Sem vložte zadání Vaší práce.

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

Bakalářská práce

LUKA – Pomocník pro japonštinu

Klára Hájková

Vedoucí práce: Ing. Martin Kopp

1. května 2015

Poděkování

Děkuji svému vedoucímu Ing. Martinu Koppovi za rady, ochotu a podnětné připomínky a své rodině za morální podporu. Také děkuji všem japonštinářům, kteří se podíleli na výběru funkcí a otestování aplikace, speciálně děkuji Aleně Hnilicové, jejíž rady pro mne byly nejužitečnější. A konečně děkuji společnosti Oracle za vynikající dokumentaci a návody k Javě.

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 1. května 2015 .....

Odkaz na tuto práci

Hájková, Klára. *LUKA – Pomocník pro japonštinu.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Cílem této práce je vytvořit funkční prototyp pomocníka při učení japonského jazyka pro české studenty. V první části byla provedena detailní rešeršní analýza používaných nástrojů, z níž vychází seznam funkcí zahrnutých do návrhu LUKA. V implementovaném prototypu tedy nechybí editor, slovník, vyhledávání japonských znaků kandži dle mnoha kritérií a rozpoznávání znaků pomocí výpočetní inteligence.

Klíčová slova japonština, umělé neuronové sítě, java, javafx

Abstract

The aim of this thesis is to develop a working prototype of a learning assistant for students of the Japanese language. In the first part, a detailed research analysis of used tools was conducted. This analysis resulted in a list of features included in the design of LUKA. In the implemented prototype there is an editor, dictionary, search engine of japanese kanji characters based on many criteria, and recognition of characters using computational intelligence.

Keywords japanese, artificial neural networks, java, javafx

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Práce s japonštinou na počítači i v dnešní době zůstává nesnadným úkolem, obzvláště, pokud se nejedná o váš rodný jazyk. Nástroje, jenž by uživateli mohly tuto práci usnadnit, neobsahují většinu potřebných funkcí. Uživatelům tedy nezbývá nic jiného, než se některých funkcí úplně vzdát, nebo pracovat s více programy.

Ovšem tento postup může skýtat jistá úskalí – například se k programům často nedodávají potřebné fonty a pro správnou funkčnost musejí být dohledány a nainstalovány ručně. Což samozřejmě platí, jak pro online aplikace, tak i pro ty, které je nutné instalovat. Dále pak při samotné práci se mohou objevit problémy s kódováním při přenosu dat z jedné aplikace do jiné. Nehledě na to, že valná většina těchto programů není nativně dostupná pro Unixové systémy. To jsou prvotní důvody, které podnítily vznik pomocníka pro studium japonštiny – LUKA.

LUKA je multiplatformní učební pomocník, jehož seznam funkcí obsahuje vše, co studenti japonštiny nejvíce používají. Jeho zaměření právě na české studenty z něho dělá naprosto unikátní program. Většina ostatních nástrojů si totiž cestu k českému studentovi našla přes mezikrok v podobě angličtiny. Uživatelé museli k zápisu znaků používat anglický přepis a hledat v anglicko-japonském a japonsko-anglickém slovníku. Částečně za to může fakt, že neexistují volně dostupné české slovníky. LUKA tedy za pomoci českých japanologů pokládá jejich základ.

Slovník tedy patří do výbavy LUKA, ačkoliv v rané verzi (této) stále využívá anglických zdrojů. Jakmile budou k dispozici česká data, slovníky se sloučí a uživatel si bude moci vybrat, chce-li používat anglický, český, či oba najednou.

Velmi důležitou funkci zastává vyhledávání znaku kandži podle mnoha kritérií. Nejvýznamnějším bodem je rozpoznávání na obrazovce označeného znaku, implementované pomocí výpočetní inteligence. Uživatel taktéž může využít vyhledávání podle třech japonských čtení, počtu tahů, vizuálních složek, ze kterých se znak skládá a dalších možností. Všechny tyto možnosti lze

kombinovat, aby uživatel využil všech informací, jež o znaku má k tomu aby mezi tisíci našel ten, který ho zajímá.

Vlastní editor, vyhledávání i slovník jsou propojené – z jednotlivých komponent lze vyhledávat ve slovníku, zobrazovat informace o jednotlivých znacích, či vkládat do editoru.

Pro práci s editorem LUKA nepotřebujete japonskou klávesnici, konkrétní operační systém ani připojení k internetu.

Kapitola 1

Japonština

Vzhledem k tomu, že se práce zabývá japonštinou, věnuji tuto kapitolu krátkému úvodu do jazyka a vysvětlení některých jeho specifik, jež jsou pro pochopení provedených rozhodnutí nezbytná. Také je potřeba zavést konvenci fonetického přepisu v dalším textu – mnohým může být blízký tzv. Hepburnův přepis (přizpůsobený pro anglicky mluvící), nicméně pro českého čtenáře je česká transkripce mnohem přirozenější.

Začneme japonskou typografií, ta se od té české velice liší. Na první pohled nejzjevnější odlišnosti:

- Směr písma – lze psát buď zprava doleva po sloupcích nebo klasicky zleva doprava po řádcích.

- Všechny znaky i interpunkční znaménka se píší do pomyslných čtverců. Jedná se tedy o neproporcionální písmo. Pro katakanu existuje ještě tzv. „half-width“ verze, která se liší jen svou poloviční šířkou, ale tou se ve své práci nezabývám.

- Mezi slovy se nepíší mezery. Je-li potřeba vizuálně rozdělit např. cizí slova kvůli pochopení, vkládá se mezi znaky puntík doprostřed řádku – テレビ·チャンネル (televizní kanál). Stejný puntík se používá i jako spojovník.

- Místo tečky se píše kroužek a čárka má jiný sklon – はい、そうです。 (Ano, je to tak.)

- Jiné zvýrazňování textu – nepoužívá se změna řezu písma, nýbrž se vkládají čárky vpravo od znaku, u vodorovného textu pod ním. Hiraganu lze případně zvýraznit přepsáním do katakany.

- Uvozovky mají podobu nedokončených hranatých závorek – 「」.

Ale japonština se od češtiny liší především písmem. V češtině používáme jen latinku, kdežto v japonštině se souběžně píše třemi druhy abeced. Jsou jimi:

- kana,
- kandži,
- rómadži (latinka).

1.1 Kana

Kanou se souhrnně nazývají dvě fonetické slabičné abecedy – hiragana (ひらがな) a katakana (カタカナ). Zatímco kandži znaky (viz níže) se vyjadřuje kořen slova, hiragana se používá hlavně pro gramatické koncovky a partikule. Katakana slouží převážně pro slova přejatá z cizích jazyků, nebo jako zvýraznění textu – podobně jako u nás kurzíva.

Hiragana i katakana sestávají ze 45 slabik a neslabičného *n*, jak ukazuje tabulka 1.1. pod každým znakem se kvůli nepravidelnostem uvádí i jeho čtení. Například „つ“ se místo „tu“ čte „cu“.

Avšak tím obsah kany nekončí. Pomocí diakritického znaménka *nigori* ゛ se mění řady *ka, sa, ta, ha* na znělé *ga, za, da, ba*. U řady *ha* se ještě používá *marunigori* ゜, který ji mění na řadu *pa*. Tyto změny zachycuje tabulka 1.2.

Další specialitou kany jsou spřežky (např. v češtině máme spřežku „ch“), které se tvoří slabikou končící na -i spolu s malým ja, ju nebo jo, například きゃ (kja v hiraganě) nebo キュ (kjo v katakaně).

1.1.1 Speciální znaky

Prodlužování samohlásek se v hiraganě liší od katakany. V hiraganě se např. „á“ napíše zdvojením „ああ“, „é“ jako „えい“ atd. Ovšem v katakaně se dlouhé samohlásky tvoří speciálním znakem „ー“, takže „á“ se zapíše jako „アー“.

Dalším speciálním znakem kany je malé cu (っ resp. ッ). Používá se při zdvojení souhlásek. Například „げっこう“ se do češtiny přepíše jako „gekkó“ (měsíční záře). Ovšem pokud se zdvojuje některá slabika z řady *n*, používá se k tomu „ん“ resp. „ン“, např. „たんに“ přepsáno jako „tanni“ (jednoduše).

Pro znaky kany se užívá opakovacích znamének, zapíše-li se znaménko za znak, zduplikuje ho. Pro hiraganu jsou to „ゝ“ a pro znělé řady „ゞ“, respektive „ヽ“ a „ヾ“, pro katakanu.

Kvůli přesnějšímu přepisu cizích slov byla katakana rozšířena. Například přidáním *nigori* ke znaku ウ(u) vzniklo ヴ(v), existuje i verze pro hiraganu, ale téměř se nepoužívá. Vznikly tímto způsobem i nové spřežky, třeba ファ(fa), フィ(fi), フェ(fe), フォ(fo).

Základní řady									
Hiragana					Katakana				
あ	い	う	え	お	ア	イ	ウ	エ	オ
a	i	u	e	o	a	i	u	e	o
か	き	く	け	こ	カ	キ	ク	ケ	コ
ka	ki	ku	ke	ko	ka	ki	ku	ke	ko
さ	し	す	せ	そ	サ	シ	ス	セ	ソ
sa	ši	su	se	so	sa	ši	su	se	so
た	ち	つ	て	と	タ	チ	ツ	テ	ト
ta	či	cu	te	to	ta	či	cu	te	to
な	に	ぬ	ね	の	ナ	ニ	ヌ	ネ	ノ
na	ni	nu	ne	no	na	ni	nu	ne	no
は	ひ	ふ	へ	ほ	ハ	ヒ	フ	ヘ	ホ
ha	hi	fu	he	ho	ha	hi	fu	he	ho
ま	み	む	め	も	マ	ミ	ム	メ	モ
ma	mi	mu	me	mo	ma	mi	mu	me	mo
や		ゆ		よ	ヤ		ユ		ヨ
ja		ji		jo	ja		ji		jo
ら	り	る	れ	ろ	ラ	リ	ル	レ	ロ
ra	ri	ru	re	ro	ra	ri	ru	re	ro
わ				を	ワ				ヲ
wa				(w)o	wa				(w)o
		ん					ン		
		n					n		

Tabulka 1.1: Základní řady slabičných abeced s českým čtením

Znělé řady									
Hiragana					Katakana				
が	ぎ	ぐ	げ	ご	ガ	ギ	グ	ゲ	ゴ
ga	gi	gu	ge	go	ga	gi	gu	ge	go
ざ	じ	ず	ぜ	ぞ	ザ	ジ	ズ	ゼ	ゾ
za	dži	zu	ze	zo	za	dži	zu	ze	zo
だ	ぢ	づ	で	と	ダ	ヂ	ヅ	デ	ド
da	dži	zu	de	do	da	dži	zu	de	do
ば	び	ぶ	べ	ぼ	バ	ビ	ブ	ベ	ボ
ba	bi	bu	be	bo	ba	bi	bu	be	bo
ぱ	ぴ	ぷ	ぺ	ぽ	パ	ピ	プ	ペ	ポ
pa	pi	pu	pe	po	pa	pi	pu	pe	po

Tabulka 1.2: Přidáním nigori a marunigori se některé řady kany mění na znělé

1.2 Kandži

Kandži (漢字) jsou znaky, které Japonsko v minulosti převzalo z Číny. Časem se z některých čínských znaků postupným zjednodušováním stala kana. A protože čínština i japonština jsou velmi odlišné jazyky (v Japonštině chybí především tóny ve výslovnosti), ke znakům přibylo japonské čtení. Takže v dnešní době může mít jeden znak dvě i více čtení, která se určují z kontextu. Rozlišujeme dva základní druhy čtení:

1. japonské (kun) – zapisované hiraganou,
2. sinojaponské (on) – původní čínské čtení, upravené pro japonštinu, zapisované katakanou.

Existuje ještě mnoho dalších druhů, do LUKA editoru bylo zahrnuto ještě tzv. nanori čtení, jež se užívá, pokud se daný znak kandži objeví ve jméně. Například pro znak 人 (osoba) vypadá čtení následovně:

čtení	kana	přepis
japonské	ひと	hito
sinojaponské	ジン	džin
nanori	ふみ	fumi

Přesný počet všech kandži není znám, ale známých znaků je několik desítek tisíc. Tolik se jich Japonci samozřejmě neučí. Aby se nějakým způsobem standardizovalo, které znaky je třeba znát, vznikly různé seznamy. Například soubor 常用漢字 (džójó kandži) obsahující 1945 znaků se vyučuje na středních školách, s touto znalostí už si studenti mohou přečíst běžný denní tisk. Co se týče digitalizované podoby, tak v současnosti je nejpoužívanějším souhrnem znaků stále JIS X 0208, ten obsahuje 6879 znaků.

Aby bylo možné znaky kandži nějakým způsobem řadit, třídit a vyhledávat v nich, existují kromě např. počtu tahů další speciální *třídy*.

1.2.1 Radikály

Radikálem nazýváme grafém, podle něhož jsou skupiny znaků řazeny ve slovnících. V některých případech radikál naznačuje oblast, do které daný znak svým významem spadá (např. člověk, slovo, mysl apod.), ale bohužel to nelze tvrdit obecně – existuje mnoho znaků, jež vznikly např. naznačením původního čínského čtení, nebo se jejich význam v průběhu historického vývoje velmi posunul. V současnosti se počet klasických radikálů ustálil na čísle 214. [1]

Ovšem seznam radikálů dnes není jednoznačným pojmem. Mohou za to nástroje, které *umožňují vyhledávání dle radikálů*, přičemž vlastně hledají podle vizuálních složek znaků a nikoliv klasických radikálů. Pravdou je, že seznam klasických radikálů a vizuálních složek se částečně překrývají, ale složek je více – 250.

1.2.2 Úrovně JLPT

Japanese-Language Proficiency Test je celosvětově uznávaná zkouška z japonštiny pro nerodilé mluvčí, která se (podle nejnovějších změn z roku 2010) dělí na pět úrovní – N5 (nejjednodušší) až N1 (nejtěžší). Pro složení zkoušky z každé úrovně, musí uchazeč umět určitou sadu znaků. Pro úroveň N1 se počet kandži šplhá až ke dvěma tisícům. Lidé, kteří se na JLPT připravují, často potřebují jen znaky pro danou úroveň plus všechny předchozí, proto existují specifické seznamy znaků právě pro JLPT. Vzhledem k tomu, že tyto zkoušky se konají i u nás, nemohou být JLPT seznamy v této práci opomíjeny.

Kapitola 2

Rešerše

Tato kapitola se věnuje analýze existujících editorů a nástrojů, které jsou českými studenty nejvíce používány a jejichž některé funkce se staly inspirací pro LUKA.

Pro podrobnější a přesnější analýzu byl mezi studenty rozeslán dotazník, ve kterém zhodnotili přednosti a nedostatky těchto nástrojů. Analýza tak zahrnuje nejen mé testování, ale i jejich zkušenosti. Dotazník odeslalo více než 70 respondentů.

2.1 WaKan

WaKan se prezentuje jako komplexní nástroj pro studenty nejen japonského, ale i čínského jazyka. Filip Kábrt jej napsal v jazyce Delphi a první verze spatřila světlo světa už v roce 2003 [2]. Za dva roky intenzivního vývoje přibylo velké množství užitečných funkcí. Od té doby se ve WaKanu nic důležitého nezměnilo.

2.1.1 Funkce

Obsahuje textový editor, znakový slovník a mimo japonštiny ještě obdobné rozhraní pro čínštinu. Funguje kompletně v češtině.

Editor má vlastní japonské IME podporující tři romanizační systémy – český, anglický (Hepburnův) a japonský (Kunreišiki). Také se snaží po spuštění příslušné funkce doplnit překlad (pod znakem) a furiganu (čtení – standardně nad znakem).

Znakový slovník používá soubory EDICT, KANJIDIC a CEDICT. Vyhledávat znaky lze podle mnoha kategorií – japonské nebo sinojaponské čtení, PinYin čtení, radikálu, počtu tahů, kódu Unicode, stupně Džójó, kódu SKIP anebo FourCornerCode. Dále umí najít složeniny s daným znakem, výsledky řadit podle několika kritérií a vytisknout si kartičky se znaky pro lepší učení.

Podporuje uživatelské seznamy slovíček / naučených znaků a jejich umisťování do složek.

2.1.2 Nedostatky

Vzhledem k tomu, že vývoj skončil již v roce 2005, tak program zůstává pozadu. Slovníky, na kterých staví většina funkcí, se časem vyvinuly a používají nový formát – EDICT2. Kvůli tomu nelze WaKan aktualizovat, a tak obsahuje mnoho špatných překladů a zároveň ve vyhledávání zobrazuje i značně zastaralá a nepoužívaná slova.

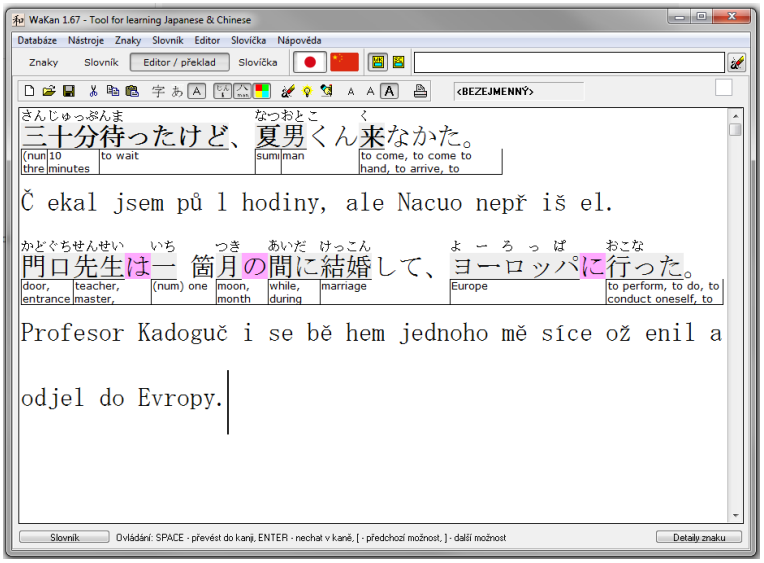
Uživatelské rozhraní by šlo jistě lépe rozvrhnout. Takto se některé užitečné funkce skrývají v nepřehledném menu a uživatel je musí složitě hledat. Často by se uživateli hodila krátká nápověda přímo u používané funkce. Například při vyhledávání pomocí radikálů 1.2.1 se dá vybrat více radikálů podržením klávesy Ctrl, ale na to uživatel musí přijít sám.

Ačkoliv WaKan vytvořil český programátor a přestože to popis programu slibuje, některé popisky v češtině nejsou. Vyhledávat ve slovníku lze sice i s českým přepisem, ale vzhledem k souběžnému používání několika transkripcí se vyhledávají i nechtěné znaky (např. v češtině ju, Hepburn yu). Respondenti v dotazníku uváděli na vyhledávání jen kritiku.

Hlavními nedostatky editoru jsou:

- Významně chybí funkce Zpět a Vpřed (Undo a Redo). Pokud si uživatel omylem smaže kus textu, přišel o něj natrvalo.
- Vkládání ze schránky někdy vloží nesmysl, větší problémy má i s vkládáním formátovaného textu z webu.
- Editor nezvládá českou diakritiku, viz Obrázek 2.1.
- Při používání IME je procházení mezi znaky složité – provádí se přes [a], ke kterým se dá na české klávesnici dostat jen přes klávesu Alt.
- Speciální symboly (např. japonskou obdobu spojovníku) si uživatel musí najít někde jinde, protože editor pro ně zvláštní tlačítko ani klávesu nemá.
- Neumožňuje napsat některé *nové* slabiky kany, např. „kwa“ – „くぁ“.
- Pro zapsání malých znaků kany se musí před příslušný romanizovaný text napsat x – xltsu napíše „っ“, což je nestandardní chování – ostatní programy používají l (jako little – malý). Ovšem, aby tohle uživatel zjistil, musí se ponořit do nápovědy, přímo v programu o tom žádná zmínka není.

Studentům japonštiny u Wakanu nejvíce chybí příkladové věty a řazení dle JLPT 1.2.2.



Obrázek 2.1: Editor českého programu WaKan si neporadí s českými znaky.

2.2 JWPce

Autorem JWPce (Japanese Word Processor) je Glenn Rosenthal a počátek vývoje se datuje až do roku 1998. Původně navrhl JWPce pro mobilní zařízení, takže má nízké nároky na systém a paměť. Poslední verze vyšla v roce 2005 [3]. Ačkoliv Glenn na svých stránkách slibuje nové funkce do další verze, 9 let se nic neděje. Nicméně velmi dobře okomentované zdrojové kódy v C++ jsou všem plně k dispozici.

2.2.1 Funkce

JWPce funguje především jako editor a svou roli plní skvěle. Má vlastní IME, postavené na Wnn[1], které konvertuje mezi kanou a kandži znaky (lze přidat i vlastní konverze). O přepis z latinky do kany se stará sám. Velmi chytře řeší psaní katakanou – stačí psát velkými písmeny a není nutné nikde nic přepínat. Podporuje anglický i japonský romanizační systém. Na rozdíl od Wakanu zvládá většinu novějších slabik a japonských symbolů. Co se týče dalšího editování, umí lehké formátování a dokonce i speciální záhlaví a patičku (datum, čísla stránek...).

Dále má v sobě zabudovaný slovník, ve kterém se dá vyhledávat podle mnoha kritérií, např. slang, idiom, archaický výraz atd. Do slovníku lze přidávat vlastní překlady.

[1]Detaily na stránkách: http://www.freewnn.org/

Kdekoliv lze zobrazit okno s podrobnými informacemi o znaku. Samotné vyhledávání znaků lze provádět až devíti různými způsoby. Bohužel jen samostatně, nedají se kombinovat.

Uživatelské rozhraní působí intuitivně a přehledně, ale česká lokalizace není. Pro překlady vytvořil Glenn utilitu, takže kdokoliv může rozhraní editoru snadno přeložit.

Jeho velkou předností je oficiální podpora funkčnosti pod Wine[2], stačí si stáhnout příslušný archiv a upravit makefile.

2.2.2 Nedostatky

I zde se vyskytuje problém zastaralých dat (slovníkové soubory, ve kterých jsou chyby). V některých případech chybně funguje vkládání ze schránky – místo znaků vloží kostičky. Česká diakritika mu taktéž dělá problémy, viz Obrázek 2.2.

JWPce si vytváří vlastní speciální formát, který nelze otevřít v žádném jiném programu, ani jej není možné vyexportovat, což se podepsalo na přenositelnosti souborů.

Značně nelogicky se vkládá malá kana. Před romanizovaný znak se píše +.

Na další problémy jsem v rámci testování nenarazila, ani z analýzy dotazníku žádné neplynou. JWPce byl ve své době opravdu kvalitním programem a dodnes se hojně používá. Například autorky české učebnice japonštiny [4], která vyšla až v roce 2007, jej přikládají k učebním materiálům na CD.

2.3 Denshi Jisho

Denshi Jisho znamená doslova „elektronický slovník“. Najdete ho na adrese jisho.org, provozuje ho Kim Ahlström a je napsaný v jazyce Perl s databází MySQL. Mezi studenty se jedná o nejpoužívanější slovník.

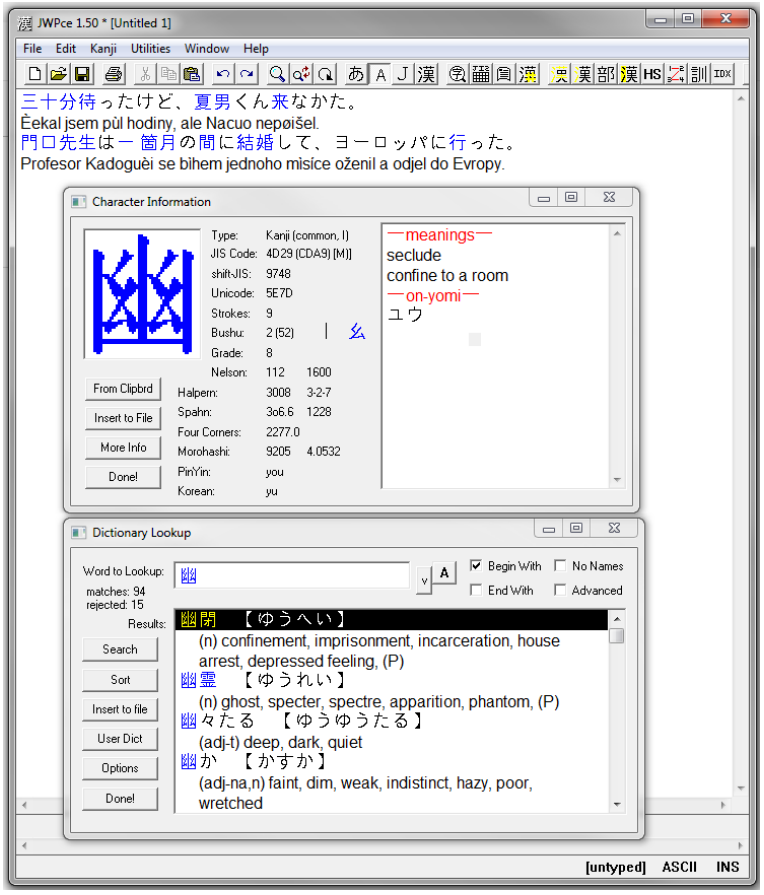
2.3.1 Funkce

Jisho se chová přesně tak, jak člověk od slovníku očekává. Dají se v něm vyhledávat slovíčka i fráze, informace o jednotlivých znacích a navíc obsahuje i příkladové věty s daným výrazem. Velká výhoda spočívá ve vstupu latinkou, text se do kany převede automaticky.

2.3.2 Nedostatky

Žádné velké nedostatky nebo chyby nemá a tak mu lze vytknout jen několik drobností. Například pokud při vyhledávání znaku zadáte čtení i počet tahů,

[2] Aplikace umožňující běh programů pro Microsoft Windows pod (převážně) Unixovými systémy.



Obrázek 2.2: Ani v editoru JWPce není možné psát si české poznámky. Na obrázkou je také okno slovníku a okno informací pro kandži znak.

nenajde nic, přestože by měl. Nikde se nepíše, že umí hledat vždy jen podle jednoho kritéria, uživateli nic nebrání, aby vyplnil všechny kolonky.

Dále by respondenti uvítali, aby se příkladové věty zobrazovaly u daných znaků a nemuselo se k nim proklikávat.

2.4 Ostatní nástroje

V dotazníku se opakovaly i dva další nástroje, jejichž celková funkcionalita nedosahuje výše zmíněných, ale oblíbenost ano, proto v testování a hodnocení nesmějí chybět. Jsou to aplikace pro memorování znaků Anki a rozšíření pro firefox Rikaichan, respektive Google Chrome Rikaikun.

2.4.1 Anki

Aplikace sloužící k memorování znaků, slovíček, frází apod. pomocí kartiček. Avšak nezaměřuje se jen na japonštinu nebo dokonce jen na jazyky – umožňuje v podstatě jakékoliv opakování, například i kytarových akordů. Podporuje audio nahrávky, videa, ale i vstup z LaTeXu. Navíc jej můžete mít všude s sebou, aplikace je dostupná nejen pro Windows, Mac OS a Linux, ale také pro Android a iOS. Přes AnkiWeb probíhá synchronizace těchto zařízení a zároveň si zde lze i opakovat [5].

Celé učení funguje na křivce zapomínání, což v principu znamená, že rozvrhne-li se opakování do delšího časového úseku, dosahuje se lepších výsledků než při naučení všeho najednou.

2.4.2 Rikaichan a Rikaikun

Doplňky do prohlížečů Rikaichan pro Mozillu Firefox a Rikaikun pro Google Chrome se hodí především pokročilejším japonštinářům. Po najetí myší na kandži znak na webové stránce zobrazí jeho čtení, slovní druh a anglický překlad. Klávesou Enter lze získat o znaku další informace, například počet tahů, radikál nebo četnost užití. Což je nesmírně užitečné, protože kvůli vyhledání jednoho neznámého znaku nemusí uživatel otevírat slovník. Oba doplňky čerpají veškeré informace ze souborů JMDICT a KANJIDICT.

Tvůrci i pro tak jednoduchou aplikaci vytvořili široké možnosti nastavení, jež zahrnuje jaké informace se mají zobrazit, klávesové zkratky a práci se schránkou.

2.5 Problém párových znaků

Párovými znaky (tzv. surrogates pairs) se v kódování UTF-16 nazývají 16bitové dvojice, které reprezentují jeden zobrazitelný znak. Horní části jsou v rozsahu od U+D800 do U+DBFF, spodní pak od U+DC00 do U+DFFF. Právě díky používání párových znaků obsáhne kódování UTF-16 až milion znaků.

Příkladem takového znaků může být 0x2A6B2 (32bitově) a jako pár v 16bitovém zápisu: 0xD869 0xDEB2.

K velkému překvapení došlo při testu funkčnosti párových znaků. Žádná z testovaných aplikací si s nimi neumí poradit. Editory JWPce a WaKan tyto znaky neumí zapsat ani zobrazit, nemluvě o vyhledávání. U Denshi Jisho se dá zjistit, že jeho databáze (soubory od Jima Breena) informace o daném párovém znaku obsahuje – vyhledání podle unicode znak nalezne, nicméně jakákoliv další interakce na stránce vyvolává chyby.

Kapitola 3

Analýza

Kapitola nejprve diskutuje výběr funkčních a nefunkčních požadavků a poté se věnuje obecným záležitostem z analýzy a návrhu programu jako celku. V závěru kapitoly se řeší požadavků na nezávislé komponenty. Detaily pro každou komponentu jsou v samostatných kapitolách, jež spojuje kapitola o jádru a celkovému spojení všech komponent dohromady.

3.1 Výběr funkčních požadavků

Cílem je nyní vybrat ty nejpoužívanější funkce z nástrojů z předchozí kapitoly, doplněné o nové požadavky studentů japonštiny. Avšak všechny v této práci implementovány nebudou a zůstanou ve formě návrhu. Jejich soupis a návrh pomohou k implementaci v budoucnu.

3.1.1 IME a transkripce

Vlastní IME hraje v editoru klíčovou roli. Pokud totiž uživatel pracuje na počítači, kde japonský vstup není nainstalovaný (na instalaci nemusí mít ani práva), tak samotný editor ztrácí svou hlavní funkčnost.

Důležitým bodem bylo vybrání transkripce (přepisu) do kany 1.1. Respondenti uvedli jako nejlepší anglickou (43 %), dále českou (40 %) a poslední japonskou (17 %). Rozhodla jsem se tedy v budoucnu do editoru zahrnout všechny tři. V první verzi (tato práce) bude jen anglická a japonská. Česká transkripce bude dodána později, vyžaduje totiž podrobnější analýzu kvůli diakritice (např. *ó* může v kaně vypadat jako おお i おう).

Ovšem IME jako takové vyžaduje kompletní kontrolu nad vstupem, jež je implementačně náročná, tudíž první verze své IME v editoru mít nebude. Jako další úskalí IME se jeví počet možností – kandži znaků existují desetitisíce a většina má více než jedno čtení. Tím pádem jejich zápis není deterministický. Aby se uživateli nezobrazoval seznam stovek irelevantních a nepoužívaných kandži znaků, resp. frází, bude se výběr dělat na základě nějaké heuristické

funkce zohledňující frekvenci používání, kontext a jiné věci. Což je nad rámec této práce.

Ale do všech vyhledávacích polí bude možné již nyní psát latinkou vybranou transkripcí a program si vstup posléze převede do kany.

3.1.2 Vyhledání a zobrazení informací o znaku

Pokud uživatel chce vložit nějaký znak, ale nezná jeho čtení (zápis), rozhodně by kvůli tomu neměl spouštět další aplikaci. Proto si myslím, že vyhledání znaku a jeho následné vložení je na místě. Vyhledání by mělo být co nejpodrobnější, aby uživatel mohl použít veškeré informace, které o znaku má. Proto jsem do těchto kritérií zahrnula čtení, význam, rozsahový počet tahů, radikály, JLPT a další skupiny (například výše zmíněný seznam džójó). Zároveň je ale zbytečné používat zastaralé (metoda čtyř rohů) nebo málo používané (SKIP) metody, stejně tak odkazy na slovníky, protože pokud má uživatel slovník v ruce, nalezne v něm zapsané i jeho čtení. Stěžejní je ovšem vyhledání znaku označeného na obrazovce. Samotné rozpoznávání bude řešeno pomocí výpočetní inteligence, jíž se podrobně věnuji v kapitole 7.

Zároveň ale vyhledání nemusí sloužit pouze k vložení znaku do editoru. Uživatel může chtít vědět, zda-li se daný znak nachází v seznamu ke zkoušce JLPT, jaká jsou jeho další čtení či jak se píše (posloupnost tahů) atd. Zobrazení informací o znaku tedy považuji za užitečnou funkci, jež nechybí ve WaKanu ani v JWPce, nemůže tedy chybět ani zde.

3.1.3 Slovník

Slovník je nepostradatelná funkce v učení jakéhokoliv cizího jazyka, jak ostatně dokazují ostatní aplikace. Již v této verzi, ve které není IME, se díky vstupu v latince stává užitečným a plně funkčním nástrojem.

Další verze již bude zahrnovat uživatelské překlady a jak už bylo řečeno v úvodu, příprava dat v češtině. Žádný česko-japonský volně dostupný slovník zatím neexistuje, což je vzhledem k dnešním technologiím trochu paradoxní. Rozhodla jsem se jej začít pomalu tvořit. Zatím mám svolení od autorek učebnice japonštiny [4] k digitalizaci a použití všech slovíček a frází, jejichž počet sahá k několika tisícům (součástí učebnicového kompletu je také samostatný slovník). Vůli pomoci rozšířit takový slovník projevilo i několik studentů Univerzity Palackého v Olomouci. To jistě bude na počáteční vývoj a testování stačit.

3.1.4 Učení znaků a slovíček

Student japonštiny se na rozdíl od studentů jiných jazyků musí poprat s velkou výzvou – naučení se stovek kandži znaků zpaměti. Spousta z nich dle dotazníku využívá opakovací program Anki. Proto bych ráda do LUKA přidala možnost učení znaků a slovíček nějakou zábavnou formou.

Tisknutí opakovacích kartiček – tzv. „flashcards“ už dnes není tak běžné, místo toho se k příležitostnému opakování používají aplikace na chytrých telefonech. LUKA bude mít v budoucnu svou část také na těchto zařízeních.

3.1.5 Formátování textu

Dnes už málokdo píše dokumenty v obyčejném textovém editoru. Ze základních stavebních kamenů byly jako požadavky pro editor vybrány následující:

- změna velikosti, barvy a fontu,
- tučný řez písma,
- zvýraznění,
- vkládání odkazů,
- zarovnání a odsazení,
- bodové a číselné seznamy.

3.1.6 Další funkce

- Příkladové věty hrají klíčovou roli v učení cizího jazyka. Proto bylo na žádost studentů zařezano jejich zobrazení u informací o znaku a ve slovníku (příští verze).
- Japonština má mnoho vlastních speciálních znaků, které běžně využívají a na české klávesnici se nevyskytují, proto bude v editoru tlačítko na jejich vložení. Seznam symbolů si bude moci uživatel rozšířit (tato verze).
- Velice užitečná funkce z WaKanu – možnost zobrazování čtení nad znakem (příští verze).
- Zahrnutí Rikaichan a Rikaikun se může zdát na první pohled nesmyslné. LUKA sice nebude modul do prohlížeče, nicméně pokud si uživatel otevře dokument, který mu poslal například jeho vyučující, může najetím na neznámý znak zjistit jeho čtení a překlad. Tím se vyhne otevření pomocného okna, které zobrazuje více informací než v danou chvíli potřebuje (příští verze).

3.2 Výběr nefunkčních požadavků

Do nefunkčních požadavků byly zahrnuty věci, jež v testovaných nástrojích chyběly uživatelům nebo jejich nepřítomnost způsobovala nějaké problémy.

Na prvním místě stojí podpora českých znaků v editoru. Bude tak možné psát v japonštině a k tomu si česky s diakritikou psát třeba poznámky. Čeština v uživatelském rozhraní by měla být pro český program samozřejmostí.

Dále také podpora zobrazování, zapisování a práce s párovými znaky 2.5.

S pomalu klesajícím podílem na trhu [6] není perspektivní dělat program pouze pro Windows. Na seznam požadavků tedy byla přidána nezávislost na operačním systému.

Uživatelé se dnes pohybují na více než jednom počítači a ne všude bude LUKA k dispozici, z toho důvodu není vhodné používat pro soubory vlastní (uzavřený) formát, ale naopak využít nějaký již existující.

3.3 Případy užití

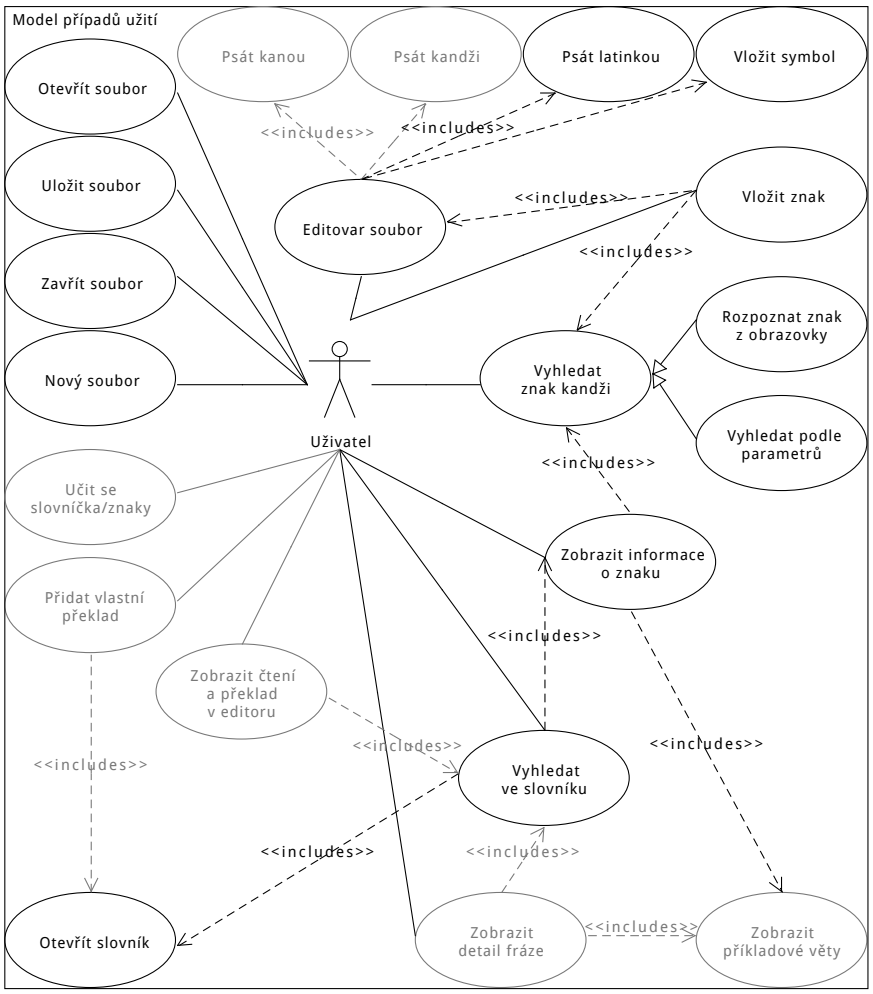
Výběr funkčních požadavků pro implementaci závisel hlavně na tom, bude-li možné funkcionalitu v budoucnu bez potíží a velkých zásahů do existujícího kódu přidat. Seznam implementovaných funkčních požadavků:

1. editor (formátování textu, otevírání, ukládání souborů apod.),
2. vkládání speciálních symbolů,
3. vyhledání znaku,
4. zobrazení informací o znaku,
5. slovník,
6. rozpoznání znaku na obrazovce,
7. transkripce latinky ve vyhledávacích polích.

Seznam funkční požadavků, jež zůstávají ve stavu návrhu:

1. IME v editoru,
2. zobrazování čtení a významu u znaku v editoru,
3. učení znaků a slovníček,
4. příkladové věty ve slovníku a u zobrazení informací o znaku,
5. vlastní překlady ve slovníku.

Model případů užití na Obrázku 3.1 reflektuje výše zmíněné seznamy a přidává mezi ně další funkční vztahy.



Obrázek 3.1: Model případů užití, šedou barvou jsou označeny části, jež nebudou implementovány.

3.4 Implementační jazyk

S ohledem na nefunkční požadavek 3.2 o nezávislosti na platformě a zhodnocení mých znalostí programovacích jazyků jsem vybrala programovací jazyk Java. Syntakticky se podobá jazyku C++. Ačkoliv by i aplikace v jazyce C++ mohla být multiplatformní, kvůli práci s japonskými znaky by vznikaly složité situace s kódováním. Java používá UTF16, čili jediný problém spočívá v párových znacích zmíněných v podkapitole 2.5.

3.5 Data

Téměř všechny známé aplikace, nejen ty testované, čerpají ze stejných zdrojů. Důvodem budiž fakt, že ony zdroje jsou kvalitní a léty prověřené.

Největší zásluhu na souboru těchto dat má profesor James Breen. Již v roce 1991 sestavil japonsko-anglický slovník v jednoduchém formátu EDICT, který se stal stavebním kamenem pro JMDict – Japanese-Multilingual Dictionary (japonsko-vícejazyčný slovník) ve formátu XML. V dnešní době slovníky spravuje EDRGD (Electronic Dictionary Research and Development Group)[3] Monashovy univerzity v Austrálii, jejímž členem je i James Breen.

EDRGD pracuje i na dalších projektech, jež jsou (nebo později budou) v LUKA editoru používány:

- KAJIDIC a KANJIDIC2 – soubory s informacemi o kandži znacích. Například čtení, počet tahů, mnohá kódování, odkazy na známé slovníky, zařazení JLPT a další. Vyhledávání znaků v LUKA čerpá hlavně z KANJIDIC2.[4]

- RADKFILE a KRADFILE – seznamy znaků a jejich rozložení na vizuální složky. Ačkoliv se tyto soubory používají v aplikacích k „vyhledávání podle radikálů“, klasické radikály 1.2.1 to nejsou.[5]

- ENAMDICT a JMnedict – databáze japonských jmen (křestních, příjmení, společností, míst, stanic atd).

Pro příkladové věty poslouží volně dostupný sourhn japonských vět a jejich vícejazyčných překladů z projektu Tatoeba[6]. Tento projekt vychází z korpusu profesora Tanaky obsahujícího pouze japonsko-anglické věty. Když se korpus v roce 2002 dostal do rukou Jamese Breena, byl zbaven spousty chyb a převeden do strojově čitelné formy. Od roku 2006 tento souhrn spravuje Tatoeba Project [7].

[3] `http://www.edrdg.org/`
[4] `http://www.csse.monash.edu.au/~jwb/kanjidic2/index.html`
[5] `http://www.csse.monash.edu.au/~jwb/kradinf.html`
[6] `http://tatoeba.org/`

3.6 Umístění a formát souborů

Celá aplikace se všemi pomocnými a zdrojovými soubory bude obsažena v jediné složce. Výhoda tohoto řešení spočívá ve snadné odinstalaci a přenositelnosti již používaného programu (zachování nastavení). S běžnou praxí, kdy se do domovského adresáře uživatele vloží konfigurační soubory, jež mnohdy zůstávají i po odinstalaci, zásadně nesouhlasím. Dále si uživatel může zjistit, kolik místa na disku zabírá a co všechno obsahuje.

Zdrojové soubory (RADKFILE a KANJIDIC2) se sloučí, aby se maximalizoval výkon a neprohledával se každý soubor zvlášť. Do databáze KANJIDIC2 se ke každému znaku, který je obsažen v RADKFILE přidá seznam jeho vizuálních složek.

Na formát výstupních souborů editoru jsou kladeny vysoké nároky, a proto nebylo snadné nějaký vybrat. Požadavky – otevřený, s bohatým formátováním a podporovaný širokou plejádou ostatních kancelářských editorů splňují dva, se kterými umí Java pracovat bez knihoven třetích stran. Jsou jimi RTF a HTML, ze kterých bude LUKA podporovat HTML.

3.7 Frameworky

LUKA nevyžaduje speciální grafické uživatelské rozhraní, z toho důvodu by plně postačila knihovna Swing obsažená v Javě. Ovšem Swing je dnes zastaralý a pomalu jej nahrazuje JavaFX, jež je součástí Java JDK / JRE od verze 8. Bude potřeba se vypořádat s tím, že některé komponenty budou ještě nějakou dobu chybět. Například balíčky a třídy pro dialogy mají být oficiálně vydány až v březnu 2015 v JavaFX 8u40 [8].

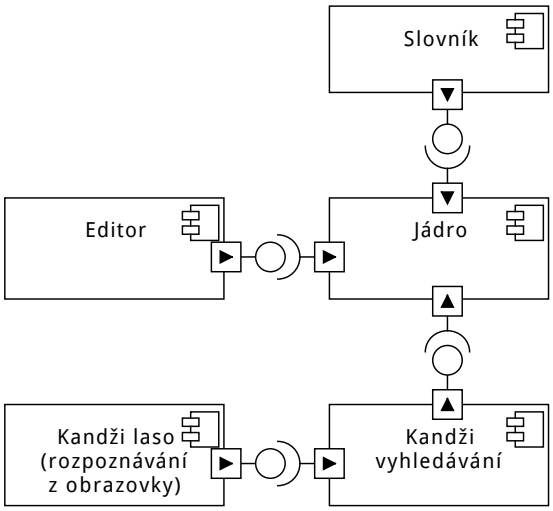
Dále pak parsování XML souborů vede dvěma hlavními cestami – SAX a DOM. Zatímco DOM načte celý dokument do paměti, SAX jej prochází po řádcích. Vzhledem k vlastnímu modelu uchování dat (jiné uspořádání, ne všechny informace zůstávají v paměti apod.) jsem se vydala cestou SAXu, respektive jeho alternativou StAX.

3.8 Nezávislé komponenty

Vzhledem ke složitosti programu je vhodné jej nějakým způsobem rozdělit na komponenty, které pracují nezávisle na sobě. Samozřejmě provázanost jednotlivých funkcí (například vkládání znaku ze slovníku i z kandži vyhledávání do editoru) klade na takové rozdělení jistá omezení.

Mé řešení spočívá v rozdělení na čtyři zcela nezávislé komponenty a jádro, jež je spojuje a přes které probíhá veškerá komunikace. Jsou to:

- Slovník
- Rozpoznávání



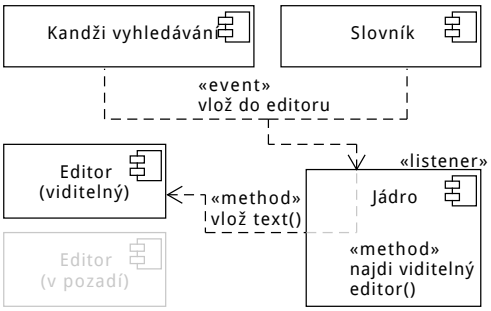
Obrázek 3.2: Propojení nezávislých komponent.

- Vyhledání znaku
- Editor

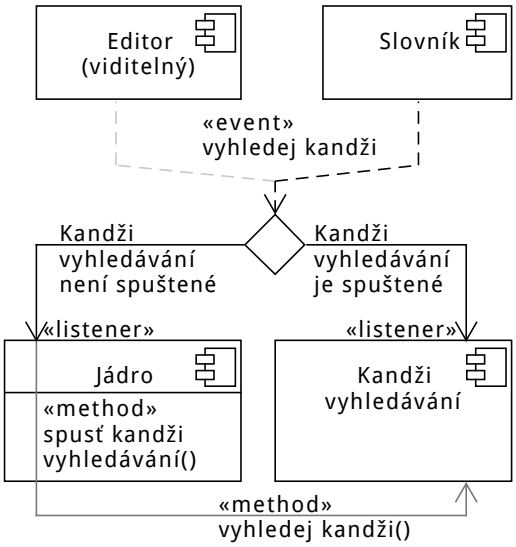
V následujícím textu se komponentám z hlediska návrhu, realizace a testování věnuji jednotlivě. Avšak nejde začít pracovat na komponentě bez předem daného rozhraní, jež by komunikovalo s jádrem (popř. s ostatními komponentami). Hrubé propojení komponent ilustruje obrázek 3.2. Je zde například vidět, že Kandži Laso rozšiřuje Kandži vyhledávání, tudíž závisí pouze na něm a komunikace tudíž neprobíhá ani s jádrem.

Komunikace ostatních komponent s editorem 3.3 ve směru vkládání textu bude probíhat přes jádro, které bude naslouchat těmto událostem. Jádro poté vybere aktuální (uživateli zobrazený) editor a požadovaný text do něho vloží pomocí funkce (už ne události). Vkládání textu tak mohou budoucí komponenty provádět jen na základě vystřelení události, netřeba žádných změň v kódu.

Vyhledání kandži a zobrazení informací o něm odkudkoliv z programu 3.4, zajišťuje komponenta Kandži vyhledávání, která opět naslouchá dané události. Změna oproti předchozímu případu tkví v tom, že vyhledávání nemusí být spuštěné, čili této události naslouchá také jádro, které v případě potřeby Kandži vyhledávání spustí a předá mu informace. Šedá čára na diagramu z editoru ilustruje případ, kdy uživatel bude chtít vyhledat znak z editoru, avšak tato funkce nebude nyní implementována. Hlavním důvodem je fakt, že v JavěFX editor jako takový není, čili bude muset být provedena analýza ohledně



Obrázek 3.3: Komunikace komponent při vkládání textu do editoru.



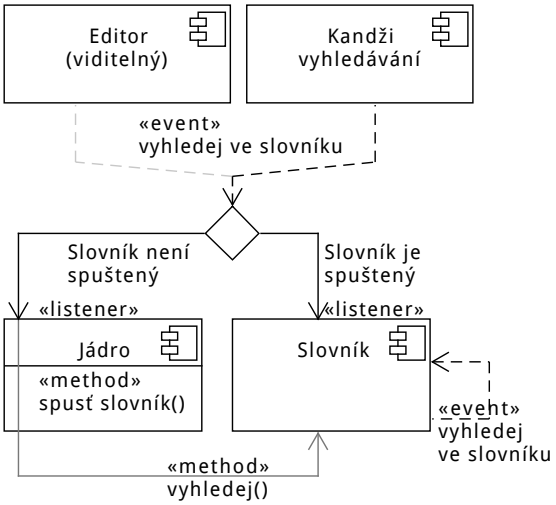
Obrázek 3.4: Komunikace komponent při vyhledávání znaku kandži.

vhodných kandidátů a způsob získání textu z takového editoru mi nyní není znám. Podrobněji v kapitole 6.

Totožným způsobem je řešeno vyhledávání ve slovníku 3.5.

3.9 Použitá architektura

V celé aplikaci se jednotně používá třívrstvá architektura MVC (model-view-controller), jen případě, kdy by byla třetí vrstva (model) zbytečná (žádná nebo



Obrázek 3.5: Komunikace komponent při vyhledávání ve slovníku.

velmi omezená interakce s diskem), zůstávají vrstvy dvě. K této architektuře patří i způsob komunikace – prezentační vrstva s logickou vrstvou komunikuje jen prostřednictvím událostí. Snižuje to provázanost mezi těmito vrstvami, což v budoucnu znamená snazší výměnu vrstev, ke které může dojít, jelikož LUKA je mým prvním programem, jež má grafické uživatelské rozhraní. Ze stejného důvodu kladu také velký důraz na co nejmenší provázanost mezi jednotlivými komponentami.

Kapitola 4

Slovník

Slovník představuje důležitou součást výuky (nejen) japonského jazyka. V dotazníku se velice často objevovalo použití pouze samotného slovníku a stížnosti na absenci příkladových vět v něm. Tato kapitola tedy řeší provedení komponenty Slovník, její návrh a budoucí zakomponování příkladových vět, realizaci, otestování (testování komunikace viz 8.3) a také detailnější způsob komunikace s ostatními komponentami.

Aby vyhledávací proces byl co nejrychlejší, slovníková data se načítaní při startu programu a zůstávají načtená. Dalším důvodem je nepříliš intuitivní formát zdrojových dat (viz níže), kdy by se musel celý záznam načíst do paměti, aby jeho data mohla být zobrazena uživateli. V momentě, kdy uživatel aktivně slovník využívá, veškeré načítané záznamy by se tak znovu načítaly a zase mazaly, což konkrétně v Javě není ideální řešení, neboť Garbage Collector se nevolá pokaždé, když na nějaký objekt už nejsou reference. Bude-li v budoucnu, či při uživatelském testování, problém s pamětí, lze nastavení upravit.

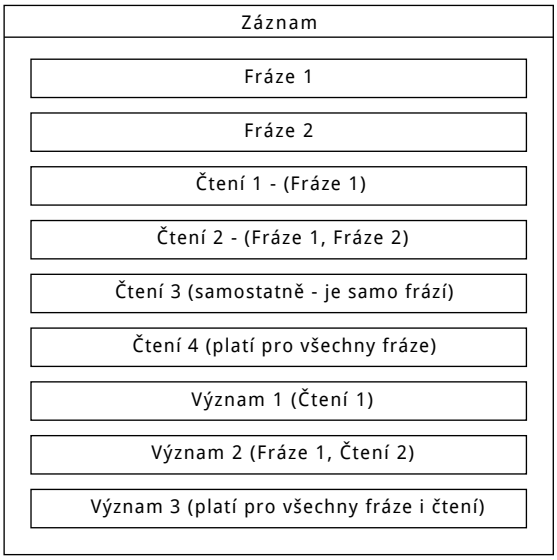
4.1 Analýza a návrh

Návrh slovníku sestával pouze z toho, jaký způsobem načítat data, jak je uchovávat a především v nich vyhledávat. Podoba grafického rozhraní vyplývá z poučení z nepříliš povedeného GUI pro Kandži vyhledávání.

4.1.1 Načítání souboru

Jako data pro slovník slouží JMDict od EDRGD, viz kapitolu 3.5. Zvolený formát tohoto XML souboru sice obsahuje všechny informace, ale vazby mezi nimi není snadné rozklíčovat. Pro jednoduchost uvažujme jen čtyři nejvýznamnější elementy:

1. záznam,



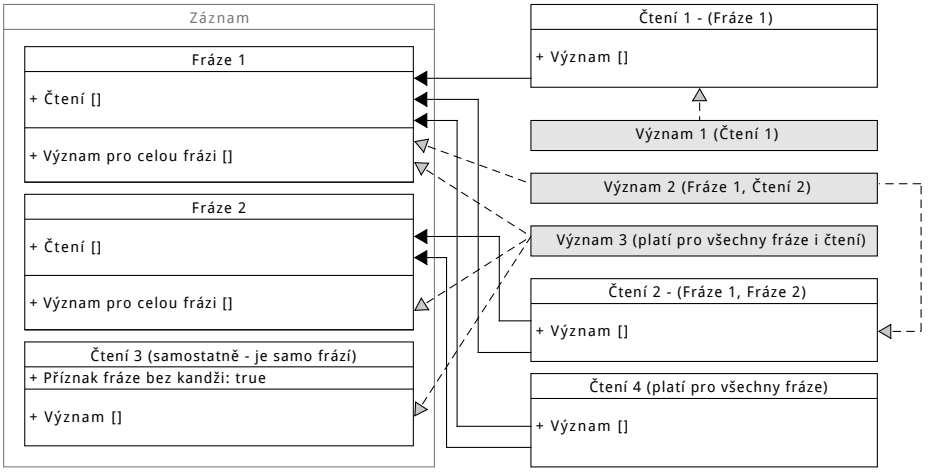
Obrázek 4.1: Ukázka formátu jednoho záznamu v XML souboru JMDict.

2. fráze (obsahující kandži),
3. čtení (psané kanou),
4. význam (soubor překladů a doplňujících informací o překladu)

Jejich uspořádání v souboru ukazuje obrázek 4.1, element záznam (entry) sdružuje všechny související informace. Každý takový záznam může obsahovat libovolné množství frází (k_ele), jedno a více čtení (r_ele) – pro každou frázi se čtení mohou různit, případně fráze mohou mít některá čtení společná. Například fráze あうんの呼吸 a fráze 阿吽の呼吸 se čtou stejně – あうんのこきゅう (aunno kokjú). Významy (sense) mohou být v záznamu také v libovolném množství (ano, i žádné) a mohou se vztahovat pro některá čtení, pro některé fráze anebo pro celý záznam.

Uložení dat v programu do nějakého použitelného modelu, jež by se dal prakticky používat (vyhledávat a vypisovat jednotlivé složky) a neobsahoval velkou spoustu redundantních položek jako zdrojový soubor (např. element význam obsahuje subelementy stagk resp. stagr, jež vymezují platnost tohoto elementu jen pro některé fráze, resp. čtení, s plným zněním dané fráze, resp. čtení), se stal nesnadným úkolem.

Nakonec zvítězil kompromis mezi požadavky a model, do kterého se převede formát z obrázku 4.1 můžete vidět na obrázku 4.2. Záznam obsahuje zvlášť jednotlivé fráze a speciální případy, kdy daná fráze neobsahuje žádný znak kandži a je tedy v XML označena jako čtení). Pro každé čtení a význam



Obrázek 4.2: Převedení formátu JMDict do objektového modelu. Pro přehlednost jsou šipky čtení černé a šipky významů šedé.

se tvoří vlastní třídy a fráze na ně mají jen reference, čímž se významně šetří paměť.

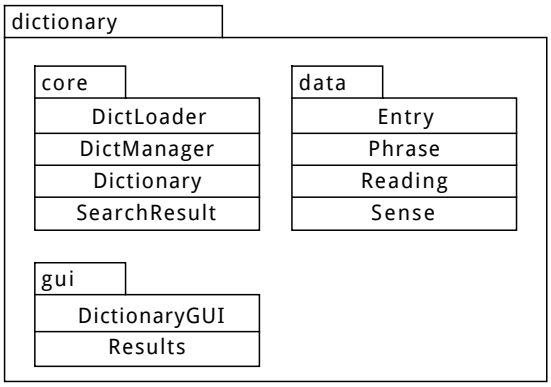
4.2 Vyhledávání

Vyhledávání probíhá ve dvou vlnách, aby slovník co nejlépe odpovídal na zadaný výraz, hledá v první vlně přesnou shodu (kromě bílých znaků a velkých / malých písmen). Ta se zobrazí jako první a seřadí se podle frekventovanosti. Druhá vlna hledá částečnou shodu, jež pro japonský vstup probíhá jednoduše – cokoliv obsahuje celý zadaný výraz vyhovuje. Musí to tak být kvůli speciálním prefixům – například slovo 仕事 znamená „práce“, přidání zdvořilostního prefixu お nezmění význam, ale pouze úroveň zdvořilosti: お仕事 – Vaše práce.[4]

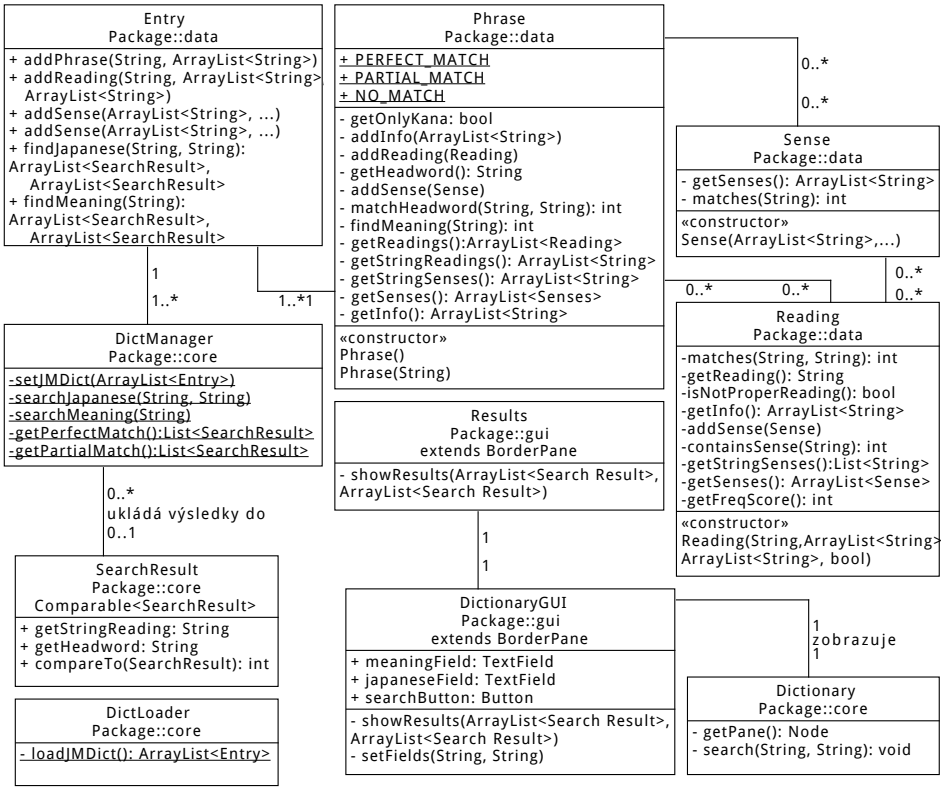
V překladu nelze hledat tímto způsobem, neboť třeba pro slovo „rain“ by vyhovovalo i slovo „train“, jež není relevantní. Hledají se tak jen začátky slov, pokud uživatel zadá slovo „rain“ vyhoví zadání překlad „heavy rain“ nebo i „rainfall“.

4.3 Realizace vyhledávání

Pro správné nalezení slov ve výpisu překladů se musí každý takový řetězec nějakým způsobem rozparsovat. Naštěstí Java v regulárních výrazech disponuje metaznakem `\b`, tzv. word boundary, sloužícímu jako *kotva* pro slova. Java má v tomto ohledu další přednost, jelikož umí kotvu vyhledat i ve všech



Obrázek 4.3: Návrh tří vrstev – balíků – Slovníku



Obrázek 4.4: Návrh architektury Slovníku

znacích UNICODE [9], nemusí se tedy provádět žádná změna, jakmile bude k dispozici slovník s českými překlady. Kotva hledá: a) před prvním znakem řetězce, b) po posledním znaku řetězce, c) mezi dvěma znaky v řetězci, kde jeden je znak slova (obecně jako metaznak `\w`) a druhý nikoliv. Vyhledává se v této podobě: `\bpřeklad`, přičemž pro slovo „rain“ regulárnímu výrazu již nevyhoví slovo „train“, ale například „rainy“ ano.

Skutečný kód pro vyhledávání ještě musí pamatovat na uživatelský vstup, takže se odstraňují přebytečné bílé znaky metodou `trim` z třídy `Utils` jádra a do regulárního výrazu se vyhledávaný text obalí proti nežádoucím znakům (které by mohly ovlivnit regulární výraz) metodou `quote`:

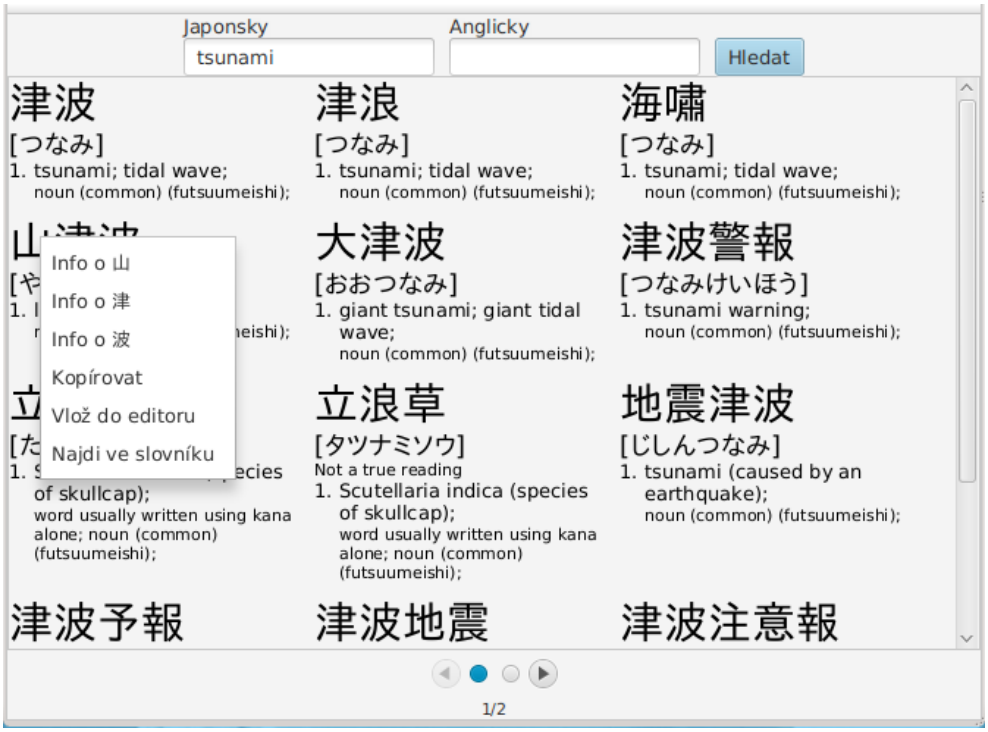
```
1    translation.matches(".*\\b" + Pattern.quote( meaning ) + ".*");
```

Jakmile proběhne vyhledání na úplnou i částečnou shodu, výsledky se řadí podle frekventovanosti. K tomuto účelu slouží třída `SearchResult`, jejíž návrh obsahuje více, než je nutné pro implementaci této verze programu – podpora k zobrazování příkladových vět. Zatím slouží jako kontejner, ze kterého se při vypisování výsledků sbírají potřebné informace. Hlavním atributem je zde čtení, jelikož právě čtení je jediným povinným elementem v záznamu. A právě podle frekventovanosti čtení se výsledky řadí – třída implementuje rozhraní *Comparable*. Podrobným průzkumem souboru JMDict jsem přišla na to, že je-li někde uvedena frekventovanost, její hodnoty jsou totožné u čtení i u fráze. Ovšem tyto hodnoty jsou textové a udávají příslušnost k různým seznamům – používané v novinách (první a druhá část), na webových stránkách, zvlášť také populární přejatá slova apod. Často čtení patří na víc takových seznamů, čili vyvstal problém, jakým způsobem výsledky řadit. Mé řešení dává těmto textovým hodnotám jisté číselné skóre, které je vyřešeno tak, aby se upřednostňovaly výsledky z prvních částí seznamů a aby čtení, potažmo fráze, často objevované v médiích měly vyšší hodnoty než ostatní. Zároveň jsou hodnoty skóre voleny tak, aby čtení, jež patří do všech druhých částí, „nepředběhlo“ čtení z první skupiny (kromě přejatých slov).

4.4 Prezentační vrstva

Uživatelské rozhraní slovníku respektuje klasické rozvržení – dvě vyhledávací pole (japonština a překlad) a tlačítko reagující na klávesu Enter, přičemž překladové pole může v dalších verzích sloužit pro angličtinu i češtinu současně. Modernější pojetí, kde vyhledávací pole je jen jedno, mi nepřijde jako ideální, jelikož by se před samotným vyhledáváním muselo zjistit, jaký jazyk uživatel vlastně zadal. A vzhledem k tomu, že japonský vstup lze zapsat v latince, takové rozhodnutí nemusí být vždy jednoznačné, například při zadání slova „rain“ může uživatel myslet déšť (anglicky) nebo linka (ライン – japonsky).

Výstup slovníku již respektuje modernější vzhled, a to dlaždice. Každý takový výsledek třídy `SearchResult` se při vypisování uživateli posílá do třídy



Obrázek 4.5: Konečný vzhled komponenty Slovník

`ResultTile`, která není v balíku slovníku, ale v balíku `controls` jádra, pro podrobnější popis viz podkapitola 8.2.4, jelikož se do budoucna předpokládá její další užití v plánovaných komponentách (jmenovitě například učící „kartičky“ – flashcards). Dlaždice výsledků se zobrazují po stránkách, které zajišťuje JavaFX komponenta `Pagination`. Finální vzhled slovníku lze vidět na obrázku 4.5.

4.5 Komunikace s ostatními komponentami

z různých částí aplikace lze výrazy překládat již teď, další jsou v plánu a jiné se teprvé jako nápady zrodí. Proto má tato komponenta ve třídě s událostmi svou událost `DICTIONARY_SEARCH`, které slovník naslouchá a jakmile ji nějaká komponenta požaduje, postačí, když dodá vyhledávané řetězce a o zbytek se již stará slovník. Pokud komponenta slovníku ještě není otevřená, řeší tuto událost jádro, které slovník spustí a předá mu vyhledávací parametry. Vyhledávací funkce slovníku se tedy nevolá přímo, čili bude možné v budoucnu její implementaci změnit, aniž by se jakkoli ovlivnily ostatní komponenty. Dále slovník (nepřímo) komunikuje s kandži vyhledáváním. Přes třídu `ResultTile` lze znaky z frází zobrazit v infopanelu kandži vyhledávání viz podkapitolu 5.3.1.

4.6 Testování

Testování funkčnosti probíhalo čtyřmi různými způsoby:

1. Testování konkrétních funkcí, jejich odolnosti vůči uživatelskému vstupu a správné načítání. Zde byla mj. odhalena záludná chyba, která se objevila při přidávání referencí třídy `Sense` ke čtení. Vzhledem k tomu, že fráze jsou oddělené, tak v momentě, kdy fráze sdílely jedno čtení, k němuž byl přiřazen význam, přidal se do třídy `Reading` pro každou frázi znovu. Přibylo tedy testování duplicit při přidávání významů.

2. Pomocí shellových skriptů puštěných na souboru se slovníkem se porovnávaly výsledky vyhledávání.

3. Kromě shellu jsem porovnávala ještě oproti Denshi Jisho, jež je prověřený uživateli, avšak jen orientačně, neboť vyhledávání v obou případech probíhá jiným způsobem.

4. Slovník testovali také uživatelé. Na funkční nedostatky již nenarazili, nicméně rozvržení výsledků nevyhovovalo všem. Někteří navrhli další vylepšení, která zahrnu do další verzí – jmenovitě změna velikosti písma a změna počtu zobrazených sloupců.

Kapitola 5

Vyhledání znaku

Vyhledání znaku – KanjiSearch – se nestará pouze o vyhledávání, ale spravuje také veškerou aktivitu Kandži Lasa. Také se odtud čerpají data pro kontextová menu KanjiTextu viz podkapitolu 8.2.2. Avšak primární zodpovědností zůstává vyhledávání, a to velice podrobné, prováděné uživatelem.

5.1 Analýza a návrh

Počet vyhledávacích kritérií (z informací, které dodává KANJIDIC2) byl stanoven na šest a lze je libovolně kombinovat:

1. čtení znaku – lze dle japonského a sinojaponského čtení a také dle čtení, jež se užívá ve jménech (nanori)
2. význam znaku
3. počet tahů (rozsahově i přesně)
4. vizuální složky (radikály)
5. úroveň zkoušky JLPT
6. náležitost do různých seznamů (znaky vyučované jednotlivě v 1. až 6. třídě, používané ve jménech a džójó)

Může se zdát, že tak podrobné vyhledávání je zbytečné, avšak pokud má uživatel neznámý znak 瞽 někde v knize a vidí u něho furiganu (čtení pro méně používané znaky) コ (ko), takových výsledků program najde zhruba 250. Na znaku si lze všimnout jeho složitosti, čili by vyhledání šlo omezit počtem tahů, případně dobře rozeznatelným radikálem v dolní části 目 – počet výsledků se snížil na 7.

Dále například při učení na určitou úroveň zkoušky JLPT si student nechá vypsat požadované znaky a dostane všechny potřebné informace na jednom místě.

Prostor pro zobrazování informací ve výpisu výsledků je omezený. Proto zde musí být jen ty nejdůležitější – podoba samotného znaku, jeho čtení a významy. Veškeré další charakteristiky patří do infopanelu, kde vyhledávací prostor náleží pouze jednomu znaku.

5.1.1 Způsob vyhledávání

Zdrojová data program načítá při startu programu z XML souborů KANJIDIC2 a KRADFILE, přičemž první obsahuje informace o znacích, druhý zase jejich radikály. Aby se načítání urychlilo, soubory se sloučily v jeden (k elementu každého znaku přibyla položka s jeho radikály).

Data zůstávají načtená po celou dobu běhu programu, protože informace o znacích hojně využívají ostatní komponenty – pokaždé tedy hledat v souboru na disku, by aplikaci zpomalilo.

5.1.2 Návrh uživatelského rozhraní

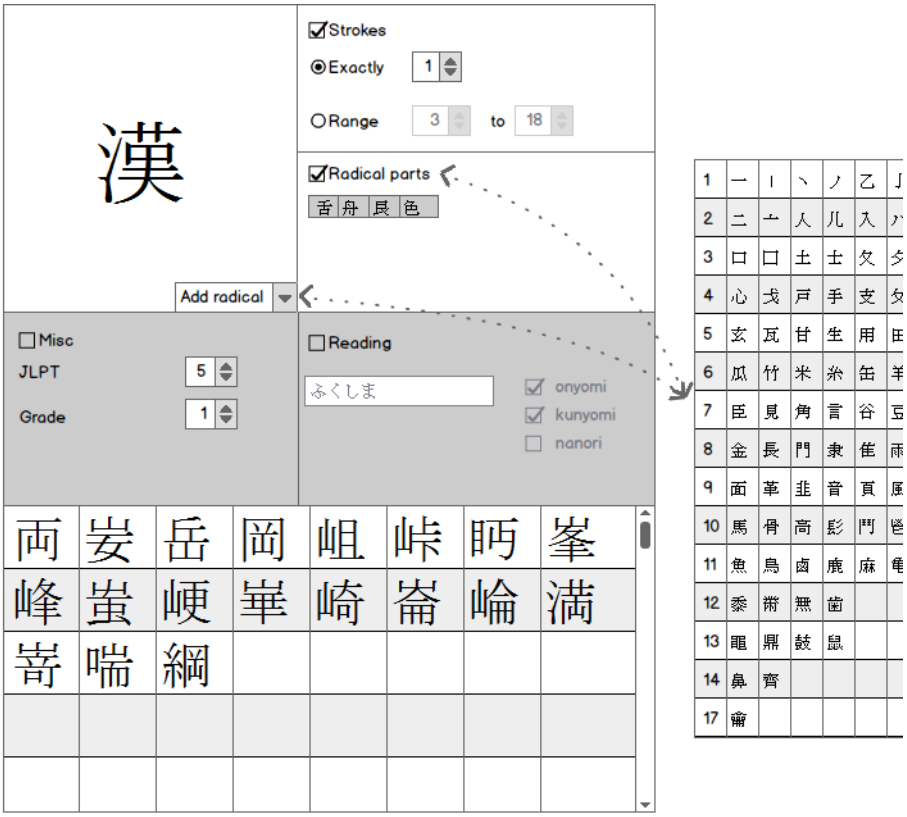
Návrh uživatelského rozhraní pro tuto komponentu proběhl ještě před změnou vyhledání pomocí výpočetní inteligence. Čili na Obrázku 5.1 zůstala část pro nakreslení znaku.

Speciálním případem vyhledávání je kritérium radikálů. Uživatelé jsou z Denshi Jisho zvyklí, že hledání probíhá okamžitě. Naneštěstí seznam všech radikálů zabírá v uživatelském rozhraní velkou plochu, proto pokud chce uživatel tímto způsobem vyhledávat, ostatní vyhledávací kritéria překryje tento seznam. Kvůli zachování možnosti kritéria kombinovat, zůstanou tyto hodnoty nastavené.

5.1.3 Návrh modelu

Navržení komponenty s sebou neneslo žádná těžká rozhodnutí, návrh na Obrázku 5.2 tak přímočaře vyplývá z definovaných požadavků. Opět zde figurují jen dvě vrsty z architektury MVC, jelikož se pouze jednorázově načte soubor s informacemi a více se na disk nesahá. Specifikem zde může být kontejner pro Kandži (viz níže) a abstraktní třída `SearchOption`, jež zastřešuje třídy všech vyhledávacích kritérií, obsahuje společné atributy (`isSelected`), implementuje metody (např. zneaktivnění GUI prvků nevybraných kritérií) a předepisuje metodu `isFilled`, kterou musí všechny zděděné třídy překrýt – z toho důvodu, že uživatel může ponechat kritérium aktivní, ale nevyplní žádné údaje, čili by se nemělo dle čeho hledat.

Třída `KanjiLoader` slouží pouze k načtení znaků při startu aplikace, její další správu již řeší třída `KanjiManager`, která je nezávislá na aktivitě či neaktivitě Kandži vyhledávání z důvodů řečených výše – ostatní komponenty její vyhledávací metodu taktéž používají.



Obrázek 5.1: Návrh uživatelského rozhraní pro Kandži vyhledávání s částí seznamu radikálů, jež vyhledávací kritéria v případě výběru radikálů úplně překrývá.

5.1.4 Tok událostí

Komponenta naslouchá události `SHOW_KANJI_INFO`, která pro daný znak zobrazí infopanel v místě, kde se objevují výsledky. Tuto událost také vystřeluje třída `KanjiText` viz podkapitola 8.2.2, kterou komponenta používá pro zobrazení znaků a jeho informací v části s výpisem výsledků. Mezi seznamem výsledků a infopanelem uživatel přepíná pomocí tlačítka „Zpět k výsledkům“. Pokud infopanel vyvolala externí komponenta, stisknutím tlačítka se zobrazí výchozí panel Kandži vyhledávání s nápovědou, jak vyhledávat.

Další událostí, kterou vyhledávání zpracovává, je uživatelské řazení výsledků. Řízení zobrazování seznamu radikálů taktéž probíhá přes události – `SHOW_RADICALS` a `HIDE_RADICALS`.

I komunikace s Lasem je vedena přes události, Laso dokonce oznámí Vyhledávání, kdy má připravené výsledky k zobrazení, což se děje vystřelením `KANJI_LASSO_RESULTS_READY`. Jediné metody, jež tyto komponenty spojují slouží k získání seznamu kandži znaků – `getResults():ArrayList<String>`

a ukončení Lasa metodou `close`.

5.2 Realizace

Kandži vyhledávání bylo realizováno jako jedna z prvních komponent a naneštěstí se zde objevilo nejvíce nedostatků Javy – chybějící pole pro čísla, omezení expanze XML apod. Také se během přechodu z návrhu k implementaci z programu odstranilo rozpoznávání ručně napsaného znaku a bylo nahrazeno Kandži Lasem.

5.2.1 Načítání dat

O načtení souborů se stará statická třída `KanjiInfoLoader` a její metoda `loadKanjis`, jejímž výstupem je seznam speciálních kontejnerů na znaky – `Kanji`. Tato třída stejně jako slovník používá `XMLStreamReader` (StAX) a jeho paměťově nenáročné a rychlé čtení. Ovšem během načítání se vyskytly chyby a výjimky. Jak se ukázalo, knihovny StAX z bezpečnostních důvodů omezují počet expanzí pro externí soubory na 64000[7]. Což se možná zdá jako pochopitelné, nicméně pokud se tohle omezení vztahuje na všechny načítané dokumenty v celé aplikaci (`XMLInputFactory` implementuje vzor `Singleton`), nastává problém. Slovník i soubory s kandži informacemi totiž obsahují stovky tisíc řádků. Naštěstí lze tuto situaci řešit následujícím kódem:

```
1 inputFactory.setProperty(
2 "http://www.oracle.com/xml/jaxp/properties/entityExpansionLimit",
3 "0" );
```

který limit zruší.

5.2.2 Kandži kontejner

Třída `Kanji` slouží jako komplexní kontejner pro práci s jedním znakem. Obsahuje o něm veškeré informace – čtení, významy, tahy, radikály apod. A všechny tyto informace umí jiným třídám předávat a odpovídat na jejich dotazování. Obsahuje například metodu:

```
1 boolean containsNanori( String hiraganaReading,
2                         String katakanaReading )
```

jež zjišťuje, jestli parametry vyhovují některému z nanori čtení. Vzhledem k uživatelskému používání divokých karet, které se od těch používaných v Javě liší, se parametry napřed musí upravit (nahradí se * za .* a otazník za tečku)[8]:

```
1  hiraganaReading.replaceAll( "\\*", "\\.\\*" )
2                 .replaceAll( "\\?", "\\." )
```

[7] `http://docs.oracle.com/javase/tutorial/jaxp/limits/using.html`

[8] Pro přehlednost uvádím kódy jen pro čtení v hiraganě.

Ale vzhledem k formátu čtení ve zdrojovém souboru, se musí provést ještě další úpravy:

```
1   "\\-?" + hiraganaReplaced + "(\\..*)?\\-?"
```

Tento regulární výraz před zadaným čtením vyfiltruje prefixy a sufixy (oddělené pomlčkou) a okuriganu (oddělenou tečkou) a vstupuje do samotné funkce `matches`:

```
1   return nanoriReadings.stream().anyMatch(  ( nanori ) ->
2       ( nanori.matches( "\\-?" + hiraganaReg + "(\\..*)?\\-?" );
```

Dále kontejner obsahuje podobné metody pro parsování významu a pro ostatní druhy informací o znaku. Kromě dotazovacích metod zde nechybí formátovaný výstup (například `getRowFormattedMeanings` pro zobrazení významů oddělených čárkou) a komparační metody. Několik lidí totiž vyslovilo přání, aby bylo možné znaky řadit i podle jiného kritéria než jsou tahy. Třída implementuje rozhraní *Comparable* a kromě požadované metody `compareTo` obsahuje ještě další, které se používají při řazení výsledků ve výpisu – podle počtu tahů, frekventovanosti, zkoušky JLPT a kombinovaný (výchozí), který nejprve řadí podle frekventovanosti a pokud znak nepatří do často používaných, řadí se dle tahů.

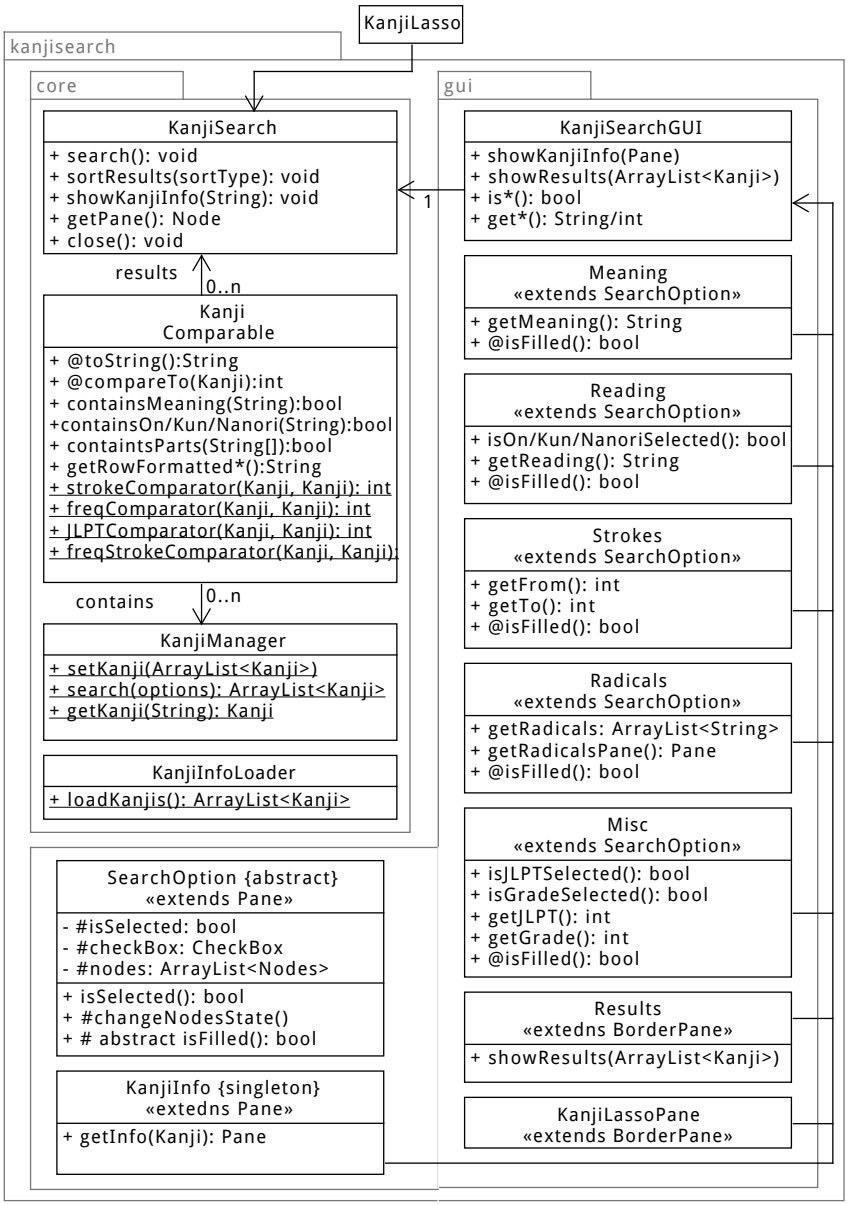
5.2.3 Vyhledávání

V momentě, kdy uživatel klikne na vyhledávací tlačítko (popř. na tlačítko s radikálem), spouští se metoda `search`. Z uživatelského rozhraní dostane vstup pomocí metod `isMeaning`, `getMeaning` apod. (v návrhu jsou tyto metody označeny jako `is*() a get*()`, kvůli přehlednosti modelu). Pokud bylo označeno čtení, přeloží uživatelský vstup pomocí konverzních tříd viz podkapitolu 8.1.3 do hiragany a katakany. Poté tato data předává třídě `KanjiManager`, jež obsahuje kontejnery se znaky a kde se provádí samotné vyhledávání pomocí metod v `Kanji`. Výsledky se seřadí pomocí implicitního komparátoru a odešlou uživatelskému rozhraní k zobrazení.

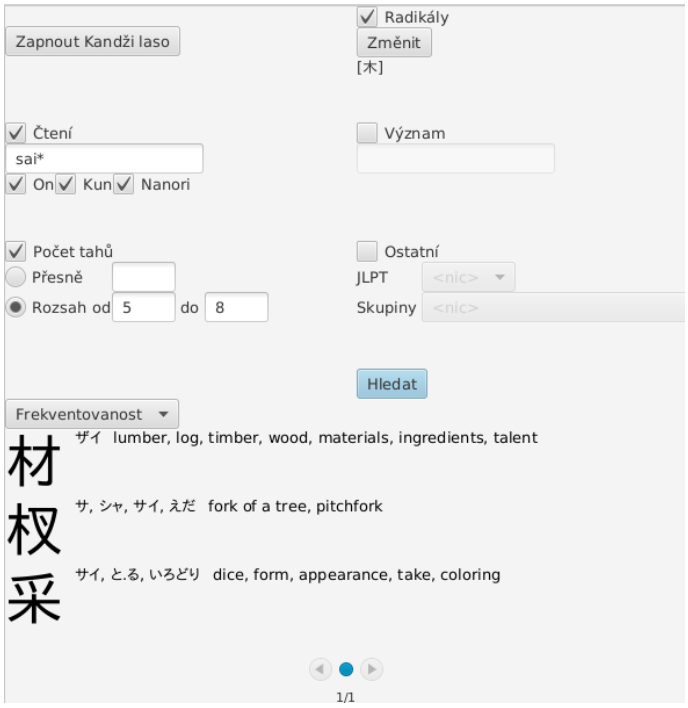
5.3 Prezentační vrstva

Implementace prezentační vrstvy (Obrázek 5.3) se liší od návrhu přidáním tlačítka pro Laso a uspořádáním výsledků. Původně se mělo čtení a význam znaku zobrazit při najetí myši na znak, což se ukázalo jako zdlouhavé řešení. Nyní jsou znaky pod sebou a základní informace mají vedle sebe v řádku. Další malou změnou oproti návrhu je výběr JLPT a skupin pomocí seznamu, nikoliv čísla.

V JavěFX v době implementace nebylo k dispozici vstupní pole, jenž by přijímalo jen číselné hodnoty. Čili bylo nutné kvůli počtu tahů takové pole vytvořit. Ale protože v pozdějších verzích bude pole potřeba nejen v této



Obrázek 5.2: Návrh architektury, včetně balíků, Kandži vyhledávání.



Obrázek 5.3: Výsledné uspořádání uživatelského rozhraní pro Kandži vyhledávání.

komponentě, bylo umístěno do balíku `controls`, podrobnější popis viz podkapitolu 8.2.3.

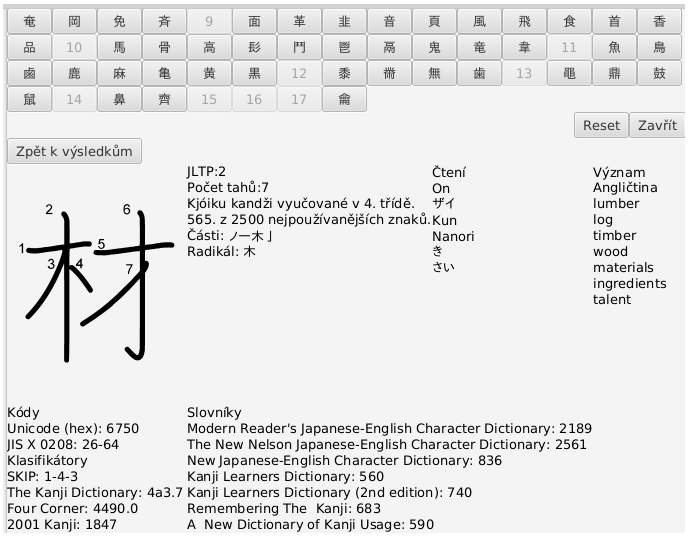
V této komponentě se nejvíce projevila absence designu (vzhledu). Hlavně vyhledávací kritéria nevypadají na pohled hezky. Design ale nebyl od začátku prioritou, tou je v první verzi funkčnost.

5.3.1 Kandži Infopanel

Příklad na Obrázku 5.4 ukazuje výpis všech dostupných informací o znaku. Zobrazuje se ve spodní části Kandži vyhledávání a překrývá nalezené znaky. Tento panel lze vyvolat odkudkoliv z programu – `KanjiSearch` (popř. jádro) naslouchá události `SHOW_KANJI_INFO`. Infopanel implementuje vzor singleton, jelikož se vždy zobrazuje maximálně jeden, čímž se recyklují použité zdroje.

Protože klasické rozepsání posloupnosti tahů vyžaduje (zvláště u složitých znaků) spoustu místa, způsob zápisu se zobrazuje v samotném znaku pomocí fontu Kanji Stroke Order[9], kdy je číslo pořadí tahu tam, kde by měl tah začínat.

[9]Jehož autory jsou Ulrich Apel, AAAA project a Wadoku project.



Obrázek 5.4: Infopanel pro kandži a část vyhledávací tabulky pro radikály.

5.4 Testování

Testování probíhalo stejně jako u slovníku – pomocí shellových skriptů puštěných na sloučené soubory KANJIDIC2 a RADKFILE, oproti Denshi Jisho (zde bylo mj. odhaleno, že Denshi neumí vyhledávat podle nanori čtení) a uživatelské testování.

Funkční chyba byla zjištěna jedna – opomenutá implementace metody resetující výběr radikálů, jež vyvolávala výjimku `UnsupportedOperation`.

Některým uživatelům opět nevyhovovalo uspořádání výsledků. V příští verzi se tedy výsledky budou zobrazovat stejně jako u slovníku – dlaždicí `ResultTile`. Dále se nepodařilo umístění scrollovacích částí – hýbe se celá komponenta, nikoliv jen výsledky. Navíc při nízkém rozlišení zabírají kritéria příliš velkou plochu. Příští verze se dočká ještě přepracování této části.

Zkušenosti z těchto nedostatků již využívá slovník, jež byl implementován jako poslední komponenta. Každá stránka výsledků má vlastní scrollovací prostor a přecházení mezi stránkami je výrazně pohodlnější. Jen zde ještě chybí uživatelská úprava písma.

Odkazy na slovníky v infopanelu, které byly do návrhu odsouhlaseny jako relevantní, se ukázaly zbytečné. Jejich místo tedy zaujme větší množství příkladových vět.

Kapitola 6

Editor

Kapitola o editoru řeší především vymezení požadavků na editor, jeho výběr a poté integraci do LUKA. Vzhledem k tomu, že vybraný editor jako takový už je hotová komponenta, tato kapitola je výrazně kratší než ostatní.

6.1 Návrh a požadavky na editor

Zodpovědnost editoru sestává především z editace a formátování textu. Otázkou bylo, má-li se editor starat i o ukládání souborů a komunikovat s třídou, jež se o diskové operace stará. Avšak z hlediska návrhu by se jednalo o špatné rozhodnutí, proto se editor stará pouze o editaci textu a služby ohledně ukládání převzala jiná třída.

Požadavky na takový editor nejsou malé, již na počátku jsem vytvořila seznam funkcí, jež musí editor zvládat a podle toho také proběhl výběr. V implementaci tak pro editor existuje rozhraní *Editable*, jež musí být implementováno pro správnou funkčnost především Kandži vyhledávání a slovníku. Obsahuje:

- vložení textu na aktuální pozici kurzoru,
- vložení formátovaného (HTML) a neformátovaného textu,
- získání označeného textu,
- získání veškerého textu,
- nastavení textu a
- smazání veškerého textu,
- získání informace o změně textu,
- změna stavu obsahu textu (změněný / nezměněný).

Dále musí umět pracovat s japonštinou a mít možnost vložit speciální japonské typografické značky a jiné symboly, včetně těch uživatelsky definovaných.

6.2 Výběr editoru

Implementace LUKA s použitím mladých knihoven JavaFX s sebou přináší mimo jiné i problém toho, že chybí některé komponenty. Kromě pole, jenž přijímá jen čísla (podrobně v podkapitole 8.2.3) a dialogových oken také neexistuje dostatečně univerzální formátovací knihovna, nad kterou by se dal vystavět plnohodnotný WYSIWYG[10] editor.

Ačkoliv existuje komponenta `HTMLEditor`[10], která umožňuje text formátovat různými způsoby, pro potřeby LUKA se nehodí zejména kvůli své uzavřenosti. API HTMLEditoru totiž obsahuje funkce pouze na nastavení a získání obsahu. Přidání textu na pozici kurzoru, získání označeného textu nebo jen zjištění, zda-li se obsah editoru změnil tak není možné nijak získat ani ovlivnit. Mimo to se při testování objevila spousta chyb, převážně ve funkcích ovlivňujících velikost fontu.

Doporučovanou alternativou k oficiálnímu HTMLEditoru se stal RichTextFX[11] využívající `TextFlow`. Kromě absence náročnějšího formátování, například seznamů, tento editor nepodporuje externí IME a jejich podporu autor ani do budoucna neplánuje[11]. Určitě by bylo vhodné, aby si uživatel metodu vstupu mohl zvolit sám a nebyl nucen do něčeho, na co není zvyklý. Navíc vzhledem k tomu, že tato (první) verze LUKA editoru nedisponuje vlastním IME, nešlo by v editoru rozumně japonštinu zapisovat.

Poslední možností, jak dosáhnout nejen pokročilého formátování textu, ale také možnosti editace z ostatních komponent, je vložení webového editoru. Nejlepšími vlastnostmi disponuje CKEditor. Kromě exportování do otevřeného a na platformě nezávislého HTML, má mnoho dalších užitečných funkcí:

- Absolutní kontrola nad vkládaným obsahem – ačkoliv lze do schránky zkopírovat dokument z webu například i s obrázky, CKEditor vkládaný obsah vyfiltruje na základě povolených doplňků.
- Na domovské stránce je možné vybrat si seznam použitých doplňků a tím jednak přizpůsobit editor podle potřeb LUKA a zároveň tak odpadá ruční odebírání v nějakém konfiguračním souboru, což značně usnadňuje budoucí aktualizace.
- Editor se dodává v mnoha vzhledech.

[10] What You See Is What You Get – editor, ve kterém se formátování provádí označením textu a kliknutím na příslušné formátovací tlačítko, výsledek je vidět ihned.

[11] `https://github.com/TomasMikula/RichTextFX`

- Na stránkách lze nalézt velmi přehlednou dokumentaci[12] nejen s API, ale také s příklady užití. Zároveň se lze podívat přímo do kódu na to, jak je která funkce implementovaná bez nutnosti cokoliv stahovat.

- Rozhraní má český překlad.

- Asistované vkládání z MS Wordu – spousta japonštinářů MS Word používá a přímé vložení z tohoto programu ne vždy dopadne podle očekávání. CKEditor tedy obsahuje tlačítko s dialogem, do kterého lze text vložit a ten je bezpečně převeden do HTML.

- Lze přímo editovat HTML kód a zároveň se CKEditor snaží takové úpravy hlídat, aby byly v souladu s normou HTML.

6.3 Propojení CKEditoru a Javy

Webový CKEditor nelze jen tak vložit do programu v Javě, musí se načítat za použití tříd `WebEngine` a `WebView`. Ovšem LUKA slibuje, že ke svému fungování uživatel nepotřebuje připojení k internetu. Takové omezení je nutné respektovat, čili se s editorem pracuje lokálně. Ve složce s editorem najdeme soubor index.html, ve kterém je mj.:

```
1         <script src="ckeditor.js"></script>
2 </head>
3 <body>
4         <textarea id="editor1" name="editor1">
5         </textarea>
```

Což představuje to nejnutnější, co musí soubor obsahovat. Veškerá další interakce se provádí z programu.

Nejprve ale musíme počkat, než se stránka načte, proto musí být na `WebEngine` připojen `ChangeListener`. V momentě, kdy se náš html soubor načte, začíná komunikace v jazyce JavaScript s CKEditorem. `WebEngine` disponuje funkcí `executeScript`, která jako parametr přijímá `String` s kódem v JavaScriptu. Iniciální kód vypadá takto:

```
1  CKEDITOR.replace( 'editor1', {
2    language: 'cs', height: 500, removeButtons:  'Maximize',
3      on : { 'instanceReady' :
4          function( evt ) {
5              evt.editor.execCommand( 'maximize' );
6          }
7      }
8  });
9  CKEDITOR.config.extraPlugins = 'autogrow';
```

[12] `http://docs.cksource.com/Main_Page`

Na prvním řádku se CKEditor vloží na textareu s id editor1, na druhém řádku se nastavuje jazyk rozhraní a odstraňuje se tlačítko na minimalizaci / maximalizaci velikosti editoru. Stránka neobsahuje nic jiného a my chceme, aby editor zabíral všechen prostor – zůstal maximalizovaný. To se nastavuje na řádku 5 v reakci na událost `instanceReady`, tedy že editor je připraven k práci. Řádek 9 přidává plugin `autogrow`, který se stará o to, aby si editor uměl dynamicky měnit velikost a reagoval tak na změnu velikosti kontejneru v Javě.

Dále se k již integrovanému seznamu symbolů připojují vybrané symboly z japonské typografie a také ze souboru `custom_symbols.txt`, do kterého si může uživatel vložit vlastní.

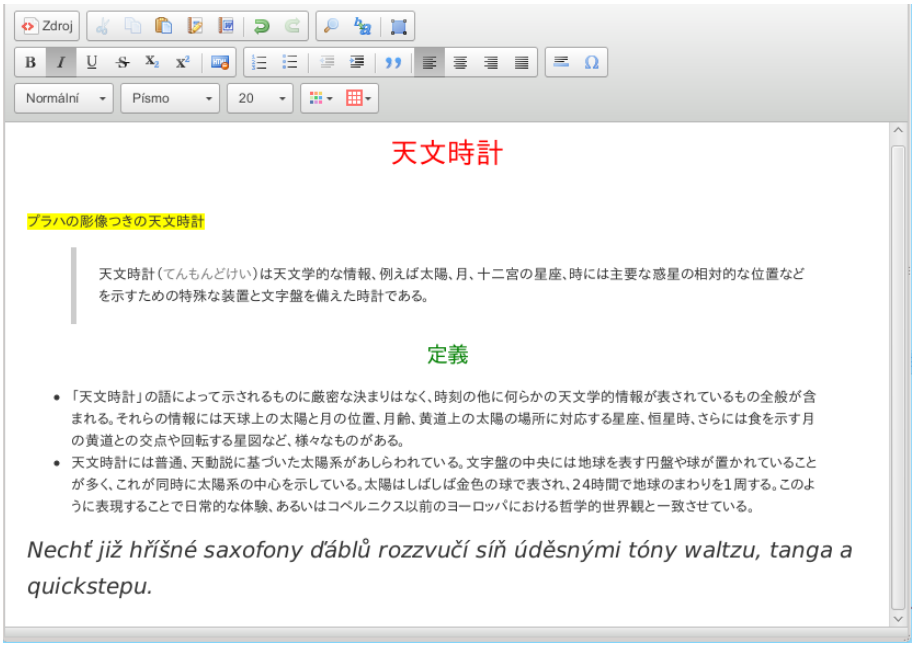
A konečně nastavuje přemosťující třídu (`JSBridge`), která nadále bude s editorem komunikovat:

```
1  javascriptBridge = ( JSObject ) engine.executeScript( "window" );
2  javascriptBridge.setMember( "java", new JSBridge() );
3
4  engine.executeScript(
5          "CKEDITOR.instances['editor1'].on("
6          + "  'change', function(evt){"
7          + "     java.documentChanged();"
8          + "  }"
9          + ");"
10 );
```

Přes funkci `documentChanged` se řídí stav obsahu editoru – jakmile uživatel obsah editoru upraví, tímto způsobem o nich dává CKEditor vědět.

6.4 Komunikace s jádrem

Editor samotný nekomunikuje s žádnou jinou komponentou, kromě jádra, protože v aplikaci se editorů používá několik. Tok dat z ostatních komponent filtruje ke správným editorům třída z jádra jménem `EditorsController`. Zde se také používají jen funkce z rozhraní *Editable*, případná výměna CKEditoru za jiný pak nepřinese žádné komplikace, dokud bude implementovat ono rozhraní.



Obrázek 6.1: Vzhled CKEditoru s upraveným textem o pražském Orloji z japonské Wikipedie a českým panagramem (text obsahující všechny znaky abecedy) k demonstraci funkčnosti české diakritiky spolu s Japonštinou.

Kapitola 7

Rozpoznání pomocí výpočetní inteligence

Tato kapitola se věnuje komponentě Kandži Laso – rozpoznávání znaků z obrazovky, jež tvoří rozšíření ke komponentě `KanjiSearch`. Nejprve se zde provádí teoretický rozbor problému, což zahrnuje generování datasetu, výběr rozpoznávací metody a také způsob předzpracování. Druhá část se zaměřuje na návrh a implementační detaily. Na konci se ještě věnuji způsobu napojení na `KanjiSearch`.

7.1 Rozpoznávací metoda

Ačkoliv původní plán cílil na vícevrstvé umělé neuronové sítě (ANN) a později také na k-nn s kd-stromy, konečné řešení leží v úplně jiné rovině. Nalezení dostatečného množství příznaků pro rozlišení ručně napsaných tisíců znaků potřebných k fungování sofistikovanější ANN (resp. k-nn) nebyl jednoduchý úkol. I přes to, že rozptyl hodnot následujících charakteristik byl vysoký, jejich rozlišovací schopnost nedostačuje, neboť se mezi nimi objevila korelace.

- Počítání pixelů ve čtyřech čtvercích.
- Počet přechodů černá-bílá – horizontálně i vertikálně, avšak rozdělené na poloviny, znaky totiž mívají horní resp. levou polovinu výrazně jinou než dolní resp. pravou.
- Počítání průměrné šířky a průměrné délky.
- Počítání uzlů apod.

K lepšímu strojovému rozpoznání znaků by tak bylo potřeba sáhnout po pokročilejších metodách extrakce příznaků jako jsou například Karhunen–Loève, obecná Houghova transformace či třeba SIFT (Scale-invariant fe-

ature transform). Žádná z nich ale nezaručuje rozpoznání ani těch nejpoužívanějších znaků s minimální chybou, jelikož ručně zapsaný znak může mít mnoho podob (mnohem více než latinka) a psaní počítačovou myší jej navíc výrazně zdeformuje.

Další nemilý fakt je ten, že se spousta znaků od sebe liší v drobných detailech. Pro ilustraci například tyto skupiny:

- 工三王玉干正
- 子丁于干千十下
- 血曲而皿自白
- 人入大犬太火天

Nemusí to být na první pohled patrné, ale v předešlém výčtu se žádný znak neopakuje, v takových případech se může splést i oko trénovaného člověka.

Kvůli těmto omezením a po poradě se studentkami japonštiny, jsme se rozhodli od ručně napsaného znaku upustit a vytvořit nástroj, jež by znaky rozpoznával přímo na obrazovce. Hodí se to zejména v případě, kdy od vyučujících dostanou naskenované materiály, či někde na webu narazí na neznámý znak v obrázku. Někteří se japonštinu učí zábavnou formou a čtou (případně tvoří amatérské překlady) online komiksy (mangy), kde se rozpoznání znaku taktéž hodí.

7.1.1 Dataset znaků

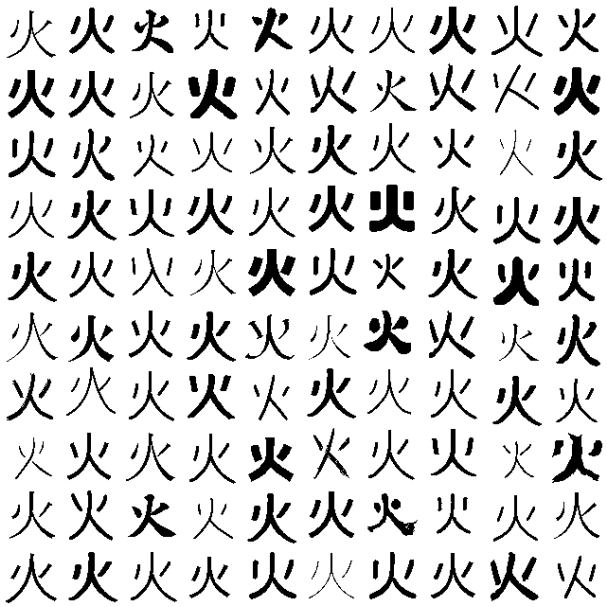
K rozpoznání jednoho znaku je potřeba o něm mít nějaké informace, které by jej charakterizovaly. Zároveň by tyto informace měly být dostatečně různorodé, aby se takový znak dal rozpoznat i lehce zdeformovaný.

Řešení jsem našla v použití volně dostupných fontů. Fonty jednak ve většině případů pokryjí všechny znaky, ale zároveň s sebou přinášejí i onu různorodost, jak ilustruje Obrázek 7.1. Navíc tvorba datasetu nezabere více než pár hodin i s předzpracováním.

Rozpoznání z obrazovky spoléhá na fakt, že znaky mají svou tištěnou formu. Naštěstí na rozdíl od latinky nemají japonské fonty příliš prostoru pro velkou stylizaci kvůli své komplikovanosti.

7.1.2 Generování a předzpracování datasetu

Jak lze vidět z Obrázku 7.1, písmena nejsou vždy na stejném místě a stejně velká. Takové informace ale pro jediný znak nejsou relevantní, čili je potřeba znaky znormalizovat jednoduchým předzpracováním.



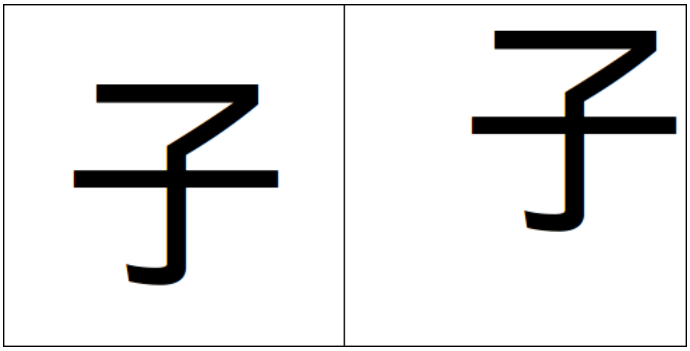
Obrázek 7.1: Různorodost v datasetu zajišťuje použití rozličných fontů.

Než se vytvoří soubor s obrázkem znaku ve velikosti 64×64 px, je ještě během generování předzpracován. Nicméně se může stát, že obrázek bude po předzpracování menší, než stanovená velikost obrázku v datasetu, proto se už v základu generuje velikost 256×256 px a případně se provede škálování.

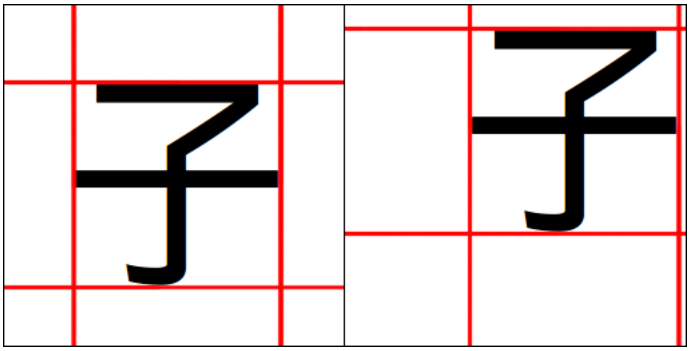
Samotný algoritmus generování pak vypadá následovně:

1. Načti příslušný font.
2. Zjisti, je-li daný znak fontem zobrazitelný (kostičky a otazníky v datasetu nechceme).
3. Pokud ano – vykresli jej do obrázku 256×256 px.
4. Pošli obrázek (7.2) na předzpracování:
   a) Najdi hranice znaku (7.3)
   b) Ořízni obrázek dle hranic znaku, vystřeď a škáluj jej na velikost 64×64 (7.4)
5. Pokračuj dalším znakem.

Výhoda tohoto postupu spočívá především v tom, že znaky, které ve fontu chybí procházejí dvojí kontrolou – přímým dotazováním na font, je-li schopen znak zobrazit a při hledání hranic znaku (prázdný obrázek se neukládá). V datasetu se pak snižuje počet chybných položek. Další kontroly obstaraly



Obrázek 7.2: Dva různá obrázky, jež vstupují do předzpracování.



Obrázek 7.3: Nalezení hranic znaků

Obrázek 7.4: Ořezání obrázků podle hranic a škálování na velikost 64×64, vzniká tedy totožný obrázek, který již vstupuje do učícího procesu.

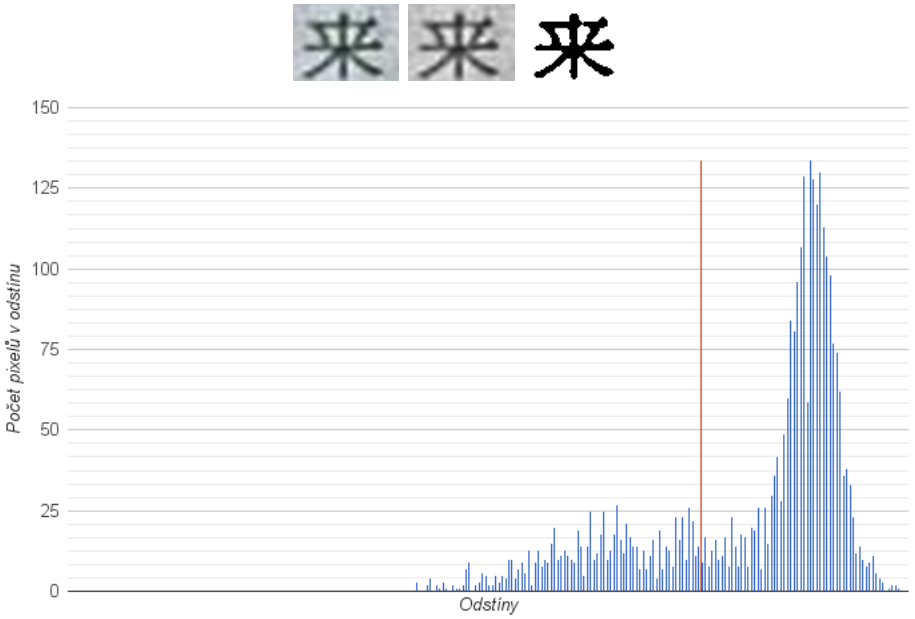
shellové skripty – smazání příliš malých souborů a vytipování „podezřelých“ obrázků na základě duplicit a ruční procházení příliš zdobných fontů.

Kdyby se některý z fontů zdál nevyhovující, není problém jej z datasetu odstranit, jelikož pro každý font byla vytvořena zvláštní složka. Opačným způsobem lze jednoduše další font do datasetu přidat.

I přes intenzivní a možná trochu agresivní mazání, dataset obsahuje necelé dva miliony položek.

7.1.3 Předzpracování

Aby neuronová síť mohla znak rozpoznat, musí být screenshot z obrazovky nejprve předzpracován. Vzhledem k tomu, že se rozpoznává jen jediný znak,



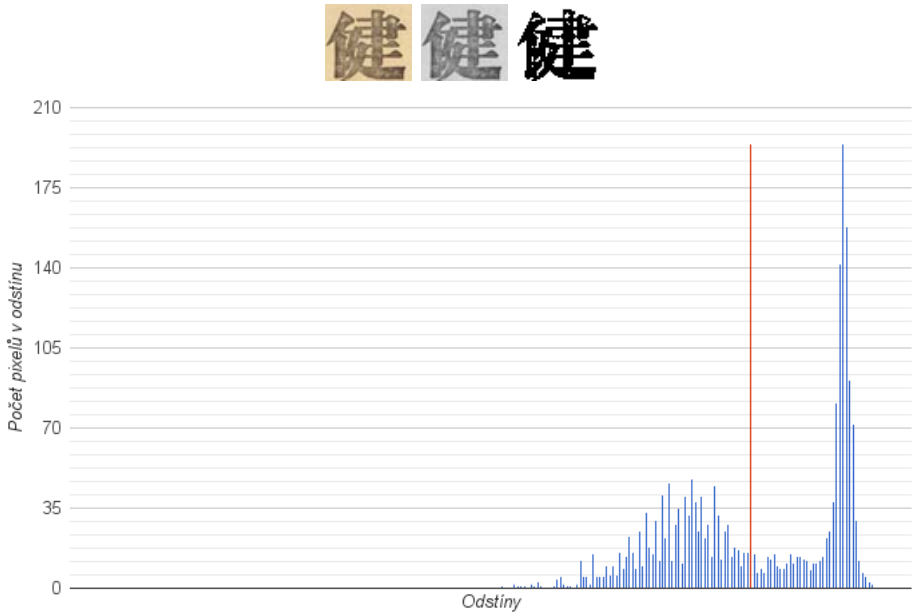
Obrázek 7.5: Průběh předzpracování z velice malého obrázku novin a histogram se zvoleným bodem prahování.

odpadá problém s prováděním segmentace textu, jež je u znaků problematická hlavně proto, že tahy ve znacích nebývají spojené (např. 小, 加, 川). Na druhou stranu by se dalo využít znalosti čtvercového tvaru všech znaků. Experimentování se segmentací tedy zatím nechávám mimo tuto práci a budu se jí věnovat v pozdějších verzích.

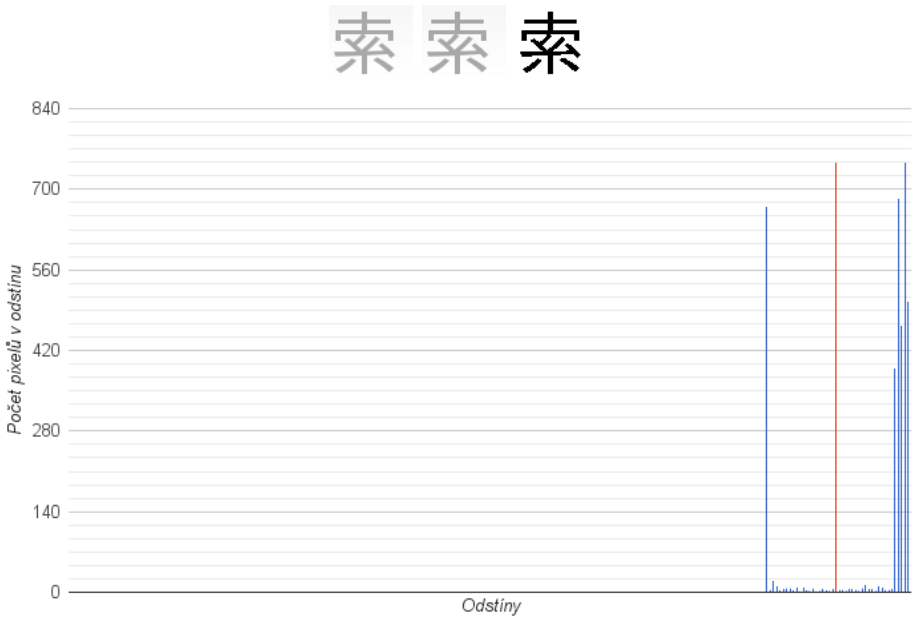
V první fázi předzpracování se provádí převedení zachyceného obrázku do 256 stupňů šedi a poté prahování (převedení obrázku do černo-bílé). Několik příkladů různých zachycených kandži a jejich cestu první fází předzpracování lze vidět na obrázcích 7.5, 7.6, 7.7 a 7.8.

Vzhledem k tomu, že během testování se ukázala konstantní hodnota prahování jako nesmyslná, bylo potřeba sáhnout po nějaké metodě, jež by ji dynamicky zjišťovala. Takových metod existuje velké množství, ale pro naše potřeby stačila nějaká opravdu jednoduchá a rychlá. Přesně takovou metodu v roce 1979 publikoval Ócu Nobujuki (大津展之), známou také jako Otsu's thresholding method. Prahování probíhá globálně (jedna hodnota prahu pro celý obrázek) a samotný práh se zjišťuje podle rozptylu ve skupinách, které chceme prahováním oddělit. V tomto případě chceme binární obraz, čili dělíme do dvou skupin – pozadí a znak.

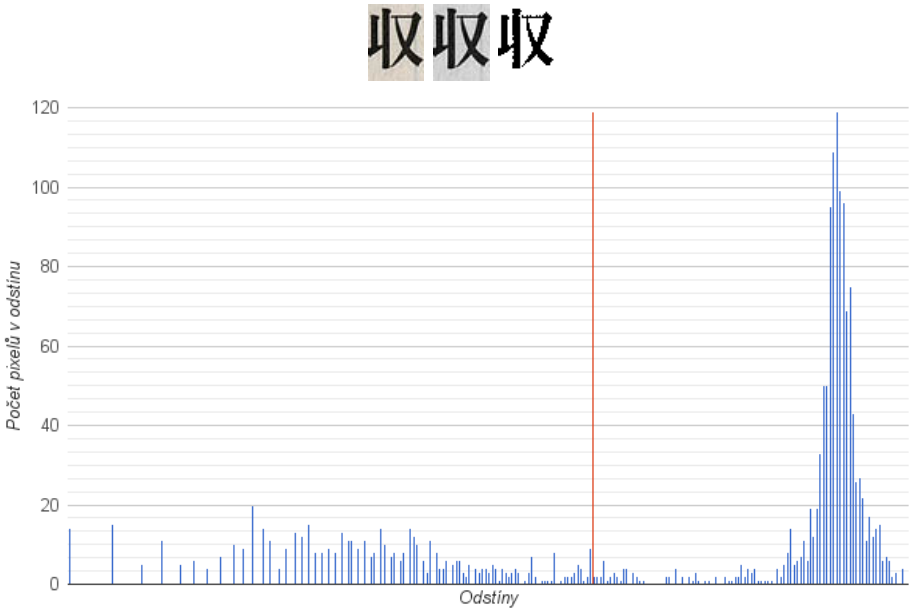
Ócuova metoda iterativně prochází všechny možné hodnoty prahu a hledá takovou hodnotu k^*, pro kterou platí rovnice 7.9. Pro úplné porozumění musíme nejprve definovat následující funkce a proměnné, aby bylo vidět, s čím rovnice pracuje. Pro podrobnější popis, viz [12].



Obrázek 7.6: Průběh předzpracování ze špatně čitelných a zažloutlých novin. Na histogramu je vidět menší celkový rozptyl.



Obrázek 7.7: Průběh předzpracování kdy pozadí a znak nekontrastují jako v předchozích případech, přesto si s nimi prahovací metoda poradila.



Obrázek 7.8: Průběh předzpracování z novin, kde je rozložení pixelů rovnoměrnější, na což také reaguje prahovací funkce a výsledek v tomto případě není zcela ideální.

Nechť L je počet stupňů šedi $[1, 2, \cdots, L]$ a počet pixelů ve stupni šedi i budeme značit jako n_i. Pro celkový počet pixelů platí $N = n_1 + n_2 + \cdots + n_L$. Poté můžeme definovat rozdělení:

$$p_i = \frac{n_i}{N}, \qquad p_i \geq 0, \quad \sum_{i=1}^{L} p_i = 1 \tag{7.1}$$

Naším cíle je rozdělení pixelů prahem k do dvou skupin C_0 a C_1 (pozadí a znak). Do skupiny C_0 patří pixely se stupní šedi $[1, \cdots, k]$ a do skupiny C_1 náleží $[k+1, \cdots, L]$. Poté je pravděpodobnost skupiny C_0, resp. C_1 následovná:

$$\omega_0 = P(C_0) = \sum_{i=1}^{k} p_i = \omega(k) \tag{7.2}$$

resp.

$$\omega_1 = P(C_1) = \sum_{i=k+1}^{L} p_i = 1 - \omega(k). \tag{7.3}$$

Zbývá nám definovat střední hodnoty:

$$\mu_0 = \sum_{i=1}^{k} iP(i|C_0) = \sum_{i=1}^{k} i\frac{p_i}{\omega_0} = \frac{\mu(k)}{\omega(k)} \tag{7.4}$$

resp.

$$\mu_1 = \sum_{i=k+1}^{L} iP(i|C_1) = \sum_{i=k+1}^{L} i\frac{p_i}{\omega_1} = \frac{\mu_T - \mu(k)}{1 - \omega(k)}, \tag{7.5}$$

kde

$$\mu(k) = \sum_{i=1}^{k} ip_i. \tag{7.6}$$

A střední hodnota pro celý obrázek:

$$\mu_T = \mu(L) = \sum_{i=1}^{L} ip_i. \tag{7.7}$$

Poslední funkcí, již je třeba definovat je rozptyl mezi skupinami C_0 a C_1, jež dělí práh k:

$$\sigma(k)_B^2 = \frac{[\mu_T\omega(k) - \mu(k)]^2}{\omega(k)[1 - \omega(k)]}. \tag{7.8}$$

Iterativním algoritmem hledáme maximum této funkce, hodnota k, pro kterou nabývá funkce maxima, je náš hledaný práh k^* [12]:

$$\sigma(k^*)_B^2 = \max_{1 \leq k\ <L} \sigma(k)_B^2. \tag{7.9}$$

Druhá fáze předzpracování funguje ve své podstatě stejně jako při generování datasetu popsaném v podkapitole 7.1.2. Naleznou se hranice znaku a poté proběhne ořezání a škálování, čímž končí fáze předzpracování.

7.1.4 Jednoduchá neuronová síť

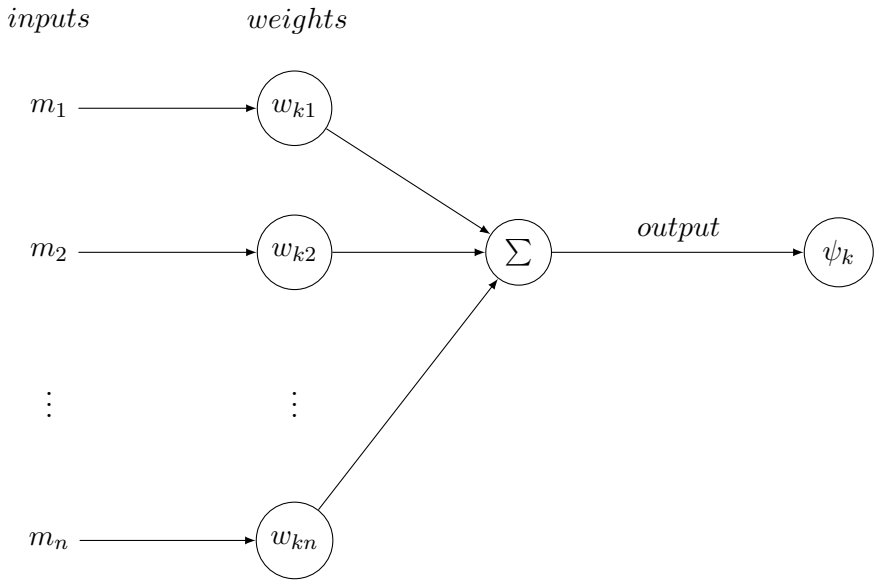
Neuronová síť je výpočetní model z oblasti umělé inteligence, předobrazem takových sítí jsou skutečné biologické neurony. Umělé neuronové sítě (ANN) se skládají ze vzájemně propojených neuronů. Modelů, jak ANN realizovat existuje celá řada, [13] používá SLP (single layer perceptron), dopřednou jednovrstvou síť perceptronů.

Do každého neuronu k (Obrázek 7.9) vstupuje bipolární matice $\mathcal{M}$ velikosti 64×64, kde „-1“ reprezentují pozadí a „1“ pixel daného znaku. Jako aktivační funkce neuronu funguje vážený součet ψ (viz níže) vstupů. Výstupem sítě je soubor znaků, jejichž hodnota ψ je nejvyšší.

Reprezentace jednotlivých neuronů realizuje třída s následující strukturou:

```
int             matrix [64][64];
int             kanjiCode;
```

Kde `matrix` je matice vah pixelů a `kanjiCode` reprezentuje 16bitový kód kandži, o kterém neuron rozhoduje.



Obrázek 7.9: Detail jednoho neuronu sítě SLP

7.1.5 Učení

Učení neuronové sítě je založeno na jednoduchém principu „učení s učitelem“. Každý neuron dostal sadu binárních matic, podle kterých nastavoval svou matici vah (na počátku byly všechny prvky nulové). Pro každou trénovací matici $\mathcal{A}$ uprav matici vah $\mathcal{W}$ následovně:

$$\mathcal{W}(i,j) = \begin{cases} \mathcal{W}(i,j) + 1 & \text{pokud } \mathcal{A}(i,j) = 1 \\ \mathcal{W}(i,j) - 1 & \text{jinak} \end{cases} \tag{7.10}$$

Tím je docíleno toho, že výskyt pixelu má dvojnásobnou váhu oproti výskytu pozadí. Výsledná matice vah pak mapuje pravděpodobnost výskytu pixelu na dané pozici, což představuje celou podstatu rozpoznávání v tomto modelu.

7.1.6 Vybavování

Vybavovací fáze (rozpoznávání) už je o něco komplikovanější. Na vstupu máme opět bipolární matici $\mathcal{M}$ velikosti 64×64, tentokrát ale matice reprezentuje znak, jež má síť rozpoznat. Každý neuron k vypočítá hodnotu ψ, definovanou jako:

$$\psi(k) = \sum_{i=1}^{64} \sum_{j=1}^{64} \mathcal{W}_k(i,j) * \mathcal{M}(i,j)$$

Zde se s článkem [13] rozcházíme. V článku se ještě pro každou matici počítala hodnota μ jako součet všech kladných hodnot v matici vah. Vzhledem k tomu, že dataset neobsahoval stejný počet obrázků pro každý font, hodnota μ, jež sloužila k vypočtení rozpoznávacího koeficientu $\mathcal{Q}(k)$:

$$\mathcal{Q}(k) = \frac{\psi(k)}{\mu(k)}$$

již není potřebná. Samozřejmě by dataset mohl být upraven, či matice znormalizovány, ale současný stav přináší jednu velkou výhodu. Vzhledem k tomu, že v datasetu se nalézá větší množství obrázků pro znaky, jež se častěji používají (v datasetu jsou fonty, které obsahovaly např. jen 200 nejpoužívanějších znaků), nabývají tak v maticích vyšších hodnot a jsou tak upřednostňovány před znaky, které umělo zobrazit jen malé procento fontů.

Rozpoznávacím kvocientem se tedy stala hodnota ψ, podle níž se znaky řadí a zobrazují uživateli.

7.1.7 Testování sítě

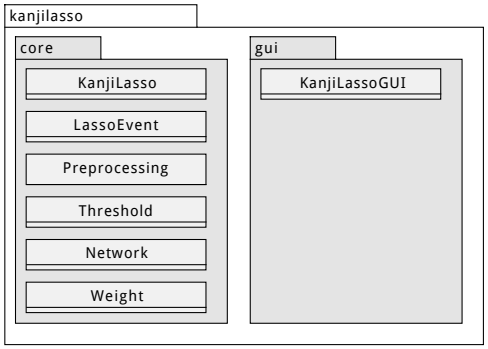
Jak se ukázalo, rozpoznání obrovskou měrou závisí na kvalitě předzpracování a tím, jak zdeformovaný a jakým způsobem uživatel znak označí. Pokud ale bude vstupem znak, který byl označen celý bez nechtěných čar vně znaku (např. sousední znak), maximálně lehce zdeformovaný, síť jej rozpozná s minimální chybou. Při veškerém testování se téměř všechny znaky zobrazily v první desítce (většinou byl chtěný znak první), zbylé znaky byly buď nečitelné i pro člověka (příliš tučný font), nebo neměly čtvercový tvar (zúžené fonty ve starých novinách).

Předzpracování funguje velmi dobře, všechny znaky z obrázků 7.5, 7.6, 7.7 a 7.8 s různým stupněm deformace síť rozpoznala. Avšak jedná se pouze o zašumění a lehkou deformaci. Pokud je znak natočen o více než několik jednotek stupňů, síť ho již nerozpozná. Rozpoznávací proces je invariantní jen k posunu, nikoliv k rotaci, převrácení apod.

I přes to považuji pravděpodobnost pixelů v této formě za nesmírný úspěch v ropoznávání tisíců znaků s takovou přesností. Svému účelu – čtení tištěného písma z obrazovky, naprosto dostačuje.

7.2 Návrh a realizace

Celé rozpoznávání dostalo jméno Kandži Laso a vlastní balíček `kanjilasso`. Protože analytický doménový model nemělo moc smysl tvořit, jelikož se v Lasu řeší spíše algoritmy než objektová analýza, rozhodla jsem se přejít rovnou k návrhu a realizaci společně. Dle architektury MVC by Laso mělo být třívrstvé, nicméně vytvářet samostatný balík pro jedinou vyčleněnou třídu se jeví jako zbytečné, proto máme vrstvy dvě, resp. dva balíky viz Obrázek 7.10, čímž máme oddělenou vrstvu prezentační `gui` od vrstvy logické `core`.



Obrázek 7.10: Architektura balíků a tříd v balíku kanjilasso

7.2.1 Načítání matic pro neuronovou síť

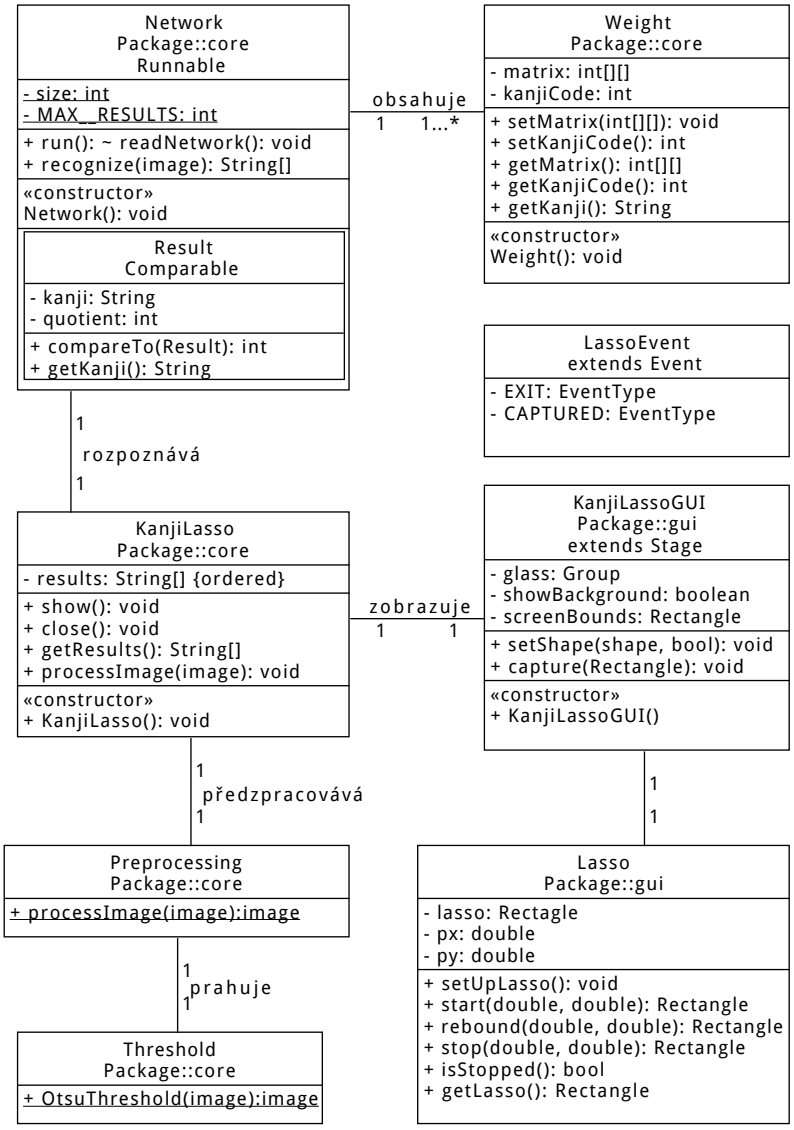
Jediný moment, kdy tato komponenta sahá na disk, představuje načítání rozpoznávacích matic do neuronové sítě třídy `Network` a naplnění hodnotami kolekci `ArrayList<Weight>`, ve které jsou matice a znaky. Při trénovacím procesu byly matice uloženy v textovém souboru jako 16bitový kód znaku (včetně párových znaků, které se na svou zobrazitelnou verzi převádí funkcí `getKanji` z třídy `Utils` v jádru) následovaný samotnou maticí. Ovšem velikost takového souboru dosahuje ke 200 MB a jeho načítání trvalo neúnosně dlouho (v průměru 31 sekund), protože se musí provádět konverze z textu na čísla. Řešením se stalo vytvoření binárního souboru a používání tříd `BufferedInputStream` a `DataInputStream` při načítání. Při zachování velikosti souboru se rychlost se snížila zhruba na 5 sekund.

Avšak i takové zpoždění by uživatel zaznamenal. Naštěstí matice zůstávají v paměti, dokud je Kandži Laso aktivní, takže není nutné načítat je u každého rozpoznávaného znaku. Proto bylo čtení matic vyčleněno do vlastního vlákna, kvůli kterému třída `Network` implementuje rozhraní `Runnable`. V mezičase se načte uživatelské rozhraní Lasa a uživatel může začít označovat. Teprve až po předzpracování označeného znaku se volají metody, jež potřebují s maticemi pracovat. Pro jistotu se zde volá synchronizace `networkReader.join();`, aby se předešlo případným problémům.

Jakmile uživatel Kandži Laso ukončí, matice jsou z paměti uvolněny a volá se explicitně Garbage Collector, jelikož může být takový úbytek v paměti znát a navíc je zcela zbytečné zabírat paměť informacemi, které nemají užití do dalšího zapnutí Lasa.

7.2.2 Logická vrstva

Nyní se pojďme věnovat logické vrstvě a tomu, co dalšího se děje při startu Lasa. V návrhu (Obrázek 7.11) máme třídu `KanjiLasso`, jež se v jádru aplikace



Obrázek 7.11: Návrh architektury komponenty Kandži Laso

Obrázek 7.12: Rozdíl v převodu znaků z novin do stupňů šedi použitím `ColorConvertOp` (nahoře) a metody NTSC. Jak lze pozorovat, metoda NTSC zachovává znak stále ostrý a vzhledem k původním novinovým *barvám* došlo jen k malé korekci, kdežto použití konverzní třídy obrázek nesmyslně ztmavuje.

volá jako první a přes ni se také řídí chování Lasa. Jakmile uživatel spustí Laso, třída `KanjiLasso` vytváří samostatné vlákno, které se postará o načtení rozpoznávacích matic, aniž by uživateli zamrzlo uživatelské rozhraní.

Jakmile uživatel označí požadovaný znak, uživatelské rozhraní o tom informuje logickou vrstvu vystřelením laso události `CAPTURED`. Obrázek nyní putuje do třídy `Preprocessing`, kde se nejprve převede do odstínů šedi. Výsledek konverze za použití knihovního převaděče `ColorConvertOp` nepostačoval, jelikož výsledný obrázek byl příliš tmavý. Proto byla v převodu do odstínů šedi použita metoda NTSC (National Television System Committee), která složkám RGB nastavuje váhy dle následujícího kódu:

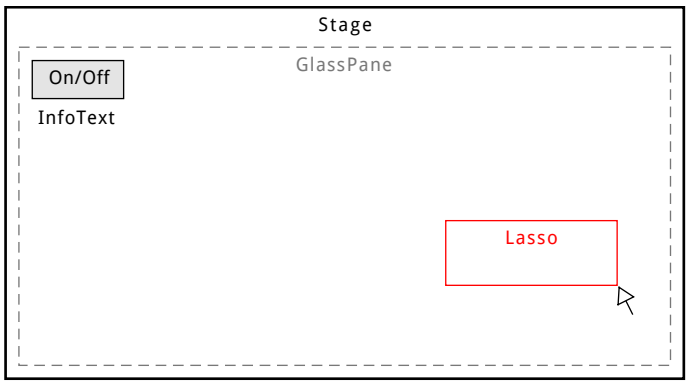
```
grayPixel = ( .299 * red + .587 * green + .114 * blue );
```

Rozdíl obou metod ilustruje Obrázek 7.12. Pro méně komplikované znaky by mohlo být ztmavení výhodou, nicméně složité znaky se stávají hůře rozpoznatelnými. Obrázek ve stupních šedi nyní prochází prahováním Ócuovou metodou 7.1.3, kde se nejprve z obrázku vytvoří matice pixelů a pole `histogram`, kde pozice i představuje stupeň šedi a hodnota `histogram[i]` reprezentuje počet pixelů tohoto stupně. Histogram se poté používá při hledání prahu[13].

Po aplikaci prahu a získání binárního obrázku se vyhledávají hranice znaku jako v podkapitole 7.1.2, pokud hranice nejsou nalezeny (prázdný obrázek), předzpracování vrací třídě `KanjiLasso` hodnotu `null`. Jinak se pokračuje v ořezání, škálování (zajišťují funkce `getSubimage` a třída `AffineTransform`) a předzpracovaný obrázek se již může rozpoznávat.

Ve třídě `Network` existuje ještě vnitřní třída `Result` implementující rozhraní `Comparable`, jež slouží pro uložení vyhledaných znaků a jejich seřazení dle rozpoznávacího koeficientu. Procházení matic zajišťuje následující kód:

[13] V programu se s malými úpravami používá efektivní kód Dr. Andrewa Greensteda `http://www.labbookpages.co.uk/software/imgProc/otsuThreshold.html`



Obrázek 7.13: Návrh uživatelského rozhraní pro Kandži Laso

```
weights.stream().forEach( ( Weight w ) -> {
          results.add(
             new Result( w.getKanji(),
                getRecognitionQuotient( w, kanji ) ) );
       } );
```

Funkce `getRecognitionQuotient` počítá rozpoznávací kvocient pro danou váhu `w` a matici rozpoznávaného znaku `kanji`:

```
private int getRecognitionQuotient(Weight w, short[][] kanji){
      int psi = 0, i, j;
      int[][] matrix = w.getMatrix();
      for ( i = 0; i < size; i++ )
          for ( j = 0; j < size; j++ )
              psi += matrix[ i ][ j ] * kanji[ i ][ j ];
      return psi;
}
```

Kolekce s výsledky se poté seřadí `Collections.sort(results)` a podle nastavené hodnoty `MAX_RESULTS` se do třídy `KanjiLasso` vrací určitý počet znaků v seřazené kolekci `ArrayList<String>`, jež se zobrazí uživateli.

7.3 Prezentační vrstva

Finální návrh této vrstvy na Obrázku 7.13 byl vytvořen až po (nalezení a) vyzkoušení potřebných technologií. Přičemž se volba knihoven JavaFX ukázala jako ne úplně dobrá. Nicméně výsledek se zdá být funkční a než budou všechny funkce JavyFX implementované, jistě postačí.

Návrh provázely dva větší problémy:

1. vytvoření průhledného okna a
2. zachycení znaku z obrazovky.

Jeden z nich se částečně podařilo vyřešit, druhý nikoliv.

7.3.1 Průhledné okno

Vytvořit *neviditelné* okno přes celou obrazovku lze jednoduše:

```
stage.initStyle( StageStyle.TRANSPARENT );
stage.setFullScreen( true );
stage.setAlwaysOnTop( true );
```

Problém tkví ve faktu, že takové okno nepřijímá žádný `MouseEvent` – nedozví se o pohybu, či kliknutí myši[14]. Jak tedy může uživatel označit znak, který nevidí? Řešení se nabízelo takové, že by uživatel daný obrázek do programu nějakým způsobem nahrál a posléze z něho rozpoznával. Což by možná nevadilo při větším množství neznámých znaků na obrázku, nicméně pro jeden znak by taková námaha byla zbytečná a rychlejší způsob by nabídlo vyhledání podle radikálů. Ale tomu se v této aplikaci chceme vyhnout. Žádné zbytečné klikání.

Tuto situaci řeší nastavení poloviční průhlednosti (resp. 51%). Položka v návrhu `GlassPane` tak dostála svému jménu a okno opravdu vypadá jako sklo. Pro pohodlnost uživatele se během tažení lasa sklo vypíná a znak tedy může být označen přesněji.

7.3.2 Zachycení znaku z obrazovky

Tento oříšek se pomocí JavaFX rozlousknout nepodařilo, nevím o způsobu, jež by to umožňoval korektně. Existuje sice `Robot` v balíku `com.sun.glass.ui`, který funguje velice dobře. Bohužel se jedná o interní API, čemuž odpovídá i návratová hodnota „chytací“ funkce a absence dokumentace, jež by pomohla s porozuměním převodu takového formátu do použitelného obrázku či matice. Respektive, převedení se zdařilo, avšak nemáme jistotu, že takové řešení funguje na všech systémech. Ani Java se na všech systémech a za všech okolností nechová stejně.

Nezbývalo než sáhnout po zastaralé třídě `Robot` z AWT, což s sebou nese významný synchronizační problém. Vývojáři JavaFX mysleli na případy, kdy programátor bude chtít do své aplikace vložit například tlačítko z knihoven Swing / AWT a připravili speciální kontejnery, jež se o synchronizaci mezi těmito složitými GUI Toolkity starají. Nicméně pro třídu `Robot` nic takového

[14] `http://docs.oracle.com/javase/8/javafx/api/javafx/stage/Window.html#opacityProperty`

není. Musí se tedy vytvořit vlákno a spustit jej pomocí synchronizační funkce `SwingUtilities.invokeLater`. Funkce `createScreenCapture` sice funguje správně, ale docházelo k synchronizačním problémům s vláknem, na kterém běží JavaFX. AWT totiž nedokáže zachytit znak, pokud jej překrývá komponenta z JavaFX (ani průhledná). Před spuštěním vlákna se tedy ještě volá:

```
stage.setFullScreen( false );
stage.setIconified( true );
```

V tuto chvíli dochází ke kolizi. Než se stihne okno minimalizovat, `Robot` obrazovku vyfotí, cca 30 % takto pořízených snímků bylo nedefinovaných (černých). Tuto nemilou situaci dočasně řeší uspání AWT vlákna na 0,2 sekundy, což se na testovaných systémech ukázalo jako postačující a uživatele neobtěžující. Každopádně jde opravdu jen o dočasné řešení.

7.3.3 Laso

Při hledání funkce, jež by uměla vyfotit část obrazovky, jsem narazila na vynikající nápad[15] člověka jménem Aljoscha Rittner, který v roce 2011 prezentoval právě takové laso, jež umožňuje vytvořit snímek z výřezu obrazovky. Část jeho kódu se v LUKA používá, ačkoliv s velkými úpravami, jelikož Rittnerův kód se dnes nedaří ani zkompilovat. Mnohé funkce se v JavaFX změnily, některým zůstalo jméno a změnila se jejich funkčnost. Přesto se jedná o jediné použitelné řešení zachytávání, které se mi podařilo najít.

7.4 Komunikace s KanjiSearch

Kandži Laso nekomunikuje s jádrem, jelikož se jedná o komponentu napojednou na `KanjiSearch`, s ní komunikuje nepřímo vystřelováním následujících událostí:

- Do `ControlEvent` událostí si přidalo `KANJI_LASSO_CLOSE`, které se používá, pokud uživatel Laso vypne – klávesou ESC, když je aktivní, nebo ze systémové lišty. Uživatelské rozhraní `KanjiSearch` se poté přepíná do módu vyhledávání podle parametrů.
- Dalším Control Eventem je `KANJI_LASSO_RESULTS_READY`, které se vystřelí v momentě, kdy síť rozpozná a seřadí možné znaky. `KanjiSearch` se poté stará o jejich zobrazení uživateli v klasickém vyhledávání.

Používání těchto eventů snižuje provázanost obou komponent na minimum a tím také závislost na konkrétní implementaci.

[15] `http://www.aljoscha-rittner.de/blog/archive/2011/03/09/javafxdev-screen-capture-tool-with-200-lines-and-500ms-startup-time/`

Kapitola 8

Jádro

Jádro spojuje ostatní komponenty do jednoho celku, řídí jejich činnost a je-li to nutné, funguje také jako směrovač dat. Kromě samotného jádra se v této kapitole věnuji všem ostatním podpůrným balíkům, třídám a funkcím, jež nemusí být nutně obsaženy v balíku s jádrem.

Analýza – Návrh – Realizace – Testování probíhaly vždy po částech v závislosti na stavu jednotlivých komponent. Pokud bylo potřeba dodat nějaký ovládací prvek (nedostupný v JavaFX, nebo jen specifický), jež by v budoucnu mohl najít využití v jiných komponentách, vložil se do balíku patřícímu k jádru programu.

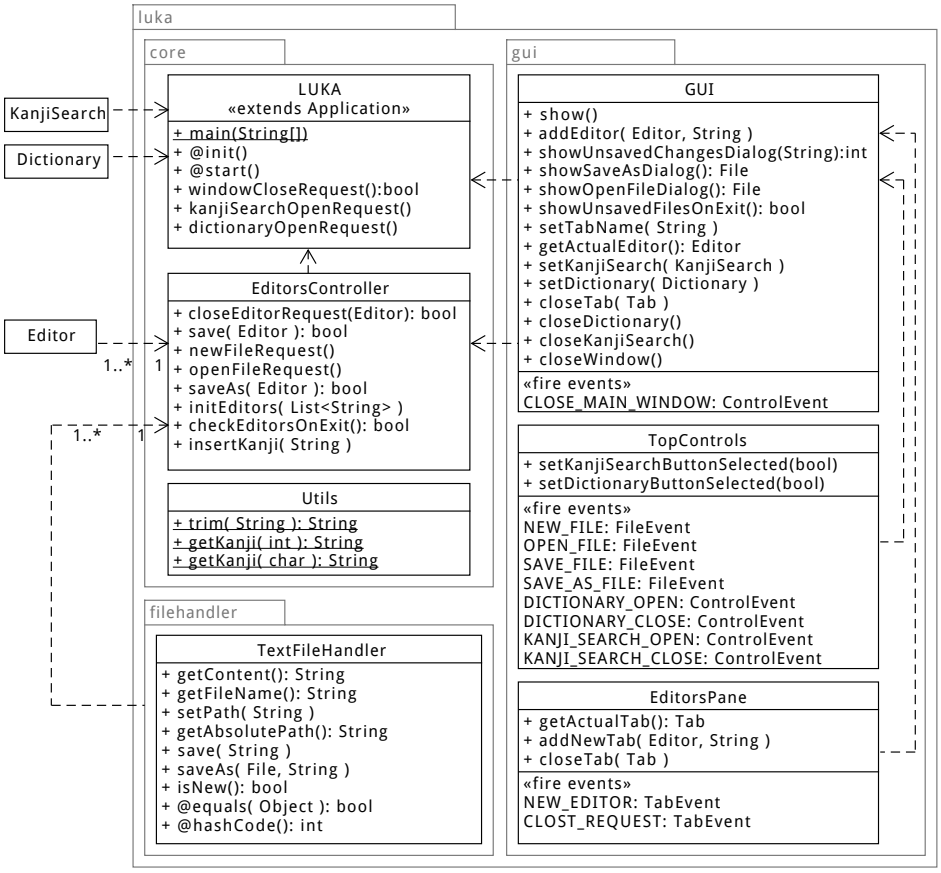
8.1 Analýza a návrh

Navržení jádra tedy vycházelo z potřeb komponent a také z definovaných komunikačních toků v podkapitole 3.8. Z té vyplývá, že jádro se stará o iniciální načtení dat (Slovníku a Vyhledávání), spouštění a ukončování komponent, ale hlavně o kontrolu nad editory, jichž může být v programu více (avšak jen jediný aktivní). Protože funkcí pro manipulaci s editory je vícero, byly vyčleněny do vlastní třídy. Celkový model návrhu, na který se nadále budu odkazovat, je na Obrázku 8.1.

8.1.1 Editory

Třída `EditorsController` se stará o veškerou interakci s editory, naslouchání událostem, jež chtějí s textem manipulovat. Aby editorová komunikace neprobíhala přes hlavní třídu jádra, obsahuje tato třída také vlastní referenci na prezentační vrstvu. Z té získává informace o tom, jaký editor momentálně uživatel vidí, chystá-li se nějaký zavřít apod.

Také komunikuje se třídou `TextFileHandler` starající se o diskové operace. Jako svůj privátní atribut obsahuje hashmapu, kde je klíčem editor a hodnotou instance třídy `TextFileHandler`.



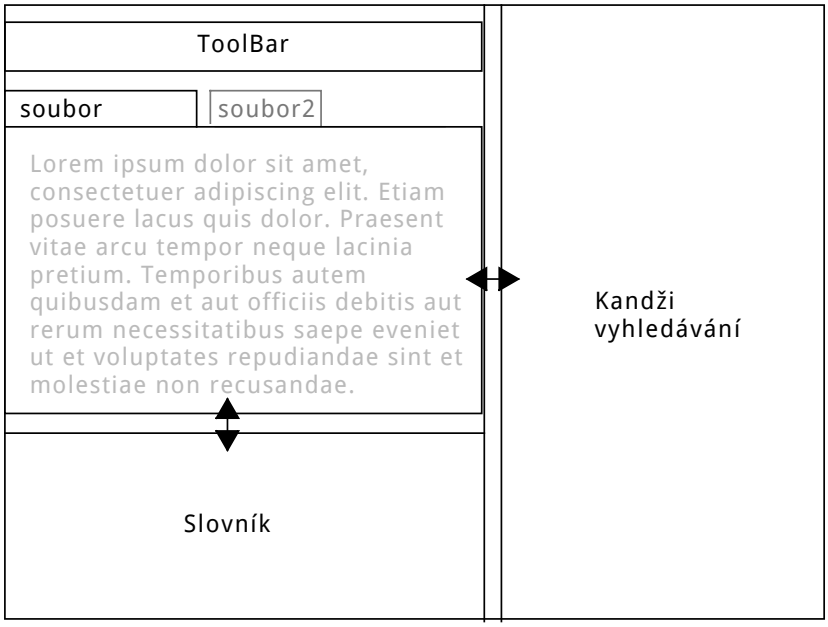
Obrázek 8.1: Návrh modelu pro jádro

Bylo potřeba vymyslet, jakým způsobem řešit situaci, kdy by uživatel chtěl otevřít jeden soubor vícekrát. Z toho důvodu přibylo v návrhu třídy překrytí metod `equals` a `hashCode`. Dva soubory jsou si rovny, pokud se rovnají jejich absolutní cesty, kromě souborů, jež žádnou cestu nemají, jelikož jsou nové a neuložené. Tento způsob zamezí otevírání jednoho souboru ve více editorech a zároveň dovolí uživateli mít libovolné množství nových souborů.

8.1.2 Návrh uživatelského rozhraní

Aby se do jednoho okna vešly tři poměrně obsáhlé komponenty – záložky s editory, slovník a kandži vyhledávání – bylo nutné zajistit jim dostatečný prostor. Avšak slovo „dostatečný“ může mít každý uživatel nastavené na jinou hodnotu, proto se bude moci s velikostí komponent hýbat, jak lze vidět na Obrázku 8.2.

Pro přepínání mezi editory slouží záložky spravované třídou `EditorsPane`.



Obrázek 8.2: Návrh rozdělení okna pro uživatelská rozhraní jednotlivých komponent

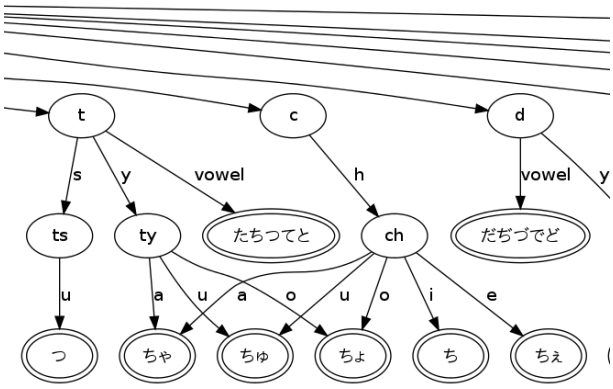
Vzhledem k tomu, že vidět lze obsah vždy jediné z nich, bude vkládání do editoru implementačně jednodušší.

Poslední samostatnou složkou uživatelského rozhraní jsou ovládací tlačítka ve třídě `TopControls`. Třída obsahuje jeden toolbar s tlačítky na ovládání ostatních komponent (otevřít / zavřít) a souborové operace. Vytvářet klasickou lištu s menu zatím nemělo smysl, jelikož ovládacích prvků je málo. Později by neměl být problém lištu případně doprogramovat.

8.1.3 Překladové automaty pro abecedy

Pro správnou funkci slovníku, kandži vyhledávání a později také IME, musí mít program způsob, jakým konvertovat z latinky do kany a zpět. K tomuto účelu slouží balík `kanaconversions` obsahující třídy s překladovými automaty pro konverze

- kana → latinka,
- hiragana → katakana,
- katakana → hiragana,



Obrázek 8.3: Výřez z návrhu překladového automatu pro převod z latinky (anglického přepisu) do hiragany.

- latinka → hiragana,
- latinka → katakana,

a rozhraní `Convertible`, které všechny implementují (pro sjednocení chování). Pro mapování znaků na slabiky v automatech slouží hashovací mapy díky jejich konstantní asymptotické složitosti pro metodu `get`.

Třídy konvertují z a do anglického přepisu a japonský (zatím) pouze z latinky. Český převod už je oproti anglickému přepisu nesmírně ztrátový, co se týče informací o původních samohláskách: ó může být おお, ale i おう. Stejně tak opačně こう může být kó (がっこう – gakkó – škola) nebo také ko'u, pokud slabikou こ končí čtení jednoho znaku a slabikou う čtení následujícího znaku začíná (こうま – ko'uma – hříbě), stejná situace nastává např. s えい a ええ [4].

Anglický přepis má také své problémy, například じ a ぢ, obojí se čte „dži“ (anglicky „ji“). Ovšem mezi nástroji panuje úzus, kdy se při napsání „dži“ zobrazí to, které se používá častěji. V tomto případě to je じ. Pokud chci zapsat ぢ, použiji jeho pravidelnou (japonskou) verzi – „di“.

K nastolení nějakého pravidla, které by řešilo problém s českým přepisem potřebuji více než konzultaci se studenty. Jelikož i dvě české učebnice japonštiny [4] a [14] s v přepisu rozcházejí, od implementace bylo upuštěno.

Grafické znázornění jednoho překladového automatu obsahuje mnoho uzlů a hran, proto na Obrázku 8.3 je jen jeho výřez.

8.2 Realizace

V této části se věnuji pouze prezentační vrstvě, jelikož všechno důležité o ostatních vrstvách již bylo řečeno v předchozí části.

8.2.1 Prezentační vrstva

Uživatelské rozhraní aplikace (třída `GUI`) podle návrhu rozděluje plochu na tři části. Děje se tak pomocí layoutu `SplitPane` a vzhledem k obsáhlosti jednotlivých komponent se pravá část okna věnuje Kandži vyhledávání a o levou polovinu se dělí `EditorsPane` se Slovníkem.

Díky prezentační vrstvě jádra (jež je aktivní po celou dobu běhu aplikace) také fungují události – hlavní třída nastavuje oknu (`Stage`) reakce na ně pomocí metody

```
1 public <T extends Event> void addEventHandler(
2         final EventType<T> eventType,
3         final EventHandler<? super T> eventHandler ) {
4
5     stage.addEventHandler( eventType, eventHandler );
6 }
```

Třída umožňuje vyvolat dialogy při práci se soubory – neuložené změny v souboru, zobrazení `FileChooser` při ukládání či otevírání souborů apod. Stejně tak ovlivňuje chování tlačítek v toolbaru a např. i jména záložek, ve kterých jsou editory. Všechna tato komunikace probíhá přes třídu `GUI`, jež zastřešuje ostatní prvky uživatelského rozhraní a stará se o zavolání funkce příslušného ovládacího prvku.

8.2.2 Text s kandži

Třída `KanjiText` rozšiřuje třídu `Text` o kontextové menu, jež umožňuje uživateli zobrazit si informace o znacích v infopanelu Kandži vyhledávání. Kontextové menu se tvoří tak, že se zobrazovaný text rozparsuje na jednotlivé znaky a pomocí dotazování se na třídu `KanjiManager` (spravující všechny znaky z KANJIDIC2) zjistí, ke kterým znakům má informace – v textu se může objevit latinka, číslice, interpunkce, cokoliv. Seznam takový znaků se poté objeví v kontextovém menu. Zvláštní pozornost je věnována také párovým znakům, pomáhá zde statická metoda `getKanji` ze třídy `Utils` jež přijímá `char` či `int` a vrací reprezentaci zobrazitelného kandži jako `String`. Do menu se také přidává možnost vložit text do editoru a vyhledat jej ve slovníku.

Kromě možností kontextového menu disponuje `KanjiText` také metodou `setBracket`, jež text obalí hranatými závorkami používanými pro zobrazení čtení. Dále pak metody `setColor` a `setSize`, kde lze použít předefinované konstanty třídy (`HUGE`, `BIG`, `MEDIUM` a `SMALL`).

8.2.3 Pole pro čísla

Pole přijímající jen celá čísla v době implementace nebylo k dispozici, proto přišla na řadu vlastní implementace – `NumberField`. Rozšiřuje `TextField` a nastavuje zde `ChangeListener`, který pomocí regulárního výrazu zajišťuje

přítomnost čísel (vložení jiných znaků se nezdaří). Pomocí metod `getNumber` a `setNumber` se nastavují / získávají hodnoty.

Potomkem této třídy je `LimitedNumberFiled`, jež používá uživatelské rozhraní Kandži vyhledávání pro zobrazení polí pro počty tahů. Pro úplnost byly implementovány čtyři konstruktory:

```
public LimitedNumberField( int min, int max );
public LimitedNumberField( int min );
public LimitedNumberField( int min, String text );
public LimitedNumberField( int min, int max, String text );
```

kdy se kromě minima a maxima povolených hodnot vkládá také iniciální hodnota. Přidaná funkcionalita této třídy zajišťuje, že uživatel nevloží hodnoty menší nebo větší, jež jsou zadané hranice. Uživatel tedy např. pro počet tahů nemůže zadat 0.

8.2.4 Dlaždice na výsledky

Třída `ResultTile` reprezentuje jednu dlaždici, jež se zobrazuje ve vyhledávání (momentálně jen slovníku, na žádost uživatelů v příští verzi i v Kandži vyhledávání). Dlaždice plně využívá `KanjiText` a skládá se ze třech hlavních částí:

- hlavní složka, ve slovníku fráze v kandži, s největší velikostí fontu,
- čtení – s nastavenými hranatými závorkami kolem textu a střední velikostí fontu,
- významy – s malou velikostí fontu (jelikož jich bývá mnoho).

Ke každé této části se ještě nejmenším fontem vypisují doplňující informace, k hlavní složce a čtení se přidávají pod ni a u významu vedle do řádku.

Ale protože výsledky s dlaždicemi se vkládají do kontejneru `ScrollPane`, musí se manuálně upravovat šířka významů kvůli jejich množství – jinak by `ScrollPane` povolil dlaždice příliš široké. K zajištění rozumné šířky slouží následující kód:

```
1 widthProperty.addListener(
2  ( ObservableValue<? extends Number> observable,
3    Number oldValue, Number newValue ) -> {
4      wrappingWidthProperty.setValue(
5       widthProperty.doubleValue() / columns - columns * 10 - 10 );
6     } );
```

kvůli kterému se při vytváření dlažice musí uvést počet zobrazovaných sloupců a také `WidthProperty` kontejneru obalujícího `ScrollPane`.

8.3 Testování

Testování jádra probíhalo postupně s implementací ostatních komponent a funguje tedy jako komplexní testování pro celou aplikaci.

Kromě jednorázových testů na jednotlivé funkce, probíhaly také testy na celé třídy (blackbox). Při takovém testování se odhalila velká spousta chyb v překladových automatech, jež způsobovaly zacyklení programu. Po opravě by všechny automaty měly být úplné a konečné.

Další chyby se týkaly zachytávání událostí, např. komponenta vystřelila událost, ale ovládací prvek, jež ji měl zachytit nebyl v tu dobu viditelný, což se konkrétně týkalo vyhledávání v reálném čase podle radikálů – jejich seznam překryl „naslouchací“ plochu a vyhledávání pomocí `KANJI_SEARCH` neproběhlo.

Jiné, menší chyby vznikaly z nepozornosti, ale jejich důvod byl brzy odhalen díky testování funkce ihned po její implementaci.

Nechybělo ani zátěžové testování se sledováním používaných zdrojů pomocí `jconsole`. Výsledky ukázaly (Obrázek 8.3), že i při největším zatížení (vyhledávání a rychlé používání Lasa) program využije maximálně 30 % CPU a s opakovaným vyhledáváním se nedostane nad 500 MB, což není tak hrozné, když vezmeme v úvahu, že drží v paměti všechny znaky a celý slovník.

Testování proběhlo na systémech Debian 7.8, x86_64 GNU / Linux, Windows 8 a verze Javy do 1.8.40.

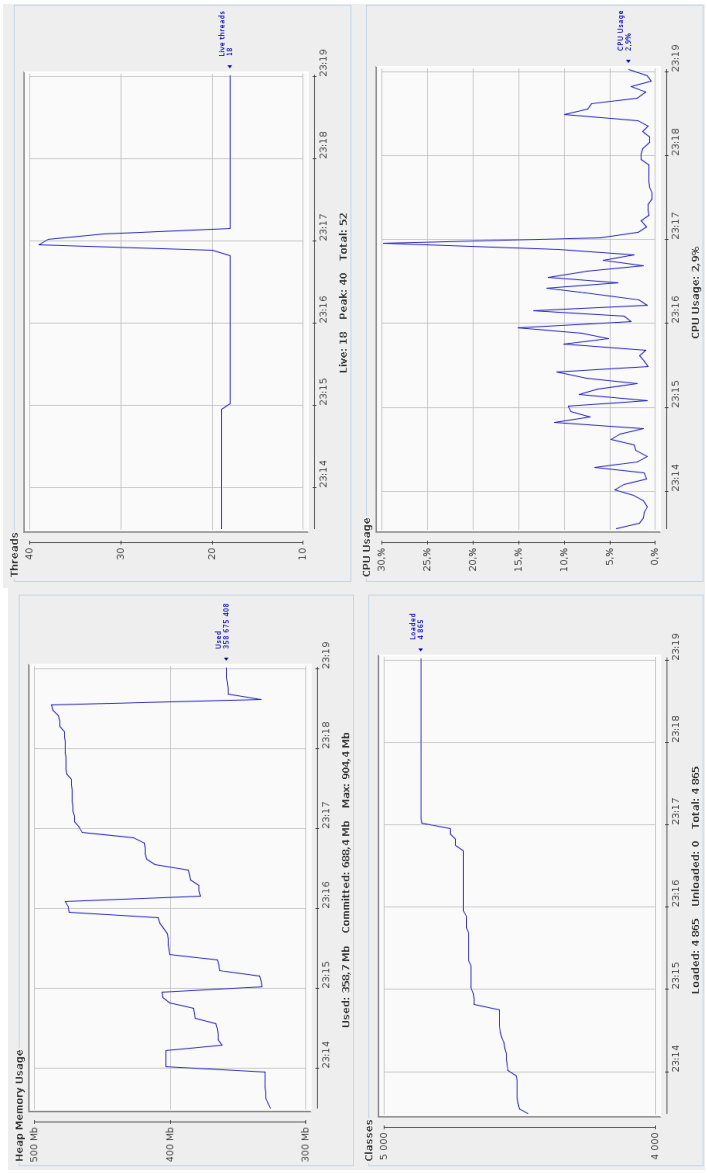
8.3.1 Chyba při zavírání záložky

Při testování správného otevírání, ukládání a zavírání souborů se ukázala další z chyb v JavěFX. Objeví se v momentě, kdy uživatel chce zavřít záložku s editorem, avšak v souboru jsou neuložené změny. Rozhraní událost zavření záložky zahodí a zobrazí uživateli dialog ohledně neuložených změn. Pokud chce uživatel i přes to záložku zavřít, nelze takovou věc korektně v kódu zapsat. Prosté `getTabs.remove(tab)` sice záložku zavře, ale již se neprovedou potřebné změny. Vývojáři o chybě vědí a její oprava se plánuje až pro verzi Java 9[16]. Tzv. workaround pro tuto chybu stále nefunguje koretně, jelikož se neaktualizuje `SelectionModel`, jež má třída využívala ke zjištění aktuální editoru. Dokud se problém nevyřeší, prohledávají se záložky v cyklu.

8.3.2 Uživatelské testování

Testování uživatelů probíhalo na systémech Windows 8 a Windows 7 (64bit). Krátce po spuštění LUKA na Windows spadlo JRE a vytvořilo soubor s chybou `EXCEPTION_ACCESS_VIOLATION`. Jak bylo později zjištěno, program vyčerpal všechnu přidělenou paměť. Proto bylo do spouštěcích souborů (.bat a .sh) přidán přepínač `-Xmx1000m` pro zvýšení paměti. Problém se tímto vyřešil.

[16]Podrobnosti na `https://javafx-jira.kenai.com/browse/RT-36471`



Obrázek 8.4: Průběh zátěžové testu na grafech z `jconsole`. Aplikace je nejnáročnější během načítání Kandži Lasa a posléze během jeho používání. První graf ukazuje využití paměti na haldě, druhý sleduje počty načtených tříd, třetí počet vláken a poslední vytížení procesoru.

Na Windows 8 se vyskytl ještě jiný problém – okno Lasa se po prvním rozpoznaném znaku nevrátilo do své původní pozice, tedy na celou obrazovku. Místo toho vypadalo, že je maximalizované, nicméně bylo o kousek posunuté mimo obrazovku. Další rozpoznávání pak neproběhlo korektně, protože souřadnice předávané třídě Robot byly taktéž posunuté. Důkladným zkoumáním kódu byla odhalena příčina – volání následujícího kódu:

```
stage.setFullScreen( false );
stage.setIconified( true );
//vyfocení obrazovky
```

```
stage.setFullScreen( true );
stage.setIconified( false );
```

Když bylo okno minimalizované a zavolala se maximalizace, posunulo se. Stačilo tedy prohodit poslední dva řádky a i na Windows 8 začalo Laso fungovat.

Závěr

Tato práce si kladla za cíl vytvořit funkční prototyp programu pomáhající studentům japonského jazyka s výukou. Funkce definované v zadání zahrnovaly specializovaný editor, konverze fonetických abeced do latinky a zpět, vyhledávání znaků kandži dle mnoha kritérií, slovník a rozpoznání napsaného znaku pomocí výpočetní inteligence. Přestože mohly některé části zůstat ve stádiu návrhu, byly implementovány všechny a v návrhu zůstaly jen nové nápady, jež se zrodily při rešerní analýze ostatních nástrojů používaných studenty a požadavků budoucích uživatelů.

Od rozpoznávání napsaného znaku bylo upuštěno z důvodu náročnosti takového úkolu – rozpoznání tisíců znaků, silně zdeformovaných psaním počítačovou myší – žádná z navržených metod nepřinesla uspokojivé výsledky. Avšak tato funkcionalita byla nahrazena jinou metodou rozpoznávání (japonštináři vřeleji přijatou), a to znaku přímo z obrazovky. Zde výsledky předčily všechna očekávání.

Další rozvoj

Mým plánem je pokračovat v práci na programu a vytvořit tak plnohodnotný nástroj, který bude užitečný všem uživatelům. Kromě implementování funkcí, jež zůstaly v této práci pouze v návrhu, bych ráda přidala i jiné (například výuku geografie Japonska pomocí vyplňování slepých map) a vylepšila rozpoznávání, aby dokázalo identifikovat nejen jediný znak, ale i celý řádek, potažmo odstavec znaků.

Kromě dalších funkcionalit plánuji za pomoci japanologů vytvořit zdrojová data (na která je sice program připraven, avšak nejsou k dispozici) s informacemi o znacích v češtině a také česko-japonský a japonsko-český slovník, volně dostupný a ve strojově čitelném formátu.

Literatura

[1] Máme rádi Japonštinu: Radikály. 2010, [cit. 2014-03-30]. Dostupné z: http://www.japonstina.net/japonske-pismo/kandzi/radikaly/

[2] Kábrt, F.: WaKan Project Website. [cit. 2014-03-30]. Dostupné z: http://wakan.manga.cz/

[3] Rosenthal, G.: JWPce Support Information. [cit. 2014-03-16]. Dostupné z: http://www.physics.ucla.edu/~grosenth/jwpce.html

[4] Nymburská, D.; Vostrá, D.; Sawatari, M.: *Japonština*. Leda, 2007, ISBN 978-80-7335-074-1.

[5] Elmes, D.: Anki – powerful, intelligent flashcards. [cit. 2014-04-08]. Dostupné z: http://ankisrs.net/

[6] Net Applications: Operating system market share. [cit. 2014-04-03]. Dostupné z: http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=11&qpcustomb=0&qpdt=1

[7] Electronic Dictionary Research and Development Group: Tanaka Corpus. [cit. 2014-04-08]. Dostupné z: http://www.edrdg.org/wiki/index.php/Tanaka_Corpus

[8] Oracle Corporation: 8u40 Update Release Notes. [cit. 2015-04-09]. Dostupné z: http://www.oracle.com/technetwork/java/javase/8u40-relnotes-2389089.html

[9] Goyvaerts, J.: Word Boundaries. [cit. 2015-04-09]. Dostupné z: http://www.regular-expressions.info/wordboundaries.html

[10] Oracle Corporation: HTMLEditor (JavaFX 8). [cit. 2015-03-09]. Dostupné z: http://docs.oracle.com/javase/8/javafx/api/javafx/scene/web/HTMLEditor.html

[11] Mikula, T.: Dropped Features. [cit. 2015-03-09]. Dostupné z: https://github.com/TomasMikula/RichTextFX/wiki/Dropped-Features

[12] Otsu, N.: A Threshold Selection Method from Gray-Level Histograms. [cit. 2015-04-06]. Dostupné z: http://web-ext.u-aizu.ac.jp/course/bmclass/documents/otsu1979.pdf

[13] V. Murarka; S. Mehta; D. Upadhyay; aj.: Alphabet Recognition Using Pixel Probability Distribution. [cit. 2015-03-29]. Dostupné z: http://www.ijser.org/researchpaper%5CAlphabet-Recognition-Using-Pixel-Probability-Distribution.pdf

[14] Krouský, I.: *Učebnice Japonštiny*. H&H, 2001, ISBN 80-86022-98-6.

Příloha **A**

Seznam použitých zkratek

GUI Graphical user interface

XML Extensible markup language

JLPT Japanese-Language Proficiency Test

IME Input Method

SKIP System of Kanji Indexing by Patterns

RTF Rich-Text Format

HTML HyperText Markup Language

JDK Java Development Kit

JRE Java Runtime Environment

MVC Model View Controller

CSS Cascading Style Sheets

Příloha B

Uživatelská příručka

B.1 Instalace

LUKA ke svému běhu potřebuje Javu verzi 8u40 a vyšší. Lze ji stáhnout na stránce `http://www.java.com/`. Po instalaci Javy zkopírujte vše ze složky `exe` umístěné na přiloženém CD do libovolného adresáře na svém systému.

B.2 Spuštění

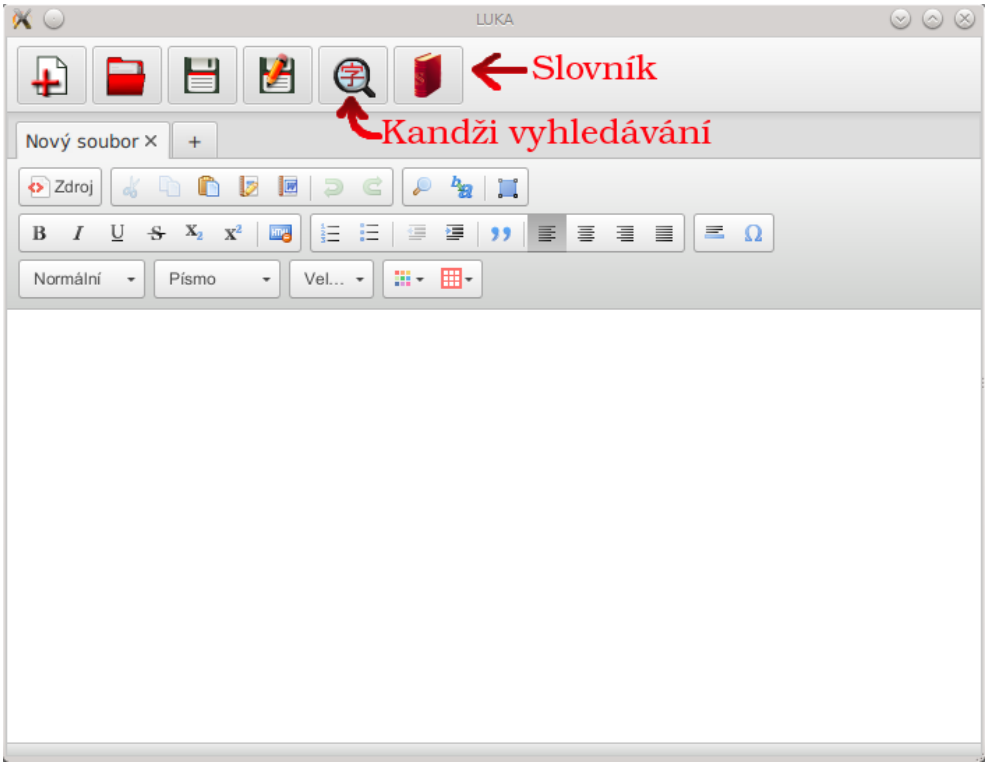
Na systémech Windows spustíte program poklepáním na `LUKA.exe`, případně pomocí dávkového souboru `LUKA.bat`. Na Unixových systémech spuštěním shellového skriptu `LUKA.sh`. Nedoporučuje se spouštet program přímo přes `LUKA.jar` kvůli nastavení limitů pro používanou paměť.

B.3 Používání programu

Po spuštění programu se objeví okno (Obrázek B.1) s editorem. Slovník lze spustit či vypnout kliknutím na příslušnou ikonku, stejně tak Kandži vyhledávání.

B.3.1 Slovník

Do políčka slovníku pro japonský výraz lze psát latinkou. Po vyhledání požadovaného výrazu se zobrazí dlaždice s výsledky (Obrázek 4.5). Kliknutím pravého tlačítka na frázi (popř. čtení) vyvoláte kontextové menu s nabídkou zobrazení informací o znacích, vložení do editoru, případně vyhledání ve slovníku.



Obrázek B.1: Okno, které se zobrazí po spuštění aplikace

B.3.2 Kandži vyhledávání

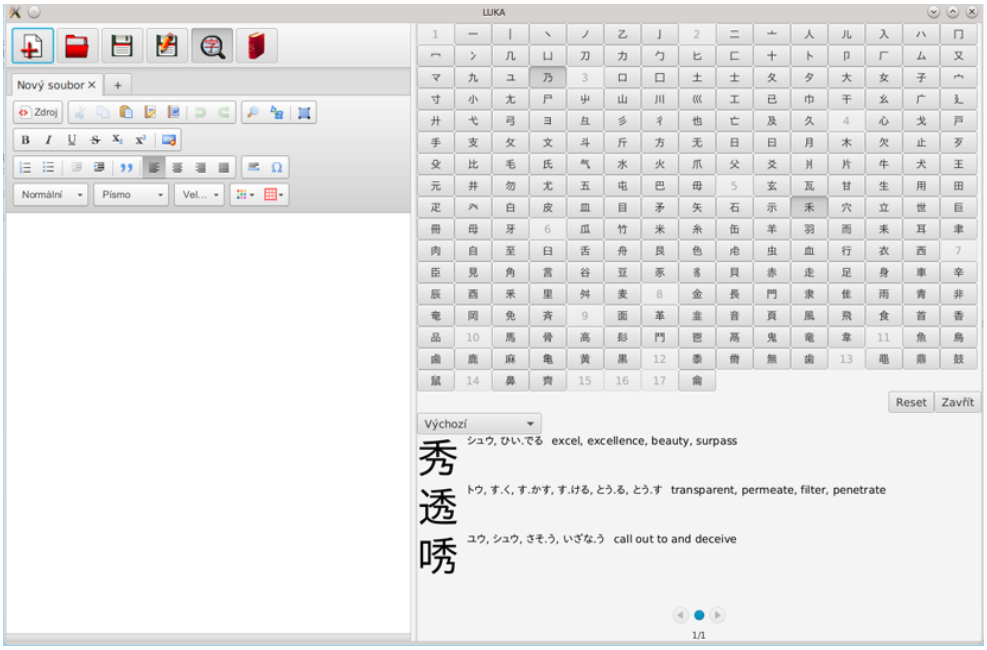
Vyhledání podle daného parametru se provede nejprve zaškrtnutím a zaktivněním vstupních polí. Při vyhledávání pomocí čtení anebo významu lze používat zástupné znaky: * pro libovolný počet znaků, ? žádný nebo jeden znak (ve čtení se jako jeden znak bere slabika kany či kandži). Příklad: `*rain*` vyhledá heavy rain, rainfall apod. a `?rain` vyhledá brain, train, grain, drain apod. Do pole se čtením lze psát latinkou.

Vyhledání dle radikálů zachovává předchozí nastavení ostatních parametrů (kvůli možnosti kombinaci všech kritérií). Spustíte jej kliknutím na tlačítko Změnit, zobrazí se tabulka radikálů, viz Obrázek B.2. V tabulce lze klikat a vyhledávání probíhá v reálném čase.

Výsledky vyhledávání lze řadit podle počtu tahů, frekvence používání, případně také zobrazit znaky ze zkoušek JLPT na předních příčkách.

Kliknutí pravého tlačítka na znak, či jeho čtení, vyvolá stejné kontextové menu jako ve slovníku.

Pokud ze slovníku nebo z výsledků kandži vyhledávání vyvoláte informaci o daném znaku, zobrazí se infopanel v dolní části kandži vyhledávání. Zde jsou o znaku všechny dostupné informace zahrnující posloupnost tahů znaku, počet



Obrázek B.2: Vyhledávání podle radikálů

tahů, překlad, čtení, příslušnost k nějaké úrovni JLPT a další, viz Obrázek 5.4.

B.3.3 Kandži Laso

Spuštění Lasa probíhá z rozhraní Kandži vyhledávání. Jakmile Laso spustíte, primární obrazovku pokryje „sklo“ s nápovědou (Obrázek B.3.3). Znak, který chcete rozpoznat, musíte mít pod sklem. Poté stačí myší táhnout (sklo zmizí) a zachytit znak do obdélníku, který se zobrazuje s červeným rámečkem během tažení. Jakmile pustíte tlačítko myši, znak je z obrazvky vyfocen a rozpoznán. Výsledky rozpoznání se zobrazují na stejném místě jako výsledky vyhledávání. Důležité je označit jen požadovaný znak a ne například i furiganu či části ostatních znaků.

Pokud chcete manipulovat s dokumenty pod sklem, musíte jej vypnout tlačítkem v levém horním rohu obrazovky. Aktivaci skla provede stejným tlačítkem a můžete pokračovat v rozpoznávání.

Obrázek B.3: Průběh rozpoznávání Kandži lasem

Příloha C

Obsah přiloženého CD

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