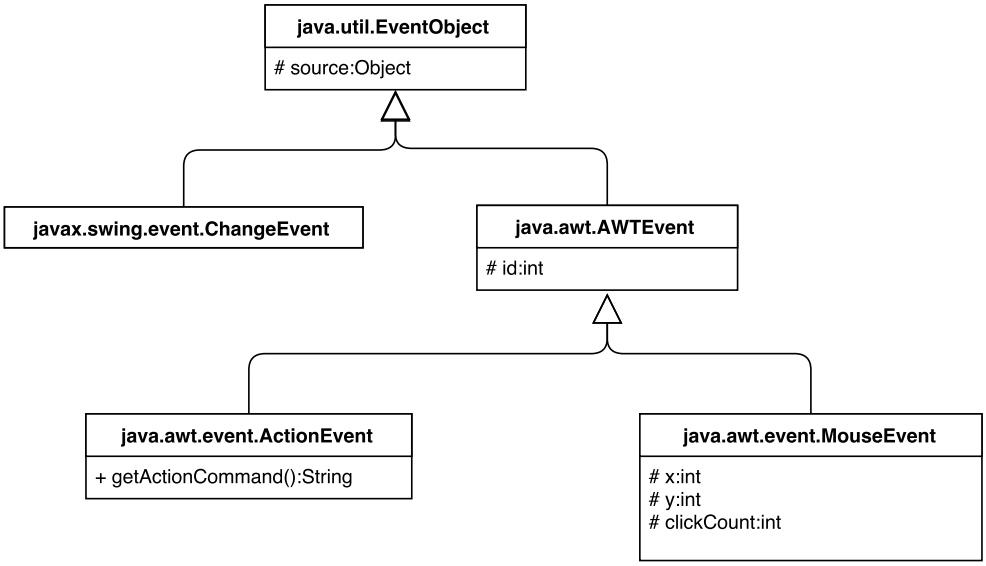
2.3 Programovací paradigmata v UI

Při programování UI s interakcemi s uživatelem nelze použít jen klasické imperativní programování, ale je potřeba myslet na „reaktivitu“ systému. To znamená, že musíme používat asynchroní spouštění bloků kódu a i asynchronně odpovídat na požadavky od uživatele aplikace. K tomuto účelu bylo vytvořeno několik programovacích paradigmat. My se podíváme na ty dnes nejvíce používané. Prvním je event-driven programming(EDP), které definuje a používá události. Dalším paradigmatem, který by měl nahradit EDP je reaktivní programování, protože řeší některé problémy EDP a je abstrakcí nad EDP. A v poslední části této podkapitoly si popíšeme funkcionálně reaktivní programování, které vychází z reaktivního programování a má být ideálním řešením pro programování uživatelských rozhraní obecně.

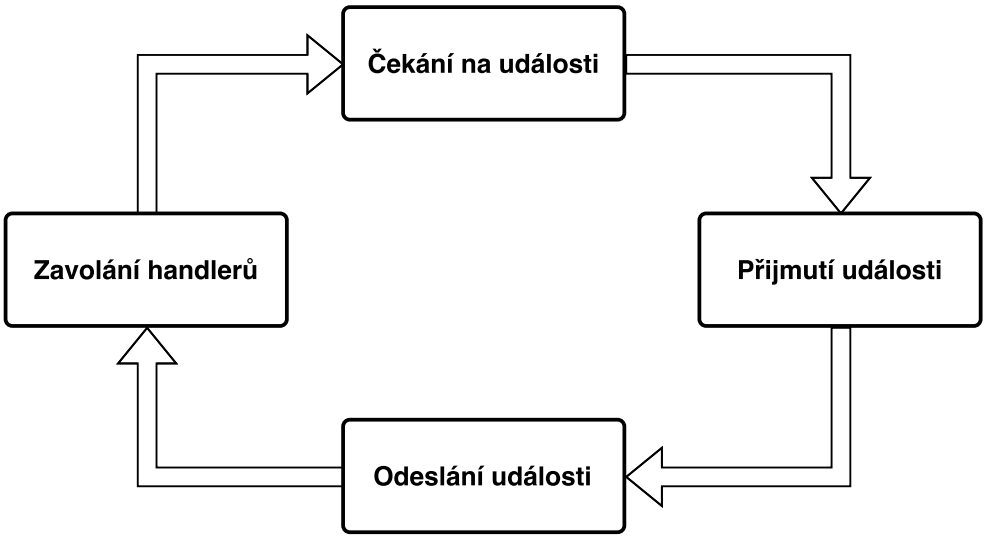
2.3.1 Event driven programming (EDP)

Programovací paradigma, ve kterém je běh programu určen pomocí událostí. Události jsou nejčastěji: uživatelská akce(kliknutí myši, stisknutá klávesa, atd.), senzory, zprávy z jiných programů a vláken. [23] EDP patří mezi jedno z nejdůležitějších konceptuálních a praktických programovacích paradigmat. [24] Lze programovat v jakémkoliv programovacím jazyce, ale snadněji se programuje pokud jazyk obsahuje vysoko úrovňové abstrakce jako je například closure. Aktuálně většina moderních programovacích jazyků, jako je například Java nebo C#, obsahuje komplexní třídní hierarchii se zacházením s událostmi. Ukázka části této hierarchie v Javě je vidět na obrázku 2.3. Více o různých implementacích a jejich použití lze se dočíst například v těchto publikacích [25](Java) [26](Javascript).

Na obrázku 2.4 je zobrazen základní princip EDP. Program vždy obsahuje hlavní smyčku, která poslouchá události. Jakmile je událost vytvořena, je následně zachycena smyčkou, doplní se do události dodatečné informace(kdo událost vytvořil a co se stalo) a pak se aktualizuje celý stav aplikace. Následně je událost předána všem registrovaným event handlerům. [25] Event handler spustí kód uložený ve svém closure. Při implementaci EDP, ale není dobré udělat jednu velkou smyčku, kde by byly všechny klauzule pro všechny události. Je potřeba vytvořit rozumný a správný design a logiku, protože by pak systém mohl být ve výsledku dostatečně neresponzivní.[26]



Obrázek 2.3: Část implementace event driven programming v Javě.



Obrázek 2.4: Princip fungování event driven programming.

Event driven programming je velice podobný návrhovému vzoru publish-subscribe (viz [27]), kde odběratelé jsou posluchači/event handleři a vydavatelé jsou uživatelské akce.

2.3.1.1 Vlastnosti event driven programování

Event driven programming se vyznačuje následujícími vlastnostmi:

Loose coupling EDP v objektově orientovaných systémech obsahuje zdrojový objekt a zachycující objekt (handler). Tato vazba má obecně tyto vlastnosti:

- **Registrace za běhu** Přidání/odebrání event handlerů je až za chodu programu. Správný event driven návrh, zaregistruje handlery hned na začátku programu a nikdy je neodregistruje. V případě potřeby změny prováděného kódu v event handleru je dobré použít místo odregistrování návrhový vzor stav (viz [28]).
- **Multicasting** Jedna událost může být poslána více event handlerům.
- **Multiplexing** Event handler může dostat údálost od různých zdrojů. Například: V GUI lze uložit soubor přes položku v menu uložit nebo přes panel a ikonu uložit.

Ovládání stavem Stav udává na jaké událostmi bude program odpovídat a jak bude na ně reagovat.

Paralelní zpracování Vícevláknové programování a EDP jsou v blízkém vztahu. Kód event handleru může být zpracován ve vlastním vlákně.

Dle popisu EDP vidíme, že EDP má největší uplatnění při progrmaování interaktivního UI. Událostmi jsou uživatelské akce a ty jsou poslouchány hlavní smyčkou aplikace, která na základě události provádí volání daných bloků v event handlerech. Dnes se stále ještě hojně používá při programování UI, například v Node.js, Javascriptu nebo Javě Swing. Ale postupně je vytlačován reaktivní programováním (viz 2.3.2).

2.3.1.2 Rozdíl mezi event-driven a message-driven

Event-driven programování a message-driven programování(MDP) jsou si hodně podobné a fungují na stejném principu. Message-driven systém má také hlavní smyčku, kde se zachytávají různé zprávy a podle obsahu zprávy se vykonává daný určený blok kódu. Ale rozdíl mezi EDP a MDP je takový, že MDP zprávy mají jasně daný jediný cíl, kam se zpráva pošle. kdežto u EDP je událost odeslána všem event handlerům, které daný typ události poslouchají(pozorují). MDP je také preferované, aby bylo asynchronní a komunikovalo pouze přes síť, kdežto události u EDP jsou pouze komunikovány lokálně. [29]

2.3.2 Reaktivní programování

Paradigma postavené na datovém toku (proudu dat) programu a propagaci změn hodnot proměnných. [30, 31, 32]. Prakticky to znamená, že by jazyk nebo framework měl poskytovat statický nebo dynamický proud dat (může se změnit kdykoliv během běhu programu) s výpočetním modelem, který automaticky propaguje změny. V tomto směru nám pak RP poskytuje jednoduchou abstrakci nad správou událostí a také automaticky spravuje běh programu.

Při použití tradičních programovacích technik (návrhové vzory, event-driven programování) jsou interaktivní programy postaveny na asynchronním callbacku. Programování pomocí callbacků, může být často velice obtížný úkol, protože nevýhodou callbacku je, že jejich pořadí spuštění je nedeterministické. Další nevýhodou je, že často nemá návratové hodnoty, takže musí provádět side-efekty v programu, aby mohl ovlivnit stav programu. [31] Jak je popsáno v [31], tak polovina chyb v aplikacích Adobe desktop byla způsobena logikou v event handlerech, který používali callback. Abychom snížili tuto chybovost je potřeba abstrakce, která zapouzdří jak logiku obsluhování události tak i správu stavů programu, což reaktivní programování nabízí.

RP je ideální paradigma pro vývoj EDP programů, které by bylo těžké naprogramovat pomocí tradičních(sekvenčních) programovacích technik, protože je nemožné predikovat nebo kontrolovat posloupnost externích událostí (vstup uživatele, vstup běhového prostředí, ...), neboli také převrácená kontrola (inverted control), kdy program je ovládán externími událostmi a ne posloupností příkazů, které napsal vývojář aplikace.

Nejvíce interaktivní částí aplikace je GUI (v případě webový aplikací frontend). V dřívějších dobách webové aplikace byly tvořené tak, že se dlouhý formulář odeslal na backend a pak se provedlo jednoduché vykreslení formuláře na frontend. Dnes je požadavek, aby aplikace byly co nejvíce reaktivní a změna jedné komponenty (textového pole) byla ihned uložena na backend. RP nám v tomto směru umožňuje vytvářet hodně interaktivní programy, které vyhovují těmto požadavkům, proto také RP získává na své popularitě jako ideální řešení pro event-driven systémy.

Přehled o možnostech reaktivního programování a celkově obecně o reaktivní programování se lze dočíst v publikaci [31].

2.3.2.1 Více významů reaktivního programování

Někdy se stává, že když lidé mluví o „reaktivním“, v kontextu vývoje nebo návrhu programů, tak můžou myslet jeden ze tří významů: Reaktivní systém (architektura a návrh), reaktivní programování (deklarativní programování založené na událostech) a funkcionálně reaktivní programování. Abychom si ujasnili významu slova reaktivita, tak reaktivita je množina návrhových principů, které pojednávají o architektuře, návrhu a implementaci systému v distribuovaném prostředí.[29]

Hlavně reaktivní programování by nemělo být zaměňované za funkcionální reaktivní programování(FRP). FRP je podmnožina reaktivního programování, kde se používá vlastností funkcionálního progrmaování (viz popis v sekci **??**).

A reaktivní systém je systém, který interaguje s prostřední tak, že je aktivovaný ze vstupních událostí z prostředí a vytváří výstupní události jako odpověď. [33] Reaktivní programování je většinou event-driven, oproti tomu reaktivní systémy jsou message-driven. Reaktivní systémy jsou většiny rozděleny do nezávislých paralelních komponent, které kooperují k vyřešení daného úkolu a vyznačují se vysokým souběžným zpracováním. [33]

2.3.2.2 Základní primitiva

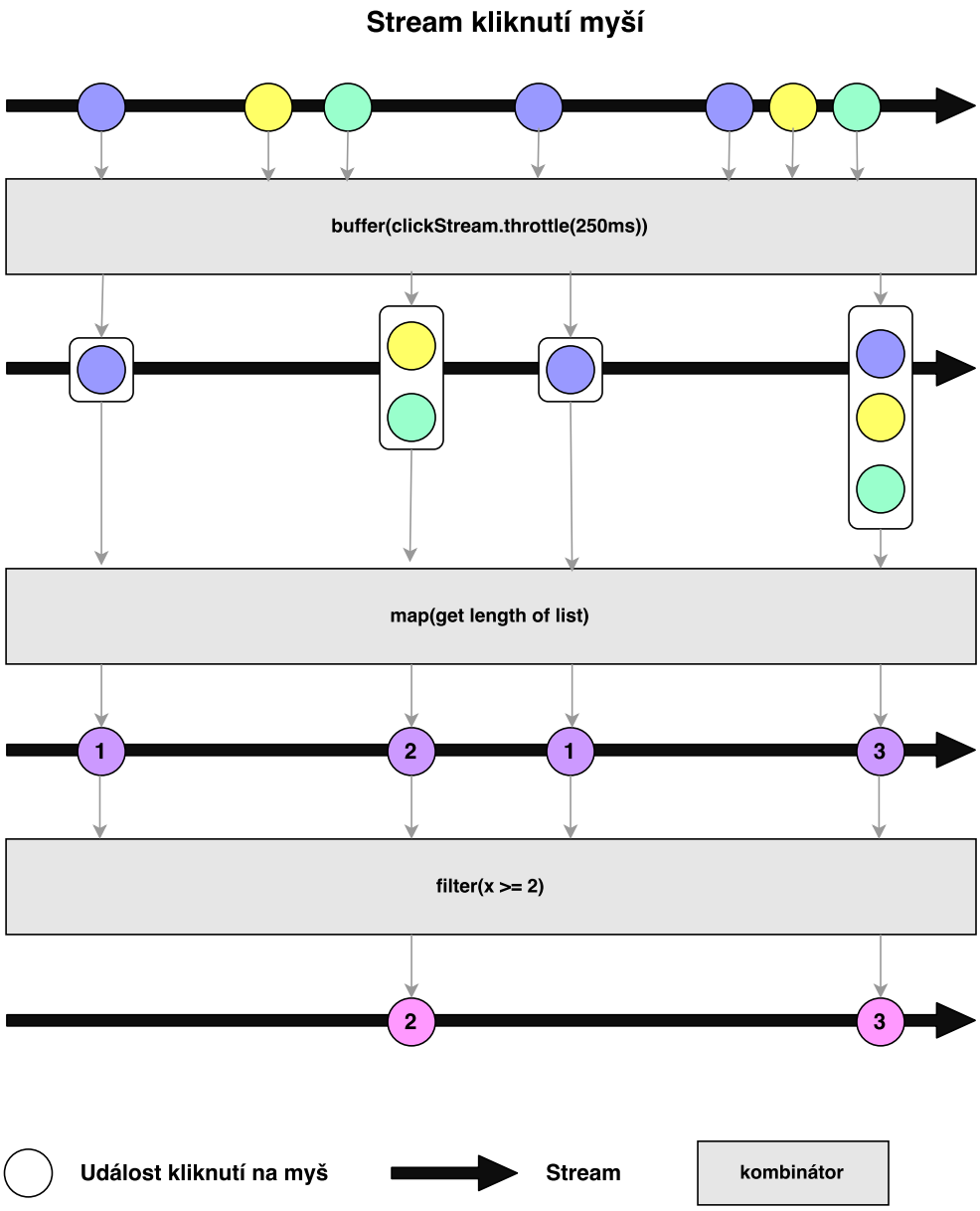
Nejdůležitějším primitivem reaktivního programování je stream, který se chová jako datový tok neboli sekvence událostí seřazené v čase. Stream může emitovat tři typy objektů: hodnotu (nějakého typu), chybu nebo ukončení (kompletní) streamu. [30] Tyto emitované hodnoty jsou emitovány asynchronně a definované funkcí, která je provedena pokud je hodnota emitována. Jiná funkce je pro hodnotu, jiná pro chybu a jiná pokud je stream ukončen. Streamy jsou asynchronní a neblokující. Poslouchání streamu se říká subscribing a přiřazené funkce jsou pozorovatelé. Stream lze vytvořit z jakéhokoli objektu a také cokoliv může být stream.

Reaktivní výrazy jsou více komplexním chováním, které jsou vytvořené přes kombinátory. Ty vezmou existující reaktivní výraz a vytvoří nový reaktivní výraz. Kombinátory jsou dalším primitvem RP, které kombinují, vytvářejí, filtrují a transformují streamy kde výsledkem použití kombinátoru je opět stream. [33] Příkladem kombinátorů jsou funkce merge (kombinace více streamů), map (mapvání a transformace streamu) a filtrace. Příklad kombinátoru mapování a filtrace je na obrázku 2.5.

2.3.2.3 Vlastnosti reaktivních systémů

Responzivnost systému Znamená rychle reagovat na všechny uživatele jak během „normálního“ chodu tak i během chyby s cílem zajistit trvale pozitivní uživatelskou zkušenost. [34]

Přizpůsobivost reaktivních systémů Responzivnost během vzniku chyby. Jinak řečeno, při správném použití návrhu a architektury je dosaženo správného fungování systému jak během normálního běhu tak chybového běhu. [34] Přizpůsobivost je dosažena několika použitím několik technik: replikace, komponenty, izolace a delegace. Poruchy mohou nastat v každé komponentě. Proto komponenty musí být odizolovány od sebe navzájem, tím se zajistí, že části systému mohou selhat, aniž by byl systém ohrožen jako celek.[35]



Obrázek 2.5: Příklad použití kombinátorů (map, filter, throttle) na streamy v RP. Adaptováno dle [30].

Pružnost reaktivních systémů Responzivnost pod zátěží systému. Škálovatelný systém je možné snadno aktualizovat na požádání, aby byla zajištěna schopnost reagovat za různých podmínek zatížení. [34]. Použitím prediktivních, reaktivních a škálovatelných algoritmů v systému se dosáhne efektivní škálovatelnosti systému. [35]

Produktivita reaktivních systémů Reaktivní systém je nejvíce produktivní architektura, která je zatím známa (v kontextu více-jádrových systémů, více-jádrových procesorů, cloudu a mobilních architektur) [29]

Reaktivní programování vysokého řádu Lze z datového toku vytvořit jiný datový tok a to takový, že výsledkem nového datového toku je další datový tok. [32]

2.3.2.4 Výpočetní modely

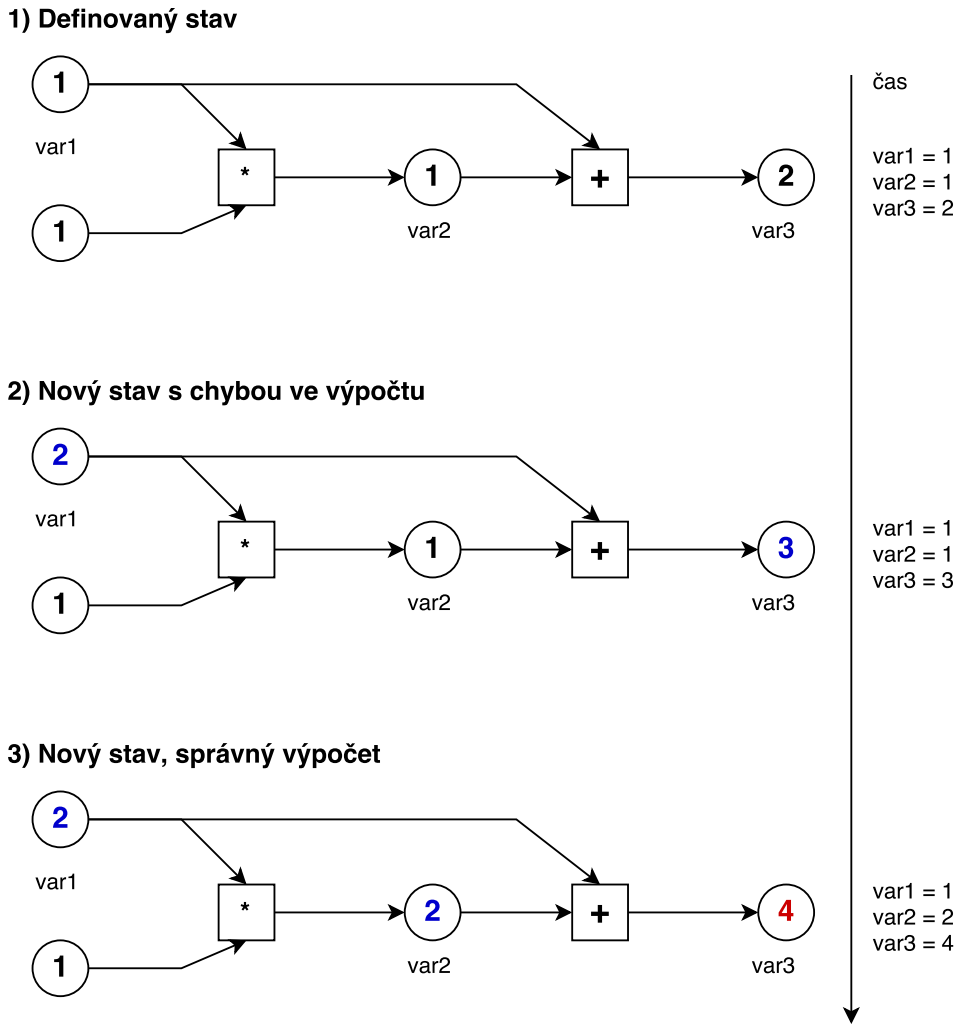
Výpočetní model v RP určuje jak se budou propagovat změny napříč celým závislostním grafem hodnot a výpočtů. Z pohledu programátora se jeví propagace změn automatická, ale propagaci změn lze provést několika způsoby, nejvíce přirozené je invalidce (push-based) nebo líné načítání hodnot (pull-based).

Pull-based Výpočet potřebuje hodnoty ve chvíli kdy jsou „pullnuté“(požadované) ze zdroje. To znamená, že propagace změn je řízena potřebou nových dat. Tento model je ale hodně kritizován, protože se často stává, že má značnou odezvu mezi událostí a odpovědí, protože všechny závislé výpočty musí být vypočteny ihned. To může vést k prostorovým a časovým leakům, které se mohou časem zvětšovat.

Push-based Jakmile má zdroj nová data, tak je „pushne“(uloží) do závislých výpočtů. Propagace je řízena dostupnosti nových dat. Tento model se dobře uplatní v reaktivních systémech, kde je potřeba vidět změnu ihned.

2.3.2.5 Chybný výpočet

Chybný výpočet je aktualizační inkonzistence, která může nastat během propagace změn. Když je výpočet spuštěn ještě předtím než jsou všechny závislé výrazy vyhodnoceny (viz obrázek **??**. To může způsobit, že nové hodnoty jsou zkombinovány se starými hodnotami, což vede k chybným výpočtům. [31] Tato chyba může nastat jedině v jazycích, které používají push-based model. Následně chyba pak vyústí k nesprávnému stavu v programu a zbytečnému přepočítání hodnot. Většina reaktivních jazyků eliminuje tyto chyby pomoci přeskládání výrazů a topologicky setřídí graf závislostí. To způsobí, že výraz je vždy výpočtech po té co jsou všechny závislé hodnoty vypočítány. Také efektivní implementace RP by se měla vyhnout zbytečným výpočtům hodnot proměnných, které se nemění.



Obrázek 2.6: Znázornění chybného a zbytečného výpočtu v reaktivní programování, adaptováno z [31].

2.3.3 Implementace reaktivních systémů

Reaktivní programování má podobné principy jako návrhový vzor observer. To znamená, že v tradičním přístupu implementace reaktivních aplikací je použití observer vzoru [36], které rozděluje komponenty na konzumenty událostí(observer) a producenty událostí(observables). Ale toto řešení bylo často kritizované, kvůli nedostatku uspořádanosti, inverzní logiky vztahů přes reaktivní entity, porušení zapouzdření a limitované čitelnosti. [37]

Reaktivní systémy musí pracovat v reálném čase a reagovat co nejrychleji, nejlépe v řádech milisekund. Reaktivní chování nemusí mít vždy jednoduché výpočty nebo může mít velké množství dat, tak je potřeba různých optimalizačních technik jako je cachování, inkrementální aktualizace [38] nebo změnou priority v odlišných částech výpočetního stromu. [32]

První funkcionální reaktivní programování bylo představeno v programovacím jazyku Haskell(Fran framework) jako podpora interaktivních animací. [39] Poté bylo implementované v jazyce Scheme, Javascriptu a ve Scale. [37] V [36] se můžeme dočíst cestu abstrakce k RP, přesněji od observer vzoru k reaktivnímu programování v jazyce Scala. Jako další ukázku implementace s vysvětlením RP si lze přečíst v [33], kde výsledkem implementace knihovna SML, která je vytváří nízko-úrovňový systém nad kterým lze snadno vytvořit reaktivní chovaní.

2.3.3.1 Výhody a nevýhody

Primární výhodou reaktivního programování je zvýšení utilizace výpočetních prostředků více-jádrového CPU nebo více-procesorového počítače a díky tomu se také zvýší výkon aplikace. Další výhodou je, že vývojář v tradičních paradigmatech se musí potýkat s řešením asynchronních a neblokujících výpočtů. Reaktivní programování tento problém řeší tím, že odstraňuje explicitní koordinaci mezi komponenty a provádí výpočty sama. [29]

Systémy postavené jako reaktivní jsou flexibilnější a škálovatelné. To usnadňuje jejich vývoji a snadněji se mění. Jsou výrazně tolerantnější k chybám a také nezpůsobí pád celého systému při výskytu chyby. Také jsou vysoce reaktivní, což umožňuje uživatelům efektivní interaktivní zpětnou vazbu. [35]

Nevýhdou reaktivního programování je, že to je nové paradigma, u kterého se musí myslet reaktivně (ve stylu streamů), což se dost liší od klasického imperativního paradigma a to může být docela obtížné.

2.3.4 Functional reactive programming

Funkcionální reaktivní programování(FRP) bylo představeno Elliotem a Hudakem [39] jako zvýšení úrovně abstrakce pro psaní reaktivních programů s důrazem na vysoko-úrovňové funkce. [40] Jinak řečeno, FRP je podmnožina reaktivního programování a používá pro svůj běh funkcionálního programování. ze kterého používá nejčastěji: funkce (například metody map, reduce, filter),

neměnitelné stavy a čisté funkce. [41, 42] To nám dává velké výhody při vývoji reaktivních systémů a UI obecně. Některé z těchto výhodu jsou: [43]

Jednoduché testování Všechny komponenty jsou jen funkce a ty se snadno testují. Funkcionální funkce závisí pouze na vstupních parametrech a nemají žádné vedlejší efekty.

Viditelný tok událostí Reaktivní programování za nás spravuje události, takže odpadá hledání posluchačů událostí ve zdrojovém kódu.

„Cestování v čase“ Umožňuje nám „cestování“ v čase v kontextu naší aplikace, protože FRP ukládá všechny stavy aplikace. Což může být dost užitečné při debuggování aplikace.

Dle výhod použití FRP je vidět, že je slibným přístupem ve vývoji uživatelského rozhraní. Také nám poskytuje vysoko-úrovňovoi, deklarativní a kompozitní abstrakci k popisu uživatelských interakcí a časově závislých výpočtů. [44]

FRP může být založené na diskrétní nebo kontinuální sémantice [42].

Diskrétní V této formulaci je myšlenka chování a událostí zkombinována do signálů, které mají vždy aktuální hodnotu a mění se diskrétně v čase. Tento přístup je často používaný jazyce ELM (**??**) nebo v event-driven FRP.

Kontinuální - Dřívější formulace FRP používali kontinuální sémantiku, zaměřující se na abstrakci nad operačními detaily, které nejsou důležité pro program. Hlavní vlastnosti této formulace jsou [41]:

- definuje hodnoty, které se mění v čase se nazývají chování nebo později signály
- definuje události, které se objevují v diskrétním bodě času
- sytém může být změněn jako odpověď na události, obecně se tomu říká „switching“
- oddělení výpočetních detailů jako je vzorkovací frekvence od reaktivního modelu

Jak již bylo řečeno, implementace dřívějších a většiny FRP jazyků předpokládá, že signály se mění kontinuálně. To znamená, že jejich implementace nepřetržitě, dokonce i když se vstup nezmění, přepočítává proměnné s poslední hodnotou v signálech, což způsobuje zbytečné přepočítávání hodnot.[44, 45]

Také poslední implementace používají model vyhodnocení na požádání (pull), naproti tomu datové řízené(push) se používají u reaktivních systémů, jako jsou GUI. Existují dva dost silné důvody proč používat metodu na požádání(pull) před datově řízenou(push): Chování se mění kontinuálně, takže čekání dokud se změní vstup nemá smysl. Metoda na požádání více sedí s

funkcionálním programování, oproti datově řízenou, které je spíše imperativní. [45]

Dále pak zpracování události může trvat delší dobu, což zpozdí celkově celý FRP systém a způsobí odezvu, která může být valká jako vzorkovací frekvence pro streamy. Řešením na tento problém může být zřetězení(pipeling) událostí. Také musí být stále respektováno, že dokud není jedna událost zpracována, tak se nemůžou zpracovávat ostatní události.[44]

V [45] je představeno řešení problémů obou implementací (pull a push) a to tak, že se zkombinují výhody obou implementací dohromady. To znamená, že hodnota bude přepočítána pouze a ihned, když se změní diskrétní nebo kontinuální vstup.

Některé implementace FRP jsou například: ELM [46], ReactiveX (implementace v několika jazycích, například v Javě, Javascriptu, Swift, Python) [47], Sodium(implementace také v několika jazycích například v Haskellu, Rust a Scale) [48].

Další vyšší abstrakcí FRP je Arrowized FRP (AFRP), které používá Hughesův šipkový kombinátory a Ross Patersonův šipkovou notaci. Více o použití a implementaci v Haskellu se můžeme dočíst v [42].

V [40] se můžeme dočíst o teoretickém základu pro FRP inspirovaný konstruktivním výkladem lineární temporální logiky (LTL). Ale tento teoretický základ je mimo tuto práci.

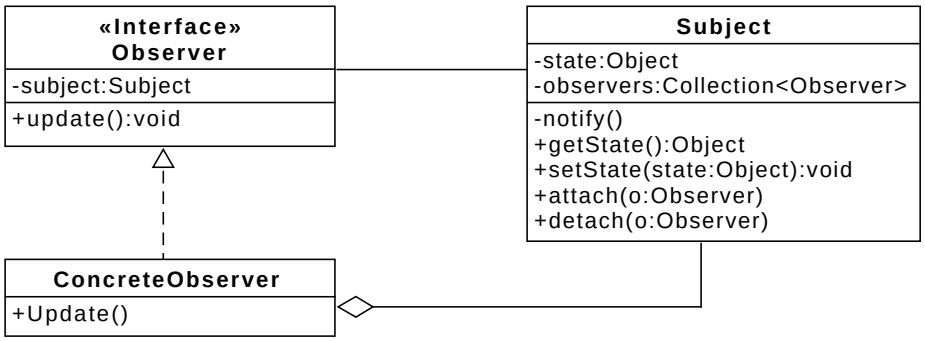
2.4 Návrhové vzory

Pří vývojí softwaru nelze pouze dodržovat jen softwarové přístupy a použít programovací paradigmata ve víře, že výsledný systém bude dobře čitelný a mít nějakou logickou strukturu. Je také dobré použít při vývoji softwaru ještě návrhové vzory, které nám dávají řešení na některé problémy při vývoji softwaru a dávájí lepší sktruktůru celé aplikace. Návrhových vzorů existuje mnoho, ale my si zde představíme jen ty, které lze použít při vývoji uživatelského rozhraní a ty ještě omezíme o vzory, které pouze určují architekturu interaktivních aplikací.

Na začátku si popíšeme základní vzor Observer, který byl inspirací pro reaktivní programování. Pak si popíšeme základní architekturu pro interaktivní systémy, kterou je MVC a následně vzory pocházející z MVC, které lze použít při programování pomocí reaktivního programování.

2.4.1 Observer

Observer vzor definuje vazbu mezi objekty a to takovou, že když se změní stav, tak jsou o této změně informování ostatní objekty. [28] Observeři se můžou k objektu přihlásit (subscribe) respektive odhlásit (unsubscribe), což znamená, že budou respektive nebudou dostávat notifikace o změně stavu objektu. UML diagram observer vzoru je vidět na obrázku 2.7.



Obrázek 2.7: UML Observer vzoru. Adaptováno dle [28].

Subject má privátní metodu notify, která se volá ve chvíli, kdy se změní stav objektu. V metodě notify se proiterují všechny zaregistrovaní observeři, kteří se registrovali přes metodu attach, a zavolá se jejich metoda update. Krátký popis jednotlivých tříd v observer vzoru je následující:

Subject Je sledován a notifikuje observery o změně stavu.

Observer Rozhraní, který definuje jakou metodu musí splňovat observer.

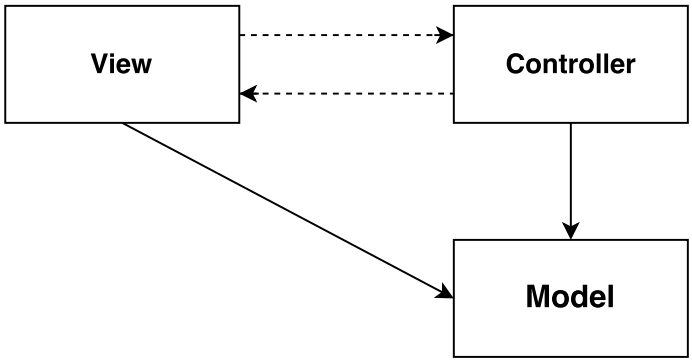
Update Metoda, která je volána v metodě notify a reaguje na změnu stavu Subject.

Notify Metoda, která notifikuje všechny observery o změně stavu voláním metody update.

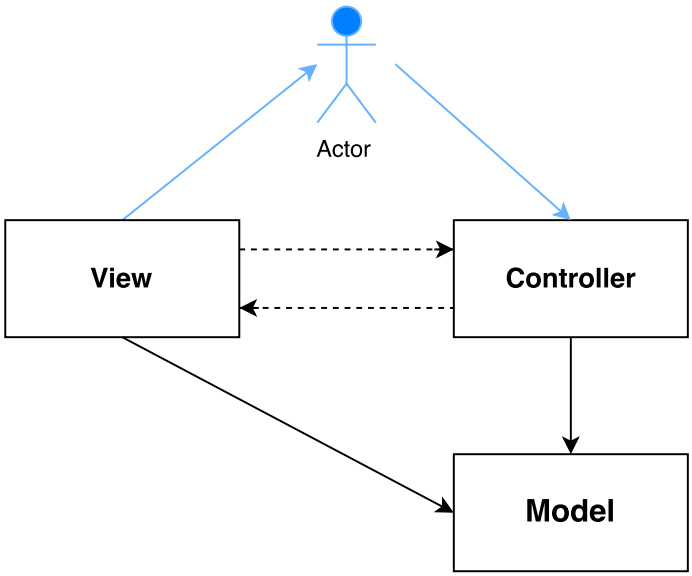
Observer vzor může být implementovaný pomocí dvou modelů systému distribuování notifikací. Pull nebo push model. Jediný rozdíl je v předávání informací observerům. V případě pull modelu, si observer sám v metodě update řekne o informace z Subjectu. Tato situace je znázorněna na obrázku 2.7. Push model naopak vkládá informace jako argumenty do update metody observeru. [28]

2.4.2 Model-View-Controller

Mode-View-Controller (MVC) je návrhový vzor nejvíce používaný v UI, kdy aplikace je rozdělena do tří částí. První částí je model, který obsahuje aktuální datovou reprezentaci nebo jinou objektovou reprezentaci databáze. Většinou model obsahuje entity domény. Druhou částí je View, což je rozhraní pro čtení modelu. A poslední částí je Controller, který obsluhuje požadavky, upravuje data v modelu a vrací odpověď na požadavek. [49]



Obrázek 2.8: MVC architektura, adaptováno z [50].



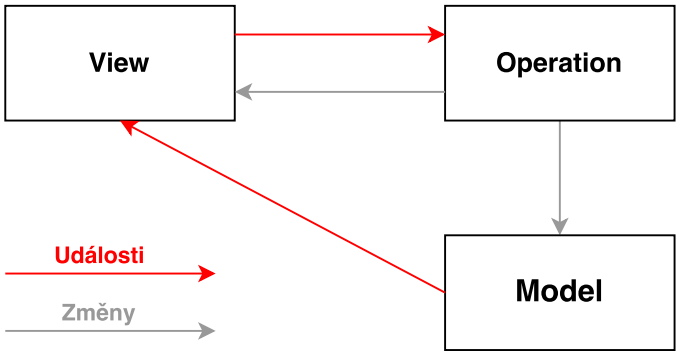
Obrázek 2.9: Interakce s MVC z pohledu uživatele, adaptováno z [50].

Jak je z popisu vidět, MVC je velice jednoduchý architektonický vzor. Proto je také jeden z nejpoužívanějších vzorů v oblasti UI. Používá se ve většině webových frameworků (například Spring a Symfony). MVC vzor je také pravzor velké škály vzorů pro UI, například MVP, MVVM, MOVE, atd.

2.4.3 Models Operations Views Events

Problémem architektury MVC je, že většina kódu je v kontroléru. [51] Řešení na tento problém slibuje architektonický vzore MOVE - Models, Operations, Views a Events.

Modely Zapouzdřuje všechno co aplikace zná. To znamená, že obsahuje en-



Obrázek 2.10: MOVE architektura, adaptováno dle [51],

tity a objekty domény, settry, gettry a případné jednoduché funkce entit.

Operace Zapouzdřují všechno co aplikace dělá. Vrstava odpovědná za změny v modelu, zobrazuje správné view a odpovídá na uživatelské interakce.

Pohledy Mediátor mezi aplikací a uživatelem. Interaguje s uživatelem a zjednodušuje proud uživatelských interakcí do událostí.

Události Používají se ke spojení všech komponent MOVE. Tato technika umožňuje mít od izolované komponenty od sebe.

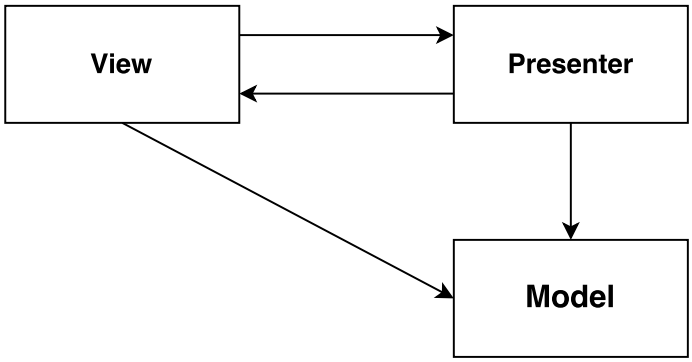
MOVE architektura částečně řeší některé problémy MVC, ale stále má některé nevýhody. Příkladem nevýhody může být menší přehlednost kódu, kvůli událostem, které spojují všechny komponenty. Výhodou je skvělá separace všech komponent.

2.4.4 Model view presenter

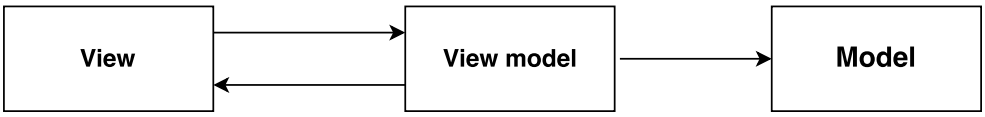
Model view presetner (MVP) je architektonický vzor vycházející z MVC, kde místo kontroléru je prezentér. Model obsahuje opět datovou a i objektovou reprezentaci databáze a navíc obsahuje i byznys logiku. View navíc obsluhuje uživatelský vstup tím, že zachytí uživatelskou interakci a zavolá danou metodu na prezentéru. Nejčastěji se data ve View zobrazuji pomocí návrhového vzoru Observer nebo pomocí data bindingu. Veškerá složitější logika pro view je přesunuta do prezentéru. Prezentér obsahuje aplikační a prezenční logiku, což znamená, že manipuluje s modelem a aktualizuje view. [52] Tomuto MVP vzoru se také někdy říká Supervising Presenter (Controller). [53]

2.4.5 Model-View-ViewModel

MVVM je variace MVP návrhového vzoru z rodiny MVC vzoru. [54] To znamená že model obsahuje data a objektovou reprezentaci databáze. View model



Obrázek 2.11: Model view presenter. Adaptováno dle [52].



Obrázek 2.12: Model view view model. Adaptováno dle [55].

je abstrakce view poskytující vlastnosti a chování. View zobrazuje data z view modelu a obsluhuje uživatelský vstup pomocí delegace na view model. View nepřistupuje k modelu přímo, ale přes mezivrstvu view model, která je kombinací modelu a prezentéru. [55]

MVVM byl navrhnut na použití vazby funkcí ve WPF k lepší separaci od view vrstvy, aby vývojář nemusel programovat GUI, které má většinou na starost UX designer. UX designér nejdříve navrhne View ve formátu XAML a pak se spojí s view modelem. [54] Kritika na vzor MVVM je od samotného autora Johna Gossmana, který poukazuje na to, že MVVM je zbytečné použít na jednoduché UI. [54] Výhoda MVVM naopak je, že je snadno pochopitelný a jednoduchý k testování(stačí testovat view model). [56]

Reaktivní programování a reaktivní frameworky (React.js, Elm, Cycle.js, RxJava, ...) představili novou cestu ve vytváření UI, tím změnili návrh UI. Tím změnili dominanci MVC architektury a MVVM se jeví jako ideální UI vzor pro použití s RP. [43]

2.5 Grafické uživatelské rozhraní

V této části se zaměříme na technologie ohledně GUI. Jistě se ptáte proč se teď zaměřit na GUI? Odpověď je jednoduchá, protože existuje více zdrojů s informacemi o GUI než UI. Také GUI je podmnožinou UI, takže si stačí udělat menší abstrakci nad GUI a většina informaci platí i pro UI. Nejdříve

se podíváme na co bychom měli dbát při modelování GUI, popíšeme si vztah DSL a GUI a nakonec se podíváme na generování GUI.

2.5.1 Modelování GUI

V [57] se můžeme dočíst o univerzálním modelu UI, který lze použití na uživatelské rozhraní jakéhokoliv interaktivního systému [58]. Tento model je zobrazen na obrázku 2.13 Dodržením nebo aspoň respektováním univerzálního modelu nám zajistí, že nebudou vynechaný žádné důležité informace ve výsledném UI. Model je rozdělen do tří stupňů obsahující tři vrstvy. Níže se je popíšeme:

1 stupeň: Struktura Obsahuje tři nejnižší úrovně. Jeden z nejdůležitějších vrstev modelu, který se musí dodržovat při návrhu UI.

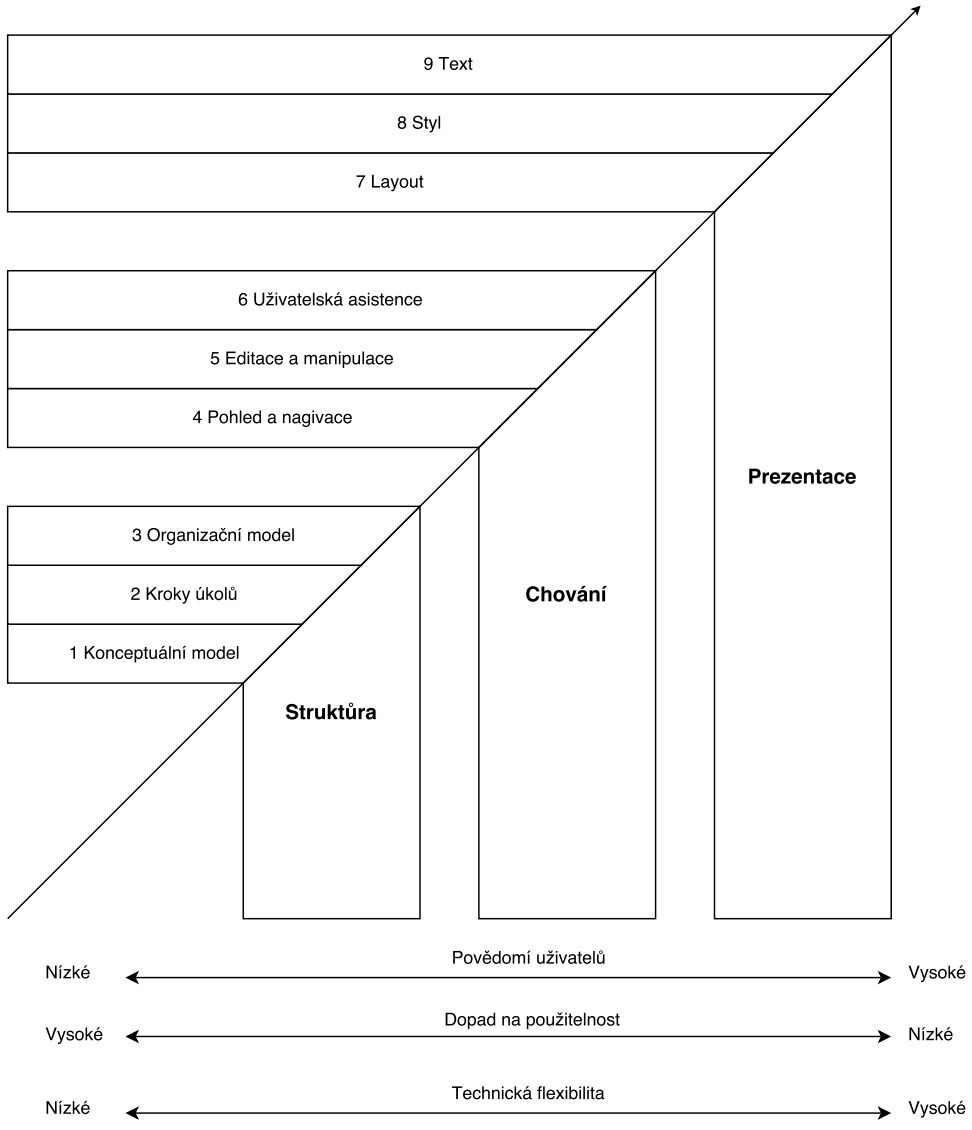
- **1 vrstva: konceptuální model** Popisuje vztah mezi UI a venkovním světem. Úkolem této vrstvy je zapůsobit na zkušenosti uživatele, aby rozpoznal jednoduché operace a predikoval funkcionalitu.
- **2 vrstva: kroky funkcí** Kroky jednotlivých funkcí k provedení dané funkce.
- **3 vrstva: organizační model** Tato vrstva je známa jako informační architektura a zaměřuje se na obsah a funkcionalitu.

2 stupeň: Chování Zaměřuje se více na interaktivní kvalitu, uživatelské očekávání a akce a systémové reakce. Je to více zaměřené na chování systému a jak uživatel interaguje se systémem.

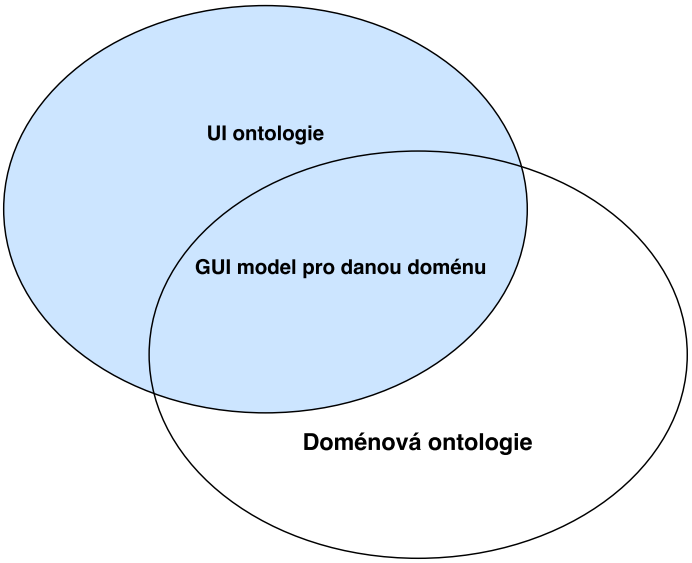
- **4 vrstva: Pohled a navigace** Akce jako například změna okna, přizpůsobení UI, seřazovaní dat nebo navigace mezi formuláři.
- **5 vrstva: Editace a manipulace** Popisuje jak uživatelské akce postihují informace uložené v databázi.
- **6 vrstva: Uživatelská asistence** Komponenty rozhraní, které zobrazují informace o stavu aplikace a aktuálně prováděné aktivitě.

3 stupeň: Prezentace Prezenční stupeň se uručuje jak uživatelské rozhraní vypadá a jaké se používají textové výrazy.

- **7 vrstva: Layout** Vrstva určuje jak jsou UI komponenty organizovány.
- **8 vrstva: Styl** Vizuální design aplikace.
- **9 vrstva: Text** Zahrnuje texty používané v aplikaci. Jak text samotná a tak i jazyk, který používá.



Obrázek 2.13: Univerzální model uživatelského rozhraní. Adaptováno z [57].



Obrázek 2.14: Ontologie při modelování GUI. Adaptováno dle [59].

2.5.1.1 Ontologie při modelování GUI

Při vytvíření UI pro danou doménu používáme stejné GUI ovládání prvky, ale také nesmíme zapomenout na doménu výsledného GUI. [59] Tuto situaci vyobrazuje obrázek 2.14. V práci [59] se můžeme dočíst o vytvořeném ontologickém frameworku pro modelování GUI. Jsou zde definovány úrovně abstrakce pro modelování UI, které můžou dobře posloužit při generování GUI. Tyto úrovně jsou:

Datové modelování Definice formátu dat, struktury dat a především domény dat a jejich hierarchie.

Vlastnosti uživatelské interakce Tato abstrakce je závislá na úrovni datového modelování a definuje uživatelskou interakci pro specifická data na specifické architektuře.

Grafické vlastnosti

Vlastnosti dané kontextem Specifikace obsahující technologické detaily a uživatelské informace.

2.5.2 DSL GUI

Doménově specificky jazyk (DSL) pro GUI je podmnožina popisných jazyků pro uživatelské rozhraní (UIDL). UIDL se skládá z vysoko úrovňového počítačového jazyka popisující charakteristiky UI s ohledem na zbytek interaktivní

aplikace.[60] Důvodem k vzniku UIDL byla potřeba vytvářet UI jako modul interaktivní aplikace než v podobě řádků kódů. Druhý důvodem byla přenositelnost mezi platformami, kde popis UI musí být v jiném jazyce než implementačním jazyce aplikace. V tomto ohledu se XML stal dobrým standardem, protože je snadno rozšířitelný, deklarativní a může být použít i neprogramátory. [60] Některých DSL založených na XML jsou například XAML[61], Intellij IDEA GUI designer form [62], XUL [63]. Skvělým přehledem a porovnání různých UIDL se můžeme dočíst v publikaci [60].

V práci [64] můžeme vidět vytvoření DSL jazyka Loa, který vznikl za účelem zjednodušení popisu definic pohledů a vazeb ve vytvořeném konceptuálním modelu. Konceptuální model vznikl dekompozicí GUI aplikace na čtyři ortogonální pohledy a vazby mezi pohledy. Přesněji je v práci popsána dekompozice vzoru MVC a MVC2, kde jeden ortogonální pohled je jedna část MVC.

Jiný pohled na DSL a GUI se můžeme dočíst v publikaci [65], kde je představen formální návrh metody pro vytváření DSL gramatik z již existující uživatelských rozhraní. Ale tento vzniklý DSL jazyk není určený pro GUI, ale pro doménu aplikace. Dále je zde řečena zajímavá myšlenka. „Pokud existuje aplikace pro specifickou doménu s GUI vytvořená z komponent a máme k dispozici reflexi spolu s možnosti určení struktury komponent, pak je možné navrhnout nástroj, který používá reflexi, a který může procházet GUI aplikaci a vytvořit návrh DSL z GUI.“ Dále také v publikaci ukazují metodu DEAL(Domain Extraction ALgorithm) (nejlépe posána zde [66]), která prochází GUI za účelem vytvoření doménových informací z GUI aplikace.

2.5.2.1 XML UIDL

Jak bylo v předchozích odstavcích řečeno tak většina UIDL jazyků jsou dialekty XML [67]. V této části si představíme ukázku XML UIDL a to pouze technologii XAML, protože všechny ostatní UIDL založené na XML jsou si dosti podobné a jen se liší v detailech nebo syntaxi, ale princip mají stejný. Popis UI pomocí XML nám dává řadu výhod oproti jiným typům UIDL (textový, zdrojový kód, ...). Některé z těchto výhod jsou například:

- snadná validace
- využití Xpath
- snadné pochopení syntaxe
- jednoduché počítačové zpracování.
- možná serializace GUI za běhu aplikace do XML
- snadná editace za běhu aplikace
- platformě a implementačně nezávislé

- existující nástroje pro editaci, validaci, atd.

Extension application markup language neboli XAML je značkovací jazyk pro popis grafické rozhraní v aplikacích společnosti Microsoft. [68] XAML je používaný v .NET od verze 3 a je používaný i Window presentation foundation (WPF) **??**. Jedna z výhod XAML je, že lze zkompilovat do BAML (binary application markup language). Níže je vidět jednoduchý příklad XAML souboru.

```
<Canvas xmlns="http://schemas.microsoft.com/client/2007"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml
            ">
        <TextBlock>Ahoj svete!</TextBlock>
</Canvas>
```

XAML obsahuje několik prvků syntaxe:

Object element Určuje typ komponenty (tlačítko, textový vstup, ...) a zapisuje se jako XML element, kde název je typ komponenty. Například:

```
<TextBox>This is a Text Box</TextBox>
```

Lze také doplňovat komponentu o atributy, které lze zapisovat jako klasický atribut XML elementu:

```
<Button Content = "Click␣Me" Height = "30" Width = "60"/
    >
```

Property element Atribut objekt elementu, ale zapsaný alternativní syntaxí.

```
<Button>
        <Button.Content>Click Me</Button.Content>
        <Button.Height>30</Button.Height>
        <Button.Width>60</Button.Width>
</Button>
```

Child element Objekt element obsažený v objekt elementu, který může obsahovat další objekt elementy, typicky se jedná o kontejnery.

```
<StackPanel Orientation = "Horizontal">
        <TextBlock Text = "Hello"/>
</StackPanel>
```

Více o XAML se lze dočíst na webových stránkách Microsoftu [61] nebo případně z praktického hlediska zde [69].

2.5.3 Generování GUI

Generování GUI a UI obecně lze rozdělit do tří kategorií podle typu zdroje informací pro generování UI. Tyto kategorie jsou: z UML, anotovaného kódu a z UIDL. Každý z těchto přístupů má své výhody a nevýhody. Ke každé kategorii si ukážeme existující případ, abychom si udělali představu o tom, jak funguje jednotlivé generování UI z dané kategorie.

2.5.3.1 Z UML

Při generování UI z UML diagramů se nejčastěji používá MDA přístup, protože jak již bylo popsáno v této práci v sekci 2.2.1, tak tento přístup obsahuje tři modely: CIM, PIM a PSM. Generování z UML používá model PIM, který ještě není platformě závislý, ale je více specifický než CIM. Příklad může být práce [70], kde z modelu UI popsaného v textové podobě jsou schopni vygenerovat GUI pro různé mobilní platformy (například Android, iOS nebo Windows phone). Podobná práce na stejné téma jen s trochu jinými kroky je popsána zde [71]. Obecně tento přístup odpovídá MBUID, který používá různé modely k vygenerování UI (task modely, doménové modely, prezentační modely, ...). Tento přístup má obvykle tři kroky:

1. Namodelování GUI v UML.
2. Transformace diagramů na XML nebo jiný vhodný formát pro generování.
3. Vygenerování GUI pro danou platformu.

2.5.4 Z anotovaného kódu

Druhý způsobem je generování z anotovaného zdrojového kódu, kde anotace kódu je pomocí dodatečných informací (metadata) k možnému vygenerování UI. Obecnou nevýhodou tohoto přístupu je, že UI je definované a separované od GUI designeru, což tento přístup řadí výsledné UI spíše mezi prototypové [72], lze tedy spíše použít pro otestování funkčnosti a nebo jako prototyp pro zákazníka v počátečních fázích vývoje. [73]

Některé problémy spojené s tímto přístupem jsou:

- Kód může být hůře čitelný kvůli anotovanému zdrojovému kódu.
- Nízká možnost přizpůsobitelnosti výsledného UI, ale lze částečně ovlivnit dodatečnámi parametry zadaných při vstupu (ale vyžaduje sofistikovanější generátor). [73]
- Aby šlo analyzovat anotavný zdrojový kód a šlo vygenerovat GUI bez pomocí programátora, je dobré použít tree-rewriting programovací jazyk.[72],

Tento přístup má obvykle dva kroky:

1. Analýza kódu k získání struktury programu.
2. Vygenerování konkrétního GUI dle získaných informací z předešlého bodu.

V práci [73] je příklad použití anotovaného kódu v jazyce Clojure, ze kterého je následně vygenerované funkcionální GUI, které se v tomto případě malinko liší v přístupu ovládání než klasické GUI.

Jiná diplomová práce [74] ukazuje možnost generování výsledného GUI z definice dat, kterou získá z webové služby ze serveru. Definice dat jsou metadata potřebná k vygenerování GUI. Výhodou tohoto přístupu je, že stačí změna pouze kódu na serveru a klient se přizpůsobí i za běhu aplikace. Samozřejmě obecné nevýhody tohoto přístupu stále platí.

A poslední zajímavou prací je [75], kde se generuje UI pro ontologické aplikace, opět pomocí anotovaného kódu a navíc i pomocí ontologických anotací.

2.5.4.1 Z UIDL

Poslední kategorií je generování UI z UIDL, které nejjednoduší a nejvíce používané. Nejvíce používané generování z UIDL je založené XML, jak již bylo zmíněno v části 2.5.2.1. V těchto UIDL je UI definováno na abstraktní úrovni a výsledné GUI je vygenerované pro konkrétní platformu pomocí platformě závislé knihovně. [72] Nevýhodou tohoto přístupu je kompletní specifikace popisného jazyka UI a nutná implementace generátoru na dané platformě, případně programovacím jazyku, který dané UIDL používá k vytvoření GUI.

Odkazy na některé takové vygenerované UI a zdrojové UIDL byly již popsané v části 2.5.2. Další ukázkou jiného přístupu v této kategorii je práce [76], kde je vytvořeno DSL jménem CQML. Zápis UI v CQML je následně kompilován do C++ a za běhu programu interpretován knihovnou CQML.

2.6 Shrnutí

Víme, že existuje několik druhů uživatelského rozhraní a vytvořit univerzální uživatelské rozhraní není lehký úkol, kvůli rozličnosti požadavků a zvyklostí na daných platformách. Také víme, že je dobré použít MBUID přístup s popisem UI ve formě XML, které nám dá řadu výhod při vývoji UI a usnadní pozdější editaci nebo znovupoužití některých částí UI. Při programování UI bychom měli nejlépe použít FRP paradigma, které nám umožní lepší kontrolu nad UI a lepší přehlednost kódu implementace obsluhy uživatelského rozhraní. S FRP lze pak zkombinovat několik návrhových vzorů pro implementaci interaktivní aplikace.

Kapitola 3

Existující řešení UUI

V této kapitole se podíváme na již existující řešení univerzálního uživatelského rozhraní, popřípadě grafického uživatelského rozhraní a na občasná zajímavá řešení pro GUI. Nejdříve se podíváme na klasické technologie v GUI například X Server. Poté se podíváme na některá zajímavá řešení GUI s FRP a nakonec se podíváme na technologie nejblíže k UUI.

3.1 Klasická UUI

S univerzální uživatelským rozhraní se setkáváme v podstatě každodenně a ani o tom nemusíme vědět. V této podkapitole se podíváme v krátkosti na dva nejvíce používané UUI systémy.

3.1.1 X Window System

X Window System (zkráceně X) je standard, který se skládá z X serveru, XLib knihovny a X klientů. Je vysoce konfigurovatelný, cross-platformní a postaven na architektuře klient-server. Používá se pro správu grafického uživatelského rozhraní na jednotlivých zařízeních. [77] Nesmíme si ale X zaměňovat s GUI knihovnou, protože neposkytuje žádné specifické komponenty, ale poskytuje pouze rozhraní pro vykreslování grafiky, získání uživatelského vstupu a podobně. Vzhled komponent výsledných komponent GUI aplikace musí určovat sama aplikace. [78] X je mezivrstva mezi softwarem a jádrem OS, která poskytuje aplikacím přístup ke grafickým možnostem a událostem ze vstupních zařízeních.

Jak již bylo zmíněno X window system se skládá z několika nezávislých komponent. Níže si je popíšeme.

X server Program v X Window System, který běží na lokalním zařízení a spravuje všechen přístup ke grafické kartě, zobrazovacím jednotkám a vstupním zařízením.

X Client program, který X server zobrazuje. Používá Xlib knihovnu, která překládá volání funkcí do zpráv protokolu X, které jsou odeslány X serveru.

X protokol Klasický protokol přenášený pomocí TCP. Používaný X na výměnu informací o GUI operacích mezi X server a X klienty. [77] Je založen na zprávách

XLib Knihovna poskytují rozhraní pro komunikaci s X serverem.

Komunikace mezi serverem a klientem probíhá následovně: [79]

1. Požadavek klienta na získání informací nebo akcí.
2. Serveru odešle odpověď na klienta.
3. Server posílá události(události ze vstupních zařízeních a události oken) všem klientům.
4. Server pošle chybovou zprávu, pokud klientský požadavek by nesprávný.

Obdobným systémem je Wayland, který se snaží nahradit X window system, který považuje za starý a složitý systém. Wayland obsahuje novou architekturu, ale stále podobnou X. Nevýhody, které X obsahuje jsou popsány zde [80] a o rozdílech mezi Wayland a X Window Systemem se můžeme dočíst zde [81].

3.1.2 Webové stránky

Jedním z nejtradičnějším „skoro“ univerzálním UI lze považovat klasické webové stránky, protože nám stačí na dané platformě implementovat webový prohlížeč, který nám umožní zobrazovat webové stránky, případně webové aplikace. Velkou nevýhodou je právě implementace prohlížeče, která obsahuje implementace zobrazení HTML a CSS, ale také implementaci skriptovacího jazyka Javascriptu.

3.2 Zajímavá řešení

3.2.1 Haxe a Haxe UI

Haxe je open source toolkit založený na moderním vysoko úrovňovém striktně typovaném jazyce. [82] Pomocí Haxe lze snadno vytvořit cross-platformní aplikace zaměřené na většinu používaným platforem. Nabízí cross-platformní kompilátor a standardní knihovnu, aby aplikace měla na všech platformách přístup k nativním možnostem dané platformy. Lze použít k vytváření různých typům aplikaci (hry, aplikace, nástroje, frameworky).

Haxe kompilátor překládá programovací jazyk Haxe do zdrojového kódu v nativním jazyce pro danou platformu nebo do výsledného binárního souboru. Kompilátor dokáže překládat kód na příklad do Javaskriptu, C++, PHP, Java, Python. Příklady použití Haxe a následné kompilace se lze dočíst v [82].

HaxeUI je open source knihovna, která používá Haxe toolkit a Open Flash Library. [83] Zaměřuje se na nejvíce používané platformy. HaxeUI lze psát v programovacím jazyce Haxe nebo v XML formátu. Výsledné UI je v nativním vzhledu pro danou platformu.

3.2.2 Haskell a MFlow

Dalším zajímavým řešení pro vytváření GUI, přesněji webového UI, je webový framework MFlow. Je naprogramován v Haskellu a snaží se zjednodušit programování webových aplikací a to tak, že přeměňuje inverzní kontrolu webové aplikace na tradiční nejvíce intuitivní imperativní programování. [84] MFlow poskytuje vysoko úrovňové DSL a také poskytuje widgety uživatelského rozhraní. Těmito widgety jsou například: dynamický widget, samoaktualizující se widget pomocí Ajaxu, kontejner widget, atd. Samotná stránka webové aplikace se pak skládá z těchto widgetů. Z programového hlediska je MFlow tvořena dvěma monádami. První je view monáda, ze které jsou vytvořeny všechny widgety. Druhou monádou je FlowM monáda, ve které je definována navigace aplikace jako sekvence. [85] Flow monáda obsahuje mechanizmus pro tracking a backtracking skrz celou navigaci.

3.2.3 Elm

Elm je funkcionální reaktivní programovací jazyk zaměřený na zjednodušení vytváření responzivního UI, především GUI webových aplikací. Elm nám poskytuje dvě hlavní výhody: jednoduchou deklarativní formu asynchronního FRP a čistě funkcionální grafický layout. [44] Obsahuje kompilátor, který vytváří Javascriptový kód s HTML tak, aby se dal co nejsnadněji a ihned nasadit. Elm je doménově specifický jazyk napsaný v Haskellu.

Používá diskrétní signály, které se používají k tomu, aby Elm zjistil jestli se hodnota signálu změnila nebo ne, tím se vyhne zbytečným výpočtům ostatních proměnných. [44] To znamená, že Elm FRP systém čeká až bude notifikován o změně hodnot v signálu. Tím se Elm řadí do implementace push-based modelů FRP, kde hodnoty jsou přepočítány jen když je změna vstupních hodnot.

Obsahuje dvě kategorie grafických primitiv: elementy a formy. Element je obdélníková komponenta o dané výšce a šířce, která může také obsahovat text, obrázky a video. Elementy lze zvětšovat, zmenšovat a také skládat dohromady a vytvářet komplexní elementy. Form je 2D grafický útvar (čáry, obrazce, text, obrázky, ...), který může mít texturu a barvu. Formy lze přesouvat, rotovat a měnit měřítko. Z těchto primitiv se pak skládají jednotlivá UI.

Základní vzor pro programování Elm aplikace je rozdělení do tří části: [46]

Model stav aplikace

Update jak se má aktualizovat stav

View jak se má zobrazit stav jako HTML

Toto rozdělení nám zajisti skvělou modularitu, znovupoužitelnost a snadné testování. Navíc tento vzor nám umožní vytvářet komplexní webové aplikace v modulárním stylu. [86] Pokud se pozorně podíváme na tento vzor, tak zjistíme, že se v podstatě jedná o klasický MVC nebo MVVM vzor.

Definice jazyka Elm a jak je napsán, je popsáno v práci Czaplickiho [87], autora jazyka Elm. Také zde navíc píše o reaktivním programování a souběžném FRP (concurrent FRP).

3.3 Aktuální řešení UUI

3.3.1 Advanced JSON Form

Advanced JSON From, jak z názvu vyplývá, tato technologie je založená na technologii JSON. Definuje dynamické formuláře, které můžou být vyplněny na jakékoliv platformě, která rozumí JSON formátu této technologie. [88] Všechny komponenty jsou definovány v JSON formátu, jak jejich validace, typ, tak i styl komponenty. Ukázka kódu je níže s korespondujícím formulářem 3.1. Důvodem proč byl použit JSON je, že oproti XML formátu má menší velikost a definice komponent jsou kratší. Nevýhodou této technologie vidím v přílišných informací v JSON souboru.

```
{
    "mainScreen":{
        "screenID":"Email",
        "screenDisplayArray":[
            {
                "localeCode":"en",
                "screenLabel":"4. Please enter a valid email
                   address here.",
                "screenHint":"Please do not leave blank."
            }
        ],
        "screenwidgetType":"textInput",
        "inputRequired":true,
        "widgetSchema":"
\"Email\": {
\"type\": \"string\",
\"pattern\":
    \"^[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,6}$\"
}"
    }
}
```

Obrázek 3.1: Ukázka formuláře Advanced JSON Form. Převzato z [89]

3.3.2 Universal Windows Platform

Universal Windows Platform (UWP) je technologie Microsoftu pro operační systém Windows 8.1 a vyšší. Tato technologie se používá pro vytváření univerzálního uživatelského rozhraní pro dané rodiny zařízení, na kterých aplikace bude výsledně nasazena.

Rodina zařízení je množina API, která je označena jménem a číslem verze. [90] Dále rodina určuje skupinu OS jaký na daných zařízení běží, specifikuje API, systémové charakteristiky a chování, které lze očekávat na zařízeních v této rodině. [91] Příklady rodiny zařízení je například: Desktop device family, Mobile device family, IoT device family. Při vývoji s UWP se vždy programuje pro jednu nebo více rodin zařízení.

Výsledné UI aplikace se pak automaticky adaptuje na dané zařízení. To znamená, že se přizpůsobí velikosti obrazovky, použije pro uživatelský vstup nejpřirozenější vstup dané platformy a podobně. Daná adaptace se řídí podle definice UI (použití adaptivních komponent). Dále používá ke svému běhu nativní API operačního systému Windows. Ukázku XAML dokumentu pro

tento typ technologie je představen v sekci 2.5.2.1.

Nevýhoda UWP je platformní zavilost (použitý jazyk pro programování aplikace, většinou C# nebo C++) a závislost na operačním systému Windows. Výhodou je rozdělení platforem do rodin zařízení, pro které daná aplikace vyvíjí a podle toho lze předem očekávat dané vlastnosti u zařízení.

3.3.3 UIML

V publikaci [92] je popsán jazyk pro uživatelské rozhraní, přesněji značkovací jazyk pro uživatelské rozhraní, který používá XML formát souboru. Pokud chceme na danou platformu použít UI popsané tímto jazykem, musíme použít popis stylu pro danou platformu k namapování na UI pro danou platformu. Interakce s UI je prováděna skrz události, které můžou být lokální (mezi elementy) nebo globální (mezi elementem a objektem reprezentující aplikační logiku).

XML soubor UI obsahuje několik částí. Ukázka UI je pod tímto popisem, ale nejdříve si popíše elementy, které jsou obsaženy v XML souboru: [92]

<description> Obsahuje seznam jednotlivých elementů, které společně tvoří rozhraní celé aplikace. Každý element musí mít jméno a třídu jako atribut. Jméno je unikátní v celém popisu.

<structure> Popisuje, který elementy z <description> jsou zobrazeny pro danou platformu a jak elementy jsou organizovány. Jednotlivé elementy jsou zde určeny podle jména elementů z <description>.

<data> Data, která jsou platformě nezávislá, ale aplikačně závislá. Všechny informace, které jsou uživateli prezentovány jsou v této části. Jednotlivá data jsou spojeny s elementy z <description> pomocí jména.

<style> Obsahuje popis stylu a data, která jsou závislá na zařízení. Tyto atributy jsou spojeny s daným elementem přes třídu z <description>.

<events> Popisuje události, které mohou nastat u jednotlivých elementů rozhraní. Tato sekce je platformě a i aplikačně závislá. Každá událost je definována těmito atributy: název, třída, zdroj a spouštěč. A obsahuje element akce, která spojuje element s metodou.

Listing 3.1: Ukázka UI, převzata z [92]

```
<?xml version="1.0" standalone="no"?>
<uiml version="2.0">
    <interface name="Figure5" class="MyApps">
        <description>
            <element name="Main" class="Main" />
            <element name="File" class="ActionGroup" />
            <element name="NewAction" class="ActionItem" />
            <element name="CloseAction" class="ActionItem" />
            <element name="QuitAction" class="ActionItem" />
        </description>
        <structure>
            <element name="Main">
                <element class="Bar">
                    <element name="File">
                        <element name="NewAction" />
                        <element name="CloseAction" />
                        <element class="Separator" />
                        <element name="QuitAction" />
                    </element>
                </element>
            </element>
        </structure>
        <data>
            <content name="Main">Example</content>
            <content name="File">File</content>
            <content name="NewAction">New</content>
            <content name="CloseAction">Close</content>
            <content name="QuitAction">Quit</content>
        </data>
        <style>
            <attribute class="Main" type="rendering" value="
                java.awt.Frame" />
            <attribute class="Main" type="size" value="100,80
                " />
            <attribute class="ActionItem" type="rendering"
                value="java.awt.MenuItem" />
            <attribute class="Separator" type="rendering"
                value="wrapper.MenuSeparator" />
            <attribute class="ActionGroup" type="rendering"
                value="java.awt.Menu" />
            <attribute class="Bar" type="rendering" value="
                java.awt.MenuBar" />
        </style>
        <events>
            <event name="SelectQuit" class="ActionSelect"
                source="QuitAction" trigger="Select">
                <action target="Main" method="exit" />
            </event>
```

```
        </events>
    </interface>
    <logic></logic>
</uiml>
```

Některé výhody tohot řešení jsou snadno vidět:

- Vzhled lze snadno změnit pomocí jednoduché editace popisu stylu.
- Jazyk je jednoduchý a platformě nezávislý (až na některé elementy).
- Snadné zpracování XML nástroji.
- Lze použít na několik platforem bez velkých změn.

3.3.4 UiGE Pipeline

V tezi Macíka [93] je popsána technologie UiGE Pipeline, která generuje kontextově-adaptivní uživatelské rozhraní pro heterogenní platformy. Tato technologie se skládá z UIP platformy, UIP serveru, UIP klienta a UIP protokolu. Dle dosavadního popisu je vidět, že technologie má architekturu klient-server. Výsledné vygenerované UI je přizpůsobeno kontextu použití na dané platformě a následně ještě optimalizované pro co nejefektivnější použití na dané platformě.

UIP platform Definuje komunikace mezi klienty a servery pomoci UIP protokolu. A obsahuje definici UIP server a UIP klienta.

UIP protokol Skládá se z event protokolu a UI protokolu. Event protokol definuje komunikaci pomocí event dokumentů, které se primárně používají pro komunikaci od UIP klienta na UIP server. UI protokol definuje komunikaci používající UI dokumenty. UI dokument popisuje data modely a popis UI.

UIP server Centrální komponenta UIP platformy, která je odpovědná za poskytování UI, dat a obsluhy událostí klientům.

UIP klient Klient, který generuje finální UI pro zobrazení uživatele a odesílá události serveru.

Popis abstraktního UI je ve formě XML. Abstraktní popis UI (AUI) načte server a na klienty odešle ve formě konkretní UI (CUI) na základě kontextu klienta, přes UI protokol. S UI zároveň server odešlé také modely (data reprezentována uživateli přes UI), které si spojí s vygenerovaným UI. Klient následně vygeneruje na základě CUI finální UI, které optimalizované pro použití na dané platformě. Klient následně odesílá události provedené na klientovi na UIP server přes event protokol.

Pro detailnější popis technologie odkazuji na tezi [93], kde je detailně popsána. Také dle popisu vidíme, že tato technologie souhlasí s našim zadáním práce, ale naše myšlenka řešení bude malinko jiná viz následující kapitola.

Kapitola 4

Koncept univerzálního uživatelského rozhraní

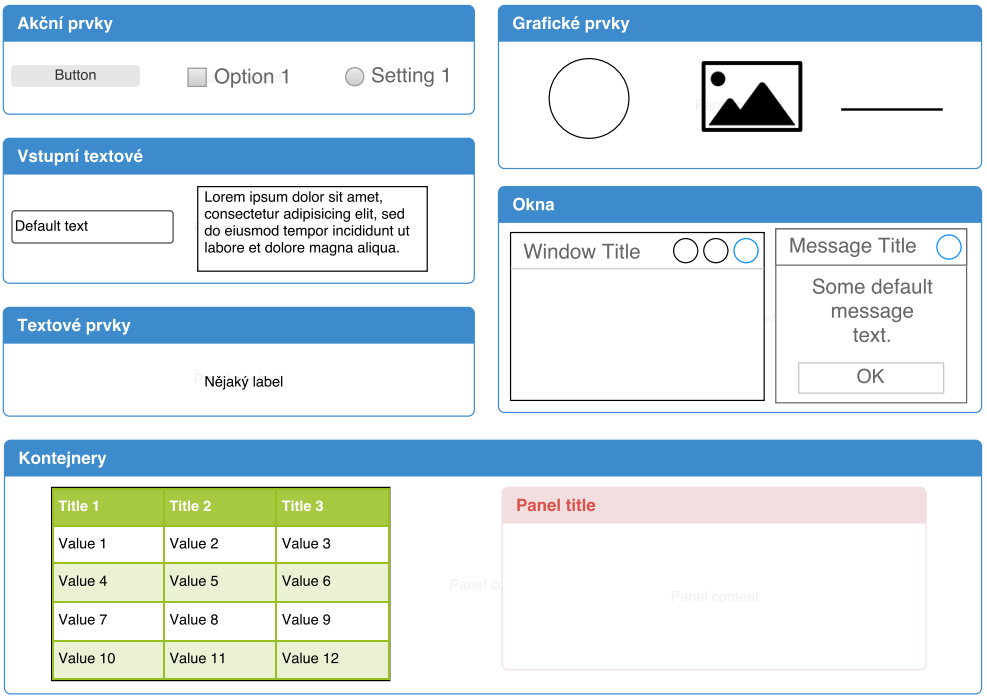
V této kapitole si představíme řešení univerzálního uživatelského rozhraní, které by mělo být nezávislé platformě i implementaci. Výsledné UI by mělo být zobrazeno dle knihovny na dané platformě a používat přirozené pvky pro danou platformu.

4.1 Popis konceptu a myšlenka

Našim úkolem je vytvořit univerzální uživatelské rozhraní (UUI), které bude nezávislé na platformě a implementaci. Tyto dvě podmínky naznačují, že UI bude muset být popsáno v jazyce, které je nezávislé na platformě a i implementaci. Jako vhodný jazyk pro tento účel dobře poslouží XML, kde jednotlivé elementy budou popsány pomocí vytvořeného DSL jazyka pro naše UUI. Abychom se vyhnuli zbytečné replikaci kódu na jiných platformách použijeme architekturu klient-server pro naše UUI. Kde jako server bude vystupovat implementována aplikace s naším UUI a klient bude interpret UUI, který bude na jiném zařízení než je server. Níže si blíže popíšeme detaily našeho konceptu.

4.2 Minimální koncept

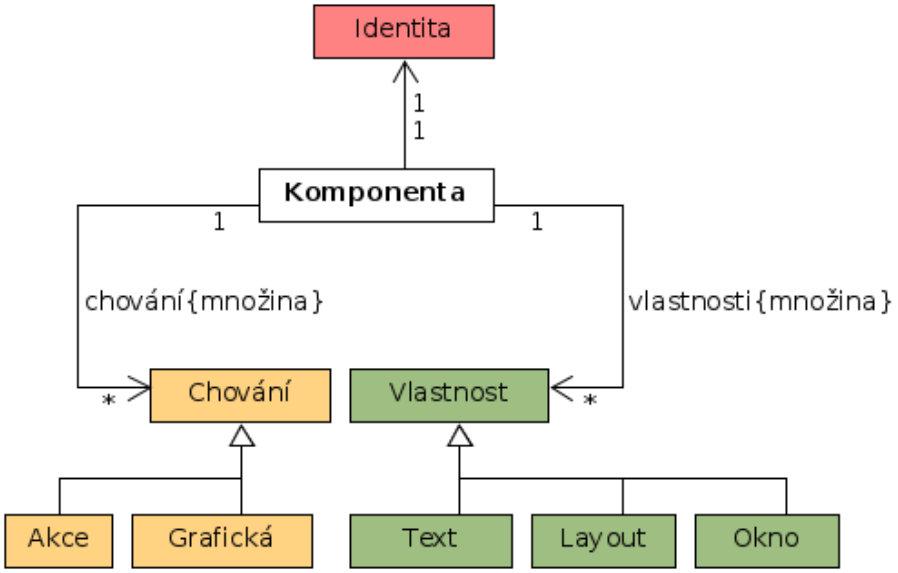
Koncept vytvoříme co možná nejjednodušší, aby byl snadno pochopitelný a rozšířitelný. Na obrázku 4.1 můžeme vidět některé základní grafické komponenty pro fungování GUI. Pokud si pozorně všimneme, tak pro vytvoření všech komponent na obrázku nám stačí tyto komponenty: akční tlačítko (libovolného vzhledu bez textu), text, grafická primitiva (čára, oblouk, atd.), okno a komponentu schopnou obsahovat jakoukoliv jinou komponentu (kontejnery). Dokonce tato množina nám stačí k vytvoření jakékoliv jiné komponenty. Například radio tlačítko se zakládá z akčního tlačítka ve tvaru kruhu a textu.



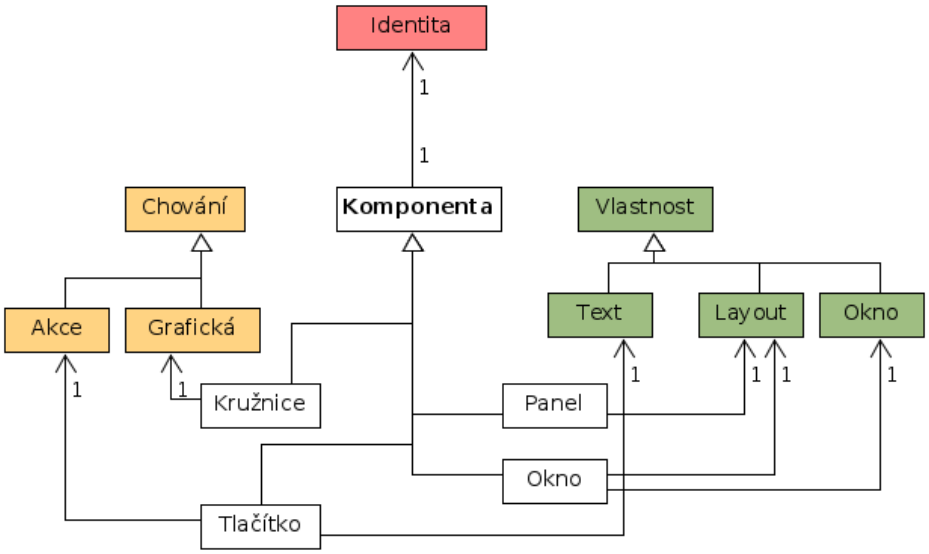
Obrázek 4.1: Základní komponenty GUI a jejich rozdělení do kategorií.

Nebo například combobox (rozbalovací menu) opět je zde akční tlačítko, text, kontejner a grafická primitiva (čtverec s výplní). Tuto množinu nazveme minimální množina.

Na obrázku 4.1 jsou také komponenty rozděleny do základních kategorií, které přesně odpovídají naší minimální množině. Nyní minimální množinu rozdělíme na dvě množiny a to na vlastnosti a chování. Kde vlastnosti jsou to co komponenty obsahují a je statické (text, pozice, velikost, atd.) a chování jsou reakce komponenty na vnější událost (akce, změna pozice, změna velikosti, atd.). Nyní nám zbývá poslední část komponenty a tou je identita, abychom rozlišili o jakou komponentu se jedná. A nyní jsme dospěli k definici komponenty. Komponenta je objekt, který obsahuje identitu, vlastnosti a chování (viz obrázek 4.2). A toto je minimální koncept. Všechny ostatní již existující prvky UI lze namapovat na komponentu. Například tlačítko: má svoji identitu (název, unikátní identifikátor), vlastnosti jako je text, velikost a pozice a chování, které jsou akce, změna pozice a získání focusu. Další popsané některé komponenty jsou zobrazeny na obrázku 4.3



Obrázek 4.2: Definice komponenty



Obrázek 4.3: Ukázka některých komponent popsané definici komponenty.

4.3 DSL a XML

Náš DSL jazyk postavíme na minimálním konceptu a formátu souboru XML, které je vhodné použít s uživatelským rozhraním [22]. Jednotlivé elementy v souboru mohou mít libovolné jméno. Identita a atributy komponenty se mapují na atributy elementu a chování je mapováno na podelementy elementu. Příklad použití DSL XML je vidět níže na tlačítku 4.1, panelu 4.2 a nebo formuláře 4.3.

Listing 4.1: Ukázka komponenty tlačítka.

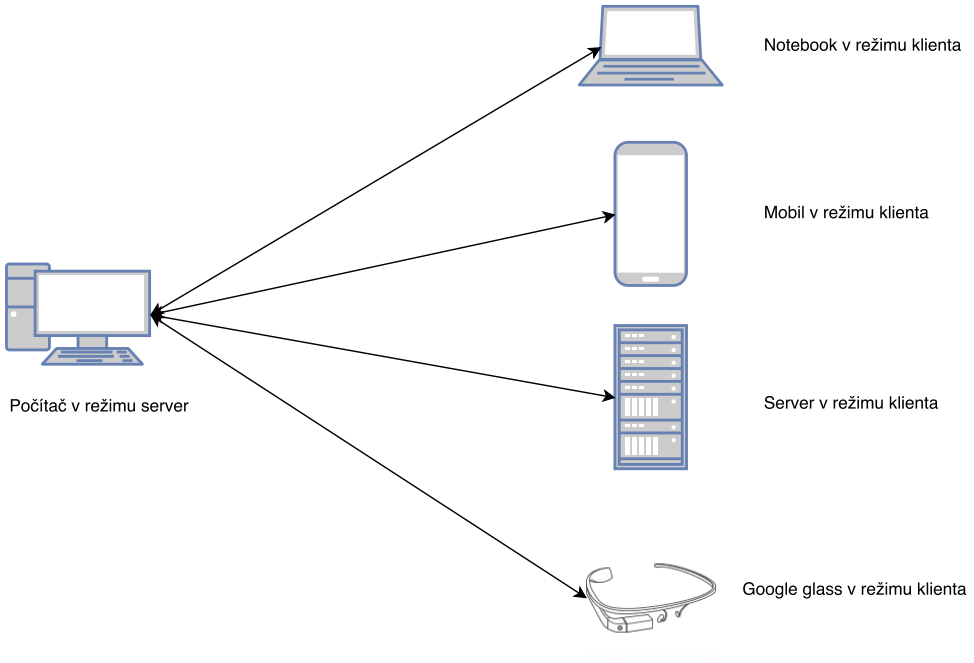
```
<button id="2" name="login" text="Login"/>
```

Listing 4.2: Ukázka komponenty panel obsahující tlačítko

```
<panel layout="horizontal">
        <components>
                <button id="2" name="login" text="Login"/>
        </components>
</panel>
```

Listing 4.3: Ukázka formuláře pro přihlášení do aplikace.

```
<?xml version="1.0" encoding="UTF-8"?>
<frame height="240" width="360" layout="vertical" text="Login
    ">
    <components>
        <panel layout="vertical">
            <components>
                <label alignment="center" text="Sign in"/>
                <label text="User name"/>
                <input type="text" name="username"/>
                <label text="Password"/>
                <input type="text" name="password"/>
                <button name="login" text="Login"/>
            </components>
        </panel>
        <panel layout="horizontal">
            <components>
                <label text="Status of System:"/>
                <label name="status" alignment="center"/>
            </components>
        </panel>
    </components>
</frame>
```



Obrázek 4.4: UUI: znázornění architektury klient-server

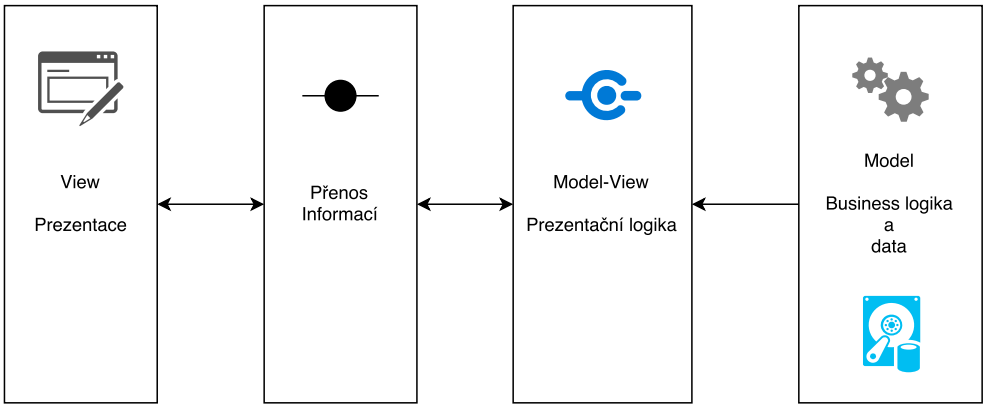
4.4 Architektura a komunikace

Jak již bylo napsáno v myšlence našeho konceptu UUI, použijeme architekturu klient server (zobrazena na obrázku 4.4). Klient a server budou spolu komunikovat přes UUI protokol. Dále použijeme funkcionální reaktivní programování, případně jen reaktivní programování a návrhový vzor MVVM. Reaktivní programování bude především použito pro obsluhování uživatelských interakcí s aplikací, jak na serveru tak na klientu. MVVM vzor se použije pro oddělení aplikační logiky, logiky uživatelské interakce a pohledu, který bude migrován na klienta (viz obrázek 4.5). Klient bude odesílat veškeré zachycené události serveru a server naopak bude klientu odesílat veškeré nastalé změny v UI. Tím implementuje naší vlastní reaktivní vrstvu přes UUI protokol pomocí push-based modelu, kdy se na klientu vždycky zobrazí vždy aktuální změny ihned po změně na serveru. Celý popis je znázorněn na obrázku 4.6.

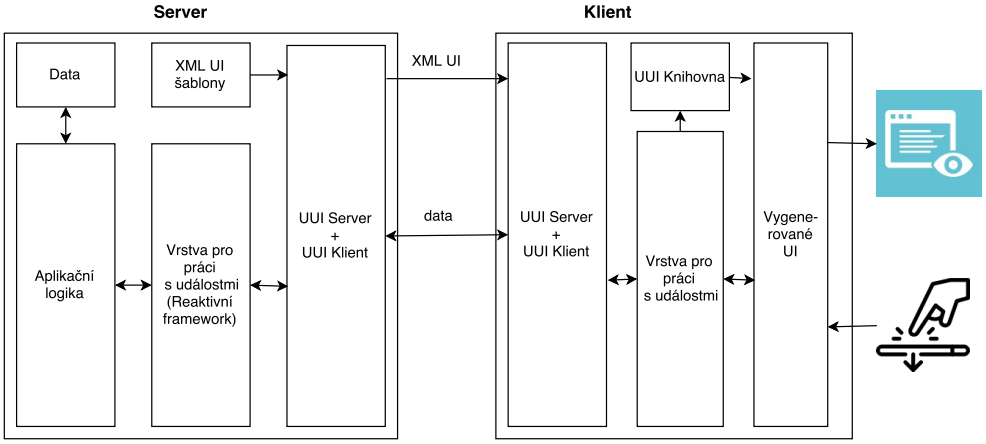
4.4.1 UUI Protokol

4.4.1.1 Zabezpečení

Protokol by měl především běžet po zabezpečeném přenosu HTTPS a obsahovat aspoň jednoduchou logiku přihlášení (například pomocí tokenu klienta). Zabezpečení je spíše implementační detail, ale zadefinujeme si ho. Klient a



Obrázek 4.5: UUI: použití vzoru MVVM



Obrázek 4.6: UUI: detailnější pohled na architekturu klient-server

server se spolu na začátku spojení spárují přes synchronizační kód, který si server sám vygeneruje. Klient odešle požadavek na server na přihlášení k odběru dat, kde obsahem požadavku bude synchronizační kód a adresa klienta. Server jako odpověď vrátí klientu unikátní identifikátor (token), který bude používat při veškeré komunikaci se serverem.

4.4.1.2 Komunikace

Komunikace mezi klientem a server je popsaná níže:

1. Klient se přihlásí k odběru na server.
2. Klient si získá aktuální zobrazené UI.

3. Klient čeká na události od serveru.
4. Server čeká na události od klienta.
5. Jakmile uživatel vytvoří událost na klientu, klient odešle tuto událost na server. Nebo jakmile server má změnu hodnoty v streamu, odešle ihned tuto změnu všem klientům.(Vždy ve formě hodnoty pro daný stream.)
6. Klient nebo server přijme daný požadavek na změnu.
7. Pokud klient se chce ukončit, odhlásí se od serveru. Jinak se pokračuje od bodu 3.

4.4.1.3 REST rozhraní

Komunikace mezi serverem a klientem definována pomocí REST rozhraní, které poskytuje základní koncové body pro poskytnutí nutných informací k zobrazení UI nebo implementace push based modelu. Důležité pro rozhraní je definice streamu. Stream v RESTovém rozhraní je dvojice: jméno a typ, který označuje typ streamu. Typ streamu jsou definované atributy a chování. Například typ streamu pro text je INPUT_TEXT a OUTPUT_TEXT. Nebo pro chování akce to je ACTION_EVENT.
RESTové rozhraní serveru:

GET /uui/form Odpověď je definice právě aktuálně zobrazené UI. Typ odpovědi je XML s DSL UUI.

GET /uui/form/name Vrátí definici UI s daným jménem. Typ odpovědi je XML s DSL UUI.

POST /uui/login Přihlásí klienta k serveru, v případě úspěšného přihlášení vrátí unikátní identifikátor pro označení klienta. Tělo požadavku musí obsahovat synchronizační kód a adresu klienta. Tělo požadavku a i odpověď je ve formátu JSON.

POST /uui/logout Odhlásí klienta od serveru. Tělo požadavku obsahuje unikátní identifikátor klienta a je ve formátu JSON.

GET /uui/streams Vrátí klientu všechny definice aktuálních streamů (proudů dat) v aktuální UI. Odpověď ve formátu JSON.

GET /uui/streams/name/type Vrátí poslední hodnotu streamu ve formátu JSON. Pouze v případě, že streamy obsahují tuto vlastnost

POST /uui/streams/name/type Pošle hodnotu na daný stream v aktuálním UI. V těle v požadavku musí být hodnota, která má být vložena do streamu. Tělo požadavku je ve formátu JSON.

RESTové rozhraní klienta:

POST /uui/form Pošle informaci o změně rozhraní. V těle rozhraní je nový formulář ve formátu XML.

POST /uui/streams/name/type Pošle hodnotu na daný stream. V těle v požadavku musí být hodnota, která má být vložena do streamu. Tělo požadavku je ve formátu JSON.

4.4.2 Knihovna zobrazení UI

Aby se na dané platformě dalo vygenerovat UI musí být zde implementována knihovna pro generování UI z DSL. Knihovna nutně nemusí implementovat všechny atributy a chování komponenty, pouze ji stačí implementovat prvky, které pro danou platformu mají „smysl“ a lze je zde implementovat/provést.

Kapitola 5

Implementace univerzálního uživatelského rozhraní

V této kapitole si ukážeme ukázkovou implementaci konceptu UUI na dvou vybraných platformách. Implementace bude následně demonstrována na jednoduché aplikaci, která je popsána v podkapitole Use case. Dále se pak podíváme popis implementací a screenshotů z aplikací. A nakonec si částečně otestuje ukázkovou implementací.

5.1 Platformy

Ukázková implementace konceptu UUI bude implementována na dvou platformách. První platformou je Java a druhá Ruby. Použitý reaktivní framework je ReactiveX [47], který má rozšíření právě na tyto platformy. Implementace v Javě SE bude určena primárně pro desktopy, takže bude obsahovat grafické uživatelské rozhraní. Implementace v Ruby bude určena pro zařízení bez grafického výstupu, to znamená pouze textová podoba UI.

5.2 Ukázková aplikace

Demonstrativní aplikace pro ukázkovou implementaci konceptu UUI bude použita jednoduchá ToDo aplikace. Vymezení funkčnosti a vzhledu je popsána níže.

5.2.1 Funkční a nefunkční požadavky

Funkční požadavky na aplikaci jsou následující:

1. Aplikace bude schopna vytvořit a smazat úkol.
2. Úkoly bude možné přiřazovat do kategorií.

3. Úkoly budou mít vlastnosti: název, prioritu a kategorii.
4. Do aplikace se bude muset uživatel přihlásit pomocí uživatelského jména a hesla.

Nefunkční požadavky na aplikaci

1. Veškeré informace bude aplikace ukládat do databáze, která bude uložena v paměti.
2. Aplikace bude používat koncept UUI.

5.2.2 Případy užití aplikace

Přihlášení do aplikace:

1. Uživatel spustí aplikaci.
2. Vyplní formulář pro přihlášení.
3. Stiskne na tlačítko přihlásit.
4. V případě správného zadání přihlašovacích údajů uživatele přihlásí do aplikace. V ostatních případech zobrazí neplatné přihlašovací údaje.

Přidání úkolu:

1. Na hlavní obrazovce aplikace uživatel klikne na „Přidat“.
2. Zobrazí se uživateli dialog pro přidání úkolů.
3. Uživatel vyplní formulář a stiskněte na tlačítko „Uložit“.
4. Aplikace uloží úkol do své databáze.

Smazání úkolu:

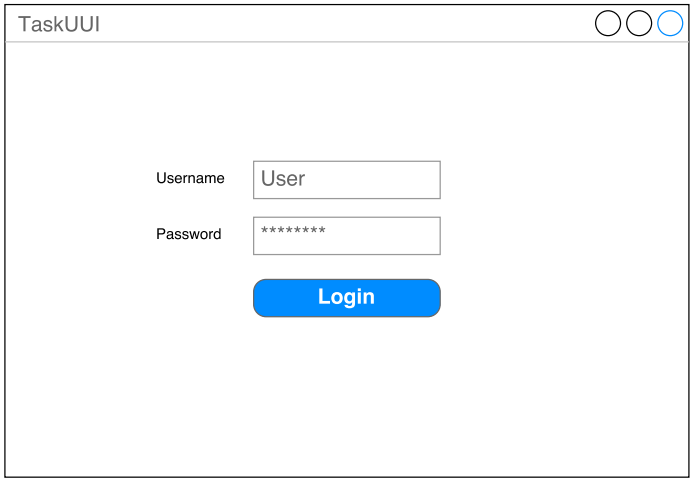
1. Na hlavní obrazovce uživatel označí úkol v tabulce a klikne na tlačítko „Odstranit“.
2. Aplikace odstraní úkol ze své databáze.

Přepnutí do stavu server:

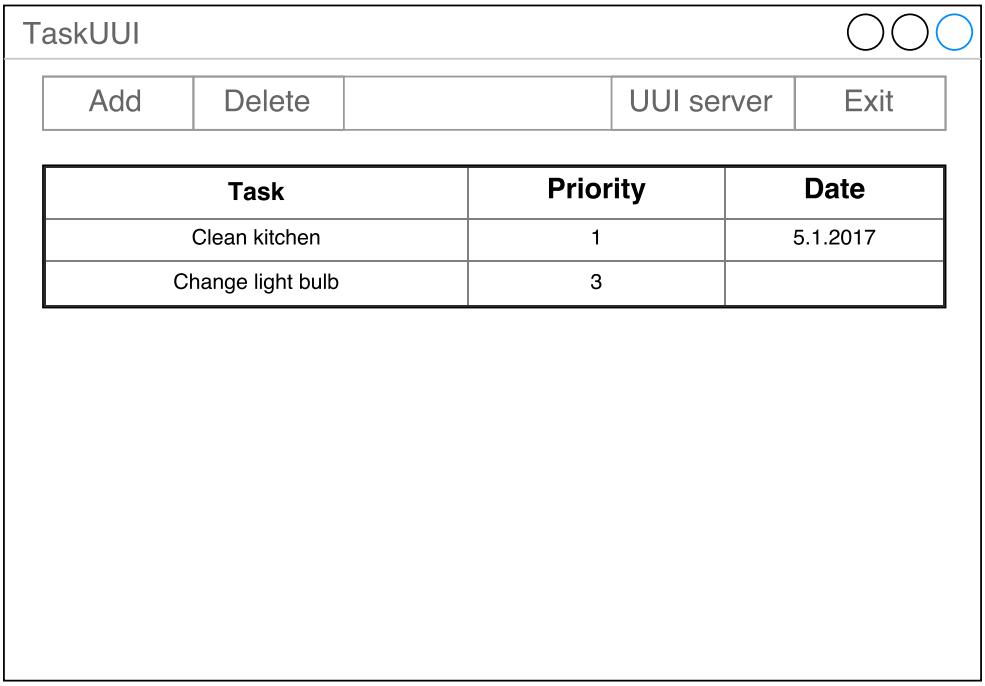
1. Na hlavní obrazovce uživatel klikne na tlačítko „Server UUI“.
2. Aplikace zobrazí obrazovku s tlačítky na start a vypnutí serveru.
3. Klient stiskne tlačítko „Start“ a aplikace spustí server.

5.2.3 Lo-Fi prototypy UI

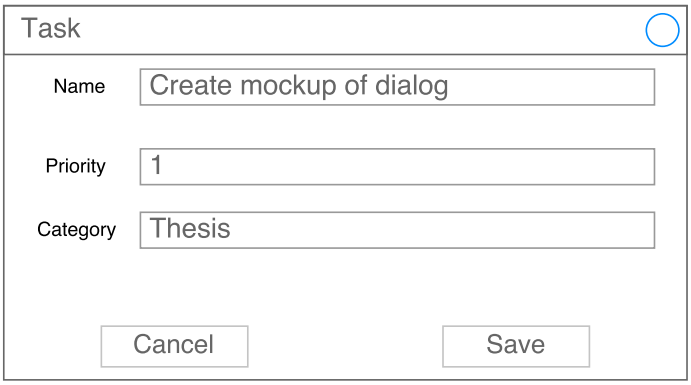
V této části jsou zobrazeny Lo-Fi prototypy formulářů aplikace, které mají být implementačně nezávislé a zobrazují design aplikace.



Obrázek 5.1: Lo-Fi prototyp ukázkové aplikace, formulář přihlášení.



Obrázek 5.2: Lo-Fi prototyp ukázkové aplikace, formulář hlavní obrazovky.



Obrázek 5.3: Lo-Fi prototyp ukázkové aplikace, formulář přidání a editace úkolu.

5.3 Popis implementace

Výsledná implementace na platformách jsou prototypy, které mají být ověřením konceptu. Ověření konceptu je zde provedeno implementací UUI knihoven, UUI Server a UUI Klienta pomocí ukázkové aplikace, která poběží na nezávislých a odlišných platformách.

5.3.0.1 Java

Implementace na platformě Java byla implementována ve verzi 1.8. Pro zjednodušení vývoje byl použit framework Spring, který obsahuje dependency injection a správu bean. Tím byl celý model implementován jako bean komponenty a spojení pomocí injection do view modelů. Pro databázi byla použita technologie H2, která běží v paměti zařízení. Na RESTové rozhraní byl použit embedded server Jetty. Pro zobrazení grafický komponent byla použita technologie Swing. Aplikace byla rozdělena do tří části podle vzoru MVVM.

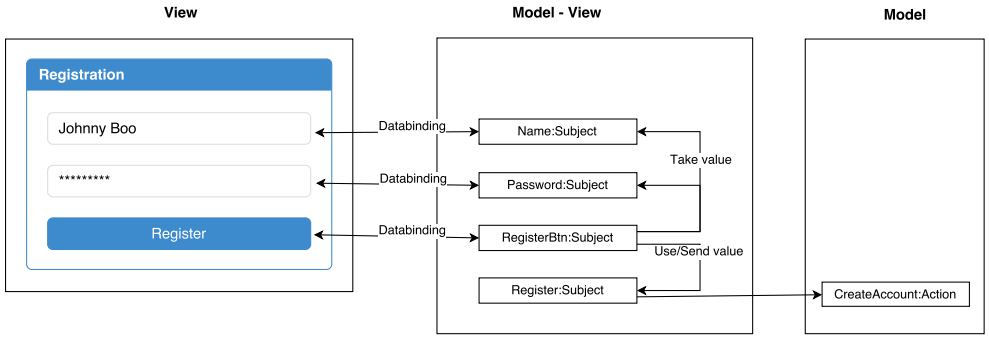
Model aplikace obsahuje logiku pro správu úkolu (uložení, načtení, smazání z databáze) a datové objekty. Nad celým modelem je napsána fasáda, která poskytuje celé rozhraní modelu v uceleném celku.

ViewModel obsahuje jednotlivé view modely pro každý formulář aplikace. Jednotlivý view model je implementovaný pomocí reaktivního programování, to znamená, že obsahuje pouze streamy pomocí kterých je spojen s vygenerovaným UI, případně s UUI Serverem. Dále obsahuje logiku pro formulář a jeho akce.

View jsou pouze XML soubory s jednotlivými formuláři, které jsou následně dál zpracovány UUI knihovnou.

Ukázka jaké bylo použité propojení MVVM modelu pomocí reaktivnéhé programování je znázorněno na obrázku 5.4.

Dále aplikace obsahuje implementaci UUI knihovny a UUI serveru. UUI knihovna obsahuje základní komponenty, vlastnosti a chování. Chování a vlastnosti jsou implementovány jako interface. Základní komponenty (tlačítko, textbox, label, panel, atd.) jsou mapovány na komponenty grafické knihovny Swing a implementují rozhraní vlastností a chování. Dále knihovna obsahuje interface pro View a ViewModel, které používá pro spojení view modelu s vygenerovaným UI. Rozhraní view obsahuje pouze metodu pro vyhledání komponenty podle jména. A rozhraní ViewModel obsahuje metodu vracení všech reaktivních spojení. Reaktivní spojení (třída ReactiveConnection) je definovaný jako stream, typ a jméno komponenty. Spojení View a ViewModel provádí třída ReactiveConnector, která vyhledá ve View komponentu definovanou jménem v reaktivním spojení a podle typu (akce, vstupní text, výstup text, . . .) připojí danou komponentu. UUI server poskytuje RESTové rozhraní konceptu UUI a také RESTového klienta pro posílání dat na UUI klienta.



Obrázek 5.4: Zobrazení jak se propojí MVVM pomocí FRP.

5.3.0.2 Ruby

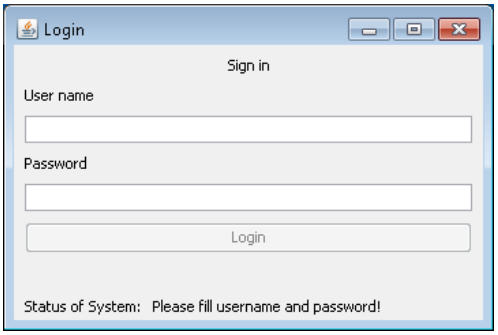
Implementace na platformě Ruby byla implementována v ruby verzi 2.0. Pro jednoduchý výpis formulářů na obrazovku byl použit gem Terminal-Table, aby se výsledné UI zobrazovalo v hezkém formátu na textový výstup terminálu. Pro jednoduchý klientský server UUI byl použit gem Sinatra s použitím serveru Thin. Jako reaktivní framework byla použita implementace ReactiveX na platformě Ruby. Tato implementace byla rozdělena pouze na dvě části a to UUI klientskou knihovnu a implementace textového klienta.

Textový klient obsahuje pouze několik metod pro získání vstupu a vykreslení na výstup. Hlavní místem aplikace je smyčka pro získání vstupu od uživatele, kde se čeká na jeho vstup. Vykreslení UI je vykreslováno pomocí reaktivních streamů. Aplikace obsahuje dva hlavní streamy. Jeden stream představuje změnu formuláře, která je přijímána ze serveru a druhý stream obsahuje změny provedené ze serveru. Jakákoliv změna v těchto dvou streamech zapříčiní vykreslení UI na textový výstup.

UUI klientská aplikace obsahuje implementaci základních UI prvků (tlačítko, label, textbox, panel, ...). Chování a vlastnosti komponenty jsou zde implementovány pomocí modulů, které obsahují očekávané proměnné u komponenty. Tyto moduly jsou následně vloženy do tříd, které reprezentují základní UI komponenty. Knihovna také obsahuje RESTového klienta pro získávání dat z UUI serveru a také jednoduchý UUI klientský server, který zde přijímá od UUI serveru změnu formuláře nebo hodnot ve streamech.

5.4 Ukázka

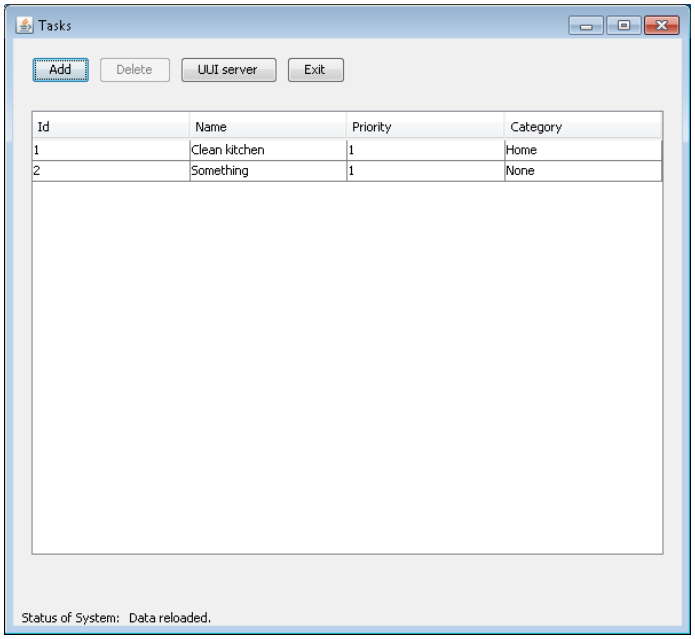
Nyní se ukážeme screenshoty z výsledné implementace v Javě a Ruby. Pro porovnání výsledků je vždy Java aplikace následována výstupem z Ruby aplikace.



Obrázek 5.5: Ukázková implementace v Javě, přihlašovací formulář

Listing 5.1: Ukázková implementace v Ruby, přihlašovací formulář

```
|-------------------------|
|          Login          |
|-------------------------|
|                         |
|   Sign in               |
|  User name              |
|  [I,T]                  |
|  Password               |
|  [I,T]                  |
|  [B]Login               |
|                         |
|  Status of System:      |
|-------------------------|
Type for [H] reload UI, but viewed data may be lost.
Type for [R] for refresh.
Type for [B] button actions.
Type for [IT] text input actions.
Type for [X] for exit.
```



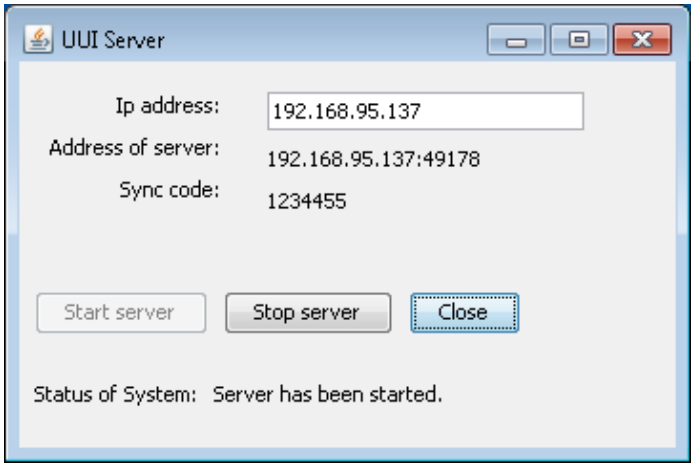
Obrázek 5.6: Ukázková implementace v Javě, hlavní formulář

Listing 5.2: Ukázková implementace v Ruby, hlavní formulář

```
|-------------------------------------------------------|
|                        Tasks                          |
|-------------------------------------------------------|
|                                                       |
|                                                       |
|    [B]Add   Delete   [B]UUI server   [B]Exit          |
|                                                       |
|    +---+---------------+----------+----------+       |
|     Id  Name            Priority   Category          |
|    +---+---------------+----------+----------+       |
|     1   Clean kitchen   1          Home              |
|     2   Something       1          None              |
|    +---+---------------+----------+----------+       |
|                                                       |
|   Status of System:  Data reloaded.                   |
|-------------------------------------------------------|
Type for [H] reload UI, but viewed data may be lost.
Type for [R] for refresh.
Type for [B] button actions.
Type for [IT] text input actions.
Type for [X] for exit.
B
------------------------
[0] add
[1] server
[2] exit
------------------------
Type [c] for cancel
Type number:
0
```

Listing 5.3: XML s DSL UUI hlavního formuláře

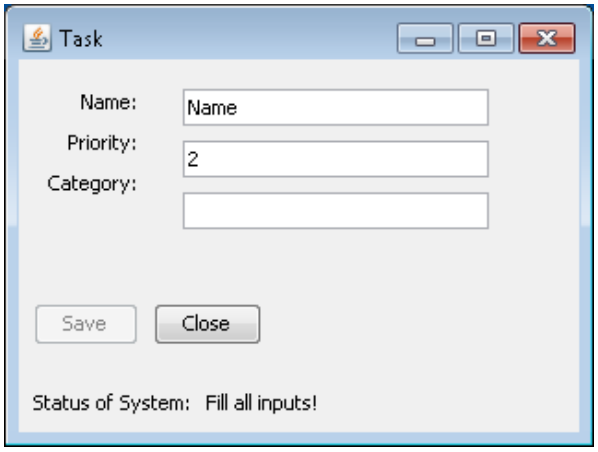
```
<?xml version="1.0" encoding="UTF-8"?>
<frame height="550" width="600" layout="vertical" text="Tasks
    ">
    <components>
        <panel layout="vertical">
            <components>
                <panel layout="horizontal">
                    <components>
                        <button name="add" text="Add"/>
                        <button name="delete" text="Delete"/>
                        <button name="server" text="UUI␣
                            server"/>
                        <space orientation="vertical"/>
                        <button name="exit" text="Exit"/>
                    </components>
                </panel>
                <panel layout="vertical">
                    <components>
                        <form name="table"/>
                    </components>
                </panel>
            </components>
        </panel>
        <panel layout="horizontal">
            <components>
                <label text="Status␣of␣System:"/>
                <label name="status" alignment="center"/>
            </components>
        </panel>
    </components>
</frame>
```



Obrázek 5.7: Ukázková implementace v Javě, formulář pro server

Listing 5.4: Ukázková implementace v Ruby, formulář pro server

```
|-------------------------------------------------------|
|                    UUI Server                         |
|-------------------------------------------------------|
|                                                       |
|                                                       |
|          Ip address:    [I,T]192.168.95.137           |
|    Address of server:    192.168.95.137:49178         |
|           Sync code:    1234455                       |
|                                                       |
|   Start server  [B]Stop server  [B]Close              |
|                                                       |
|   Status of System:  Server has been started.         |
|-------------------------------------------------------|
Type for [H] reload UI, but viewed data may be lost.
Type for [R] for refresh.
Type for [B] button actions.
Type for [IT] text input actions.
Type for [X] for exit.
B
----------------------
[0] stop
[1] close
----------------------
Type [c] for cancel
Type number:
1
```



Obrázek 5.8: Ukázková implementace v Javě, formulář pro úkol

Listing 5.5: Ukázková implementace v Ruby, formulář pro úkol

```
|-------------------------------------------------|
|                    Task                         |
|-------------------------------------------------|
|                                                 |
|                                                 |
|       Name:     [I,T]Name                       |
|   Priority:     [I,T]2                          |
|   Category:     [I,T]                           |
|                                                 |
|  Save  [B]Close                                 |
|                                                 |
|  Status of System:  Please fill inputs !        |
|-------------------------------------------------|
Type for [H] reload UI, but viewed data may be lost.
Type for [R] for refresh.
Type for [B] button actions.
Type for [IT] text input actions.
Type for [X] for exit.
it
------------------------
[0] name
[1] priority
[2] category
------------------------
Type [c] for cancel
Type number:
2
Type text:
Category
```

5.5 Testovaní

5.5.1 Uživatelské testování

5.5.1.1 Cílová skupina

Cílovou skupinou pro použití výsledných aplikací, by měli běžní uživatelé. Proto testování by mělo být provedeno běžnými uživateli. Na testování jsem proto vybral dva běžné uživatele (jeden méně zkušený a druhý hodně zkušený s prací s počítačem).

5.5.1.2 Test case

Vzhledem k tomu, že aplikace jednoduchá tak i testovací plán pro testera bude jednoduchý. Plán by měl obsahovat pokud možno všechny funkcionality aplikace.

- Testovací plán pro běžnou práci s aplikací.
  1. Spusťte aplikaci a přihlaste se.
  2. Přidejte úkol.
  3. Odstraňte úkol.
  4. Ukončete aplikaci.
- Spuštění hlavní aplikace jak server a připojení klienta.
  1. Spusťme aplikaci a přihlaste se.
  2. Přepnete aplikaci do režimu server.
  3. Spusťte na jiném zařízení klienta a připojte se k serveru.
  4. Následně proveďte předchozí testovací plán.
  5. Ukončete aplikaci.

5.5.1.3 Výsledky testování

Ani jeden z testerů neměl problém s provedení obou testovacích plánu. Což bylo pravděpodobné dáno tím, že aplikace byla jednoduchá pro testování. Věci, které testeři ocenili byla rychlá reakce systému a stále zobrazení informací a stavu aplikace a co právě vykonává. Jako velkou výhodu viděli v možnosti migrace uživatelského rozhraní na jiné zařízení. Časté poznámky, co by se dalo zlepšit, se často týkaly rozvržení komponent a o jejich možném grafickém zpracování.

5.5.2 Výkon

Výsledný výkon aplikace by velký. Aplikace reagovala ihned na změnu jakékoli hodnoty. Také bylo vidět, že reaktivní framework použili všechny dostupné prostředky CPU k tomu, aby přepočítal celý výpočetní strom proměnných. Také spojení mezi klientem a serverem probíhalo velice rychle. Ale jakmile se připojilo větší množství klientů (4 a více), začal systém být méně responzivní. Také se při měření objevilo časté odesílání stejných nebo nepatrných změn hodnot ve streamech. Tento problém lze snadno vyřešit tím, že se hodnoty z streamu nebudou ihned odesílat nýbrž seskupovat, případně se budou vytvářet okna z danou velikostí a vezme se vždy poslední hodnota v daném okně. Také by bylo dobré koncept ještě otestovat na složitější aplikaci, kde by dopad změn hodnot byl větší a přenos formulářu by obsahoval velké formuláře.

5.6 Zhodnoceni implementace

Implementace konceptu byla snadná, protože koncept je dostatečně jednoduchý, to znamená, že šel snadno namapovat na výslednou syntaxi pro danou platformu. Chování a vlastnosti komponenty byly namapované na rozhraní a výsledné základní komponenty implementovali tyto rozhraní. Rozhraní rozhraní a chování nám pomohli při generování CUI (identifikace vlastností a chování u komponent a jejich výsledné namapování hodnot) a také při propojování CUI a ViewModelu pomocí reaktivních konektorů.

Pokud bychom porovnali implementace na platformách mezi sebou, tak Ruby se snadněji implementovalo, protože má snadnější syntaxi. Ale jako velkou nevýhodu Ruby bych přisuzoval chybějící dokumentaci a chyby v implementaci reaktivního frameworku. Naopak u Java platformy byla velká výhoda existující knihovny pro mapování reaktivního frameworku na Swingovské komponenty co žádost usnadnilo práci při propojování vygenerovaného GUI.

Na ukázkách výsledných implementací je vidět, že koncept funguje a je zároveň „reaktivní“. Samozřejmě implementace lze ještě optimalizovat a zlepšit implementaci a přístup k implementaci byl ověření konceptu pomocí prototypu a to implementace splňují.

Závěr

Na začátku práce jsme si vytvořili teoretický základ pro náš koncept. Zjistili, že existuje několik programovacích paradigmat pro vývoj uživatelské interakce uživatelského rozhraní, kde ideálním kandidátem vyplynulo funkcionální reaktivní programování. Udělali si přehled v architektonických vzorech pro interaktivní aplikace, kde stále dominuje MVC a MVP vzor s jejich různými variacemi.

V kapitole konceptu univerzálního uživatelského rozhraní se nám podařilo vytvořit minimální množinu, která je potřeba k vytvoření všech komponent uživatelského rozhraní. Zadefinovali jsme si komponentu v kontextu našeho konceptu univerzálního uživatelského rozhraní, pomocí které jsem schopni implementovat jakoukoliv komponentu uživatelského rozhraní. Navrhli jednoduchý DSL jazyk popsaný formátem

V ukázkové implementaci jsme si ukázali jak je jednoduché implementovat minimální koncept jak v grafické podobě na Javovské platformě, tak v textové podobě na platformě Ruby. Také si implementací dokázali funkčnost celé architektury klient-server pro migraci uživatelského rozhraní mezi nezávislými platformami.

Ještě si odpovíme a otázky, které jsme si položili v úvodu práce. První otázka byla, jestli lze použit migraci uživatelského rozhraní na klasické aplikace. Náš vytvořený koncept univerzálního uživatelského rozhraní na tuto otázku sám odpovídá a to architekturou klient-server, ve kterém pomocí UUI protokolu se migruje uživatelské rozhraní na klienta. Druhou otázkou bylo, jestli jsme schopni výsledné migrované UI přizpůsobit dané platformě. V implementaci konceptu UUI je nám podařilo přizpůsobit klienta na textovou podobu, která měla výstup na terminál.

Budoucí vývoj UUI

Koncept našeho UUI se ukázal jako skvělý a solidní základ pro další vývoj. Ukázková implementace nám ukázala, že v konceptu jsou stále některé mezery,

například by bylo dobré zadefinovat kombinátory pro reaktivní streany v UUI server a UUI klientu, aby se každá drobná změna neposílal ihned a nezahltila UUI protokol. Dalším možným vývoj je implementace UUI interpreta na více platforem. Také by bylo vhodné výsledné vygenerované UI na dané platformě automaticky optimalizovat pro danou platformu, protože ne vždy designér UI je schopen pokrýt všechny interpretace UI na všech platformách.

Celkové zhodnocení

Celkově práce splnila cíle a zadání, které jsme si nastavili na začátku práce. Definovali jsme úspěšně koncept univerzální uživatelského rozhraní a úspěšně implementovali prototyp pro náš koncept UUI. Rozhodně téma univerzálního uživatelského rozhraní je zajímavé a v budoucnu můžeme očekávat další vývoj možný vývoj v tomto směru. Ať už se jedná jen migraci uživatelského rozhraní nebo o migraci celkové aplikace na jiné zařízení.

Literatura

[1] Fowler, M.: *Destilované UML*. Grada, myslíme v ... vydání, Červen 2009, ISBN 978-80-247-2062-3.

[2] Kosek, J.: *PHP a XML*. Grada, profesionál it vydání, Září 2009, ISBN 978-80-247-1116-4.

[3] Command-Line Interface. `https://en.wikipedia.org/w/index.php?title=Command-line_interface&oldid=696107358`, Prosinec 2015.

[4] Refreshable Braille Display. `https://en.wikipedia.org/w/index.php?title=Refreshable_braille_display&oldid=749818846`, Listopad 2016.

[5] GUI Definition. `http://www.linfo.org/gui.html`, 2004.

[6] Graphical User Interface. `https://en.wikipedia.org/w/index.php?title=Graphical_user_interface&oldid=696137698`, Prosinec 2015.

[7] Bačíková, M.; Porubän, J.; Lakatoš, D.: Introduction to Domain Analysis of Web User Interfaces. *Proceedings of the Eleventh International Conference on Informatics, INFORMATICS*, 2011: s. 115–120.

[8] Object-Oriented User Interface. `https://en.wikipedia.org/w/index.php?title=Object-oriented_user_interface&oldid=691896363`, Listopad 2015.

[9] User Interface Design. `https://en.wikipedia.org/w/index.php?title=User_interface_design&oldid=686000493`, Říjen 2015.

[10] Nielsen, J.: 10 Heuristics for User Interface Design: Article by Jakob Nielsen. `https://www.nngroup.com/articles/ten-usability-heuristics/`, 1995.

[11] Sollenberger, K.: 10 User Interface Design Fundamentals. http://blog.teamtreehouse.com/10-user-interface-design-fundamentals, 2012-08-07T11:21:02+00:00.

[12] User Interface Design Basics. http://www.usability.gov/what-and-why/user-interface-design.html, Květen 2014.

[13] Microsoft: Design Apps for the Windows Desktop - Windows App Development. https://developer.microsoft.com/en-us/windows/desktop/design, 2016.

[14] Apple: macOS Human Interface Guidelines: App Styles and Anatomy. https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/OSXHIGuidelines/index.html, 2016.

[15] Gnome: Pokyny K Rozhraní pro Člověka v GNOME. https://developer.gnome.org/hig/stable/, 2014.

[16] Model Driven Architecture. https://cs.wikipedia.org/w/index.php?title=Model_driven_architecture&oldid=12613303, Květen 2015.

[17] OMG: MDA. http://www.omg.org/mda/, 2016.

[18] Normalized Systems. https://en.wikipedia.org/w/index.php?title=Normalized_Systems&oldid=750557947, Listopad 2016.

[19] Oorts, G.; Huysmans, P.; Bruyn, P. D.; aj.: Building Evolvable Software Using Normalized Systems Theory: A Case Study. In *2014 47th Hawaii International Conference on System Sciences*, Leden 2014, s. 4760–4769, doi:10.1109/HICSS.2014.585, 00004.

[20] Meixner, G.; Calvary, G.; Coutaz, J.: Introduction to Model-Based User Interfaces. https://www.w3.org/TR/2014/NOTE-mbui-intro-20140107/, 2013.

[21] Delgado, A.; Estepa, A.; Troyano, J. A.; aj.: Reusing UI Elements with Model-Based User Interface Development. *International Journal of Human-Computer Studies*, ročník 86, Únor 2016: s. 48–62, ISSN 1071-5819, doi:10.1016/j.ijhcs.2015.09.003.

[22] Paternò, F.: Models for Universal Usability. In *Proceedings of the 15th Conference on L'Interaction Homme-Machine*, IHM '03, New York, NY, USA: ACM, 2003, ISBN 1-58113-803-2, s. 9–16, doi:10.1145/1063669.1063672.

[23] Event-Driven Programming. https://en.wikipedia.org/w/index.php?title=Event-driven_programming&oldid=693019367, Listopad 2015.

[24] Hansen, S.; Fossum, T.: Events Not Equal to GUIs. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, New York, NY, USA: ACM, 2004, ISBN 1-58113-798-2, s. 378–381, doi:10.1145/971300.971430.

[25] Parsons, D.: Event-Driven Programming. In *Foundational Java*, London: Springer London, 2012, ISBN 978-1-4471-2478-8 978-1-4471-2479-5, s. 417–464.

[26] Davis, H.: Understanding Events and Event-Driven Programming. In *Learn How to Program*, Berkeley, CA: Apress, 2004, ISBN 978-1-59059-113-0 978-1-4302-1113-6, s. 259–297.

[27] Publish/Subscribe. `https://msdn.microsoft.com/en-us/library/ff649664.aspx`, 2004.

[28] Bishop, J.: *C# 3.0 Design Patterns*. O'Reilly Media, 2007, ISBN 978-0-596-52773-0.

[29] Klang, J. B.: Reactive Programming vs. Reactive Systems. `https://www.oreilly.com/ideas/reactive-programming-vs-reactive-systems`, Prosinec 2016.

[30] Staltz, A.: The Introduction to Reactive Programming You've Been Missing. `https://gist.github.com/staltz/868e7e9bc2a7b8c1f754`, Únor 2016.

[31] Bainomugisha, E.; Carreton, A. L.; van Cutsem, T.; aj.: A Survey on Reactive Programming. *ACM Comput. Surv.*, ročník 45, č. 4, Srpen 2013: s. 52:1–52:34, ISSN 0360-0300, doi:10.1145/2501654.2501666.

[32] Reactive Programming. `https://en.wikipedia.org/w/index.php?title=Reactive_programming&oldid=688503129`, Listopad 2015.

[33] Pucella, R.: Reactive Programming in Standard ML. In *1998 International Conference on Computer Languages, 1998. Proceedings*, Květen 1998, s. 48–57, doi:10.1109/ICCL.1998.674156.

[34] Webber, K.: What Is Reactive Programming? `https://blog.redelastic.com/what-is-reactive-programming-bc9fa7f4a7fc`, Srpen 2014.

[35] Bonér, J.; Farley, D.; Kuhn, R.; aj.: The Reactive Manifesto. `http://www.reactivemanifesto.org/`, 2014.

[36] Maier, I.; Rompf, T.; Odersky, M.: Deprecating the Observer Pattern. Technická zpráva, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2010.

[37] Salvaneschi, G.; Margara, A.; Tamburrelli, G.: Reactive Programming: A Walkthrough. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, ročník 2, Květen 2015, s. 953–954, doi: 10.1109/ICSE.2015.303.

[38] Guizzardi, G.: Ontological Patterns, Anti-Patterns and Pattern Languages for Next-Generation Conceptual Modeling. In *Conceptual Modeling*, ročník 8824, editace E. Yu; G. Dobbie; M. Jarke; S. Purao, Cham: Springer International Publishing, 2014, ISBN 978-3-319-12205-2 978-3-319-12206-9, s. 13–27.

[39] Elliott, C.; Hudak, P.: Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, New York, NY, USA: ACM, 1997, ISBN 978-0-89791-918-0, s. 263–273, doi:10.1145/258948.258973.

[40] Cave, A.; Ferreira, F.; Panangaden, P.; aj.: Fair Reactive Programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, New York, NY, USA: ACM, 2014, ISBN 978-1-4503-2544-8, s. 361–372, doi:10.1145/2535838.2535881.

[41] Functional Reactive Programming. `https://en.wikipedia.org/w/index.php?title=Functional_reactive_programming&oldid=691494610`, Listopad 2015.

[42] Nilsson, H.; Courtney, A.; Peterson, J.: Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, New York, NY, USA: ACM, 2002, ISBN 1-58113-605-6, s. 51–64, doi:10.1145/581690.581695.

[43] Heinrichs, M.: MVC Is Dead – What Comes Next? 2016.

[44] Czaplicki, E.; Chong, S.: Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, New York, NY, USA: ACM, 2013, ISBN 978-1-4503-2014-6, s. 411–422, doi: 10.1145/2491956.2462161.

[45] Elliott, C. M.: Push-Pull Functional Reactive Programming. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-508-6, s. 25–36, doi: 10.1145/1596638.1596643.

[46] Elm. `http://elm-lang.org/`, 2016.

[47] ReactiveX. `http://reactivex.io/`, 2016.

[48] Blackheath, S.: SodiumFRP. https://github.com/SodiumFRP/sodium, 2016.

[49] Rao, S.: JAVA Ravin: What Are MVC1 and MVC2 Design Patterns ? 2012.

[50] Bernard, B.: Úvod Do Architektury MVC. 2009-05-07T00:00:00+01:00.

[51] Irwin, C.: MVC Is Dead, It's Time to MOVE On. https://cirw.in/blog/time-to-move-on, 2012.

[52] Bernard, B.: Prezentační Vzory Z Rodiny MVC. 2009-05-11T00:00:00+01:00.

[53] Microsoft: The Model-View-Presenter (MVP) Pattern. https://msdn.microsoft.com/en-us/library/ff649571.aspx, 2007.

[54] Model–view–viewmodel. https://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93viewmodel&oldid=747684642, Listopad 2016.

[55] Bernard, B.: Alternativy K MVC a Závěrečné Poznámky. 2009-05-15T00:00:00+01:00.

[56] Heinrichs, M.: MVC Is Dead – What Comes next? – Part 2. 2016.

[57] Baxley, B.: Universal Model of a User Interface. In *Proceedings of the 2003 Conference on Designing for User Experiences*, DUX '03, New York, NY, USA: ACM, 2003, ISBN 978-1-58113-728-6, s. 1–14, doi:10.1145/997078.997090.

[58] Hashim, R.; Abidin, S.: User Interface Development for a Computer-Based User Study: The Universal Model Approach. In *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, Duben 2010, s. 109–114, doi:10.1109/WAINA.2010.187.

[59] Shahzad, S.; Granitzer, M.; Helic, D.: Ontological Model Driven GUI Development: User Interface Ontology Approach. In *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, Listopad 2011, s. 214–218.

[60] Guerrero-Garcia, J.; Gonzalez-Calleros, J.; Vanderdonckt, J.; aj.: A Theoretical Survey of User Interface Description Languages: Preliminary Results. In *Web Congress, 2009. LA-WEB '09. Latin American*, Listopad 2009, s. 36–43, doi:10.1109/LA-WEB.2009.40.

[61] Microsoft: XAML Syntax In Detail. https://msdn.microsoft.com/en-us/library/ms788723(v=vs.110).aspx, 2016.

[62] GUI Designer Basics. https://www.jetbrains.com/help/idea/2016.3/gui-designer-basics.html, 2016.

[63] Mozilla: XUL. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL, 2014.

[64] Beaudoux, O.; Clavreul, M.; Blouin, A.: Binding Orthogonal Views for User Interface Design. In *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, VAO '13, New York, NY, USA: ACM, 2013, ISBN 978-1-4503-2070-2, s. 4:1–4:5, doi: 10.1145/2489861.2489865.

[65] Baciková, M.; Porubän, J.; Lakatos, D.: Defining Domain Language of Graphical User Interfaces. In *2nd Symposium on Languages, Applications and Technologies*, *OpenAccess Series in Informatics (OASIcs)*, ročník 29, editace J. P. Leal; R. Rocha; A. Simões, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, ISBN 978-3-939897-52-1, s. 187–202, doi:http://dx.doi.org/10.4230/OASIcs.SLATE.2013.187.

[66] Bačíková, M.; Porubän, J.: DSL-Driven Generation of Graphical User Interfaces. *Open Computer Science*, ročník 4, č. 4, 2014: s. 204–221, ISSN 2299-1093, doi:10.2478/s13537-014-0210-9.

[67] User Interface Markup Language. https://en.wikipedia.org/w/index.php?title=User_interface_markup_language&oldid=678862111, Září 2015.

[68] Extensible Application Markup Language. https://cs.wikipedia.org/w/index.php?title=Extensible_Application_Markup_Language&oldid=12642425, Červen 2015.

[69] tutorialspoint.com: XAML Tutorial. https://www.tutorialspoint.com/xaml/index.htm, 2016.

[70] Lachgar, M.; Abdali, A.: Generating Android Graphical User Interfaces Using an MDA Approach. In *Information Science and Technology (CIST), 2014 Third IEEE International Colloquium in*, Říjen 2014, s. 80–85, doi:10.1109/CIST.2014.7016598.

[71] Sabraoui, A.; Koutbi, M.; Khriss, I.: GUI Code Generation for Android Applications Using a MDA Approach. In *2012 International Conference on Complex Systems (ICCS)*, Listopad 2012, s. 1–6, doi:10.1109/ICoCS.2012.6458567.

[72] Jelinek, J.; Slavik, P.: GUI Generation from Annotated Source Code. In *Proceedings of the 3rd Annual Conference on Task Models and Diagrams*, TAMODIA '04, New York, NY, USA: ACM, 2004, ISBN 1-59593-000-0, s. 129–136, doi:10.1145/1045446.1045470.

[73] Podloucký, M.: *Automated GUI Generation for Functional Data Structures*. Diplomová práce, Univerzita Karlova v Praze, Květen 2015.

[74] Martin, T.: *Aspektově Orientovaný Vývoj Uživatelských Rozhraní pro Java SE Aplikace*. Diplomová práce, České Vysoké učení technické v Praze, Březen 2015.

[75] Jakub, D.: *Flexibilní Uživatelské Rozhraní pro Ontologické Aplikace*. Diplomová práce, České Vysoké učení technické v Praze, Červen 2014.

[76] Michal, H.: *Jazyk pro Definici Uživatelského Rozhraní*. Diplomová práce, České Vysoké učení technické v Praze, Květen 2015.

[77] An Introduction to X. `http://www.linfo.org/x.html`, 2006.

[78] X Window System. `https://cs.wikipedia.org/w/index.php?title=X_Window_System&oldid=14427483`, Prosinec 2016.

[79] X Server Definition. `http://www.linfo.org/x_server.html`, 2006.

[80] Wayland Architecture. `https://wayland.freedesktop.org/docs/html/ch03.html`, 2016.

[81] Wayland (Display Server Protocol). `https://en.wikipedia.org/w/index.php?title=Wayland_(display_server_protocol)&oldid=757206650`, Prosinec 2016.

[82] Foundation, H.: Haxe. `https://haxe.org/`, 2017.

[83] HaxeUI. `http://haxeui.org/`, 2017.

[84] Corona, A.: MFlow: Stateful, RESTful Web Framework. `https://hackage.haskell.org/package/MFlow`, 2015.

[85] Corona, A.: MFlow. `https://www.schoolofhaskell.com/user/agocorona/MFlow-tutoria`, 2013.

[86] Czaplicki, E.: Elm-Architecture-Tutorial. `https://github.com/evancz/elm-architecture-tutorial`, 2016.

[87] Czaplicki, E.: Elm: Concurrent FRP for Functional GUIs. 2012.

[88] Fizachi, D.: Advanced JSON Form Specification Chapter 1: Introduction - CodeProject. `http://www.codeproject.com/Articles/1102431/JSON-Form-Specification-Chapter-1-Introduction`, 2016.

[89] Fizachi, D.: Advanced JSON Form Specification Chapter 2: Input Widgets - CodeProject. `https://www.codeproject.com/articles/1105201/advanced-json-form-specification-chapter-2-input-widgets`, 2016.

[90] Microsoft: Intro to the Universal Windows Platform. https://msdn.microsoft.com/windows/uwp/get-started/universal-application-platform-guide, 2016.

[91] Microsoft: What's a Universal Windows Platform (UWP) App? https://msdn.microsoft.com/cs-cz/windows/uwp/get-started/whats-a-uwp, 2016.

[92] Abrams, M.; Phanouriou, C.; Batongbacal, A. L.; aj.: UIML: An Appliance-Independent XML User Interface Language. *Computer Networks*, ročník 31, č. 11–16, Květen 1999: s. 1695–1708, ISSN 1389-1286, doi:10.1016/S1389-1286(99)00044-4.

[93] Macík, M.: *Automatic User Interface Generation*. Dizertační práce, České Vysoké učení technické v Praze, Praha, 2016.

Příloha **A**

Seznam použitých zkratek

UI Uživatelsko rozhraní

CLI Příkazová řádka

GUI Grafické uživatelské rozhraní

OOUI Objektově orientované rozhraní

WUI Webové uživatelské rozhraní

EDP Event driven programming

MDP Message driven programming

RP Reaktivní programování

FRP Funkcionální reaktivní programování

MVC Model-view-controller

MOVE Models operations views events

MVP Model-view-presenter

MVVM Model-view-viewmodel

WPF Windows presentation foundation

MDA Model driven architecture

CIM Computation independent model

PIM Platform independent model

PSM Platform specific model

NS Normalizované systémy

SOC Separation of concerns

DvT Data version transparency

AvT Action version transparency

SoS Separation of states

MBUID Model based UI development

DSL Domain specific language

XML Extensible markup language

XAML Extensible application markup language

WPF Windows presentation foundation

BAML Binary application markup language

UIDL User interface description language

X X Window System

JSON Java script object notation

UWP Universal Windows Platform

OS Operační systém

API Aplication programming interface

UIML User interface markup language

REST Representational state transfer

Příloha B

Obsah přiloženého CD

- `source` ..... Zdrojové kdy implementace
  - `JavaSE` ..... Obsahuje implementaci na platformě Java
  - `RubyTextClient` ..... Obsahuje implementaci na platformě Ruby
  - `bin`
    - `Java-UUI-Server.jar` ..... Zkompilovaná implementace v Javě
    - `TextClient-0.1.0.gem` ... Zabalená implemetace v Ruby do gemu
- `text` ..... Text práce
  - `thesis.pdf` ..... Text práce ve formátu PDF
  - `source` ..... Obsahuje zdrojové soubory textu práce