FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Inteligentní osobní asistent pro OS Windows
Student:	Jindřich Kuzma
Vedoucí:	Ing. Stanislav Kuznetsov
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Vytvořte aplikaci pro OS Windows, která bude sloužit jako inteligentní osobní asistent(ka). Aplikace bude na základě zpráv v přirozeném jazyce (angličtina) spouštět naprogramované funkce.

1) Analyzujte existující řešení a možnosti knihoven.
2) Vyberte vhodné knihovny pro problematiku NLP a neuronových sítí.
3) Navrhněte aplikaci.
4) Implementujte aplikaci.
5) Otestujte aplikaci na základě případů užití.
6) Nasaďte aplikace na lokálním stroji.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 21. prosince 2017

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Inteligentní osobní asistent pro OS Windows

Jindřich Kuzma

Katedra softwarového inženýrství
Vedoucí práce: Ing. Stanislav Kuznetsov

10. května 2018

Poděkování

V první řadě bych rád poděkoval Ing. Stanislavu Kuznetsovi za vedení mé bakalářské práce. Dále své rodině, kamarádům, spolubydlícímu a přítelkyni za podporu během celého studia.

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, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 10. května 2018 ……

Odkaz na tuto práci

Kuzma, Jindřich. *Inteligentní osobní asistent pro OS Windows*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Cílem práce je navrhnout a implementovat aplikaci pro operační systém Windows, která bude sloužit jako inteligentní osobní asistent. Aplikace bude na základě zpráv v přirozeném jazyce spouštět naprogramované funkce. Práce je založena na použití vhodné knihovny pro zpracování přirozeného jazyka. Knihovna je integrována jako lokální Python server, který komunikuje s aplikací psanou v C++ přes HTTP. Implementovaná aplikace obsahuje 12 spustitelných funkcí, které tvoří asistenta, a je možné jednoduše přidat další, lze ji také spustit na lokálním stroji.

Klíčová slova NLP, inteligentní osobní asistent, Python, chatbot, C++

Abstract

The goal of this thesis is to design an application for the Windows operating system, which will serve as an Intelligent Personal Assistant. Based on message inputs in natural language, the application will launch corresponding programmed functions. This thesis is based on the proper use of a library for Natural Language Processing functions. It is integrated as a local Python server that communicates with the application written in C++ over HTTP. The implemented application contains 12 executable functions that make up the assistant and it is possible to run it on a local machine.

Keywords NLP, intelligent personal assistant, Python, chatbot, C++

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Úvod

V dnešní době je ve světě informatiky velice rozšířené nasazování umělé inteligence ke zpříjemnění lidského života. Zpracování přirozeného jazyka je jednou z nejpopulárnějších podoblastí umožňující strojům lépe komunikovat s uživatelem. Existuje nespočet využití sahajících od prodeje produktů online po ovládání přístrojů v domácnosti. Ve své práci se zaměřím na inteligentní osobní asistenty.

Inteligentní osobní asistent je aplikace zpřístupňující uživatelům přirozenější rozhraní k funkcím systému a nejrůznějším aplikacím. Rozvoji těchto asistentů napomáhá především využívání API[1] jednotlivých aplikací a také nedávný vývoj v oblasti umělých neuronových sítí. Asistenti jsou tak schopni zpracovat zprávu v přirozeném jazyce v rozumném čase a následně podle ní provést požadované úkony.

V práci se pokusím vlastního asistenta navrhnout a následně jej i implementovat spolu se základní sadou funkcí, které budou moci sloužit jako referenční funkce pro možné rozšíření. Teoretická část bude obsahovat přehled možných řešení a i několik již funkčních řešení. Pro operační systém Windows je tímto řešením asistentka Cortana, u které se ovšem objevují nepříjemnosti, kterým se u svého asistenta pokusím vyvarovat. Pro vytvoření asistenta nestačí pouze překládat přirozený jazyk, ale je nutné naprogramovat jednotlivé funkcionality a zvolit technologie, které mu umožní obstát v 21. století, například správné kódování zpráv.

[1] Application Programming Interface

Kapitola 1

Cíl práce

Práce je rozdělena do teoretické a praktické části. V teoretické části představím existující řešení inteligentních asistentů a knihovny, které se dají využít pro vytvoření vlastního asistenta. Následně vyberu nejvhodnější základ pro praktickou část. V té proběhne návrh architektury, která bude schopná dostatečné flexibility pro implementaci většiny funkcionalit, které by mohl asistent obsluhovat. Architektura bude následně použita ve vlastní implementaci a pomocí několika funkcí bude prokázána kvalita návrhu a funkčnost celého asistenta.

Část I

Teoretická část

Kapitola 2

Existující řešení

Osobní asistenti začali být populární na začátku druhé dekády 21. století s vydáním Siri. Tím se odstartovala malá revoluce, která postupně uchvátila velké technologické firmy. Ty po jejím vzoru začaly vydávat své vlastní asistenty. [1]

U všech asistentů zmíněných v příslušných kapitolách je rozmanitost funkcí, které ovládají velice rozsáhlá a proto lze uvažovat, že až na některé speciální příkazy zvládají obsloužit stejné povely. K tomuto závěru jsem došel po prozkoumání zdrojů uvedených u jednotlivých kapitol.

2.1 Alexa

Asistentka vyvíjená firmou Amazon s moderním využitím. Alexa totiž cílí na inteligentní domácnosti a lze ji pořídit v podobě zařízení Amazon Echo.

Echo je malá chytrá krabička obsahující reproduktor a připojení k wifi, jelikož Alexa existuje jako cloudové řešení. Všechna data se tudíž zpracovávají na serverech Amazonu a pro lepší ovládání vznikla aplikace pro mobilní telefony umožňující nastavení Echa a komunikaci s Alexou. [2], [3]

Toto ovšem není vše, co má pomoci Alexe prorazit na trhu. Spolu s asistentem přišlo i rozhraní pro vytváření vlastních funkcí. Takže není problém Alexu naučit nějaký ten trik či dva navíc. Také je vytvořena podpora pro integrování Amazon Alexa do produktů třetích stran v podobě Alexa Voice Service. [4]

Asistenta lze obohatit o další funkce pomocí služby Alexa Skills. Ta nabízí integraci funkcí třetích stran přímo do zařízení Echo. [3]

Z chytré krabičky se ovšem stává pouze drahý Bluetooth reproduktor pokud zrovna není v dosahu wifi síťe. Nemluvě o tom, že bez připojení k internetu není ani možné zařízení spárovat. [5]

2.2 Siri

Siri je asistent vyvíjený firmou Apple a dostupný pro produkty iPhone, iPad, Mac, Apple Watch, Apple TV a HomePod. Funguje na hlasových povelech a pro spuštění stačí vyslovit „Hey Siri“. [6]

Přesto, že Siri lze najít pouze na výrobcích značky Apple, podle [7] se pyšní 500 milióny aktivními uživateli. K tomuto faktu přispívá i uzavřenost operačního systému iOS a vylepšení, která se v systému iOS 11 postarala o zvýšení kvality a přidání mužské verze programu Siri. [7]

Po osobní zkušenosti bych vytkl závislost na internetovém připojení. Bez něj jsem se dočkal pouze odpovědi ať zkusím zprávu opakovat později.

2.3 Google Assistent

Věci lze dělat ve velkém a u Googlu to platí dvojnásob. Svého asistenta totiž protlačili skoro všude. Lze jej najít například v mobilních telefonech, chromeboocích či autech. [8]

Google také nabízí produkt podobný Alexe. Tímto produktem je Google Home [9], který je založený na cloudovém řešení, a tudíž také nefunguje bez přístupu k internetu. Ovšem u Google asistenta je tomu jinak. Ten je schopný si zachovat podmnožinu svých funkcionalit i bez přístupu na internet. [10]

Co se porovnání se Siri týče, je na tom Google asistent přeci jen o něco lépe, jak lze vidět ve videu. [11] Náskok má především ve schopnosti zpracovat jak řeč tak i příkazy jako takové, Siri je ovšem nutné přiznat kvalitnější zpracování informací.

2.4 Cortana

Cortana, původně postava z videohry „Halo“, byla představena firmou Microsoft jako inteligentní asistentka primárně určena pro systém Windows 10. Jedná se tedy o řešení, které má stejný účel jako aplikace navrhovaná v praktické části této práce. [12]

Stejně jako u předchozích řešení ani Microsoft nezůstal pouze u podpory jednoho zařízení. Podpora se postupně rozšířila do oblasti mobilních telefonů a míří i do chytrých domácností jako zařízení zvané INVOKE. [12], [13]

Cortana se zaměřuje na funkce z oblasti organizace informací, pomocné aplikace jako je slovník a samozřejmě nechybí ani podpora aplikací třetích stran. [14]

I v tomto případě se jedná o asistenta založeného na cloudovém řešení. S tím přichází i potřeba sdílet spoustu někdy i privátních informací s datovými centry Microsoftu. Tím ale také uživatelé přichází o možnost použití aplikace bez připojení k internetu. [15]

2.5 Bixby

Asistent od firmy Samsung se zaměřuje na integraci s telefonem Samsung Galaxy S9. Jedná se o hlasové ovládání funkcí telefonu. Podporovanými jazyky jsou angličtina, čínština a korejština, ale plánuje se rozšíření o další. [16]

Samsung má s Bixbym velké plány a chce jej doplnit o možnost komunikace s ostatními zařízeními. Nyní již zvládá přehrávat videa na televizi. Hlavním záměrem je, aby se asistent přizpůsoboval uživateli. [17]

Hlasem lze ovládat i aplikace třetích stran, které se integrují s asistentem. Ovšem opět je zde potřeba aktivní připojení k internetu k jeho fungování. [16], [17]

2.6 Mark I

Opensource řešení od firmy Mycroft. Asistent je primárně určen k fungování na zařízení využívající Raspberry Pi, které si lze koupit jako komplet a je také možné si jej doma sestavit ze správných součástek. [18]

Asistent je dostupný i pro jiné linuxové distribuce a experimentálně je možné mít jej i v androidu. Podporovaným jazykem je prozatím pouze angličtina. [19]

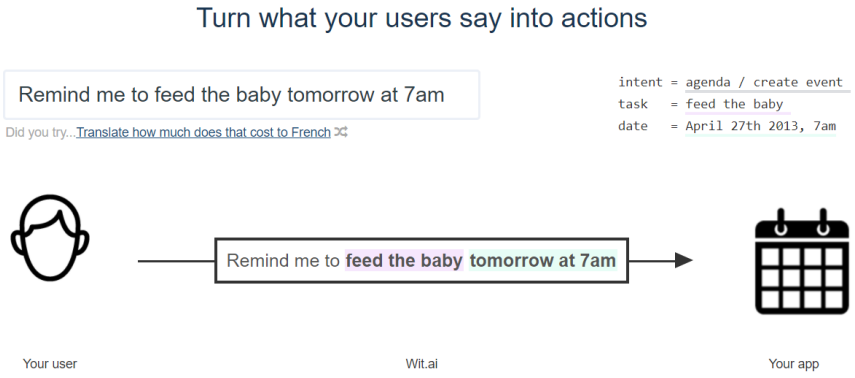
Zásluhou toho, že Mark I je opensource, je možné jej používat bez internetového připojení, bohužel tím ale uživatel ztratí velké množství funkcí a navíc tato verze není oficiálně podporovaná. Každý má ale možnost si asistenta upravit, a tím je mezi zmíněnými řešeními jedinečný. [20]

Kapitola 3

Existující knihovny

Zpracovávat zprávu od uživatele lze několika způsoby. Nejsnadnější cestou je pomocí podmínek vyhledávat ve zprávách od uživatele slova a podle nich spouštět příkazy. Tento způsob je ale neudržitelný a hodí se pouze pro dočasnou substituci za složitější řešení.

Rozvoj algoritmizace v oblasti strojového učení přinesl několik nových řešení. Pokrok se nezastavil jen u vytváření algoritmů, ale pokračoval dále a dnes jsou k dispozici knihovny a služby, které zpracovávají zprávy v přirozeném jazyce v rámci několika sekund a pro jejich konfiguraci stačí vyplnit konfigurační soubor. Jednoduchý příklad, jak daleko abstrakce zvládne zajít lze vidět na obrázku 3.1. Souhrnně jsou tyto technologie označovány pojmem NLP[2].



Obrázek 3.1: Příklad zpracování zpráv pomocí NLU z [21]

Na základě zjištěných informací z dokumentací následujících knihoven je zřetelné, že všechny dále uvedené knihovny lze použít s mírnými odchylkami

[2]Natural Language Processing

stejným způsobem. Co se funkčností týče, základní funkce jsou stejné a jsou jimi schopnost rozeznat entity a záměr z uživatelovy zprávy (pro lepší představu lze uvést, že zpráva „Ahoj.“ znázorňuje záměr „pozdrav“). Tudíž popis bude zaměřen na rozdílnosti.

3.1 Dialogflow

Dříve známé jako API.AI, později převzato Googlem. Jedná se o službu založenou na vytváření agentů. Pro agenty se definují záměry a entity určené k rozeznání, lze přidat i akce, které se mají spustit při určitém záměru. [22]

Dialogflow je možné integrovat do jiných aplikací, jako třeba Skype nebo Facebook Messenger. Integrace umožňuje definovat odpovědi k daným zprávám uživatele. [23]

Použití rozpoznávání hlasu je možné, ovšem při více jak 15 000 požadavků za měsíc je nutné přejít na placenou verzi. U této verze se platí drobná částka za každý požadavek, ale součástí je i technická podpora, která u bezplatné varianty není k dispozici. [24]

Další zdroj informací [25].

3.2 Luis

Knihovna vyvíjená firmou Microsoft. Jedná se o cloudové řešení dostupné pod platformou Azure. Komunikace probíhá prostřednictvím API, které umožňuje vše od vytváření nových projektů po samotné zpracování zpráv. [26], [27]

Prostřednictvím Azure je možné přidat i zpracování řeči. Dále je také přidán Bot Framework, jenž je určen k vytváření interaktivních aplikací využívajících konverzaci k obsluze. Program takto vytvořený ale musí být spuštěn na již zmíněné cloudové službě. [28], [29]

Služba je do 10 000 požadavků zdarma, následně je zpoplatněna. V době psaní této práce je Luis schopná podporovat 12 jazyků s rozdílnou kvalitou zpracování. [27], [30]

Další zdroj informací [31].

3.3 Adapt Mycroft

Adapt Mycroft je open source knihovna určená k běhu na malých zařízeních jako jsou IoT. Knihovna vznikla pod záštitou firmy, co vytvořila asistenta Mark I. [32]

Adapt je převážně určen pro jednoduší zařízení a aplikace, u kterých je předpokládané využití menšího množství funkcí. Hlavní výhodou oproti řešením využívajících cloud je nezávislost na internetovém připojení, protože knihovnu lze spustit lokálně. [32]

Vše je licencováno pod licencí Apache, a tudíž lze zdrojový kód libovolně upravovat. Díky použitému programovacímu jazyku funguje knihovna na každém zařízení, obsahujícím interpreter pro Python 2.7. [32]

Další zdroj informací [32].

3.4 Wit.ai

Jednoduché a efektivní řešení od Facebooku. Jedná se o online službu, ke které lze přistoupit pomocí HTTP[3] požadavků. K těmto požadavkům je nutná autentizace. [33], [34]

Zpracování zpráv probíhá stejně jako u předešlých knihoven na základě definování záměru a entit pomocí větných příkladů. Wit přichází s množinou předem definovaných entit. [35]

Využít knihovnu lze bezplatně bez jakéhokoliv omezení na četnost dotazů. Kromě zpracování zpráv také obsahuje zpracování hlasu. Jazyková podpora obsahuje přes 70 jazyků. [36]

Další zdroj informací [21].

3.5 Rasa

Další open source knihovna, která přichází ve dvou variantách. Obě varianty využívají Python a je možné je spouštět lokálně, což odstraňuje nutnost internetového připojení. [37], [38]

Obě řešení jsou na sebe napojená. RASA NLU[4] zpracovává zprávy v přirozeném jazyce do formátu, kterému rozumí počítač (v tomto případě JSON). RASA CORE se zaměřuje na zpracování dat které vzniknou v NLU části a vytvoření odpovědi pro uživatele.Jednoduché znázornění interakce NLU a CORE částí lze vidět na obrázku 3.2. [39], [38]

Další zdroj informací [40].

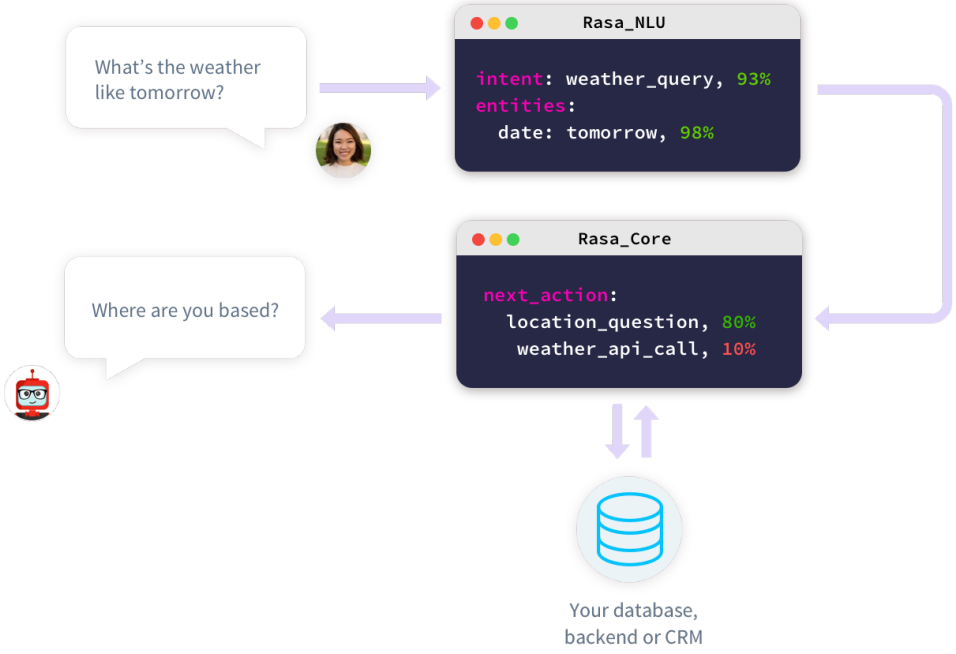
3.5.1 NLU

RASA NLU přináší možnost zpracovávat zprávy bez jakékoliv závislosti na okolí. Velkou výhodou je bohatá možnost konfigurace. Lze si vybrat rozdílné implementace algoritmů pro strojové učení. Podpora různých jazyků se pohybuje okolo 12, ale každý si může přidat vlastní podporu pro libovolný jazyk. [41], [42]

Celou knihovnu lze zabalit do HTTP serveru, který pomocí API odpovídá na požadavky. Server lze spustit lokálně a interagovat s ním může jakýkoliv jiný jazyk podporující HTTP. V serverové podobě může spoustu lidí oslovit možnost emulace jiných knihoven, kdy odpovědi mají stejnou podobu jako

[3] Hypertext Transfer Protocol

[4] Natural Language Understanding



Obrázek 3.2: Zjednodušené znázornění jak probíhá spolupráce RASA NLU a RASA CORE z [39]

by byly přímo od Wit.ai, Dialogflow nebo Luis. Tato možnost může velice usnadnit případný přechod z těchto platforem na RASA NLU. [43]

Další zdroj informací [38].

3.5.2 Core

Funkce CORE knihovny jsou určeny k obsluze rozsáhle konverzace mezi uživatelem a strojem. Programátor připraví jednoduché příběhy, které definují tok konverzace. Příkladem této konverzace může být objednávání zboží na e-shopu pomocí chatbota. Hlavním přínosem je efektivní zpracování jak dobré cesty (komunikace jde podle plánu), tak i odchylky od této cesty, těmi lze uvažovat například zadání dodatečných informací (množství objednaného zboží v případě e-shopu). Tento přístup je preferovaný především proto, že slouží jako náhrada za masivní konstrukce z podmíněných volání vytvářejících stavový prostor. [39]

Model pro konfiguraci lze vytvářet interaktivně přímo používáním knihovny. Ke svému fungování využívá NLU část určenou k zpracování přirozeného jazyka (lze využít RASA NLU, LUIS, Diaogflow a Wit.ai). Po zpracování a zís-

kání záměru ze zprávy zavolá RASA CORE příslušnou funkci podle toho, jak je nastaven model. [39], [44]

Další zdroj informací [37].

3.6 Snips NLU

Knihovně velice podobná k RASA NLU a také psaná v jazyce Python. Hlavním rozdílem je, že nenabízí spuštění v podobě HTTP serveru a podle mého názoru disponuje horší dokumentací. Aktuální podpora obsahuje 6 jazyků. [45]

Další zdroj informací [45].

Kapitola 4

Technologie

4.1 Programovací jazyk

Když dojde na implementaci asistenta, zvolil jsem C++, Python a Javu jako nejvíce relevantní jazyky. K tomuto rozhodnutí mě vedlo několik důvodů. C++ jsem vybral, protože má velice blízko k systémové úrovni. Python a Javu jsem vybral, jelikož obsahují rozsáhlé knihovny pro umělou inteligenci.

4.1.1 C++

C++ je nízkoúrovňový a velice komplexní jazyk. Programátor má na starost zprávu všech zdrojů, a proto často dochází k chybám. Jako kontrast k těmto obtížím je ale C++ velice rychlý a efektivní jazyk. Hlavní využití tedy najde především v oblasti vývoje videoher, zde je rychlost potřeba pro práci s 3D grafikou a také v oblasti desktopových aplikací. Jako nemalý plus lze považovat velkou komunitu, díky které se může programátor poprat s nešvary jazyka lépe.[46]

4.1.2 Python

Jako jazyk s dynamickým typováním se Python zaměřuje především na velice rychlý vývoj programů. Python je objektově orientovaný skriptovací jazyk a ke svému běhu používá interpretr. [47]

Velkou předností je schopnost pracovat s ostatními jazyky a efektivně spojovat jednotlivé komponenty. Ty mohou být psané například v C++, ale i v Javě. [47][48]

4.1.3 Java

Java byla vyvinuta v 90. letech původně pro mobilní zařízení. Je určena jako náhrada za C++ a proto je syntaxe velmi podobná.[49]

Hlavním cílem Javy je prevence vůči chybám způsobeným programátorem a klade důraz na bezpečnost. Jednou z předností je také schopnost spustit kód nezávisle na platformě.[49]

4.2 Kódování

Asistent je založen na práci s řetězci znaků, a proto je potřeba rozhodnout, jak se budou znaky reprezentovat v paměti. Jednoduchá podpora pouze anglických znaků není dostatečná už jen z pohledu možného rozšíření o další jazyky (v této práci proběhne návrh asistenta pracující pouze s angličtinou). Jako jediné vhodné řešení vidím použití kódování UTF-8 z rodiny UNICODE a odkázal bych se na , kde lze najít důvody pro použití právě tohoto kódování. Hlavním důvodem je dominance v oblasti internetu.

4.2.1 UTF-8

UTF-8 používá proměnnou délku znaků, kdy se každý znak skládá z 1 až 4 oktetů. Znaky jsou převáděny z číselné hodnoty, která jim byla přiřazena ve znakové sadě UNICODE.[50]

4.3 Podpůrné technologie a postupy

Z výše uvedených informací je jasné, že udělat inteligentního osobního asistenta je možné. Existuje spousta knihoven, které lze použít, ale k vytvoření dobrého programu to nestačí. Existuje spousta způsobů, jak si vývoj zjednodušit. Lze použít metodiky pro lepší přehlednost a udržitelnost zdrojového kódu. Dále je potřeba se seznámit s běžně používanými protokoly a strukturami pro uchovávání dat.

4.3.1 API

API je zkratka pro Application Programming Interface. Přináší abstrakci nad určitými problémy a umožňuje skrýt implementaci za předem definovaný kontrakt. Kontraktem se rozumí definované metody/volání určené k provedení určité činnosti. API lze najít na různých místech, od internetových aplikací, které nabízejí API v podobě zpracovávání informací při určitém volání, až po knihovny v programovacích jazycích. Ty zase nabízejí definované rozhraní pro práci s nimi. [51](1-15)

Hlavním přínosem API je skrytí implementace, programátor tedy nemusí problematice rozumět, ale prostřednictvím kontraktu mezi volajícím a volaným může efektivně pracovat třeba s formáty obrázků bez znalosti jejich vnitřní stavby. [51](1-15)

4.3.2 JSON

„*JSON (JavaScript Object Notation) je odlehčený formát pro výměnu dat. Je jednoduše čitelný i zapisovatelný člověkem a snadno analyzovatelný i generovatelný strojově.*“[52] Vznikl jako náhrada za XML[5] v místech, kde je potřeba vyměňovat menší množství dat. Definice byla standardizována v RFC 8259. Stavebním kamenem jsou objekty a pole, která lze skládat dohromady. Objekty jsou tvořeny hodnotou a klíčem, pole jsou zase po sobe jdoucí hodnoty. Samotná data jsou pak uložena jako řetězec znaků, číslo, boolean nebo hodnota null. Ostatní formáty je potřeba převést na jednu ze zmíněných možností, nejčastěji řetězec znaků. [53]

[5]Extensible Markup Language

Kapitola 5

Shrnutí

Tímto jsem shrnul základy inteligentních asistentů. Většina asistentů je konstruována pro fungování na cloudu. To jim dává možnost fungovat na více zařízeních s různými architekturami, ovšem může se objevit otázka zda je vhodné, aby uživatelé vše odesílali velkým firmám.

Pro svého asistenta, kterého budu navrhovat v praktické části této práce, jsem se rozhodl zvolit variantu, která nebude využívat internetové připojení pro zpracovávání zpráv. Pro tento účel využiji NLP knihovnu RASA NLU zmíněnou v části o knihovnách.

Knihovna je založená na programovacím jazyce Python, ale to neznamená, že jej musím použít. Raději využiji jazyk C++, který lze daleko lépe integrovat do systému a také má nižší využití zdrojů. Python použiji jako scriptovací jazyk pomocí technologie embeded Python a spuštění HTTP serveru pro zpracování zpráv na záměr a entity. Samotné C++ obohatím o soubor knihoven BOOST pro ulehčení práce s daty uloženými ve formátu JSON a práci s HTTP.

Komunikace s uživatelem bude probíhat pomocí jednoduchého chatovacího okna, uživatel bude psát zprávy určené ke zpracování a provedení a od asistenta bude dostávat odpovědi o zpracování. Interakce má připomínat chatování s opravdovým člověkem. Toto okno bude implementováno pomocí Windows API. Hlasové zadávání jsem se rozhodl nepoužít, jelikož ne každý počítač disponuje mikrofonem a zadávání na klávesnici je často efektivnější, na rozdíl od mobilních zařízení.

Část II

Praktická část

Kapitola 6

Analýza

V teoretické části jsem shrnul veškeré potřebné technologie a postupy, které jsou podle mého názoru nezbytné pro vytvoření vlastního asistenta. Proces vytváření jsem rozdělil do několika částí. Nejdříve stanovím požadavky na asistenta jako množinu podporovaných funkcí, poté navrhnu architekturu a na jejím základě provedu implementaci asistenta. Na závěr aplikaci otestuji podle případů užití.

Po vzoru již existujících řešení jsem se rozhodl pojmenovat celou aplikaci Steve. Konkrétní jméno je vybráno bez jakéhokoliv důvodu, jednoduše mi přirostlo k srdci.

Na asistenta je kladen jen jediný požadavek, a tím je množství funkcí. Protože je nemožné implementovat stejný počet funkcí, jako lze nalézt u konkurenčních řešení, vybral jsem podmnožinu, která je ideální pro představení základní funkčnosti asistenta. Na obrázku 6.1 jsou uvedeny jednotlivé funkce, kterými bude asistent disponovat.

K podmnožině funkcí je nutné přiřadit záměry. Tyto záměry budou použity v NLU modulu. V následujícím seznamu je ke každé funkci přiřazen záměr:

greet pozdrav,

goodbye vypnutí asistenta,

current_time zobrazení aktuálního času nebo data,

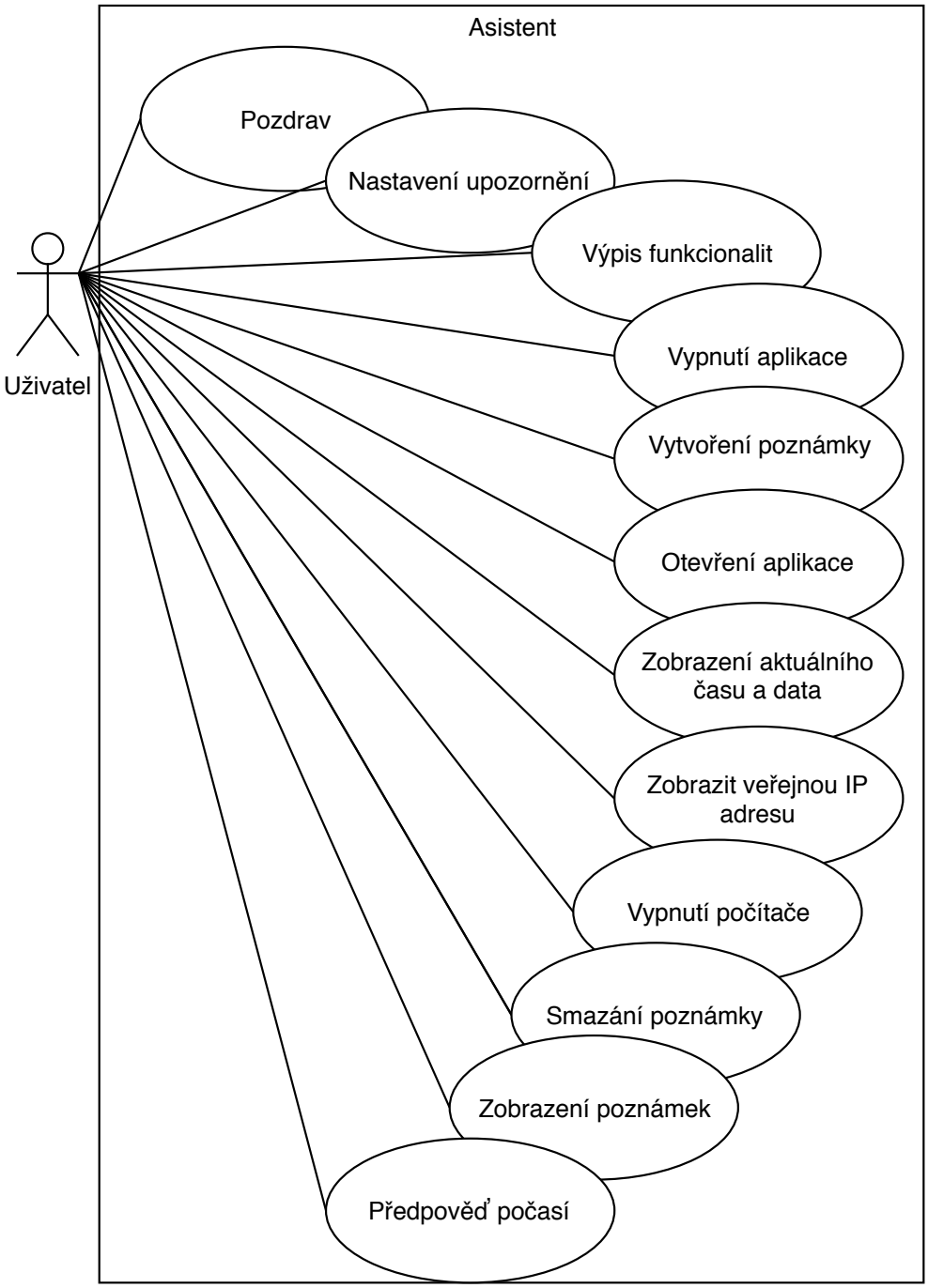
my_ip vypsání veřejné adresy zařízení,

create_note vytvoření poznámky,

show_note zobrazení poznámek,

delete_note smazání poznámky,

shutdown_pc vypnutí počítače,



Obrázek 6.1: Diagram případů užití vlastního asistenta

run_application spuštění aplikace,

alarm nastavení upozornění,

today_weather vypsání informací o dnešním počasí,

describe_yourself popis funkcí asistenta.

Jednotlivé funkce lze označit za funkční požadavky ve vztahu jedna funkce – jeden požadavek. Co se týče nefunkčních požadavků, je potřeba zohlednit fakt, že asistent je určen k nepřetržitému běhu, proto jsem požadavky definoval takto:

- použití technologií definovaných na konci teoretické části,
- nulová zátěž asistenta, pokud nezpracovává zprávu,
- fungování offline.

Kapitola 7

Návrh architektury

Architekturu budu navrhovat postupně v několika krocích. Začnu jednoduchým řešením a postupně jej budu inkrementálně zlepšovat. Vše doplním o drobné ukázky kódu, pro něž využiji jazyk C++, především pro možnost zobrazení předpisu třídy obsaženého v hlavičkových souborech. Některé části budou mírně změněné oproti podobě ve zdrojovém kódu, či budou popsány pseudokódem pro lepší názornost.

Předtím, než začnu s prvním návrhem, je ale potřeba vytvořit alespoň kostru celého asistenta. Rozhodl jsem se, že celý asistent bude realizován jako jedna třída zvaná `Core` obsahující komponenty pro dosažení funkcionalit. Tato třída bude obsahovat metodu `Core::run`, u které se bude očekávat že bude spuštěna jako samostatné vlákno. Toto vlákno bude autonomně zpracovávat příkazy. Příkazy si bude třída sama načítat z jiné třídy implementující rozhraní `CommunicationInterface` především proto, aby se snížilo množství zodpovědnosti třídy. Předpis této třídy lze vidět v ukázce kódu 1.

```
class CommunicationInterface
{
public:
  virtual string read()
  { return ""; }
  virtual void write(string message)
  { return; }
};
```

Výpis kódu 1: Předpis třídy CommunicationInterface

Pro tento základ jsem se rozhodl hlavně proto, aby zpracování neblokovalo jakékoliv uživatelské prostředí a aby byla zajištěna samostatnost asistenta. V následujících krocích se toto rozhodnutí již nebude měnit hlavně proto, že by se muselo jednat o změnu celého návrhu a ne pouze o inkrementální

zlepšování.

7.1 Základ pro asistenta

To, jak se asistent bude tvářit na venčí, je rozhodnuté, teď budu navrhovat vnitřní strukturu celé třídy `Core`. Práci asistenta lze rozdělit do dvou částí. Na začátku se zpracuje zpráva tak, aby jí rozuměl stroj, a následně se spustí příslušná funkce.

7.1.1 Zpracování zpráv

Aby bylo možné zpracovat příkazy, vytvořím rozhraní `TextProcessor`. Toto rozhraní bude implementovat pouze jednu metodu. Ta bude sloužit ke zpracování textu a následné vytvoření struktury pro uložení zpracovaných dat. Zpracování proběhne uvnitř `TextProcessor::process`. V ukázce kódu 2 lze vidět jednoduchou podobu tohoto rozhraní.

```
class TextProcessor
{
public:
  virtual ProcessedMessage process(string message) = 0;
};
```

Výpis kódu 2: Rozhraní TextProcessor

V ukázce kódu 2 představující rozhraní `TextProcessor` je uvedena návratová hodnota funkce `TextProcessor::process` (funkce zpracovávající zprávu) jako `ProcessedMessage`. Jedná se o další důležitý objekt pro udržení celistvosti celé architektury. Jeho hlavním účelem je uchovávat informace o zpracované zprávě. Pro funkci asistenta využívám dvou objektů, jedním je záměr a druhým pole entit (vycházím z předpokladu využití jedné z knihoven zmíněné dříve v této práci v sekci zabývající se tímto tématem). Tyto informace se vyextrahují z příchozí zprávy. Protože se ale musí dopravit do místa zodpovědného za provedení jednotlivých funkcí, pro tento účel data zabalím do třídy a zároveň tím vytvořím i kontrakt pro přistupování k těmto datům. Data se musí zpracovat z formátu JSON, toto zpracování proběhne rovnou v dané třídě a zajistí se tím i efektivní naplnění datových kontejnerů. Pro správné uložení těchto dat je ale potřeba znát jejich přesnou podobu. Je potřeba uložit záměr, ten lze uložit jednoduše jako znakový řetězec do datového typu `string`, ten je kompatibilní s UTF-8. Při ukládání entit již nastává problém, jež vzniká z povahy dat, entita je reprezentována jako dvojice hodnota a klíč. Jelikož nikde není zaručena unikátnost klíče v rámci jedné zprávy, je potřeba hodnoty pro každý klíč ukládat do pole. To nemusí být na škodu a tohoto lze využít při návrhu lepších funkcionalit (nastavení upozornění na 2 různé

časy v rámci jedné zprávy). Zvolenou podobu kontejneru lze vidět na ukázce kódu 3.

```
class ProcessedMessage
{
public:
  virtual bool from_json(std::string json);
  string get_intent() const;
  vector<string> get_entity(string entity_name) const;
private:
  string intent;
  map<string, vector<string>> entities;
};
```

Výpis kódu 3: Třída ProcessedMessage zodpovědná za správu dat získaných zpracováním v NLP modulu.

Metoda `ProcessedMessage::from_json` je virtuální, aby bylo možné změnit její připadnou implementaci ve zděděné podtřídě využívající jinou NLU knihovnu. Tato knihovna pak může mít jiný formát dat a takto lze v programu použít případně dvě rozdílné knihovny.

7.1.2 Spouštění funkcionalit

Funkce samotné budou implementovat rozhraní `Command`. Jedná se o velice jednoduchý návrh s pouze jednou metodou určenou k provedení dané funkce. Jako parametr pro funkci použiji třídu `ProcessedMessage` definovanou v předchozím odstavci, z které je možné získat jednotlivé entity a případně i samotný záměr, pokud to bude potřeba. Předpis je vidět v ukázce 4.

```
class Command
{
public:
  virtual string execute(ProcessedMessage const & message)=0;
};
```

Výpis kódu 4: Předpis rozhraní Command

To jak vybrat správnou funkci, z dat získaných z uživatelovy zprávy zajistí třída `CommandsHolder`. Ta bude udržovat instanci všech tříd implementující rozhraní `Command`. Jednotlivé instance bude možné adresovat pomocí jejich záměru. Pro toto lze využít datový typ mapa. Úroveň abstrakce umožní výběr a spuštění správné funkce v několika krátkých krocích uvnitř metody `CommandsHolder::execute`, její možnou podobu a také předpis celé třídy lze vidět na ukázce kódu 5.

```
class CommandsHolder
{
public:
  CommandsHolder();
  string execute(ProcessedMessage const message);
private:
  map<string, Command* > commands;
};

string CommandsHolder::execute(ProcessedMessage const & message)
{
  string intent = message.get_intent();
  auto command = commands.get(intent);
  if (command)
  {
    return exec->second->execute(parameters);
  }
  return "No command found.";
}
```

Výpis kódu 5: Třída CommandHolder s metodou execute

7.1.3 Propojení komponent

Posledním krokem pro kompletní zprovoznění asistenta je správné provedení posloupnosti následujících kroků, které se budou provádět uvnitř metody `Core::run`.

1. Převzít zprávu od uživatele.
2. Zpracovat zprávu pomocí NLU modulu.
3. Pomocí třídy `CommandHolders` spustit funkcionalitu.
4. Poslat odpověď zpět uživateli.

Na obrázku 7.1 je zobrazen diagram komponent celé architektury. Stěžejní prvek tvoří třída `Core`.

7.2 Jak si pamatovat informace napříč zprávami

Jádro aplikace zvládá jednoduchou komunikaci typu otázka – odpověď, to ale pro fungování asistenta nestačí, může nastat scénář, u kterého je potřeba vést konverzaci, například takto:

Uživatel Smaž všechny poznámky.

Asistent Opravdu mám smazat všechny poznámky?

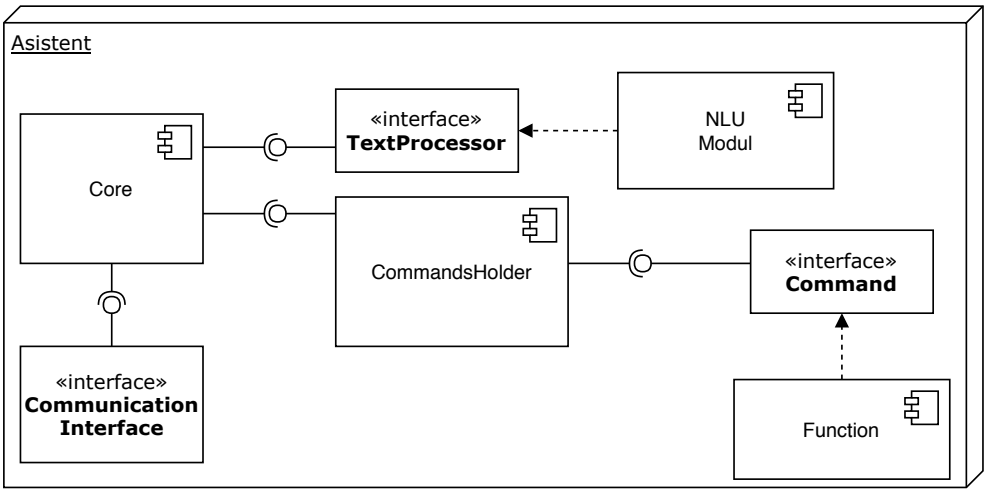
Uživatel Ano.

Asistent Poznámky smazány.

V aktuálním návrhu nelze takovou konverzaci vést, protože si asistent neuchovává stav (nemá informace o tom, co zpracovával v předešlém průchodu). Nyní ukážu, jak toto vyřešit.

První myšlenka je taková, že se bokem do vyhrazené struktury uloží stav, tudíž v případě mazání poznámek dojde k poznamenání, že uživatel právě chce mazat poznámky. Obdrží jednoduchou odpověď s dotazem na ujištění. Při příštím zpracování zprávy dojde k využití této uložené informace a je možné pak zvolit správný průběh, tedy poznámku smazat nebo ne podle přání uživatele. I přesto, jak triviálně toto řešení zní, může dojít ke komplikacím. Nejdříve je potřeba změnit celý postup zpracování zprávy, aby se zohlednilo ukládání stavů, dále se tyto stavy mohou začít větvit a bude potřeba do nich ukládat informace, například které poznámky se mají smazat. Toto řešení již není tak hezké a spíše vytváří nepříjemné větvení. Pokud bude výměn informací více v rámci jednoho požadavku, začne vznikat chaos už jen v tom, jak vždy správně navázat při dalším průchodu celého mechanizmu.

Nové řešení využívá odlišný přístup. Cílem je, aby obsluha konverzace v rámci jedné metody asistenta vypadala podobně jako je tomu na ukázce kódu 6. Ukázka představuje pseudokód ideální metody `Command::execute`, která si uchovává všechny získané zdroje na zásobníku a je schopná komunikovat s uživatelem. V této variantě může nastat problém s napsáním špatné



Obrázek 7.1: Diagram komponent asistenta

odpovědi od uživatele, kdy odpoví místo ano nebo ne požadavkem na jinou funkcionalitu. Zde tedy dojde k odpovědi od asistenta (například „Neplatná operace“), a poté je na programátorovi, jestli se zeptá znovu, nebo ukončí zpracovávání daného záměru. Osobně si myslím, že není potřeba tento problém řešit, protože jeho řešení by příliš zkomplikovalo logiku asistenta a troufnu si tvrdit, že by možnost změny tématu uprostřed konverzace mohla navodit nepřirozený pocit.

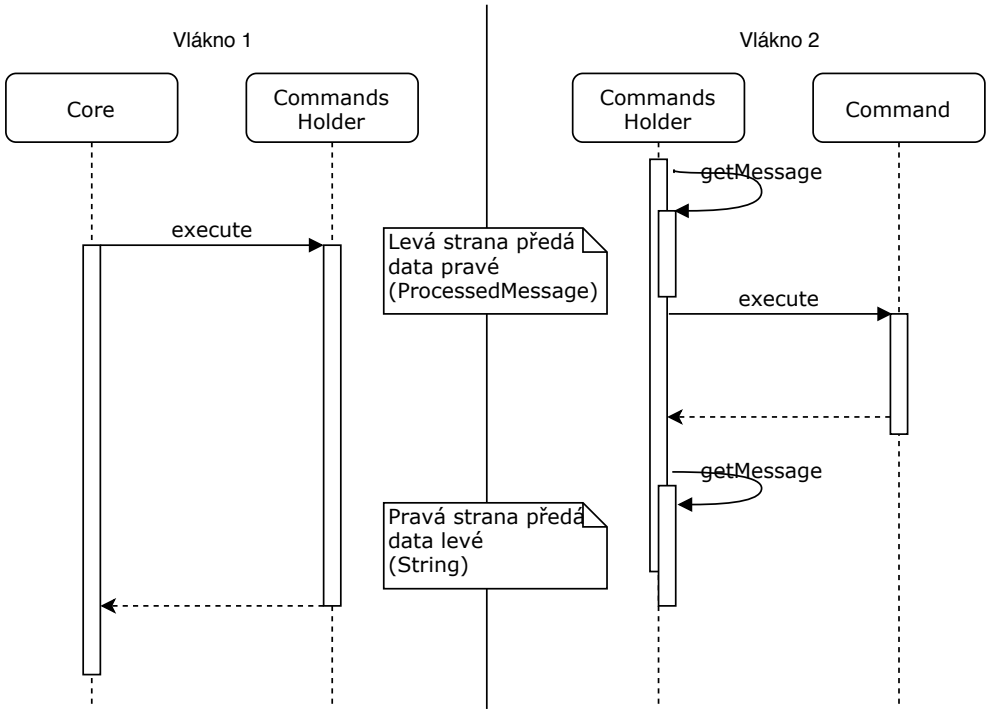
```
string Command::execute(ProcessedMessage const & message)
{
  i=message.entita("číslo poznámky")
  odpověď = zeptejSeNaPotvrzení()
  pokud odpověď.záměr == ano{
    smažPoznámku(i)
    return "Poznámky smazány."
  }
  return "Operace ukončena."
}
```

Výpis kódu 6: Třída CommandHolder s metodou execute

Dle mého názoru je toto řešení daleko více přívětivé pro psaní jednotlivých funkcionalit asistenta. Problém ovšem je, jak zařídit, aby se aktuální pozice v metodě `Command::execute` uložila spolu se všemi proměnnými. Je potřeba zavolat `return`, aby se text vypsal uživateli, ten ale metodu nenávratně opustí. Záchrana však přichází v podobě vícevláknového provedení. Idea je taková, že jedno vlákno se stále stará o zpracování zprávy od uživatele jako dříve, ale když se vlákno dostane do `CommandsHolder::execute`, bude zastaveno ve prospěch druhého vlákna, které bude obsluhovat jednotlivé implementace `Command::execute`. Druhé vlákno se zase naopak spustí pouze aby obsloužilo požadavek na zpracování příkazu, a následně se pomocí synchronizace zastaví ve prospěch prvního vlákna. Zde se využije vlastnost vlákna, a to je samostatný zásobník, na kterém zůstanou uložené veškeré informace. Pro lepší znázornění je vše popsáno na obrázku 7.2

Stále ale přetrvává problém, jak získat odpověď od uživatele zpátky do místa probíhající metody. Toho lze docílit tím, že do metody jako další parametr použijeme samotný `CommandsHolder` a funkce si sama zařídí odpověď, tedy při zpracovávání další zprávy od uživatele zprávu převezme a bude pokračovat v provádění kódu samotné funkce. Tímto způsobem dojde k efektivní výměně informací mezi volanou funkcí určitého záměru a uživatelem.

```
string Command::execute(ProcessedData
  const & message, CommandsHolder * cm)
```



Obrázek 7.2: Sekvenční diagram znázorňující pokročilé zpracování v metodě CommandsHolder::execute

Dříve jsem zmínil problém tohoto návrhu v podobě nemožnosti zpracování jiného druhu zprávy, pokud se očekává odpověď, například ano či ne, a uživatel zažádá o vypsání poznámek. K tomuto připadá další skupina zpráv. Ty, u kterých lze očekávat, že uživatel odpoví a rozvine konverzaci v rámci daného tématu, ale odpovědět nemusí a může kompletně změnit téma. Příkladem této zprávy je „Aktuálně je venku 25°C, pokud chcete, mohu vám vypsat další informace“, na toto lze reagovat souhlasem nebo jednoduše přejít k jinému tématu.

Řešit tento druh komunikace lze pomocí výjimek. Pokud se nebude jednat o očekávanou zprávu (v tomto případě potvrzení), vyhodí se výjimka a část kódu zodpovědná za volání jednotlivých implementací `Command::execute` ji odchytne a správně zavolá vhodnou metodu s parametrem získaným z odchycené výjimky. Pro přehlednost přikládám pseudokód z metody.

```
odpověď = vypišInformaceAZískejOdpověď()
pokud odpověď.záměr == ano{
  return "Další informace."
}
throw odpověď
```

Po popsání mechanizmu může vyznít tento celý systém zpracovávání jako příliš komplikovaný a uchovávání stavů se jeví jako jednodušší a hlavně přívětivější. I když logika zpracování nabere na složitosti, stále se zachová jednoduchost vytváření zpráv. Hlavní výhodou je možnost vrátit se do metody `Command::execute` v libovolném bodě zpracování. Je tudíž možné uchovávat si informace z předchozí zprávy a zároveň postupně pokračovat jednotlivými kroky a otázkami směrem k úspěšnému splnění uživatelova požadavku.

Jednoduchým příkladem může být otevření souboru. Tím, že se pomyslný stav metody neukládá mimo, je jednoduché hlídat si jak uzavření souboru (protože nevzniká polymorfní struktura obsahující ukazatel na soubor pro uložení stavu), tak zamezit vytváření velkého množství funkcí umožňujících přecházet mezi jednotlivými kroky zpracování. Metoda `Command::execute` nemusí obsahovat spoustu podmínek, podle kterých by vybírala správnou část kódu určenou ke zpracování v závislosti na průběhu konverzace. Zamezí se tak tomu, aby se kód stal velice rychle nepřehledný a začaly se v něm objevovat chyby.

Vše ovšem není stále dokonalé, pokud dojde na vytváření složitých dialogů mezi uživatelem a aplikací, které budou doprovázeny bohatým větvením, a nebude se tedy jednat o jednoduchou, přímočarou komunikaci, kde je zájem pouze na získání potvrzení od uživatele, či získání některé informace, jako je název města při předpovědi počasí, začne vznikat nežádoucí větvení v rámci implementace kódu. Tohoto rizika jsem si vědom a považuji jej za akceptovatelné. Především z povahy celého systému, kdy se nejedná o aplikaci určenou k rozsáhle konverzaci, ale k rychlé a efektivní obsluze požadavků od uživatele, kde je na dříve zmíněných příkladech ukázán jasný přínos v podobě jednodušího návrhu funkcionalit.

7.3 Rozšířená podpora jednotlivých funkcionalit

Metody `Command::execute` jsou nyní schopné komunikovat s uživatelem, ale zbývá dořešit poslední problém, a tím je komunikace mezi jednotlivými metodami. Jako příklad lze opět použít poznámky. Od asistenta se očekává, že pokud zvládá poznámky mazat, musí je také umět vytvořit a bylo by též vhodné, aby je zvládl zobrazit. V aktuální fázi návrhu z předešlé kapitoly jsou ale jednotlivé funkcionality od sebe odříznuty, tudíž pokud se poznámky vytvoří, nelze je jakkoli zpřístupnit funkci určené k zobrazení.

Hlavním požadavkem na toto propojení je umožnit jednotlivým třídám implementující `Command::execute` sdílet data, případně určité funkcionality, nebo jim také zpřístupnit funkce, ke kterým by se jinak nedostaly (přímá komunikace s GUI[6], pokud například bude existovat funkce umožňující změnu barvy pozadí).

[6]Graphical user interface

Řešit propojení funkcí lze několika způsoby. Jedno z možných řešení je ukládat informace do globálních proměnných nebo využít návrhový vzor „singleton“. Tato řešení jsou ovšem v rozporu s dobrým designem a z dlouhodobého hlediska neudržitelná. Rozhodl jsem se vyřešit tuto situaci pomocí API, které bude obsahovat jednotlivé komponenty. Ty se budou vytvářet ve třídě `Core` spolu s třídou `CoreAPI` (ta fyzicky znázorňuje API), která je bude všechny obsahovat.

Data lze ukládat do struktur a funkce lze předávat pomocí tříd, tudíž je nutné tyto struktury a třídy předat do jednotlivých metod obsluhující požadavky. Toto předání proběhne v rámci API, ke kterému získají objekty zděděné od třídy `Command` přístup během jejich inicializace jako parametr v konstruktoru, případně lze zavolat metodu pro inicializaci v jazcích bez podpory konstruktorů.

API objekty mohu zpřístupnit pomocí dvou různých variant implementace třídy `CoreAPI`. První z nich je jednoduchá implementace pomocí settrů a gettrů. Tato implementace je jednoduchá a pro malé množství komponent efektivní. Pro větší množství komponent je vhodnější využít druhou variantu, v níž se využije mapování jednotlivých komponent na znakové řetězce. U druhé varianty lze počítat s nutností přetypování u staticky typovaných jazyků. Na ukázce kódu 7 lze vidět předpis druhé varianty.

```
class CoreAPI
{
public:
  COMPONENT * getComp(const string name) const;
  void registerComponent(string name, COMPONENT * comp);
private:
  map<string, COMPONENT *> components;
}
```

Výpis kódu 7: Předpis třídy CoreAPI

Příklad s poznámkami lze vyřešit velice elegantně, začne se vytvořením komponenty obsluhující poznámky, ta bude reprezentována třídou s jednoduchým polem pro uložení poznámek a metodami pro uložení nové poznámky a pro výpis jednotlivých poznámek. To ovšem nejsou jediné funkce. Lze implementovat mazání poznámek a spoustu podpůrných funkcí, jako jsou rozdílné metody formátování výpisu. Viz ukázka 8.

Komponentu si poté každá třída určená k obsluze poznámek uloží během volání jejího konstruktoru.

```
ShowNoteCommand(CoreAPI* api):mNote_m(api->getComp("notes")){}
CreateNoteCommand(CoreAPI* api):mNote_m(api->getComp("notes")){}
```

```
class NotesManager
{
public:
    string getFormatedNotes() const;
    void addNote(const string note);
private:
  vector<string> notes;
}
```

Výpis kódu 8: Jednoduchý předpis třídy NotesManager

Metody `ShowNoteCommand::execute` a `CreateNoteCommand::execute` mohou po inicializaci spolupracovat pomocí volání metod třídy `NotesManager`.

Drobná obtíž tohoto řešení spočívá v nutnosti vytvářet komponenty během spouštění aplikace. Zároveň nelze přidat komponentu později, kdy je aplikace již určitou dobu spuštěna. I s tímto se ovšem dá pracovat, pro tento účel vytvořím komponentu s názvem `LazyLoader`. Tato komponenta bude pracovat podobně jako samotná třída `CoreAPI`, s tím rozdílem, že vkládání komponent může proběhnout později. Během inicializace si třída uloží `LazyLoader` a když dojde k nutnosti využití objektu, který nelze vytvořit během spuštění aplikace, dotáže se zvolené komponenty jestli již proběhlo vytvoření objektu. Pokud neproběhlo, může to oznámit uživateli a případně jej nasměrovat k akci určené k vytvoření daného objektu (příkladem objektu, který neexistuje během spuštění aplikace může být vyskakovací okno).

7.4 Výsledek návrhu

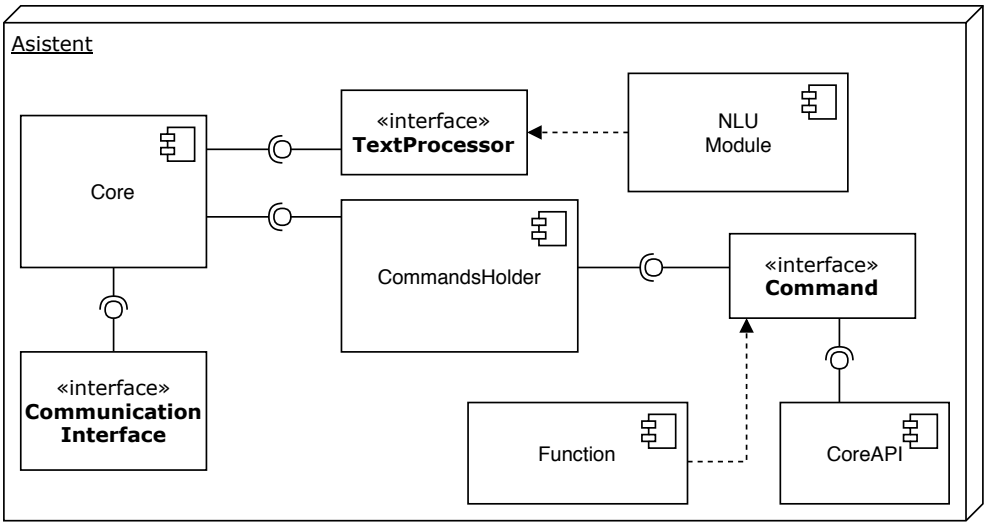
Architektura podporuje základní formu obsluhy uživatelova požadavku, ale také podporuje daleko více. Je možné získat další odpovědi od uživatele, to může sloužit k získávání potvrzení, či dodatečného zjištění parametru pro funkci. Ale také lze propojit jednotlivé funkce, aby bylo možné obsloužit více požadavků se stejným základem. Dále bylo navrhnuto jak řešit některé speciálními případy, například zpráva na kterou může uživatel odpovědět volitelně.

Rozhraní `CommunicationInterface` slouží k napojení vstupu a výstupu. To umožňuje volbu libovolného rozhraní pro komunikaci s uživatelem, od klasické desktopové aplikace po využití webových aplikací. Také je potřeba dodat implementaci třídy `TextProcessor`, která bude zpracovávat zprávy v přirozeném jazyce a bude schopná naplnit obsahem třídu `ProcessedMessage`. A zde jsou jednotlivé kroky znázorňující postup přidávání jednotlivých funkcionalit:

- naučit TextProcessor rozpoznat závěr a entity,
- pokud je potřeba, vytvořit komponenty do API,

- vytvořit třídu implementující rozhraní Command,
- a v posledním kroku stačí implementovanou třídu přidat do CommandsHolder::inicialize, aby bylo možné jí zavolat.

Diagram komponent se mírně změnil, a to obohacením o komponentu znázorňující API. Aktualizovaný diagram je znázorněn obrázkem 7.3.



Obrázek 7.3: Výsledný diagram komponent asistenta

Kapitola 8

Implementace

V této kapitole proběhne implementace celého asistenta, navážu zde na návrh architektury, přidám GUI pro komunikaci s uživatelem a poté vytvořím jednotlivé funkcionality. V další kapitole proběhne testování a zhodnocení výsledku implementace.

Pro vývoj použiji Visual Studio 2015 s doplňkem pro C++ a Python ve verzi 3.6.

8.1 Jádro asistenta

Jádro asistenta tvoří jedna stěžejní třída `Core`, ta zaobaluje všechny ostatní potřebné komponenty, její implementace je velice jednoduchá, především proto, že se zde provádí jen jednoduchá výměna dat a inicializace API.

Nejdůležitější částí je metoda `Core::run`. Její účel je převzít zprávu od uživatele a vrátit zpět odpověď. Protože funkce asistenta mohou být časově náročné (několik sekund), bude metoda spuštěna v samostatném vláknu, tím nedojde k blokování GUI. Vlákno bude spuštěno zvenčí, třída jej nebude sama spouštět a ukončovat, ale bude spuštěno částí zodpovědnou za správu celé instance třídy `Core`. Kód pro metodu lze vidět na ukázce 9.

Metoda využívá objektů, které jsem označil jako „handle“. Tyto objekty jsou implementacemi rozhraní `CommunicationInterface`. Zvolil jsem oddělené objekty pro vstup a výstup, tudíž lze využít rozdílné třídy pro tyto operace. Metody `Core::SetOutputHandle` a `Core::SetInputHandle` předají ukazatele na tyto objekty v podobě parametrů. Kromě této inicializace je ještě přítomna metoda `Core::initialize`. Ta je zodpovědná za inicializaci všech vnořených objektů, jako je `CoreAPI`, `TextProcessor` a `CommandsHolder`. Operace se neprovádějí v konstruktoru protože se jedná o výpočetně náročné operace a je tedy vhodné je provést až vláknem určeným k obsluze dané třídy a snížit tím blokování zbytku aplikace, proto je metoda volána v `Core::run`.

```
void Core::run()
{
  initialize();
  while (run)
  {
    string message = input_handle->read();
    if (!message.empty())
    {
      ProcessedMessage pm = txtProc->process(message);
      string output = commandsHolder->execute(pm);
      output_handle->write(output);
    }
  }
  finalize();
}
```

Výpis kódu 9: Implementace metody run třídy Core

Další třídou je `CommandsHolder`. Zde je nutné věnovat pozornost synchronizaci vláken, aby nebyl procesor zbytečně zatížen. V návrhu je tato třída implementována pomocí vlastního vlákna, které umožňuje jednotlivým funkcionalitám zpětnou komunikaci s uživatelem. Toto vlákno zde spustím během inicializace a postarám se, aby si hlavní vlákno uzamklo zámek. Vlákno spuštěné uvnitř třídy se poté zastaví a čeká na uvolnění zámku, které signalizuje, že proběhla výměna informací směrem dovnitř a může si zámek přivlastnit. Tím také začne informace zpracovávat a spustí příslušnou funkcionalitu. Po jejím ukončí nebo žádosti o další vstup od uživatele se zámek uvolní, získá jej opět hlavní vlákno a může si přečíst data, která mají směřovat k uživateli.

Tato výměna dat bude probíhat pomocí dvou funkcí. Jedna, volající zvenčí, se jmenuje `CommandsHolder::execute`, druhá, která s ní spolupracuje, se jmenuje `CommandsHolder::data_exchange`. Obě metody jsou vzájemně synchronizovány. První se celá provede i ukončí během zpracování jedné uživatelské zprávy. Druhá pak zůstává zablokována v průběhu svého vykonávání a čeká, dokud nebude nahrána zpráva od uživatele. Ke druhé metodě má přistup i jakákoli třída implementující rozhraní `Command`, takže může komunikovat s uživatelem způsobem znázorněným v návrhu.

Kromě výměny dat je `CommandsHolder` zodpovědný za inicializaci funkcionalit. K této inicializaci využije instanci třídy `CoreAPI`, která je mu předána z třídy `Core`. Tu pak využije během inicializace jednotlivých instancí reprezentujících funkce asistenta.

Pro implementování rozhraní TextProcessor jsem zvolil využití jednoduché třídy schopné komunikovat přes HTTP především proto, že zpracovávání přirozeného jazyka bude probíhat pomocí knihovny RASA NLU uvedené v te-

oretické části. Protože knihovna není psaná v C++, budu ji využívat jako jednoduchý server, od kterého dostanu zpracovanou odpověď na každý požadavek v přirozeném jazyce. Obsluhu HTTP zařídí knihovna C++ BOOST, konkrétně její část nazvaná ASIO a BEAST. Následně po obdržení odpovědi ve formátu JSON ji zpracuji opět za pomoci knihovny BOOST, v tomto případě využiji datový typ `Property tree` určený přímo k práci s JSON formátem. Zpracování odpovědi proběhne způsobem nastíněným v kapitole zasvěcené návrhu architektury uvnitř třídy `ProcessedMessage`, její předpis bylo možné vidět v ukázce 3.

Posledním krokem ke kompletnosti už je jen integrace RASA NLU serveru přímo do aplikace.

8.1.1 Připojení NLP knihovny

I přesto že Python jako skriptovací jazyk je hojně používán například k vytváření pluginů do grafických aplikací, stále je netriviální záležitostí tuto integraci provést. Především z důvodu mé neznalosti celého jazyka, jsem se rozhodl jít tou nejsnadnější cestou, a to je přistupování k celé knihovně RASA NLU jako k celistvému modulu bez jakékoliv možnosti přímé komunikace mezi jazyky. Komunikace není potřeba, protože využívám služeb HTTP serveru. I tak ale stále potřebuji umožnit svému asistentovy, aby si mohl tento server spustit. K tomuto účelu existuje Python C API.

Samotné API je velice rozmanité a umožňuje kompletní propojení obou jazyků. Já však využiji pouze tu část, která mi dovolí spustit libovolný soubor napsaný v Pythonu, jako by se jednalo o celý program. Předtím, než toto ale provedu, je nutné si dát pozor na povahu funkce, která toto zajišťuje. Funkce je blokující a čeká, dokud neskončí běh souboru. To vyřeším tak, že během inicializace vytvořím nové vlákno, a to se zablokuje. Poté při ukončování aplikace lze využít funkce, která oznámí interpretru Pythonu, aby se ukončil, čímž dojde k odblokování vytvořeného vlákna, a to se může ukončit.

Kód pro spuštění modulu obsahující server je vidět v ukázce kódu 10.

Integrací se nedosáhne jen spuštění serveru, ale také dojde k odstranění nutnosti instalovat interpretr Pythonu přímo na počítač uživatele. Vzniká tím ale také povinnost při distribuci softwaru přibalit runtime a knihovnu Python.dll, obě jsou obsažené v základní instalaci Pythonu.

8.2 Implementace GUI

Grafická rozhraní jsou často se měnící částí softwaru a slouží pouze k zpřístupnění business logiky. Proto jsem se rozhodl, vytvořit pouze minimalistické GUI, které bude dostatečné pro fungování asistenta. Na výběr jsem měl mezi webovým rozhraním a klasickou desktopovou aplikací. Rozhodl jsem se pro druhou variantu, protože umožňuje lepší integraci do systému a snižuje nároky na zdroje, tedy není nutné spouštět webový server.

```
string module_path = "...\\rasa_nlu\\server.py";
Py_SetPythonHome(L"Python36");
Py_Initialize();
FILE* PythonScriptFile = nullptr;
fopen_s(&PythonScriptFile, module_path.c_str(), "r");
if (PythonScriptFile)
{
  if(PyRun_SimpleFile(PythonScriptFile, module_path.c_str()) == -1)
    error();
  fclose(PythonScriptFile);
}
Py_FinalizeEx();
```

Výpis kódu 10: Spouštění Python modulu

Pro svého asistenta použiji klasický vzhled, který se bude snažit přiblížit fungováním k již zaběhlým aplikacím, jako je například Skype. Bude se skládat z:

- okna pro zadání zprávy,
- tlačítka pro odeslání zprávy ke zpracování,
- okna pro zobrazení celé konverzace.

V rámci této práce nemá smysl více se zabývat návrhem GUI především k výše popsaných důvodů.

Aby přeci jen celá aplikace byla alespoň trochu uživatelsky přívětivá, využiji pro zobrazení zpráv uživatele a asistenta technologii zvanou Rich Edit. Jedná se o technologii vyvinutou firmou Microsoft pro stylizaci textu.[54]

K výběru GUI se váže i jeho obsluha. Pro ni se ve WINAPI využívá zpracování zpráv. Tyto zprávy může odeslat kdokoli. Pro mě je nejdůležitější zprávou `BN_CLICKED`. Ta je odeslána pokud uživatel stiskne tlačítko určené k odeslání jeho zprávy ke zpracování asistentem. Při stisknutí tlačítka se musí uživatelova zpráva zapsat do třídy implementující rozhraní `CommunicationInterface`. Po úspěšném zapsání, si asistent může zprávu přečíst a zpracovat. Nyní je potřeba asistentovu odpověď vrátit zpět uživateli. K tomu využiji svou vlastní definici zprávy. Tato zpráva se pošle z třídy implementující `CommunicationInterface` a oznámí, že GUI může zobrazit odpověď. Touto metodou se dá jednoduše dosáhnout synchronizace komponent GUI a celého asistenta, aniž by vznikala nutnost neustálého běhu aplikace kvůli čekání ve smyčce, která by zbytečně zatěžovala procesor.

Výslednou podobu GUI je možné vidět na obrázku 9.5.

8.3 Implementace funkcí

Jádro aplikace je hotové, GUI je také připravené a přichází na řadu implementace jednotlivých funkcionalit asistenta. Ty budu implementovat podle postupu specifikovaného v kapitole s návrhem.

Pro každý záměr je potřeba vytvořit sadu zpráv v přirozeném jazyce, které budou nejlépe znázorňovat způsob, jakým uživatel zažádá o spuštění dané funkce. Takto například vypadá jedna z mnoha definovaných zpráv pro nastavení upozornění:

```
{
  "text": "Set alarm to 10:30",
  "intent": "alarm",
  "entities": [
    {
     "start": 13,
     "end": 18,
     "value": "10:30",
     "entity": "time",
     "extractor": "ner_crf"
    }
  ]
}
```

Kvalita rozpoznávání stoupá s množstvím definovaných příkladů. Z dokumentace k RASA NLU[38] lze zjistit, že minimum je 2 příklady pro každý záměr. Pro optimální fungování je pak doporučeno 20 až 30 příkladů.

Spolu se zprávami jsou definovány i entity určené k extrakci. Pro ty je také potřeba definovat více příkladů a po vlastní zkušenosti doporučuji využít pro ně jednoduchý vzor, u textu to například může být umístění do uvozovek, tím se velice zvýší rozpoznávací schopnosti. Odpověď na zprávu „Set alarm to 12:35“ po nadefinování několika další zpráv vypadá takto (pro zkrácení zápisu byla vyjmuta část zobrazující, s jakou pravděpodobností proběhla predikce):

```
{
  "intent": {
    "name": "alarm",
  },
  "entities": [
    {
      "start": 13,
      "end": 18,
      "value": "12:35",
      "entity": "time",
      "extractor": "ner_crf"
```

```
    }
  ]
  "text": "Set alarm to 12:35"
}
```

Po definování příkladů je potřeba celý model natrénovat, pro tento účel jsem vytvořil sekundární program zvaný `train.exe` (je přiložen na médiu v adresáři se spustitelnými soubory). Ten spouští modul `train.py` z knihovny RASA NLU stejným způsobem, jakým je spouštěn server.

Před implementací funkcionalit připravím několik komponent do API, které mi umožní plnohodnotně využít potenciál asistenta. Jednou takovou komponentou určitě musí být správa poznámek. Dále je potřeba vytvořit komponentu spravující čas, ta bude určena pro zajištění fungování upozornění. Nakonec vytvořím komponentu obsluhující předpověď počasí, která bude sloužit k dodržení správného designu a to tím, že odděluje komunikaci s API z internetu od samotného rozhraní `Command`, určeného ke komunikaci s uživatelem. Takto nakonec vypadá přehled jednotlivých komponent s jejich popisem a deklarací.

- NotesManager spravuje poznámky. Zajišťuje jejich persistenci a umožňuje k nim přistupovat napříč jednotlivými metodami `Command::run`.

  ```
  class NotesManager
  {
  public:
    void add_note(const string note);
    string show_notes() const;
    bool delete_note(const int note_number);
    string get_note(const int note_number);
    unsigned int notes_count();
  private:
    vector<string> notes;
  };
  ```

- TimeManager slouží ke správě časově závislých událostí. Jednou z takových událostí může být nastavení upozornění na určitý čas. Je možné vytvořit libovolnou událost, která bude spuštěna v nastavený čas, pokud implementuje rozhraní `TimedTask`. Zde kontrakt zaručuje, že v určitý čas bude spuštěna metoda `TimedTask::perform`.

  ```
  class TimeManager
  {
  public:
    void start();
    void end();
  ```

```
  void add_event(shared_ptr<TimedTask> timed_task);
private:
  bool run;
  thread time_thread;
  set<shared_ptr<TimedTask>> tasks;
};

class TimedTask
{
private:
  time_point time;
public:
  TimedTask(time_point input_time) :time(input_time){}
  time_point get_time() { return time; }
  virtual void perform() = 0;
};
```

- WeatherService je určena k práci s předpovědí počasí. Slouží jako ukázka, jak pracovat s API třetích stran, která jsou vystavena na internetu.

```
//Used for weather forecast
//https://openweathermap.org
class WeatherService
{
public:
  // returns property tree
  // containing parsed json
  pt::ptree weather_today(string city);
private:
  string http_get_request(string address);
};
```

Každá funkcionalita musí implementovat rozhraní `Command` a konkrétně metodu `Command::execute`. Většina z nich je velice jednoduchá, u některých se jedná o pouhé vrácení návratové hodnoty v podobě textu. U složitějších metod je ale nutné dodat logiku umožňující komplexní odpovědi . Ty jsem se rozhodl uvést, spolu s drobným popisem jejich fungování. U jednotlivých příkladů je uveden záměr, který vede k jejich spuštění.

greet Slouží k jednoduché odpovědi, pokud uživatel napíše například „Hello“. Metoda je navržena komplexněji, než je nutné, k předvedení možností, jak přidat více různých odpovědí na jeden požadavek. Zde se počítá počet volání metody, který se ukládá v instanční proměnné.

```
string HelloCommand::execute(ProcessedMessage
  const & message_data, CommandsHolder * command_holder)
{
  ++execution_number;
  switch(execution_number)
  {
    case 1:
      return "Hello, my name is Steve.";
    default:
      return "Hello.";
  }
}
```

create_note Tato metoda vytvoří poznámky. Za poznámku je považováno vše, co je označené jako entita „context“. Je zde využita možnost použití více entit se stejným klíčem. Při napsání zprávy „Create notes ”Hi”and ”Hello”“ se vytvoří dvě poznámky. Další vlastností je využití objektu z API, konkrétně komponenta určená ke zprávě poznámek uložená v instanční proměnné `notes_manager`.

```
string CreateNoteCommand::execute(ProcessedMessage
  const & message_data, CommandsHolder * command_holder)
{
  auto result = message_data.get_entity("context");
  if (result.size() == 0)
    return "To create note I need content.";
  for(std::string context : result)
    notes_manager->add_note(context);
  return "Noted.";
}
```

show_notes V předchozím bodě se poznámky přidávají do komponenty na poznámky. V této metodě je možné je vypsat. Výpis je zformátován přímo v komponentě, tudíž stačí pouze vrátit výsledek volání.

```
string ShowNoteCommand::execute(ProcessedMessage
  const & message_data, CommandsHolder * command_holder)
{
  return notes_manager->show_notes();
}
```

current_time Umožňuje zobrazit aktuální čas a datum. Podle toho, jestli uživatel specifikuje ve svém požadavku slovo „date“ nebo „time“, se vybere správná odpověď.

```
string CurrentDateTimeCommand::execute(ProcessedMessage
  const & message_data, CommandsHolder * command_holder)
{
  bool time = false;
  bool date = false;
  for (auto str : message_data.get_entity("value"))
  {
    if (str == "time")
      time = true;
    if (str == "date")
      date = true;
  }
  if (time && date)
    return "Actual time is "+ get_time()
      +" date is " + get_date() + ".";
  if (time)
    return "Current time is " + get_time() + ".";
  if (date)
    return "Current date is " + get_date() + ".";
  return "I dont know what to do.";
}
```

shutdown_pc Příkaz, který vypne počítač. Uživatel má možnost vypnutí přerušit po dobu jedné minuty, pokud pošle jakoukoliv zprávu. Metoda `CommandsHolder::safe_data_exchange` znázorňuje výměnu dat mezi uživatelem a metodou `Command::execute`, parametrem je zpráva určená k vypsání uživateli.

```
string ShutDownCommand::execute(ProcessedMessage
  const & message_data, CommandsHolder * command_holder)
{
  system("shutdown -s -t 60");
  string message = "Your PC will shutdown in 60 seconds"+
    ",any command will cancel this operation";
  command_holder->safe_data_exchange(message);
  system("shutdown -a");
  return "Shutdown aborted.";
}
```

describe_yourself Tato metoda znázorňuje využití výjimek ke zpracování jen některých zpráv. Pokud uživatel nechce získat další informace, tedy neodpoví souhlasem, metoda `Command::fallthrough` vyhodí výjimku, kterou poté odchytí `CommandsHolder`, a ten využije data z výjimky ke spuštění jiné funkcionality.

```
string DescribtionCommand::execute(ProcessedMessage
  const & message_data, CommandsHolder * command_holder)
{
  string message = "I am your personal assistent Steve"+
    ", I can tell you more about my functions if you wish.";
  auto data = command_holder->safe_data_exchange(message);
  if (data.get_intent() != "affirm")
    fallthrough(data);
  return functions_describtion();
}
```

Je implementováno sice malé množství funkcionalit, ale prokázal jsem, že asistent je schopný vést konverzaci s uživatelem, zpracovávat jeho zprávy a být interaktivní. Je tedy položen kvalitní základ umožňující konkurovat asistentům popsaným na začátku této práce, a to především díky návrhu kvalitní architektury.

Kapitola 9

Testování

Vzhledem k tomu, že se nejedná o aplikaci provádějící náročné výpočty či práci s důležitými daty, je nejdůležitější otestovat, jestli zvládne zpracovat alespoň základní množinu zpráv a převést je na funkcionality.

Vybral jsem případy užití, které jsou větvené, a tudíž u nich mohou nastat chyby. Pro každý testovaný případ uvedu scénář představující očekávaný průběh konverzace a následně přidám konkrétní snímky obrazovky.

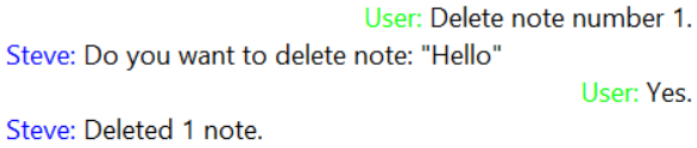
delete_note Předpoklad k tomuto průběhu je vytvoření poznámky s obsahem „Hello“. Poznámka je vytvořena s označení 1, toto označení lze zjistit požadavkem na vypsání všech poznámek. Obrázek 9.1.

Uživatel Delete note number 1.

Asistent Should I really delete note: ”Hello”.

Uživatel Yes.

Asistent Note deleted.



Obrázek 9.1: Testování záměru delete_note

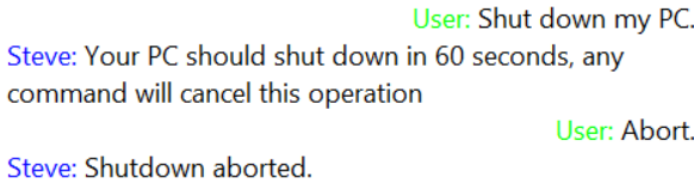
shutdown_pc Systém po odeslání první zprávy zobrazí úspěšně tabulku s oznámením, že systém se do minuty vypne. Zrušení této operace také funguje správně. Obrázek 9.2.

Uživatel Shut down my PC.

Asistent PC will shut down in 60s, any message will abort process.

Uživatel Abort.

Asistent Operation aborted.



Obrázek 9.2: Testování záměru shutdown_pc

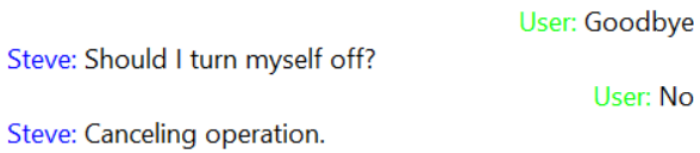
goodbye Po rozloučení se asistent může vypnout, uživatel může buďto souhlasit, či ne. Obrázek 9.3.

Uživatel Goodbye.

Asistent Should i turn myself off?

Uživatel No.

Asistent Okay.



Obrázek 9.3: Testování záměru goodbye

describe_yourself Při popisu svých funkcí může asistent zobrazit kompletní přehled svých funkcionalit, nebo lze přejít k jiné funkcionalitě. Oba případy lze vidět na obrázku. Zde je v implementaci využita metoda s výjimkami popsána během návrhu. Obrázek 9.4.

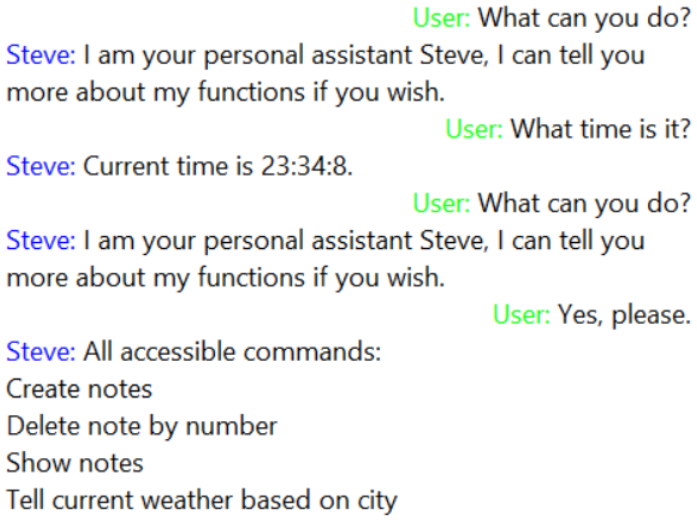
Uživatel What can you do?

Asistent I am Steve, I can tell you more if you wish.

Uživatel Yes. / Something else.

Asistent … (2 rozdílné odpovědi)

Funkční požadavky jsou splněny, co se týče nefunkčních požadavků, jsou také dodrženy – aplikace díky RASA NLU může fungovat bez připojení k internetu a využití zdrojů je popsáno v další kapitole. Použité technologie jsou shodné s technologiemi vybranými v teoretické části.



Obrázek 9.4: Testování záměru describe_yourself

9.1 Zhodnocení

Aplikace splňuje vše, co lze od inteligentního osobního asistenta očekávat, zvládá zatím jen pár příkazů, ale je připravená na rozšíření o další. Implementace funkcí je jednoduchá a velice agilní, jak bylo znázorněno v příslušných kapitolách o implementaci.

Po stránce výkonu aplikace mírně ztrácí. Zprávy zpracovává v řádu vteřin, ale projevila se cena za využití lokálního modulu pro NLU. Aplikaci trvá několik minut, než je schopna vše načíst, pro lepší informovanost uživatele jsem přidal zprávu, kterou uživatel obdrží, jakmile je asistent připraven pracovat. Toto vše se také projevilo na celkovém využití RAM paměti. Využití se pohybuje okolo 5MB pro samotného asistenta, ale razantně vzroste po načtení RASA NLU. Celkový přehled využití zdrojů počítače lze vidět v tabulce 9.1.

	Vytížení
CPU	0%
RAM	710MB
Síť	0Mb/s

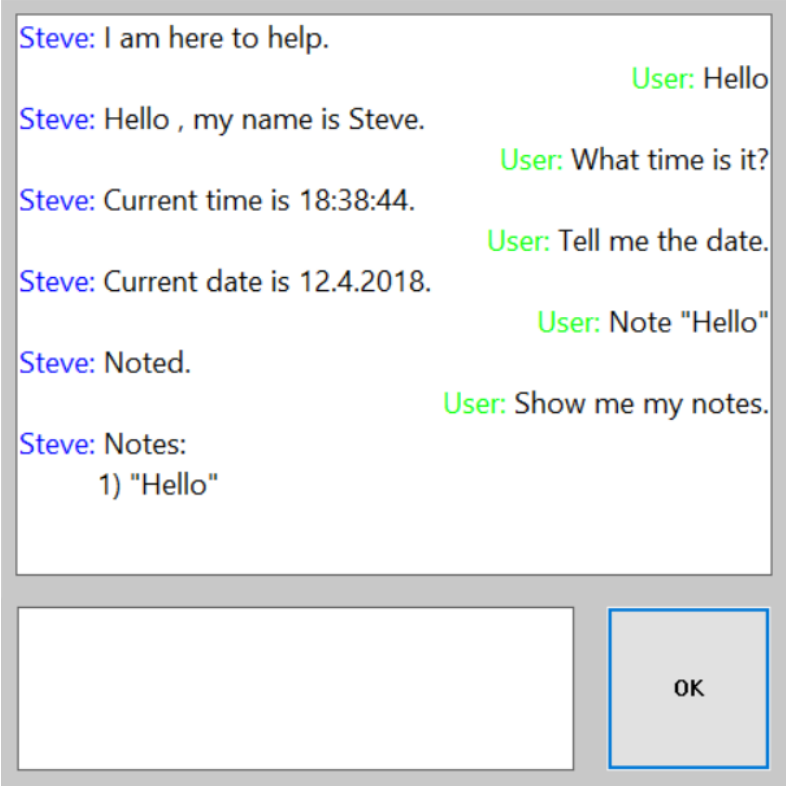
Tabulka 9.1: Vytížení PC během klidového stavu asistenta.

U aplikace se předpokládá, že ji uživatel zapne spolu se zapnutím počítače a v průběhu používaní mu bude asistent k dispozici. Důraz byl kladen na synchronizaci jednotlivých vláken, a tudíž vytížení procesoru je v klidovém stavu (tj. doba kdy se nezpracovávají zprávy) nulové.

Asistent je schopný fungovat bez připojení k internetu, ale funkce založené

na využívání API vystavených na internetu nebudou k dispozici. Funkcionality, jako jsou třeba poznámky, má uživatel k dispozici stále. Výslednou podobu asistentova GUI spolu s ukázkou fungování lze vidět na obrázku 9.5.

Navíc lze stisknout CTRL + . a asistent se přemístí do popředí.



Obrázek 9.5: Snímek obrazovky s asistentem Steve

9.2 Další možná rozšíření

Asistent je hotový, ale stále se dá vylepšit. Za konkurencí v tuto chvíli zaostává hlavně v množství podporovaných funkcí. Pro některé uživatele může také využití RAM paměti představovat zátěž, v tomto případě by šlo využít metody emulace jiné knihovny, kterou RASA NLU nabízí, a využít online řešení. Uživatel by pak měl možnost si vybrat knihovnu nejlépe vyhovující jeho potřebám.

Dalším možným vylepšením je přidání podpory skriptování. C++ neumožňuje přidávat další kód po zkompilování, pokud by ale šlo definovat funkcionality například pomocí Pythonu, mohl by si každý uživatel svého asistenta rozšířit o funkce, které by mu nejvíce vyhovovaly, a měl by možnost si je i sám vytvářet.

Pro zkvalitnění interakce s uživatelem lze rozšířit GUI o oblast určenou k zobrazení jiného obsahu, než jen text, tím mohou být nejrůznější grafy či obrázky. To by umožnilo vytvářet sofistikovanější funkce, které by zvládly lépe předat informace. Architektura je na takový zásah připravena v podobě API, kdy lze definovat třídu obsluhující danou zobrazovací oblast a metoda `Command::execute` má možnost prostřednictvím API měnit obsah.

Závěr

Během několika iterací jsem dosáhl návrhu architektury, který umožňuje implementovat pokročilé funkce asistenta, tedy plynulou konverzaci, jak bylo ukázáno během testování. Tato architektura je také připravena na případné rozšíření o skriptovací jazyk.

Implementace aplikace zvládá všechny funkce definované během návrhu. Aplikaci lze spouštět na lokálním stroji s operačním systémem Windows. Podle výsledků je nejslabším článkem část zodpovědná za zpracování přirozeného jazyka.

Díky strojovému učení je rozpoznání zpráv a jejich následné navázání na funkcionality velice přesné a schopné zpracovat i drobné překlepy. Problém se ovšem objeví, pokud uživatel zadá zprávy, které nemají definované rozdělení, a tudíž dojde ke špatnému zpracování.

Práce mi umožnila seznámit se s fungováním inteligentních asistentů, aktuálními technologiemi a rozšířil jsem si zkušenosti v oblasti návrhu většího softwaru.

Bibliografie

1. MUTCHLER, AVA. *Voice Assistant Timeline: A Short History of the Voice Revolution - Voicebot* [online]. 2017 [cit. 2018-05-06]. Dostupné z: https://www.voicebot.ai/2017/07/14/timeline-voice-assistants-short-history-voice-revolution/.
2. AMAZON MOBILE LLC. *Amazon Alexa – Aplikace na Google Play* [online]. 2018 [cit. 2018-04-04]. Dostupné z: https://play.google.com/store/apps/details?id=com.amazon.dee.app&hl=cs.
3. AMAZON. *Amazon Echo (2nd generation) — Alexa Speaker* [online]. © 1996–2018 [cit. 2018-04-04]. Dostupné z: https://www.amazon.com/gp/product/B0749WVS7J.
4. AMAZON. *Alexa Voice Service* [online]. © 2010–2018 [cit. 2018-04-04]. Dostupné z: https://developer.amazon.com/alexa-voice-service.
5. PROTECT AMERICA. *Will Alexa Work Without Wi-Fi? A Guide | Protect America* [online]. 2017 [cit. 2018-04-04]. Dostupné z: https://www.protectamerica.com/home-security-blog/tech-tips/will-alexa-work-without-wifi-a-guide_15493.
6. APPLE. *iOS - Siri - Apple* [online]. © 2018 [cit. 2018-04-04]. Dostupné z: https://www.apple.com/ios/siri/.
7. DIMITROV, Plamen. *Siri now actively used on 500+ million iOS devices, adoption rate grows by 25% in 7 months* [online]. 2018 [cit. 2018-04-04]. Dostupné z: https://www.phonearena.com/news/Siri-used-on-500-million-iOS-devices-adoption-rate-grows-by-25-percent_id101908.
8. GOOGLE. *Google Assistant - Make Google do it* [online] [cit. 2018-04-04]. Dostupné z: https://assistant.google.com/.

9. GOOGLE. *Google Home - Smart Speaker & Home Assistant - Google Store* [online] [cit. 2018-04-04]. Dostupné z: https://store.google.com/us/product/google_home.

10. TOOMBS, Cody. *The Google Android App Now Supports Limited Voice Commands For Offline Use* [online]. 2015 [cit. 2018-04-04]. Dostupné z: https://www.androidpolice.com/2015/09/28/the-google-android-app-now-supports-limited-voice-commands-for-offline-use/.

11. TWIT NETCAST NETWORK. *Siri v Google Assistant - YouTube* [online]. 2018 [cit. 2018-04-04]. Dostupné z: https://www.youtube.com/watch?v=ehjX0fGG9Nc.

12. MICROSOFT. *Cortana | Your Intelligent Virtual & Personal Assistant | Microsoft* [online]. © 2018 [cit. 2018-04-05]. Dostupné z: https://www.microsoft.com/en-us/windows/cortana.

13. MICROSOFT. *Microsoft Cortana – Digital assistant – Aplikace na Google Play* [online]. 2018 [cit. 2018-04-05]. Dostupné z: https://play.google.com/store/apps/details?id=com.microsoft.cortana&hl=cs.

14. MICROSOFT. *Cortana Skills* [online]. © 2017 [cit. 2018-04-05]. Dostupné z: https://www.microsoft.com/en-us/cortana/skills/featured.

15. MICROSOFT. *Cortana and privacy – Microsoft privacy* [online]. © 2018 [cit. 2018-04-05]. Dostupné z: https://privacy.microsoft.com/en-us/windows-10-cortana-and-privacy.

16. SAMSUNG. *Bixby | Aplikace | Samsung Česká republika* [online]. © 1995–2018 [cit. 2018-05-07]. Dostupné z: http://www.samsung.com/cz/apps/bixby/.

17. WINKELMAN, Steven. *How to Use Bixby | Everything You Need to Know | Digital Trends* [online]. 2018 [cit. 2018-05-07]. Dostupné z: https://www.digitaltrends.com/mobile/samsung-bixby-how-to-use/.

18. MYCROFT AI. *Mycroft Mark 1 - Mycroft* [online]. © 2017 [cit. 2018-05-07]. Dostupné z: https://mycroft.ai/mark1/.

19. MYCROFT AI. *Mycroft Documentation - Mycroft* [online]. © 2017 [cit. 2018-05-07]. Dostupné z: https://mycroft.ai/documentation/.

20. REID, Kathy. *Updated status or wiki for offline use - General Discussion - Mycroft Community Forum* [online]. 2018 [cit. 2018-05-07]. Dostupné z: https://community.mycroft.ai/t/updated-status-or-wiki-for-offline-use/3154.

21. WIT.AI. *Wit.ai* [online]. © 2018 [cit. 2018-04-09]. Dostupné z: https://wit.ai/.

22. DIALOGFLOW. *Agents | Dialogflow* [online] [cit. 2018-04-10]. Dostupné z: https://dialogflow.com/docs/agents.

23. DIALOGFLOW. *Integrations | Dialogflow* [online] [cit. 2018-04-10]. Dostupné z: https://dialogflow.com/docs/integrations/.

24. DIALOGFLOW. *Dialogflow* [online] [cit. 2018-04-10]. Dostupné z: https://dialogflow.com/pricing/.

25. DIALOGFLOW. *Dialogflow* [online] [cit. 2018-04-10]. Dostupné z: https://dialogflow.com/.

26. MICROSOFT. *Cognitive Services APIs Reference* [online]. © 2017 [cit. 2018-04-09]. Dostupné z: https://westus.dev.cognitive.microsoft.com/docs/services/5890b47c39e2bb17b84a55ff/operations/5890b47c39e2bb052c5b9c2f.

27. MICROSOFT. *Pricing - Language Understanding Intelligent Services API | Microsoft Azure* [online]. © 2018 [cit. 2018-04-09]. Dostupné z: https://azure.microsoft.com/en-us/pricing/details/cognitive-services/language-understanding-intelligent-services/.

28. MICROSOFT. *Language Understanding (LUIS) | Microsoft Azure* [online]. © 2018 [cit. 2018-04-09]. Dostupné z: https://azure.microsoft.com/en-us/services/cognitive-services/language-understanding-intelligent-service/.

29. MICROSOFT. *Microsoft Bot Framework* [online]. © 2017 [cit. 2018-04-09]. Dostupné z: https://dev.botframework.com/.

30. MICROSOFT. *Support localization using LUIS apps in Azure | Microsoft Docs* [online] [cit. 2018-04-09]. Dostupné z: https://docs.microsoft.com/en-us/azure/cognitive-services/LUIS/luis-supported-languages.

31. MICROSOFT. *LUIS: Homepage* [online]. © 2017 [cit. 2018-04-09]. Dostupné z: https://www.luis.ai/home.

32. MYCROFT AI. *Adapt - Mycroft* [online]. © 2017 [cit. 2018-04-10]. Dostupné z: https://mycroft.ai/documentation/adapt/.

33. WIT.AI. *Wit.ai* [online]. © 2018 [cit. 2018-04-09]. Dostupné z: https://wit.ai/jobs.

34. WIT.AI. *Wit — HTTP API* [online] [cit. 2018-04-09]. Dostupné z: https://wit.ai/docs/http/20170307.

35. WIT.AI. *Wit — Recipes* [online] [cit. 2018-04-09]. Dostupné z: https://wit.ai/docs/recipes.

36. WIT.AI. *Wit.ai* [online]. © 2018 [cit. 2018-04-09]. Dostupné z: https://wit.ai/faq.

37. RASA TECHNOLOGIES GMBH. *The Rasa dialogue engine — Rasa Core 0.8.5 documentation* [online]. © 2017 [cit. 2018-04-10]. Dostupné z: https://core.rasa.com/index.html.

38. RASA TECHNOLOGIES GMBH. *Language Understanding with rasa NLU — rasa NLU 0.11.3 documentation* [online]. © 2017 [cit. 2018-04-10]. Dostupné z: https://nlu.rasa.com/index.html.

39. RASA TECHNOLOGIES GMBH. *Motivation — Rasa Core 0.8.4 documentation* [online]. © 2017 [cit. 2018-04-10]. Dostupné z: https://core.rasa.com/motivation.html.

40. RASA TECHNOLOGIES GMBH. *Rasa: Open source conversational AI for enterprise* [online]. © 2018 [cit. 2018-04-10]. Dostupné z: http://rasa.com/.

41. RASA TECHNOLOGIES GMBH. *Language Support — Rasa NLU 0.11.3 documentation* [online]. © 2017 [cit. 2018-04-10]. Dostupné z: https://nlu.rasa.com/languages.html.

42. RASA TECHNOLOGIES GMBH. *Configuration — Rasa NLU 0.11.3 documentation* [online]. © 2017 [cit. 2018-04-10]. Dostupné z: https://nlu.rasa.com/config.html.

43. RASA TECHNOLOGIES GMBH. *Using Rasa NLU as a HTTP server — Rasa NLU 0.11.3 documentation* [online]. © 2017 [cit. 2018-04-10]. Dostupné z: https://nlu.rasa.com/http.html.

44. RASA TECHNOLOGIES GMBH. *Interactive Learning — Rasa Core 0.8.5 documentation* [online]. © 2017 [cit. 2018-04-10]. Dostupné z: https://core.rasa.com/tutorial_interactive_learning.html.

45. *Snips Natural Language Understanding — Snips NLU 0.13.3 documentation* [online] [cit. 2018-05-07]. Dostupné z: https://snips-nlu.readthedocs.io/en/latest/index.html.

46. CODEMENTOR. *Why Learn C++ - Best Programming Language* [online]. 2016 [cit. 2018-04-16]. Dostupné z: http://www.bestprogramminglanguagefor.me/why-learn-c-plus-plus.

47. PYTHON. *What is Python? Executive Summary | Python.org* [online]. © 2001–2018 [cit. 2018-04-16]. Dostupné z: https://www.python.org/doc/essays/blurb/.

48. PYTHON. *Comparing Python to Other Languages | Python.org* [online]. ©2001–2018 [cit. 2018-04-16]. Dostupné z: https://www.python.org/doc/essays/comparisons/.

49. LEAHY, Paul. *What Is Java Computer Programming Language?* [online]. 2018 [cit. 2018-04-16]. Dostupné z: https://www.thoughtco.com/what-is-java-2034117.

50. *UTF-8 and Unicode Standards* [online]. 2014 [cit. 2018-04-17]. Dostupné z: http://www.utf-8.com/.

51. REDDY, Martin. *API design for C*. Boston: Morgan Kaufmann, 2011. ISBN 978-0-12-385003-4.

52. JSON. *JSON* [online] [cit. 2018-04-16]. Dostupné z: `http://www.json.org/json-cz.html`.

53. HASSMAN, Martin. *JSON : jednotný formát pro výměnu dat - Zdroják* [online]. 2008 [cit. 2018-04-16]. Dostupné z: `https://www.zdrojak.cz/clanky/json-jednotny-format-pro-vymenu-dat/`.

54. MICROSOFT. *Rich Edit (Windows)* [online]. © 2018 [cit. 2018-05-06]. Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/desktop/bb787605(v=vs.85).aspx`.

Příloha A

Seznam použitých zkratek

API Application Programming Interface

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

NLP Natural Language Processing

NLU Natural Language Understanding

RAM Random-access Memory

XML Extensible Markup Language

Příloha B

Obsah přiloženého DVD

- `readme.txt` ..... stručný popis obsahu DVD
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF